Introduzione alla programmazione assembly in LEGv8

Rev 1.1

Francesco Vodisca

A.A. 2021/2022

Indice

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.

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

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

Script 1: 2.1: prima somma

Scegliamo di ricevere gli input nei registri X0 e X1 e di mettere il risultato in X2[1]. Possiamo scrivere il seguente codice per LEGv8

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

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 le operazioni (ad esempio 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 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

```
long long max(long long a, long long b){
  if(a>b)
    return a;
  return b;
}
```

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] 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

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

oppure, utilizzando le pseudoistruzioni:

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

Il controllo *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 >= 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;[2]

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[3] dei 4 flag precedentemente attivati per determinare se saltare o meno all'istruzione specificata da un'etichetta (*exit* nel nostro caso, 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

```
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

```
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:

[2] È possibile approfondire il funzionamento di queste istruzioni nella sezione Operazioni condizionali

[3] Vedi tabella Condizioni per i salti condizionali e loro verifica hardware per sapere quali

```
long long median3(long long a, long long b, long long c){
  long long temp;
  if(a>c){
    temp=a;
    a=c;
    c=temp;
  }
  if(a>b){
    temp=a;
    a=b;
    b=temp;
  }
  if(b>c){
    temp=b;
    b=c;
    c=temp;
  }
  return b;
}
```

Script 2: 2.2.1: Median3

La soluzione è 2.2.1: Median3. 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.

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[4]. 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 programmiamo 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[5] o su carta. Nella sezione 10 vedremo come si può programmare Assembly su

[4] 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.

[5] 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.

un Raspberry Pi, 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: 2.1: prima somma. 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

```
ADDI X0, XZR, #4
ADDI X1, XZR, #7
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 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).
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.
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 anteponendolo 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

```
int a = 5;
int* p;              // dichiaro il puntatore p di tipo intero
p = &a;              // assegno al puntatore p l'indirizzo della variabile intera a
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[6], 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.[7]
Se sfogliate il manuale di LEGv8 (o ARMv8, o x86, o RISC-V, o qualsiasi altra architetura) noterete che non esistono registri per i puntatori. Questo perché per l'ALU *tutto è numero*[8]. 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

[6] 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.

[7] 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

[8] Pitagora ha da sempre avuto ragione

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

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

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

```
/*
input: X0 -> &a    X1 -> n
*/

    MOV X9, XZR    // i=0
while:  CMP X9, X1     // i<n
    B.GE exit
    LSL X10, X9, #3 // i*8
    ADD X10, X10, X0  // &arr[i]
    LDUR X11, [X10, #0] // *arr[i]
    ADDI X11, X11, #1 // arr[i]++
    STUR X11, [X10, #0]
    B while
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)[9].

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.

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

[9] 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.

Gb per chip, fornendo quindi una capacità totale di molti GB, se non TB nel caso dei server[10]. Proprio per il fatto che i registri sono pochi, si trovano all'interno dell'ALU e di conseguenza accedere al loro contenuto è un'operazione istantanea, 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)[11] ed ha tempi di accesso maggiori, dovuti a differenze architetturali. Questo comporta delle latenze non trascurabili quando si deve reperire un dato dalla memoria principale, e per questo motivo è un'operazione da ridurre il più possibile dove non è necessaria.
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.

	Registri	Memoria principale
Dimensioni	$2 \cdot 256B$ Nel LEGv8: 32 registri interi e 32 registri per i numeri a virgola mobile	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

[10] 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 (DDR4) può fornire 128 GB per DIMM, divisi in 18 chip composti da 4 chip sovrapposti da 16 Gb ciascuno. Per approfondire ulteriormente l'argomento vi suggerisco questa pagina della Micron sulle DDR5, nonché l'introduzione alle memorie nel corso di Dispositivi e Sistemi Elettronici.

[11] 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.

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. 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 operazioni 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 X9] di 8 bit con *LSR XZR, X9, #8*) e verificare se è 0 oppure 1 l'ultima cifra.[12]

```
while(*arr!=0){
  if(*arr>=97 && *arr<=122)
    *arr -= 32;
  p++;
}
```

In assembly diventa

```
/*
X0 -> *arr       X9 -> temp
*/

strMaiusc:  LDUR X9, [X0, #0] // temp=*arr
            CBZ X9 exit       // while(*arr!=0)

            CMPI X9, #97      // if(*arr>=97)
            B.LT finewhile
            CMPI X9, #122     // if(*arr<=122)
            B.GT finewhile
            SUBI X9, X9, #32  // temp-=32
            STUR X9, [X0, #0] // *arr=temp

finewhile:  ADDI X9, X9, #8   // arr++
            B strMaiusc
exit:
```

Vediamo ora come vengono rappresentati i numeri a virgola mobile. 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[13]. Questi registri non usano la convenzione dei registri X e non dispongono di nessun "DZR" (un analogo del registro XZR [o X31] degli interi che viene sempre

[12] 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*.

[13] 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). Perché fare questa differenza tra i registri interi e quelli per le operazioni floating point? Perché il LEGv8 supporta lo standard IEEE 754 a 64 e 32 bit. Questi non fanno uso di una rappresentazione posizionale ma di una convenzione dove il primo bit è il bit di segno, i successivi 8 (11) bit sono per memorizzare l'esponente (a cui

letto 0 e sul quale non si può memorizzare alcun valore). Inoltre non dispongono di operazioni logiche ed immediate, e non fanno uso delle stesse operazioni matematiche dei numeri interi. Per operare con i double è quindi necessario caricare sempre i valori di partenza da memoria, in quanto non è possibile nemmeno convertire direttamente un valore intero (contenuto in un registro X) in un valore double (contenuto in un registro D), questo perché il LEGv8 non è dotato dell'hardware di conversione. È però possibile trasferire un valore bit a bit da un registro all'altro facendo uso dello stack (sarebbe più corretto usare la memoria dinamica, ma non sapendo usarla dobbiamo fare uso dello stack; si veda la sezione 6).

Approfondimento sul trasferimento bit a bit

Vediamo un esercizio che fa uso di questo trucchetto.
Un algoritmo C alquanto interessante è il *FastInverseSquareRoot*, per calcolare il reciproco della radice quadrata[14]. Ecco il codice C. Per una spiegazione del suo funzionamento vi rimando a questo video.

```c
float FastInverseSquareRoot( float n){

  long i;
  float x, y;
  const float treMezzi = 1.5F;

  x= n*0.5F;
  y=n; //creiamo una copia lavorativa del numero
  i= *(long *) &y; // convertiamo bit a bit il float in un intero cosi' da poterlo manipolare
  i= 0x5f3759df -(i >> 1); //trucchetto matematico con i logaritmi. il numero sarebbe
    1.3211836173e+19
  y= *(float *) &i; //riconversione a float bit a bit
  y = y * (treMezzi - (x*y*y)); //metodo di newton per eliminare l'errore
  //y = y * (treMezzi - (x*y*y)); // seconda iterazione (facoltativa) per migliorare
    ulteriormente il risultato
  return y;
}
```

Ed ecco il corrispondente codice Assembly

```
/*
input: X0 -> &(0x5f3759df)    X1 -> &0.5F   X2 -> &1.5F   S0 -> n
  X9 -> &n(stack)   X10 -> i    X15 -> 0x5f3759df
  S5 -> 0.5f    S15 -> 1.5F
output: S0 -> y
*/

```

viene aggiunto un bias: $2^8 - 1 = 127$ nel caso di 32 bit e $2^{11} - 1 = 1023$ nel caso di 64 bit) e gli ultimi 23 (52) bit per memorizzare la frazione, ovvero la mantissa a cui viene sottratto 1 per guadagnare un bit di memoria (il primo bit significativo sarà sempre 1, quindi non ha senso memorizzarlo). Il numero memorizzato sarà quindi ottenuto da $(-1)^{\text{bitDiSegno}} \cdot (1 + \text{frazione}) \cdot 2^{\text{esponente - bias}}$

[14] I programmatori di videogiochi si possono rendere subito conto dell'importanza di tale algoritmo: quando si deve applicare colore alle superfici (o meglio, ai triangoli di cui sono composte le varie superfici) è necessario calcolare il vettore normale alla superficie, ma per ottenere risultati corretti è necessario normalizzarlo, ovvero 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 integrate 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.

```
LDURS S5, [X1,#0] // carico nei registri le costanti
LDURS S15, [X2,#0]
LDURW X15, [X0, #0]

FMULS S10, S5, S0 // x=n*0.5F  (lasciamo y in S0 visto che n non ci serve piu')

SUBI SP, SP, #16  // allochiamo una quadword nello stack come da convenzione
STURS S0, [SP, #0]
LDURW X9, [SP, #0]  // i=*(long *) &y

LSL X9, X9, #1    // i >> 1
SUB X9, X15, X9   // i = 0x5f3759df - (i >> 1)

STURW X9, [SP, #0]
LDURS S0, [SP, #0]  // y= *(float *) &i
ADDI SP, SP, #16  // ripristiniamo lo stack

FMULS S1, S0, S0  // y*y
FMULS S1, S1, S10 // x*y*y
FSUBS S1, S15, S1 // 1.5F - x*y*y
FMULS S0, S1, S0  // y = y * (1.5F - x*y*y)
```

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.

```
double mediaAritmetica(double *arr, long long len){

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

}
```

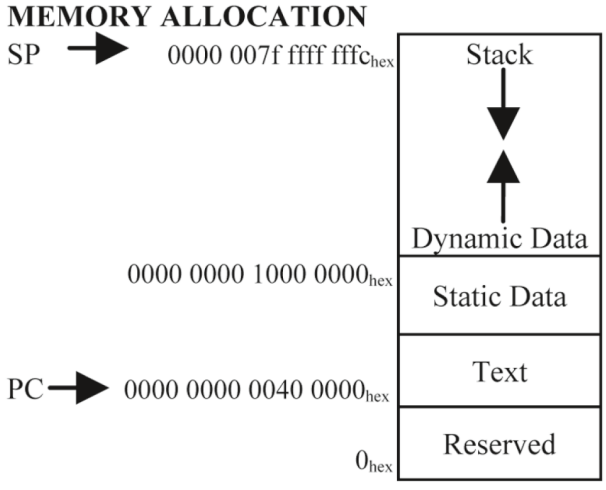
Bisogna prestare molta attenzione all'utilizzo delle costanti e della memoria. Si possono fare diverse scelte su cosa passare come parametro (o puntatore ad una posizione nello stack) oppure calcolarlo con altre variabili. Nel 5.1: media aritmetica di un array di double ho scelto di ricevere in input 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 (2.1: prima somma). 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 decidiamo di scrivere l'output in X0.

```
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 anche per allocare array facciamo 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[15] 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[16]. 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 }
```

[15] Il Link Register contiene l'indirizzo dell'istruzione del programma precedente alla quale l'esecuzione si era fermata prima di chiamare questa procedura

[16] 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.

```
/* swap
input: X0 -> &a
*/
swap: LDUR X9, [X0, #0]   // copyOfa[j]
      LDUR X10, [X0, #8]  // copyOfa[j+1]
      STUR X10, [X0, #0]  // a[j+1]= copyOfa[j]
      STUR X9, [X0, #8]   // a[j] = copyOfa[j+1]
      BR LR               // torno il controllo alla procedura chiamante

/* bubbleSort
input: X0 -> &arr   X1 -> len
*/
bubbleSort: SUBI SP, SP, #48    // alloco tre quadword nello stack come da convenzione ...
            STUR LR, [SP, #0]   // ... anche se devo memorizzare solo 5 doubleword
            STUR X19, [SP, #8]  // len
            STUR X20, [SP, #16] // i
            STUR X21, [SP, #24] // j
            STUR X22, [SP, #32] // &arr

            ADDI X20, XZR, #1   // i=1
            MOV X19, X1         // copio len e l'indirizzo dell'array in altri registri ...
            MOV X22, X0         // ... cosi' vengono preservati anche dopo le chiamate a swap
extFor:     CMPI X20, X19       // i-len
            B.GE exit

            SUBI X21, X20, #1   // j=i-1
intFor:     CMP X21, XZR        // j-0
            B.LT endExtFor

            LSL X9, X21, #3     // j*8
            ADDI X9, X9, X      // &arr[j]
            LDUR X10, [X0, #0]  // arr[j]
            LDUR X11, [X0, #8]  // arr[j+1]
            CMP X10, X11        // arr[j]-arr[j-1]
            B.GE endIntFor
            MOV X0, X9          // metto &arr[j] in X0 per swap
            BL swap           // chiamo swap

endIntFor:  SUBI X21, X21, #1       // j--
            B intFor

endExtFor:  ADDI X20, X20, #1       // i++
            B extFor

exit:       LDUR LR, [SP, #0]     // ripristino i 5 valori modificati nello stack
            LDUR X19, [SP, #8]
            LDUR X20, [SP, #16]
            LDUR X21, [SP, #24]
            LDUR X22, [SP, #32]
            ADDI SP, SP, #48    // ripristino lo stack
            BR LR       // "return void" (torno il controllo al chiamante)
```

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

```
long long fib(long long n){
  long long a = 0, b = 1, temp;
  for(n-=2; n>=0; n--){
    temp = a;
    a = b;
    b += temp;
  }
  return b;
```

```
}
```

Script 3: 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.

```
long long fib(long long n){
  long long f[n];
  f[0]=0;
  f[1]=1;
  for(long long i=2; i<=n; i++){
    f[i] = f[i-1] + f[i-2];
  }
  return f[n];
}
```

Script 4: 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

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 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).

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 quanto l'unità di controllo abilita la scrittura sul registro privato, ovvero solamente quando invochiamo istruzioni algebriche intere che terminano per S.
I quattro flag 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;

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;

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;

	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

```
fact(long long n){
  long long ris=1;
  while(n>1){
    ris*=n;
    n--;
  }
}
```

Script 5: Fattoriale iterativo

```
/*
input: X0 -> n
  X9 -> fact(n)_parziale
output: X0 -> fact(n)
*/

fact:   ADDI X9, XZR, #1  // ris=1

while:  CMPI X0, #1       // se(n-1)<=0
        B.LE end          // allora esci
        MUL X9, X9, X0    // ris*=n
        SUBI X0, X0, #1   // n--
        B while           // cicla sul while

end:    MOV X0, X9      // sposta il risultato in X0
```

Script 6: Fattoriale iterativo (assembly)

Una versione ricorsiva, invece, può essere la seguente

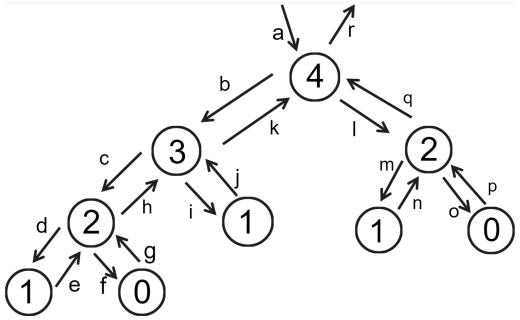
```
factRic(long long n){
  if(n<=1)
    return n;
  return n*factRic(n-1);
}
```

Script 7: Fattoriale ricorsivo

```
/*
input: X0 -> n
output: X0 -> factRic(n)
*/

factRic:  CMPI X0, #1        // if(n<=1)
          B.LE end           // esci (return n)

          SUBI SP, SP, #16   // preparo lo stack (a colpi di quadword) per memorizzare
          STUR X0, [SP, #0]  //  n
          STUR LR, [SP, #8]  //  LR
          SUBI X0, X0, #1    // preparo n-1 in X0
          BL factRic         // e chiamo factRic(n-1)

          LDUR X9, [SP, #0]  // ripristino n
          MUL X0, X0, X9     // factRic(n-1) * n
          LDUR LR, [SP, #8]  // ripristino LR
          ADDI SP, SP, #16   // ripristino SP
end:
```

Script 8: Fattoriale ricorsivo (assembly)

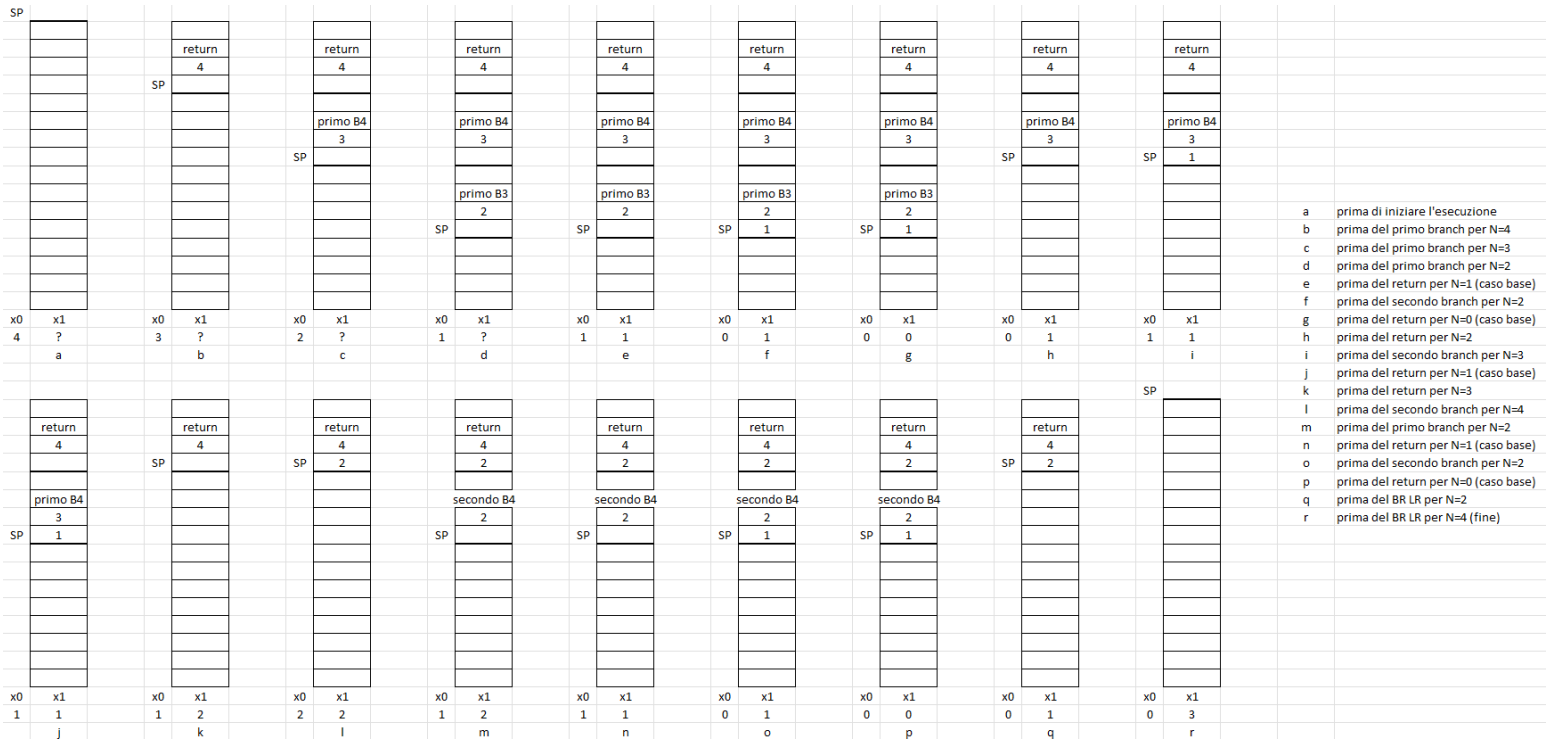
L'esecuzione di un programma ricorsivo può essere visualizzata come un cammino su un albero (un grafo non orientato privo di cicli). Esistono vari modi per passare da un ramo all'altro dell'albero, io ho scelto di ripristinare il valore di n prima di ritornare dal primo ramo; si poteva scegliere di ripristinarlo dopo il secondo ramo.



Vediamo ora la versione ricorsiva della sequenza di Fibonacci.

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

Script 9: Fibonacci ricorsivo



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

Script 10: Fibonacci ricorsivo

Alcune note su questo codice

- Nel caso base (quindi le prime quattro 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 14 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

```
long long fibArr[n];
fibArr[0]=0;
fibArr[1]=1;
for(int i=n-1; i>=2; i--)
  fibArr[i]=-1;
```

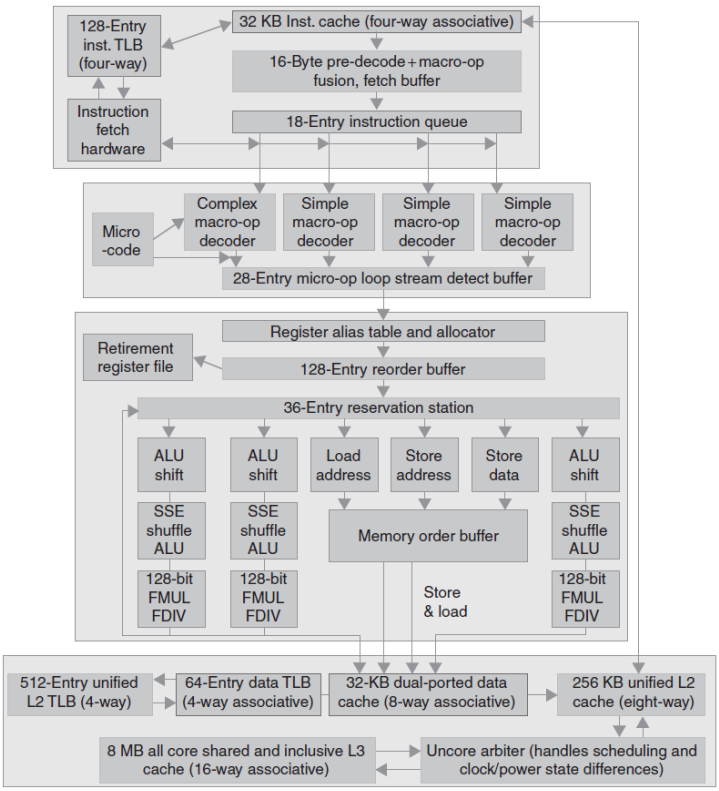
```
6
7  long long fib(int n, long long *fibArr){
8    if(fibArr[n]<0)
9      fibArr[n] = fib(n-1, fibArr) + fib(n-2, fibArr);
10   return fibArr[n];
11 }
```

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 $fibArr[n]$.

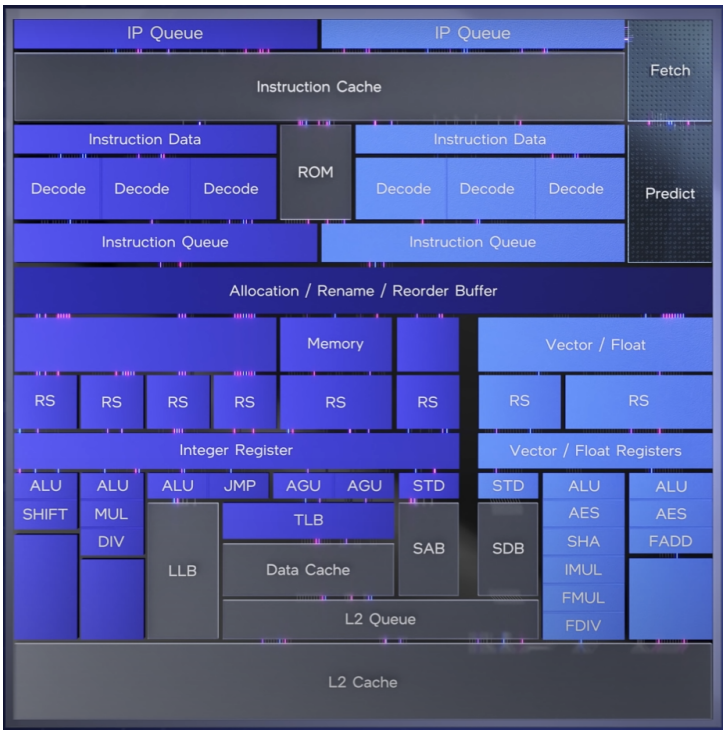
9 Commenti

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.



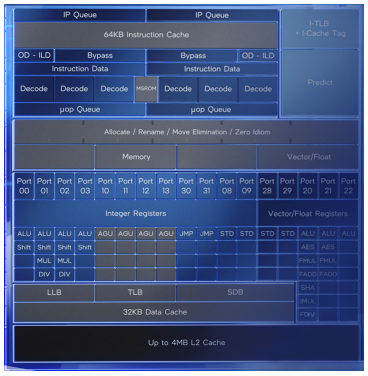
(a) Pipeline di un processore Intel Core i7 970



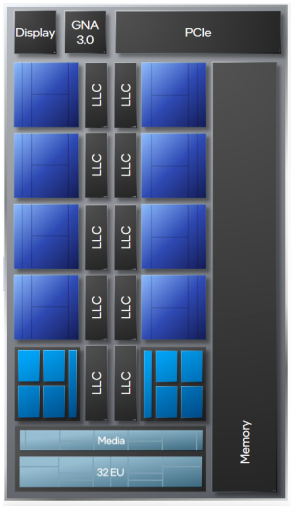
(b) Struttura di un processore Intel Core moderno

Figura 1: 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).

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



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



(c) Schematico di un processore Intel di dodicesima generazione

Figura 2: 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 backgroung, 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.

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.

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 nell'informatica (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 2n-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 assemblativo a programmi a livello ISA è detto Assembler o assemblatore. A ogni istruzione del linguaggio assemblatore 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 4: Una classificazione a sette livelli di astrazione nell'informatica

Bando alle ciancie, vediamo come si può programmare praticamente in Assembly. 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 su Internet l'architettura nel caso in cui aveste una versione precedente alla terza.

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 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[17]. 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 dottorato un'architettura nello specifico (per i più interessati).
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.

[17] 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

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

```
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
$-o\ NomeOut$	Specifica il nome del file di output	$gcc\ -o\ main\ main.c$
$-S$	Esegui solo il processo di compilazione, quindi crea solo il file assembly	$gcc\ -S\ main.c$
$-v$	Manda a schermo i risultati dei vari processi del compilatore	$gcc\ -v\ main.c$
$-Olvl$	Specifica il livello di ottimizzazione, dove *lvl* è un numero da 0 a 3, di default a 0.	$gcc\ -O3\ main.c$

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:

```
.global main

main:
  MOV X0, #2
  BR LR

```

La prima riga dice al compilatore che tutto ciò 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

```
gcc test.s
./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 [programma finito con esito positivo] con il comando *return 0;* alla fine del *main*) Se volessimo vedere anche il codice d'errore potremmo concatenare all'ultimo comando anche *echo $?*, quindi

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

Questo ci mostra il contenuto del registro $X0$, quindi 2.
Incominciamo a fare qualche programma un po' più interessante: scriviamo ora il codice per eseguire la somma misto mare del capitolo 2.

```
.global main

main:
    MOV X0, #2      // inizializzo a
    MOV X1, #5      // inizializzo b

    MOV X13, #3     // preparo 3
    MOV X14, #4     // preparo 4

    MUL X0, X0, X13 // 3*a
    MUL X1, X1, X14 // 4*b
    SDIV X9, X1, X13// 4b/3
    MUL X9, X9, X13 // (int(4b/3))*3
    SUB X1, X1, X9  // 4b%3
    ADD X0, X0, X1  // 3a+ ((4b)%3)
    BR LR

```

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 $X1 \rightarrow X7$ 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.

```
// sezione dei dati
.data
    res: .asciz "4*5=%d\n"  // asciz e' uno dei modi per dichiarare una stringa

// sezione del codice
.text
.global main
.extern printf

main:
    SUB SP, SP, #16 // alloco spazio nello stack
    STUR LR, [SP]   // memorizzo il link register
    MOV X1, #4      // preparo 4
    MOV X9, #5      // preparo 5
    MUL X1, X1, X9  // 4*5
    LDR X0, =res    // metto in x0 il puntatore alla stringa res
    BL printf       // printf("4*5=%d\n")

    LDUR LR, [SP]   // ripristino il link register
    ADD SP, SP, #16 // ripristino lo stack pointer
    BR LR           // esco dalla procedura

```

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, ma con una chiamata a funzione,

quindi scorporando lo swap in una funzione esterna.

```
.data
    txt: .asciz "la mediana e' %d\n"

.text
.global main
.extern printf

swap:
    LDUR X9, [X0]
    LDUR X10, [X1]
    STUR X9, [X1]
    STUR X10, [X0]
    BR LR


median3:  // calcola la mediana di 3 numeri
    SUB SP, SP, #32
    STUR LR, [SP, #24]
    STUR X0, [SP, #0]
    STUR X1, [SP, #8]
    STUR X2, [SP, #16]

    CMP X0, X2
    B.LE cond2
    MOV X0, SP
    ADD X1, SP, #16 // notate come non serve specificare ADDI in ARMv8
    BL swap
    LDUR X0, [SP]
    LDUR X1, [SP, #8]
    LDUR X2, [SP, #16]

cond2:
    CMP X0, X1
    B.LE cond3
    MOV X0, SP
    ADD X1, SP, #8
    BL swap
    LDUR X1, [SP, #8]

cond3:
    CMP X1, X2
    B.LE return
    ADD X0, SP, #8
    ADD X1, SP, #16
    BL swap

return:
    LDUR X0, [SP, #8]
    LDUR LR, [SP, #24]
    ADD SP, SP, #32
    BR LR


main: // utilizzo il main come funzione di test
    SUB SP, SP, #16
    MOV X0, #4
    MOV X1, #3
    MOV X2, #7
    STUR LR, [SP, #8]
    BL median3
    MOV X1, X0
```

```
    LDR X0, =txt
    BL printf
    LDUR LR, [SP, #8]
    ADD SP, SP, #16
    BR LR

```

Scriviamo ora un programma che somma due numeri a virgola mobile

```
.data
    res: .asciz "%f + %f = %f\n"
    num1: .double 0.2
    num2: .double 0.7

.text
.global main
.extern printf

main:
    SUB SP, SP, #16
    STUR LR, [SP]

    LDR X9, =num1   // per caricare un'etichetta bisogna utilizzare il load normale
    LDR X10, =num2
    LDUR D0, [X9]   // poi per leggere il valore si utilizza un load unscaled
    LDUR D1, [X10]
    FADD D2, D0, D1

    LDR X0, =res
    BL printf

    LDUR LR, [SP]
    ADD SP, SP, #16
    BR LR

```

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

```
.data
    arr: .skip 32
    uno: .double 1.0
    mezzo: .double .5
    str: .asciz "La media e' %lf\n"

.text
.global main
.extern printf

media:
    LDR X15, =uno // carico il puntatore ad 1.0
    LDUR D15, [X15] // carico 1.0
    FSUB D0, D15, D15 // azzero A
    FSUB D1, D15, D15 // azzero  (double)i che mi servira' per la divisione finale
    LSL X1, X1, #3  // n*8
    ADD X1, X1, X0
loop: CMP X0, X1  // i<n
    B.GE exit
    LDUR D2, [X0] // *arr[i]
    FADD D0, D0, D2 // A+= arr[i]
    FADD D1, D1, D15  // ((double)i)++
    ADD X0, X0, #8  // i++
    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  .gloabl main
6  .extern printf
7
8  strMaiusc:
9      LDR X10, =str
10 loop:
11     LDURB W9, [X10]
12     CBZ W9, [X10]
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 proccessore 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.

11 Ulteriori esercizi

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

```
long long strToInt(char *str){
  long long ris = 0, mul=1, i=0;
  if(str[0] == '-'){
    i++;
    mul=-1;
  }else if(str[0] == '+')
    i=1;
  while(str[i]!='\0'){
    if(str[i]<48 || str[i] > 57)
      return -1;
    ris*=10;
    ris += (long long) (str[i]-48);
    i++;
  }
  ris*=mul;
  return ris;
}
```

Script 11: Conversione di una stringa in un intero

```
void convolution(double *x, double *h, double *y, long long n, long long k){
  long long start, i, j;
  double temp;
```

```
  for(i=0; i<n; i++){
    start = i - k/2;
    temp=0;
    for(j=0; j<k; j++){
      if(start+j>=0 && start+j<n)
      temp += x[start+j] * h[j];
    }
    y[i] = temp;
  }
}
```

Script 12: Convoluzione di un array generico di double con un kernel di dimensione non specificata

```
void countingSort(unsigned long long *arr, unsigned long long len, unsigned long long max){

  unsigned long long counter[max];
  unsigned long long i,j;

  for(j=0; j<max; j++)  //resettiamo l'array
  counter[j]=0;

  for(i=0; i<len; i++)  // contiamo quanti sono i numeri da 0 a max
  counter[arr[i]]++;

  i=0, j=0;
  while(i<len){     // sistemiamo nell'array originale tutti i numeri da 0 a max
    while(counter[j]){  // finche' sono presenti (ovvero finche' counter[j]>0)
      arr[i] = j;
      counter[j]--;
      i++;
    }
    j++;
  }
}
```

Script 13: Algoritmo di ordinamento CountingSort

Nelle future revisioni aggiungerò ulteriori esercizi

12 Soluzione esercizi

```
input:
X0 -> a
X1 -> b
X2 -> c

output:
X0 -> somma

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

Script 14: 2.1.1: somma di tre numeri

```
/*
input: X0 -> a     X1 -> b
output: X0 -> 3a+(4b)%3
*/
```

```
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
```

Script 15: 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
```

Script 16: 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 X0, X0, X2
```

Script 17: 5.1: 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
```

Script 18: Fibonacci iterativo

```
1   /*
2   input: X0 -> n
3     X1 -> fib(n)     X9 -> i    X13 -> nOriginale    X14 -> *fibArr
4   */
5
6           ADD X13, X0, XZR  // salvo il valore originale di n
7           LSL X9, X0, #3    // preparo sia il # di byte per lo stack che per il loop
8           SUB SP, SP, X9    // alloco nello stack lo spazio per fibArr
9           MOV X14, SP   // *fibArr = SP
10
11          SUBI X11, XZR, #1 // memorizzo -1
12          STUR XZR, [X14,#0]  // fibArr[0]=0
13          ADDI X15, XZR, #1 // memorizzo 1
14          STUR X15, [X14,#8]  // fibArr[1]=1
15
16  init:   SUBIS X9, X9, #2  // se i<=2
17          B.LE fib        // esci dal loop
18          ADD X10, X14, X9  // altrimenti posizionati in fibArr[i]
19          STUR X11, [X10,#0]  // e memorizza -1, quindi fibArr[i] = -1
20          SUBI X9, X9, #8 // i--
21          B init
22
23  fib:    LSL X9, X0, #3    // n*8
24          ADD X9, X9, X14 // *fibArr[n]
25          LDUR X1, [X9, #0] // fibArr[n]
26          SUBS X1, X1, XZR  // se fibArr[n]>=0
27          B.GE finish   // ritorna il valore che hai trovato
28
29          SUBI SP, SP, #24  // altrimenti prepara lo stackpointer per 3 elementi
30          LDUR LR, [SP, #8] // carica link register
31          LDUR X0, [SP, #0] // carica n
32          SUBI X0, X0, #1 // n-1
33          BL fib        // calcola fib(n-1)
34          STUR X1, [SP, #16]  // salva nello stack fib(n-1)
35          LDUR X0, [SP, #0] // ripristina il valore di n
36          SUBI X0, X0, #2 // n-2
37          BL fib        // calcola fib(n-2)
38          LDUR LR, [SP, #8] // ripristina il link register
39          LDUR X12, [SP, #16] // ripristina fib(n-1)
40          LDUR X0, [SP, #0] // ripristina il valore di n (serve per controllo finale
41          ADD X1, X12, X1 // fib(n)= fib(n-1) + fib(n-2)
42          STUR X1, [X9, #0] // fibArr[n] = fib(n)
43          ADDI SP, SP, #24  // ripristina lo stack
44
45  finish: SUBS XZR, X13, X0 // verifica se abbiamo appena calcolato l'n-esimo numero
46          B.NE return   // se si', svuota lo stack (fai quello che c'e' qui sotto)
47
48          LSL X9, X13, #3 // n*8
49          ADD SP, SP, X9    // "svuoto" l'array dallo stack
50
```

```
return: BR LR
```

Script 19: Fibonacci iterativo con memoria

```
/*
input: X0 -> n
  X1 -> fib(n)  X9 -> i   X13 -> nOriginale   X14 -> *fibArr
output: X0 -> fib(n)
*/

fibRicMem:                  // dichiariamo ed inizializziamo fibArr[n]
      ADD X13, X0, XZR      // salvo il valore originale di n
      LSL X9, X0, #3        // preparo il # di byte sia per lo stack che per il loop
      SUB SP, SP, X9        // alloco nello stack lo spazio per fibArr
      MOV X14, SP           // *fibArr = SP

      SUBI X11, XZR, #1     // memorizzo -1
      ADDI X15, XZR, #1     // memorizzo 1
      STUR XZR, [X14,#0]    // fibArr[0]=0
      STUR X15, [X14,#8]    // fibArr[1]=1

init: SUBI X9, X9, #8       // i-- (in X9 c'e' n, ma dobbiamo riempire da n-1 a 2)
      SUBIS X9, X9, #16     // se i<2
      B.LT fib              // esci dal loop
      ADD X10, X14, X9      // altrimenti posizionati in fibArr[i]
      STUR X11, [X10,#0]    // e memorizza -1, quindi fibArr[i] = -1
      B init

fib:  LSL X9, X0, #3        // n*8
      ADD X9, X9, X14       // *fibArr[n]
      LDUR X1, [X9, #0]     // fibArr[n]
      SUBS X1, X1, XZR      // se fibArr[n]>=0
      B.GE finish           // ritorna il valore che hai trovato

      SUBI SP, SP, #24      // altrimenti prepara lo stackpointer per 3 elementi
      LDUR LR, [SP, #8]     // carica link register
      LDUR X0, [SP, #0]     // carica n
      SUBI X0, X0, #1       // n-1
      BL fib                // calcola fib(n-1)
      STUR X1, [SP, #16]    // salva nello stack fib(n-1)
                            // n-2 (sta volta ripristino dopo n-2, quindi utilizzo l'altro
      SUBI X0, X0, #1         // metodo esplorativo dell'albero)
      BL fib                // calcola fib(n-2)
      LDUR X12, [SP, #16]   // ripristina fib(n-1)
      ADD X1, X12, X1       // fib(n)= fib(n-1) + fib(n-2)
      STUR X1, [X9, #0]     // fibArr[n] = fib(n) (ricordiamoci di salvare fib(n) in memoria,
                              // altrimenti tutta la fatica di utilizzare l'array e' vana)
      LDUR LR, [SP, #8]     // ripristina il link register
      LDUR X0, [SP, #0]     // ripristina il valore di n (serve per controllo finale)
      ADDI SP, SP, #24      // ripristina lo stack

end:  SUBS XZR, X13, X0     // verifica se abbiamo appena calcolato l'n-esimo numero
      B.NE exit             // se si', svuota lo stack (ovvero fai le due righe qui sotto)

      LSL X9, X13, #3       // n*8
      ADD SP, SP, X9        // "svuoto" l'array dallo stack

exit: BR LR                 // questo return viene eseguito sempre, mentre le precedenti due
                              //righe solo all'ultima esecuzione
```

Script 20: Fibonacci ricorsivo con memoria

```
/*
input:  X0 -> *str
  X9 -> ris    X10 -> i     X11 -> str[i]    X12 -> 10    X13 -> mul
  '0' = 48     '9' = 57     '\0' = 0     '-' = 45     '+' = 43
output: X0 -> ris
*/
        LDURB X11, [X0,#0]  // str[0]
        ADDI X12, XZR, #10  // salvo 10 x la moltiplicazione
        ADDI X10, XZR, #0 // i=0
        MOV X9, XZR
        ADDI X13, XZR, #1
        SUBIS XZR, X11, #45// controlla se c'e' il segno
        B.NE piu

        SUBI X13, XZR, #1
meno:   ADDI X10, X10, #1 // i++ 1 byte perche' stiamo lavorando con char

piu:    SUBIS XZR, X11, #43 // se non e' '-' allora verifica che non sia '+'
        B.NE loop
        ADDI X10, X10, #1 //1 byte in quanto stiamo lavorando con char

loop:   LDURB X11, [X0, #0] // str[i]
        SUBIS XZR, X11, #0  // str[i] ?= '\0'
        B.EQ return
        SUBIS XZR, X11, #48 // str[i] ?< '0'
        B.LT retErr
        SUBIS XZR, X11, #57 // str[i] ?> '9'
        B.GT retErr
        BR LR

        MUL X9, X9, X12 // ris*=10
        SUBI X11, X11, #48  // str[i]-48
        ADD X9, X9, X11 // ris += str[i]-48
        ADDI X10, X10, #1 // i++
        BL loop

retErr: SUBI X0, XZR, #1  // se arriviamo qui significa che e' stato trovato un carattere
        BR LR     // diverso da un numero, quindi ritorniamo -1 come da testo

return: MUL X0, X11, X13  // se siamo arrivati qui allora abbiamo trovato '\0', ovvero
        BR LR     // la fine della stringa e quindi ritorniamo il numero calcolato
```

Script 21: Conversione di una stringa in un intero

```
/*
input:  X0 -> x    X1 -> h    X2 -> y    X3 -> n
      X4 -> k    X5 -> &(double)0
X9 -> i    X10 -> j     X11 -> start
D9 -> temp     D10 -> x[start+j]    D11 -> h[j]
*/
          MOV X9, XZR      // i=0
          LDURD D31, [X5, #0]   // load (
    double)0
extloop:  CMP X9, X3       // i-n
          B.GE exit
          LSR X11, X4, #1   // k/2
          SUB X11, X9, X11     // i-k/2
          FADDD D9, D31, D31     // temp=0

          MOV X10, XZR       // j=0
intloop:  CMP X10, X4      // j-k
          B.GE fineconv

          ADDS X12, X11, X10     // start+j
          B.LT intloop
          CMP X12, X3      // start+j<n
          B.GE intloop
          LSL X12, X12, #3     // (start+j)*8
          ADD X12, X12, X0
          LDURD D10, [X12, #0]  // x[start+j]
          LSL X13, X10, #3
          ADD X13, X13, X1
          LDURD D11, [X13, #0]  // h[j]
          FMULD D10, D10, D11   // x[start+j]
    * h[j]
          FADDD D9, D9, D10   // temp += x[
    start+j] * h[j]
          ADDI X10, X10, #1
          B intloop

fineconv: LSL X13, X9, #3   // i*8
          ADD X13, X13, X2     // y[i]
          STURD D9, [X13, #0]   // y[i] = temp
          ADDI X9, X9, #1
          B extloop

exit:     BR LR
```

Script 22: Convoluzione di un array generico con un kernel di dimensione non specificata

```
/*
input:  X0 -> x    X1 -> h    X2 -> y    X3 -> n
      X4 -> k
X9 -> i    X10 -> j     X11 -> start
D9 -> temp     D10 -> x[start+j]    D11 -> h[j]
*/
          MOV X9, XZR      // i=0
          LDURD D31, [X5, #0]   // load (
    double)0
extloop:  CMP X9, X3       // i-n
          B.GE exit
          LSR X11, X4, #1   // k/2
          SUB X11, X9, X11     // i-k/2
          FADDD D9, D31, D31     // temp=0

          MOV X10, XZR       // j=0
intloop:  CMP X10, X4      // j-k
          B.GE fineconv

          ADDS X12, X11, X10     // start+j
          B.LT intloop
          CMP X12, X3      // start+j<n
          B.GE intloop
          LSL X12, X12, #3     // (start+j)*8
          ADD X12, X12, X0
          LDURD D10, [X12, #0]  // x[start+j]
          LSL X13, X10, #3
          ADD X13, X13, X1
          LDURD D11, [X13, #0]  // h[j]
          FMULD D10, D10, D11   // x[start+j]
    * h[j]
          FADDD D9, D9, D10   // temp += x[
    start+j] * h[j]
          ADDI X10, X10, #1
          B intloop

fineconv: STURD D9, [X2, #0]     // *y = temp
          ADDI X2, X2, #8   // y++
    incremento ed uso y per puntatore
          ADDI X9, X9, #1
          B extloop

exit:     BR LR
```

Script 23: Convoluzione di un array letto per puntatore

```
  /*
  input:  X0 -> &arr     X1 -> len    X2 -> max
  X9 -> i    X10 -> j
  */

CountingSort:
        MOV X9, XZR    //i=0
        MOV X10, XZR     //j=0
        LSL X15, X2, 3     //X15=max*8
        SUB SP, SP, X15    //riduco lo SP per contenere counter[max]
for1:   CMP X10, X2    //se j>=max esco dal for
        B.GE for2
        LSL X11, X10, 3    //X11=j*8
        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
```

Script 24: CountingSort