

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 processor and its RISC-V assembly language. It uses a simple computer system, called the DE0-Nano Basic Computer, which includes the Nios V/m processor. The system is implemented as a circuit that is downloaded into the Field Programmable Gate Array (FPGA) device on the DE0-Nano board. 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. This lab 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 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*.

Perform the following:

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

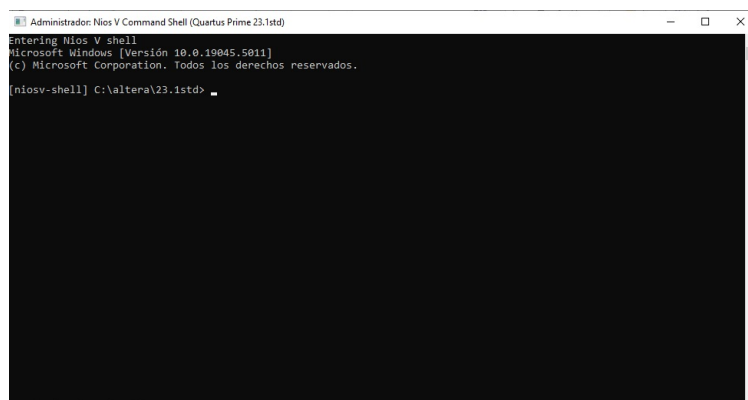


Figure 1: The Nios V Command Shell Window.

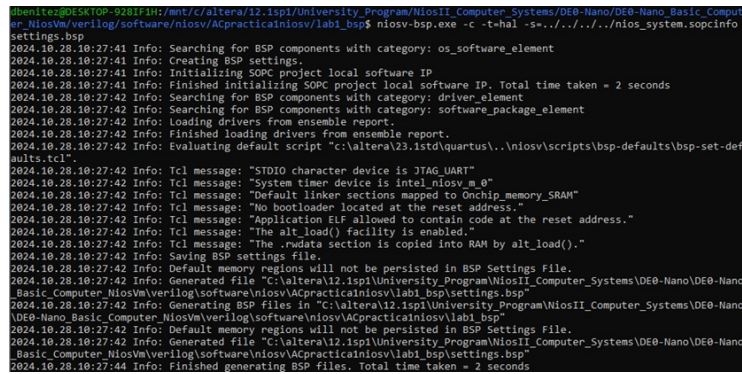
To run an application program it is necessary to create a new BSP project.

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. It 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
$ bash
$ 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.



```
ipenitez@DESKTOP-9281F1H:/mnt/c/altera/12.1sp1/University_Program/NiosII_Computer_Systems/DE0-Nano/DE0-Nano_Basic_Computer_Systems/verilog/software/niosv/ACpracticalniosv/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: 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-defaults.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_NiosV\verilog\software\niosv\ACpracticalniosv\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_NiosV\verilog\software\niosv\ACpracticalniosv\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_NiosV\verilog\software\niosv\ACpracticalniosv\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.

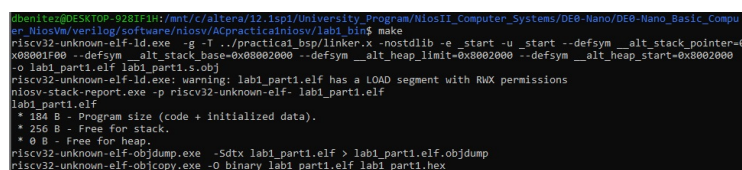
```
$ cd lab1_bin
$ riscv32-unknown-elf-as.exe lab1_part1.s -o lab1_part1.s.obj
```

The output is a file called: `lab1_part1.s.obj`.

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

```
$ 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
```

The output is a file called: `lab1_part1.elf`. Figure 3 shows the messages displayed in the Nios V Command Shell window.



```
ipenitez@DESKTOP-9281F1H:/mnt/c/altera/12.1sp1/University_Program/NiosII_Computer_Systems/DE0-Nano/DE0-Nano_Basic_Computer_Systems/verilog/software/niosv/ACpracticalniosv/lab1_bin$ make
riscv32-unknown-elf-ld.exe -g -T ../practical_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
riscv32-unknown-elf-ld.exe: warning: lab1_part1.elf has a LOAD segment with R/X 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.

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

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

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

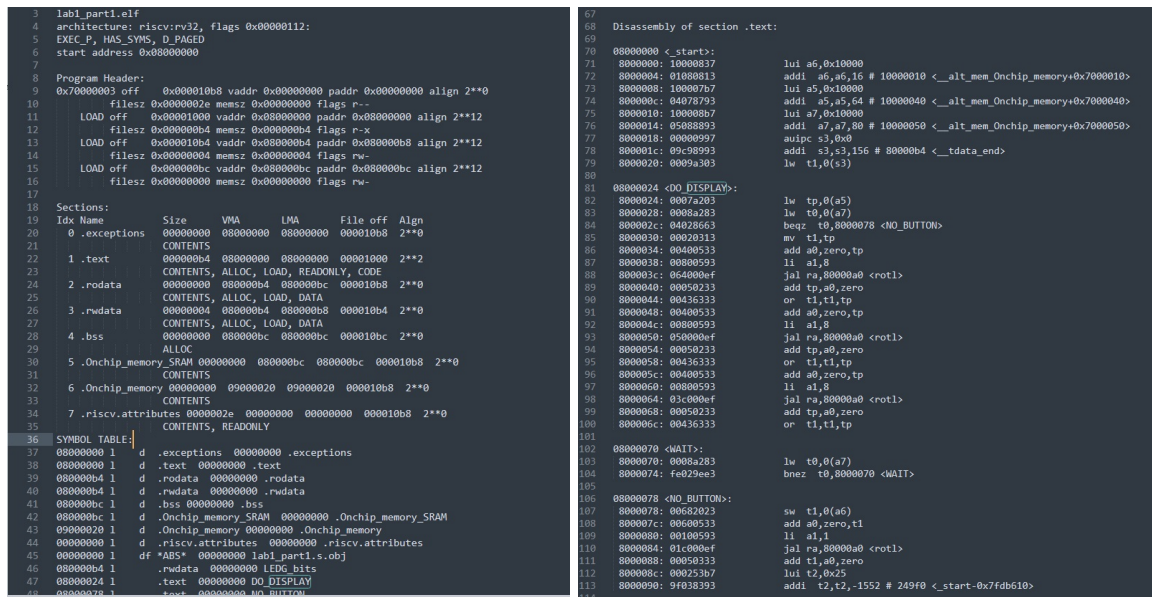


Figure 4: Content of the file `lab1_part1.elf.objdump`.

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.

```
$ 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.

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

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

```
$ quartus_pgm.exe -c 1 -m JTAG -o "p;DE0-Nano-Basic-Computer_22jul24.sof@1"
```

Watch the change in state of the blue LEDs on the DE0-Nano board that correspond to LOAD and GOOD, which will blink as the circuit is being downloaded. Figure 5 shows the messages displayed on the screen.

14. Having downloaded the sof configuration 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.

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

This command also run the program.

```

benitez@DE05KTOP-9281F1H:/mnt/c/altera/12.1sp1/University_Program/NiosII_Computer_Systems/DE0-Nano/DE0-Nano_Basic_Computer_Systems/verilog/software/niosv/ACpracticainiosv/lab1_bin$ quartus_pgm.exe -c 1 -m JTAG -o "p;../../../../DE0-Nano_Basic_Computer.sof@1"
Info: Running Quartus Prime Programmer
Info: Version 23.1std.0 Build 991 11/28/2023 SC Standard Edition
Info: Copyright (C) 2023 Intel Corporation. All rights reserved.
Info: Your use of Intel Corporation's design tools, logic functions
Info: and other software and tools, and any partner logic
Info: functions, and any output files from any of the foregoing
Info: (including device programming or simulation files), and any
Info: associated documentation or information are expressly subject
Info: to the terms and conditions of the Intel Program License
Info: Subscription Agreement, the Intel Quartus Prime License Agreement,
Info: the Intel FPGA IP License Agreement, or other applicable license
Info: agreement, including, without limitation, that your use is for
Info: the sole purpose of programming logic devices manufactured by
Info: Intel and sold by Intel or its authorized distributors. Please
Info: refer to the applicable agreement for further details, at
Info: https://fpgasoftware.intel.com/eula.
Info: Processing started: Mon Oct 28 11:06:19 2024
Info: Command: quartus_pgm -c 1 -m JTAG -o p;../../../../DE0-Nano_Basic_Computer.sof@1
Info (213045): Using programming cable "USB-Blaster [USB-0]"
Info (213011): Using programming file ../../../../../../DE0-Nano_Basic_Computer.sof with checksum 0x005DE1F1 for device EP4CE22K10F101
Info (209060): Started Programmer operation at Mon Oct 28 11:06:20 2024
Info (209010): Configuring device index 1
Info (209017): Device 1 contains JTAG ID code 0x020F30D0
Info (209007): Configuration succeeded -- 1 device(s) configured
Info (209013): Successfully performed operation(s)
Info (209001): Ended Programmer operation at Mon Oct 28 11:06:22 2024
Info: Quartus Prime Programmer was successful. 0 errors, 0 warnings
Info: Peak virtual memory: 4446 megabytes
Info: Processing ended: Mon Oct 28 11:06:22 2024
Info: Elapsed time: 00:00:03
Info: Total CPU time (on all processors): 00:00:00

```

Figure 5: Messages displayed on the screen after configuring the DE0-Nano board.

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".



Figure 6: LEDs on the board are turning on and off.

Part II. Designing 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`.

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!

```
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, x16, 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.

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` file and download the `.elf` program using `niosv-download`.
3. Then, configure the DE0-Nano board using the `quartus_pgm.exe` command and `DE0_Nano_Basic_Computer_22jul24.s` 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

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

OpenOCD complies with the remote gdbserver protocol and, as such, can be used to debug remote targets. GDB works with OpenOCD. Terminal-2 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-2: GDB debugger

```
$ cd lab1_part2
$ sh
$ riscv32-unknown-elf-gdb
(gdb) target extended-remote localhost:3333
(gdb) file lab1_part2.elf
(gdb) load
(gdb) info registers
```

The third terminal is used as above in Part I for compiling, linking, downloading and executing programs.

Terminal-3: Program execution

```
$ cd lab1_part2
$ sh
compiling
linking
configuring FPGA
downloading program
```

In Terminal-2, read the memory address `0x08000038` using the following command:

Terminal-2: GDB debugger

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

The data value shown on Terminal-2 should be the maximum of the list of numbers indicated in the `.data` section of the source code (see Figure 7).

Now, stop the program at the last Branch instruction which is loaded in 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-2: GDB debugger

```
(gdb) b *0x8000034
(gdb) continue
(gdb) info registers
```

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-2: 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-2: 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-2: GDB debugger

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

Question 2.

What will be the result of this execution?