Computer Architecture - Lab Assignment 1

RISC-V instruction set architecture and programming of Nios V/m processor

This is an introductory exercise that involves Intel/Altera's Nios V/m soft processor and its RISC-V assembly language. It uses a simple computer hardware system, called the *DE0-Nano Basic Computer*, which includes the Nios V/m processor. The hardware system is implemented as an electronic circuit that is downloaded into the FPGA device on the *Terasic DE0-Nano* board ([1]). This exercise illustrates how programs written in the RISC-V assembly language can be executed on the DE0-Nano board.

To prepare for this exercise you have to know the Nios V/m processor architecture and its RISC-V assembly language ([2, 3]). This lab assignment consists of four parts. Part I below describes the procedure for compiling, linking RISC-V architecture assembler programs, and running the programs on the DE0-Nano board.

Part I. Executing an example program

In this part we will use the **Nios V Command Shell** to download the DE0-Nano Basic Computer SoC circuit into the FPGA device and execute a sample program. This tool is part of the *Intel/Altera Quartus Prime Standard 23.1 Design Suite* ([3]).

Perform the following:

1. Turn on the power to the DE0-Nano board.
2. Open the Nios V Command Shell, which leads to the window in Figure 1.

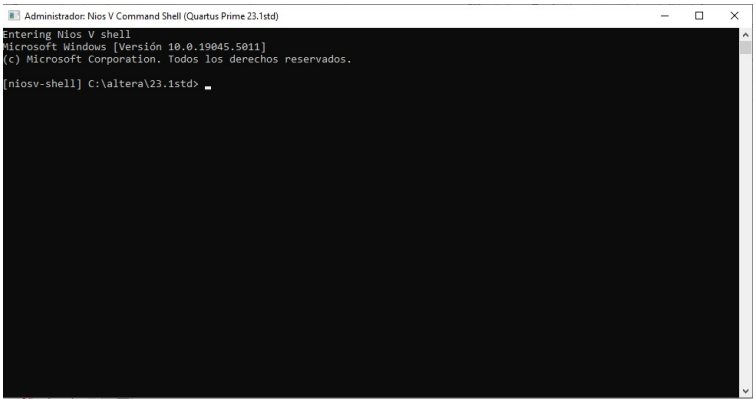


Figure 1: The Nios V Command Shell window.

To run an application program it is necessary to create a new **BSP project**.

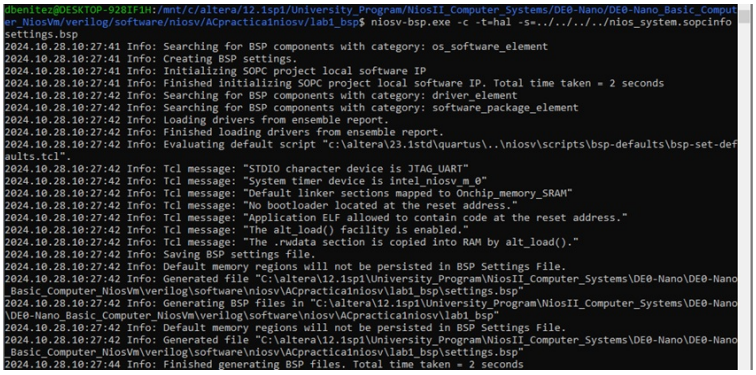
3. Create a new BSP directory called for example: `lab1_bsp`.

4. Select a predesigned *System-on-Chip* file: `nios_system_22jul24.sopcinfo`. This file was obtained by using the *Platform Designer* tool ([3]). The computer hardware system integrates a Nios V/m soft processor and a 8 KB on-chip SRAM memory, in addition to an I/O controller for the LEDs available in the board.

5. Execute the following command in the Nios V Command Shell window.

```
$ cd lab1_bsp
$ sh
$ niosv-bsp.exe -c -t=hal -s=nios_system_22jul24.sopcinfo settings.bsp
```

The output provides two files called: `settings.bsp` and `linker.x`. Figure 2 shows the messages displayed in the Nios V Command Shell window.



```
dbenitez@DESKTOP-928IF1H:/mnt/c/altera/12.1sp1/University_Program/NiosII_Computer_Systems/DE0-Nano/DE0-Nano_Basic_Comput
er_NiosVm/verilog/software/niosv/ACpractica1niosv/lab1_bsp$ niosv-bsp.exe -c -t=hal -s=../../../../nios_system.sopcinfo
settings.bsp
2024.10.28.10:27:41 Info: Searching for BSP components with category: os_software_element
2024.10.28.10:27:41 Info: Creating BSP settings.
2024.10.28.10:27:41 Info: Initializing SOPC project local software IP
2024.10.28.10:27:41 Info: Finished initializing SOPC project local software IP. Total time taken = 2 seconds
2024.10.28.10:27:42 Info: Searching for BSP components with category: driver_element
2024.10.28.10:27:42 Info: Searching for BSP components with category: software_package_element
2024.10.28.10:27:42 Info: Loading drivers from ensemble report.
2024.10.28.10:27:42 Info: Finished loading drivers from ensemble report.
2024.10.28.10:27:42 Info: Evaluating default script "c:\altera\23.1std\quartus\..\niosv\scripts\bsp-defaults\bsp-set-def
aults.tcl".
2024.10.28.10:27:42 Info: Tcl message: "STDIO character device is JTAG_UART"
2024.10.28.10:27:42 Info: Tcl message: "System timer device is intel_niosv_m_0"
2024.10.28.10:27:42 Info: Tcl message: "Default linker sections mapped to Onchip_memory_SRAM"
2024.10.28.10:27:42 Info: Tcl message: "No bootloader located at the reset address."
2024.10.28.10:27:42 Info: Tcl message: "Application ELF allowed to contain code at the reset address."
2024.10.28.10:27:42 Info: Tcl message: "The alt_load() facility is enabled."
2024.10.28.10:27:42 Info: Tcl message: "The .rwdata section is copied into RAM by alt_load()."
2024.10.28.10:27:42 Info: Saving BSP settings file.
2024.10.28.10:27:42 Info: Default memory regions will not be persisted in BSP Settings File.
2024.10.28.10:27:42 Info: Generated file "C:\altera\12.1sp1\University_Program\NiosII_Computer_Systems\DE0-Nano\DE0-Nano
_Basic_Computer_NiosVm\verilog\software\niosv\ACpractica1niosv\lab1_bsp\settings.bsp"
2024.10.28.10:27:42 Info: Generating BSP files in "C:\altera\12.1sp1\University_Program\NiosII_Computer_Systems\DE0-Nano
\DE0-Nano_Basic_Computer_NiosVm\verilog\software\niosv\ACpractica1niosv\lab1_bsp"
2024.10.28.10:27:42 Info: Default memory regions will not be persisted in BSP Settings File.
2024.10.28.10:27:42 Info: Generated file "C:\altera\12.1sp1\University_Program\NiosII_Computer_Systems\DE0-Nano\DE0-Nano
_Basic_Computer_NiosVm\verilog\software\niosv\ACpractica1niosv\lab1_bsp\settings.bsp"
2024.10.28.10:27:44 Info: Finished generating BSP files. Total time taken = 2 seconds
```

Figure 2: Messages displayed after executing the `niosv-bsp.exe` command.

6. Now, create a new directory for the RISC-V assembler program written in RISC-V assembly language and its building, for example: `lab1_bin`.

7. Copy the sample program `lab1_part1.s` to this directory.

8. The source file `lab1_part1.s` contains the application program. This file specifies the starting point in the selected application program. The default symbol is start, which is used in the selected sample program.

9. Execute the following command in the Nios V Command Shell window for assembling the source code.

```
$ cd lab1_bin
$ riscv32-unknown-elf-as.exe lab1_part1.s -o lab1_part1.s.obj
```

If no error are encountered, the assembler provides an output file called: `lab1_part1.s.obj`.

10. Execute the following command in the Nios V Command Shell window for linking and reporting the program.

```
$ riscv32-unknown-elf-ld.exe -g -T ../lab1_bsp/linker.x -nostdlib
-e _start -u _start --defsym __alt_stack_pointer=0x08001F00 --defsym
__alt_stack_base=0x08002000 --defsym __alt_heap_limit=0x8002000 --defsym
__alt_heap_start=0x8002000 -o lab1_part1.elf lab1_part1.s.obj
$ niosv-stack-report.exe -p riscv32-unknown-elf- lab1_part1.elf
```

If no error are encountered, the linker provides an output file called: `lab1_part1.elf`. Figure 3 shows the messages displayed in the Nios V Command Shell window.

11. Execute the following command in the Nios V Command Shell window for generating a disassembled RISC-V program.

```
$ riscv32-unknown-elf-objdump.exe -Sdtx lab1_part1.elf > lab1_part1.elf.objdump
```

```
dbenitez@DESKTOP-928IF1H:/mnt/c/altera/12.1sp1/University_Program/NiosII_Computer_Systems/DE0-Nano/DE0-Nano_Basic_Compu
er_NiosVm/verilog/software/niosv/ACpractica1niosv/lab1_bin$ make
riscv32-unknown-elf-ld.exe  -g -T ../practica1_bsp/linker.x -nostdlib -e _start -u _start --defsym __alt_stack_pointer=0
x08001F00 --defsym __alt_stack_base=0x08002000 --defsym __alt_heap_limit=0x8002000 --defsym __alt_heap_start=0x8002000
-o lab1_part1.elf lab1_part1.s.obj
riscv32-unknown-elf-ld.exe: warning: lab1_part1.elf has a LOAD segment with RWX permissions
niosv-stack-report.exe -p riscv32-unknown-elf- lab1_part1.elf
lab1_part1.elf
 * 184 B - Program size (code + initialized data).
 * 256 B - Free for stack.
 * 0 B - Free for heap.
riscv32-unknown-elf-objdump.exe  -Sdtx lab1_part1.elf > lab1_part1.elf.objdump
riscv32-unknown-elf-objcopy.exe -O binary lab1_part1.elf lab1_part1.hex
```

Figure 3: Messages displayed after executing the `riscv32-unknown-elf-ld.exe` and `niosv-stack-report.exe` commands.

The output file is called: `lab1_part1.elf.objdump`. Figure 4 shows the RISC-V machine instructions, addresses and disassembled instructions that are included in the `lab1_part1.elf.objdump` file.

```
lab1_part1.elf
architecture: riscv:rv32, flags 0x00000112:
EXEC_P, HAS_SYMS, D_PAGED
start address 0x08000000

Program Header:
0x70000003 off    0x000010b8 vaddr 0x00000000 paddr 0x00000000 align 2**0
         filesz 0x0000002e memsz 0x00000000 flags r--
    LOAD off    0x00001000 vaddr 0x08000000 paddr 0x08000000 align 2**12
         filesz 0x000000b4 memsz 0x000000b4 flags r-x
    LOAD off    0x000010b4 vaddr 0x080000b4 paddr 0x080000b8 align 2**12
         filesz 0x00000004 memsz 0x00000004 flags rw-
    LOAD off    0x000000bc vaddr 0x080000bc paddr 0x080000bc align 2**12
         filesz 0x00000000 memsz 0x00000000 flags rw-

Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .exceptions   00000000  08000000  08000000  000010b8  2**0
                  CONTENTS
  1 .text         000000b4  08000000  08000000  00001000  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  2 .rodata       00000000  080000b4  080000bc  000010b8  2**0
                  CONTENTS, ALLOC, LOAD, DATA
  3 .rwdata       00000004  080000b4  080000b8  000010b4  2**0
                  CONTENTS, ALLOC, LOAD, DATA
  4 .bss          00000000  080000bc  080000bc  000010bc  2**0
                  ALLOC
  5 .Onchip_memory_SRAM 00000000  080000bc  080000bc  000010b8  2**0
                  CONTENTS
  6 .Onchip_memory 00000000  09000020  09000020  000010b8  2**0
                  CONTENTS
  7 .riscv.attributes 0000002e  00000000  00000000  000010b8  2**0
                  CONTENTS, READONLY
SYMBOL TABLE:
08000000 l    d  .exceptions  00000000 .exceptions
08000000 l    d  .text  00000000 .text
080000b4 l    d  .rodata  00000000 .rodata
080000b4 l    d  .rwdata  00000000 .rwdata
080000bc l    d  .bss 00000000 .bss
080000bc l    d  .Onchip_memory_SRAM  00000000 .Onchip_memory_SRAM
09000020 l    d  .Onchip_memory 00000000 .Onchip_memory
00000000 l    d  .riscv.attributes  00000000 .riscv.attributes
00000000 l    df *ABS*  00000000 lab1_part1.s.obj
080000b4 l       .rwdata  00000000 LEDG_bits
08000024 l       .text  00000000 DO_DISPLAY
```

```
Disassembly of section .text:

08000000 <_start>:
 8000000: 10000837          lui a6,0x10000
 8000004: 01080813          addi  a6,a6,16 # 10000010 <__alt_mem_Onchip_memory+0x7000010>
 8000008: 100007b7          lui a5,0x10000
 800000c: 04078793          addi  a5,a5,64 # 10000040 <__alt_mem_Onchip_memory+0x7000040>
 8000010: 100008b7          lui a7,0x10000
 8000014: 05088893          addi  a7,a7,80 # 10000050 <__alt_mem_Onchip_memory+0x7000050>
 8000018: 00000997          auipc s3,0x0
 800001c: 09c98993          addi  s3,s3,156 # 80000b4 <__tdata_end>
 8000020: 0009a303          lw  t1,0(s3)

08000024 <DO_DISPLAY>:
 8000024: 0007a203          lw  tp,0(a5)
 8000028: 0008a283          lw  t0,0(a7)
 800002c: 04028663          beqz  t0,8000078 <NO_BUTTON>
 8000030: 00020313          mv  t1,tp
 8000034: 00400533          add a0,zero,tp
 8000038: 00800593          li  a1,8
 800003c: 064000ef          jal ra,80000a0 <rotl>
 8000040: 00050233          add tp,a0,zero
 8000044: 00436333          or  t1,t1,tp
 8000048: 00400533          add a0,zero,tp
 800004c: 00800593          li  a1,8
 8000050: 050000ef          jal ra,80000a0 <rotl>
 8000054: 00050233          add tp,a0,zero
 8000058: 00436333          or  t1,t1,tp
 800005c: 00400533          add a0,zero,tp
 8000060: 00800593          li  a1,8
 8000064: 03c000ef          jal ra,80000a0 <rotl>
 8000068: 00050233          add tp,a0,zero
 800006c: 00436333          or  t1,t1,tp

08000070 <WAIT>:
 8000070: 0008a283          lw  t0,0(a7)
 8000074: fe029ee3          bnez  t0,8000070 <WAIT>

08000078 <NO_BUTTON>:
 8000078: 00682023          sw  t1,0(a6)
 800007c: 00600533          add a0,zero,t1
 8000080: 00100593          li  a1,1
 8000084: 01c000ef          jal ra,80000a0 <rotl>
 8000088: 00050333          add t1,a0,zero
 800008c: 000253b7          lui t2,0x25
 8000090: 9f038393          addi  t2,t2,-1552 # 249f0 <_start-0x7fdb610>
```

Figure 4: Content of the disassembled file called: `lab1_part1.elf.objdump`.

Now, it is needed to download the soft SoC system associated with this program onto the DE0-Nano board. Make sure that the power to the DE0-Nano board is turned on.

12. Execute the following command in the Nios V Command Shell window to test if the USB connection is established between the host computer and DE0-Nano board.

```
$ jtagconfig.exe
```

The following message must appear in the Nios V Command Shell window. In the case the message does not show, repeat again the command.

Output message

```
1) USB-Blaster [USB-0]
020F30DD 10CL025(Y|Z)/EP3C25/EP4CE22
```

13. Execute the following command in the Nios V Command Shell window for programming the FPGA circuit that is integrated in the DE0-Nano board using the `.sof` configuration file.

```
$ quartus_pgm.exe -c 1 -m JTAG -o "p;DE0_Nano_Basic_Computer_22jul24.sof@1"
```

Note the change in state of the blue LEDs on the DE0-Nano board that correspond to info messages *LOAD* and *GOOD*. These LEDs will blink as the `.sof` file is being downloaded into the FPGA. Figure 5 shows the messages displayed on the screen.

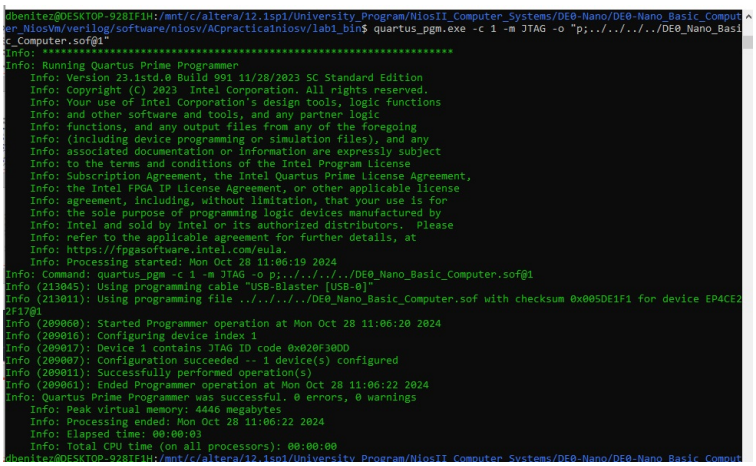


Figure 5: Messages displayed on the screen after configuring the FPGA circuit integrated into the DE0-Nano board with the SoC configuration called *DE0-Nano Basic Computer*.

14. Having downloaded the `sof` configuration called *DE0-Nano Basic Computer* into the FPGA chip on the DE0-Nano board, we can now load and run programs on this SoC computer. In the Nios V Command Shell window, execute the following command for downloading the `.elf` program into the main memory of the soft computer.

    ```
    $ niosv-download.exe -g lab1_part1.elf
    ```

    This command also run the `.elf` program.

15. Observe the LEDs located on the board are turning on and off quickly (see Figure 6). This test provides an indication that the DE0-Nano board is functioning properly. Stop the execution of the sample program by typing "CTRL+c" in the Command Shell window.

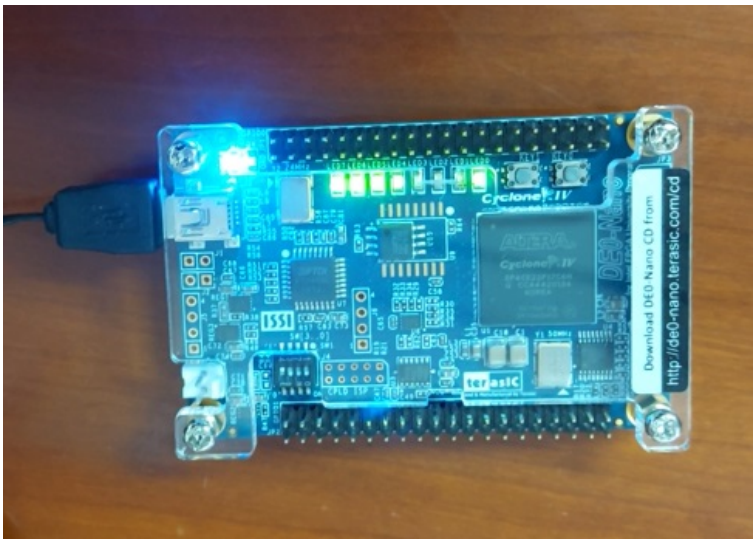


Figure 6: LEDs on the board are turning on and off after downloading and executing the `lab1_part1.elf` program.

Part II. Designing and debugging a simple program

Now, we will explore some features of the programming framework for Nios V by using a simple application program written in the RISC-V assembly language. Consider the program in Figure 7, which finds the

largest number in a list of 32-bit integers that is stored in the memory. This program is available in the file `lab1_part2.s`.

lab1_part2.s

```
.text /* executable code follows */

.global _start
_start:

/* initialize base addresses of parallel ports */
la x15, RESULT /* x15:  point to the start of data section */
lw x16, 4(x15) /* x16:  counter, initialized with n */
addi x17, x15, 8 /* x17:  point to the first number */
lw x18, (x17) /* x18:  largest number found */

LOOP:
addi x16, x16, -1 /* Decrement the counter */
beq x16, zero, DONE /* Finished if r5 is equal to 0 */
addi x17, x17, 4 /* Increment the list pointer */
lw x19, (x17) /* Get the next number */
bge x18, x19, LOOP /* Check if larger number found */
add x18, x19, zero /* Update the largest number found */
j LOOP

DONE:
sw x18, (x15) /* Store the largest number into RESULT */

STOP:
j STOP /* Remain here if done */

.data /* software variables follow */

RESULT:
.skip 4 /* Space for the largest number found */

N:
.word 7 /* Number of entries in the list */

NUMBERS:
.word 4, 5, 3, 6, 1, 8, 2 /* Numbers in the list */

.end
```

Figure 7: Example of RISC-V program for the Nios V/m soft processor.

Note that some sample data is included in this program. The data section of program starts at hex address `0x08000038`, as specified by the `.data` assembler directive and shown in `lab1_part2.elf.objdump` file. The first word (4 bytes) is reserved for storing the result, which will be the largest number found. The next word specifies the number of entries in the list. The words that follow give the actual numbers in the list.

Make sure that you understand the program in Figure 7 and the meaning of each instruction in it. Note the extensive use of comments in the program. You should always include meaningful comments in programs that you will write!

Perform the following steps for compiling and executing the program:

1. Create a new directory; we have chosen the directory named `lab1_part2`. Copy the file `lab1_part2.s` into this directory.

2. Now, follows the same steps done in Part I for assembling with `riscv32-unknown-elf-as.exe` file, linking with `riscv32-unknown-elf-ld.exe`, and report with `niosv-stack-report.exe` and `riscv32-unknown-elf-objdump.exe`.
3. Then, configure the DE0-Nano board using the `quartus_pgm.exe` command and `DE0_Nano_Basic_Computer_22jul24.sof` file and download the `.elf` program using `niosv-download`.

The next steps will use the GNU Debugger (GDB) for RISC-V processors. Open three Nios V Command Shell terminals.

Terminal-1 opens the Open On-Chip Debugger (OpenOCD). OpenOCD is part of the Nios V Command Shell tool chain and provides on-chip programming and debugging support with a layered architecture of JTAG interface.

Terminal-1: OpenOCD

```
$ cd lab1_part2
$ sh
$ openocd-cfg-gen ./niosv.cfg
$ openocd -f ./niosv.cfg
```

The second terminal is used as above in Part I for compiling, linking and configuring the FPGA device. In this section of the assignment, it has been prepared a `Makefile` file to automatize these steps. The options for the `Makefile` can be viewed using: `make -help`.

Terminal-2: Compile, link, configure and run

```
$ cd lab1_part2
$ sh
# compiling and linking
$ make
# configuring the FPGA
$ make configure
# load and run the program
$ make download
```

OpenOCD complies with the remote gdbserver protocol and, as such, can be used to debug remote targets. GDB works with OpenOCD. Terminal-3 opens the GDB tool that is part of the Nios V Command Shell tool chain. GDB uses several commands to interact with the Nios V architecture.

Terminal-3: GDB debugger

```
$ cd lab1_part2
$ sh
$ riscv32-unknown-elf-gdb
(gdb) target extended-remote localhost:3333
```

Note in Figure 8 the message provided by `gdb` after executing the `target` command. The current value of Program Counter register is `0x08000034`. This the memory address of the `jump LOOP` instruction.

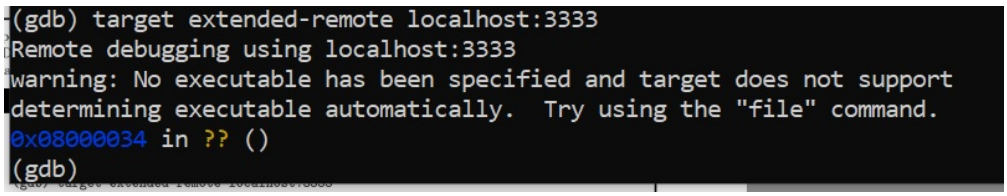
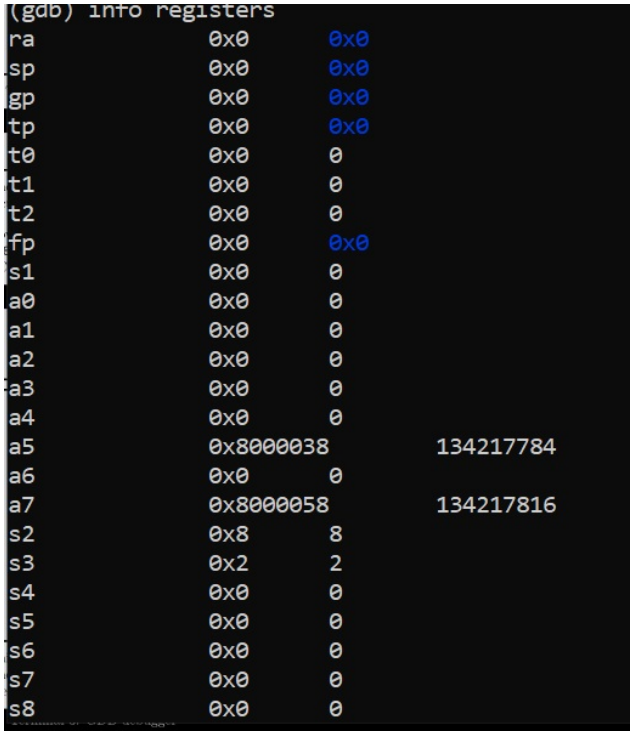


Figure 8: Output message after executing `target` command.

The data stored in registers can be viewed using the `info` command. Figure 9 shows the output provided by this command.

Terminal-3: GDB debugger

```
(gdb) info registers
```



```
(gdb) info registers
ra             0x0      0x0
sp             0x0      0x0
gp             0x0      0x0
tp             0x0      0x0
t0             0x0      0
t1             0x0      0
t2             0x0      0
fp             0x0      0x0
s1             0x0      0
a0             0x0      0
a1             0x0      0
a2             0x0      0
a3             0x0      0
a4             0x0      0
a5             0x8000038        134217784
a6             0x0      0
a7             0x8000058        134217816
s2             0x8      8
s3             0x2      2
s4             0x0      0
s5             0x0      0
s6             0x0      0
s7             0x0      0
s8             0x0      0
```

Figure 9: Output message after executing `info` command. The contents of some registers of the RISC-V instruction set architecture are shown.

Additionally, you can read the memory address `0x08000038` using the following command:

Terminal-3: GDB debugger

```
(gdb) x 0x08000038
```

As can be seen in Figure 10, the data value shown on Terminal-3 should be the maximum of the list of numbers indicated in the `.data` section of the source code (see Figure 7).

```
(gdb) x 0x08000038
0x8000038:      0x00000008
(gdb)
```

Figure 10: Output message after executing `x` command. The maximum value of a list of numbers is saved at the `0x08000038` memory address.

Now, run from the beginning and stop the program at the last branch instruction which is loaded into the memory location `0x8000034`. To do this, a *breakpoint* is inserted as indicated in the following box. Finally, the value of the PC register is read.

Terminal-3: GDB debugger

```
(gdb) b *0x8000034
(gdb) set $pc = 0x8000000
(gdb) continue
(gdb) info registers
(gdb) x 0x08000038
```

Question 1.
Examine the disassembled code of the `lab1_part2.elf` file (see `lab1_part2.elf.objdump` file). Note the difference in comparison with the original source code. Make sure that you understand the meaning of each instruction. Observe also that your program was loaded into memory locations with the starting address `0x08000000`. These addresses correspond to the on-chip SRAM memory, which was selected when specifying the system parameters.
Note that the pseudoinstruction `la x15, RESULT` in the original source code has been replaced with two machine instructions, `auipc a5,0x0` and `addi a5,a5,56`, which load the 32-bit address `RESULT` into register `a5` in two parts. `auipc a5,0x0` initializes the `a5` register to the current value of the PC register: `0x08000000`. `addi a5,a5,56` adds 56 =`0x38` to the `a5` register. The register `x5` is named `a5`.

- Examine the disassembled code to see the difference in comparison with the original source program. Make sure that you understand the meaning of each instruction.

This time add a breakpoint at address `0x8000024`, so that the program will automatically stop executing whenever the `bge` branch instruction at this location is about to be executed. Run the program and observe the contents of registers `s2` and `s3` each time this breakpoint is reached.

Terminal-3: GDB debugger

```
(gdb) b *0x8000024
(gdb) continue
(gdb) info registers
```

Return to the beginning of the program by setting the Program Counter to 0. Now, single step through the program. Watch how the instructions change the data in the processor's registers.

Terminal-3: GDB debugger

```
(gdb) b *0x8000000
(gdb) j *0x8000000
(gdb) stepi
(gdb) info registers
```

Remove the breakpoint. Then, set the Program Counter to `0x8000008`, which will bypass the first two instructions which load the address `RESULT` into register `a5`. Also, set the value in register `a5` to `0x8000004`. Run the program.

Terminal-3: GDB debugger

```
(gdb) j *0x8000008
(gdb) continue
(gdb) info registers
```

Question 2.

- What will be the result of this execution?

Part III

Instructions and data are represented as patterns of `1`s and `0`s. In this part, we will examine how instructions are encoded. We will do this by replacing the instruction `bge x18, x19, LOOP` in the program in Figure 7 with the instruction `blt x18, x19, LOOP`. However, instead of replacing this instruction in the source code and then recompiling and loading the modified program into main memory, we will load the original program and then make the desired change directly in the program that is already loaded in the memory. To do this it is necessary to derive the machine-code representation of the `blt` instruction.

Perform the following steps:

1. Derive the machine code representation of the instruction `blt x18, x19, LOOP`. In the *Nios V Processor Reference Handbook* ([3]), we can find that the `blt` instruction has the format shown in Figure 11. In this case, use registers `x18` and `x19` as registers `A` and `B`, respectively, and determine the instruction code needed to branch to the instruction at location `LOOP`.

RISC-V Instruction Set

Core Instruction Formats

31 27	26 25	24 20	19 15	14 12	11 7	6 0	
funct7		rs2	rs1	funct3	rd	opcode	R-type
imm[11:0]			rs1	funct3	rd	opcode	I-type
imm[11:5]		rs2	rs1	funct3	imm[4:0]	opcode	S-type
imm[12|10:5]		rs2	rs1	funct3	imm[4:1|11]	opcode	B-type
imm[31:12]					rd	opcode	U-type
imm[20|10:1|11|19:12]					rd	opcode	J-type

RV32I Base Integer Instructions

Inst	Name	FMT	Opcode	funct3	funct7	Description (C)	Note
`blt`	Branch <	B	1100011	0x4		`if(rs1 < rs2) PC += imm`	

0x 8000024 : ff3948e3

1111 1111 0011 1001 0100 1000 1110 0011

1	111111	10011	10010	100	1000	1	1100011
imm[12]	imm[10:5]	rs2	rs1	funct3	imm[4:1]	imm[11]	opcode

Figure 11: Format for the `blt` instruction.

2. Reload your original program. Then, execute the program once, stopping at the end.
3. Place the new `blt` instruction code at the memory address where the `bge` instruction was loaded. Additionally, verify that the new instruction is placed in the memory address where the `bge` instruction was loaded, `0x8000024`.

   Terminal-3: GDB debugger

   ```
   (gdb) set int0x8000024 = 0xff3948e3
   (gdb) x 0x08000024
   ```

4. Set the Program Counter to `0x8000000`, the beginning of the program, and run the program using the GDB command `continue`.

Question 3.

- What is the result provided by the execution of program?
- What are the values saved in register `x18` and memory location `0x8000038`?

Part IV

In this part, you are required to write a RISC-V assembly language program that generates the first n

numbers of the *Fibonacci* series. In this series, the first two numbers are 0 and 1, and each subsequent number is generated by adding the preceding two numbers. For example, for $n = 8$, the series is

$$0, 1, 1, 2, 3, 5, 8, 13$$

Your program should store the numbers in successive memory word locations starting at `0x1000`. Place a test value n in location `0xffc`.

Perform the following steps:

1. Create a new directory: `lab1_part4`.
2. Write the source code of an assembly language program that computes the desired Fibonacci series, and place the file in the directory `lab1_part4`.
3. Compile, link and run your program using the DE0-Nano board.
4. Examine the memory locations starting at `0x1000` to verify that your program is correct.

References

[1] Terasic. DE0-Nano User Manual, 2013.

[2] Intel. Nios V Embedded Processor Design Handbook, 2023.

[3] Intel. Nios V Processor Software Developer Handbook, 2023.