

Assembler ARM con tool GNU

Marco Danelutto

October 26, 2021

1 Workflow

Per compilare ed eseguire un programma scritto in Assembler ARM con gli strumenti GNU si devono sostanzialmente eseguire una serie di passi, secondo un workflow come quello descritto in Figura 1. Nel seguito useremo l'acronimo `RBARM` per caratterizzare le parti e gli strumenti che fanno riferimento all'utilizzo di tool GNU su architettura con processore ARM (e.g. Raspberry, Odroid, etc) e l'acronimo `CCARM` per caratterizzare le parti e gli strumenti che fanno riferimento all'utilizzo della toolchain cross compiler di GNU sotto LINUX. Come spiegato a lezione, gli strumenti `CCARM` si possono utilizzare sia sotto Linux che sotto Windows in maniera relativamente semplice:

- sotto Linux, è sufficiente installare quattro pacchetti:
`gcc-arm-linux-gnueabi`, `qemu` e `qemu-user` e infine `gdb-multiarch`.
Per installare i pacchetti, da prompt di shell root va digitato il comando
`apt install gcc-arm-linux-gnueabi qemu qemu-user gdb-multiarch`¹.
- sotto Windows, è necessario attivare l'ambiente WSL2 (istruzioni al link <https://docs.microsoft.com/it-it/windows/wsl/install-win10>) e successivamente l'app "Linux Ubuntu 20.04" dall'app store Microsoft. A quel punto, lanciando l'applicazione "ubuntu" si ottiene un terminale con una shell `bash` nel quale si possono installare i pacchetti `gcc-arm-linux-gnueabi`, `qemu` e `gdb-multiarch` utilizzando la stessa procedura seguita in Linux.

Per quanto riguarda Mac OS/X, la cosa più semplice è quella di installare una macchina virtuale Ubuntu utilizzando Virtual Box. E' sufficiente installare la versione base e aggiungere i pacchetti menzionati qui sopra.

1.1 Scrittura del programma

Il programma viene scritto, normalmente in un file `.s` (detto *file sorgente*), utilizzando un normale editore di testo in grado di salvare in puro formato ASCII (text only). Nello scrivere il programma occorre rispettare la sintassi GNU che differisce dalla sintassi ARM ufficiale per poche ma importanti cose:

¹assumo che abbiate una distribuzione Linux Ubuntu

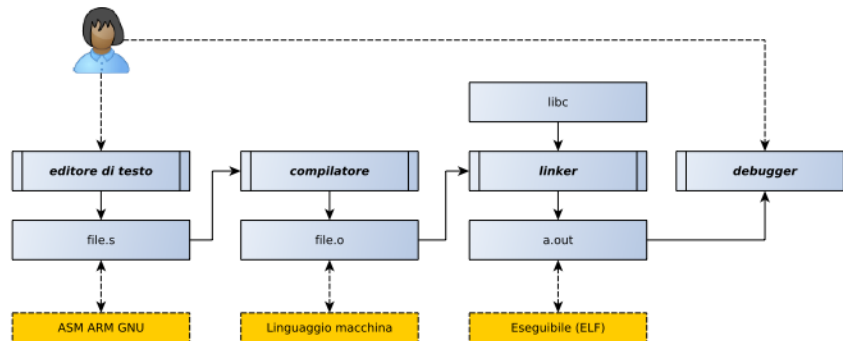


Figure 1: Workflow per la realizzazione di un programma in Assembler ARM con i tool GNU

- le etichette sono seguite dal carattere “:” (in sintassi ARM non vanno messi i due punti);
- le direttive (vedi Sez. 2) sono tutte sostanzialmente diverse come sintassi.

1.2 Compilazione

Per produrre un binario occorre compilare il file sorgente. Questo può essere fatto

- o utilizzano `gcc` direttamente, con l’opzione `-c`, oppure chiamando direttamente l’assemblatore (comando `as`) ;
- In caso di utilizzo dei tool cross-compiler i due comandi si chiamano rispettivamente:

`arm-linux-gnueabi-gcc` e `arm-linux-gnueabi-as`.

L’output nei due casi è un file `.o` (con `gcc -c`) o `a.out` (con `as`) che è detto *file oggetto*. Per essere eseguito deve essere processato mediante il linker che provvede a collegare le librerie necessarie e ad aggiungere quanto serve a far eseguire il `main`.

1.3 Linking

Per collegare le librerie necessarie e rendere eseguibile il file oggetto è necessario invocare il linker passandogli come parametri tutti i file oggetto da includere nell’eseguibile. Il linker GNU è invocato utilizzando di nuovo `gcc` tipicamente con il parametro `-o nomeeseguibile`. Questo avviene invocando:

- `gcc file.o [file2.o ...] -o nomeeseguibile;`

- `arm-linux-gnueabi-hf-gcc file.o [file2.o ...] -static -o nomeeseguibile`

CCARM

L'opzione `-static` serve per includere le funzioni di libreria utilizzare direttamente nel codice. Dal momento che le librerie sorgenti ARM non si possono trovare nel posto dove stanno di solito le librerie (lì ci saranno quelle compilate in assembler x86), al momento dell'esecuzione `qemu-arm` non sarebbe in grado di risolvere i riferimenti dinamici.

L'output del linker è un file `nomeeseguibile.o`, in assenza del `-o` che ne specifica il nome, il file `a.out`. Questo è detto *file eseguibile* o semplicemente *eseguibile*.

1.3.1 Compilazione e linking (non separati)

Naturalmente è possibile compilare e linkare il proprio codice utilizzando un unico comando, che è sempre `gcc` (RBARM) o `arm-linux-gnueabi-hf-gcc` (CCARM). Di fatto il comando `gcc` è una specie di vera e propria *shell di compilazione* che a seconda dei parametri e dei tipi di file passati come argomento effettua diverse funzioni che vanno dalla semplice compilazione alla generazione di un eseguibile completo (azione di default). Per compilare un programma assembler nel file `myProg.s` e generare un eseguibile `myProg.exe` possiamo utilizzare i comandi:

- `gcc -o myProg.exe myProg.s`, oppure
- `arm-linux-gnueabi-hf-gcc -o myProg.exe myProg.s`

RBARM

CCARM

Qualora il codice in `myProg.s` chiamasse funzioni definite in altri file (e.g. `fun.s` e `util.s`) si può elencare tutti i file sorgente nella lista di parametri del comando `gcc`²:

- `gcc -o myProg.exe myProg.s fun.s util.s`, oppure
- `arm-linux-gnueabi-hf-gcc -o myProg.exe myProg.s fun.s util.s`

RBARM

CCARM

Notare che questo ha lo stesso effetto dell'utilizzo della compilazione separata e del linking eseguito successivamente, come avviene nella sequenza di comandi (RBARM, per CCARM vanno cambiati come al solito i nomi dei comandi `gcc` in `arm-linux-gnueabi-hf-gcc` e nel comando che effettua il linking (l'ultimo della serie qui sotto) va aggiunta l'opzione `-static` per evitare problemi con le librerie dinamiche quando eseguiremo il programma con una `qemu-arm myProg.exe`):

```
gcc -c myProg.s
gcc -c fun.s
gcc -c util.s
gcc -o myProg.s myProg.o fun.o util.o
```

²come avviene per la compilazione di programmi distribuiti in più file sorgenti in C

1.4 Esecuzione

Per eseguire il programma eseguibile, da riga di comando se ne può digitare il nome. In caso di utilizzo degli strumenti cross-compiler, occorre utilizzare il comando `qemu-arm` seguito dal nome dell'eseguibile e dagli eventuali parametri.

- se avete compilato l'eseguibile su una macchina dotata di processore arm e tool GNU con un comando `gcc -o myProg ...` potete eseguire il programma con `./myProg` seguito dagli eventuali parametri di riga di comando; RBARM
- se avete usato il crosscompiler (`arm-linux-gnueabi-gcc -o myProg ...`) potete eseguire il programma con `qemu-arm myProg` seguito dagli eventuali parametri di riga di comando. Qualora si verificasse un problema di caricamento delle librerie, ricompilate l'eseguibile utilizzando anche il flag `-static`:
`gcc -o myProg -static ...` CCARM

1.5 Debugging

Il debugger GNU si chiama `gdb`. La documentazione completa del debugger la trovate su <https://sourceware.org/gdb/current/onlinedocs/gdb/>.

Nel caso utilizzate una macchina con processore ARM, per eseguire il debugger basta: RBARM

- compilare con l'opzione `-g` (necessaria per mantenere nell'eseguibile la tabella dei simboli (etichette) del programma)
- eseguire il debugger con il comando `gdb myProg`.

Se invece state utilizzando il cross compiler la procedura è leggermente più complessa. CCARM

- compilate il programma con l'opzione `-ggdb3`
- lanciate il programma in modalità debug con il comando
`qemu-arm -g 12345 myProg &`
La `&` a fine riga serve per mandare il programma in background, senza dover aprire un altro terminale
- aprite il debugger `gdb-multiarch` con i parametri
`gdb-multiarch -q --nh -ex 'set architecture arm'`
`-ex 'file myProg' -ex 'target remote localhost:12345'`
Il numero di porta per il debugger può essere un numero qualunque di una porta effimera (da 1K a 32K-1) purchè libera. Deve essere utilizzato identico nel comando `qemu-arm -g` e nel comando che lancia il debugger dopo `localhost:`. Analogamente, nell'opzione `-ex 'file` va indicato il nome del programma eseguibile che è stato lanciato mediante il comando `qemu-arm -g`

2 Direttive

Il programma assembler accetta istruzioni dell'assembler ARM e *direttive* ovvero istruzioni che non sono vere e proprie istruzioni assembler ma che servono per definire le varie sezioni del codice e/o eventuali aree di memoria riservate per i dati del programma³.

Le direttive di interesse sono queste:

- **.data**
definisce una sezione di dati, ovvero con sole direttive che riservano aree di memoria. All'interno della sezione **.data** tipicamente troviamo direttive:
 - **.word**
seguita da valori separati da virgole, che riserva un'area di memoria di tante parole quanti sono i valori che seguono, inizializzata con i valori che seguono la **.word**;
 - **.byte**
seguita da valori separati da virgole, che riserva un'area di memoria di tanti byte quanti sono i valori che seguono, inizializzata con i valori che seguono la **.byte**;
 - **.fill**
seguita da un numero intero (che deve essere un multiplo di 4), che riserva un'area di memoria di tanti byte quanto vale il parametro intero;
 - **.string**
seguita da una stringa fra virgolette, che riserva un'area di memoria sufficiente a contenere i caratteri della stringa seguiti da un carattere con codice 0.

Tutte queste direttive sono normalmente precedute da un'etichetta, che può essere utilizzata per reperirne l'indirizzo base.

- **.text**
definisce un'area di codice. All'interno della sezione troviamo normalmente una o più direttive
 - **.global <etichetta>**
che denotano i simboli che devono essere resi noti al debugger⁴

nonchè istruzioni assembler che costituiscono il codice vero e proprio.

³In realtà ci sono anche direttive che servono per la compilazione condizionale, per le macro, etc. ma noi non le abbiamo mai utilizzate nel corso

⁴più in generale resi pubblici in fase di linking

2.0.1 Esempio di direttive

```
1  .data                @ definisce una sezione dati
2
3  a:  .word 1,2,3,4    @ definisce un vettore di 4 parole
4                        @ che contengono i valori da 1 a 4
5                        @ e il cui indirizzo base e' a
6
7  b:  .fill 32          @ riserva un area di memoria da 8 parole
8                        @ (32 = 8 * 4 byte) il cui indirizzo base e' b
9
10 c:  .string "ciao"    @ riserva 5 byte per contenere i caratteri
11                        @ della stringa in codice ASCII e il
12                        @ carattere NULL (codice 0) di fine stringa
13
14  .text                @ definisce una sezione che contiene codice
15  .global main         @ il simbolo main viene esportato nella
16                        @ tabella dei simboli da utilizzare fuori
17  .global mdfun        @ simbolo mdfun esportato
18
19 main: mov r0, #0
20       ldr r1, =d
21       ldr r1,[r1]
22       b cont
23 d:    .word 12345      @ esempio di costante in mezzo al codice
24                        @ va bene, basta evitare di "passarci"
25                        @ in esecuzione (va messa ad un indirizzo
26                        @ non "raggiunto" dal PC ...) come qui
27 cont: add r1, r1, #1
```

3 Chiamate di libreria

Il linker invocato mediante `gcc` permette di utilizzare alcune funzioni di libreria utili per gestire diverse funzionalità che in assembler puro sarebbero molto difficili da programmare. Tipicamente, `printf`, `gets`⁵, `scanf`, `atoi`, etc.

Per le chiamate alla `libc` valgono le solite convenzioni dell'assembler ARM:

- i parametri in ingresso sono passati mediante R0, R1, R2, R3 e, se in numero maggiore a 4, mediante lo stack;
- il parametro di ritorno è restituito in R0;
- i registri R0-R3 non sono preservati, quelli da R4 in sù sì.

Dunque per chiamare una `printf` dovremmo utilizzare:

- R0 per contenere l'indirizzo della stringa di formato
- R1 per contenere il primo parametro (quello che va nel primo campo %), se presente

⁵da utilizzare con tutte le cautele del caso e solo per i piccoli programmi realizzati come esercizi per il corso

- R2 per contenere il secondo parametro (quello che va nel secondo campo %), se presente
- R3 per contenere il terzo parametro (quello che va nel terzo campo %), se presente
- altri parametri (se presenti) sullo stack.

Per esempio, il codice che segue chiama una `printf`:

```

1  .data
2  ...
3 f: .string "Primo intero : %d secondo : %d \n"
4  ...
5  .text
6  ...
7  ldr R0, =f
8  mov R1, ...
9  mov R2, ...
10 bl printf
11 mov ..., R0

```

Al termine della `bl` il registro `R0` contiene il numero di caratteri effettivamente stampati.

3.1 Funzioni glibc

Le funzioni messe a disposizione dalla `glibc` sono quelle che trovate alla pagina https://www.gnu.org/software/libc/manual/html_node/Function-Index.html. Nella tabella che segue ve ne riporto alcune che potrebbero essere utili da richiamare dal codice assembler.

Nome	Funzione
<code>atoi, atof</code>	conversione di stringhe in interi e float
<code>malloc, calloc</code>	allocazione di memoria sullo heap
<code>rand, srand</code>	generazione numeri pseudocasuali
<code>getc</code>	legge caratteri dallo standard input
<code>isalnum, isalpha, isblank</code>	testa tipo caratteri
<code>memcpy</code>	copia aree di memoria
<code>memset</code>	inizializza aree di memoria
<code>printf, scanf</code>	input/output
<code>strcmp, strcpy, strlen</code>	manipolazione stringhe

3.2 Malloc

Due parole in particolare per allocare memoria senza ricorrere a direttive tipo `.fill`, ovvero dinamicamente. Possiamo chiamare la `malloc` della `libc` per questo scopo. La `malloc` prende un solo parametro (dimensione in bytes della memoria da allocare). Per allocare un certo numero di parole conviene dunque caricare il numero delle parole da allocare in un registro e successivamente moltiplicare il registro per 4 utilizzando una `LSL Rx, Rx, #2`. Il valore di ritorno

della `malloc` sarà l'indirizzo base dell'area di memoria allocata. In caso di errore il valore restituito sarà `NULL` (ovvero 0). Il risultato della `malloc` andrà testato *sempre* prima di utilizzare l'area di memoria ottenuta, pena un errore di protezione.

Il codice che segue fa vedere come allocare un segmento di memoria di 8 parole in cui andiamo a mettere tutti valori 7.

```

1  .data
2  err:  .string "Errore nella malloc\n"
3  succ: .string "Malloc ha avuto successo (ind = %d)\n"
4
5  .text
6  .global main
7
8  main: mov r0, #8           @ 8 parole
9        lsl r0, r0, #2       @ *4 perche' sono byte
10       bl malloc            @ chiama la malloc
11       cmp r0, #0           @ controlla se ret = NULL
12       beq error            @ in caso segnala errore
13       mov r1, r0           @ salva indirizzo mem allocata
14       mov r0, #0           @ i = 0
15       mov r2, #7           @ valore da utilizzare per scrivere
16 loop: str r2, [r1,r0]      @ x[i] = 7
17       add r0, r0, #4       @ i++ (in byte i+=4)
18       cmp r0, #32          @ siamo alla fine
19       beq fine             @ se si esci
20       b loop               @ altrimenti for i+1
21 error: ldr r0, =err         @ in caso di errore
22       bl printf            @ stampa messaggio ed esci
23       mov r7, #1           @ exit
24       svc 0
25 fine:  ldr r0, =succ        @ stampa messaggio di successo
26       push {r1}            @ ind serve per la free, va salvato
27       bl printf            @ puo' sporcare r0-r3
28       pop {r1}             @ ripristino ind -> r1 dallo stack
29       mov r0, r1           @ chiama la free: ptr -> r0
30       bl free              @ free(memoria allocata)
31       mov r7, #1           @ exit
32       svc 0

```

Dopo l'etichetta `fine` si stampa un messaggio di successo che indica quale indirizzo è stato restituito. In seguito chiamiamo una `free` per liberare la memoria allocata. Poichè l'indirizzo da liberare era in `r1` prima della call alla `printf` va salvato o in un registro alto o sullo stack, dal momento che i registri da `r0` a `r3` non sono mantenuti nelle chiamate (vedi Sec. 3). In questo caso salviamo il contenuto di `r1` sullo stack prima di passarlo alla `printf` (`push` alla riga 26) e poi lo recuperiamo dallo stack dopo il ritorno dalla `printf` (`pop` alla riga 28).

4 Single step execution

Riassumiamo i principali comandi che permettono di eseguire un programma step by step (una istruzione alla volta) utilizzando il debugger.

Una volta lanciato il debugger, potete cominciare mettendo un **breakpoint** al punto in cui volete cominciare a osservare l'esecuzione del programma step by step. Tipicamente, se volete osservare l'effetto delle istruzioni a partire dal **main**, al prompt date un comando **break main** (o, abbreviato, **b main**). Facendo partire il programma con il comando **run** (o, abbreviato, **r**) l'esecuzione si ferma immediatamente prima dell'esecuzione della istruzione etichettata con **main**. **Attenzione:** se si utilizzano gli strumenti crosscompiler, quando si fa partire il debugger multiarch l'esecuzione è già partita e dunque va dato un comando **continue** invece che **run** per procedere nell'esecuzione del programma, dopo l'eventuale **break main**. A questo punto:

- **tui reg general**
fa vedere nel terminale nella parte alta il valore dei registri general, nella parte bassa il prompt di **gdb** e nella parte intermedia il codice assembler con l'istruzione che sta per essere eseguita in evidenza:

```

Register group: general
r0      0x1          1
r1      0xffffd2a4   -77148
r2      0xffffd2ac   -77140
r3      0x10424      66596
r4      0x0          0
r5      0x0          0

B+>8    main:  mov r0, #0      @ descrittore stdin (0 stdin, 1 stdout,
9        ldr r1, =strbuf @ indirizzo del buffer (&buffer)
10       mov r2, #1      @ lunghezza della stringa (non posso ut
11       lsl r2, r2, #9   @ di 8 bit per la costante, dunque uso
12
13       mov r7, #3      @ la syscall read ^^C^ la numero 3 (ved

remote Thread 1.18592 In: main          L8    PC: 0x10424
unknown register group 'general'
(gdb)

```

Per vedere i registri in virgola mobile, si può indicare **float** al posto di **general**. **all** fa vedere invece tutti i registri (general, floating point, di controllo, ...);

- **next**
(abbreviato **n**) permette di eseguire la prossima istruzione. Nel caso sia una chiamata di funzione/procedura, la funzione/procedura viene eseguita per intero e il debugger si ferma alla prossima istruzione (quella che segue la **bl**);
- **step**
(abbreviato **s**) permette di eseguire la prossima istruzione. Qualora l'istruzione sia una **bl** il debugger entra nella funzione/procedura chiamata e si ferma alla prima istruzione della funzione;
- **continue**
(abbreviata **c**) permette di continuare senza single step fino al prossimo

breakpoint.

Quest'opzione deve essere utilizzata quando si lavora con `gdb-multiarch` invece dell'opzione `run` per far partire l'esecuzione del programma!

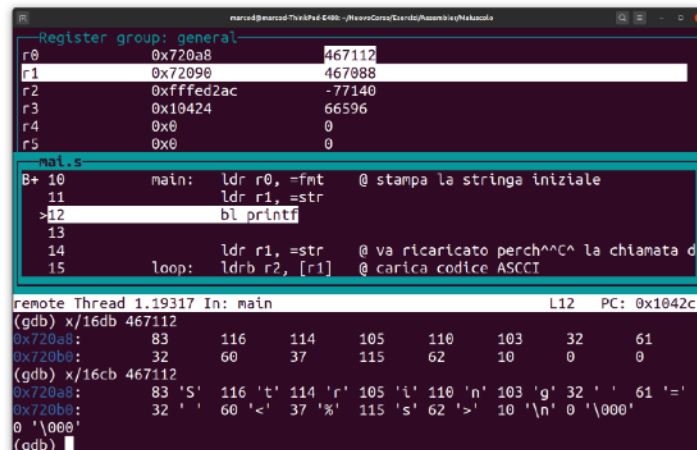
CCARM

- **x**

serve per vedere il contenuto di un'area di memoria di indirizzo noto. Il comando prevede un carattere / dopo la `x` seguito da

- un numero, che indica quanti elementi far vedere
- un formato, carattere che indica che formato utilizzare per i vari elementi (`d` per decimale, `x` per esadecimale, `b` per binario, ...)
- un'ampiezza, carattere che indica se vogliamo vedere parole (`w`), byte (`b`), ...
- un indirizzo, che è l'indirizzo base da cui cominciamo a vedere i valori

Per esempio `x/16dw 16384` ci farà vedere 16 parole (`w`) in formato decimale (`d`) a partire dall'indirizzo (decimale) 16384. Avremmo potuto utilizzare un indirizzo esadecimale facendolo precedere dal prefisso `0x` come al solito.



```
Register group: general
r0 0x720a8 467112
r1 0x72090 467088
r2 0xffffd2ac -77140
r3 0x10424 66596
r4 0x0 0
r5 0x0 0

main
B+ 10 main: ldr r0, =fmt @ stampa la stringa iniziale
11 ldr r1, =str
>12 bl printf
13
14 ldr r1, =str @ va ricaricato perch^^^ la chiamata d
15 loop: ldrb r2, [r1] @ carica codice ASCII

remote Thread 1.19317 In: main L12 PC: 0x1042c
(gdb) x/16db 467112
0x720a8: 83 116 114 105 110 103 32 61
0x720b0: 32 60 37 115 62 10 0 0
(gdb) x/16cb 467112
0x720a8: 83 'S' 116 't' 114 'r' 105 'i' 110 'n' 103 'g' 32 ' ' 61 '='
0x720b0: 32 ' ' 60 '<' 37 '%' 115 's' 62 '>' 10 '\n' 0 '\000'
0 '\000'
(gdb)
```

Nella figura si vede come possiamo osservare una stringa in memoria come codici decimali e come caratteri.

- **info registers**

permette di vedere i registri anche senza attivare il modo tui. `info registers` fa vedere tutti i registri. Se ne vogliamo vedere uno solo basta far seguire il nome del registro: `info registers r0`, per esempio, fa vedere il solo contenuto del registro `r0` (si può abbreviare con `info reg r0`)

4.0.1 Esempio di esecuzione single step senza modo tui

Supponiamo di voler eseguire passo passo il programma di divisione fra interi listato nella Sez. 5. Lanciamo il debugger nel solito modo. Qui di sotto trovate l'output relativo all'esempio, in cui:

CCARM

- linea 2: si lancia l'esecuzione del programma in modalità debug
- linea 4: si lancia il debugger multiarch
- linea 11: mettiamo un breakpoint all'etichetta `main` (entry point del codice)
- linea 13: facciamo partire il programma (è una `continue` invece che una `run` per via del fatto che stiamo utilizzando il cross compiler, vedi Sez. ??)
- linea 17: il debugger si ferma mostrando il codice della prossima istruzione da eseguire (in questo caso, la prima, quella che corrisponde all'etichetta `main`)
- linea 18: `next`: esegui l'istruzione e fermati alla prossima. Diamo il comando `next` (abbreviato `n` per altre 5 volte (riga 20, 22, 25, 27 e 29). Un ritorno carrello senza immissione di caratteri al prompt di `gdb` ripete l'ultimo comando impartito.
- linea 31: dopo l'esecuzione della `sub r1, r1, r2` e prima dell'esecuzione della `sub r1, r1, r2` chiediamo quanto valga il registro `r1`: vale 14 perchè è partito da 17 e abbiamo appena fatto l'iterazione 0 della divisione
- linea 33: `next` altra istruzione. A questo punto vedo che la prossima istruzione è la `b loop` e quindi il loop ricomincia. Decido di proseguire fino alla fine delle iterazioni, e quindi:
- linea 35: metto un breakpoint all'etichetta `end` e
- linea 37: continuo fino al prossimo breakpoint
- linee 42, 44: chiedo di vedere i valori di `r0` e `r1` (rispettivamente risultato e resto della divisione)
- linea 46: i risultati sono corretti, `continue`. Dal momento che non ci sono altri breakpoint attivi, il programma termina con la `exit`.

```

1 marcod@marcod-ThinkPad-E480
2 :~/NuovoCorso/Esercizi/Assembler/Div$ qemu-arm -g 12345 ./a.out &
3 [1] 24706
4 marcod@marcod-ThinkPad-E480
5 :~/NuovoCorso/Esercizi/Assembler/Div$ gdb-multiarch -q --nh -ex '
6   set architecture arm'
7   -ex 'file a.out' -ex 'target remote localhost:12345'
8 The target architecture is assumed to be arm
9 Reading symbols from a.out...
10 Remote debugging using localhost:12345
11 0x00010328 in _start ()
12 (gdb) b main
13 Breakpoint 1 at 0x10424: file div.s, line 7.
14 (gdb) c
15 Continuing.

```

```

16 Breakpoint 1, main () at div.s:7
17 7 main:    mov r1, #17 @ dividendo
18 (gdb) n
19 8    mov r2, #3  @ divisore
20 (gdb)
21 10    mov r0, #0  @ azzero il risultato
22 (gdb)
23 loop () at div.s:11
24 11 loop:  cmp r1, r2  @ se divisore maggiore di dividendo -> fine
25 (gdb)
26 12    blt end
27 (gdb)
28 13    sub r1, r1, r2  @ altrimenti toglì divisore dal dividendo
29 (gdb)
30 14    add r0, r0, #1  @ e incrementa il risultato
31 (gdb) info register r1
32 r1                0xe                14
33 (gdb) n
34 15    b loop      @ e ricomincia
35 (gdb) b end
36 Breakpoint 2 at 0x10444: file div.s, line 17.
37 (gdb) c
38 Continuing.
39
40 Breakpoint 2, end () at div.s:17
41 17 end:    mov r2, r1
42 (gdb) info register r0
43 r0                0x5                5
44 (gdb) info register r1
45 r1                0x2                2
46 (gdb) c
47 Continuing.
48 Il risultato e' 5 resto 2
49 [Inferior 1 (process 1) exited with code 032]
50 (gdb) q
51 [1]+  Exit 26                  qemu-arm -g 12345 ./a.out
52 marcod@marcod-ThinkPad-E480

```

5 Disassembler

Qualora di voglia visualizzare il contenuto di un file oggetto o eseguibile, possiamo utilizzare due comandi:

RBARM

- **od**
che può mostrare il contenuto di un file in vari formati (decimale, binario, esadecimale, ...) ma che non “disassembla” eventuali istruzioni assembler
- **objdump**
che disassembla sia il contenuto di un file oggetto che quello di un file eseguibile. In particolare, per vedere il disassemblato di un file occorre utilizzare l'opzione **-d**:
`objdump -d file.o`

Quando si utilizzi il cross compiler, invece che `objdump` dovremo utilizzare `arm-linux-gnueabi-hf-objdump`⁶.

Ad esempio, supponiamo di avere il file che contiene la routine per che calcola la divisione intera di due numeri:

```

1  .data
2  fm: .string "Il risultato vale %d resto %d\n"
3  .text
4  .global main
5  main:  mov r1, #17 @ dividendo
6        mov r2, #3  @ divisore
7        mov r0, #0  @ azzero il risultato
8
9  loop:  cmp r1, r2  @ se divisore maggiore di dividendo -> fine
10       blt end
11       sub r1, r1, r2 @ altrimenti togli divisore dal dividendo
12       add r0, r0, #1 @ e incrementa il risultato
13       b loop      @ e ricomincia
14
15  end:   mov r2, r1
16       mov r1, r0
17       ldr r0, =fm
18       bl printf
19
20       mov r7, #1  @ quindi esci con questo valore come esito
21       svc 0

```

Se compiliamo il file con un `gcc -c div.s` otteniamo un `div.o` che risulta essere un file oggetto:

```

1  marcod$ arm-linux-gnueabi-hf-gcc -c div.s
2  marcod$ ls -lstr | tail -1
3  4 -rw-rw-r-- 1 marcod marcod 836 nov 1 14:55 div.o
4  marcod$ file div.o
5  div.o: ELF 32-bit LSB relocatable, ARM, EABI5 version 1 (SYSV), not
        stripped
6  marcod$

```

Richiedendo un `objdump -d` otteniamo il seguente output:

```

1  marcod$ arm-linux-gnueabi-hf-objdump -d div.o
2
3  div.o:      file format elf32-littlearm
4
5
6  Disassembly of section .text:
7
8  00000000 <main>:
9      0: e3a01011  mov r1, #17
10     4: e3a02003  mov r2, #3
11     8: e3a00000  mov r0, #0
12
13  0000000c <loop>:
14     c: e1510002  cmp r1, r2
15    10: ba000002  blt 20 <end>

```

⁶di solito i comandi nativi per RBARM sono disponibili in CCARM facendo procedere il nome del comando dal prefisso `arm-linux-gnueabi-hf`

```

16 14: e0411002 sub r1, r1, r2
17 18: e2800001 add r0, r0, #1
18 1c: eaffffff b c <loop>
19
20 00000020 <end>:
21 20: e1a02001 mov r2, r1
22 24: e1a01000 mov r1, r0
23 28: e59f0008 ldr r0, [pc, #8] ; 38 <end+0x18>
24 2c: ebfffffe bl 0 <printf>
25 30: e3a07001 mov r7, #1
26 34: ef000000 svc 0x00000000
27 38: 00000000 .word 0x00000000
28 marcod$

```

Come si vede, il disassemblatore mostra sia la versione esadecimale che la versione disassemblata delle parole nel file. Considerate la riga 15 del listato precedente:

```
10: ba000002 blt 20 <end>
```

Questa è un'istruzione di salto. I primi 4 bit dell'istruzione rappresentano la condizione di salto (vedi Tab. 6.3 del libro di testo). In questo caso il valore **b** rappresenta la configurazione 1101_2 ovvero la condizione LT^7 . I 4 bit che seguono sono utilizzati per rappresentare l'istruzione di salto (**a** = 10_{10} = 1010_2 : i primi due bit (10) dicono che è un'istruzione di salto, i secondi che è un branch normale (non branch & link). Gli ultimi 24 bit ($0x..02$) sono l'offset. Tenendo conto che il PC al momento dell'esecuzione dell'istruzione contiene l'indirizzo dell'istruzione + 8 (per convenzione ARM), e che l'offset viene utilizzato come numero di parole, sommargli due significa saltare alla **mov** etichettata con la **end**. Infatti il PC + 8 sarebbe l'indirizzo della **add** alla linea 12, +1 parola sarebbe l'indirizzo della **b loop** alla riga 13 e +2 parole sarebbe l'indirizzo della **mov** alla linea 15.

Da notare che se disassemblassimo un file eseguibile, otterremmo tutta una serie di altre istruzioni che sono relative a ciò che viene linkato alle istruzioni che abbiamo scritto nel file **.s**. Questa cosa si può vedere semplicemente considerando la dimensione del disassemblato:

```

1 marcod$ arm-linux-gnueabi-hf-gcc -c div.s
2 marcod$ arm-linux-gnueabi-hf-objdump -d div.o | wc
3      26      93      576
4 marcod$ arm-linux-gnueabi-hf-gcc div.s
5 marcod$ arm-linux-gnueabi-hf-objdump -d a.out | wc
6     214     1046     7002
7 marcod$

```

Il disassemblato del file oggetto è di 26 linee, mentre quello del file eseguibile è di 214 linee. Lasciamo come esercizio l'opzione di vedere cosa ci sia effettivamente nel disassemblato dell'eseguibile.

⁷che è vera se è vera $N \oplus V$, vedi Sez. 6.3.2 del libro di testo

6 Istruzioni compilate (*pseudo-istruzioni*)

L'assembler mette a disposizione delle istruzioni che sembrano istruzioni del linguaggio macchina ma in realtà non lo sono. Tipico esempio sono le istruzioni `push {...}` e `pop {...}` che permettono di inserire/togliere sullo/dallo stack un insieme di registri specificato fra parentesi graffe. Le due istruzioni vengono "compilate" dal programma assembler come load/store multiple (LDM e STM, vedi Sez. 6.3.7. del libro di testo).

Altro esempio è l'istruzione che permette il caricamento di costanti lunghe in un registro. L'istruzione:

```
1 ldr r0, =0xff00ff00
```

carica nel registro `r0` il valore `0xff00ff00` che rappresenta una costante da 32 bit. Dal momento che non possiamo esprimere una costante così lunga in un formato che prevede per *tutta* l'istruzione un codice da 32 bit, il programma assembler traduce questa istruzione nel seguente modo:

- utilizza una parola del codice per contenere la costante da caricare, ovvero in vicinanza della posizione della `ldr r0, =...` nel codice piazza una `.data 0xff00ff00`. La locazione viene scelta in modo che la `.data` venga saltata nell'esecuzione del codice
- sostituisce la `ldr r0, =0xff00ff00` con una `ldr` che utilizza come base il PC e come offset un offset opportuno che la faccia puntare alla `.data` di cui sopra.

Per esempio, il codice:

```
1 ldru.o:      file format elf32-littlearm
2
3 Disassembly of section .text:
4
5 00000000 <main>:
6   0: e59f0004  ldr r0, [pc, #4]    ; c <main+0xc>
7   4: e3a07001  mov r7, #1
8   8: ef000000  svc 0x00000000
9  c: ff00ff00  .word 0xff00ff00
```

7 System call

Per effettuare una system call occorre fare due cose:

- preparare i parametri da passare utilizzando le stesse convenzioni utilizzate per la chiamata di procedura/funzione, ovvero utilizzare i registri da `r0` a `r3` (nell'ordine e per quanti ne sono necessari) per i primi 4 parametri e poi passare il resto dei parametri utilizzando lo stack.
- invocare la chiamata di sistema utilizzando una `svc 0` invece che una `bl` come si usa per chiamare procedure o funzioni. La `svc 0` ha un parametro

che viene passato mediante `r7` che rappresenta il *numero* della chiamata di sistema da effettuare.

La tabella che segue contiene i numeri di alcune delle syscall POSIX/Linux, fra quelle che sono più semplici da utilizzare e più utili per gli esercizi svolti nel nostro corso⁸:

Syscall	Numero	r0	r1	r2
exit	1	codice ritorno		
read	3	descrittore	ind buffer	dim buffer
write	4	descrittore	ind buffer	dim buffer
open	5	ind filename	flags	mode
close	6	descrittore		

(NOTA: i descrittori (normalmente già aperti) per standard input, output ed error sono rispettivamente 0, 1 e 2)

7.1 Esempio di chiamata di sistema: read

Supponiamo di voler leggere dei caratteri dallo standard input. La chiamata di sistema che legge caratteri da un file è (`man 2 read`):

`ssize_t read(int fd, void *buf, size_t count);` dove `ssize_t` è da intendersi sinonimo di intero a 32 bit, nel nostro specifico caso.

Lo standard input è sempre rappresentato dal descrittore (`fd`) 0.

Il codice che segue esegue una lettura da terminale:

```

1  .data
2  buf:  .fill 1024                @buffer da 1023 caratteri
3  msg:  .string "Immetti un numero:\n"  @ stringa di prompt
4  ris:  .string "Immessa %s (vale %d)\n" @ formato output programma
5
6  .text
7  .global main
8
9  main: push {lr}    @ salvo il punto di ritorno del main
10
11  ldr r0, =msg
12  bl printf    @ stampo il prompt (chiamata di fun)
13              @ adesso chiamo la read
14  mov r0, #0    @ parametri read : descrittore fd=0
15  ldr r1, =buf    @ parametri read : indirizzo buffer
16  mov r2, #1
17  lsl r2,r2,#10   @ parametri read : lunghezza buffer
18  mov r7, #3      @ read e' la syscall 3
19  svc 0          @ questa e' una chiamata di syscall
20  sub r0, r0, #1   @ r0 e' il numero di caratteri letti
21  ldr r1, =buf    @ rendo la stringa NULL terminated
22  mov r2, #0      @ costante NULL in r2
23  str r2, [r1,r0]  @ NULL al posto del \n
24  mov r0, r1      @ adesso la converto in intero

```

⁸la lista di tutte le syscall, con i loro parametri, la trovate a https://syscalls.w3challs.com/?arch=arm_thumb


```

25 bl    atoi
26 mov r2, r0    @ val della stringa terzo param printf
27 ldr r0, =ris    @ primo param ind stringa di formato
28 ldr r1, =buf    @ indirizzo del buffer con stringa letta
29 bl printf    @ chiamo le print
30 pop {lr}    @ prima di restituire il controllo,
31    @ ripesco l'indirizzo di ritorno dallo stack
32    @ (LR sporcato dalle bl a printf e atoi ...)
33 bx lr    @ e ritorno

```

Il programma stampa un prompt (`printf` della linea 12) e chiama una `read` (`svc 0` della linea 19). L'utente deve digitare a questo punto un numero intero che verrà memorizzato come caratteri ASCII nell'area di memoria riservata con la `fill` all'indirizzo `buf`. Normalmente l'input è terminato con un ritorno carrello. Se immettiamo i tre caratteri '1', '2' e '3' seguiti da un ritorno carrello, la `read` restituirà 4 (numero di caratteri letti) e il contenuto del buffer `buf` sarà (linee ottenute utilizzando il debugger):

```

1 (gdb) info reg r1
2 r1                0x2102c                135212
3 (gdb) x/8cb 135212
4 0x2102c:  49 '1'  50 '2'  51 '3'  10 '\n'  0 '\000'  0 '\000'  0
              '\000'  0 '\000'
5 (gdb)

```

Con le linee 20–23 andiamo a sostituire il `\n` finale con un `NULL`:

```

1 (gdb) n
2 21    ldr r1, =buf    @ rendo la stringa NULL terminated
3 (gdb)
4 22    mov r2, #0    @ costante NULL in r2
5 (gdb)
6 23    str r2, [r1,r0]    @ NULL al posto del \n
7 (gdb)
8 24    mov r0, r1    @ adesso la converto in intero
9 (gdb) x/8cb 135212
10 0x2102c:  49 '1'  50 '2'  51 '3'  0 '\000'  0 '\000'  0 '\000'  0
              '\000'  0 '\000'
11 (gdb)

```

quindi invochiamo una `atoi` (funzione di libreria `glibc`) per convertire la stringa in un intero:

```

1 (gdb) n
2 25 bl    atoi
3 (gdb)
4 26    mov r2, r0    @ val della stringa terzo param printf
5 (gdb) info reg r0
6 r0                0x7b                123
7 (gdb)

```

e finalmente stampiamo il messaggio che riporta stringa letta e valore calcolato (`printf` alla linea 29).

Notare che il programma funziona regolarmente (anche in presenza della syscall `read`) in entrambi gli ambienti (RBARM e CCARM) e in entrambi i modi (esecuzione diretta, esecuzione mediante debugger). E' possibile che eseguendolo

col debugger prompt e caratteri immessi “sporchino” il layout delle finestre ottenuto mediante tui `reg`.

8 Terminazione di un programma

Un programma scritto in assembler si comporta come un programma scritto in C, ai fini della terminazione. Può terminare in due modi:

- con una `return`, dal momento che il `main` viene chiamato dal codice nel preambolo linkato dal `gcc` come ogni altra funzione. In questo caso è il codice della libreria C (anche nel caso il `main` sia scritto direttamente in assembler come nei nostri programmi) che si preoccupa della terminazione e di restituire il controllo al sistema operativo;
- con una `exit`; in questo caso è direttamente il `main` che invoca la `exit` per terminare il processo in esecuzione e restituire così il controllo al sistema operativo.

In assembler ARMv7 le due opzioni corrispondono a codici diversi.

- La `return` può essere implementata in diversi modi. Tipicamente con una `mov pc,lr`, una `mov r15, r14` o una `bx LR`. In tutti a casi, il registro `r0` conterrà il codice di ritorno della `return`;
- la `exit` richiede una `syscall` ed è implementata come spiegato nella Sez. 7 mediante il codice

```
1      mov R7,#1      @ numero di call = 1
2      svc 0          @ super visor call 0
```

Naturalmente anche la `exit` accetta in `r0` il valore da passare come codice di ritorno (`exit(codrit)`).

9 Docker

Il listato che segue è quando dovete includere nel `Dockerfile` per creare un docker con tutti gli strumenti necessari per compilare ed eseguire codice ArmV7 sia su macchine Windows/Linux con Intel che su macchine Mac OS/X sempre con processori Intel.

```
1 ### start from current ubuntu image
2 FROM ubuntu:groovy
3 MAINTAINER marcod
4 ### install base programming tools
5 RUN apt update
6 RUN apt install -y vim nano gcc-arm-linux-gnueabi qemu-user gdb-
  multiarch
7 ### set up service ssh and net tools
8 RUN apt install -y ssh openssh-server net-tools
9 RUN service ssh start
```

```
10 ### fix user
11 RUN useradd -ms /bin/bash ae2020
12 USER ae2020
13 WORKDIR /home/ae2020
```

Seguendo le istruzioni sui tutorial Docker (e.g. <https://docs.docker.com/get-started/part2/>) potete semplicemente creare un container docker, utilizzarlo per esercitarvi con l'assembler e anche montare delle directory locali in modo che possano essere viste all'interno del container ed evitare così la necessità di copiare i file nel container e dal container.

Per esempio:

- `docker build -t marcod/ae2020 .` eseguito in una directory dove c'è il `Dockerfile` con i contenuti visti sopra crea il container
- `docker run -it marcod/ae2020 bash` lancia il container con una shell interattiva in cui potete utilizzare i cross-tool discussi in questo capitolo
- `docker run --mount src=/home/marcod/.../dirlocale, target=/home/ae2020/Dummy,type=bind -it marcod/ae2020 bash` lancia il container con la directory locale visibile come sottodirectory `Dummy` della home nel container

Notare che potrebbe essere necessari i diritti di root per compiere queste operazioni. Per uscire dal container basta lanciare il comando `exit` dalla shell. **Attenzione:** *eventuali file creati nel container (non nella eventuale directory montata come illustrato sopra) verranno persi!*