



PROGETTO 04 PRINT / SCAN



OBIETTIVO

Consegna:

realizzare un programma assembly RISC-V in grado di stampare su monitor terminale numeri con segno ad n cifre. Ad ogni dato interamente stampato, viene inoltre aggiunto un ritorno a capo. Il codice deve prevedere anche la stampa di stringhe normali di testo. Rendere il codice esportabile e modulare, per essere importato in ogni altro programma assembly. Eseguire il test tramite una funzione di moltiplicazione tra un vettore ed una costante, ogni elemento del vettore deve essere moltiplicato per un dato valore costante.

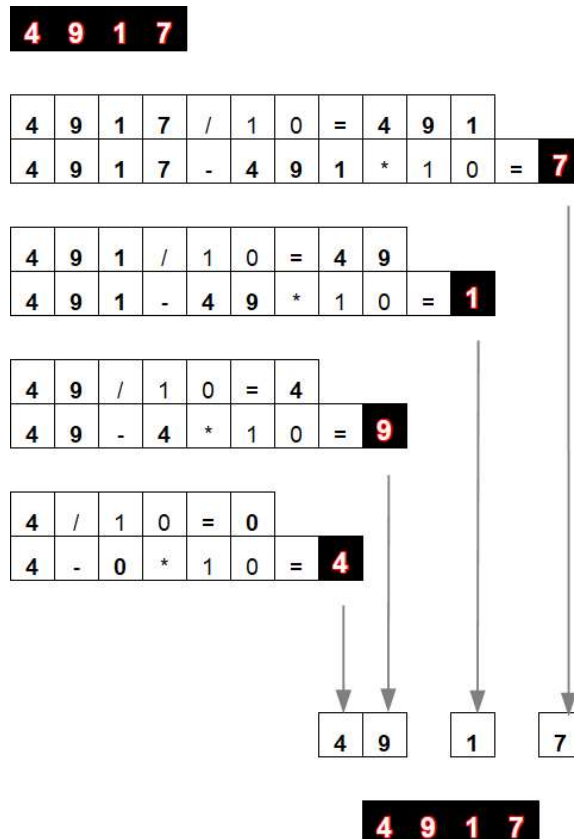
Approfondimento personale:

integrare il programma della print con la lettura di valori di tipo signed byte da terminale. L'utente può dunque inserire valori numerici (formattati in caratteri) composti da segno e tre cifre, ovvero sxxx (es. +007, -045, +117).

DESCRIZIONE

Il programma presentato è suddiviso su più file per rendere maggiormente chiaro il codice prodotto e per poter essere trasportabile su ogni altro file, potrà infatti essere comodamente importato e quindi incluso nel file permettendo all'utente di accedere alle proprie funzioni. Il programma prevede una stampa formattata di numeri con segno ad n cifre e mette a disposizione la classica funzione print assembly per stampare anche semplici notazioni testuali.

Mentre le funzioni per scrivere testo e ritorno a capo sono intuitive, il processo per stampare i numeri richiede maggior attenzione. L'algoritmo richiede infatti di stampare una cifra alla volta e legare tutte le cifre stampate effettuando nessun ritorno a capo se non alla fine del numero. L'algoritmo per individuare la cifra singola ed associarne la giusta potenza in base decimale prevede l'utilizzo di una funzione ricorsiva. In particolare per trovare cifra la cifra si divide per 10 il numero ed il risultato intero della divisione moltiplicato per 10 sottratto al dato in considerazione, ritorna la cifra di peso minore. Applicando l'algoritmo in maniera ricorsiva ad ogni volta il numero preso avrà un peso minore nell'ordine di grandezza della potenza. Il processo termina quando il numero diviso e moltiplicato è minore di 10, vuol dire che si è raggiunta l'unità del numero.



SVILUPPO

Analisi algoritmi:

Print numbers:

l'algoritmo per stampare a monitor un numero composto da più cifre si basa sulla ricorsione e la possibilità di effettuare divisioni su numeri interi. Il numero viene infatti subito diviso per 10, formattato come intero è come se subisse una sorta di "shift right" in base 10. Calcolato il prodotto del valore ottenuto per 10, viene poi sottratto quest'ultimo risultato al numero originale ottenendo pertanto la cifra meno significativa come valore numerico. Da cui applicando il processo tante volte quante le cifre del numero è possibile ricomporre il numero cifra per cifra.

REALIZZAZIONE

```
ts_print_scan.s X tsPrint.s tsScan.s mulConstVect.s
C: > Users > Semmy17 > Desktop > Università > Architettura Calcolatori > Progetti Esame > Prog_004 > ts_print_scan.s
1  #-----PROGRAM TO CHECK SCAN AND PRINT FUNCTION FOR NUMBERS FROM TERMINAL-----#
2
3  #-----NUMBER AND MNEMONIC NAME REGISTER RISC-V-----#
4  # x0 -x4 #  zero, ra, sp, gp, tp
5  # x5 -x7 #  t0, t1, t2
6  # x8 -x9 #  fp/s0, s1
7  # x10-x17 # a0, a1, a2, a3, a4, a5, a6, a7
8  # x18-x27 # s2, s3, s4, s5, s6, s7, s8, s9, s10, s11
9  # x28-x31 # t3, t4, t5, t6
10
11 #-----MODULE-----#
12 .include "tsPrint.s"
13 .include "tsScan.s"
14 .include "mulConstVect.s"
15
16 #-----constant-----#
17 .global _start
18     # system code interrupt
19     .equ _SYS_EX, 93
20     # constant
21
22 #-----global symbol-----#
23 .section .data
24     vect: .space 4      # space to allocate 4 byte
25     const: .byte 1      # multiplier setted 1
26     res: .space 16      # space to allocate 16 byte
27
28 .section .rodata
29     # message for terminal
30     inVect: .string "Insert 4 numbers:\n"
31     inConst: .string "Insert 1 multiplier:\n"
32     outRes: .string "Result of operation is:\n"
```

Il programma assembly RISC-V si compone di 4 files per rendere il codice chiaro e trasportabile, le funzioni importate infatti nel programma principale sono funzioni usate e compatibili con altri programmi assembly scritti.

Il codice riportato compone l'header del codice, sono infatti inclusi i file di appoggio per le funzioni necessarie da importare, la dichiarazione di costanti (interpretati dal programma come immediati), le dichiarazioni di dati globali.

La sezione data raccoglie infatti i dati disponibili per lettura e scrittura all'interno del programma, il dato data indica infatti un array di 4 valori di tipo dati da inserire da terminale, così come id dato di tipo byte const accetta in ingresso un valore che farà da moltiplicatore per tutti gli elementi del vettore data, andando a compiere uno store dei risultati nel vettore res che conterrà 4 valori di tipo word. Le stringhe inserite nella rodata (Read Only Data) sono riportate per rendere l'interfaccia più chiara all'utente inserendo alcuni messaggi da visualizzare tramite la funzione print.

```

34 #-----code-----#          64 # call to print function for message          78 # call to print function for message
35 .section .text                65 la a1, inConst          79 la a1, outRes
36                               66 li a2, 21                80 li a2, 25
37 #-----main program-----#   67 jal print                81 jal print
38 _start:                       68                               82
39 # call to print function for message 69 # scan number input and allocate in var multiplier 83 # operation on number and allocate in res vector
40 la a1, inVect                 70 # prepare calling to function 84 # prepare calling to function
41 li a2, 18                     71 la a0, const             85 la a0, res
42 jal print                     72 jal scanSignedNumbersTilCarriage 86 la a1, vect
43                               73                               87 la a2, const
44 # scan number input and allocate in data vector 74 jal printCarriage # print one line empty 88 li a3, 4
45 la s1, vect # address of data vector 89 jal MCV
46 #-----forscan-----#       90
47 li s0, 0 # index for cycle    91
48 forscan:                     92 # print signed numbers and carriage function
49 # user code                   93 la s1, res # address of data vector
50 # prepare calling to function 94 #-----forprint-----#
51 mv a0, s1                    95 li s0, 0 # index for cycle
52 jal scanSignedNumbersTilCarriage 96 forprint:
53                               97 # user code
54 addi s1, s1, 1 # increment byte pointer 98 # prepare calling to function
55 # condition exit              99 lw a0, 0(s1)
56 li t0, 4                     100 jal ra, printSignedNumbersAndCarriage
57 addi s0, s0, 1                101 addi s1, s1, 4 # increment word pointer
58 blt s0, t0, forscan           102 # condition exit
59                               103 li t0, 4
60 jal printCarriage # print one line empty 104 addi s0, s0, 1
                                   105 blt s0, t0, forprint
                                   106
                                   107 _end:
                                   108 li a7, _SYS_EX
                                   109 ecall
                                   110

```

_start:

identificato come il programma principale di ogni programma assembly RISC-V, la label `_start` identifica l'inizio del classico main all'interno della sezione `text`. Il main in sostanza svolge il compito di funzione principale e chiamante di tutte le altre funzioni del programma per gestirne l'esecuzione ed interfacciamento. All'interno del suo corpo sono riconoscibili i richiami alle diverse funzioni di “scan”, “print” (numeri e stringhe) ed MCV.

Per quanto concerne il richiamo delle funzioni di scan e print in generale, i metodi sono molto simili. Viene infatti definita una struttura `for` per elaborare i 4 dati da inserire o scrivere su terminale, ed il ciclo itera su un registro di tipo “saved” di appoggio poiché la funzione chiamata “scan” o “print”, sovrascriverebbe un normale registro “temporary”. Ad ogni iterazione, rispettivamente, alla funzione di “scan” viene passato un indirizzo di locazione della memoria su cui salvare il dato letto, mentre alla funzione “print” un valore da stampare su terminale. I richiami aggiuntivi alla funzione print fuori dalla struttura dei cicli `for`, interessano la scrittura su monitor di messaggi di tipo stringa e quindi semplici notazioni riportate dal programma per l'utente oltre a dei ritorni a capo per formattare meglio la stampa, mentre quello della scan fuori da ogni ciclo interessa solo l'assegnazione di un valore al dato `const`.

Il richiamo della funzione MCV, infine, è ottenuto dopo l'inserzione dei diversi dati e prima del print finale. Alla funzione, si vedrà più avanti, come vengano solo passati puntatori e non valori od indici, la funzione infatti itera tramite metodo puntatori per ottimizzare il processo.

```

1  #-----TS PRINT SIGNED NUMBERS-----#
2
3  #-----print signed numbers and carriage function-----#
4  # the function uses:
5  #   - argument register:
6  #       - a0: number to print
7  #   - temporary register t0, t1, t2, t3
8  #   - global symbol appPrint, carriage, plusPrint, minusPrint
9
10 #-----constant-----#
11 .global _start
12     # system code interrupt
13     .equ _SYS_WR, 64
14     # constant
15     .equ asciiOffset, 48
16
17 #-----global symbol-----#
18 .section .data
19     appPrint: .byte '!'
20
21 .section .rodata
22     carriage: .byte '\n'
23     plusPrint: .byte '+'
24     minusPrint: .byte '-'
25
26 #-----code-----#
27 .section .text

```

tsPrint.s:

identificato come il file per l'implementazione della stampa su terminale, il codice è suddiviso in quattro funzioni ovvero print, printSignedNumbersAndCarriage, printSign e printCarriage. Il codice sebbene importato come modulo esterno, si deve appoggiare ad alcune variabili in memoria ed alcuni immediati.

I dati globali della sezione rodata sono i segni “+” e “-” per indicare la positività dei numeri, ed il valore “/n” per il ritorno a capo da inserire a fine testo. Alla sezione rodata si affianca la sezione data per la variabile di appoggio per la rappresentazione dei diversi caratteri numerici, mentre le costanti sono il numero 48, offset dei caratteri numerici in codice Ascii, e il valore 64, da passare come valore di richiamo interrupt verso il sistema operativo per scrivere su terminale.

```

29 #-----print function-----#
30 print:
31     li a0, 0          # code terminal
32     # la a1, string
33     # li a2, length
34     li a7, _SYS_WR    # interrupt code
35     ecall             # system call
36     jr ra

```

Passando dunque al codice del modulo, la prima funzione a cui fare riferimento è quella di print. Alla funzione vengono passati come argomenti il puntatore in memoria della stringa di testo da stampare e la lunghezza della stringa. Il codice all'interno della funzione si riduce solamente al richiamo dell'interrupt verso il sistema operativo, vengono infatti solo passati alcuni altri argomenti per richiamare la funzione tramite la ecall. Si passa infatti il codice di richiamo terminale in a0 e il valore di interrupt associando l'immediato prima dichiarato. A giustificare la scelta di creare una funzione propria per la print è il fatto che l'utente può accedere anche a questa funzione e dunque riuscire tramite una procedura

semplice ed immediata a stampare delle stringhe di testo locate in memoria. Con un solo modulo dunque si include sia la stampa di numeri con segno che quella di stringhe normali.

```
38 # function to print signed numbers and final carriage
39 printSignedNumbersAndCarriage:
40     addi sp, sp, -8
41     sd ra, 0(sp)
42     jal printSign
43     jal printNumbers
44     jal printCarriage
45     ld ra, 0(sp)
46     addi sp, sp, 8
47     jr ra
```

La funzione `printSignedNumbersAndCarriage` risulta essere la funzione madre tra quelle del modulo, è infatti la funzione di richiamo per la stampa del numero su terminale. Il codice è semplice ed immediato, infatti la funzione non fa altro che coordinare le diverse procedure di stampa del dato. Viene infatti subito stampato il segno del numero, poi il numero stesso ed infine il ritorno a capo. L'unica accortezza sta nel fatto di salvare lo stato del return address che conterrà il valore di ritorno dalla funzione ovvero il valore del Program Counter a cui si punterà dopo aver terminato l'esecuzione della funzione intera. Il PC infatti sarà calcolato come il PC + 4 poiché verrà salvato nel registro `ra` il valore dell'istruzione successiva (RISC-V codifica istruzioni a 4 byte) all'interno dell'istruzione memory. Salvando e ripristinando il valore del `ra` prima e dopo all'interno della funzione, si evita di sovrascriverne il contenuto e non potervi riaccedere cadendo così in una situazione di loop irrimediabile.

```
49 # function that only print sign
50 printSign:
51     bge a0, zero, major
52     la a1, minusPrint
53     sub a0, zero, a0
54     beq zero, zero, sign
55     major:
56     la a1, plusPrint
57     sign:
58     li a2, 1
59     addi sp, sp, -16
60     sd ra, 0(sp)
61     sd a0, 8(sp)
62     jal print
63     ld ra, 0(sp)
64     ld a0, 8(sp)
65     addi sp, sp, 16
66     jr ra
```

La prima tra le funzioni chiamate dalla `printSignedNumbersAndCarriage` è la funzione di segno. All'interno della funzione viene infatti confrontato il valore del registro `a0`, contenente il valore di cui stamparne il segno, con lo zero. Nel caso sia soddisfatta la condizione di `bge` (branch greater equal), la funzione caricherà all'interno del registro `a1` l'indirizzo del segno "+" in memoria, al contrario, oltre a caricare il segno opposto "-", il registro `a0` verrà invertito di segno tramite un'istruzione di `sub`, facendo così, il numero da valutare in fase di stampa sarà positivo e dunque accettabile come calcoli da svolgere. A questo punto si salta l'assegnamento dell'indirizzo del segno "+" al registro `a1` e si continua, caricando come lunghezza del dato da stampare il valore 1 e richiamando la

funzione di print. Da notare ancora una volta come occorre salvare lo stato del ra e lo stato del registro a0 che nella funzione print verrà sovrascritto.

```
68 # function that only print numbers
69 printNumbers:
70     li t0, 10
71     div t1, a0, t0
72     mul t2, t1, t0
73     sub t3, a0, t2
74     addi sp, sp, -16
75     sd ra, 0(sp)
76     sd t3, 8(sp)
77     blt t2, t0, exitPrintNumbers
78     mv a0, t1
79     jal printNumbers
80 exitPrintNumbers:
81     ld t0, 8(sp)
82     addi t0, t0, asciiOffset
83     la a1, appPrint
84     sb t0, 0(a1)
85     jal print
86     ld ra, 0(sp)
87     addi sp, sp, 16
88     jr ra
```

La seconda tra le funzioni chiamate dalla printSignedNumbersAndCarriage è la funzione printNumbers che si occupa di scrivere cifra a cifra il numero ricostruito in serie di caratteri. Il codice segue l'algoritmo prima spiegato per la stampa di numeri a più cifre significative. Come argomento di ingresso la funzione accetta nel registro a0 il numero in decimale. Posto un divisore/moltiplicatore come 10 nel registro t0 si possono effettuare i vari calcoli di divisione, moltiplicazione ed infine sottrazione. Ottenuto il risultato su un registro di appoggio t3 si salva sullo stack il tale registro ed il ra. Questa operazione risulta necessaria per due motivi: il primo ci permette di effettuare la ricorsione non perdendo dati importanti come le cifre dei cicli precedenti dunque sapere dove puntare alla fine della ricorsione, quindi avere salvato un return address. Il secondo motivo invece sta nella composizione e stampa del numero. Ad ogni ciclo infatti il registro t3 avrà salvata la cifra meno significativa del numero in decimale, quindi l'unità. Per stampare dunque il numero ricomposto bisogna stampare l'ultima cifra trattata nel ciclo per prima e la prima cifra trattata come ultima. In altre parole ad ogni ciclo si ottiene la cifra meno significativa quindi all'ultimo ciclo corrisponde quella più importante; stampare tutto alla fine significa stampare prima la cifra più significativa e poi tutte le altre; stampare ciclo a ciclo significa stampare il numero "specchiato".

Fatto il salvataggio sullo stack si necessita di verificare se procedere con la ricorsione o meno. Nel caso infatti che t2, ovvero il numero di ingresso privato dell'ultima cifra, sia minore di t0, posto a 10, il ciclo uscirà, altrimenti continuerà la ricorsione passando però come argomento alla nuova funzione il valore di t1 ad a0, ovvero la divisione tra il registro a0 di ingresso e t0. La ricorsione svolgerà per cui n cicli, alla fine dei quali la condizione di uscita risultando verificata, farà procedere il codice nella parte finale. Se prima le cifre sono state salvate sullo stack, la prima istruzione sarà infatti quella di recuperare l'ultima cifra salvata per stamparla. Viene infatti effettuata una load dallo stack, aggiunto l'immediato 48 per trasformare il numero in carattere Ascii, salvato il valore ottenuto all'interno della "variabile" di appoggio appPrint e poi chiamata la funzione di print, passando come argomento l'indirizzo di a1 e la lunghezza 1 salvata in precedenza nel registro a2. Effettuata per cui la stampa si riporta dallo stack il return address all'interno del registro ra e si chiama una jalr, facendo attenzione ad aumentare il valore dello stack

pointer di 16 byte. Da cui dunque il ciclo si ripete altre n-1 volte per la stampa di ogni cifra fino a quando il ra non punterà alla funzione chiamante printSignedNumbersAndCarriage.

```
90 # function that only print carriage
91 printCarriage:
92     li a0, 0
93     la a1, carriage
94     li a2, 1
95     li a7, _SYS_WR
96     ecall
97     jr ra
```

Terza ed ultima funzione da considerare è la printCarriage che non effettua altro che una print di un ritorno a capo. Utilizzata anche nel “main”, serve per mandare a capo dopo la stampa di tutte le cifre del numero oppure anche a lasciare per ordine uno spazio bianco all'interno della finestra del terminale.

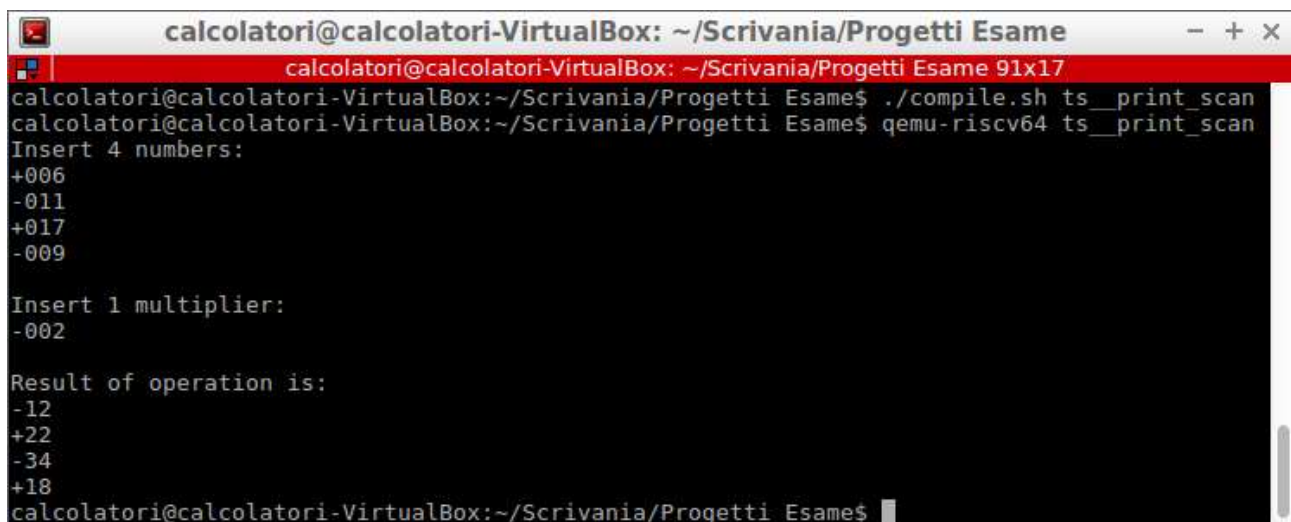
```
1  #-----TS MULTIPLIER CONSTANT-VECTOR-----#
2
3  #-----multiplier constant-vector function-----#
4  # the function uses:
5  #   - argument register:
6  #       -a0: address vector result
7  #       -a1: address vector input
8  #       -a2: address variable multiplier
9  #       -a3: vector length
10 #   - temporary register t0, t1, t2, t3
11
12 MCV:
13     slli t0, a3, 2      # multiply size and length
14     add t0, t0, a0      # end of address
15     lb t2, 0(a2)        # var
16
17 #-----forMCV-----#
18 forMCV:
19     # user code
20     lb t1, 0(a1)        # data [*]
21     mul t3, t1, t2
22     sw t3, 0(a0)        # res [*] = data [*] * var
23     addi a0, a0, 4      # increment word pointer
24     addi a1, a1, 1      # increment byte pointer
25     # condition exit
26     blt a0, t0, forMCV
27
28     jr ra
29
```

mulConstVect.s:

identificato come la funzione di test tra un vettore ed una costante di ingresso, associa ad ogni valore di tipo word la moltiplicazione di un vettore ed un dato di tipo byte. In ingresso alla funzione sono infatti riportati ai registri a0, a1 ed a2 gli indirizzi dei dati res, data, const, mentre nel registro a3 viene inserita la lunghezza del vettore destinazione, in questo caso 4 elementi. Eseguendo un accesso ottimizzato tramite i puntatori, per risparmiare codice all'interno del for viene subito calcolato l'indirizzo massimo che si raggiungerà in memoria sul vettore destinazione. Tramite uno shift left di 2 posizioni dell'argomento a3 viene calcolato la lunghezza del vettore, sommato all'indirizzo base si accede direttamente elemento per elemento. E' da subito possibile caricare la costante

const su un registro di appoggio t2 poiché non cambierà il suo valore all'interno del loop. Con il for dunque si acceda ad ogni elemento del vettore e moltiplicandolo per la costante si effettua una store tipo word nell'indirizzo associato al registro a0. Da qui si procede eseguendo l'accesso, tramite un addi, agli elementi successivi dei vettori ingresso e destinazione. In questo punto si vede come l'ottimizzazione tramite puntatori permette di ridurre il codice di diverse istruzioni, i registri a0 ed a1 hanno infatti solo bisogno di essere incrementati di un'unità nel caso di dati tipo byte e di quattro unità nel caso di dati di tipo word. Effettuando un confronto tra l'indirizzo dell'elemento successivo e quello massimo del vettore destinazione, si ritorna od esce dal for.

TEST



```
calcolatori@calcolatori-VirtualBox: ~/Scrivania/Progetti Esame
calcolatori@calcolatori-VirtualBox: ~/Scrivania/Progetti Esame 91x17
calcolatori@calcolatori-VirtualBox:~/Scrivania/Progetti Esame$ ./compile.sh ts__print_scan
calcolatori@calcolatori-VirtualBox:~/Scrivania/Progetti Esame$ qemu-riscv64 ts__print_scan
Insert 4 numbers:
+006
-011
+017
-009

Insert 1 multiplier:
-002

Result of operation is:
-12
+22
-34
+18
calcolatori@calcolatori-VirtualBox:~/Scrivania/Progetti Esame$
```

Impostazioni emulazione:

per realizzare un test pratico del programma scritto in assembly ci si avvale degli strumenti di compilazione ed emulazione installati su una macchina virtuale Linux, capace di simulare l'architettura RISC-V ed interpretare dunque il codice assemblato. I comandi da impartire tramite monitor terminale sono quelli di compilazione, utile per creare l'eseguibile del file basandosi sul codice scritto tramite Visual Studio Code, e lo strumento di esecuzione qemu in grado di emulare l'architettura RISC-V ed eseguire il file assemblato in precedenza. Non sarà necessario in questo caso appoggiarsi al sistema di debugging gdb poiché è possibile tramite terminale, ricorrendo alle funzioni di "scan" e "print", inserire e leggere i dati ciclanti dal programma.

Emulazione:

In fase di esecuzione il programma dunque presenta subito all'utente la richiesta di inserire 4 valori di tipo byte. La funzione "scan" (di seguito approfondita), permette l'input degli ingressi formattati però in segno e cifre lette (sxxx), riempiendo con degli 0 le cifre non significative. Inseriti i 4 dati del vettore si inserisce la costante moltiplicativa sempre tramite "scan" e si esegue l'operazione di moltiplicazione riportando in uscita i dati calcolati e stampandoli tramite la "print".

OTTIMIZZAZIONI

Il programma, se non la funzione MCV, non presenta particolari ottimizzazioni a livello di codice per rendere il modulo capace di offrire all'utente un set completo di richiami a livello di print e scan. Nello specifico, il modulo tsPrint.s presenta diverse funzioni al suo interno, alcune delle quali capaci di lavorare separatamente, permettendo per cui il richiamo per scopi aggiuntivi (inserimento stringhe di testo tramite print, ritorni a capo tramite printCarriage). Il modulo risulta pertanto completo di funzioni accessorie oltre a quella di "print" normale di numeri a più cifre. Il codice in questo caso non risulta ottimizzato poiché i basic blocks sono ristretti, ed essendoci molti richiami a funzioni di appoggio per compiti specifici, bisogna accedere spesso in stack, questo comporta utilizzo di memoria non necessario che dilunga di molto i tempi di esecuzione. La scelta è stata fatta comunque per avere con un unico modulo diverse funzioni accessorie ed inoltre perché la stampa a terminale è spesso fatta solo in fase di debug e sviluppo del codice. Un discorso analogo vale il modulo tsScan.s. L'unica parte più ottimizzata è infatti una routine che potrebbe essere di uso comune come la funzione MCV, accedere a vettori tramite puntatori è infatti un'ottimizzazione propria di compilatori attuali.

APPRONFONDIMENTO PERSONALE

Descrizione:

l'approfondimento personale in questione tratta la realizzazione di un modulo in codice assembly RISC-V capace di leggere da terminale numeri di tipo byte con segno e salvarli su un indirizzo specificato in ingresso alla funzione.

Scan numeri terminale:

il codice prodotto assume la stessa struttura e caratteristiche del codice prodotto per le varie funzioni di print. Suddiviso in funzioni capaci di lavorare in maniera autonoma aiuta l'utente a "dialogare" con l'interfaccia terminale in fase di sviluppo e debug.

```
1  #-----TS SCAN SIGNED NUMBERS-----#
2
3  #-----scan signed numbers and carriage function-----#
4  # the function uses:
5  #   - argument register:
6  #       - a0: address variable to allocate scan number
7  #   - temporary register t0, t1, t2, t3, t4, t5, t6, s0
8  #   - global symbol appScan, plusScan, minusScan
9
10 #-----constant-----#
11 .global _start
12     # system code interrupt
13     .equ _SYS_SC, 63
14     # constant
15     .equ asciiOffset, 48
16
17 #-----global symbol-----#
18 .section .data
19     appScan: .byte '!', '!', '!', '!', '!'
20
21 .section .rodata
22     plusScan: .byte '+'
23     minusScan: .byte '-'
```

tsScan.s:

identificato come il file per l'implementazione della lettura su terminale, il codice è suddiviso in tre funzioni ovvero scan, scanSignedNumbersTileCarriage, scanNumbers. Il codice sebbene importato come modulo esterno, si deve appoggiare ad alcune variabili in memoria ed alcuni immediati.

I dati globali della sezione rodata sono i segni "+" e "-" per indicare la positività dei numeri. Alla sezione rodata si affianca la sezione data per la variabile di appoggio per la rappresentazione dei diversi caratteri numerici, mentre le costanti sono il numero 48, offset dei caratteri numerici in codice Ascii, e il valore 63, da passare come valore di richiamo interrupt verso il sistema operativo per leggere da terminale.

```

25 #-----code-----#
26 .section .text
27
28 #-----scan function-----#
29 scan:
30     li a0, 0      # code terminal
31     # la a1, string
32     # li a2, length
33     li a7, _SYS_SC # interrupt code
34     ecall         # system call
35     jr ra

```

Scan:

passando dunque al codice del modulo, la prima funzione a cui fare riferimento è quella di scan. Alla funzione vengono passati come argomenti il puntatore in memoria della stringa di testo da stampare e la lunghezza della stringa. Il codice all'interno della funzione si riduce solamente al richiamo dell'interrupt verso il sistema operativo, vengono infatti solo passati alcuni altri argomenti per richiamare la funzione tramite la ecall. Si passa infatti il codice di richiamo terminale in a0 e il valore di interrupt associando l'immediato prima dichiarato.

```

37 # function to print numbers and final carriage
38 scanSignedNumbersTilCarriage:
39     addi sp, sp, -16
40     sd ra, 0(sp)
41     sd s0, 8(sp)
42     mv s0, a0
43     la a1, appScan
44     li a2, 5
45     jal scan
46     jal scanNumbers
47     ld ra, 0(sp)
48     ld s0, 8(sp)
49     addi sp, sp, 16
50     jr ra

```

scanSignedNumbersTilCarriage:

La funzione scanSignedNumbersTilCarriage risulta essere la funzione madre tra quelle del modulo, è infatti la funzione di richiamo per la stampa del numero su terminale. La solita accortezza da considerare sta nel fatto di salvare lo stato del return address che conterrà il valore di ritorno dalla funzione ovvero il valore del Program Counter a cui si punterà dopo aver terminato l'esecuzione della funzione intera. Il PC infatti sarà calcolato come il PC + 4 poiché verrà salvato nel registro ra il valore dell'istruzione successiva (RISC-V codifica istruzioni a 4 byte) all'interno dell'istruzione memory. Salvando e ripristinando il valore del ra prima e dopo all'interno della funzione, si evita di sovrascriverne il contenuto e non potervi riaccedere cadendo così in una situazione di loop irrimediabile. Altra considerazione da notare sta nell'appoggio di un saved register come registro di memoria. I saved register necessitano, all'interno della funzione chiamata, di essere salvati e ripristinati poiché, secondo la convenzione, non è compito della funzione chiamante preoccuparsi di questa procedura. Il registro s0 è infatti destinato a memorizzare il dato nel registro a0, invece di accedere più volte allo stack, è preferibile usare questo registro per rendere il codice più veloce. Utilizzato dunque questo stratagemma per l'ottimizzazione del codice, si passa al richiamo delle diverse funzioni, passando ai dovuti registri i corretti argomenti.

```

52 #-----scannumbers function-----#
53 scanNumbers:
54     addi t3, a1, 1
55     li t2, 0
56     #-----forscannumbers-----#
57     li t0, 0
58     forScanNumbers:
59         # user code
60         lb t4, 0(t3)
61         addi t4, t4, -asciiOffset
62         addi t3, t3, 1
63         li t5, 100
64         li t6, 10
65         #-----forpot-----#
66         li t1, 0
67         forPot:
68             # user code
69             beq t1, zero, pot0
70             div t5, t5, t6
71             pot0:
72             # condition exit
73             addi t1, t1, 1
74             ble t1, t0, forPot
75             mul t4, t4, t5
76             add t2, t2, t4
77         # condition exit
78         li t4, 3
79         addi t0, t0, 1
80         blt t0, t4, forScanNumbers
81         la t1, plusScan
82         lb t1, 0(t1)
83         lb t0, 0(a1)
84         beq t0, t1, nop
85         sub t2, zero, t2
86     nop:
87     sb t2, 0(s0)
88     jr ra
89

```

scanNumbers:

in questa funzione viene ricomposto il numero da terminale e memorizzato all'interno della locazione di memoria specificata. Con la prima istruzione, si memorizza nel registro t3 il byte successivo all'indirizzo base dell'indirizzo della variabile di appoggio per la funzione scan. Successivamente viene inizializzato i registri t0 e t2 al valore 0 e si entra nel ciclo forScanNumbers. All'interno del forScanNumbers viene caricato il valore del primo carattere numerico salvato nella variabile di appoggio appScan. Commutato il carattere in numero valore, togliendo l'offset Ascii di 48 passato come immediato, si aggiunge un'unità al registro t3 per puntare al prossimo carattere memorizzato. Si caricano in seguito i valori 100, 10 ed 1 nei rispettivi registri t5, t6 e t1. Si parte pertanto con un secondo ciclo forPot per associare al numero appena trovato il giusto valore in potenza. Il ciclo forPot viene ripetuto tante volte quante il valore corrente dell'indice del forScanNumbers più uno. Infatti ad ogni iterazione del ciclo forPot viene valutato se il proprio indice è diverso da 0 e quindi ad ogni iterazione viene diviso il registro t5 per t6. In questo modo in base a quante volte il ciclo itera, il valore del registro t5 sarà minore e dunque anche il peso associato al numero letto sarà inferiore. Alla fine del ciclo forPot infatti il registro t4 è moltiplicato per t5 e viene sommato al registro t2 che contiene il numero finale da salvare in memoria. Svolto il ciclo forScanNumbers viene caricato il segno letto ed uno di confronto per capire se il numero inserito è negativo o positivo. Tramite la beq infatti si prende il numero originale o quello

opposto. Ultima operazione sta nel salvare il valore del registro t2, utilizzando il registro di appoggio s0 contenente l'indirizzo per memorizzare il dato; da cui è possibile ritornare alla funzione chiamante e successivamente al main.

CONCLUSIONI

Il progetto approfondito dimostra come tramite alcune funzioni e nozioni base si possa creare un sistema completo per debug, sviluppo e manutenzione di programmi in codice assembly. Riuscire infatti ad avere un'interfaccia semplice e diretta, permette di non accedere al debugger qemu e risparmiare tempo importante per la stesura del codice. Il debugger sebbene utile risulta essere scomodo in quanto ogni volta bisogna accedervi e ripercorrere il programma. Nonostante sia una conoscenza base utile e fondamentale nei primi passi di apprendimento, con codici più complessi può comportare uno spreco di tempo fondamentale. Preparare moduli e funzioni pre-impostati e collaudati permette di avere un codice leggibile e facile riducendo di molto i tempi di debug. Anche a basso livello è infatti da notare come una programmazione strutturata aiuti l'utente in ogni progetto.