20/21 October 2021

TESTING AND FAULT TOLERANCE

Laboratory Session 3: “Test Programs: Toolchain and Debug”

Student Name	Student Surname	Student ID
Salvatore Gabriele	La Greca	S281589

SAVE THIS FILE IN PDF AND SUBMIT IT

Politecnico di Torino

Riccardo Cantoro (riccardo.cantoro@polito.it) Nick Deligiannis (nikolaos.deligiannis@polito.it)

[A] Cross-Compilation & Binutils

For the purposes of this exercise we are going to use a RISC-V processor (implementation: RI5CY). We are going to explore the architecture of the system while we attempt to execute some code (assembly & C) on the core and observe the effects of the code execution. The first step is to get familiar with the gnu-toolchain of the RISC-V. Inside the folder there is a compilation bash script that invokes the `riscv32 gcc` binary. Look at the code and observe the `Makefile` under the sbst directory. You can see that we use a custom *linker* script for our compilation. Linking is the process of collecting and combining various pieces of code and data into a single file that can be loaded (copied) into memory and executed. The linker script is also responsible of organizing the memory chunk (memory map) that will be given (typically from the OS) to the executable into regions that nest sections.

[A.1] Tasks:

1. Inspect the linker script and report the starting and ending addresses of:
   a. The region in which the code is stored
   b. The region in which the data are stored
2. Which region nests the TEST_DATA section?

```
1.a START = 0x00000000, LENGTH = 0x20000
1.b START = 0x00200000, LENGTH = 0x04000
2   It's in the so called dataram
3   It's in the so called instrram
```

3. Which region nests the TEST_RODATA section?

💡 You can find useful information about linker scripting here

The next step is to compile the sbst program to observe its structure. Invoke the compilation bash script as:

- `bash compile_sbst.sh`

Observe that the files `sbst.hex` and `sbst.elf` have been generated in the sbst directory. We are interested in the `.elf` file. ELF stands for Executable and Linkable format and a file using the format consists of an ELF header, followed by a program header table or a section header table, or both. The ELF header is always at offset zero of the file. The program header table and the section header table's offset in the file are defined in the ELF header. The GNU `toolchain` comes with a very powerful tool that can assist with the understanding of such files. This binary is called `readelf`.

[A.2] Tasks:

1. Invoke the `riscv32 readelf` and report the starting and ending addresses of:
   a. The TEST_DATA section.
   b. The TEST_RODATA section.
2. Invoke the `riscv32 readelf` to read the symbol table and carefully observe the symbols:
   a. `__TEST_DATA_START` and `__TEST_DATA_END`.
   b. `__TEST_RODATA_START` and `__TEST_RODATA_END`.

   What is their connection with the TEST_DATA and TEST_RODATA sections respectfully?

```
1.a TEST_DATA   START 0x00200000 END 0x00200003
1.b TEST_RODATA START 0x00001f60 END 0x00001f73

2. They are addresses pointing at the start and after the end address
respectively for the two mentioned sections. They delimit the boundaries of
the two sections.
```

- Instead of the `readelf` binary you can also use the memory map file that has been created in the `sbst` directory after the compilation (`.map` file)

One more useful tool that exists in the GNU `toolchain` is `objdump`. The binary provides many functionalities, but we will use it primarily as a disassembler to look at the structure of our executable programs for debugging purposes. For instance, you can use it as:

```
riscv32-unknown-elf-objdump -D sbst.elf | less
```
or
```
riscv32-unknown-elf-objdump -D sbst.elf > objdump.out && gedit objdump.out
```

and iterate through the disassembly of all sections of the `.elf` file.

[B] The crt0 & Boot-time self-test

The `crt0` (which stands for C RunTime 0) is a set of execution startup routines linked into a C program that performs any initialization work required before calling the program's main function. Under your `sbst` directory you can find the file `crt0.S` that contains the initialization phase for our environment. You can see that many vital parts of the processor and the memory sections are initialized there. For instance, the `register file` of the processor, the `stack pointer`, the `global pointer`, the allocation of memory for the `.bss` section etc.

[B.1] Tasks:

1. Draw below the chain of function calls that take place until control reaches the `main` function of our program.
2. What is the name of the initial (the very first) routine that is invoked?
3. Which is the address of the routine?

1) atexit → __libc_init_array → _sbst_boot → main
2) _start
3) 0x80

- You can use the `objdump` to assist you

In many digital systems, it is common to have a test procedure (self-test) that takes place during their power-on/booting. This can be proven beneficial especially for in-field testing where no direct access to the device is possible.

[C] Assembly self-test routine

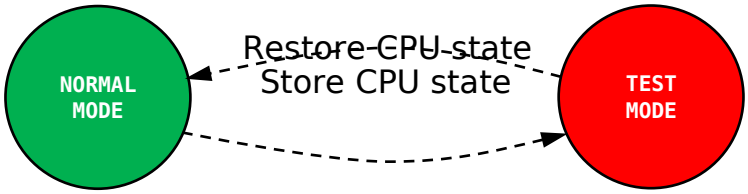
While assuming such a scenario we are going to compose a short testing routine to be executed, right before the invocation of our `main` function, to target the `register file` of the processor and then we are going to write the result (signature) of the test on a specific memory address. This code will be written in the `sbst_boot.S` assembly file that is available in the `sbst` directory. This is the main topic of the exercise. Now that you have a basic understanding of the structure of the core and the different procedures that take place in software, you are asked to compose a small "test" routine for the `register file` of the processor. The test is given in pseudocode format below:

[C.1] Tasks:

1. Implement the test routine:

```
1.store the current state of the machine
2.if (interrupts == ON)
   A.disable interrupts
   B.remember to restore them at step 7
3.For i ← 1 to 31:
   A.REG[i] ← RF_PATTERN[i]
4.signature := 0x0
5.For i ← 1 to 31:
   A.signature ← signature +¿ REG[i]
6.test_result ← signature
7.if interrupts were ON at step 2
   A.re-enable interrupts
8.restore the state of the machine
```

Typically, when you launch a test routine you have first to preserve the state of the machine in order to be able to return to the normal operation mode. To better understand the test procedure, you can interpret it as a finite-state machine:



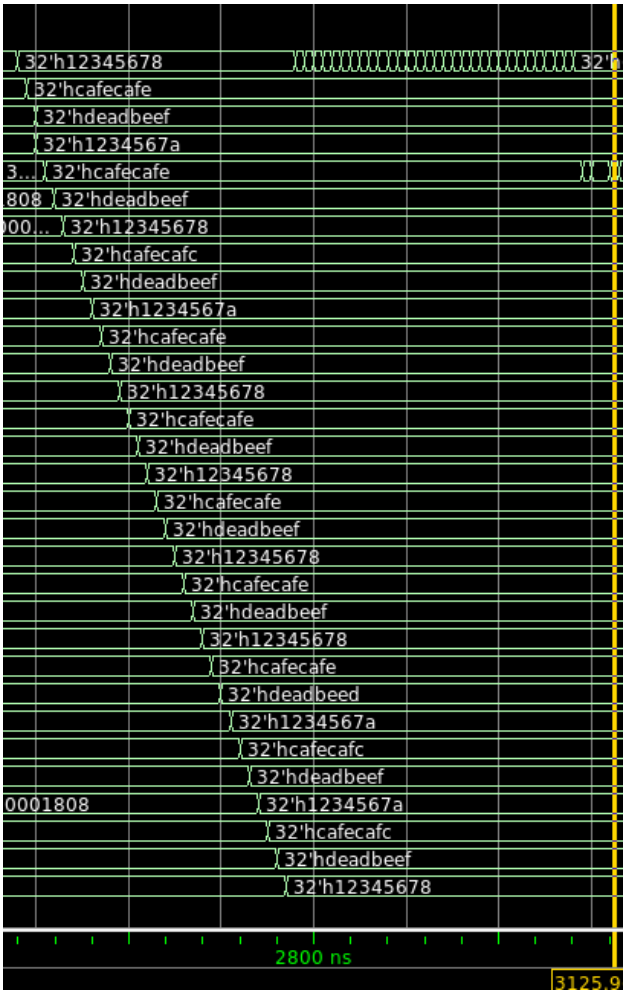
The steps 1,2 and 7,8 respectively, implement the transitions back and forth from the normal state of the processor to the test routine and back.

All of your code should be written inside the `sbst_boot.S` file in assembly. You will find useful comments to guide you through each step of the routine.

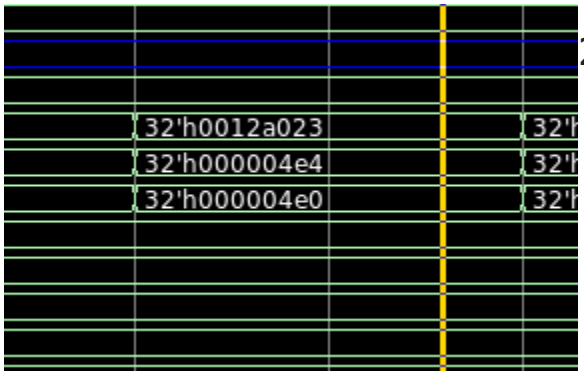
[C.2] Tasks:

1. Run a logic simulation on the RTL of the processor and provide a snapshot of the waveforms of the register file, showing the stamps in which, the values are written in the registers.
2. In the waveform of the logic simulation, find the instruction that stores the value of the signature in the designated memory address and provide a screenshot.

1)


32'h12345678
32'hcafecafe
32'hdeadbeef
32'h1234567a
32'hcafecafe
32'hdeadbeef
32'h12345678
32'hcafecafc
32'hdeadbeef
32'h1234567a
32'hcafecafe
32'hdeadbeef
32'h12345678
32'hcafecafe
32'hdeadbeef
32'h12345678
32'hcafecafe
32'hdeadbeef
32'h12345678
32'hcafecafe
32'hdeadbeed
32'h1234567a
32'hcafecafc
32'hdeadbeef
32'h1234567a
32'hcafecafc
32'hdeadbeef
32'h12345678
2800 ns

2)


32'h0012a023
32'h000004e4
32'h000004e0

Appendix A: Files of LAB3

- **All files listed here are included in your remote /home directory under riscv_testing_2021 folder.**

Filename	Description
`compile_sbst.sh`	Bash script invoking the rv32 compiler
`compile_testbench.sh`	Bash script compiling the RTL of RI5CY for vsim
`run_rtl_nogui.sh`	Bash script invoking Questasim in shell mode
`run_rtl_gui.sh`	Bash script invoking Questasim with gui.
`sbst/crt0.s`	C RunTime 0 ASM code for sbst
`sbst/link.ld`	Linker script for our sbst
`sbst/sbst_boot.s`	ASM for our sbst routine
`run_rtl_[gui|nogui].sh`	Invoke Questasim with/without GUI for RTL sim
`run_gate_[gui|nogui].sh`	Invoke Questasim with/without GUI for Gate sim

Appendix B: RISC-V Register ABI

Register	ABI Name	Description
x0	zero	Hardwired zero
x1	ra	Return Address
x2	sp	Stack pointer
x3	gp	Global pointer
x4	tp	Thread pointer
x5-7	t0-2	Temporaries
x8	s0/fp	Saved register/frame pointer
x9	s1	Saved
x10-11	a0-1	Function arguments/return values
x12-17	a2-7	Function arguments
x18-27	s2-11	Saved registers

x28-31	t3-6	Temporaries

Appendix C: CSRs & Instructions in ASM

The core we have supports control and status registers (CSRs) and thus, control status assembly instructions to check for example the status of the core, enable debugging for the program counter etc.

After inspecting the documentation of the platform you should be able to identify the register of interest for the handling of interrupts within the core.

You can use the following assembly instructions to read/set/unset bits to a specific CSR:

- `csrrs rd, CSR, r1: // csr-read-set`
  - `If the r1 register is set to x0, then the instruction does not modify the CSR and simply reads its value and stores it into rd.`
  - `Otherwise, the value of r1 register is used as a bit-mask. Each high bit in r1 register corresponds to the respective CSR bit being set (if that bit is writable)`
- `csrrc rd, CSR, r1: // csr-read-clear`
  - `If the r1 register is set to x0, then the instruction does not modify the CSR and simply reads its value and stores it into rd.`
  - `Otherwise, the value of r1 register is used as a bit-mask. Any bit that is high on r1 will cause the corresponding bit of the CSR to be cleared (if that bit is writable)`

- You can search in the RI5CY manual for information about the supported CSRs and locate the one that you need.
- You can search in the RISC-V User Manual for extra information about CSRs and CSR instructions.

Appendix D: RISC-V assembler directives

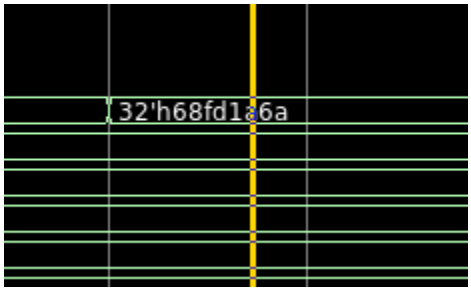
Here is a table of some of the RISC-V directives and relocation functions for the assembler and their context.

Directive	Arguments	Context
`.align`	integer	align to power of 2
`.file`	“filename”	emit filename file local symbol table
`.globl`	symbol_name	emit symbol_name to symbol table (scope global)
`.local`	symbol_name	emit symbol_name to symbol table (scope local)
`.comm`	symbol_name,size,align	emit common object to .bss section
`.common`	symbol_name,size,align	emit common object to .bss section
`.ident`	“string”	accepted for source compatibility
`.section`	[{.text,.data,.rodata,.bss}]	emit section and make current
`.size`	symbol,symbol	accepted for source compatibility
`.text`		emit .text section (if not present) and make current
`.data`		emit .data section (if not present) and make current
`.rodata`		emit .rodata section (if not present) and make current
`.bss`		emit .bss section (if not present) and make current
`.string`	“string”	emit string
`.long`	expression [,expression]*	32-bit comma separated words
RELOCATION FUNCTIONS		
Notation	**Description**	**Instruction / Macro**
`%hi(symbol)`	absolute (HI20)	`lui`
`%lo(symbol)`	absolute (LO12)	`load,store,add`

- You can find a complete table here

BONUS: For those who dare!

In LAB1 we have learned (amongst others) how to perform a fault injection in the environment of `Questasim` by utilizing the `force` command. Attempt to perform a fault injection of a stuck-at-0 fault in the 5th bit of register 7. How does this affect the signature value of our sbst procedure? Attach a snapshot of the waveforms after the injection of the fault, showing the register that reads the signature value from the memory.




```
Without fault: 0x68fd1a6a
With fault:    0x68fd1a4a
```