



**UNIVERSITÀ  
DEGLI STUDI  
DI TRIESTE**



Dipartimento di  
**Ingegneria  
e Architettura**

# Introduzione alla programmazione assembly in LEGv8

Rev 1.3.1

Francesco Vodisca

Tratto dal corso di  
Architetture dei Sistemi Digitali  
tenuto dal prof. Alberto Carini

Dipartimento di Ingegneria e Architettura  
Università degli Studi di Trieste

A.A. 2021/2022

# Indice

<b>Indice</b>	<b>1</b>
Codici . . . . .	2
Soluzioni codici . . . . .	3
Registro delle modifiche . . . . .	3
<b>1 Introduzione</b>	<b>4</b>
<b>2 Le basi</b>	<b>4</b>
<b>3 Intermezzo 1: verifichiamo i codici scritti</b>	<b>7</b>
<b>4 Operiamo sugli array</b>	<b>9</b>
<b>5 Intermezzo 2: altri tipi di dati</b>	<b>13</b>
5.1 IEEE 754 . . . . .	14
5.2 Approfondimento sul trasferimento bit a bit . . . . .	17
<b>6 Programmazione funzionale</b>	<b>19</b>
<b>7 Intermezzo 3: i salti</b>	<b>23</b>
7.1 Operazioni condizionali . . . . .	24
<b>8 Ricorsione</b>	<b>25</b>
<b>9 Approfondimenti</b>	<b>29</b>
9.1 ALU . . . . .	29
9.2 Lock . . . . .	31
9.3 Approccio alla compilazione su carta . . . . .	33
9.4 Dove continuare . . . . .	39
<b>10 ARMv8 su RaspberryPi</b>	<b>40</b>
10.1 Introduzione all'architettura . . . . .	42
10.2 Alcuni codici in ARMv8 . . . . .	44
10.3 Ottimizziamo i codici: Branchless Programming, Loop Unrolling e SIMD . . . . .	49
10.4 GDB per Linux . . . . .	54
10.5 Comunicare con il RP dal proprio PC . . . . .	56
<b>11 Ulteriori esercizi</b>	<b>57</b>
<b>12 Soluzione esercizi</b>	<b>59</b>

# Codici

2.1: prima somma . . . . .	5
2.2.1: Median3 . . . . .	7
5.1: Stringa maiuscola . . . . .	14
5.2: Fast Inverse Square Root . . . . .	18
5.3: Media aritmetica . . . . .	19
6.1: bubbleSort . . . . .	21
6.1: Fibonacci iterativo . . . . .	23
6.2: Fibonacci iterativo con memorizzazione di tutti i valori calcolati . . . . .	23
8.1: Fattoriale iterativo . . . . .	25
8.2: Fattoriale ricorsivo . . . . .	26
8.3: Fibonacci ricorsivo . . . . .	27
8.4: Fibonacci con memoria . . . . .	29
9.3: Chiamata a FIR1D . . . . .	34
10.2.1: Stringa maiuscola senza branch . . . . .	50
10.2.2: Stringa maiuscola con loop unrolling . . . . .	52
11.1: Conversione di una stringa in un intero . . . . .	58
11.2: Convoluzione di un array generico di double con un kernel di dimensione non nota . . . . .	58
11.3: Algoritmo di ordinamento CountingSort . . . . .	59
11.5: Autocorrelazione . . . . .	59

# Soluzioni codici

2.1: prima somma . . . . .	5
5.1: Stringa maiuscola . . . . .	14
5.2: Fast Inverse Square Root . . . . .	18
6.1: bubbleSort . . . . .	22
8.1: Fattoriale iterativo (assembly) . . . . .	26
8.2: Fattoriale ricorsivo . . . . .	26
8.3: Fibonacci ricorsivo . . . . .	27
9.3.1: Chiamata a FIR1D . . . . .	38
10.2.1: Stringa maiuscola senza branch . . . . .	51
10.2.2: Stringa maiuscola con Loop Unrolling . . . . .	53
2.1.1: somma di tre numeri . . . . .	59
2.1.2: misto mare . . . . .	60
2.2.1: Median3 . . . . .	60
5.3: media aritmetica di un array di double . . . . .	61
6.1: Fibonacci iterativo . . . . .	61
8.4.1: Fibonacci iterativo con memoria . . . . .	62
8.4.2: Fibonacci ricorsivo con memoria . . . . .	63

9.3.2: Chiamata a FIR1D ottimizzato . . . . .	64
11.1: Conversione di una stringa in un intero . . . . .	65
11.2: Convoluzione di un array . . . . .	66
11.3: CountingSort . . . . .	67
11.5: Autocorrelazione . . . . .	68

## Registro delle modifiche

### Rev 1.1 24/04/2023

Prima pubblicazione.

### Rev 1.2 10/06/2023

Migliorata la numerazione dei codici;

Corretto i codici *Stringa maiuscola*, *Fibonacci ricorsivo*;

Corretto alcuni errori ortografici;

Aggiunto un ulteriore modo per verificare la correttezza dei codici;

Aggiunto ulteriori indicazioni sul funzionamento di ARMv8 su Raspberry Pi;

### Rev 1.3 15/09/2023

Aggiunti i codici *Chiamata a FIR1D* e *Autocorrelazione*;

Accennato a ulteriori metodi per programmare ARMv8/LEGv8 su Linux;

Aggiunto paragrafo sullo standard IEEE 754;

Approfondito paragrafo su Lock;

Aggiunto schema di ragionamento per la programmazione Assembly;

Aggiunta sottosezione sugli approfondimenti;

Aggiunta sottosezione sulle ottimizzazioni (Branchless Programming, Loop Unrolling e SIMD);

Aggiunta sottosezione sul GDB (GNU Debugger);

Aggiunta sottosezione sul come usare il RP via SSH e SFTP;

### Rev 1.3.1 08/11/2023

Correzione errori;

# 1 Introduzione

Ho voluto scrivere questa guida per annotare le domande e risolvere i dubbi che mi hanno posto alcuni miei compagni di corso mentre studiavano il corso di Architetture dei Sistemi Digitali. Per dare una continuità alle note ho scelto di racchiudere anche le altre parti del programma di Assembly, ponendo maggiore attenzione agli aspetti critici della programmazione LEGv8.

È fortemente consigliato avere delle buone basi di C per comprendere meglio alcuni concetti chiave della programmazione Assembly, quindi consiglio di leggere bene la parte di C svolta dal professore in aula. Nonostante ciò mi soffermerò su alcuni aspetti critici della programmazione C che influenzano anche la programmazione Assembly.

Questa guida seguirà un approccio strettamente pratico e mirato alla comprensione autonoma del linguaggio attraverso la pratica. Tutti i concetti della programmazione in LEGv8 del corso di Architetture dei Sistemi Digitali verranno affrontati, ma sono stati riorganizzati secondo quello che ho ritenuto essere un ordine di complessità crescente.

Tra gli strumenti consigliati per affrontare la lettura della guida c'è la Quick Reference Guide, reperibile sulla pagina Moodle del corso oppure inclusa nel libro di riferimento *Computer Organization and Design, ARM Edition*.

Spero possiate trovare questa guida utile e che vi fornisca le basi per affrontare con tranquillità la parte di Assembly dell'esame. Nel caso in cui voleste segnalare qualche correzione o approfondire qualche argomento ulteriormente non esitate a contattarmi. Ringrazio molto coloro che mi hanno segnalato alcuni errori nelle precedenti versioni e che mi hanno fornito utili pareri su come migliorare la guida.

## 2 Le basi

Per programmare in Assembly è essenziale conoscere l'architettura del calcolatore, quindi sapere quali sono i registri, come accedervi e conoscere le operazioni che possiamo effettuare con essi. I registri ci accompagneranno lungo tutta la guida, quindi è meglio introdurli fin da subito. Un registro è un tipo di memoria del processore. È situato all'interno del processore stesso, nei pressi dell'ALU (Arithmetic Logic Unit), il che consente di avere accesso istantaneo ai dati contenuti in esso.

Nel LEGv8 ci sono 32 registri per le operazioni intere e 32 per le operazioni con i numeri a virgola mobile. Esistono, inoltre, alcuni vincoli sui registri interi, in quanto vengono utilizzati per effettuare qualsiasi tipo di operazione (non solo numerica). La tabella sottostante elenca le funzionalità dei registri.

Nome registro/i	Numero	Uso	Il valore dev'essere ripristinato all'uscita?
X0 - X7	0 - 7	Input/Output	No
X8	8	Puntatore a un indirizzo di memoria per tornare un risultato troppo grande (non lo useremo mai)	No
X9 - X15	9 - 15	Temporanei	No
X16 - X18	16 - 18	Temporanei (meglio evitare di usarli)	No
X19 - X27	19 - 27	Registri preservati tra le chiamate	Si
SP	28	Stack Pointer	Si
FP	29	Frame Pointer	Si
LR	30	Link Register	Si
XZR	31	Registro sempre nullo, sia in lettura che in scrittura	/

Tabella 1: I 32 registri X del LEGv8 e la loro funzione

Per iniziare noi utilizzeremo solamente i primi registri e a mano a mano che ci serviranno gli altri farò riferimento a questa tabella.

## Sommiamo!

Creiamo il nostro primo programma: sommiamo due numeri e mettiamolo in un terzo. Il codice C di questo metodo sarebbe

```
1 long long somma(long long a, long long b){
2     return a+b;
3 }
```

### 2.1: prima somma

Scegliamo di ricevere gli input nei registri X0 e X1 e di mettere il risultato in X2<sup>1</sup>. Possiamo scrivere il seguente codice per LEGv8

```
1 ADD X2, X0, X1
```

### 2.1: prima somma

Si potrebbero fare notevoli miglioramenti a questo codice, ma li lascio alle sezioni future.

Ovviamente quanto appena visto per la somma può valere per qualsiasi altra operazione. Basta munirsi di QuickReferenceGuide e trovare il comando Assembly corretto. Per esercizio vi lascio sperimentare sia le altre operazioni, sia la somma con tre addendi, sia un misto mare di tutte

<sup>1</sup>Scelta poco ortodossa. Di solito i risultati si mettono nel registro X0 e se ce ne dovesse essere più di uno allora si riempiono gli altri registri X1->X7 (vedi tabella 1), ma per semplificare al momento possiamo sorpassare su questo dettaglio. Approfondiremo meglio nel capitolo 6

le operazioni (ad esempio  $\text{ris} = 3a + ((4b) \% 3)$ , dove  $\%$  sta per modulo). In fondo alla guida vi lascio la soluzione a [2.1.1: somma di tre numeri](#) e a [2.1.2: misto mare](#).

## E se...

Incominciamo ad aggiungere istruzioni per sfruttare tutte le potenzialità della nostra macchina di Turing: i salti!

O meglio, vediamo intanto un tipo di salto, il salto condizionale ed incondizionale. Nel LEGv8 esistono ulteriori due tipi di salti: i Branch to Label e i Branch to Register. Analizzeremo meglio questi due tipi di salti nelle prossime sezioni ([Programmazione funzionale](#) e [Intermezzo 3: i salti](#))

Supponiamo di voler verificare quale tra due numeri sia il maggiore. In C scriveremmo il seguente metodo

```
1 long long max(long long a, long long b){
2     if(a>b)
3         return a;
4     return b;
5 }
```

Come prima scegliamo di ricevere gli input nei registri X0 e X1, ma metteremo il risultato in X0. Un possibile codice Assembly per LEGv8 è

```
1     SUBS XZR, X0, X1 // a-b con attivazione dei flag
2     B.GT exit      // if(a>b) [return a e' implicito, in quanto a e' gia' in X0]
3     ADD X0, XZR, X1 // else return b
4 exit :
```

oppure, utilizzando le pseudoistruzioni:

```
1     CMP X0, X1     // a-b con attivazione dei flag
2     B.GT exit      // if(a>b) [return a e' implicito, in quanto a e' gia' in X0]
3     MOV X0, X1     // else return b
4 exit :
```

Il controllo  $\text{if}(a > b)$  viene svolto da due istruzioni in Assembly: SUBS(CMP) (l'operazione di sottrazione che abilita la scrittura dei flag) e il salto condizionale B.GT (branch if greater than). Nel caso in cui non si verifichi la condizione (quindi se  $b \geq a$ ) allora dobbiamo restituire il valore di b: abbiamo scelto di mettere il risultato in X0, ma b si trova in X1, quindi dobbiamo "spostare" il contenuto di X1 in X0 con l'istruzione ADD(MOV).

**SUBS** Similmente al classico SUB effettua l'operazione di sottrazione, ma in più attiva i 4 flag relativi all'operazione scrivendoli nell'apposito registro privato;<sup>2</sup>

**CMP** è una pseudoistruzione, ovvero un'istruzione che semplifica la sintassi di un'altra istruzione per uno scopo specifico. Nel nostro caso CMP è come una sottrazione dove il valore non dev'essere salvato. Vedi MOV per ulteriori dettagli sul registro XZR.

**B.condizione** I salti condizionali leggono alcuni<sup>3</sup> dei 4 flag precedentemente attivati per determinare se saltare o meno all'istruzione specificata da un'etichetta (*exit* nel nostro caso,

---

<sup>2</sup>È possibile approfondire il funzionamento di queste istruzioni nella sezione [Operazioni condizionali](#)

<sup>3</sup>Vedi tabella [Condizioni per i salti condizionali e loro verifica hardware](#) per sapere quali

presente in qualsiasi parte del codice, sia sopra, sia sotto l'istruzione corrente). Per approfondire vedi il capitolo sulle [Operazioni condizionali](#).

**MOV** Per capire il funzionamento della pseudoistruzione MOV bisogna conoscere i [I 32 registri X del LEGv8 e la loro funzione](#).

Interessante è il registro XZR, ovvero un registro a 64 bit che è sempre nullo, sia in lettura che in scrittura. Questo significa che se dovessimo preparare un numero (supponiamo 56) nel registro X9 potremmo usare un'istruzione del tipo

```
1 ADDI X9, XZR, #56
```

Se invece dovessimo spostare il contenuto del registro X1 nel registro X0 (com'era necessario nel programma appena visto) possiamo usare l'istruzione

```
1 ADD X0, X1, XZR
```

Per brevità è stata introdotta la pseudoistruzione MOV che fa lo stesso identico lavoro ma senza dover specificare il registro XZR.

Come esercizio vi propongo la mediana di tre valori. Questo è il codice C:

```
1 long long median3(long long a, long long b, long long c){
2     long long temp;
3     if(a>c){
4         temp=a;
5         a=c;
6         c=temp;
7     }
8     if(a>b){
9         temp=a;
10        a=b;
11        b=temp;
12    }
13    if(b>c){
14        temp=b;
15        b=c;
16        c=temp;
17    }
18    return b;
19 }
```

### 2.2.1: Median3

La soluzione è [Soluzione esercizi](#). Una possibile ottimizzazione consisterebbe nel creare una funzione esterna *swap* che viene chiamata ogni volta è necessario fare uno scambio tra due valori. Vedremo meglio quest'ottimizzazione nella sezione [Programmazione funzionale](#). Se volete un'anticipazione si trova nella sezione [Alcuni codici in ARMv8](#).

## 3 Intermezzo 1: verifichiamo i codici scritti

Quando scriviamo un codice C (o in qualsiasi altro linguaggio di alto livello) abbiamo la possibilità di compilarlo ed eseguirlo. Questo è possibile grazie al compilatore (che nel caso di C

può essere GCC) il quale genera un file eseguibile<sup>4</sup>. Un caso diverso è quello dei linguaggi interpretati (come Python, Matlab, ...), i quali interpretano, per l'appunto, il codice o i comandi inviati in fase di esecuzione.

Normalmente programiamo codici che verranno eseguiti sulla stessa macchina. Questo significa che il file eseguibile (.exe per Windows, .out per Unix) sono compilati per l'architettura x86 e vengono eseguiti su macchine basate su architettura x86. Ora, però, stiamo programmando LEGv8, una riduzione del set di istruzioni dell'ARMv8, ma stiamo codificando sempre su macchine x86<sup>5</sup> o su carta. Nella sezione 10 vedremo come si può programmare Assembly su un Raspberry Pi e nella sezione 9.3 vedremo alcune tecniche per la programmazione su carta, ma intanto vediamo come si può simulare un processore LEGv8 sui nostri PC.

Il simulatore del LEGv8 si può scaricare da [questa repository di GitHub](#). Una volta scaricata avviamo il simulatore lanciando il file `LEGv8_Simulator.html` contenuto nella sottocartella `war`. Si aprirà una pagina del vostro browser predefinito con tre sezioni principali: a destra c'è la logica interna ad una possibile implementazione dell'architettura LEGv8, in alto a sinistra c'è lo spazio per scrivere il codice, in basso a sinistra ci sono i valori attuali dei registri (e dei quattro flag).

Proviamo a verificare ed eseguire il primo codice Assembly che abbiamo incontrato: **Sommiamo!**. Il codice si aspetta di trovare nei registri X0 e X1 le variabili di input a e b rispettivamente, quindi per prima cosa mettiamo in questi due registri due numeri, poi scriviamo il codice.

Avremo quindi

```
1 ADDI X0, XZR, #4
2 ADDI X1, XZR, #7
3 ADD X2, X0, X1
```

Per assemblare il codice basta cliccare su *Assemble* e, una volta assemblato, per eseguirlo riga per riga bisogna cliccare su *Execute Instruction*. Quindi dopo aver premuto tre volte su *Execute Instruction* dovremmo vedere nel registro X2 il valore 0xb, che è la rappresentazione esadecimale di 11.

Provate a verificare i codici fino ad ora scritti. Ho scelto di proporvi adesso quest'intermezzo in quanto, ahimè, per dei problemi al simulatore non è possibile *eseguire* codici con istruzioni più complesse (se non tramite trucchetti complessi che richiedono la padronanza completa del linguaggio). Non è possibile allocare spazio nello stack in quanto non viene interpretata bene

---

<sup>4</sup>In realtà gli stadi sono 4 con quattro: compilatore, assemblatore, linker e loader. Il *compilatore* traduce un codice di alto livello in un codice assembly; l'*assemblatore* genera i file oggetto per il codice e per le librerie personali; il *linker* risolve le dipendenze statiche e genera un file eseguibile; il *loader* - integrato all'interno del sistema operativo - carica il programma nella memoria e risolve le dipendenze dinamiche (ad esempio caricando in fase di esecuzione le librerie dinamiche). Compilatori moderni (come ad esempio GCC/G++ per il linguaggio C/C++) sono in grado di effettuare tutti questi quattro passi, generando tutti i file a partire dai precedenti. Vi lascio [questo video](#) per approfondire l'argomento.

<sup>5</sup>Ho assunto che il vostro PC è basato sull'architettura x86 in quanto fino al 2020 la "totalità" dei PC, desktop e laptop, avevano al loro interno un processore Intel o AMD. Se invece utilizzate un Mac Mini o un Macbook dal 2021 in poi, la vostra macchina è dotata di un processore proprietario di Apple, l'M1 o M2 nelle loro rispettive varianti, basati su architettura ARMv8 e in futuro su ARMv9. I cellulari, invece, sono tutti basati su architettura ARMv7 o i più recenti su ARMv8. Inoltre, se a casa avete un Raspberry Pi 3/4 o poche altre schede con microcontrollori, allora avete a vostra disposizione un processore ARMv8.

la dimensione indicata né è possibile eseguire codici che presentano qualsiasi tipo di salto o scritture/letture su memoria (sebbene sia possibile assemblarli senza ricevere errori).

Un ulteriore strumento di controllo può essere [questo sito](#). Consente di scrivere un codice in un linguaggio di alto livello tra gli oltre 30 disponibili e di ricevere in output il corrispondente codice assembly per varie architetture con compilatori di diverse generazioni.

Le opzioni che vi suggerisco di selezionare per ottenere dei codici il più possibile simili a quelli presentati nel corso sono:

- Impostare il linguaggio di partenza a C;
- Impostare il compilatore ad ARMv8-a clang 9.0.0 (il meno recente);
- Aggiungere ai flag del compilatore `-O1`.

Anche così i risultati ottenuti non sono dei migliori. I codici semplici vengono correttamente tradotti con alcune leggere differenze dovute alle semplificazioni del LEGv8 rispetto all'ARMv8, mentre quelli complessi presentano delle espressioni un po' misteriose, ma tutto sommato forniscono una buona linea guida su come scrivere i codici LEGv8. Compilatori più nuovi ottimizzano i codici con nuove pseudo-istruzioni implementate in successive revisioni dell'ARMv8, che quindi non vengono trattate nel corso. Utilizzare diverse ottimizzazioni per il compilatore comporta un aumento delle istruzioni "misteriose" se si utilizza nessuna ottimizzazione, quindi `-O0` o nessun flag, oppure una drastica diminuzione delle righe di codice a causa delle pseudo-istruzioni implementate se si utilizzano i flag `-O2` e `-O3`. Così il codice diventa più corto, l'esecuzione richiede meno tempo e il lavoro di ottimizzazione viene fatto direttamente dallo scheduler hardware interno al processore, che è sicuramente più ottimizzato di quello generico implementato nei vari compilatori.

Se qualcuno dovesse riuscire a trovare un modo semplice per ovviare a questi problemi può contattarmi tramite Teams o comunicarlo al docente, in modo da aggiornare questa guida per i futuri studenti.

Due ulteriori modi per avere un ambiente Linux che gira su architettura ARM possono essere l'uso di un emulatore - come la versione *User-mode emulation* di [QEMU](#) - , oppure l'uso di [Termux](#) sul proprio cellulare Android.

Se invece avete un MacBook con processore M1 (o successivi) o un RaspberryPi (v3 o successivi) allora vi suggerisco di vedere la sezione [ARMv8 su RaspberryPi](#): vedremo come verificare ed eseguire tutti i codici svolti a lezione direttamente sul nostro dispositivo.

## 4 Operiamo sugli array

Prima di capire cosa sono gli array e come possiamo manipolarli efficacemente è meglio avere ben chiaro cosa sia un puntatore. Quando dichiariamo una variabile stiamo dedicando ad

essa una locazione della memoria di dimensione pari alla dimensione del tipo di dato richiesto. La memoria è indicizzata, ovvero ad ogni posizione nella memoria è associato un indirizzo di memoria. Ecco allora che risulta più chiaro il ruolo del puntatore: esso "punta", nel senso di indicizza, una variabile nella memoria. Ma quindi cos'è di preciso un puntatore? È una variabile anch'esso. In C li dichiariamo postponendo un asterisco `*` al tipo di variabile che stiamo dichiarando (oppure antepoendolo al nome); così stiamo specificando che ciò che verrà dichiarato è a tutti gli effetti un puntatore ad una variabile del tipo indicato. Un esempio è

```
1 int a = 5;
2 int* p;           // dichiaro il puntatore p di tipo intero
3 p = &a;          // assegno al puntatore p l'indirizzo della variabile intera a
4 printf("%d\n", *p); // stampa il valore contenuto nell'indirizzo puntato da p (ovvero il
   valore di a)
```

In questo codice abbiamo dichiarato una variabile di tipo intero `a`, abbiamo dichiarato un puntatore `p` di tipo intero, a cui associamo l'indirizzo di `a`. Poi mandiamo a schermo il contenuto della variabile puntata dal puntatore `p`, quindi a tutti gli effetti stiamo mandando a schermo il valore di `a`.

Notate l'utilizzo del simbolo `&` per indicare l'indirizzo di `a` e il duplice uso del simbolo `*`, usato sia in fase di dichiarazione dell'array, sia per leggere il valore della variabile da lui puntata. Vi invito a sperimentare tutte le possibili combinazioni di operazioni con questi due simboli e verificare quali sono quelle valide e quali non lo sono, ma soprattutto a provare ad "indovinare" quale sia il risultato atteso per ogni combinazione. È un ottimo lavoro per verificare di aver compreso appieno i puntatori.

Vediamo ora come si traduce tutto questo nel LEGv8.

Il LEGv8 è un'architettura a 64 bit<sup>6</sup>, quindi affinché tutte le posizioni di memoria siano indicizzate i puntatori devono essere delle variabili a 64 bit.

Per convenzione i puntatori indicizzano i Byte di memoria, quindi per semplicità nel LEGv8 l'unità più piccola di lavoro è il Byte, non il bit, o meglio, all'utente vengono dati comandi per lavorare con interi da 8 (Byte), 16 (Half-Word), 32 (Word) o 64 (Double Word) bit oppure con numeri a virgola mobile da 32 (Float) o 64 (Double) bit. Vi suggerisco di provare a creare un programma che estragga uno o più bit da un numero, o che sommi i bit di un numero. È un buon esercizio per sperimentare i comandi base di Assembly.<sup>7</sup>

---

<sup>6</sup>quindi la memoria virtuale massima supportata è idealmente di  $2^{64} = 18.4$  miliardi di miliardi di byte! 16 ExaByte ( $10^{19}$ )! È un enorme miglioramento dai  $2^{32} = 4$  GigaByte ( $10^9$ ) delle ormai obsolete architetture a 32 bit. Quando si sente parlare de "La nuova versione di Windows non *supporterà* più processori a 32 bit" si intende dire che le nuove versioni di Windows supporteranno solamente processori con registri a 64 bit, quindi non sarà possibile installare queste nuove versioni sui processori di 20 anni fa. Viceversa è ancora possibile installare su computer a 64 bit sistemi operativi (o più banalmente applicazioni) sviluppate con in mente architetture a 32 bit: saranno semplicemente inutilizzati i 32 bit alti dei vari registri.

<sup>7</sup>Ci sono molte soluzioni possibili. Queste sono alcune di quelle possibili:

- Fare un AND con il bit da estrarre e poi shiftare a destra;
- Shiftare a sinistra di  $63 - n$  posizioni a sinistra e poi 63 posizioni a destra

Se sfogliate il manuale di LEGv8 (o ARMv8, o x86, o RISC-V, o qualsiasi altra architettura) noterete che non esistono registri per i puntatori. Questo perché per l'ALU *tutto è numero*<sup>8</sup>. Per utilizzare i puntatori, quindi, dobbiamo utilizzare sempre i registri  $X0 \rightarrow X31$  e prestare attenzione all'interpretazione che *noi* facciamo del contenuto dei registri.

I puntatori possono indicizzare una singola variabile nella memoria, oppure possono puntare anche al primo valore di un array.

Un array è una raccolta di elementi dello stesso tipo disposti in locazioni di memoria consecutive. Per semplicità consideriamo array di *long long*, quindi di elementi a 64 bit. Una prima operazione con gli array potrebbe essere sommare 1 a tutti i suoi elementi. Per fare questo dobbiamo conoscere la posizione del primo elemento dell'array e la sua lunghezza. In C un possibile codice è il seguente

```
1 for(int i=0; i<n; i++)
2   a[i]++;
```

Ricevendo in input l'indirizzo dell'array in  $X0$  e la sua lunghezza in  $X1$ , in LEGv8 questo codice diventa

```
1 /*
2 input: X0 -> &a   X1 -> n
3 */
4
5     MOV X9, XZR    // i=0
6 while: CMP X9, X1 // i<n
7     B.GE exit
8     LSL X10, X9, #3 // i*8
9     ADD X10, X10, X0 // &arr[i]
10    LDUR X11, [X10, #0] // *arr[i]
11    ADDI X11, X11, #1 // arr[i]++
12    STUR X11, [X10, #0]
13    ADDI X9, X9, #1
14    B while
15 exit:
```

Ci sono 3 istruzioni nuove.

### LSL Logical Shift to the Left.

Sposta i 64 bit del registro  $X9$  di  $\#3$  posizioni a sinistra e memorizza il nuovo numero in  $X10$ . In binario fare uno shift a sinistra di una posizione significa moltiplicare per due, mentre fare uno shift a destra con **LSR** significa dividere un numero per 2 (arrotondato per difetto)<sup>9</sup>.

### LDUR LoaD Unscaled Register.

È l'istruzione del LEGv8 per scrivere nel primo registro ( $X11$ ) il valore contenuto nella posizione nella memoria principale specificata dal secondo registro ( $X10$ ) a cui vengono aggiunti  $\#0$  byte. Vengono letti quindi 8 byte (64 bit) a partire da quell'indirizzo.

---

<sup>8</sup>Pitagora ha da sempre avuto ragione

<sup>9</sup>Questo è valido solo se  $X9$  è un numero positivo. Se fosse negativo uno shift logico a destra aggiunge uno zero in testa, rendendolo positivo. Discorso analogo vale uno shift logico a sinistra: se la seconda cifra più significativa fosse 1 allora il numero diventerebbe negativo. Bisogna quindi prestare attenzione quando si lavora con i numeri con segno.

**STUR** STore Unscaled Register.

È l'operazione duale alla precedente, quindi serve per caricare il valore contenuto nel primo registro ( $X11$ ) nei 64 bit della memoria che partono dall'indirizzo contenuto nel secondo registro a cui viene sommato l'offset (in byte)  $X10 + \#0$ .

Abbiamo nominato più volte i registri e adesso abbiamo nominato la memoria principale. Quali sono le differenze? Molteplici.

Innanzitutto le dimensioni. I registri nel LEGv8 sono 64: 32 registri interi e 32 che supportano le operazioni a virgola mobile secondo lo standard IEEE-754 (vedi prossima sezione), mentre la memoria principale è formata da più chip di capienze che variano nell'ordine di 8 Gb a 128 Gb per chip, fornendo quindi una capacità totale di molti GB, se non TB nel caso dei server<sup>10</sup>. I registri sono pochi in quanto si trovano all'interno dell'ALU, questo per aver accesso al loro contenuto istantaneamente, ovvero il valore è già disponibile all'ALU per eseguire l'operazione in corso e non c'è alcun tempo di attesa dei dati. Il contrario avviene per la memoria principale, in quanto si trova più distante dall'ALU (all'esterno del processore)<sup>11</sup> ed ha tempi di accesso maggiori, dovuti a differenze architettoniche. Questo comporta delle latenze non trascurabili quando si deve reperire un dato dalla memoria principale, e per questo motivo i caricamenti e scaricamenti dalla memoria sono delle operazioni da ridurre il più possibile dove non siano necessarie.

Fattore importantissimo da ricordare è che mentre la memoria principale è indicizzabile, ovvero possono esistere dei puntatori che puntano ad un suo indirizzo, i registri *non* sono indicizzabili. Questo significa che non è possibile "puntare" ad un altro registro con un puntatore. O si passa direttamente il valore del registro alla funzione, oppure si carica il valore del registro nello stack (una parte della RAM, si veda la sezione 6) e lo si accede tramite un puntatore alla sua posizione nello stack, passando quest'ultimo alla funzione.

Ecco una tabella riassuntiva delle differenze.

---

<sup>10</sup>Un po' di numeri reali: il più piccolo banchetto di RAM DDR5 (l'ultima generazione per i PC al momento della scrittura) offre una capacità minima di 8 GB ed è costituito da 9 chip di memoria da 8 Gb ciascuno eccetto uno che è interamente dedicato all'ECC (Error Correction Code); un banchetto performante per server (DDR5) può fornire 512 GB per DIMM, divisi in 18 (=16+2) chip composti da 8 chip sovrapposti da 32 Gb ciascuno. Per approfondire ulteriormente l'argomento vi suggerisco [questa pagina della Micron sulle DDR5](#), [questa pagina di Tom's Hardware](#) sui chip di memoria di Samsung, nonché l'introduzione alle memorie nel corso di Dispositivi e Sistemi Elettronici.

<sup>11</sup>Per semplicità assumeremo che esista un'unica memoria principale. Nella realtà esiste la memoria virtuale, che è la massima memoria supportata dal sistema operativo, la memoria fisica, ovvero la capacità dei chip di RAM installata sul dispositivo sommata alla capacità della memoria d'archiviazione, e più livelli di cache. Questi ultimi sono interni al processore (ma esterni all'ALU) e hanno la funzione di memorizzare una piccola parte della RAM in un luogo più vicino all'ALU, in modo da ridurre i tempi d'accesso da qualche centinaio di cicli di clock a qualche decina se non anche meno di cicli.

	Registri	Memoria principale
Dimensioni	$2 \cdot (32 \cdot 8) = 512B$ nel LEGv8, in architetture più prestanti più del doppio per supportare unità avanzate	Molti GB, in alcuni casi (server) anche TB
Tempo di accesso	Istantaneo	1→500 cicli di clock
Posizione	Interni alla ALU	Esterna al processore, in uno suo slot dedicato oppure saldata sulla scheda madre nelle vicinanze del processore
Indicizzabile?	NO!	Si, a colpi di Byte

Tabella 2: Tabella riassuntiva delle differenze tra registri e memoria principale

## 5 Intermezzo 2: altri tipi di dati

Fino ad adesso abbiamo utilizzato solamente long long, ovvero interi a 64 bit. Il LEGv8, come l'ARMv8, l'x86 e altre architetture supportano anche altri formati dati, interi e a virgola mobile. Per poter essere gestiti però è necessario abilitare l'hardware al supporto di questo tipo di dati ed è quindi necessario utilizzare istruzioni specifiche per ciascuno di essi. Vediamo ad esempio come si può creare un programma che converte una stringa in maiuscolo. In C una stringa è un'array di caratteri, quindi un array di interi ad 8 bit (almeno che non vengano utilizzate librerie dedicate). Questo significa che operiamo con dei byte e non più con delle doubleword. Questo si rifletterà sulla lettura e scrittura dei dati, ma in fase di utilizzo non ci saranno differenze. Questo perché stiamo abilitando l'hardware a lavorare solamente con la parte bassa dei registri X (ricordo, *repetita iuvant*, che i registri X sono registri a 64 bit), nel caso dei byte stiamo usando solamente gli 8 bit bassi del registro. Per questo motivo quando si lavora con byte, half-word o word si utilizzano i registri W.

Attenzione! Non viene gestito l'overflow o underflow, in quanto in fase di caricamento con *LDURB* (LoaD Unscaled Register Byte) sarà l'hardware aggiuntivo a mettere gli 8 bit nella parte bassa del registro e ad azzerare gli altri 56 bit. Questo implica che poi possiamo utilizzare le classi che operano sui registri a 64 bit e se dovessimo tener conto di overflow o underflow non possiamo far uso dei flag (in quanto non vengono attivati dalla logica) ma dobbiamo effettuare noi dei controlli manuali con delle sottrazioni (nel caso del byte, ad esempio) di  $2^8 = 256$  (oppure traslare a destra il registro [supponiamo W9] di 8 bit con *LSR XZR, W9, #8*) e verificare se è 0 oppure 1 l'ultima cifra.<sup>12</sup>

<sup>12</sup>Utilizzando il *SUBS* stiamo facendo due operazioni in una: sia sottraiamo che attiviamo i flag, poi dobbiamo effettuare un branch condizionale. Lo shift logico, invece, non attiva i flag, i quali devono essere attivati successivamente tramite un *CBZ/CBNZ*.

```

1 while(*arr!=0){
2     if(*arr>=97 && *arr<=122)
3         *arr -= 32;
4     p++;
5 }

```

## 5.1: Stringa maiuscola

In assembly diventa

```

1 /*
2 X0 -> *arr      X9 -> temp
3 */
4
5 strMaiusc: LDURB X9, [X0, #0] // temp=*arr
6           CBZ X9 exit      // while(*arr!=0)
7
8           CMPI X9, #97     // if(*arr>=97)
9           B.LT finewhile
10          CMPI X9, #122    // if(*arr<=122)
11          B.GT finewhile
12          SUBI X9, X9, #32 // temp-=32
13          STURB X9, [X0, #0] // *arr=temp
14
15 finewhile: ADDI X0, X0, #1 // arr++
16           B strMaiusc
17 exit:

```

## 5.1: Stringa maiuscola

Esistono ulteriori modi per ottimizzare questo codice, ma non sono applicabili nel LEGv8<sup>13</sup>, quindi se foste interessati ad ottimizzarlo ulteriormente vi invito a leggere la sezione [Ottimizziamo i codici: Branchless Programming, Loop Unrolling e SIMD](#).

Vediamo ora come vengono rappresentati i numeri a virgola mobile.

### 5.1 IEEE 754

In C esistono due tipi di numeri a virgola mobile: i float (32 bit) e i double (64 bit). In LEGv8 questi vengono memorizzati nei 32 registri D<sup>14</sup>. Al corso di Reti Logiche abbiamo visto com'è possibile rappresentare i numeri decimali secondo la notazione posizionale, ovvero scegliendo dopo quale bit posizionare la virgola e nominando parte intera tutti i bit alla sua sinistra e parte decimale quelli alla sua destra. Questo metodo però non è efficace per numeri molto grandi o molto piccoli. Per ovviare a questo difetto in base dieci utilizziamo la notazione scientifica, ovvero c'è un numero compreso tra 1 e 10 (quest'ultimo escluso) moltiplicato per una potenza

<sup>13</sup>In realtà è possibile "sbrogliare il ciclo", ma visto che nel LEGv8 non c'è la possibilità di lavorare in parallelo sui dati allora si rischia soltanto di complicare la comprensione del codice, quindi relego anche questa sezione agli approfondimenti.

<sup>14</sup>I registri D sono registri a 64 bit distinti dai registri X. Esistono poi i registri S che sono la parte bassa dei registri D (quindi fisicamente sono gli stessi registri, ma dei quali vengono usati solo metà bit).

intera di 10. È possibile fare la stessa cosa in binario e lo standard IEEE 754 ne fornisce un'implementazione. Il LEGv8 implementa lo standard IEEE 754-1985 a 32 e 64 bit, dove il primo bit indica il segno del numero, ovvero è 0 se il numero è positivo e 1 se è negativo; l'esponente di 8 o 11 bit (a seconda che si faccia uso dello standard a 32 o a 64 bit), e infine la frazione di 23 o 52 bit. Il numero così rappresentato sarà quindi

$$(-1)^{\text{bitDiSegno}} \cdot (1 + \text{frazione}) \cdot 2^{\text{esponente} - \text{bias}}$$

Cosa sono quell' $(1 + \text{frazione})$  e quell' $\text{esponente} - \text{bias}$ ? Nella notazione binaria la prima cifra significativa sarà sempre un 1, quindi non è necessario memorizzarla. La mantissa, ovvero il numero senza esponente sarà quindi  $(1 + \text{frazione})$ . Noi vogliamo rappresentare numeri sia grandi, ma anche piccoli. Fa quindi comodo avere un'esponente che possa essere sia negativo che positivo. Ecco quindi che all'esponente effettivo del numero aggiungiamo un bias, ovvero scaliamo tutto di metà intervallo rappresentabile ( $2^{(8-1)} - 1 = 127$  per lo standard a 32 bit e  $2^{(11-1)} - 1 = 1023$  per quello a 64 bit).

Se ad esempio dobbiamo rappresentare il numero  $(19.34865)_{10}$  questo è il procedimento da seguire. Per prima cosa convertiamo la parte intera guardando il resto della divisione per due, quindi

$$\begin{aligned} 19\%2 &= 1 \\ 9\%2 &= 1 \\ 4\%2 &= 0 \quad \Rightarrow \quad (19)_{10} = (10011)_2 \\ 2\%2 &= 0 \\ 1\%2 &= 1 \end{aligned}$$

Si procede ora con la parte decimale, moltiplicando per due la parte decimale del passo precedente e posizionando il bit della parte intera del risultato della moltiplicazione precedente nella frazione, quindi

$$\begin{aligned} 0.34865 \cdot 2 &= 0.6973 \\ 0.6973 \cdot 2 &= 1.3946 \\ 0.3946 \cdot 2 &= 0.7892 \quad \Rightarrow \quad (0.34865)_{10} = (01011001010000010010000001011011110000000001\dots)_2^{15} \\ 0.7892 \cdot 2 &= 1.5784 \\ &\vdots \end{aligned}$$

La mantissa sarà quindi la concatenazione delle due stringhe appena trovate, quindi  $1001101011001010000010010000001011011110000000001\dots$

Non serve memorizzare il primo bit, quindi possiamo ometterlo dalla frazione, i successivi 23 o 52 bit sono quelli significativi.

Lavoriamo ora con l'esponente: la virgola nel numero appena scritto starebbe dopo la quinta cifra, quindi l'esponente sarà  $5 + 127(1023) = 132(1028)$ , che dobbiamo trasformare in binario. Le due rappresentazioni negli standard IEEE 754 a 32 e 64 bit sono quindi

---

<sup>15</sup>Stiamo assumendo che la virgola sia prima del primo bit, in quanto stiamo rappresentando solamente la parte decimale di un numero

	32 bit	64 bit
segno	0	0
esponente	10000100	10000000100
frazione	00110101100101000001001	0011010110010100000100100000010110111100000000011010
numero completo (in esadecimale)	42 1A CA 09	40 43 59 41 20 5B C0 1A
numero decimale realmente rappresentato	19.3486499786376953125	19.34864999999999923829818726517260074615478515625

Notiamo come i due numeri decimali realmente rappresentati non sono esattamente il numero di partenza, questo perché il numero di bit utilizzati è finito e inferiore al numero di bit necessario per rappresentare interamente in binario il numero decimale di partenza.

Ovviamente un computer non deve fare questa conversione, in quanto non sa leggere la notazione decimale. Quello che può fare, invece, è convertire un numero intero in uno a virgola mobile e viceversa. Queste unità di calcolo sono presenti nell'ARMv8, ma non sono presenti nel LEGv8 (come molte altre che non vengono trattate nel corso di Architetture dei Sistemi Digitali).

Vediamo ora quali sono i numeri speciali dello standard IEEE 754.

	32 bit		64 bit		generico	
	Esponente	Frazione	Esponente	Frazione	Esponente	Frazione
0	0	0	0	0	0	0
Qualsiasi numero	1-254	qualsiasi	1-2046	qualsiasi	$1 - 2^{(\#bitExp-1)} - 1$	qualsiasi
$\pm\infty$	255	0	2047	0	$2^{(\#bitExp-1)}$	0
NaN (Not a Number)	255	$\neq 0$	2047	$\neq 0$	$2^{(\#bitExp-1)}$	$\neq 0$
$\pm$ Numeri denormalizzati <sup>16</sup>	0	$\neq 0$	0	$\neq 0$	0	$\neq 0$

<sup>16</sup>I numeri denormalizzati consentono di rappresentare numeri ancora più piccoli (in modulo, quindi con esponente più piccolo) rispetto al minimo numero "normale" rappresentabile. Questo è possibile in quanto il bit nascosto è 0 (ovvero quel bit che non è necessario salvare in quanto normalmente è sempre 1, quindi omettiamo dalla notazione frazionale), ecco perché il nome "denormalizzato": il numero non è più rappresentato nell'equivalente binario della notazione scientifica. Il numero diventa quindi  $(-1)^{\text{bitDiSegno}} \cdot (\text{frazione}) \cdot 2^{1 - \text{bias}}$



[questo video.](#)

```
1 float FastInverseSquareRoot( float n){
2
3     long i;
4     float x, y;
5     const float treMezzi = 1.5F;
6
7     x= n*0.5F;
8     y=n; //creiamo una copia lavorativa del numero
9     i= *(long *) &y; // convertiamo bit a bit il float in un intero cosi' da poterlo manipolare
10    i= 0x5f3759df -(i >> 1); //trucchetto matematico con i logaritmi. il numero sarebbe
        1.3211836173e+19
11    y= *(float *) &i; //riconversione a float bit a bit
12    y = y * (treMezzi - (x*y*y)); //metodo di newton per eliminare l'errore
13    //y = y * (treMezzi - (x*y*y)); // seconda iterazione (facoltativa) per migliorare
        ulteriormente il risultato
14    return y;
15 }
```

## 5.2: Fast Inverse Square Root

Ed ecco il corrispondente codice Assembly

```
1 /*
2 input: X0 -> &(0x5f3759df)   X1 -> &0.5F   X2 -> &1.5F   S0 -> n
3     X9 -> &n(stack)   X10 -> i   X15 -> 0x5f3759df
4     S5 -> 0.5f   S15 -> 1.5F
5 output: S0 -> y
6 */
7
8 LDURS S5, [X1,#0] // carico nei registri le costanti
9 LDURS S15, [X2,#0]
10 LDURW X15, [X0, #0]
11
12 FMULS S10, S5, S0 // x=n*0.5F (lasciamo y in S0 visto che n non ci serve piu')
13
14 SUBI SP, SP, #16 // allochiamo una quadword nello stack come da convenzione
15 STURS S0, [SP, #0]
16 LDURW X9, [SP, #0] // i=*(long *) &y
17
18 LSL X9, X9, #1 // i >> 1
19 SUB X9, X15, X9 // i = 0x5f3759df - (i >> 1)
20
21 STURW X9, [SP, #0]
22 LDURS S0, [SP, #0] // y= *(float *) &i
23 ADDI SP, SP, #16 // ripristiniamo lo stack
24
25 FMULS S1, S0, S0 // y*y
26 FMULS S1, S1, S10 // x*y*y
27 FSUBS S1, S15, S1 // 1.5F - x*y*y
28 FMULS S0, S1, S0 // y = y * (1.5F - x*y*y)
```

## 5.2: Fast Inverse Square Root

dividerlo per il suo modulo. Il modulo di un vettore si calcola con la radice quadrata della somma del quadrato delle componenti, quindi potenzialmente basterebbe chiamare le due operazioni di radice quadrata e divisione implementate direttamente nelle librerie standard. Sì, è possibile ed è estremamente dispendioso in termini di risorse; è ciò che è sempre stato fatto prima che nel 1999 gli sviluppatori del videogioco Quake 3 ideassero questo trucchetto matematico. Ecco spiegata l'utilità di quest'algoritmo.

Notate come sia necessario caricare da memoria tutte le costanti float (quella intera avremmo potuto inserirla manualmente) e l'uso che viene fatto dello stack: non stiamo convertendo un intero in un float e viceversa, stiamo spostando bit a bit la stessa sequenza di 32 bit da un registro intero ad uno float. Il trucco di quest'algoritmo sta proprio nel manipolare un numero a virgola mobile come fosse un intero (a causa della notazione dei numeri a virgola mobile dello standard IEEE 754).

Come esercizio per casa vi suggerisco di fare un codice che esegue la media aritmetica di un array di double.

```
1 double mediaAritmetica(double *arr, long long len){
2
3     double *fineArr; // fineArr e' un puntatore. L'idea e' di ciclare sull'array tramite
      puntatore
4     double A=0;
5     fineArr=arr+len; // punto alla fine dell'array per sapere quando fermarmi
6     while(arr<fineArr){ //\0 corrisponde al numero 0, quindi e' come ...
7         A += *arr; // ... scrivere while(str[i]!=0) oppure direttamente while(str[i])
8         arr++;
9     }
10    return A/(double)len; // attenzione! pensate bene a come fare questa conversione (casting)!
11
12 }
```

### 5.3: Media aritmetica

Bisogna prestare molta attenzione all'utilizzo delle costanti e della memoria. Si possono fare diverse scelte su cosa passare come parametro (o puntatore ad una posizione nello stack) oppure calcolarlo con altre variabili. Nel [Soluzione esercizi](#) ho scelto di ricevere in input il puntatore all'array, la lunghezza della stringa (solo come intero) e il puntatore alla costante  $1.F$ , mentre l'output è nel registro  $D0$ . Una scelta differente per l'output potrebbe consistere nel caricare il valore della media aritmetica in una locazione di memoria stabilita (della quale abbiamo ricevuto in input l'indirizzo). Adesso queste scelte sono molto arbitrarie (anzi, vi invito a provare tutte queste combinazioni); nel mondo del lavoro invece queste scelte saranno dettate da richieste più specifiche o da programmi pre-esistenti che dispongono i dati in determinate locazioni di memoria.

## 6 Programmazione funzionale

Fino ad adesso abbiamo lavorato solamente con parti del corpo dei vari codici. Prendiamo ad esempio il primo programma ([Sommiamo!](#)). Abbiamo codificato solo la seconda riga e abbiamo ridotto il *return* a un semplice spostamento del risultato nel registro prefissato (nel nostro caso  $X0$ ).

Questo va bene finché il codice che abbiamo scritto viene inserito all'interno di un programma, ma se invece dovessimo creare una funzione che esegue questo estratto di codice e basta, ricevendo gli input e scrivendo l'output in determinati registri da e per una procedura chiamante,

è necessario sapere come restituire il controllo a questa funzione. Ecco allora che il nostro programma viene chiamato programma foglia. Un programma foglia è un codice che non chiama nessun'altra funzione; un programma non foglia invece chiama almeno una funzione, che può essere anche sé stessa ([programmi ricorsivi](#)).

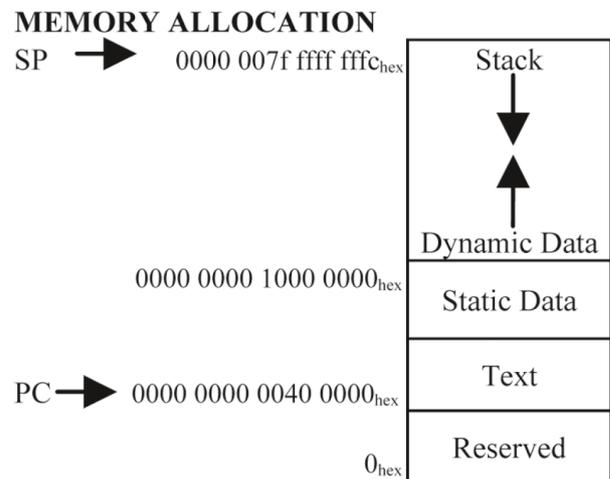
Rendiamo foglia la [2.1: prima somma](#). Ci aspettiamo di ricevere i due input in X0 e X1 e scriviamo l'output in X0, come da convenzione.

```
1 somma:  ADD X0, X0, X1
2         BR LR
```

Abbiamo dovuto aggiungere l'etichetta corrispondente all'inizio della procedura e l'istruzione di *branch to register*, ovvero di salto a un registro. Vedremo meglio nel prossimo capitolo i dettagli di questo (e degli altri) salti, ci basta intanto sapere che l'istruzione *BR LR* è il modo nel LEGv8 di ritornare il controllo alla procedura chiamante.

Ottimo. Vediamo ora come chiamare la funzione somma appena creata da un altro metodo. Prima però dobbiamo introdurre un nuovo concetto: lo stack.

Quando dobbiamo caricare un programma in memoria il loader dedica una o più pagine a questo programma, in base alla dimensione del testo, dei dati statici e dei potenziali dati dinamici. La dimensione delle singole pagine è fissa ed è dettata dal sistema operativo (nel caso di Windows su un processore x86 sono normalmente *4kB*) e all'interno di questa/e pagina/e allocate nella memoria virtuale vengono collocati il testo del programma e i dati statici nella parte bassa della pagina. Il resto dello spazio viene lasciato libero alla memoria dinamica (che si estende dal basso verso l'alto) e allo stack (che si estende dall'alto verso il basso).



La memoria dinamica serve ad allocare nuove variabili in corso d'opera, ad esempio se si vuole dichiarare un array di 1000 long long che verrà utilizzato temporaneamente solamente durante l'esecuzione del programma questo non potrà essere interamente contenuto nei registri del processore. Bisogna allocarlo nella memoria dinamica, tenere nel processore il puntatore all'array e leggere e scrivere pochi dati alla volta, quanti sono concessi dal numero di registri disponibili. Nel LEGv8 è possibile utilizzare la memoria dinamica, ma quest'argomento non fa parte del corso, quindi noi anche per allocare array faremo uso dello stack.

Lo stack serve invece a tener traccia delle chiamate alle varie funzioni e poter ripristinare il processore ad uno stato precedente. Quando dobbiamo chiamare una funzione dobbiamo effettuare un salto (vediamo meglio nel capitolo successivo cosa significa), ma poi vogliamo ritornare al punto del programma dove ci siamo fermati e ritrovarci gli stessi valori nei registri (tuttalpiù con un valore modificato dalla chiamata alla procedura). Ecco quindi che vogliamo salvare sicuramente il punto di ritorno (quindi l'indirizzo dell'istruzione di chiamata alla subroutine). Questo

compito è delegato al *BL*, o *branch and link*, il quale prima di saltare alla nuova procedura salva l'indirizzo dell'istruzione corrente nel *Link Register*, il registro X30 o LR.

Vi ricordate il *BR LR* della procedura foglia? Serve a caricare il valore<sup>18</sup> contenuto nel Link Register nel Program Counter (*PC*). Visto che stiamo sovrascrivendo il valore del link register e questo è uno dei registri che dev'essere preservato nelle chiamate (per evidenti motivi) è necessario salvare nello stack il precedente valore del link register.

Come si accede allo stack? Con lo *Stack Pointer*. Questo è un registro che contiene l'indirizzo dell'ultima locazione di memoria occupata dallo stack. Abbiamo visto che lo stack si trova nella parte alta della pagina di memoria del programma, quindi per allocare nuovo spazio nello stack si decrementa il valore dello Stack Pointer dello spazio necessario<sup>19</sup>. Bisogna ricordarsi poi che le procedure che fanno uso dello stack devono ripristinare il precedente valore dello stack pointer prima di ritornare il controllo alla procedura chiamante.

Eventualmente *possono* essere salvati nello stack ulteriori registri che non vengono preservati nelle chiamate a procedure (si veda tabella 1), mentre *devono* venir salvati nello stack i registri che necessitano di essere preservati nelle chiamate a procedura. Ovviamente non ha senso salvare registri che non vengono sporcati dalla procedura, ma solo quelli che verranno utilizzati dal programma.

Vediamo ora un esempio con un programma di ordinamento che fa uso di una funzione swap.

```
1 void swap(long long *a){
2     long long temp;
3     temp = *a;
4     *a = *(a+1);
5     *(a+1) = temp;
6 }
7
8 void bubbleSort(long long *arr, long long len){
9     long long i, j;
10    for (i=1; i<len; i++){
11        for (j=i-1; j>=0; j--){
12            if (arr[j]>arr[j+1]){
13                swap(&arr[j]);
14            }
15        }
16    }
17 }
```

## 6.1: bubbleSort

---

<sup>18</sup>Il Link Register contiene l'indirizzo dell'istruzione del programma precedente alla quale l'esecuzione si era fermata prima di chiamare questa procedura

<sup>19</sup>In realtà esiste una convenzione nell'ARMv8 che viene estesa anche al LEGv8, la quale impone che lo stack venga allocato a colpi di quadword, ovvero 16 Byte. Questo perché il LEGv8 è dotato di istruzioni in grado di leggere e scrivere su memora fino a 128 bit, quindi 16 Byte.

```

1  /* swap
2  input: X0 => &a
3  */
4  swap: LDUR X9, [X0, #0]    // copyOfa[j]
5         LDUR X10, [X0, #8] // copyOfa[j+1]
6         STUR X10, [X0, #0] // a[j+1]= copyOfa[j]
7         STUR X9, [X0, #8]  // a[j] = copyOfa[j+1]
8         BR LR              // torno il controllo alla procedura chiamante
9
10 /* bubbleSort
11 input: X0 => &arr  X1 => len
12 */
13 bubbleSort: SUBI SP, SP, #48 // alloco tre quadword nello stack come da convenzione ...
14             STUR LR, [SP, #0] // ... anche se devo memorizzare solo 5 doubleword
15             STUR X19, [SP, #8] // len
16             STUR X20, [SP, #16] // i
17             STUR X21, [SP, #24] // j
18             STUR X22, [SP, #32] // &arr
19
20             ADDI X20, XZR, #1 // i=1
21             MOV X19, X1      // copio len e l'indirizzo dell'array in altri registri ...
22             MOV X22, X0      // ... cosi' vengono preservati anche dopo le chiamate a swap
23 extFor:     CMPI X20, X19    // i=len
24             B.GE exit
25
26             SUBI X21, X20, #1 // j=i-1
27 intFor:     CMP X21, XZR     // j=0
28             B.LT endExtFor
29
30             LSL X9, X21, #3  // j*8
31             ADDI X9, X9, X    // &arr[j]
32             LDUR X10, [X0, #0] // arr[j]
33             LDUR X11, [X0, #8] // arr[j+1]
34             CMP X10, X11     // arr[j]-arr[j-1]
35             B.GE endIntFor
36             MOV X0, X9       // metto &arr[j] in X0 per swap
37             BL swap         // chiamo swap
38
39 endIntFor:  SUBI X21, X21, #1 // j--
40             B intFor
41
42 endExtFor:  ADDI X20, X20, #1 // i++
43             B extFor
44
45 exit:      LDUR LR, [SP, #0] // ripristino i 5 valori modificati nello stack
46             LDUR X19, [SP, #8]
47             LDUR X20, [SP, #16]
48             LDUR X21, [SP, #24]
49             LDUR X22, [SP, #32]
50             ADDI SP, SP, #48 // ripristino lo stack
51             BR LR          // "return void" (torno il controllo al chiamante)

```

## 6.1: bubbleSort

Vi propongo come esercizio due versioni di Fibonacci, entrambe iterative.

```

1 long long fib(long long n){
2     long long a = 0, b = 1, temp;
3     for(n-=2; n>=0; n--){
4         temp = a;
5         a = b;
6         b += temp;
7     }
8     return b;
9 }

```

### 6.1: Fibonacci iterativo

Per questa seconda versione è necessario utilizzare la memoria dinamica (noi useremo lo stack per mancanza di istruzioni) per creare un array di memorizzazione.

```

1 long long fib(long long n){
2     long long f[n];
3     f[0]=0;
4     f[1]=1;
5     for(long long i=2; i<=n; i++){
6         f[i] = f[i-1] + f[i-2];
7     }
8     return f[n];
9 }

```

### 6.2: Fibonacci iterativo con memorizzazione di tutti i valori calcolati

Le soluzioni le trovate sempre nella sezione [12](#)

## 7 Intermezzo 3: i salti

È estremamente importante capire le differenze tra i tre tipi di branch (salti), in modo da saperli utilizzare al meglio (e possibilmente il minor numero di volte possibile, ma pur sempre facendo funzionare i programmi).

Abbiamo incontrato vari tipi di salti, i quali possono essere raggruppati in tre categorie in base al loro funzionamento:

**Salti e salti condizionali:** In questa categoria rientrano le istruzioni

**B:** Branch, ovvero un salto incondizionale all'indirizzo dell'istruzione puntata dall'etichetta;

**B.cond:** Branch if condition, ovvero un salto condizionale all'indirizzo dell'istruzione puntata dall'etichetta, che viene effettuato solo se la condizione *.cond* è verificata, quindi se sono attivi i flag specifici (vedi paragrafo [7.1](#) e tabella [3](#) per approfondire il funzionamento dei flag);

**CBZ:** Compare and Branch if Zero, ovvero effettua un salto solo se il registro da noi indicato contiene il valore 0;

**CBNZ:** Compare and Branch if Not Zero, come il precedente, ma viene effettuato il salto solo se il registro contiene un valore diverso da zero;  
(queste ultime due istruzioni sono utili per aumentare la leggibilità del codice Assembly, ma per il computer equivalgono rispettivamente a *CMPI X0, #0 + B.EQ X0 label* oppure *CMPI X0, #0 + B.NE X0 label*)

Questi quattro salti hanno in comune lo stesso funzionamento, ovvero sostituiscono al Program Counter l'indirizzo dell'istruzione indicata con *label*. Più precisamente, all'attuale valore del PC sommano un offset determinato in fase di linking/loading del programma.

**Branch and Link:** In questa categoria rientra solamente il **BL**, ovvero il Branch and Link. Con quest'istruzione stiamo chiedendo al processore di scrivere l'attuale valore del PC nel Link Register (LR, o X30) e successivamente di inserire nel PC l'indirizzo dell'istruzione indicata con *label* (come prima, aggiungendo un offset all'attuale valore del PC).

**Branch to Register:** Il **BR** (branch to register), infine, fa parte di quest'ultima categoria. Stiamo dicendo al processore di leggere il valore contenuto nel LR e metterlo nel PC.

Rivediamo ora quando si usa quale tipo di salto:

- Usiamo i branch/branch condizionali quando dobbiamo tradurre un'istruzione *if* o un ciclo *while/for* (vedi sezioni 2 e 4);
- Usiamo i branch and link quando dobbiamo chiamare una funzione per poi dover riprendere l'esecuzione del codice al punto di interruzione (vedi sezione 6);
- Usiamo i branch to register quando dobbiamo uscire dall'esecuzione di una funzione, tornando quindi il controllo alla funzione chiamante (vedi sezione 6).

## 7.1 Operazioni condizionali

Abbiamo nominato più volte le operazioni condizionali e abbiamo detto che queste fanno uso dei bit di flag attivati dall'ALU quando chiamiamo un'operazione intera "che finisce in S" (quindi ADDS, ADDIS, SUBS, SUBIS, ANDS, ANDIS). Vediamo ora un piccolo approfondimento sulla logica hardware responsabile di questi controlli.

I 4 flag sono quattro bit di un registro privato, ovvero leggibile solamente dal processore e non modificabile direttamente dall'utente. Quando invochiamo l'istruzione SUBS viene attivata la logica per eseguire la classica operazione di sottrazione e in più viene ordinato al controllore di abilitare il registro privato dei flag alla scrittura, così l'ALU può sovrascrivere questi quattro bit. Per essere più precisi, l'ALU ha delle connessioni che calcolano i flag ad ogni operazione algebrica, ma questi vengono scritti solamente quando l'unità di controllo abilita la scrittura sul registro privato, ovvero solamente quando invochiamo istruzioni algebriche intere che terminano per S.

I quattro flag del registro NZCV sono

- N (negative)** Se il risultato dell'operazione ha un 1 nel bit più significativo questo bit viene posto ad 1, altrimenti a 0;
- Z (zero)** Se il risultato dell'operazione era 0 (ovvero tutti i bit del registro di output sono zero) allora questo bit di flag viene posto ad 1;
- C (carry)** Come il caso precedente, ma per operazioni con i numeri unsigned, quindi se il risultato non è rappresentabile in 64 bit, il bit di flag viene impostato ad 1;
- V (overflow)** Se la somma o differenza di due numeri con segno va in overflow, ovvero genera un numero non rappresentabile in 63 bit, andando quindi a sovrascrivere il bit di segno, il bit di flag viene impostato ad 1;

	Signed numbers		Unsigned numbers	
=	B.EQ	Z=1	B.EQ	Z=1
≠	B.NE	Z=0	B.NE	Z=0
<	B.LT	N!=V	B.LO	C=0
≤	B.LE	$\overline{Z = 0 \ \& \ N = V}$	B.LS	$\overline{Z = 0 \ \& \ C = 1}$
>	B.GT	Z=0 & N=V	B.HI	Z=0 & C=1
≥	B.GE	N=V	B.HS	C=1

Tabella 3: Condizioni per i salti condizionali e loro verifica hardware

## 8 Ricorsione

Un ulteriore livello di complessità è dato dalla ricorsione, ovvero far chiamare ad una funzione sé stessa. Se si ha seguito tutto fino a questo punto non ci sono novità rispetto alle sezioni precedenti, se non il fatto che un programma ricorsivo in assembly fa uso di tutto quanto abbiamo visto fin'ora. Quindi dove sta la difficoltà?

Probabilmente avrete già incontrato questa tecnica di programmazione (estremamente inefficiente e da evitare a qualsiasi costo in ambito reale, se possibile) in Fondamenti di Informatica.

Un esempio banale è il calcolo del fattoriale dell'n-esimo numero. La sua versione iterativa è

```

1 fact (long long n){
2   long long ris=1;
3   while (n>1){
4     ris*=n;
5     n--;
6   }
7 }
```

### 8.1: Fattoriale iterativo

```

1 /*
2 input: X0 -> n
3   X9 -> fact(n)_parziale
4 output: X0 -> fact(n)
5 */
6
7 fact:  ADDI X9, XZR, #1 // ris=1
8
9 while: CMPI X0, #1      // se(n-1)<=0
10       B.LE end        // allora esci
11       MUL X9, X9, X0   // ris*=n
12       SUBI X0, X0, #1  // n--
13       B while         // cicla sul while
14
15 end:   MOV X0, X9      // sposta il risultato in X0

```

## 8.1: Fattoriale iterativo (assembly)

Una versione ricorsiva, invece, può essere la seguente

```

1 factRic(long long n){
2   if(n<=1)
3     return n;
4   return n*factRic(n-1);
5 }

```

## 8.2: Fattoriale ricorsivo

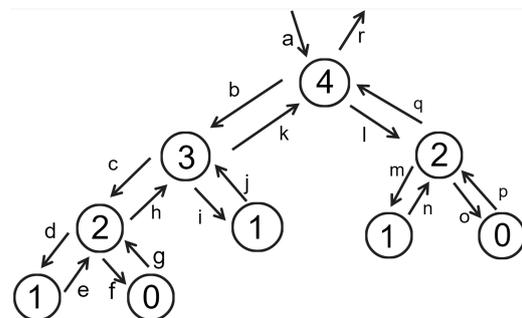
```

1 /*
2 input: X0 -> n
3 output: X0 -> factRic(n)
4 */
5
6 factRic: CMPI X0, #1      // if(n<=1)
7         B.LE end        // esci (return n)
8
9         SUBI SP, SP, #16 // preparo lo stack (a colpi di quadword) per memorizzare
10        STUR X0, [SP, #0] // n
11        STUR LR, [SP, #8] // LR
12        SUBI X0, X0, #1   // preparo n-1 in X0
13        BL factRic       // e chiamo factRic(n-1)
14
15        LDUR X9, [SP, #0] // ripristino n
16        MUL X0, X0, X9    // factRic(n-1) * n
17        LDUR LR, [SP, #8] // ripristino LR
18        ADDI SP, SP, #16 // ripristino SP
19 end:

```

## 8.2: Fattoriale ricorsivo (assembly)

L'esecuzione di un programma ricorsivo può essere visualizzata come un cammino su un albero (un grafo non orientato privo di cicli). Esistono vari modi per passare da un ramo all'altro dell'albero, io ho scelto di ripristinare il valore di  $n$  prima di



ritornare dal primo ramo; si poteva scegliere di ripristinarlo dopo il secondo ramo.

Vediamo ora la versione ricorsiva della sequenza di Fibonacci.

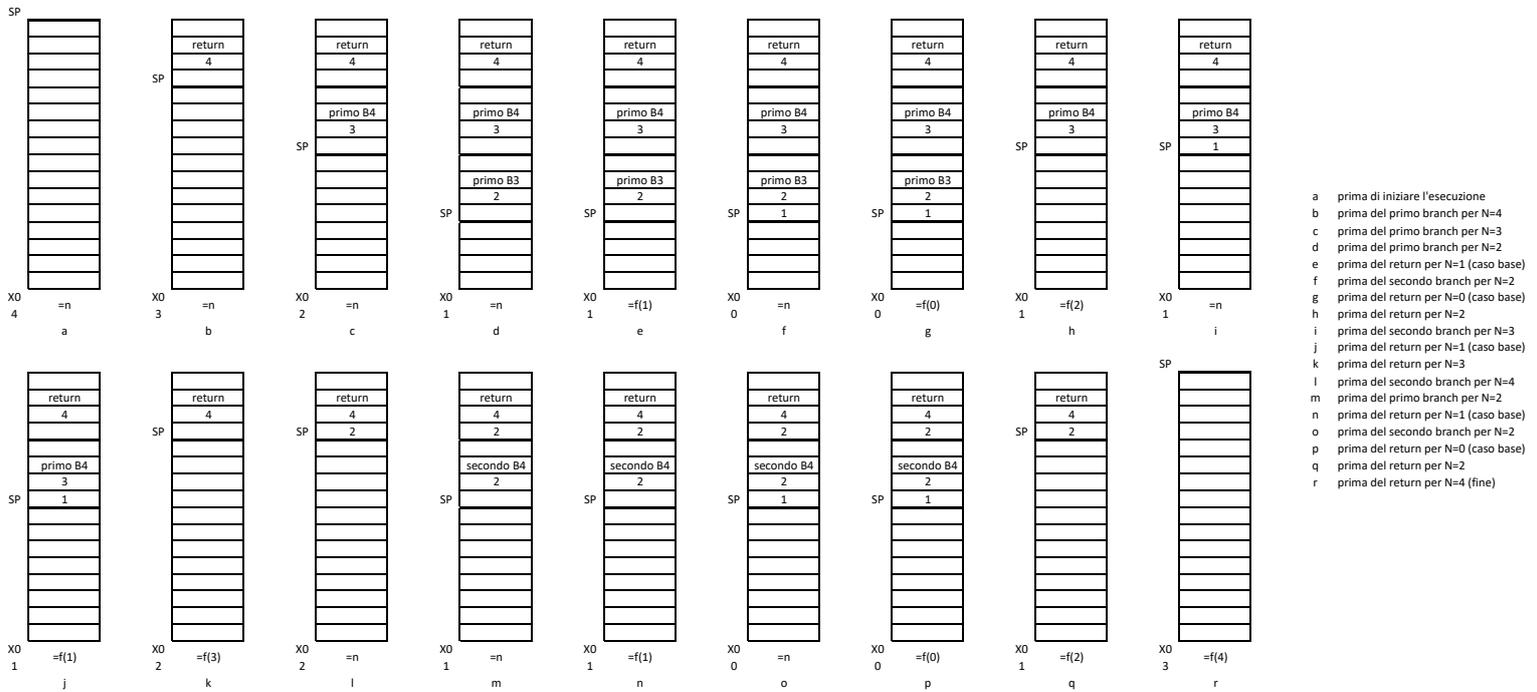
```
1 long long fib(long long n){
2     if (n<=1)
3         return n;
4     return fib(n-1) + fib(n-2);
5 }
```

### 8.3: Fibonacci ricorsivo

```
1 Fib:  CMP X0, #1      // vedi se siamo nei casi base (n==0 || n==1)
2       B.LE  Exit      // se siamo nei casi base allora esci (X0: 0 se n==0, 1 se n==1)
3
4 Else: SUBI  SP, SP, #32 // preparo lo stack per ricevere LR, n e fib(n)
5       STUR LR, [SP, #16]
6       STUR X0, [SP, #8]
7
8       SUBI  X0, X0, #1 // n-1
9       BL   Fib        // fib(n-1)
10      STUR  X0, [SP, #0] // memorizzo il nuovo fib(n-1)
11
12      LDUR  X0, [SP, #8] // ripristino il valore di n
13      SUBI  X0, X0, #2 // n-2
14      BL   Fib        // fib(n-2)
15
16      LDUR  X9, [SP, #0] // ripristino fib(n-1)
17      LDUR  LR, [SP, #16] // ripristino link register
18      ADD   X0, X9, X0 // fib(n) = fib(n-1) + fib(n-2)
19      ADDI  SP, SP, #32 // svuoto lo stack
20
21 Exit: BR   LR        // ritorno fib(n) memorizzato in X1
```

### 8.3: Fibonacci ricorsivo (assembly)

È un po' complesso tener conto dello stato dei registri (e delle variabili) ad ogni passaggio. Ho quindi pensato di rappresentare i passaggi nello schema alla pagina seguente, dove gli "stati" di visualizzazione sono gli stessi rappresentati nel diagramma ad albero della pagina precedente. Un altro modo estremamente utile per visualizzare lo stato dei registri e dello stack è utilizzando [ARMv8 su RaspberryPi](#).



### Alcune note su questo codice

- Nel caso base (quindi le prime due righe di codice) non ho allocato nuovo spazio nello stack. Non è necessario. Non è nemmeno un errore se lo avessimo allocato anche qui, ma è uno spreco di memoria (e l'obiettivo della programmazione assembly è quello di azzerare gli sprechi di risorse dando il completo controllo al programmatore).
- Ci potrebbe essere un piccolo "baco" nel programma nel caso in cui il codice ricevesse un numero negativo alla prima chiamata (ovvero la prima volta, quando chiamiamo noi la funzione): non è definita (almeno non così) la sequenza di Fibonacci per numeri negativi, quindi il programma restituisce come risultato il numero di ingresso, lasciando alla funzione chiamante verificare la validità del risultato ottenuto. Vi invito a modificare la funzione appena scritta oppure a creare la funzione chiamante (tipo un "wrapper" della funzione *Fib*), verificando la validità del numero.
- Notate come alla riga 12 venga ripristinato il valore di  $n$  dallo stack. Potevamo ripristinare il suo valore solamente qui? No. Dipende come il programmatore ha deciso di seguire la perlustrazione dell'albero di Fibonacci. Discorso inverso va invece fatto per il LR, in quanto dev'essere ripristinato quando siamo sicuri di non dover chiamare più alcuna funzione.

Lascio a voi come esercizio scrivere il codice assembly dell' $n$ -esimo numero di fibonacci ma utilizzando un array per memorizzare i valori già calcolati. Questo è il codice C per la versione ricorsiva. Vi invito a scrivere il codice C (e Assembly) anche per la versione iterativa.

```

1 long long fibArr[n];
2 fibArr[0]=0;
3 fibArr[1]=1;
4 for(int i=2; i<=n; i++)
5     fibArr[i]=-1;
6
7 long long fib(int n, long long *fibArr){
8     if(fibArr[n]<0)
9         fibArr[n] = fib(n-1, fibArr) + fib(n-2, fibArr);
10    return fibArr[n];
11 }

```

## 8.4: Fibonacci con memoria

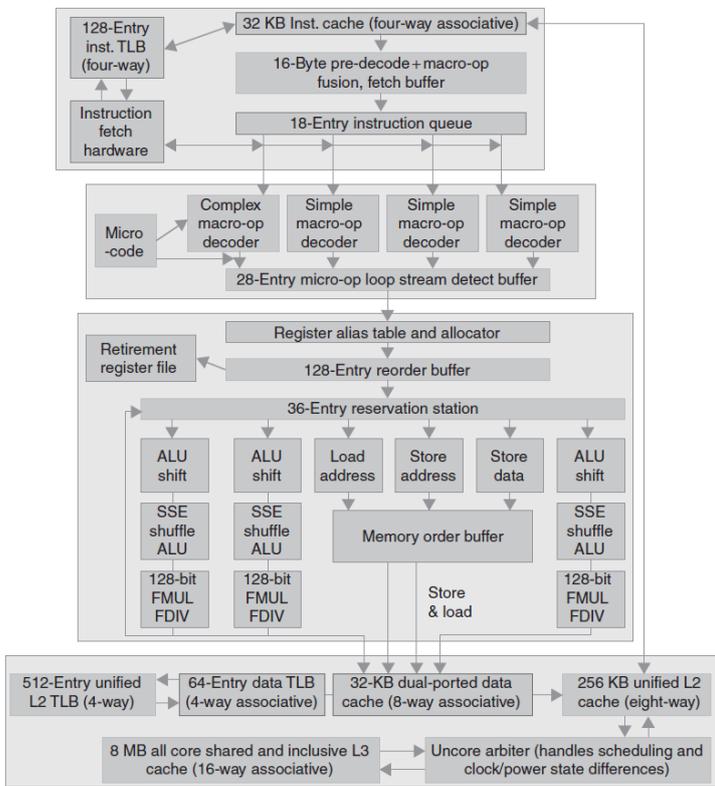
NB: le prime cinque righe servono a creare l'array ed inizializzarlo (con i due valori base e tutti gli altri a  $-1$ ). Poi c'è la ricorsione vera e propria. Attenzione a dove mettete le etichette per i salti! Non serve dichiarare ogni volta l'array *fibArr[n]*.

# 9 Approfondimenti

## 9.1 ALU

Più volte nella guida ho detto *l'ALU*, dando l'idea che nei processori ci sia una sola unità in grado di svolgere qualsiasi tipo di calcolo. Così non è, infatti i processori moderni (basati su qualsiasi architettura) dispongono di unità di calcolo specializzate per il calcolo vettoriale, matriciale (o, più in generale, tensoriale), di tracciamento dei raggi di luce, codifica/decodifica video e audio, per interfacciarsi più velocemente con le periferiche (I/O, dGPU, RAM, Thunderbolt o altre espansioni PCIe, USB o SATA/NVME). Poi alcuni processori dedicati a certi lavori (server, workstation) dispongono di unità per effettuare controlli di sicurezza, per la scalabilità dei processori, ovvero per consentire la comunicazione tra processori differenti e la condivisione della RAM e della cache di ultimo livello e molte altre.

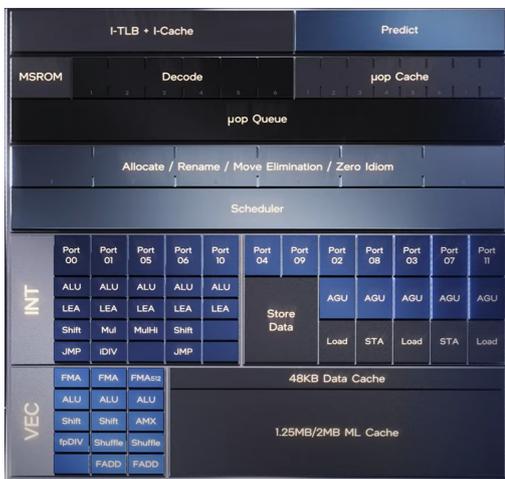
Non serve però essere dei processori da server per poter supportare operazioni in parallelo. È possibile, infatti, avere un processore composto da più core, ovvero più unità di calcolo indipendenti. Inoltre, questi core possono a loro volta contenere più unità di calcolo per eseguire una stessa funzione, consentendo quindi di raccogliere, decodificare eseguire e memorizzare più operazioni contemporaneamente. Questi processori vengono detti superscalari e se dispongono di unità di buffer, detto reorder buffer, per eseguire l'operazione di register renaming e quindi consentire l'esecuzione out-of-order, allora il processore è anche dotato di una pipeline dinamica.



Pipeline di un processore Intel Core i7 970

Struttura di un processore Intel Core moderno

Tabella 4: Notate come esistano notevoli unità di calcolo all'interno di un core dei processori e come nel corso degli anni si sia manifestata la necessità di aumentare il numero di unità che svolgono la stessa funzione, di aumentare la capacità dei vari livelli di cache e le dimensioni dei buffer includere ulteriori unità logiche, nonché di includere nuove unità specializzate, come quelle per i calcoli matriciali (AMX).



Core performante di un processore Intel di dodicesima generazione (blu)

Core efficiente di un processore Intel di dodicesima generazione (azzurro)

Schematico di un processore Intel di dodicesima generazione

Tabella 5: Per aumentare le prestazioni dei processori moderni senza far fondere il silicio a causa delle elevate temperature i produttori di processori stanno adottando nuove tecniche costruttive: inseriscono più core nei processori, ma alcuni di questi sono più prestanti - quindi dedicati ai carichi di lavoro pesanti, quali rendering, gaming, AI/ML, . . . -, mentre gli altri sono più efficienti - quindi dedicati ai lavori in background, quali aggiornamenti, scansione dell'antivirus, funzionalità vitali del sistema operativo, . . .-; viste le dimensioni sempre maggiori delle singole unità di calcolo è più conveniente produrle separatamente e poi unirle in fase di packaging, quindi le aziende stanno implementando nuovi ponti di interconnessione dei chip; avere più cache è essenziale per certe tipologie di lavori, quindi i costruttori stanno implementando le 3D V-Cache, ovvero verticalmente sopra al die viene collocata un piano di memoria cache di ultimo livello. Per approfondire ulteriormente l'argomento vi invito a [guardare la presentazione](#) o a [sfogliare le slide](#) di Intel Architecture Day 2021.

## 9.2 Lock

Quando due programmi (o lo stesso programma che lavora in parallelo su più unità di calcolo) necessitano di leggere o scrivere su una stessa locazione della memoria si possono verificare dei conflitti di dati, ovvero nel tempo che trascorre tra la richiesta di lettura e la lettura effettiva può avvenire una modifica da parte di un'altra unità di calcolo. Questo è da evitare a tutti i costi, quindi bisogna implementare un controllo ulteriore su questi registri per impedire che due unità di calcolo possano accedere allo stesso registro contemporaneamente.

Introduciamo quindi due nuove operazioni che ci consentono di lavorare con queste locazioni di memoria condivise: LDXR (LoaD from eXclusive memory to Register) e STXR (STore to eXclusive memory from Register). La locazione di memoria sarà quindi libera se è 0, mentre sarà già occupata se è diversa da 0. Vediamo con un esempio come funziona il lock.

```
1      ADDI X11, XZR, #1      (1)
2 again: LDXR X10, [X25, #0] (2)
3      CBNZ X10, again      (3)
4      STXR X11, X9, [X25] (4)
5      CBNZ X9, again      (5)
6
7      /* ... */          (6)
8
9 unlock: STUR XZR, [X25, #0] (7)
```

L'operazione di lock, sostanzialmente, si occupa di segnalare in maniera opportuna il fatto che allo stato attuale il processore (ossia una qualche routine in esecuzione) sta lavorando su dei dati e pertanto deve averne accesso esclusivo. In altre parole, fare un lock significa attivare una sorta di flag che indichi una condizione di occupato cosicché altri processi concorrenti non vadano ad intaccare i dati. Convenzionalmente, un flag a 1 indica un lock attivo mentre lo 0 segnala che la locazione di memoria è libera. Analizziamo il codice soprastante, riga per riga:

1. Salviamo semplicemente il valore 1 in un registro, perché tornerà utile di seguito;
2. Carichiamo il valore dell'indirizzo di memoria puntato dal registro X25 (in questo caso). In X10 verrà caricato il valore contenuto nella locazione di memoria condivisa;
3. Verifichiamo il valore letto: se il valore è diverso da 0 significa che il lock è attivo (sarà stato attivato da qualche altro processo/unità...), quindi dobbiamo rimanere in attesa;
4. Se non è attivo un blocco, salviamo il valore 1 precedentemente predisposto per impostare un nuovo lock, segnalando quindi che vogliamo avere accesso esclusivo a quella locazione di memoria condivisa.
5. Se il registro contenente l'esito dello Store-Esclusivo (X9 nel nostro caso) ha valore diverso da 0, significa che la scrittura è fallita perché il valore in memoria a X25 è stato cambiato da qualche altro processo/unità/...: dobbiamo quindi ricominciare la procedura di verifica;
6. Se siamo giunti qui, il blocco è stato correttamente impostato e abbiamo accesso esclusivo alla locazione di memoria di nostro interesse. Possiamo quindi eseguire tutte le operazioni

richieste e quando dobbiamo utilizzare la locazione di memoria condivisa possiamo leggere e scrivere normalmente il suo valore con le operazioni *non* esclusive;

7. Per eliminare il blocco, è sufficiente aggiornare il valore nella locazione di memoria condivisa con una semplice scrittura del valore 0.

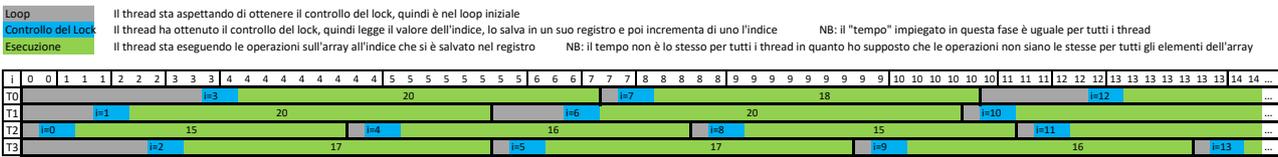
Approfondiamo meglio cosa si può fare una volta che abbiamo ottenuto l'accesso esclusivo alla locazione di memoria puntata da X25.

Possiamo sicuramente usarla per memorizzare dei risultati delle operazioni, ma se questo risultato fosse 0 perderemmo l'accesso esclusivo alla locazione di memoria senza rendercene conto! Quello che si può fare, ad esempio, è sfruttare questa locazione di memoria condivisa per verificare l'accesso ma leggere e scrivere un'altra locazione di memoria solo quando abbiamo ottenuto l'accesso esclusivo della precedente. Se estendiamo questo ragionamento a tutti gli altri processi/unità, allora abbiamo realizzato un esempio di sistema di sincronismo.

Vediamo un esempio pratico. Supponiamo di voler aggiungere una costante ad ogni elemento di un array molto grande, o comunque di voler fare la stessa operazione su tutti questi elementi. Potremmo far fare questo lavoro ad un unico processo, ma impiegherebbe tanto tempo. Possiamo applicare la tecnica del loop-unrolling, ma se le operazioni sono più complesse di qualche addizione e sottrazione questo metodo si rivela essere poco efficace. Un'altra opzione è spartire il lavoro su più processi (threads), che possono lavorare separatamente in maniera asincrona, ma vogliamo comunque fare in modo che questi non lavorino sullo stesso elemento contemporaneamente, o svolgano un'operazione che è già stata svolta da un altro thread. Ecco che torna utile usare una locazione di memoria condivisa per gestire l'indicizzazione dell'array: ogni thread utilizzerà una locazione di memoria (ad esempio quella puntata da X25) per ottenere l'accesso all'indice dell'array (contenuto in un'altra locazione, ad esempio X26); inoltre, imponiamo nella stesura del codice che un programma non possa né leggere né scrivere sulla seconda locazione di memoria almeno che non abbia ottenuto il controllo della prima. Una volta ottenuto il controllo si legge l'indice contenuto nella seconda locazione di memoria e si incrementa il suo valore di uno, cedendo poi il controllo della prima locazione di memoria<sup>20</sup>.

---

<sup>20</sup>Facciamo un esempio con quattro threads (processi). In qualche modo (non ci interessa come in questo momento) sono stati creati questi quattro thread - che chiameremo T0, T1, T2 e T3 - e viene inizializzato a 0 l'indice dell'array (la locazione di memoria puntata da X26). Questi quattro processi eseguono lo stesso identico programma. Inizialmente competono tutti e quattro per ottenere l'accesso alla locazione di memoria condivisa (X25) e uno di questi lo ottiene, supponiamo T2. T2 legge il valore dell'indirizzo puntato da X26, quindi 0, se lo salva in un registro interno al core e incrementa di 1 l'indice dell'array, quindi nella locazione di memoria puntata da X26 ora troveremo 1. Infine cede il controllo della locazione di memoria condivisa. Mentre T2 fa questi passaggi, gli altri tre thread sono rimasti all'interno del loop per ottenere l'accesso, fino a quando T2 non rilascia il controllo. Supponiamo che ora T1 riesce ad ottenere il controllo: farà gli stessi passaggi che ha fatto precedentemente T2, mentre T2 sta continuando l'esecuzione del programma sull'elemento dell'array in posizione 0 e T0 e T3 stanno ancora aspettando di avere il controllo. Quando T1 sblocca la memoria condivisa, supponiamo che T3 riesca ad ottenere l'accesso: leggerà che l'indice è 2, fa gli stessi passaggi dei precedenti thread e cede il controllo per poi iniziare l'esecuzione. Adesso abbiamo quindi T2, T1 e T3 che stanno eseguendo le operazioni sull'array rispettivamente sul primo, secondo e terzo elementi dell'array (supponiamo che ci siano tante operazioni, quindi che l'esecuzione richieda molti cicli) e T0 che sta aspettando di ottenere l'accesso. È l'unico a competere per l'accesso quindi lo ottiene sicuramente, legge l'indice 3, si salva 3 in un registro, memorizza 4 nell'indice e cede il controllo di X25. Ora tutti e quattro i thread sono al lavoro, ognuno su un



Un programma così costruito può essere eseguito su un numero arbitrario di thread senza che questi entrino in conflitto tra loro: l'unica locazione di memoria per cui "competono" è quella che abbiamo predisposto noi per essere condivisa (ovvero quella puntata da X25).

### 9.3 Approccio alla compilazione su carta

Fino ad adesso abbiamo visto molta teoria con qualche esempio pratico, ma non abbiamo definito un metodo per approcciare la "compilazione manuale". Ne esistono molti, vi riporto quello che utilizzo io.

1. Innanzitutto bisogna leggere bene il codice C per farci un'idea su cosa faccia il codice ed individuare subito alcuni punti critici su cui poi presteremo maggiore attenzione
2. Poi si guarda l'input e l'output della funzione principale (le funzioni chiamate da altre funzioni le analizzeremo dopo) e ci si segna in quali registri saranno gli input ed output. I puntatori stanno tutti nei registri X, le variabili intere nei registri X, le variabili a virgola mobile nei registri D, sempre dal registro più basso a quello più alto.
3. Poi si analizza il corpo della funzione. Per ogni istruzione devo controllare:
  - quante istruzioni di Assembly devo usare per eseguire l'operazione in C?
  - quali registri utilizzerà quest'operazione? Ovvero: sto lavorando con numeri interi, numeri a virgola mobile o puntatori?
  - ho tutti i dati disponibili nei registri giusti? Se si posso continuare, se no devo scaricarli dalla memoria o dallo stack
  - Il risultato dove lo metto? In un registro nuovo? Sovrascrivo uno già esistente? Quale?<sup>21</sup>

Poi bisogna fare distinzione tra i vari tipi di istruzioni C:

**if** Ho selezionato opportunamente la condizione di verifica? (ovvero: ho "invertito" la condizione di ingresso nell'if per saltare quando non si verifica?)

indice differente dell'array, ma eseguendo lo stesso identico codice! Una volta che un thread finisce l'esecuzione riprende dall'inizio tutti i passaggi, ma l'indice è stato modificato dagli altri thread, quindi l'esecuzione continua fino a quando non si raggiunge la fine dell'array.

Se volessimo aggiungere un altro thread possiamo farlo senza dover alterare il codice del nuovo o di quelli già in esecuzione.

<sup>21</sup>All'inizio potete mettere i risultati in registri nuovi ad ogni operazione. Molto presto, però, vi renderete conto che è necessario sovrascrivere i registri, in quanto sono molto pochi, quindi è buona abitudine iniziare il prima possibile ad imparare a vedere quali valori serviranno in futuro (quindi si guarda se la stessa variabile in C viene utilizzata nelle righe successive).

**Ciclo for/while** Ho impostato la condizione iniziale? Ho scelto opportunamente la condizione d'uscita? Ho aggiunto l'incremento del contatore prima di chiudere il loop? (ricordarsi di mettere la verifica della condizione d'uscita prima della prima esecuzione!)

**Ciclo do-while** Ho impostato la condizione iniziale? Ho scelto opportunamente la condizione d'uscita? Ho aggiunto l'incremento del contatore prima di chiudere il loop? (qui la verifica si fa alla fine delle operazioni)

**Chiamata a funzione** Ho disponibili tutti i valori che mi servono passare alla funzione come input? Sono nei registri giusti, quelli che la funzione si aspetta? Ho salvato nello stack il link register e tutte le variabili nei registri non preservati che mi serviranno dopo la chiamata dalla funzione? (ovvero: controllo se la variabile in C mi serve anche dopo la chiamata alla funzione, se sì e non l'ho salvata in un registro "auto-salvato" allora devo salvarla ora nello stack) La chiamata alla funzione è all'interno di un loop? (quindi devo stare attento a salvare il contatore del loop, altrimenti non ne uscirò mai) Ho ripristinato i registri una volta uscito dalla funzione?

4. Controlliamo ora che il codice appena scritto funzioni, provando ad eseguirlo in mente o, se è più complesso, facendo una tabella che contiene i valori principali ad ogni ciclo di clock.
5. Infine si può pensare a come ottimizzare il programma, ma questo passaggio è opzionale: si può provare a spostare le istruzioni mantenendo le precedenze delle operazioni ma evitando le bolle (quindi lasciando almeno 1 operazione tra una somma e l'altra e almeno 2 operazioni tra un caricamento e l'utilizzo della variabile caricata), ove possibile.

Questi punti chiave funzionano bene per programmi semplici, compresi quelli degli esami dei corsi di Architetture dei Sistemi Digitali. Per programmi più complessi che richiedono il riutilizzo dei registri bisogna prestare più attenzione.

Vediamo questo all'opera con un esempio pratico: *Chiamata a FIR1D*. Ecco il codice C:

```
1 double FIR1_D(double *h, double *X, long long N);
2
3 double LMS1(double d, double mu, double *h, double *X, long long N){
4
5     long long i;
6     double y, e, em;
7
8     y= FIR1_D(h, X, N);
9     e = d - y;
10    em = e * mu;
11    for(i = 0; i < N; i++)
12        h[i] += em * X[i];
13    return y;
14
15 }
```

### 9.3: Chiamata a FIR1D

Nella sezione [Soluzione esercizi](#) ho riportato una soluzione ottimizzata del codice che andiamo a creare ora.

Nel codice C sono riportate due funzioni: *FIR1\_D* e *LMS1*. Della prima non dobbiamo scrivere alcuna riga di codice: è una funzione già pronta (scritta precedentemente, o da qualcun altro, o presa da una libreria, indifferente) di cui ci interessa solamente cosa richiede in input e cosa lascia in output.

Partiamo quindi da *LMS1*. Ci aspettiamo in input (in ordine): un double (*d*), un double (*mu*), un puntatore a un double (*&h*), un puntatore a un double (*&X*) e un intero (*N*). In output invece forniamo un double (*y*). La prima nota del codice Assembly sarà quindi

```
1 /*
2 INPUT:  X:  X0 -> &h, X1 -> &X, X2 -> N
3         D:  D0 -> d,  D1 -> mu
4
5 OUTPUT:      D0 -> y
6 */
```

Molti dei codici successivamente presentati potrebbero sicuramente essere svolti in altri modi, ma gli input e output per qualsiasi compilazione manuale del corrente codice C devono essere sempre questi.

Poi vengono dichiarate delle variabili senza inizializzazione: queste linee non hanno una traduzione letterale in Assembly, ma dobbiamo tener presente il tipo di dato di ciascuna variabile, in quanto successivamente faremo uso di quest'informazione.

Ora c'è la chiamata alla funzione *FIR1\_D*, quindi dobbiamo salvare nello stack il link register e le variabili che vengono utilizzate dopo, ovvero *d*, *e*, *mu*, *&h*, *&X* (quindi nel nostro caso tutte, anche se non è sempre detto che sia così). Sono 6 registri X che dobbiamo salvare, quindi  $6 \cdot 8 = 48$  byte, che è un multiplo di 16. Dobbiamo anche posizionare le variabili richieste da *FIR1\_D* nell'ordine corretto (usando ad esempio la pseudoistruzione *MOV*): nel nostro caso sono già posizionate correttamente, quindi non è necessaria alcuna operazione prima del branch and link. Si può quindi chiamare la funzione *FIR1\_D*. Successivamente bisogna ripristinare tutte le variabili salvate prima della chiamata. Il Link Register lo si può ripristinare adesso o a fine programma, è indifferente. Basta che lo si ripristini dopo l'ultima chiamata ad una funzione (nel nostro codice c'è solo una chiamata, ma in altri potrebbero essercene di più oppure la chiamata potrebbe essere dentro un ciclo for: in questi casi conviene ripristinare il LR in fase di chiusura della procedura principale).

```
8 LMS1: SUBI SP, SP, #48
9       STURD D0, [SP, #0] // carico d
10      STURD D1, [SP, #8] // carico mu
11      STUR X0, [SP, #16] // carico &h
12      STUR X1, [SP, #24] // carico &X
13      STUR X2, [SP, #32] // carico N
14      STUR LR, [SP, #40] // carico LR
15
16      BL FIR1_D
17      // assumo che FIR1_D mi ritorni il risultato in D0
```

```

18
19     LDURD D1, [SP, #0] // scarico d
20     LDURD D2, [SP, #8] // scarico mu
21     LDUR X0, [SP, #16] // &h
22     LDUR X1, [SP, #24] // &X
23     LDUR X2, [SP, #32] // scarico N

```

NB: ho ripristinato  $d$  in D1 e  $mu$  in D2, non com'erano prima della chiamata, ovvero in D0 e D1 rispettivamente, in quanto D0 è già occupato da  $y$ , la variabile restituita da `FIR1_D`.

Ora posso continuare con l'istruzione successiva alla chiamata alla funzione. In un registro double salverò la differenza di due double. Ho tutte le variabili? Sì. Dove posso salvare il risultato?  $d$  non mi serve più, quindi posso sovrascriverlo. Oppure, visto che i registri double richiesti sono pochi, posso scrivere su un altro registro, in modo che poi mi sia più semplice riordinare le istruzioni per evitare le bolle. Analogo ragionamento si può fare con la successiva istruzione.

```

25     FSUBD D3, D1, D0 // e = d - y
26     FMULD D4, D3, D2 // em = e * mu

```

Ora passiamo al ciclo for. Dobbiamo inizializzare  $i$  a 0 (prima dobbiamo scegliere dove metterlo, e per comodità direi nel primo registro temporaneo disponibile, quindi in X9). Poi a me piace lavorare per puntatore in quanto si risparmia qualche istruzione e all'interno di un ciclo può far comodo. In questo caso è possibile in quanto  $i$  non è usato come variabile ma solo come indice del ciclo<sup>22</sup>. La condizione in C per rimanere nel ciclo è se  $i < N$ , che per noi diventerà un confronto tra X9 e X2, dove saltiamo (quindi usciamo dal ciclo) se  $X9 \geq X2$ . Lavorando per puntatore dobbiamo incrementare  $i$  del numero di byte corretto, quindi visto che  $h$  ed  $X$  sono array di double allora incrementiamo di 8 byte. Mi preparo, inoltre, anche  $N$  moltiplicato per 8, così non devo usare due registri per  $i$  e per  $i \cdot 8$  e posso sfruttare tutti i benefici della mia scelta di ciclare per indirizzo/puntatore.

```

28     MOV X9, XZR
29     LSL X2, X2, #3
30 for: CMP X9, X2
31     B.GE exit
32
33
34     // qui inseriamo il corpo del ciclo
35
36
37
38
39     ADDI X9, X9, #8
40     B for

```

Vediamo ora il corpo del ciclo. Quella singola riga di C corrisponde a molte righe di Assembly, in quanto stiamo leggendo e scrivendo da memoria (quindi dobbiamo scaricare e caricare valori nella memoria), nonché facendo due operazioni di moltiplicazione e di somma. In ordine, dobbiamo:

<sup>22</sup>Un esempio del contrario - ovvero un caso in cui è "necessario" ciclare per valore - è il programma [11.5: Autocorrelazione](#). Vi consiglio di svolgere questo esercizio in autonomia prima di guardare la soluzione o comunque un esercizio simile.

1. prepararci il puntatore  $\&X[i]$
2. leggere  $X[i]$
3. moltiplicare  $em \cdot X[i]$
4. preparare il puntatore  $\&h[i]$
5. leggere  $h[i]$
6. sommare  $h[i] + (em \cdot X[i])$
7. caricare in memoria il nuovo valore di  $h[i]$

```

32     ADD X11, X9, X1
33     LDURD D2, [X11, #0]
34     FMULD D2, D4, D2
35     ADD X10, X9, X0
36     LDURD D1, [X10, #0]
37     FADDD D1, D1, D2
38     STURD D1, [X10, #0]

```

Se non ci fosse quel "+=" ma solo un "=" - ovvero se non dovessimo aggiungere al corrente valore di  $h[i]$  quello appena calcolato - allora non servirebbero i passaggi 5 e 6.

Ora manca solo il *return*, quindi mettere  $y$  in D0, ma c'è già (ho scelto di non sovrascrivere il registro D0, quindi ho preservato il valore di ritorno della chiamata a *FIR1\_D*, ovvero  $y$ ); ci manca soltanto da ripristinare il link register, lo stack pointer e ritornare il controllo alla funzione precedente.

```

42 exit: LDUR LR, [SP, #40]
43       ADDI SP, SP, #48
44       BR LR

```

E il programma è completo. Ecco il testo intero:

```

1 /*
2 INPUT: X: X0 -> &h, X1 -> &X, X2 -> N
3 D: D0 -> d, D1 -> mu
4 X9 -> i
5 OUTPUT: D0 -> y
6 */
7
8 LMS1: SUBI SP, SP, #48
9     STURD D0, [SP, #0] // carico d
10    STURD D1, [SP, #8] // carico mu
11    STUR X0, [SP, #16] // carico &h
12    STUR X1, [SP, #24] // carico &X
13    STUR X2, [SP, #32] // carico N
14    STUR LR, [SP, #40] // carico LR
15
16    BL FIR1_D
17    // assumo che FIR1_D mi ritorni il risultato in D0
18
19    LDURD D1, [SP, #0] // scarico d
20    LDURD D2, [SP, #8] // scarico mu
21    LDUR X0, [SP, #16] // &h
22    LDUR X1, [SP, #24] // &X
23    LDUR X2, [SP, #32] // scarico N
24
25    FSUBD D3, D1, D0 // e = d - y
26    FMULD D4, D3, D2 // em = e * mu
27
28    MOV X9, XZR
29    LSL X2, X2, #3
30 for: CMP X9, X2
31     B.GE exit
32     ADD X11, X9, X1
33     LDURD D2, [X11, #0]
34     FMULD D2, D4, D2
35     ADD X10, X9, X0
36     LDURD D1, [X10, #0]
37     FADDD D1, D1, D2
38     STURD D1, [X10, #0]
39     ADDI X9, X9, #8
40     B for
41
42 exit: LDUR LR, [SP, #40]
43     ADDI SP SP, #48
44     BR LR

```

### 9.3.1: Chiamata a FIR1D

## 9.4 Dove continuare

Voglio dedicare queste ultime sezioni agli approfondimenti, per soddisfare anche coloro che vogliono sfruttare le conoscenze del corso di Architetture dei Sistemi Digitali come piattaforma di lancio per esplorare il mondo dell'elettronica digitale.

Incomincio citandovi [Ben Eater](#), che con le sue serie su come [costruire un processore a 8 bit da zero](#) e [come costruire un computer con un 6502](#) mi ha ispirato molto ad intraprendere questo corso di studi. Sul suo sito e sul suo canale YouTube potete trovare contenuti interessanti anche per altri corsi, come per Reti di Telecomunicazioni, nonché vari spunti per progetti personali. Se voleste approfondire l'architettura di alcuni processori vi posso suggerire di sfogliare i link presentati nella sezione [Introduzione all'architettura](#). Può tornar utile saper leggere questi datasheet o quelli dei microcontrollori, come quello dell'[Arduino Uno Rev3](#) (o meglio, del suo microcontrollore, l'[ATMega328P](#)) o di un [ESP32](#).

Nel corso di Elettronica vedrete molti datasheet, sia dei singoli componenti elettronici, sia di schede intere.

Un giornale online di elettronica che può fornire ulteriori idee è [AllAboutCircuits](#). Assieme alla rivista [Spectrum](#) della IEEE<sup>23</sup> e al blog [EEVblog](#) costituiscono le mie fonti principali di informazione in tutti gli ambiti dell'Elettronica.

Nel corso di Architetture dei Sistemi Digitali abbiamo parlato di processori. E se invece volessimo sapere come funziona l'architettura delle schede grafiche? O di qualche altro dispositivo? Purtroppo non è possibile reperire molte informazioni online, in quanto la maggior parte del software è proprietario. Un esempio di "scheda grafica" fatta in casa è quella fatta da Ben Eater<sup>24</sup>.

Interessanti sono anche le documentazioni di [AMD](#), [Intel](#) e [NVidia](#), i tre principali produttori di schede grafiche desktop: forniscono informazioni generiche sull'architettura (unità presenti, capienza delle memorie e la loro disposizione nel chip, dimensioni dei bus, ...). Ulteriori informazioni si possono trovare su [OpenCL](#), uno standard basato sul linguaggio C/C++ per la programmazione parallela su molteplici architetture. Se foste interessati a questi argomenti vi posso suggerire come corso introduttivo [Programmazione Avanzata e Parallela](#), preso dal terzo anno di Intelligenza Artificiale, e per coloro che vogliono approfondire ulteriormente l'argomento (magari alla magistrale) vi suggerisco i due corsi [Foundations of High Performance Computing](#) e [Parallel Programming for HPC](#) dalla magistrale di DSSC (Data Science and Scientific Computing).

Per chi fosse più interessato al lato software ho trovato estremamente utile [questa guida](#) su alcune ottimizzazioni possibili per i codici C. Assieme a [questa guida](#) sulla programmazione

---

<sup>23</sup>Ve ne parlerà meglio il prof. Carrato nel corso di Elettronica, vi anticipo solo che grazie all'Università abbiamo l'accesso completo a questa rivista

<sup>24</sup>Ecco i video: [parte 1](#), [parte 2](#), [parte 3](#), [parte 4](#)

dinamica può migliorare notevolmente le proprie abilità di ottimizzazione di scrittura dei codici.

Oltre alla già nominata playlist della Intel sull'[architettura dei processori](#) ho trovato estremamente interessante la [seconda playlist](#) su varie tecnologie di punta dell'azienda di Santa Clara. Ho scoperto l'esistenza di varie branche dell'elettronica e dello sviluppo software, come ad esempio i [computazione neuromorfa](#), dei chip in grado di allenare ed utilizzare in maniera estremamente più efficace delle reti neurali. Lo sviluppo parte dall'hardware, con il chip Loihi, per passare al firmware Magma e arrivare alle librerie Lava; il tutto viene poi integrato nella [OneAPI](#), un'API di Intel per semplificare lo sviluppo di codice su più piattaforme.

## 10 ARMv8 su RaspberryPi

Esistono linguaggi di programmazione che generano file eseguibili su qualsiasi piattaforma (vedi Python e Java [quest'ultima fa uso della Java Virtual Machine installata sui pc]), altri che sono dipendenti dalla piattaforma, ovvero che il codice può essere eseguito sul dispositivo corrente e pochi altri che utilizzano la stessa architettura (set di istruzioni), lo stesso sistema operativo (come ad esempio C/C++). Assembly è una categoria assestante. È un livello ancora sottostante, ovvero può essere eseguito solamente su un microprocessore specifico (ricordo che Fortran è stato ideato proprio per ovviare a questo problema, ovvero che tutti i programmatori che lavoravano sullo stesso progetto dovevano essere dotati dello stesso "microprocessore", o meglio, della stessa macchina in grado di leggere ed interpretare i nastri perforati). Vi riporto per completezza [Una classificazione a sette livelli di astrazione nel mondo digitale](#) (probabilmente la riconoscerete, è la tabella che presenta il professor Fabris al corso di Fondamenti di Informatica). Noi stiamo quindi lavorando al livello 4.

Livello		
-1	Elettronico	Siamo al livello dei transistor che formano i circuiti elettronici che costituiscono le porte logiche di cui è composto un calcolatore. A questo livello il singolo transistor è in conduzione o in interdizione, e si raggiunge un livello di dettaglio che viene in genere trascurato nella progettazione dei calcolatori.
0	Logico	Questo è il primo livello utile nella progettazione di un computer. La macchina è formata da porte logiche o gate. Ogni porta riceve in ingresso dei segnali binari e calcola una semplice funzione Booleana (AND, OR, ...). Collegando opportunamente le porte di base si ottengono relazioni logiche complesse per i circuiti costituenti, p.es. si può realizzare una memoria di un bit (bistabile). Combinando n memorie di un bit si può formare un registro capace di memorizzare un numero binario compreso tra 0 e $2^n - 1$ . Mediante le porte si realizzano i circuiti logici il cui funzionamento è regolato dalle leggi dell'algebra Booleana e dell'elettronica digitale. La macchina logica del livello 0 viene progettata dal costruttore dei vari componenti ed è puramente hardware.
1	Micro-architettura	A questo livello troviamo tutti gli elementi nell'architettura di base del computer, e cioè i registri generali usati come memoria locale, la ALU che esegue semplici operazioni logico-aritmetiche, gli elementi di connessione tra registri e ALU, i registri dedicati al controllo (PC, RI, ...) e la circuiteria dell'Unità di Controllo. Il percorso dei dati può essere gestito da un programma controllabile dall'esterno, chiamato microprogramma (come nel caso dell'architettura CISC), oppure da una specifica circuiteria (come nel caso dell'architettura RISC).
2	ISA	È costituito dall'Instruction Set Architecture ed è quindi governato dal linguaggio macchina. Offre visione ed accesso diretto a tutte le risorse fisiche del sistema, tramite una specifica interfaccia di livello. Fornisce un'interfaccia tipicamente software (e quindi programmabile) ai livelli superiori. Si può agire al livello 1 tramite interpretazione (microprogramma) o disporre di un'esecuzione diretta a livello 0.
3	Sistema Operativo	È un'estensione del livello ISA ottenuta aggiungendo servizi che vengono eseguiti (interpretati) da un programma del livello ISA chiamato appunto Sistema Operativo. Esso garantisce l'operatività di base del calcolatore, coordinando e gestendo le risorse hardware di processamento e di memorizzazione, le periferiche, le risorse e le attività software legate ai vari processi in uso, funge da interfaccia con l'utente, consentendo l'impiego di altri software d'utente, come le varie applicazioni o le librerie.
4	Assembler	A questo livello si fornisce una rappresentazione simbolica di uno dei livelli sottostanti, impiegando sequenze alfanumeriche pensate per essere mnemoniche e comprensibili. Infatti i linguaggi binari dei livelli più bassi sono difficili (o praticamente impossibili) da usare per un programmatore. Il programma che traduce da programmi in linguaggio assembler a programmi a livello ISA è detto Assembler o assemblatore. A ogni istruzione del linguaggio assembler corrisponde un'istruzione del linguaggio macchina che viene eseguita direttamente. I programmi a questo livello non sono usati dal programmatore medio, che deve realizzare programmi applicativi, ma solo dai programmatori di sistema.
5	Linguaggi applicativi	È caratterizzato dall'impiego di linguaggi come C, C++, Java, BASIC, LISP, Prolog, ..., chiamati per l'appunto linguaggi di alto livello. Sono impiegati per la realizzazione di programmi applicativi. Il più delle volte la traduzione è affidata a un compilatore, mentre in alcuni casi si usa un interprete.

Tabella 6: Una classificazione a sette livelli di astrazione nel mondo digitale

Bando alle ciancie, vediamo come si può programmare praticamente in Assembly.

## 10.1 Introduzione all'architettura

La maggior parte di noi avrà un computer Windows, quindi sicuramente avrete un sistema basato sull'architettura x86. Per coloro di voi che utilizzano un MacBook, invece, c'è una possibilità che se è sufficientemente recente (2020 in poi) abbia un processore della serie M (M1/M2 con tutte le loro varianti), che sono processori basati su ARMv8.5-A, la quinta revisione dell'architettura ARMv8 per i core performanti. In quest'ultimo caso si può sfruttare direttamente il vostro computer per programmare in ARMv8. Apple installa di default sui dispositivi MacOS il suo compilatore proprietario *Clang*, che è basato su LLVM ma ha una sintassi identica a quella di GCC. Sebbene non venga trattato in questa guida, è possibile utilizzarlo, anche se vi consiglio di cercare online qualche guida per installare GCC. Un esempio potrebbe essere [questa](#), che fa uso di brew, o alternativamente potete scaricare XCode di Apple dall'AppStore con le relative estensioni per GCC.

Per tutti gli altri esistono varie schede (anche non troppo costose e che possono girare per le vostre case, potenzialmente) basate sull'architettura ARMv8, un esempio sono i RaspberryPi dal 3 in poi. Esistono diverse revisioni anche a posteriori delle generazioni precedenti dei RP, quindi vi invito a controllare sul [sito del produttore](#) l'architettura nel caso in cui aveste una versione precedente alla terza.

NB: nel processo di installazione è necessario selezionare la versione del sistema operativo a 64 bit, altrimenti non è possibile sfruttare i registri a 64 bit e nel caso in cui aveste un RP4 con 8 GB di RAM non tutte le applicazioni sarebbero in grado di sfruttarla al massimo. Vi suggerisco [questo articolo](#) della stessa ARM che parla dei vantaggi dell'architettura a 64 bit rispetto a quella a 32 bit.

Il LEGv8 è un'architettura inventata su carta (nel senso che non esiste alcun processore reale basato su LEGv8) allo solo scopo didattico. I due autori, Patterson ed Hennessy, hanno voluto semplificare ulteriormente l'architettura ARMv8 per poter insegnare le basi dell'architettura dei calcolatori agli studenti. Perché farlo con un'architettura RISC (come l'ARMv8) e non con una CISC (come l'x86, estremamente diffusa tra i computer degli studenti, sicuramente all'epoca della scrittura del libro)? Perché la complessità dell'architettura x86 è nettamente superiore a quella di qualsiasi architettura RISC<sup>25</sup>. Vi invito a consultare la [documentazione dell'implementazione della Intel dell'architettura x86](#), nonché la [documentazione della ARM per l'architettura ARMv8 e ARMv9](#) (pubblicata a fine 2021). Confrontatele poi con quanto abbiamo incontrato nel libro di testo di questo corso. Penso così possiate comprendere che è necessario partire da qualcosa di più semplice in un corso base del secondo anno di Ingegneria Elettronica ed Informatica, per poi eventualmente approfondire alla magistrale o con un

---

<sup>25</sup>In realtà gli stessi autori hanno scritto anche una guida sul MIPS, sull'interfaccia hardware-software, nonché più di recente una revisione del libro per l'architettura [RISC-V](#)

dottorato un'architettura nello specifico (per i più interessati). Se però voleste approfondire la programmazione Assembly su un'architettura x64 vi suggerisco [questo corso](#).

Vediamo ora come possiamo verificare i codici scritti in precedenza direttamente su un RaspberryPi, quindi compilarli ed eseguirli su una macchina vera.

Non vedremo tutti i comandi dell'ARMv8 e nemmeno tutte le funzionalità. Per questi potete consultare il [manuale dell'ARM Cortex A53](#), uno dei core del processore del RaspberryPi (che da ora in avanti chiamerò per brevità RP), o di qualche altro processore ARM più moderno. L'obiettivo di questo capitolo è quello di dare la possibilità a coloro che possiedono un calcolatore con processore ARMv8 di testare i codici dell'architettura LEGv8, quindi effettivamente limitandoci a completare i codici già scritti in LEGv8 con il resto del file Assembly. Infatti fino ad adesso abbiamo scritto solamente il codice che corrisponde al segmento di testo del file Assembly. Dobbiamo in più capire come dire all'assemblatore quale sia la funzione principale (il *main* del C), come dichiarare le variabili e gli array statici ed come chiamare le eventuali librerie esterne. Non possiamo però trascurare come utilizzare il compilatore. Visto che utilizzerò una piattaforma Linux (RaspberryPi OS, ex Raspbian) il compilatore GCC è già installato. Possiamo verificare la sua versione (o la sua corretta installazione) scrivendo nella linea di comando

```
1 gcc --version
```

Sono pochi i comandi del compilatore che ci serviranno, ma per completezza riporto alcuni flag principali nel caso in cui voleste sperimentare.

Opzione	Descrizione	Esempio
<code>-o NomeOut</code>	Specifica il nome del file di output	<code>gcc -o main main.c</code>
<code>-S</code>	Esegui solo il processo di compilazione, quindi crea solo il file assembly	<code>gcc -S main.c</code>
<code>-v</code>	Manda a schermo i risultati dei vari processi del compilatore	<code>gcc -v main.c</code>
<code>-Olvl</code>	Specifica il livello di ottimizzazione, dove <i>lvl</i> è un numero da 0 a 3, di default a 0.	<code>gcc -O3 main.c</code>
<code>-g</code>	Il compilatore aggiunge ulteriori "commenti" al codice, utili in fase di debug per tener traccia delle variabili.	<code>gcc -g main.c</code>

Ottimo. Scriviamo quindi il primo programma sul RP: creiamo un file *test.s* e digitiamo le seguenti righe, ricordandoci sempre di lasciare una riga vuota alla fine del codice:

```
1 .global main
2
3 main:
4     MOV X0, #2
5     BR LR
6
```

La prima riga dice al compilatore che l'istruzione etichettata *main*, quindi *MOV X0, #2*, dev'essere la prima istruzione ad essere eseguita.

Compiliamo ed eseguiamo il codice con i seguenti due comandi

```
1 gcc test.s
2 ./test
```

Il nostro programma non ha output, quindi non visualizziamo nulla. Possiamo però scoprire qual'è il codice di uscita del nostro programma (il codice che in C settiamo sempre a 0 - ovvero programma finito con esito positivo - mediante il comando *return 0*) alla fine del *main*. Per farlo basta concatenare all'ultimo comando anche *echo \$?*, quindi

```
1 ./test ; echo $?
```

Questo ci mostra il contenuto del registro *X0*, quindi 2.

## 10.2 Alcuni codici in ARMv8

Incominciamo a fare qualche programma un po' più interessante: scriviamo ora il codice per eseguire la **somma misto mare** del capitolo 2.

```
1 .global main
2
3 main:
4     MOV X0, #2      // inizializzo a
5     MOV X1, #5      // inizializzo b
6
7     MOV X13, #3     // preparo 3
8     MOV X14, #4     // preparo 4
9
10    MUL X0, X0, X13 // 3*a
11    MUL X1, X1, X14 // 4*b
12    SDIV X9, X1, X13 // 4b/3
13    MUL X9, X9, X13 // (int(4b/3))*3
14    SUB X1, X1, X9  // 4b%3
15    ADD X0, X0, X1  // 3a+ ((4b)%3)
16    BR LR
17
```

Praticamente identico al codice che avevamo scritto in precedenza per il LEGv8.

È un po' noioso però scrivere ogni volta *echo ?* per visualizzare l'output. E se dovessimo visualizzare più risultati? Possiamo utilizzare la funzione *printf* già presente nella libreria *libc* di GCC. Basta dichiarare che stiamo utilizzando una libreria esterna, inserire nella sezione dei dati una stringa che manderemo in output e in fase di chiamata mettere in *X0* il puntatore alla stringa e in *X1* → *X7* i valori da stampare. Essendo questa una funzione esterna al codice principale dobbiamo ricordarci di salvare il link register.

Ecco un esempio di una **moltiplicazione semplice**.

```
1 // sezione dei dati
2 .data
3     res: .asciz "4*5=%d\n" // asciz e' uno dei modi per dichiarare una stringa
4
5 // sezione del codice
6 .text
7 .global main
8 .extern printf
9
10 main:
```

```

11  SUB SP, SP, #16 // alloco spazio nello stack
12  STUR LR, [SP]  // memorizzo il link register
13  MOV X1, #4     // preparo 4
14  MOV X9, #5     // preparo 5
15  MUL X1, X1, X9 // 4*5
16  LDR X0, =res   // metto in x0 il puntatore alla stringa res
17  BL printf      // printf("4*5=%d\n")
18
19  LDUR LR, [SP]  // ripristino il link register
20  ADD SP, SP, #16 // ripristino lo stack pointer
21  BR LR         // esco dalla procedura
22

```

Notate la separazione tra la sezione delle variabili statiche *data* e quella del codice, ovvero *text*. Importante è anche la riga 16, in quanto stiamo caricando l'indirizzo della variabile, in questo caso una stringa. Nel LEGv8 davamo per scontato di trovarci in un determinato registro già il puntatore, qui invece dobbiamo leggerlo dalla sezione dati.

Vediamo ora come si eseguono operazioni condizionali e chiamate a procedure con il programma **median3**, lo stesso che abbiamo incontrato nella sezione 2.2: *E se...*, ma con una chiamata a funzione, quindi scorporando lo swap in una funzione esterna.

```

1  .data
2      txt: .asciz "la mediana e' %d\n"
3
4  .text
5  .global main
6  .extern printf
7
8  swap:
9      LDUR X9, [X0]
10     LDUR X10, [X1]
11     STUR X9, [X1]
12     STUR X10, [X0]
13     BR LR
14
15
16  median3: // calcola la mediana di 3 numeri
17     SUB SP, SP, #32
18     STUR LR, [SP, #24]
19     STUR X0, [SP, #0]
20     STUR X1, [SP, #8]
21     STUR X2, [SP, #16]
22
23     CMP X0, X2
24     B.LE cond2
25     MOV X0, SP
26     ADD X1, SP, #16 // notate come non serve specificare ADDI in ARMv8
27     BL swap
28     LDUR X0, [SP]
29     LDUR X1, [SP, #8]
30     LDUR X2, [SP, #16]
31
32  cond2:
33     CMP X0, X1
34     B.LE cond3
35     MOV X0, SP
36     ADD X1, SP, #8
37     BL swap

```

```

38     LDUR X1, [SP, #8]
39
40 cond3:
41     CMP X1, X2
42     B.LE return
43     ADD X0, SP, #8
44     ADD X1, SP, #16
45     BL swap
46
47 return:
48     LDUR X0, [SP, #8]
49     LDUR LR, [SP, #24]
50     ADD SP, SP, #32
51     BR LR
52
53
54 main: // utilizzo il main come funzione di test
55     SUB SP, SP, #16
56     MOV X0, #4
57     MOV X1, #3
58     MOV X2, #7
59     STUR LR, [SP, #8]
60     BL median3
61     MOV X1, X0
62     LDR X0, =txt
63     BL printf
64     LDUR LR, [SP, #8]
65     ADD SP, SP, #16
66     BR LR
67

```

Scriviamo ora un programma che **somma due numeri a virgola mobile**

```

1 .data
2     res: .asciz "%f + %f = %f\n"
3     num1: .double 0.2
4     num2: .double 0.7
5
6 .text
7 .global main
8 .extern printf
9
10 main:
11     SUB SP, SP, #16
12     STUR LR, [SP]
13
14     LDR X9, =num1 // per caricare un'etichetta bisogna utilizzare il load normale
15     LDR X10, =num2
16     LDUR D0, [X9] // poi per leggere il valore si utilizza un load unscaled
17     LDUR D1, [X10]
18     FADD D2, D0, D1
19
20     LDR X0, =res
21     BL printf
22
23     LDUR LR, [SP]
24     ADD SP, SP, #16
25     BR LR
26

```

Notate che per i numeri double si opera come per le altre variabili nella memoria statica del programma: prima si carica in un registro l'indirizzo della variabile, poi si carica il valore ef-

fettivo della variabile. Ho voluto sottolineare questo in quanto nel LEGv8 non esiste un'unità hardware che possa convertire un numero intero in un float o viceversa, come abbiamo visto nell'[Approfondimento sul trasferimento bit a bit](#). Quindi se nel nostro programma necessitiamo di un numero a virgola mobile dobbiamo inizializzarlo. Nel LEGv8 il primo passaggio viene trascurato, in quanto supponiamo di avere già in uno dei registri di input (X0 → X7) l'indirizzo della locazione di memoria che contiene il numero richiesto.

Ora vediamo un programma esempio per l'utilizzo degli array: la **media dei valori di un array di double**.

```

1  .data
2      arr: .skip 32
3      uno: .double 1.0
4      mezzo: .double .5
5      str: .asciz "La media e' %lf\n"
6
7  .text
8  .global main
9  .extern printf
10
11 media:
12      LDR X15, =uno // carico il puntatore ad 1.0
13      LDUR D15, [X15] // carico 1.0
14      FSUB D0, D15, D15 // azzero A
15      FSUB D1, D15, D15 // azzero (double)i che mi servira' per la divisione finale
16      LSL X1, X1, #3 // n*8
17      ADD X1, X1, X0
18 loop: CMP X0, X1 // i<n
19      B.GE exit
20      LDUR D2, [X0] // *arr[i]
21      FADD D0, D0, D2 // A+= arr[i]
22      FADD D1, D1, D15 // ((double)i)++
23      ADD X0, X0, #8 // i++
24      B loop
25
26 exit: FDIV D0, D0, D1
27      BR LR
28
29 main: // main di test
30      SUB SP, SP, #16
31      STUR LR, [SP]
32
33      LDR X0, =arr
34      LDR X9, =uno
35      LDR X10, =mezzo
36      LDUR D0, [X9]
37      LDUR D1, [X10]
38
39      FADD D2, D0, D1
40      STUR D2, [X0]
41      STUR D1, [X0, #8]
42      STUR D0, [X0, #16]
43      STUR D1, [X0, #24]
44      // adesso avremo arr={1.5, 0.5, 1.0, 0.5}
45
46      MOV X1, #4
47      BL media
48      LDR X0, =str
49      BL printf

```

```

50
51     LDUR LR, [SP]
52     ADD SP, SP, #16
53     BR LR
54

```

## Infine convertiamo una stringa in maiuscolo

```

1  .data
2  str: .asciz "Ciao!"
3
4  .text
5  .globl main
6  .extern printf
7
8  strMaiusc:
9      LDR X10, =str
10 loop:
11     LDURB W9, [X10]
12     CBZ W9, [X10] exit
13     CMP W9, #97
14     B.LT finewhile
15     CMP W9, #122
16     B.GT finewhile
17     SUB W9, W9, #32
18     STURB W9, [X10]
19
20 finewhile:
21     ADD X10, X10, #1
22     B loop
23 exit: BR LR
24
25
26 main:
27     SUB SP, SP, #16
28     STUR LR, [SP]
29     BL strMaiusc
30     LDR X0, =str
31     BL printf
32     LDUR LR, [SP]
33     ADD SP, SP, #16
34     BR LR
35

```

Notate l'uso del registro *W9* invece di *X9*. Questo perché in ARMv8 byte, half word e word utilizzano i registri *W*, mentre le doubleword utilizzano i registri *X*. Per caricare e leggere dalla memoria i valori, invece, si utilizza *LDURB/STURB* per i byte, *LDURH/STURH* per le halfword e *LDUR/STUR* per word e doubleword. Se si dovesse caricare numeri con segno allora esistono le versioni con la *S* per ciascuno. Per ulteriori informazioni vi suggerisco di consultare il manuale dell'ARMv8.

Così abbiamo visto come possiamo verificare tutti i codici del LEGv8 con un processore ARMv8.

Questo non significa che abbiamo coperto tutte le funzioni dell'ARMv8 o che questi codici siano i più efficienti per l'ARMv8. Esistono infatti numerose istruzioni che non sono state implementate nella sua versione ridotta, il LEGv8, come ad esempio i [registri vettoriali](#), il [SIMD](#) (Single Input Multiple Data), tutte le operazioni con la memoria condivisa (per un

accenno vedi [Lock](#)), molte pseudoistruzioni che racchiudono più operazioni in un'unica riga o abilitano l'uso semplificato delle operazioni polinomiali. Per scoprire di più vi invito a consultare la [guida completa dell'ARMv8](#) (con le novità portate dall'ARMv9) oppure la guida ridotta [ArmInstructionSetOverview](#).

### 10.3 Ottimizziamo i codici: Branchless Programming, Loop Unrolling e SIMD

Abbiamo analizzato nel capitolo 7 i vari tipi di branch, ma non abbiamo nominato le problematiche relative al loro utilizzo.

I salti incondizionali sono i meno problematici, in quanto è possibile riordinare manualmente le istruzioni o, se il processore è dotato di una pipeline dinamica, il riordino viene effettuato nello scheduler hardware nel backend del processore<sup>26</sup>.

I salti con link (BL) e a registro (BR) possono essere problematici per processori poco prestanti, in quanto è necessario avere la routine nella cache L1 delle istruzioni<sup>27</sup>. Se così non fosse allora si deve andarla a recuperare nel livello superiore di cache oppure nella memoria principale nel caso in cui non dovesse essere disponibile nemmeno in quelli, quindi bisogna aspettare qualche centinaio, se non migliaio di cili di clock. Inoltre, è possibile che la procedura invocata sia una funzione di una libreria dinamica, quindi che dev'essere appena caricata (solo al primo utilizzo) nella memoria principale dal disco di archiviazione, quindi l'esecuzione del thread riprenderà appena tra qualche milione (SSD) se non miliardo (HDD) di cicli di clock.

I salti condizionali, infine, creano problemi nel caso in cui l'unità predittiva (branch predictor) ha sbagliato la predizione del salto: sono state caricate nella pipeline le istruzioni sbagliate, quindi è necessario svuotare tutta la pipeline fino alla fase di Execute, sostituendo tutte le

---

<sup>26</sup>È forse necessario nominare le differenze tra processore statico e dinamico e tra pipeline statica e dinamica, nelle loro quattro possibili combinazioni. Il processore può essere diviso in due macro parti: frontend e backend; ognuna è composta da due *fasi*: IF (Instruction Fetch) e ID (Instruction Decode), EX (EXecute) e WB (Write Back). Nel LEGv8 ci sono 5 *stadi*, infatti la quarta *fase* viene eseguita in due *stadi*: il MEM (MEMory) e il WB (Write Back). Processori più moderni, come quelli presentati nella figura 5, gli stadi per ogni fase sono molti di più, ma le fasi sono le stesse. Nel frontend, in fase di decodifica, viene eseguito il raggruppamento delle istruzioni e lo "spacchettamento" di quelle più complesse, generando in entrambi i casi le cosiddette *pops* (micro operazioni). Nel backend, più precisamente tra il Reorder Buffer e le Reservation Stations, c'è lo scheduler hardware del processore, che si occupa del riordino delle *pops*.

Un processore si dice dinamico se è dotato delle unità di raggruppamento e spacchettamento delle istruzioni in *pops*. Se dovessero essere assenti il processore viene detto statico ed è compito del compilatore fornire al processore direttamente le *pops*. Un processore ha pipeline dinamica se è dotata dello scheduler hardware. Nel caso contrario viene detto statico ed è compito del compilatore riordinare le istruzioni in modo da evitare eventuali stalli.

<sup>27</sup>Nella tabella 2 e nella pagina precedente abbiamo parlato dei registri e della RAM, soltanto accennando all'esistenza della cache. La seconda parte del corso di Architetture dei Sistemi Digitali approfondisce meglio quest'aspetto, quindi suggerisco di seguirla anche a chi non deve dare quella parte dell'esame. [Queste](#) sono le slide del prof. Medvet (io ho seguito le sue lezioni), mentre [qui](#) trovate quelle del prof. Trevisan. Aggiungo una piccola nota sui vari tipi di cache nei processori. Sulle specifiche dei processori viene indicata la quantità di cache L2 e "Last Level Cache" (=L3), mentre nei manuali dell'architettura si trovano, oltre ad informazioni più dettagliate sulle precedenti, anche informazioni sulla cache L1, divisa in L1 Data Cache e L1 Instruction Cache. Nell'immagine 5 si può trovare sia 48 kB di Data Cache, sia 1.25 o 2 MB di ML (Middle Level, = L2) cache. Nell'immagine più a destra invece si nota la LLC (Last Level Cache), la più grande, condivisa tra i vari core ed accessibile contemporaneamente da tutti (nella grafica viene rappresentata come se fosse "spezzettata" per rappresentare questa funzionalità dei nuovi processori x86).

operazioni con delle no-op (no operation, ovvero operazioni che non fanno nulla) e accodando dal prossimo ciclo le pops corrette nell'Instruction Fetch.

È possibile evitare queste problematiche? In parte sì. Negli ambiti dov'è necessario ridurre al minimo i consumi elettrici o dove non è possibile avere latenze elevate si utilizza tutta la compilazione statica (ovvero in fase di compilazione si aggiunge un flag per segnalare al linker di includere anche il codice delle funzioni dinamiche, non solo l'indirizzo per dove trovarle). Vi suggerisco caldamente di leggere [questa pagina della NASA](#) sulle linee guida da seguire per i codici di *F'* - il framework dell'ente americano per il "rapid development and deployment of embedded systems and spaceflight applications" -, nonché il [potere delle 10 regole](#) della JPL. Per i branch condizionali invece esiste un rimedio assai più efficace: la programmazione senza salti.

Ovviamente non è possibile eliminare tutti i salti condizionali, come ad esempio quelli per uscire dai cicli, ma è possibile convertire le istruzioni *if* in delle operazioni che fanno uso delle operazioni logiche per confrontare valori, ma che sfruttano il risultato binario dell'operazione. Tradotto in Assembly, esistono delle operazioni che leggono i flag attivati dall'ALU e scrivono "1" o "0" in un registro da noi specificato a seconda del valore del bit di flag. È più semplice spiegare questo con un esempio, quindi vediamo come si applica la programmazione senza salti al programma *Stringa maiuscola*.

Abbiamo incontrato una prima versione del codice *Stringa maiuscola* nel paragrafo 5.1. Si può rendere più efficiente questo codice eliminando i branch condizionali, quindi effettivamente togliendo la possibilità che si creino bolle all'interno della pipeline. Purtroppo in LEGv8 non è possibile lavorare con i flag (se non con i branch condizionali), quindi ho utilizzato un'istruzione dell'ARMv8 (CSET) che legge il bit del registro NZCV corrispondente al segno.

Il codice C diventa

```
1 void strMaiusc(char *arr){
2     while(*arr!=0){
3         *arr -= 32 * ((*arr >= 97) && (*arr <= 122));
4     }
5 }
```

### 10.2.1: Stringa maiuscola senza branch

mentre quello Assembly diventa

```

1 /*
2 X0 => *arr      W9 => temp
3 */
4
5 strMaiusc:
6     LDURB W9, [X0, #0] // temp=*arr
7     CBZ W9 exit      // while(*arr!=0) questo deve rimanere
8
9 while: SUB W10, W9, #97 // SUBI nel LEGv8
10    AND W10, W10, #255 // ANDI nel LEGv8
11    CMP W10, #26      // 122-97+1
12    CSET W10, cc
13    LSL W10, W10, #5
14    SUB W9, W9, W10
15    AND W9, W9, #255 // ANDI nel LEGv8
16    STURB W9, [X0, #0]
17    ADD X0, X0, #1    // ADDI nel LEGv8
18    LDURB W9, [X0, #0]
19    CBNZ W9, while
20
21 exit:

```

### 10.2.1: Stringa maiuscola senza branch

Voglio far notare come ho eseguito il controllo del primo elemento della stringa al di fuori del ciclo e ho inserito il controllo del loop a fine ciclo (dopo l'incremento dell'indice): così facendo ho risparmiato un'istruzione (un branch incondizionale).

Possiamo migliorare ancora il codice.

Noi stiamo leggendo elementi di un array di caratteri (ovvero di byte) e stiamo effettuando la stessa operazione su questi registri. Inoltre, il processore deve riordinare molte istruzioni (per com'è scritto ora il codice) e comunque durante l'esecuzione sarà costretto ad includere delle bolle, in quanto non è possibile riordinarle in modo da evitare stalli. Supponendo di conoscere la lunghezza dell'array<sup>28</sup> possiamo però applicare la tecnica del loop unrolling, ovvero dello "sbrogliare il ciclo": ripetere più volte la stessa istruzione ma cambiando indice degli elementi. In C una possibile rappresentazione di questo metodo con il codice *Stringa Maiuscola* è la seguente

---

<sup>28</sup>Potenzialmente si può fare anche con un array di lunghezza ignota, quindi ad esempio finché non compare un determinato numero (o carattere, nel caso della *Stringa Maiuscola*), ma è necessario utilizzare dei salti condizionali e il codice si allungherebbe notevolmente. Ho abbozzato su carta una possibile soluzione, ma penso si possa fare di meglio, quindi non la presento qui.

```

1 void strMaiusc(char *arr, int n){
2     char *endString;
3     endString = arr + n;
4     while(arr < endString){
5         *arr -= 32 * ((*arr >= 97) && (*arr <= 122)); // I
6         *(arr+1) -= 32 * ((*arr+1 >= 97) && (*(arr+1) <= 122)); // II
7         *(arr+2) -= 32 * ((*arr+2 >= 97) && (*(arr+2) <= 122)); // III
8         arr += 3;
9     }
10 }

```

## 10.2.2: Stringa maiuscola con loop unrolling

Vi invito a svolgere il codice Assembly e a riorganizzare le istruzioni in modo da evitare le bolle. È un buon ripasso per vedere se si è capito come funziona la pipeline del LEGv8 e i miglioramenti dovuti ai forwarding, ovvero al passaggio di dati tra i buffer tra gli stadi del LEGv8. Il compilatore clang però svolge il compito diversamente da come lo farei io a mano (mi sembra giusto, visto che conosce tutte le pops specifiche dell'architettura). [Qui](#) potete trovare un esempio dell'ARM stessa.

Per completezza vi riporto un estratto del codice generato con il compilatore impostato con l'ottimizzazione `-O2`:

```

1 strMaiusc:
2     add    x1, x0, x1
3     cmp    x0, x1
4     b.cs   exit
5     nop
6 while: ldrb    w4, [x0]           // I
7     add    x0, x0, #0x3         // arr+=3
8     ldurb  w3, [x0, #-2]       // II
9     ldurb  w2, [x0, #-1]       // III
10    sub    w6, w4, #0x61        // I
11    and    w6, w6, #0xff        // I
12    sub    w8, w4, #0x20        // I
13    sub    w5, w3, #0x61        // II
14    cmp    w6, #0x1a           // I
15    and    w5, w5, #0xff        // II
16    sub    w7, w3, #0x20        // II
17    sub    w6, w2, #0x61        // III
18    and    w8, w8, #0xff        // I
19    csel   w4, w8, w4, cc       // I
20    and    w7, w7, #0xff        // II
21    cmp    w5, #0x1a           // II
22    and    w6, w6, #0xff        // III
23    sub    w5, w2, #0x20        // III
24    csel   w3, w7, w3, cc       // II
25    and    w5, w5, #0xff        // III
26    cmp    w6, #0x1a           // III
27    csel   w2, w5, w2, cc       // III
28    sturb  w4, [x0, #-3]       // I
29    sturb  w3, [x0, #-2]       // II
30    sturb  w2, [x0, #-1]       // III
31    cmp    x1, x0              // arr<endString
32    b.hi   while
33 exit:  ret

```

### 10.2.2: Stringa maiuscola con Loop Unrolling

Trovo questo codice estremamente interessante, in quanto - oltre ad usare registri in modo "strano" rispetto alle usuali convenzioni, cosa che ci consentirebbe di sbrogliare ulteriormente il ciclo - fa uso della "nuova" istruzione *CSEL*, generando una nuova sequenza:

**LDUR - SUB - AND - SUB - AND - CMP - CSEL - STUR**

invece della precedente

**LDUR - SUB - AND - CMP - CSET - [LSL - SUB]<sup>29</sup> - AND - STUR**

Questo perché la nuova sequenza conserva *arr* e si prepara *arr - 32*, selezionando poi quello corretto. Notiamo come in entrambi i casi dopo l'istruzione di *CMP* c'è subito quella di *CSEL/CSET*: questo perché l'ALU dell'ARMv8 aggiorna i flag ad ogni operazione logica ed aritmetica. Così facendo risolviamo qualsiasi stallo e il programma potrà eseguire sicuramente senza alcuna bolla.

Torniamo un passo indietro: perché mi sono fermato a 3 sbrogliature? Per mantenere leggibile il codice e perché a mano rischiamo di finire i registri "sporcabili". GCC ha ovviato a quest'ul-

<sup>29</sup>In realtà l'ARMv8 sfrutta un'operazione combinata per effettuare lo shift logico di 5 (sottrazione di  $2^5 = 32$ ) e la sottrazione in un'unica *pop*. Il LEGv8 non è dotato di questa *pop*, quindi per rendere più comprensibile il codice ho preferito separare le due istruzioni

timo problema facendo uso di tutti i registri (compreso X8!) e sovrascrivendoli una volta che non servono più per una sbrogliatura. Vi invito a provare a trovare quant'è il numero massimo di sbrogliature effettuabili sfruttando tutti i registri sporcabili e quelli da salvare. Il codice sarà sicuramente più lungo, quindi farlo a mano non è molto conveniente, men che meno pensare al numero di sbrogli da effettuare quando si sta scrivendo il codice C (si suppone che non tutte le volte che dovremo scrivere un codice nel futuro sarà Assembly). Per risolvere questi problemi esistono sicuramente librerie già pronte che, combinate con ottimizzazioni del compilatore e sfruttando tutto l'hardware specifico della nostra architettura, ci consentono di risparmiare molta fatica e scrivere soltanto in un linguaggio di alto livello come il C.

Ecco che entra in gioco il SIMD, ovvero Single Instruction Multiple Data, una prima forma anche hardware di DLP (Data Level Parallelism). Il concetto è simile a quello del loop unrolling, solo che al posto di usare diversi registri per ogni variabile si utilizzano dei registri specifici per lavorare con diversi elementi dello stesso formato: i registri vettoriali. Inoltre, tutte le operazioni vengono eseguite in parallelo, invece che in sequenza come nel loop unrolling e nella programmazione "classica", ovvero SISD. In ARMv8 le unità logiche e il set di istruzioni dedicato alla programmazione SIMD si chiama [NEON](#).

Grazie alla programmazione SIMD possiamo eseguire la stessa operazione (Single Instruction) *contemporaneamente* su più elementi (Multiple Data) dello stesso tipo. Nell'ARMv8 ci sono i registri vettoriali (V) a 128 bit (= 16 Byte = una quadword), differenti da quelli interi (X). I registri floating point (D), invece, sono la parte bassa dei registri vettoriali. Per approfondire l'argomento si vedano i capitoli 3.6 → 3.12 e 6.3 → 6.15 del libro di testo *Computer Organization and Design: ARM Edition*.

Come accennavo nel paragrafo precedente, visualizzare queste modifiche in Assembly può essere complicato, in quanto non sempre la migliore ottimizzazione è la più lineare (ovvero sbrogliare il ciclo in  $\frac{16}{\text{lunghezza in Byte del tipo dato}}$  e far uso dei registri vettoriali). Se volete provate ad utilizzare l'ottimizzazione `-O3` invece della precedente `-O2`: vedrete utilizzati i registri vettoriali, sebbene non saranno estremamente ottimizzati: bisognerebbe usare sia i corretti comandi per il preprocessore, sia i corretti flag in fase di compilazione. In una prossima revisione cercherò quali sono i flag del compilatore (e i corretti header nel file C) per compilare il programma sfruttando al meglio le unità di calcolo NEON.

## 10.4 GDB per Linux

Finora abbiamo visto come scrivere del codice e come eseguirlo. Se invece volessimo verificare che si comporti come vorremmo possiamo usare uno strumento come GDB, GNU Debugger. Questo potentissimo strumento offre molte funzionalità, che potete scoprire per intero a [questo link](#) ([questa](#) invece è una guida interessante che contiene a fondo pagina un elenco ridotto dei comandi principali), ma in questa sezione ci soffermeremo solo su alcune di queste.

Per poter usufruire del debugger bisogna aggiungere il flag `-g` in fase di compilazione. Poi possiamo lanciare l'eseguibile tramite GDB con il comando

```
1 gdb ./nomeFile
```

Per uscire dall'ambiente di debug basta scrivere *quit* oppure *q* (ogni comando del GDB ha una sua abbreviazione).

Potrebbe far comodo visualizzare sia il codice Assembly che i nostri comandi, e - nel caso in cui fosse presente - anche il codice C: per far ciò esiste il comando *layout next*. Premendo invio si ripete l'ultimo comando, quindi premete invio finché non siete soddisfatti dalla grafica.

Prima di avviare l'esecuzione vogliamo inserire un breakpoint all'inizio del programma e/o in altri punti del codice di nostro interesse. Possiamo farlo attraverso il comando *break* o la sua abbreviazione *b* con dopo il nome dell'etichetta o della riga del codice C o Assembly, quindi

```
1 break main
```

Per ulteriori informazioni sui breakpoint o per osservare (*watch*) una variabile e fermare il programma quando cambia valore oppure rilevare *catch* un'eccezione, un caricamento di librerie dinamiche, ecc. vi invito a leggere [questi paragrafi](#).

Ora possiamo lanciare l'esecuzione del programma con *run* o *r*. Se il programma richiede qualche parametro possiamo forniglieli in fase di chiamata semplicemente separandoli con uno spazio. Il programma inizia l'esecuzione e si ferma o al primo errore, oppure al primo breakpoint inserito. Da qui possiamo scegliere come muoverci nel codice:

**next** : consente di passare alla successiva riga del codice, quindi se la riga è una funzione viene eseguita la funzione. La sua variante **ni**, ovvero next into, consente di visualizzare riga per riga il codice della funzione chiamata (se ce ne dovesse essere una);

**step** : consente di passare alla successiva istruzione, quindi se l'istruzione è una funzione si entra dentro l'esecuzione della funzione. La sua variante **si**, ovvero step into, consente di visualizzare istruzione per istruzione il codice della funzione chiamata (se ce ne dovesse essere una);

**continue** : continua l'esecuzione del codice fino al successivo breakpoint

**finish** : consente di finire l'esecuzione di una funzione (utile, ad esempio, se non si è interessati a vedere il resto del funzionamento di una sottofunzione)

Prima abbiamo visto come fare per poter visualizzare il codice con la grafica. Ma se volessimo personalizzare la visualizzazione? O se volessimo vedere automaticamente tutti i registri? Si può fare grazie a TUI (Terminal User Interface). La [guida ufficiale](#) contiene tutto il funzionamento, che vi suggerisco di leggere. A me piace impostare il prompt dei comandi verticale o se possibile a tutto schermo e usare il seguente layout:

```
1 tui new-layout franzlayout {-horizontal asm 1 regs 2} 2 status 0 cmd 1
2 layout franzlayout
3 tui reg general
```

Per muoversi su e giù con le finestre basta usare le frecce della tastiera<sup>30</sup>, per selezionare un'altra finestra basta usare il comando *focus* seguito dalla sigla della finestra; per ridisegnare lo schermo se si dovessero formare brutti artefatti c'è *redraw*.

## 10.5 Comunicare con il RP dal proprio PC

Il mio Raspberry Pi 3B+ ha 1 GB di RAM e per qualche strano motivo il sistema a 64 bit gira più lentamente rispetto a quello a 32 bit. Questi due fattori, combinati con l'ambiente desktop, implicano che avviare qualsiasi programma con una grafica "avanzata" e qualche funzionalità in più del semplice blocco di testo richiede un'eternità. È il caso, ad esempio, di Visual Studio Code, un'IDE della Microsoft che consente di programmare in qualsiasi linguaggio grazie alle estensioni.

Le estensioni, infatti, consentono di aggiungere al programma ulteriori funzioni estremamente utili, rendendolo uno strumento versatile per iniziare a codificare in qualsiasi linguaggio di programmazione. Una di queste funzionalità consiste nella possibilità di sfruttare SFTP e SSH per controllare il RP da remoto, ovvero da un altro computer connesso alla stessa rete.

Tra le estensioni di VSC cerchiamo *SFTP* di *Natizyskunk* e installiamola. Poi può far comodo installare anche *Arm Assembly* di *dan-c-underwood*, in modo da avere la sintassi ARM evidenziata.

La pagina di *SFTP* contiene una più che esaustiva guida su come utilizzare l'estensione. Riassumo comunque i passaggi principali da eseguire:

1. Aprite una nuova finestra di VSC e nella sezione *Explorer* selezionate una cartella che verrà sincronizzata con il RP.
2. con *Ctrl + Shift + P* si apre il pannello dei comandi<sup>31</sup> e inserite *SFTP: config*
3. Nel file *.json* compilate i campi:

**host** con l'IP del RP connesso alla rete<sup>32</sup>

**username** con il nome utente che usate per accedere al RP

**password** con la password che usate per accedere al RP (questo campo è da creare)

**remotePath** con il percorso della cartella dove volete salvare tutti i codici<sup>33</sup>

---

<sup>30</sup>Le frecce non funzionano bene con i registri, per questo motivo suggerisco di allargare la finestra dei registri come nel mio layout personalizzato e mettere a schermo intero il prompt dei comandi di VSC, nonché di usare un font  $\leq 12$  per il prompt dei comandi.

<sup>31</sup>Sul mio pc c'è anche una barra di ricerca in alto: la clicco, scrivo  $>$  e poi continuo ad inserire il comando

<sup>32</sup>Per trovare l'IP del RP potete usare qualche programma di scansione della rete locale dal vostro pc, o andare nelle impostazioni del router e cercare tra i dispositivi connessi, oppure aprire un prompt dei comandi sul RP e digitare *ip a*: sotto la voce *eth0>inet* trovate l'IP del RP.

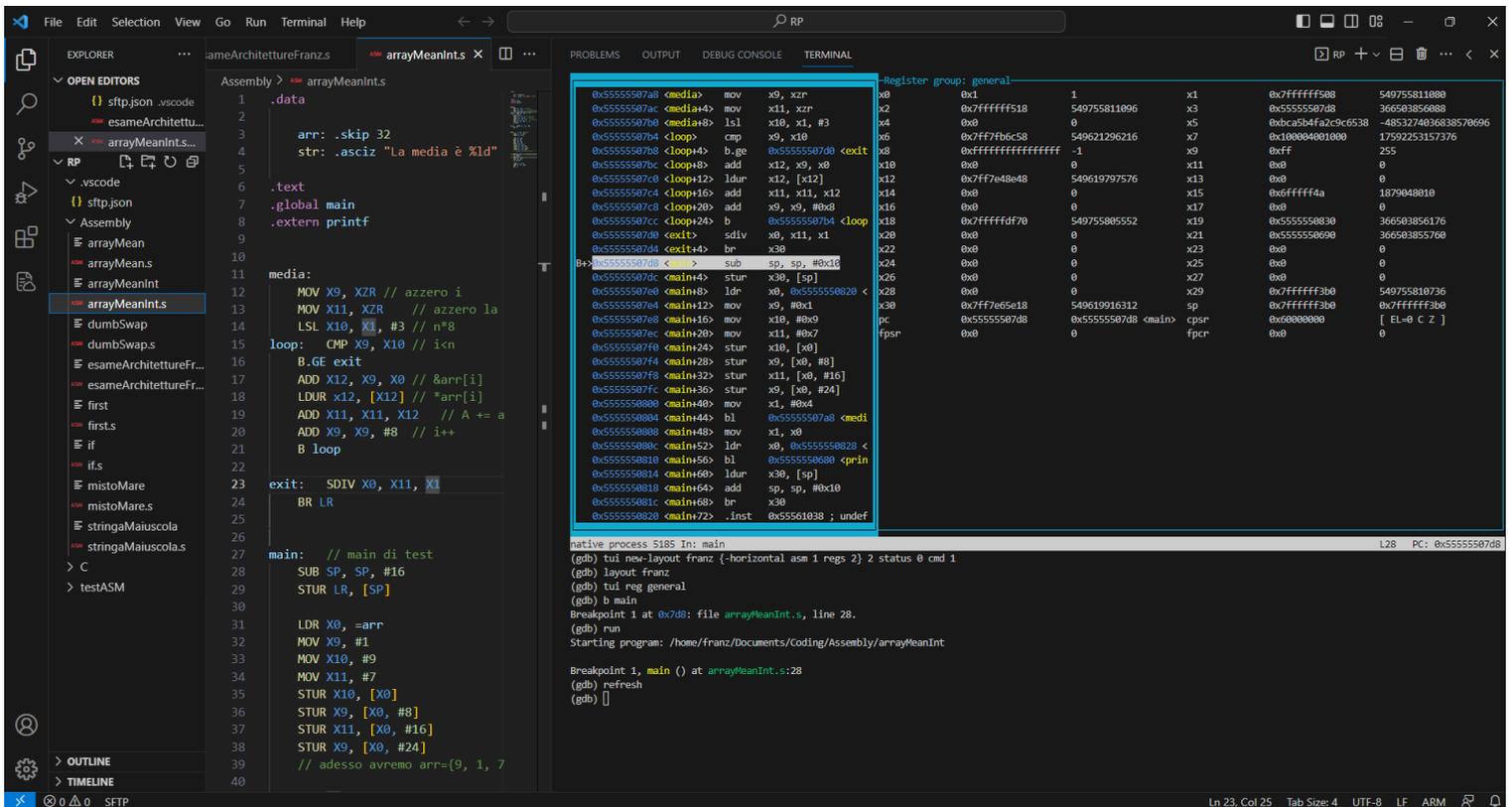
<sup>33</sup>Per trovare questo indirizzo andate sul RP e, se lo usate con la grafica aprite la cartella, click destro, apri nel CMD, e scrivete *pwd*, se usate il RP via SSH, invece, andate nella cartella con il classico *cd* (change directory) e poi scrivete *pwd*.

`uploadOnSave` vi suggerisco di metterlo true, in quanto è comodo e vi consente di non dover caricare sul RP manualmente il codice ogni volta che effettuate una modifica.

Gli altri parametri li potete lasciare come stanno. Salvate il file.

4. Digitate il comando *SFTP: Open SSH in Terminal*: vi si aprirà il prompt dei comandi integrato in VSC con la connessione al RP.
5. Digitate *SFTP: Download Project* per scaricare la cartella con i file già contenuti sul RP (nel caso in cui dovessero esserci, altrimenti potete saltare questo passaggio). Se invece avete creato tutti i programmi sul vostro PC e volete caricarli sul RP basta assicurarsi che siano presenti nella cartella

Ora siete pienamente configurati per poter lavorare comodamente dal vostro pc sul RP. Ringrazio Luca Fontanot per il suggerimento e il supporto.



## 11 Ulteriori esercizi

Presento qui ulteriori esercizi in ordine di difficoltà crescente. Le soluzioni sono sempre nel capitolo successivo.

```

1 long long strToInt(char *str){
2     long long ris = 0, mul=1, i=0;
3     if(str[0] == '-'){
4         i++;
5         mul=-1;
6     }else if(str[0] == '+')
7         i=1;
8     while(str[i]!='\0'){
9         if(str[i]<48 || str[i] > 57)
10            return -1;
11        ris*=10;
12        ris += (long long) (str[i]-48);
13        i++;
14    }
15    ris*=mul;
16    return ris;
17 }

```

### 11.1: Conversione di una stringa in un intero

```

1 void convolution(double *x, double *h, double *y, long long n, long long k){
2     long long start, i, j;
3     double temp;
4     for(i=0; i<n; i++){
5         start = i - k/2;
6         temp=0;
7         for(j=0; j<k; j++){
8             if(start+j>=0 && start+j<n)
9                 temp += x[start+j] * h[j];
10        }
11        y[i] = temp;
12    }
13 }

```

### 11.2: Convoluzione di un array generico di double con un kernel di dimensione non nota

```

1 void countingSort(unsigned long long *arr, unsigned long long len, unsigned long long max){
2
3     unsigned long long counter[max];
4     unsigned long long i,j;
5
6     for(j=0; j<max; j++) //resettiamo l'array
7         counter[j]=0;
8
9     for(i=0; i<len; i++) // contiamo quanti sono i numeri da 0 a max
10        counter[arr[i]]++;
11
12    i=0, j=0;
13    while(i<len){ // sistemiamo nell'array originale tutti i numeri da 0 a max
14        while(counter[j]){ // finche' sono presenti (ovvero finche' counter[j]>0)
15            arr[i] = j;
16            counter[j]--;
17            i++;
18        }
19        j++;
20    }
21 }

```

### 11.3: Algoritmo di ordinamento CountingSort

```

1 float autocorrl(float *x, long long l, long long n);
2
3 void autocorrel(float *x, float *y, long long len){
4     long long n;
5     for(n=0; n<len; n++)
6         y[n] = autocorrl(x, len-n; n);
7 }

```

### 11.5: Autocorrelazione

Nelle future revisioni aggiungerò ulteriori esercizi

## 12 Soluzione esercizi

```

1 input :
2 X0 -> a
3 X1 -> b
4 X2 -> c
5
6 output :
7 X0 -> somma
8
9 ADD X0, X0, X1
10 ADD X0, X0, X2

```

#### 2.1.1: somma di tre numeri

```

1 /*
2 input: X0 -> a    X1 -> b
3 output: X0 -> 3a+(4b)%3
4 */
5
6 ADDI X9, XZR, #3 //salvo 3 e 4 in due registri
7 ADDI X10, XZR, #4 // per usarli con la moltiplicazion
8 MUL X0, X0, X9 // 3*a
9 MUL X1, X1, X10 // 4*b
10 SDIV X11, X1, X9 // (4*b)/3, attenzione, questa e' la divisione intera quindi moltiplicando
11 MUL X11, X11, X9 // di nuovo per 3 ottengo un numero <= 4b. sottraendo questo numero
12 SUB X1, X1, X11 // dal precedente otteniamo il modulo, quindi il resto della divisione
13 ADD X0, X0, X1 // 3*a + (4*b)%3

```

## 2.1.2: misto mare

```

1 median3: SUBS XZR, X0, X2 // a>c
2         B.LE if2 // salto con la logica inversa
3         MOV X9, X0 // temp=a
4         MOV X0, X2 // a=c
5         MOV X2, X9 // c=temp
6
7 if2:    SUBS XZR, X0, X1 // a>b
8         B.LE if3
9         MOV X9, X0
10        MOV X0, X1
11        MOV X1, X9
12
13 if3:    SUBS XZR, X1, X2 // b>c
14        B.LE exit
15        MOV X9, X1
16        MOV X1, X2
17        MOV X2, X9
18
19 exit:   MOV X0, X1 // return b

```

## 2.2.1: Median3

```

1 /*
2 input: X0 -> &arr   X1 -> len   X2 -> 1.0F
3 X9 -> fineArr
4 D0 -> A   D1 -> *arr   D2 -> i
5 output: D0 -> A
6 */
7
8     LDURD D3, [X2, #0] // leggo la costante 1.F
9     FSUBD D0, D3, D3 // azzero A ed i. avrei potuto fare LDURD D0(D2), [X2, #0] ma ...
10    FSUBD D2, D3, D3 // ... i load richiedono piu' tempo di una somma/sottrazione
11    LSL X1, X1, #3
12    ADD X1, X9, X0
13
14 while: CMP X0, X1
15         B.GE exit
16         LDURD D1, [X1, #0] // *arr
17         FADDD D0, D0, D1 // A += *arr
18         FADDD D2, D2, D3 // i++
19         ADDI X1, X1, #8 // arr++
20         B while
21
22 exit:  FDIVD D0, D0, D2

```

### 5.3: media aritmetica di un array di double

```

1 fib:  MOV X9, XZR // a=0
2       ADDI X10, XZR, #1 // b=1
3 for:  CMP X0, XZR
4       B.LT exit
5       MOV X11, X9 // temp = a
6       MOV X9, X10 // a = b
7       ADD X10, X10, X11 // b += temp
8       SUBI X0, X0, #1 // n--
9       B for
10
11 exit: MOV X0, X10 // return b
12       BR LR

```

### 6.1: Fibonacci iterativo

```

1 // codice C iterando sull'array per puntatore
2 long long fibIterMem(long long n){
3     long long fibArr[n]
4     fibArr[0] = 0;
5     fibArr[1] = 1;
6     long long *index, *endArr;
7     index = fibArr+2;
8     endArr = fibArr+n
9     while(index < fibArr){
10        *index = *(index-1) + *(index-2);
11        index++;
12    }
13 }
14
15 // corrispondente codice Assembly
16 fibIterMem:
17     LSL X9, X0, #3        // preparo il # di byte per lo stack
18     MOV X12, SP          // preparo il puntatore alla fine dell'array
19     SUB SP, SP, X9       // alloco nello stack lo spazio per fibArr
20
21     SUBI X11, XZR, #1    // memorizzo -1
22     STUR XZR, [SP, #0]  // fibArr[0]=0
23     ADDI X15, XZR, #1   // memorizzo 1
24     STUR X15, [SP, #8]  // fibArr[1]=1
25
26     ADDI X9, SP, #16    // &fibArr[2]
27 fib:  CMP X9, X12        // if &fibArr[i] < &fibArr[n+1]
28     B.GE return
29     LDUR X10, [X9, #-8] // *fibArr[i-1]
30     LDUR X11, [X9, #-16]// *fibArr[i-2]
31     ADD X10, X10, X11   // fibArr[i-1] + fibArr[i-2]
32     STUR X10, [X9, #0] // fibArr[i] = fibArr[i-1] + fibArr[i-2]
33     ADDI X9, X9, #8     // i+1
34     B fib
35
36 return: MOV X0, X10     // restituisce l'n-esimo numero di fibonacci (per restituire il
37                    // puntatore all'array bisogna usare i dati dinamici)
38     MOV SP, X12        // ripristino lo stackpointer
39     BR LR

```

#### 8.4.1: Fibonacci iterativo con memoria

NB: nonostante abbia chiamato il programma "con memoria", ovvero vengono salvati tutti i numeri di Fibonacci fino all'n-esimo in un array, quest'ultimo non viene restituito in output, in quanto è stato dichiarato nello stack e non nella memoria dinamica, quindi dev'essere deallocato alla fine dell'esecuzione. Per allocarlo nella memoria dinamica è necessario sfruttare il Frame Pointer. [Questo](#) è un esempio di programma che fa uso del FP.

```

1 fibRicMem:
2     LSL X9, X0, #3           // preparo lo spazio nello stack per l'array
3     ADDI X9, X9, #1         // mi preparo posto nello stack anche per n originale
4     SUB SP, SP, X9          // alloco lo spazio nello stack
5     ADD X10, SP, X9         // punto alla fine dell'array
6     MOV X1, SP
7     STUR X0, [X10, #0]     // salvo n originale
8
9     STUR XZR, [SP, #0]     // fibArr[0] = 0
10    MOVI X11, #1
11    STUR X11, [SP, #8]     // fibArr[1] = 1
12    MOVI X11, #-1         // preparo -1 per dopo
13
14    ADDI X12, X9, #16      // &fibArr[2]
15 init: CMP X12, X10        // &fibArr[i] <= &fibArr[n]
16    B.GE fib
17    STUR X11, [X12, #0]   // fibArr[i] = -1
18    ADDI X12, X12, #8     // i++
19    B init
20
21 // ora mi servono solo X0 (n), X1 (&fibArr), X10 (&fibArr[n+1])
22 fib: LSL X9, X0, #3
23    ADD X9, X1, X9         // &fibArr[n]
24    LDUR X11, [X9, #0]    // *fibArr[n]
25    CMP X11, XZR          // fibArr[n]<0
26    B.LT if
27    MOV X0, X11           // return fibArr[n]
28    BR LR
29
30 if:  SUBI SP, SP, #32     // alloco nuovo spazio nello stack per n, LR e fibArr[n-1]
31    STUR X0, [SP, #0]     // carico n
32    STUR LR, [SP, #8]     // carico LR
33    SUBI X0, X0, #1       // n-1
34    BL fib                // fib(n-1, fibArr)
35
36    STUR X0, [SP, #16]    // carico fibArr[n-1]
37    LDUR X0, [SP, #0]     // scarico n
38    SUBI X0, X0, #2       // n-2
39    BL fib                // fib(n-2, fibArr)
40
41    LDUR X9, [SP, #0]     // scarico n
42    LSL X9, X9, #3
43    ADD X9, X9, X1        // &fibArr[n]
44    LDUR X11, [SP, #16]   // scarico fibArr[n-1]
45    ADD X0, X0, X11       // fibArr[n-1] + fibArr[n-2]
46    STUR X0, [X9, #0]    // fibArr[n] = fibArr[n-1] + fibArr[n-2]
47
48    LDUR LR, [SP, #8]     // ripristino LR
49    ADDI SP, SP, #32     // dealloco lo stack
50    BR LR

```

#### 8.4.2: Fibonacci ricorsivo con memoria

NB: abbiamo sempre detto che quando c'è una chiamata ad una funzione si assume che tutti i registri  $X0 \rightarrow X15$  vengano sporcati. Allora perché non ho salvato  $n$  nello stack? Potenzialmente avrei potuto, ma ho preferito utilizzare un registro per motivi di velocità di accesso alla memoria, consapevole del fatto che  $X1$  non viene sporcato, in quanto non sto chiamando alcuna funzione esterna al codice da me scritto e durante le chiamate (quindi dentro *fib*) non sporco  $X1$ .

```

1 /*
2 INPUT: X0 -> &h X1 -> &X X2 -> N D0 -> d D1 -> mu
3 X9 -> i
4 OUTPUT: D0 -> y
5 */
6
7 // ho riordinato le istruzioni in modo da evitare il maggior numero possibile di bolle
8 LMS1:SUBI SP, SP, #48
9 STURD D0, [SP, #0] // carico d
10 STURD D1, [SP, #8] // carico mu
11 STUR X0, [SP, #16] // carico &h
12 STUR X1, [SP, #24] // carico &X
13 STUR X2, [SP, #32] // carico N
14 STUR LR, [SP, #40] // carico LR
15
16 BL FIR1_D // ricevero' il risultato in D0
17
18 LDUR X2, [SP, #32] // scarico N (scarico prima N anticipando un'operazione: evito bolle)
19 LDURD D1, [SP, #0] // scarico d
20 LDURD D2, [SP, #8] // scarico mu
21 LDUR X0, [SP, #16] // scarico &h (scarico qui queste variabili: tolgo bolle sia ai ...
22 LDUR X1, [SP, #24] // scarico &X ...caricamenti precedenti, sia a questi due caricamenti)
23 MOV X9, XZR // i=0 (anche questo l'ho spostato per evitare una bolla con CMP)
24 LSL X2, X2, #3 // N*=8
25 FSUBD D3, D1, D0 // e = d - y
26 FMULD D4, D3, D2 // em = e * mu
27
28 CMP X9, X0 // if i<N ((i-n)>=0 ?)
29 B.GE exit
30 for:ADD X11, X1, X9 // &X[i]
31 LDURD D2, [X11, #0] // X[i]
32 ADD X10, X0, X9 // &h[i]
33 LDURD D1, [X11, #0] // h[i]
34 ADDI X9, X9, #8 // ho tolto l'istruzione di LSL (spostata per evitar bolle)
35 FMULD D2, D4, D2 // em * X[i]
36 FADDD D1, D1, D2 // h[i] += em * X[i] (in queste ultime righe non riesco ad evitar bolle)
37 STURD D1, [X10, #0]
38 CMP X9, X0 // rispetto all'altra versione ho spostato il controllo di uscita dal...
39 B.LT for // ...ciclo alla fine per risparmiare un'istruzione e ho aggiunto fuori...
40 //...dal ciclo il primo controllo (il ciclo e' un for/while, non un do-while)
41 exit:LDUR LR, [SP, #40]
42 ADDI SP, SP, #48
43 BR LR // ho fatto in modo che in D0 ci sia sempre y, quindi non devo ripristinarlo

```

### 9.3.2: Chiamata a FIR1D ottimizzato

```

1 /*
2 input: X0 -> *str
3   X9 -> ris   X10 -> i   X11 -> str[i]   X12 -> 10   X13 -> mul
4   '0' = 48   '9' = 57   '\0' = 0   '-' = 45   '+' = 43
5 output: X0 -> ris
6 */
7     LDURB X11, [X0,#0] // str[0]
8     ADDI X12, XZR, #10 // salvo 10 x la moltiplicazione
9     ADDI X10, XZR, #0 // i=0
10    MOV X9, XZR
11    ADDI X13, XZR, #1
12    SUBIS XZR, X11, #45 // controlla se c'e' il segno
13    B.NE piu
14
15    SUBI X13, XZR, #1
16 meno: ADDI X10, X10, #1 // i++ 1 byte perche' stiamo lavorando con char
17
18 piu:   SUBIS XZR, X11, #43 // se non e' '-' allora verifica che non sia '+'
19       B.NE loop
20       ADDI X10, X10, #1 //1 byte in quanto stiamo lavorando con char
21
22 loop:  LDURB X11, [X0, #0] // str[i]
23       SUBIS XZR, X11, #0 // str[i] ?= '\0'
24       B.EQ return
25       SUBIS XZR, X11, #48 // str[i] ?< '0'
26       B.LT retErr
27       SUBIS XZR, X11, #57 // str[i] ?> '9'
28       B.GT retErr
29       BR LR
30
31       MUL X9, X9, X12 // ris*=10
32       SUBI X11, X11, #48 // str[i]-48
33       ADD X9, X9, X11 // ris += str[i]-48
34       ADDI X10, X10, #1 // i++
35       BL loop
36
37 retErr: SUBI X0, XZR, #1 // se arriviamo qui significa che e' stato trovato un carattere
38         BR LR // diverso da un numero, quindi ritorniamo -1 come da testo
39
40 return: MUL X0, X11, X13 // se siamo arrivati qui allora abbiamo trovato '\0', ovvero
41         BR LR // la fine della stringa e quindi ritorniamo il numero calcolato

```

### 11.1: Conversione di una stringa in un intero

```

1 /*
2 input: X0 -> x X1 -> h X2 -> y X3 -> n
          X4 -> k X5 -> &(double)0
3 X9 -> i X10 -> j X11 -> start
4 D9 -> temp D10 -> x[start+j] D11 -> h[j]
5 */
6     MOV X9, XZR // i=0
7     LDURD D31, [X5, #0] // load (
double)0
8 extloop: CMP X9, X3 // i-n
9     B.GE exit
10    LSR X11, X4, #1 // k/2
11    SUB X11, X9, X11 // i-k/2
12    FADDD D9, D31, D31 // temp=0
13
14    MOV X10, XZR // j=0
15 intloop: CMP X10, X4 // j-k
16    B.GE fineconv
17
18    ADDS X12, X11, X10 // start+j
19    B.LT intloop
20    CMP X12, X3 // start+j<n
21    B.GE intloop
22    LSL X12, X12, #3 // (start+j)*8
23
24    ADD X12, X12, X0
25    LDURD D10, [X12, #0] // x[start+j]
26    LSL X13, X10, #3
27    ADD X13, X13, X1
28    LDURD D11, [X13, #0] // h[j]
29    FMULD D10, D10, D11 // x[start+j]
30    * h[j]
31    FADDD D9, D9, D10 // temp += x[
start+j] * h[j]
32    ADDI X10, X10, #1
33    B intloop
34
35 fineconv: LSL X13, X9, #3 // i*8
36    ADD X13, X13, X2 // y[i]
37    STURD D9, [X13, #0] // y[i] = temp
38    ADDI X9, X9, #1
39    B extloop
40 exit: BR LR

```

```

1 /*
2 input: X0 -> x X1 -> h X2 -> y X3 -> n
          X4 -> k
3 X9 -> i X10 -> j X11 -> start
4 D9 -> temp D10 -> x[start+j] D11 -> h[j]
5 */
6     MOV X9, XZR // i=0
7     LDURD D31, [X5, #0] // load (
double)0
8 extloop: CMP X9, X3 // i-n
9     B.GE exit
10    LSR X11, X4, #1 // k/2
11    SUB X11, X9, X11 // i-k/2
12    FADDD D9, D31, D31 // temp=0
13
14    MOV X10, XZR // j=0
15 intloop: CMP X10, X4 // j-k
16    B.GE fineconv
17
18    ADDS X12, X11, X10 // start+j
19    B.LT intloop
20    CMP X12, X3 // start+j<n
21    B.GE intloop
22    LSL X12, X12, #3 // (start+j)*8
23
24    ADD X12, X12, X0
25    LDURD D10, [X12, #0] // x[start+j]
26    LSL X13, X10, #3
27    ADD X13, X13, X1
28    LDURD D11, [X13, #0] // h[j]
29    FMULD D10, D10, D11 // x[start+j]
30    * h[j]
31    FADDD D9, D9, D10 // temp += x[
start+j] * h[j]
32    ADDI X10, X10, #1
33    B intloop
34
35 fineconv: STURD D9, [X2, #0] // *y = temp
36    ADDI X2, X2, #8 // y++
37    incremento ed uso y per puntatore
38    ADDI X9, X9, #1
39    B extloop
40 exit: BR LR

```

11.2: Convoluzione di un array generico con un kernel di dimensione non specificata, a sinistra tramite lettura per valore, a destra per puntatore

```

1  /*
2  input:  X0 => &arr    X1 => len    X2 => max
3  X9 => i    X10 => j
4  */
5
6  CountingSort:
7      MOV X9, XZR        //i=0
8      MOV X10, XZR       //j=0
9      LSL X15, X2, 3     //X15=max*8
10     SUB SP, SP, X15    //riduco lo SP per contenere counter[max]
11 for1:  CMP X10, X2     //se j>=max esco dal for
12     B.GE for2
13     LSL X11, X10, 3    //X11=j*8
14     ADD X11, SP, X11   //X11=&counter[j]
15     STUR XZR, [X11, #0] //counter[j]=0
16     ADDI X10, X10, #1  //j++
17     B for1            //rientro nel primo for
18 for2:  CMP X9, X1     //se i>=len esco dal for
19     B.GE for_out
20     LSL X11, X9, 3    //X11=i*8
21     ADD X11, X0, X11   //X11=&arr[i]
22     LDUR X11, [X11, #0] //X11=arr[i]
23     LSL X11, X11, 3    //X11=arr[i]*8
24     ADD X11, SP, X11   //X11=&counter[arr[i]]
25     LDUR X12, [X11, #0] //X12=counter[arr[i]]
26     ADDI X12, X12, #1  //X12=counter[arr[i]]+1
27     STUR X12, [X11, #0] //counter[arr[i]]++
28     ADDI X9, X9, #1   //i++
29     B for2            //rientro nel secondo for
30 for_out:MOV X9, XZR    //i=0
31     MOV X10, XZR     //j=0
32 while1: CMP X9, X1    //se i>=len esco dal primo while
33     B.GE out1
34 while2: LSL X11, X10, 3 //X11=j*8
35     ADD X11, SP, X11   //X11=&counter[j]
36     LDUR X12, [X11, #0] //X12=counter[j]
37     CMP X12, XZR      //se counter[j]<=0 esco dal secondo while
38     B.LE out2
39     SUBI X12, X12, #1  //X12=counter[j]-1
40     STUR X12, [X11, #0] //counter[j]--
41     LSL X11, X9, 3    //X11=i*8
42     ADD X11, X0, X11   //X11=&arr[i]
43     STUR X10, [X11, #0] //arr[i] = j
44     ADDI X9, X9, #1   //i++
45     B while2          //rientro nel secondo while
46 out2:  ADDI X10, X10, #1 //j++
47     B while1          //rientro nel primo while
48 out1:  ADD SP, SP, X15 //ripristino lo SP
49     BR LR             //ritorno al chiamante

```

### 11.3: CountingSort

```

1  /*
2  INPUT: X0 -> &x, X1 -> &y, X2 -> len
3
4  OUTPUT: nessuno
5  */
6
7  autocorrel:
8      SUBI SP, SP, #48    //alloco 3 quadword per ospitare 5 doubleword
9      STUR LR, [SP, #32]
10     STUR X19, [SP, #0] // uso i registri "autosalvati" in quanto mi consentono ...
11     STUR X20, [SP, #8] // ... di risparmiare otto istruzioni all'interno del ciclo: ...
12     STUR X21, [SP, #16] // ... il salvataggio, prima della chiamata, e il ripristino, ...
13     STUR X22, [SP, #24] // ... dopo la chiamata alla funzione, delle quattro variabili
14     MOV X19, XZR        // X19=n
15     MOV X20, X0         // X20=&x
16     MOV X21, X1         // X21=&y
17     MOV X22, X2         // X22=len
18
19     CMP X19, X22
20     B.GE exit
21 for: MOV X0, X20        // X0 -> &x
22     SUB X1, X22, X19    // X1 -> len - n
23     MOV X2, X19        // X2 -> n
24     BL autocorrl
25
26     LSL X9, X19, #2
27     ADD X9, X21         // X9 = &y[n]
28     STURS S0, [X9, #0] // y[n] = autocorrl(...)
29
30     ADDI X19, X19, #1   // mi muovo di un solo byte
31     CMP X19, X22       // il controllo della condizione e' alla fine, ma ce n'e' ...
32     B.LT for          // ... uno anche prima del ciclo
33
34 exit: LDUR X19, [SP, #0]
35     LDUR X20, [SP, #8]
36     LDUR X21, [SP, #16]
37     LDUR X22, [SP, #24]
38     LDUR LR, [SP, #32]
39     ADDI SP, SP, #48
40     BR LR

```

## 11.5: Autocorrelazione