

CALCOLATORI

Esercizi al calcolatore Risc-V

Marco Roveri

marco.roveri@unitn.it



UNIVERSITÀ DEGLI STUDI DI TRENTO

**Dipartimento di Ingegneria
e Scienza dell'Informazione**

Obiettivi

- In questa lezione vedremo
 - Come installare una tool-chain GNU per cross-compilazione di programmi C/RISC-V in ambiente Linux X86
 - Come compilare un programma RISC-V e linkarlo ad un driver C per creare un eseguibile
 - Come visualizzare i diversi segmenti dei file oggetto
 - Come eseguire in un simulatore il programma generato
 - Come debuggare attraverso il simulatore e/o il GNU debugger gdb

GNU Cross Toolchain

- Un cross-compilatore è un ambiente che consente di prendere un programma (e.g. un programma C o C++) e di generare un eseguibile per una piattaforma diversa
 - Compilazione in host X di un eseguibile MS Windows
 - Compilazione in host X di un eseguibile Mac Os X
 - Compilazione in host X di un eseguibile per ARM
 - Compilazione in host X di un eseguibile per MIPS
 - Compilazione in host X di un eseguibile per RISC-V
- Dove host X potrebbe essere Linux su X86, Linux ARM, Mac OS X, MS Windows, ...
- Per RISC-V:
 - <https://github.com/riscv/riscv-gnu-toolchain>



Emulatore di Processore

- Un emulatore di processore è un programma che esegue su un host computer con una certa architettura (e.g. X86) e consente di emulare una altra architettura (e.g. ARM, RISC-V, MIPS, SPARC, ...) consentendo di eseguire programmi compilati per l'architettura emulate
 - VMWare/VirtualBox: emulatori commerciali per X86
 - QEmu (www.qemu.org): è un generic e open source machine emulator and virtualizer
 - ✓ QEMU può virtualizzare x86, PowerPC, 64-bit POWER, S390, 32-bit, 64-bit ARM e RISC-V
 - Spike RISC-V ISA Simulator che implementa un modello funzionale del processore RISC-V, e su cui gira un RISC-V Proxy Kernel, pk, un ambiente di esecuzione leggero che consente di eseguire statically-linked RISC-V ELF eseguibili.
 - ✓ È stato progettato per supportare limitate I/O capability trasformandole in equivalenti chiamate a istruzioni dell' host computer
 - ✓ <https://github.com/riscv/riscv-tools>

Pre-compiled RISC-V GNU toolchain and spike

- Una versione pre-compilate per Linux X86 della RISC-V GNU toolchain (gcc, as, ld, libc, ...) e del simulatore spike (e pk) generate da
 - <https://github.com/riscv/riscv-gnu-toolchain>
 - <https://github.com/riscv/riscv-tools>
- E' scaricabile da (grazie a Prof. Matthieu Moy)
 - <https://matthieu-moy.fr/spip/?Pre-compiled-RISC-V-GNU-toolchain-and-spike&lang=en>
- L'archivio deve essere estratto nella directory /opt/riscv/ e richiederà circa 200 Mb di spazio disco.

Istruzioni installazione

-  Ubuntu 18.04 (e successive):
 - `wget 'https://matthieu-moy.fr/spip/IMG/gz/riscv.tar.gz' -O /tmp/riscv.tar.gz`
 - `cd /opt/`
 - `sudo tar -xzf /tmp/riscv.tar.gz; rm -f /tmp/riscv.tar.gz`
 - `sudo apt install libmpc3 device-tree-compiler guile-2.0-lib libpython-all-dev`
 - `export PATH=${PATH}:/opt/riscv/bin`
-  Fedora 30:
 - `wget 'https://matthieu-moy.fr/spip/IMG/gz/riscv-fedora.tar.gz' -O /tmp/riscv-fedora.tar.gz`
 - `sudo mkdir -p /home/tpetu/Enseignants/matthieu.moy/mif08/`
 - `cd /home/tpetu/Enseignants/matthieu.moy/mif08/`
 - `sudo tar -xzf /tmp/riscv-fedora.tar.gz; rm -f /tmp/riscv.tar.gz`
 - `export PATH=${PATH}:/home/tpetu/Enseignants/matthieu.moy/mif08/riscv/bin`

Compilazione di un programma

- Supponiamo di avere un programma C decomposto in due file

```
// main.c
#include <stdio.h>
```

```
int func_1(int x);
```

```
int main() {
    int a;
    a = func_1(10);
    printf("a = %d\n", a);
}
```

```
// func_1.c
int func_1(int i) {
    return i * i + i;
}
```

- `prompt> riscv64-unknown-elf-gcc -g -S func_1.c`
 - Genera assembly file `func_1.s`
- `prompt> riscv64-unknown-elf-gcc -g main.c func_1.s -o main`
 - Genera eseguibile `main` a partire da `main.c` e da assembly file `func_1.s`

Esecuzione e Debugging del programma

- `prompt> spike -d /opt/riscv/riscv64-unknown-elf/bin/pk main`
`bbl loader`
`a = 110`
- `prompt> riscv64-unknown-elf-gdb main`
`GNU gdb (GDB) 8.3.0.20190516-git`
`...`
`(gdb) target sim`
`load main`
`Loading section .text, size 0xbd84 lma 0x100b0`
`Loading section .rodata, size 0xd18 lma 0x1be40`
`...`
`Start address 0x100c6`
`(gdb) break func_1`
`(gdb) run`

Collegiamo il debugger ad un
simulatore della target architecture
(Risc-V in questo caso)

Esecuzione debugging di un programma assembler

- Supponiamo di dover scrivere un programma assembler che implementa una funzione `nome_func` che prende come argomento un array di interi e un intero che rappresenta la dimensione dell'array e ritorna la media dei valori dell'array
- Supponiamo di salvarlo in un file `nome_func.s` con struttura come sotto illustrato inserendo l'assembler che fa il calcolo della media al posto del <BODY DELLA FUNZIONE>

```
.text
    .globl nome_func
    .type  nome_func, @function
nome_func:
    .cfi_startproc
    <BODY DELLA FUNZIONE>
    .cfi_endproc
    .size  nome_func, .-nome_func
```

Esecuzione debugging di un programma assembler (cont.)

```
# a0 = A, a1 = n
li      a5,0          # indice i su array
li      a3,0          # accumulatore
loop:   bge    a5, a1, end  # condizione for
        slli    a4, a5, 2    # calcolo offset per A[i]
        add     a4, a0, a4    # calcolo indirizzo A[i]
        lw      a4, 0(a4)    # carico A[i]
        addw    a3, a4, a3    # accumulo valore letto
        addiw   a5, a5, 1    # incremento indice
        j       loop
end:    divw    a0, a3, a1    # calcolo media
ret
```

Esecuzione debugging di un programma assembler (cont.)

- Creiamo un driver main.c in C che chiami la funzione nome_func, e magari stampi a video i valori dell'array e la media calcolata:
 - Mettiamo nel file main.c un prototipo per specificare la segnatura della funzione nome_func

```
// main.c
#include <stdio.h>
void nome_func(int A[], int s);
int main() {
    int A[5] = {10, 3, 2, 1, 4};
    int media = 0;
    for(int j = 0; j < 5; j++) printf("A[%d] = %d\n", j, A[j]);
    media = nome_func(A, 5);
    printf("La media e': %d\n", media);
}
```

Esecuzione debugging di un programma assembler (cont.)

- `prompt> riscv64-unknown-elf-gcc -g main.c nome_func.s -o main`
 - Genera eseguibile main a partire da main.c e da assembly file nome_func.s
- A questo punto posso eseguirlo con spike e/o debuggarlo con gdb
- Questo meccanismo possiamo ripeterlo per qualunque altro programma assembly che vogliamo eseguire

Alcuni comandi GDB

- (gdb) run
Esegue il programma caricato
- (gdb) break function_name
inserisce un break point ad ogni chiamata della funzione function_name
- (gdb) step
Fa step in nella funzione che si sta per eseguire
- (gdb) finish
ritorna dalla funzione che si sta eseguendo e si ferma alla istruzione immediatamente successive alla chiamata
- (gdb) continue
Continua esecuzione fino al prossimo break point ammesso che ci sia
- (gdb) print var
Stampa il valore della variabile var

Alcuni comandi GDB (cont.)

- (gdb) print *var
Stampa il valore a cui punta la variabile var (supposta essere un puntatore ad un indirizzo di memoria)
- (gdb) print \$a4
Stampa contenuto registro \$a4
- (gdb) print *(\$a4)
equivalente ad 0(a4) stampa contenuto della memoria a cui punta il registro a4
- (gdb) print *(\$a4+4)
equivalente a 4(a4) stampa contenuto della memoria a cui punta il registro a4 con offset 4

Lista simboli in object file

- `prompt> nm file.o`
stampa i simboli nell'object file `file.o`
 - E.g.:
`prompt> riscv64-unknown-elf-gcc -c trova_d.c`
`prompt> riscv64-unknown-elf-nm trova_d.o`
....
0000000000000000 T main
 U printf
 U puts
 U trova_elemento

Stampa informazioni contenute in un object file

- `prompt> readelf -a file.o`
Stampa tutte (-a) le informazioni contenute in un object file
 - E.g.:
`prompt> riscv64-unknown-elf-gcc -c trova.s`
`prompt> riscv64-unknown-elf-readelf -a trova.o`
- `prompt> objdump -d -x file.o`
Stampa tutti gli headers (-x) e disassembla (-d) mostrando le istruzioni assembly contenute nell'object file.
 - E.g.:
`prompt> riscv64-unknown-elf-gcc -c trova.s`
`prompt> riscv64-unknown-elf-objdump -d -x trova.o`

trova.o: file format elf64-littleriscv
trova.o
architecture: riscv:rv64, flags 0x00000011:
HAS_RELOC, HAS_SYMS
start address 0x0000000000000000

Sections:

Idx	Name	Size	VMA	LMA	File off	Algn
0	.text	0000001c	0000000000000000	0000000000000000	00000040	2**1
	CONTENTS, ALLOC, LOAD, RELOC, READONLY, CODE					
1	.data	00000000	0000000000000000	0000000000000000	0000005c	2**0
	CONTENTS, ALLOC, LOAD, DATA					

...

SYMBOL TABLE:

00000000000000002 | .text0000000000000000 loop

..

000000000000000016 | .text0000000000000000 end_1

000000000000000018 | .text0000000000000000 end

...

00000000000000000 g F .text 00000000000000001c trova_elemento

...

Disassembly of section .text:

0000000000000000 <trova_elemento>:

0: 4781 li a5,0

0000000000000002 <loop>:

2: 00b7da63 bge a5,a1,16 <end_1>

2: R_RISCV_BRANCH end_1

6: 00279713 slli a4,a5,0x2

a: 972a add a4,a4,a0

c: 4318 lw a4,0(a4)

e: 00c70563 beq a4,a2,18 <end>

e: R_RISCV_BRANCH end

12: 2785 addiw a5,a5,1

14: b7fd j 2 <loop>

14: R_RISCV_RVC_JUMP loop

0000000000000016 <end_1>:

16: 57fd li a5,-1

0000000000000018 <end>:

18: 853e mv a0,a5

1a: 8082 ret

Alcuni link utili

- Risorse RISC-V
 - <https://riscv.org/software-status/>
- Come fare debugging con GDB e spike
 - <https://linux-audit.com/elf-binaries-on-linux-understanding-and-analysis/>
 - <https://github.com/riscv/riscv-isa-sim>
- Tutorial debug con gdb
 - <https://www.cs.umd.edu/~srhuang/teaching/cmsc212/gdb-tutorial-handout.pdf>
 - <https://www.youtube.com/watch?v=bWH-nL7v5F4>
- ARM GNU Tool chain
 - <https://developer.arm.com/tools-and-software/open-source-software/developer-tools/gnu-toolchain/gnu-rm/downloads>
 - https://www.acmesystems.it/arm9_toolchain
- Debugging ARM code con QEMU e cross-compilazione
 - <http://doppioandante.github.io/2015/07/10/Simple-ARM-programming-on-linux.html>