ARCHITETTURA DEGLI ELABORATORI

A.A. 2020-2021

Università di Napoli Federico II
Corso di Laurea in Informatica

Docenti
Proff. Luigi Sauro gruppo 1 (A-G)
Silvia Rossi gruppo 2 (H-Z)

ARCHITETTURA ARM

for Loops

```
for (initialization; condition; loop operation)
   statement
```

- **`initialization:`** eseguita prima che il loop inizi
- **`condition:`** condizione di continuazione che è verificata all'inizio di ogni iterazione
- **`loop operation:`** eseguita alla fine di ogni iterazione
- **`statement:`** eseguito ad ogni iterazione, ovvero fintantoché la condizione di continuazione è verificata

for Loops

C Code

```
// adds numbers from 1-9
int sum = 0


for (i=1; i!=10; i=i+1)
   sum = sum + i;
```

ARM Assembly Code

for Loops

C Code

```
// adds numbers from 1-9
int sum = 0


for (i=1; i!=10; i=i+1)
   sum = sum + i;
```

ARM Assembly Code

```
; R0 = i, R1 = sum
  MOV   R0, #1          ; i = 1
  MOV   R1, #0          ; sum = 0

FOR
  CMP R0, #10           ; R0-10
  BEQ DONE              ; if (i==10)
                        ; exit loop
  ADD R1, R1, R0        ; sum=sum + i
  ADD R0, R0, #1        ; i = i + 1
  B   FOR               ; repeat loop

DONE
```

for Loops: Decremented Loops

In ARM, i loop decrescenti fino a 0 sono più efficienti

C Code

```
// adds numbers from 1-9
int sum = 0


for (i=9; i!=0; i=i-1)
   sum = sum + i;
```

ARM Assembly Code

```
; R0 = i, R1 = sum
  MOV   R0, #9          ; i = 9
  MOV   R1, #0          ; sum = 0

FOR
  ADD   R1, R1, R0      ; sum=sum + i
  SUBS  R0, R0, #1      ; i = i - 1
                        ; and set flags
  BNE   FOR             ; if (i!=0)
                        ; repeat loop
```

Si risparmiano 2 istruzioni per ogni iterazione:

- Si accorpano decremento e comparazione: `SUBS R0, R0, #1`
- Solo un branch invece di due

Programming Building Blocks

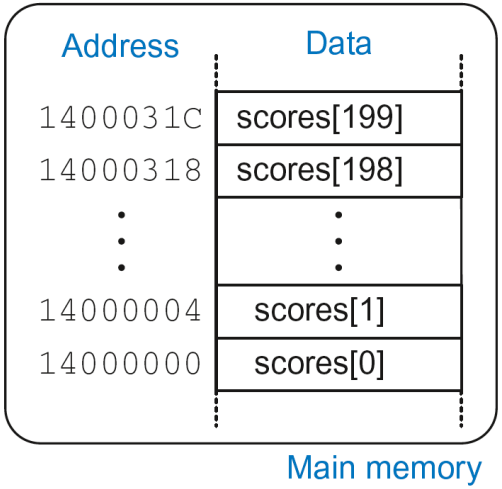
- **Data-processing Instructions**
- **Conditional Execution**
- **Branches**
- **High-level Constructs:**
  - if/else statements
  - for loops
  - while loops
  - **arrays**
  - function calls

Arrays

- Access large amounts of similar data
  - **Index:** access to each element
  - **Size:** number of elements

Arrays

- Consente l'accesso ad una quantità di dati simili
  - **Index:** accesso ad un qualche elemento
  - **Size:** numero di elementi
- Esempio: array di parole
  - **Base address** = 0x14000000 (indirizzo di scores[0])
  - Gli altri elementi sono raggiungibili a partire dal base address



Accessing Arrays

C Code

```
int array[5];
array[0] = array[0] * 8;
array[1] = array[1] * 8;
```

ARM Assembly Code

```
; R0 = array base address
```

Accessing Arrays

C Code

```
int array[5];
array[0] = array[0] * 8;
array[1] = array[1] * 8;
```

ARM Assembly Code

```
; R0 = array base address
  MOV R0, #0x60000000          ; R0 = 0x60000000

  LDR R1, [R0]                 ; R1 = array[0]
  LSL R1, R1, 3                ; R1 = R1 << 3 = R1*8
  STR R1, [R0]                 ; array[0] = R1

  LDR R1, [R0, #4]             ; R1 = array[1]
  LSL R1, R1, 3                ; R1 = R1 << 3 = R1*8
  STR R1, [R0, #4]             ; array[1] = R1
```

Arrays using for Loops

C Code

```
int array[200];
int i;

for (i=199; i >= 0; i = i - 1)
     array[i] = array[i] * 8;
```

ARM Assembly Code

```
; R0 = array base address, R1 = i
```

Arrays using for Loops

C Code

```
int array[200];
int i;

for (i=199; i >= 0; i = i - 1)
     array[i] = array[i] * 8;
```

ARM Assembly Code

```
; R0 = array base address, R1 = i
  MOV R0, 0x60000000
  MOV R1, #199

FOR
  LDR  R2, [R0, R1, LSL #2]      ; R2 = array(i)
  LSL  R2, R2, #3                ; R2 = R2<<3 = R2*8
  STR  R2, [R0, R1, LSL #2]      ; array(i) = R2
  SUBS R1, R1, #1                ; i = i - 1
                                 ; and set flags
  BPL  FOR                       ; if (i>=0) repeat loop
```

ASCII Code

- American Standard Code for Information Interchange
- Each text character has unique byte value
  - For example, S = 0x53, a = 0x61, A = 0x41
  - Lower-case and upper-case differ by 0x20 (32)

Cast of Characters

#	Char	#	Char	#	Char	#	Char	#	Char	#	Char
20	space	30	0	40	@	50	P	60	`	70	p
21	!	31	1	41	A	51	Q	61	a	71	q
22	"	32	2	42	B	52	R	62	b	72	r
23	#	33	3	43	C	53	S	63	c	73	s
24	$	34	4	44	D	54	T	64	d	74	t
25	%	35	5	45	E	55	U	65	e	75	u
26	&	36	6	46	F	56	V	66	f	76	v
27	‘	37	7	47	G	57	W	67	g	77	w
28	(	38	8	48	H	58	X	68	h	78	x
29	)	39	9	49	I	59	Y	69	i	79	y
2A	*	3A	:	4A	J	5A	Z	6A	j	7A	z
2B	+	3B	;	4B	K	5B	[	6B	k	7B	{
2C	,	3C	<	4C	L	5C	\	6C	l	7C	|
2D	–	3D	=	4D	M	5D	]	6D	m	7D	}
2E	.	3E	>	4E	N	5E	^	6E	n	7E	~
2F	/	3F	?	4F	O	5F	_	6F	o		

Programming Building Blocks

- **Data-processing Instructions**
- **Conditional Execution**
- **Branches**
- **High-level Constructs:**
  - if/else statements
  - for loops
  - while loops
  - arrays
  - **function calls**

Chiamate di funzioni

- **Caller:** funzione chiamante, in questo caso `main`
- **Callee:** funzione chiamata, in questo caso `sum`

C Code

```
void main()
{
  int y;
  y = sum(42, 7);
  ...
}

int sum(int a, int b)
{
  return (a + b);
}
```

Function Conventions

- **Caller:**
  - passes **arguments** to callee
  - jumps to callee

Contratto

- **Caller:**
  - Passa gli argomenti al callee
  - Esegue un jump al callee
- **Callee:**
  - Esegue la funzione chiamata
  - ritorna il risultato al caller
  - ritorna nel punto di chiamata del caller
  - non deve sovrascrivere i registri e la memoria usata dal caller

Convenzioni ARM per chiamata a funzione

- **Chiamata a funzione:** branch and link `BL`
- **Return** da funzione: ripristina in PC il valore del link register `MOV PC, LR`
- **Argomenti:** `R0-R3`
- **Valore di ritorno:** `R0`

Esempio di chiamata a funzione

C Code

```
int main() {
  simple();
  a = b + c;
}

void simple() {
  return;
}
```

ARM Assembly Code

```
0x00000200 MAIN     BL  SIMPLE
0x00000204          ADD R4, R5, R6
...

0x00401020 SIMPLE   MOV PC, LR
```

`void` significa che `simple` non ritorna un valore

Function Calls

C Code

```
int main() {
  simple();
  a = b + c;
}

void simple() {
  return;
}
```

ARM Assembly Code

```
0x00000200 MAIN     BL  SIMPLE
0x00000204          ADD R4, R5, R6
...

0x00401020 SIMPLE   MOV PC, LR
```

BL branches to SIMPLE
LR = PC + 4 = 0x00000204

MOV PC, LR makes PC = LR
(the next instruction executed is at 0x00000204)

Input Arguments and Return Value

ARM conventions:

- Argument values: `R0` - `R3`
- Return value: `R0`

Argomenti e valore di ritorno

Convenzioni ARM:

- Argomenti: `R0` - `R3`
- Valore di ritorno: `R0`

C Code

```
int main(){
  int y;
  ...
  y = diffofsums(2, 3, 4, 5);  // 4 arguments
  ...
}

int diffofsums(int f, int g, int h, int i){
  int result;
  result = (f + g) - (h + i);
  return result;                // return value
}
```

Input Arguments and Return Value

ARM Assembly Code

```
; R4 = y
MAIN
   ...
   MOV R0, #2          ; argument 0 = 2
   MOV R1, #3          ; argument 1 = 3
   MOV R2, #4          ; argument 2 = 4
   MOV R3, #5          ; argument 3 = 5
   BL DIFFOFSUMS       ; call function
   MOV R4, R0          ; y = returned value
   ...
; R4 = result
DIFFOFSUMS
   ADD R8, R0, R1      ; R8 = f + g
   ADD R9, R2, R3      ; R9 = h + i
   SUB R4, R8, R9      ; result = (f + g) - (h + i)
   MOV R0, R4          ; put return value in R0
   MOV PC, LR          ; return to caller
```

Input Arguments and Return Value

ARM Assembly Code

```
; R4 = result
DIFFOFSUMS
  ADD R8, R0, R1      ; R8 = f + g
  ADD R9, R2, R3      ; R9 = h + i
  SUB R4, R8, R9      ; result = (f + g) - (h + i)
  MOV R0, R4          ; put return value in R0
  MOV PC, LR          ; return to caller
```

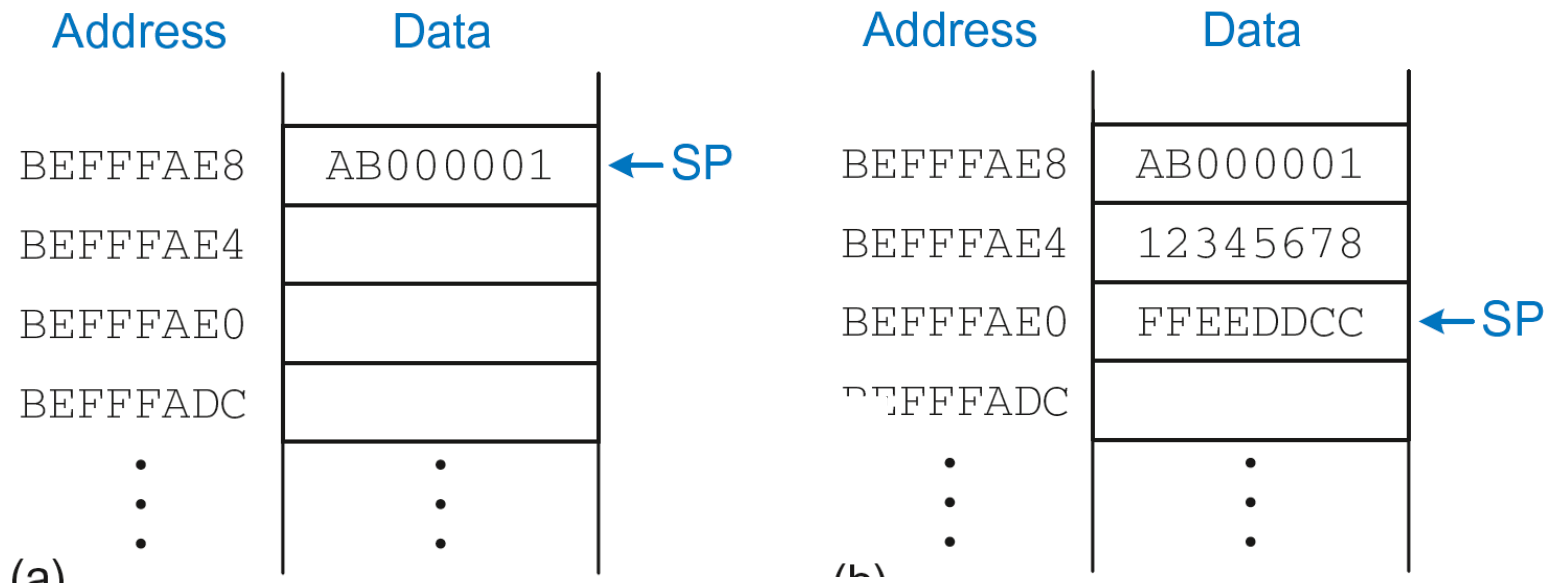
- `diffofsums` sovrascrive indebitamente i 3 registri `R4`, `R8`, `R9` dedicati alle saved variables della funzione chiamante `main`
- `diffofsums` invece dovrebbe usare lo *stack* per memorizzare temporaneamente i valori di questi registri prima di operare su di essi

Lo stack delle funzioni

- Lo stack costituisce la memoria usata per salvare temporaneamente il valore delle variabili
- E' organizzato come una pila last-in-first-out (LIFO)
- Operazioni sullo stack
  - ***Expands*:** usa più memoria quando necessaria (push)
  - ***Contracts*:** usa meno memoria quando (pop)

Lo stack delle funzioni

- In termini di indirizzi di memoria, l'espansione dello stack avviene in senso decrescente (dagli indirizzi maggiori a quelli più piccoli)
- Stack pointer: SP *punta* al top dello stack



Stack expands by 2 words

Come le funzioni usano lo stack

- Le funzioni chiamate non devono avere side effect inattesi
- Invece, nell esempio precedente `diffofsums` sovrascrive indebitamente i 3 registri `R4`, `R8`, `R9`

ARM Assembly Code

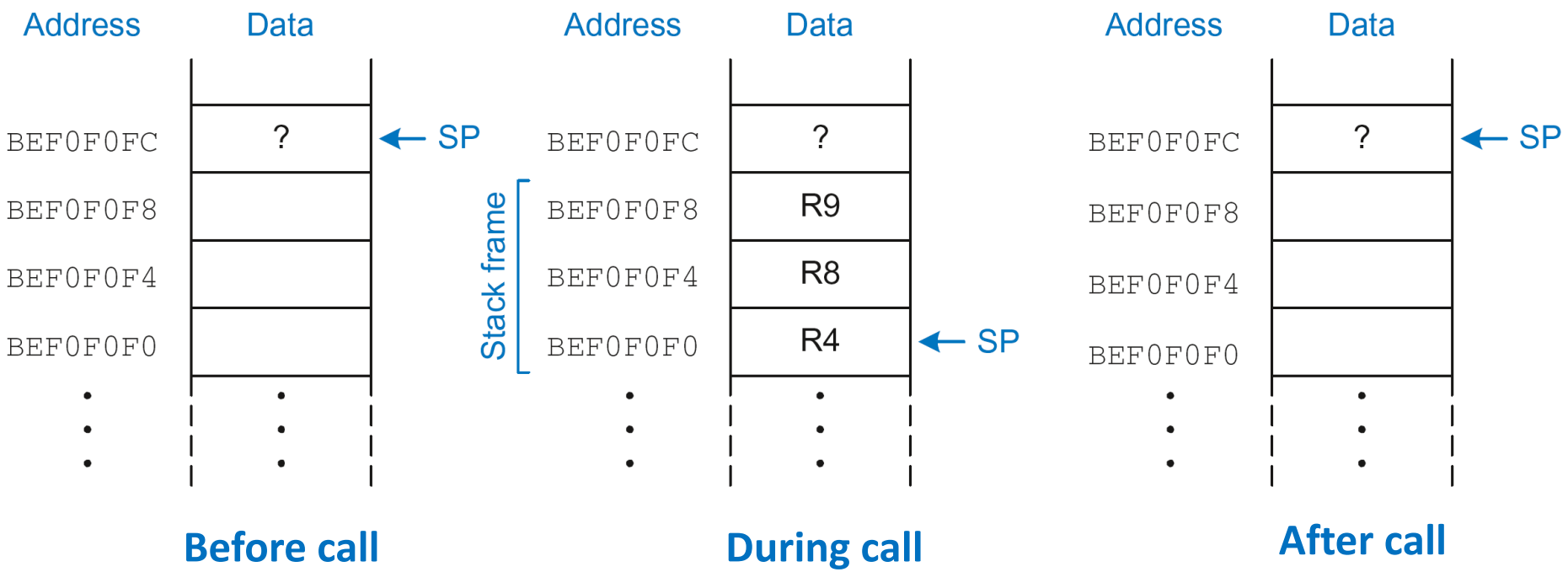
```
; R4 = result
DIFFOFSUMS
  ADD R8, R0, R1      ; R8 = f + g
  ADD R9, R2, R3      ; R9 = h + i
  SUB R4, R8, R9      ; result = (f + g) - (h + i)
  MOV R0, R4          ; put return value in R0
  MOV PC, LR          ; return to caller
```

Memorizzare i registri sullo stack

ARM Assembly Code

```
; R2 = result
DIFFOFSUMS
  SUB SP, SP, #12     ; make space on stack for 3 registers
  STR R4, [SP, #-8]   ; save R4 on stack
  STR R8, [SP, #-4]   ; save R8 on stack
  STR R9, [SP]        ; save R9 on stack
  ADD R8, R0, R1      ; R8 = f + g
  ADD R9, R2, R3      ; R9 = h + i
  SUB R4, R8, R9      ; result = (f + g) - (h + i)
  MOV R0, R4          ; put return value in R0
  LDR R9, [SP]        ; restore R9 from stack
  LDR R8, [SP, #-4]   ; restore R8 from stack
  LDR R4, [SP, #-8]   ; restore R4 from stack
  ADD SP, SP, #12     ; deallocate stack space
  MOV PC, LR          ; return to caller
```

Memorizzare i registri sullo stack



Registers

Preserved *Callee-Saved*	**Nonpreserved** *Caller-Saved*
R4-R11	**R12**
R14 (LR)	**R0-R3**
R13 (SP)	**CPSR**
stack above SP	**stack below SP**

Pop e push

- Salvare e ripristinare registri dallo stack è una operazione così frequente che ARM mette a disposizione delle istruzioni specifiche, Pop e Push, che consentono di avere un codice più succinto.
- Push consente di salvare in memoria più registri aggiornare consistentemente lo stack: `PUSH {R4, R8, R9}`
- Pop ripristina uno o più registri e incrementa consistentemente il valore dello stack: `POP {R4, R8, R9}`

Storing Saved Registers only on Stack

ARM Assembly Code

```
; R2 = result
DIFFOFSUMS
  STR R4, [SP, #-4]! ; save R4 on stack
  ADD R8, R0, R1     ; R8 = f + g
  ADD R9, R2, R3     ; R9 = h + i
  SUB R4, R8, R9     ; result = (f + g) - (h + i)
  MOV R0, R4         ; put return value in R0
  LDR R4, [SP], #4   ; restore R4 from stack
  MOV PC, LR         ; return to caller
```

Storing Saved Registers only on Stack

ARM Assembly Code

```
; R2 = result
DIFFOFSUMS
  STR R4, [SP, #-4]! ; save R4 on stack
  ADD R8, R0, R1     ; R8 = f + g
  ADD R9, R2, R3     ; R9 = h + i
  SUB R4, R8, R9     ; result = (f + g) - (h + i)
  MOV R0, R4         ; put return value in R0
  LDR R4, [SP], #4   ; restore R4 from stack
  MOV PC, LR         ; return to caller
```

Notice code optimization for expanding/contracting stack

Nonleaf Function

ARM Assembly Code

```
STR LR, [SP, #-4]!      ; store LR on stack
BL  PROC2               ; call another function
...
LDR LR, [SP], #4        ; restore LR from stack
jr  $ra                 ; return to caller
```

Nonleaf Function Example

C Code

```
int f1(int a, int b) {
  int i, x;
  x = (a + b)*(a - b);
  for (i=0; i<a; i++)
    x = x + f2(b+i);
  return x;
}
int f2(int p) {
  int r;
  r = p + 5;
  return r + p;
}
```

Nonleaf Function Example

C Code

```
int f1(int a, int b) {
  int i, x;
  x = (a + b)*(a - b);
  for (i=0; i<a; i++)
    x = x + f2(b+i);
  return x;
}
int f2(int p) {
  int r;
  r = p + 5;
  return r + p;
}
```

ARM Assembly Code

```
; R0=a, R1=b, R4=i, R5=x
F1
  PUSH {R4,  R5, LR}
  ADD   R5,  R0, R1
  SUB   R12, R0, R1
  MUL   R5,  R5, R12
  MOV   R4,  #0
FOR
  CMP   R4, R0
  BGE   RETURN
  PUSH {R0, R1}
  ADD   R0, R1, R4
  BL    F2
  ADD   R5, R5, R0
  POP  {R0, R1}
  ADD   R4, R4, #1
  B     FOR
RETURN
  MOV   R0, R5
  POP  {R4, R5, LR}
  MOV   PC, LR
```

```
; R0=p, R4=r
F2
  PUSH {R4}
  ADD   R4, R0, 5
  ADD   R0, R4, R0
  POP  {R4}
  MOV   PC, LR
```