

# Progetto e analisi di programmi per sistemi dedicati

## Lezione 07 di Sistemi dedicati

Docente: Giuseppe Scollo

Università di Catania  
Dipartimento di Matematica e Informatica  
Corso di Laurea Magistrale in Informatica, AA 2017-18

### Indice

1. Progetto e analisi di programmi per sistemi dedicati
2. argomenti della lezione
3. microprocessori, toolchain
4. da C ad assembly (ARM): un esempio
5. analisi del codice oggetto
6. rappresentazione dei tipi di dati
7. variabili nella gerarchia di memoria
8. chiamate di funzione: un esempio
9. costruzione dell'area di attivazione
10. disposizione in memoria del programma
11. riferimenti

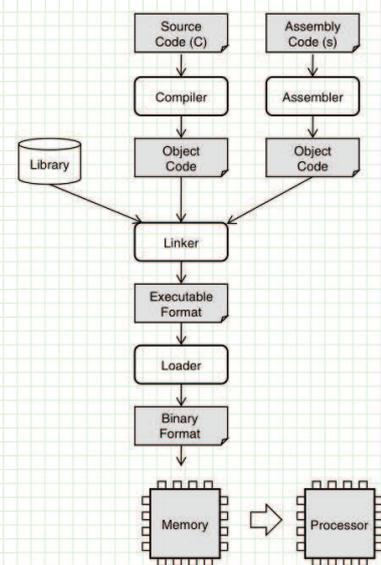
di che si tratta:

- motivazioni per l'uso di microprocessori dedicati
- toolchain per lo sviluppo di programmi
- strumenti per l'analisi di codice oggetto
- rappresentazione dei tipi di dati
- variabili nella gerarchia di memoria
- compilazione di chiamate di funzione
- disposizione in memoria di programmi eseguibili
- esempi in C e assembly ARM

## microprocessori, toolchain

il microprocessore è il componente programmabile di maggior successo negli ultimi decenni... perché?

- separazione del software dall'hardware con la definizione di un insieme di istruzioni
- ampia disponibilità di software di supporto allo sviluppo di programmi, anche in linguaggi di alto livello
- alta efficienza delle opzioni di riuso di componenti e di interoperabilità con altri componenti, sia hardware (bus standard) che software (librerie)
- alta scalabilità, e.g. lunghezza di parola da 4 a 64 bit, uso di un microprocessore con funzione di coordinamento in una complessa architettura SoC ecc.



Schaumont, Figure 7.1 - Standard design flow of software source code to processor instruction

## da C ad assembly (ARM): un esempio

```
int gcd(int a[5], int b[5]) {
    int i, m, n, max;
    max = 0;
    for (i=0; i<5; i++) {
        m = a[i];
        n = b[i];
        while (m != n) {
            if (m > n) m = m - n;
            else n = n - m;
        }
        if (max < m) max = m;
    }
    return max;
}
int a[] = {26, 3, 33, 56, 11};
int b[] = {87, 12, 23, 45, 17};
int main() {
    return gcd(a, b);
}
```

Schaumont, Listing 7.1 - A C program to find a maximum GCD

Diagram illustrating the mapping between C code and assembly instructions. Labels like 'main:' and '.L19:' are linked to specific instructions. Mnemonics like 'str', 'ldr', 'cmp', 'beq', 'rsbgt', 'rsble', 'cmplt', 'movgt', 'ldrgt', 'b', and '.align' are also shown.

Schaumont, Figure 7.2 - Elements of an assembly program produced by gcc

```
gcd:
    str    lr, [sp, #-4]!
    mov    lr, #0
    mov    ip, lr
.L13:
    ldr    r3, [r0, ip, asl #2]
    ldr    r2, [r1, ip, asl #2]
    cmp    r3, r2
    beq    .L17
.L11:
    cmp    r3, r2
    rsbgt r3, r2, r3
    rsble r2, r3, r2
    cmplt r3, r2
    bne    .L11
.L17:
    add    ip, ip, #1
    cmp    lr, r3
    movlt  lr, r3
    cmp    ip, #4
    movgt  r0, lr
    ldrgt  pc, [sp], #4
    b     .L13
a:
    .word  26, 3, 33, 56, 11
b:
    .word  87, 12, 23, 45, 17
main:
    str    lr, [sp, #-4]!
    ldr    r0, .L19
    ldr    r1, .L19+4
    ldr    lr, [sp], #4
    b     gcd
.L19:
    .align 2
    .word  a
    .word  b
```

Schaumont, Listing 7.2 - ARM assembly dump of Listing 7.1

## analisi del codice oggetto

l'esempio appena visto è sviluppato con il cross-compilatore GNU arm-linux-gcc, reperibile come pacchetto Debian nel repository di Gezel:

[rijndael.ece.vt.edu/gezel2repo/pool/main/a/arm-linux-gcc](http://rijndael.ece.vt.edu/gezel2repo/pool/main/a/arm-linux-gcc)

il codice assembly simbolico è ottenuto dal sorgente C mediante il comando:

```
/usr/local/arm/bin/arm-linux-gcc -c -S -O2 gcd.c -o gcd.s
```

il comando per generare l'eseguibile ARM ELF è:

```
/usr/local/arm/bin/arm-linux-gcc -O2 gcd.c -o gcd
```

è anche possibile ottenere il codice simbolico dall'eseguibile ELF mediante un *disassemblatore*, in questo esempio con il seguente comando:

```
/usr/local/arm/bin/arm-linux-objdump -d gcd
```

l'output del disassemblatore mostra anche il codice binario associato a ogni istruzione simbolica e gli indirizzi associati alle etichette

l'uso di questo strumento e di altri strumenti di utilità associati ai compilatori per l'analisi di codice eseguibile sarà ulteriormente esplorato in esercitazioni di laboratorio

## rappresentazione dei tipi di dati

il codesign hardware/software efficiente richiede una comprensione congiunta di architettura di sistema e del software

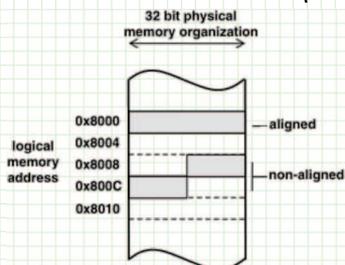
la rappresentazione dei tipi di dati è un buon punto di partenza, i compilatori ne conoscono le differenze di:

- dimensione di memoria
- implementazione di basso livello delle operazioni

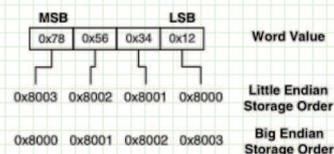
la tabella 7.1 mostra come C li traduce ai tipi di dati nativi supportati da processori a 32 bit

C data type	
char	8-bit
short	signed 16-bit
int	signed 32-bit
long	signed 32-bit
long long	signed 64-bit

Schaumont, Table 7.1 - Compiler data types



Schaumont, Figure 7.7 (a) - Alignment of data types



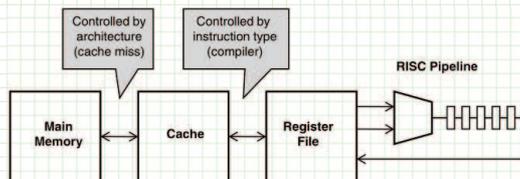
Schaumont, Figure 7.7 (b) - Little-endian and Big-endian storage order

un'organizzazione della memoria basata sulla parola richiede allineamento ai confini di parola, per eseguire un trasferimento di parola con un solo accesso a memoria il compilatore genera direttive a tal fine

l'ordinamento dei byte, in qualche caso persino quello dei bit, è rilevante al codesign hardware/software nella transizione dal software all'hardware e viceversa

## variabili nella gerarchia di memoria

un altro aspetto importante della rappresentazione dei dati è il tipo di memoria fisica allocata



Schaumont, Figure 7.8 - Memory hierarchy

la gerarchia di memoria è trasparente a programmi di alto livello, e.g. in C, ma il controllo di basso livello influisce sulle prestazioni; ecco un esempio:

```
void accumulate(int *c, int a[10]) {
    int i;
    *c = 0;
    for (i=0; i<10; i++) *c += a[i];
}
```

/usr/local/arm/bin/arm-linux-gcc -O2 -c -S accumulate.c

genera il codice seguente in accumulate.s :

```

mov    r3, #0
str    r3, [r0, #0]
mov    ip, r3
.L6:
ldr    r2, [r1, ip, asl #2]    ; r2 = a[i]
ldr    r3, [r0, #0]          ; r3 = *c (memory)
add    ip, ip, #1            ; increment loop ctr
add    r3, r3, r2
cmp    ip, #9
str    r3, [r0, #0]          ; r3 → *c (memory)
movgt pc, lr
b      .L6
```

nell'esempio, il valore della variabile accumulatore viaggia su e giù nella gerarchia di memoria un controllo limitato è possibile in C con l'uso di specificatori di classe di memoria e qualificatori di tipo

Storage specifier	Type qualifier
register	const
static	volatile
extern	

## chiamate di funzione: un esempio

le chiamate di funzioni sono la struttura fondamentale della gerarchia comportamentale dei programmi: ecco un esempio della loro traduzione in linguaggio macchina

```
int accumulate(int a[10]) {
    int i;
    int c = 0;
    for (i=0; i<10; i++)
        c += a[i];
    return c;
}
int a[10];
int one = 1;
int main() {
    return one + accumulate(a);
}
```

Schaumont, Listing 7.4 - Sample program

la compilazione del programma senza ottimizzazione mostra la creazione, nella pila, dell'area di attivazione, che viene dinamicamente associata all'esecuzione della funzione per ospitare variabili locali e salvataggio di registri

in questo caso il registro r0 è usato per il passaggio del parametro e del risultato; quando i parametri sono molti sono passati nell'area di attivazione

l'uso del registro FP (*frame pointer*) permette la nidificazione delle chiamate e la ricorsione

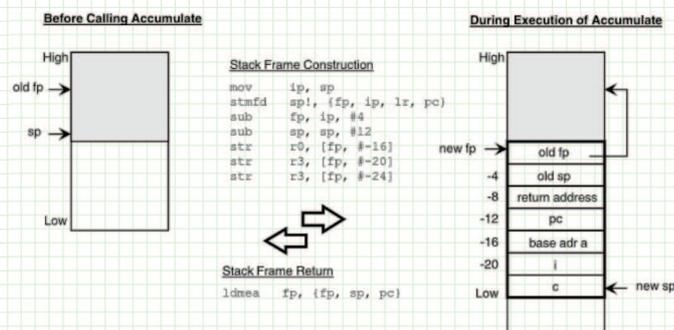
```
accumulate:
    mov     ip, sp
    stmfd  sp!, {fp, ip, lr, pc}
    sub    fp, ip, #4
    sub    sp, sp, #12
    str    r0, [fp, #-16]           ; base address a
    mov    r3, #0
    str    r3, [fp, #-24]         ; c
    mov    r3, #0
    str    r3, [fp, #-20]         ; i
.L2:
    ldr    r3, [fp, #-20]
    cmp    r3, #9
    ble    .L5
    b      .L3
.L5:
    ldr    r3, [fp, #-20]           ; i * 4
    mov    r2, r3, asl #2
    ldr    r3, [fp, #-16]
    add    r3, r2, r3
    ldr    r2, [fp, #-24]
    ldr    r3, [r3, #0]
    add    r3, r2, r3
    str    r3, [fp, #-24]         ; c = c + a[i]
    ldr    r3, [fp, #-20]
    add    r3, r3, #1
    str    r3, [fp, #-20]         ; i = i + 1
    b      .L2
.L3:
    ldr    r3, [fp, #-24]         ; return arg
    mov    r0, r3
    ldmea fp, {fp, sp, pc}
```

Schaumont, Listing 7.6 - Accumulate without compiler optimizations

## costruzione dell'area di attivazione

la figura 7.9 mostra la costruzione dell'area di attivazione nella pila

il registro SP punta alla cima occupata della pila, che cresce verso il basso: queste convenzioni si riflettono nel suffisso *fd* (*full, descending*) dell'istruzione di trasferimento multiplo *stmfd* per il salvataggio in pila di registri



Schaumont, Figure 7.9 - Stack frame construction

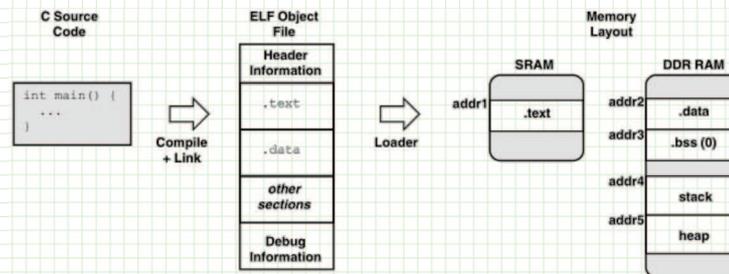
il ripristino dei registri salvati e il rientro si hanno con un'unica istruzione di trasferimento multiplo

in questo caso è appropriato il suffisso *converso* *ea* (*empty, ascending*), notando che è FP, non SP, il registro di base per l'indirizzo da cui iniziare il trasferimento

## disposizione in memoria del programma

per la rappresentazione fisica del programma e delle sue strutture dati nella gerarchia di memoria, occorre distinguere fra:

- *disposizione statica del programma*: organizzazione dell'output di compilatore e linker in un file ELF (o in una ROM)
- *disposizione dinamica del programma*: organizzazione in memoria di un programma eseguibile durante l'esecuzione



Schaumont, Figure 7.10 - Static and dynamic program layout

- il caricatore può assegnare sezioni differenti del programma ELF a differenti tipi di memoria
- il layout dinamico può avere sezioni assenti nel file ELF, per dati dinamici (stack, heap ecc.)

## riferimenti

letture raccomandate:

Schaumont, Ch. 7, Sect. 7.1, 7.3

per sperimentazione:

installazione del cross-compilatore arm-linux-gcc

per ulteriore consultazione:

Schaumont Ch. 7, Sect. 7.2

Introduction to the ARM® Processor Using Intel FPGA Toolchain - For Quartus

Prime 16.1, Intel Corp. - FPGA University Program, November 2016