Nios II Simulator

Learning Goal: Assembly language basics.

Requirements: Nios2Sim Simulator.

1 Introduction

During this lab, you will analyze and write simple programs in assembly language. For this, you will use a small simulator, which displays the memory and the registers content at each step of the execution.

In the following table, the conventional function and names[1] of the 32 registers of the **Register File** are listed.

Register	Name	Function
`r0`	`zero`	0x00000000
`r1`	`at`	Assembler Temporary
`r2`	`v0`	Return Value
`r3`	`v1`	Return Value
`r4`	`a0`	Register Arguments
`r5`	`a1`	Register Arguments
`r6`	`a2`	Register Arguments
`r7`	`a3`	Register Arguments
`r8`	`t0`	Temporary Register
`r9`	`t1`	Temporary Register
`r10`	`t2`	Temporary Register
`r11`	`t3`	Temporary Register
`r12`	`t4`	Temporary Register
`r13`	`t5`	Temporary Register
`r14`	`t6`	Temporary Register
`r15`	`t7`	Temporary Register

Register	Name	Function
`r16`	`s0`	Saved Register
`r17`	`s1`	Saved Register
`r18`	`s2`	Saved Register
`r19`	`s3`	Saved Register
`r20`	`s4`	Saved Register
`r21`	`s5`	Saved Register
`r22`	`s6`	Saved Register
`r23`	`s7`	Saved Register
`r24`	`et`	Exception Temporary
`r25`	`bt`	Breakpoint Temporary
`r26`	`gp`	Global Pointer
`r27`	`sp`	Stack Pointer
`r28`	`fp`	Frame Pointer
`r29`	`ea`	Exception Return Address
`r30`	`ba`	Breakpoint Return Address
`r31`	`ra`	Return Address

2 The Nios 2 Simulator

In this section, we will see how to simulate step by step an assembly program with the **Nios2Sim** simulator.

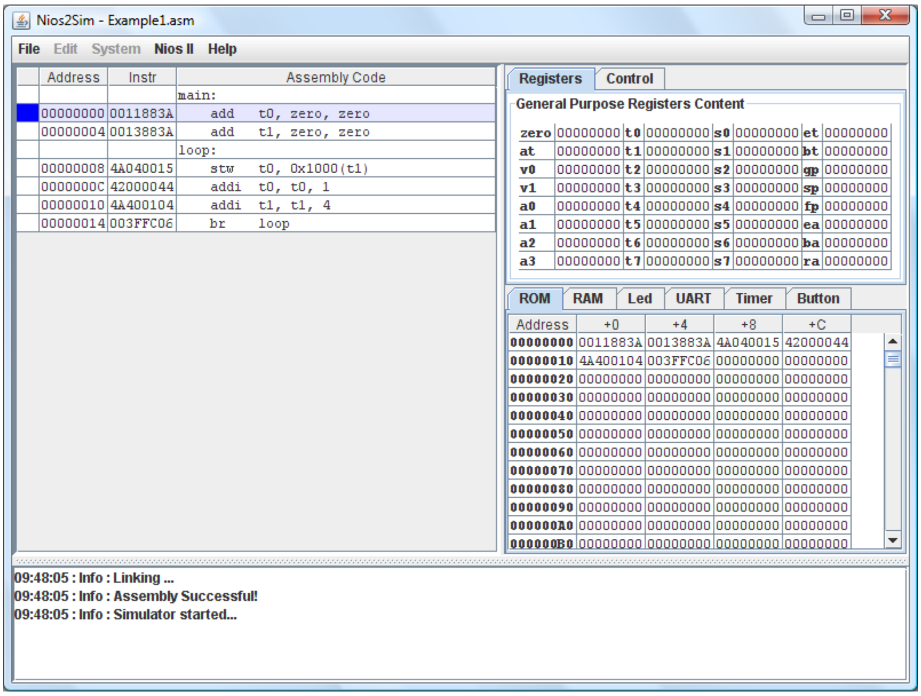
- Download the **Nios2Sim** simulator from the web page of the course.
- Execute the **Nios2Sim** simulator by double clicking on the `.jar` file, note that you need a Java RE for that.

[1] To improve the readability of the code, we extend the official Nios II registers naming: we added names to the registers `r2` to `r23`, which were unnamed. These names are only supported by the **Nios2Sim** simulator.

- Copy the following code example to the **Simulator**.

```
main:
    add     t0, zero, zero
    add     t1, zero, zero
loop:
    stw     t0, 0x1000(t1)
    addi    t0, t0, 1
    addi    t1, t1, 4
    br      loop
```

- Start the assembly by selecting Nios II > Assemble (Ctrl+1). If the assembly succeeded, you should see the message "**Assembly Successful!**".
- Select Nios II > Start Simulation (Ctrl+2) to start the simulation . This will load the simulation layout:



  - The table on the left lists the assembly code with, for each line, its corresponding address and **Nios II** instruction word. The blue line is the next instruction that the simulator will execute (i.e., the current PC address).
  - The upper-right table lists the registers of the **Register File** status. To edit a register value, double-click on it and enter a number. The values are represented in hexadecimal. Letting your mouse over a value will display a pop up box with the binary and decimal representations.
  - The table on the bottom-right displays the memory content. The memory is organized in the same way that for the labs **Memories** and **Multicycle Nios II**. You will find the LEDs module and the other peripherals. Each memory module and peripheral has its own tab. Similarly to the registers, you can modify a value by double-clicking on it, and get the different representations of this value by letting your mouse over it.

- You can execute the next step of the program by selecting Nios II > Execute a Step (Ctrl+E). Every change in the registers or in the memory during this step will be highlighted in red.

  - Execute some steps and observe the evolution of the memory and registers content.

- By selecting Nios II > Run (Ctrl+R), the simulator will execute the program until it reaches a breakpoint or the maximal number of execution steps (500). Every change that occurs in the registers or in the memory during this interval will be highlighted in red.
  - Place a breakpoint inside the loop (at address `0x0008` for example). To do this, double-click on the desired line while in the simulation view.
  - Run the execution several times and observe the evolution of the memory and registers content.

- To terminate the simulation and return to the edition of the assembly code, select Nios II > End Simulation.

3 Analyzing a Program in Assembly Language

3.1 Example 1

- Copy the following program to the **Nios2Sim** simulator and save it (File > Save As...).

```
main:
    addi    a0, zero, data   ; a0: Data address
    addi    a1, zero, 4      ; a1: Number of elements
    call    proc
    break                    ; end of the program

proc:
    add     v0, zero, zero   ; v0 = 0
    add     t0, a0, zero     ; t0 = a0
    add     t1, zero, zero   ; t1 = 0
outer:
    cmpltu  t3, t1, a1       ; t3 = (t1 < a1)
    beq     t3, zero, return ; if (!t3) goto return
    ldw     t4, 0(t0)        ; t4 = mem[t0]
    addi    t5, zero, 32     ; t5 = 32
inner:
    beq     t5, zero, next   ; if (!t5) goto next
    andi    t2, t4, 1        ; t2 = t4 & 1
    add     v0, v0, t2       ; v0 = v0 + t2
    srli    t4, t4, 1        ; t4 = t4 >> 1
    addi    t5, t5, -1       ; t5 = t5 - 1
    br      inner            ; goto inner
next:
    addi    t1, t1, 1        ; t1 = t1 + 1
    addi    t0, t0, 4        ; t0 = t0 + 4
    br      outer            ; goto outer
return:
    ret                      ; return to caller

data:                        ; data initialization
    .word 1
    .word 3
    .word 0xAAAAAAAA
    .word 0xFFFFFFFF
```

The `main` procedure starts at address 0. The **PC** being initialized to 0, this is the first procedure that the **CPU** will execute after a reset.

`main` procedure description:

- The registers `a0` and `a1` are initialized.
- The argument `a0` is initialized to the address of the `data` section.
- The argument `a1` is initialized to the length of the `data` section.
- The procedure `proc` is called.
- The program execution is terminated by the **`break`** instruction.

The **`.word`** statement is used to set manually the value of a word in the instruction memory. In this program, it's used to initialize some data.

3.1.1 Exercise

- Describe the function of this program.
  - Modify the entries of the `data` section to test different situations.
- Is it necessary to use two instructions for the test of the `outer` loop (**`cmpltu`** and **`beq`**)? Can we replace them by a single instruction? If it's possible, simplify the program.
  - Simulate to verify that it still behaves correctly.
- The program does not cover cases with arithmetic overflows. Show the instruction(s) where an overflow can potentially occur. Ignore the instructions dealing with addresses or indices.
- Correct the program to take into account arithmetic overflows. The program will return -1 in case of an overflow (two's complement representation). Note that now, the return value is signed.
  - Simulate and modify the **Register File** content to create an overflow. Verify the behavior.
- The `inner` loop is executed a fixed number of times. Can this be improved? If it is possible, simplify the program.
  - Simulate to verify that it is still correct.

3.2 Example 2

- Copy the following program to the **Nios2Sim** simulator and save it (File > Save As...).

```
main:
    addi    a0, zero, data1 ; a0: Data1 address
    addi    a1, zero, data2 ; a1: Data2 address
    call    proc
    break                   ; end of the program

proc:
    add     t0, a0, zero
    add     t1, a1, zero
    add     v0, zero, zero
    addi    v1, zero, 1
outer:
    ldw     t2, 0(t0)
```


Version 1.8 of 16th August 2012, EPFL ©2012

```
    ldw     t3, 0(t1)
inner:
    andi    t4, t2, 0xff
    andi    t5, t3, 0xff
    bne     t4, t5, end2
    beq     t4, zero, end
    addi    v0, v0, 1
    srli    t2, t2, 8
    srli    t3, t3, 8
    andi    t4, v0, 3
    bne     t4, zero, inner
    addi    t0, t0, 4
    addi    t1, t1, 4
    br      outer
end2:
    add v1, zero, zero
end:
    ret

data1:
    .word 0xAABBCCDD
    .word 0x66778899
    .word 0x00125678
    .word 0x00000000
data2:
    .word 0xAABBCCDD
    .word 0x66778899
    .word 0x00345678
    .word 0x00000000
```

3.2.1 Exercise

- Describe the function of this program.
  - Modify the entries of `data1` and `data2` to test different situations.
- Can we determine if the program is working on *signed* or *unsigned* data? Identify the instruction(s) working with the data to answer to the question.
- Modify the program to work on signed bytes, and to compare the signs only. Now, two bytes are considered as equal if their signs are the same.
  - Modify the entries of `data1` and `data2` to test different situations.

4 Writing a Program in Assembly Language

4.1 Counting positive numbers

Write a procedure labeled `count` that counts the number of strictly positive elements of a 32-bit array. Save your file with the name `count.asm`. The parameter `a0` contains the location of the array and `a1` its length. The number of strictly positive elements is returned in `v0`. You can use the same structure as the one used in the program of the subsection 3.1.

4.2 Modifying data format

Write a procedure labeled `modifyFormat` that modifies each elements of an array of unsigned values. Save your file with the name `modifyFormat.asm`. The parameter `a0` contains the location of the array and `a1` its length. The array content must be modified according to the following figure:

	31 20 10 0
original	???????????abcdefghijk??????????
modified	0000000000000000011111abcdefghijk

- The 16 most significant bits are set to 0.
- The bits 11 to 15 are set to 1.
- The 11 least significant bits are copied from the bits 10 to 20 of the original word.

4.3 String concatenation

For this exercise, implement a concatenation operation for null-terminated, ASCII encoded strings, which appends two input strings to produce a third output string. A null-terminated string is represented as sequence of 8-bit non-zero ASCII characters terminated by a zero (null character). For example, the strings "FP" and "GA" are respectively encoded as `0x465000` and `0x474100` and produces a third string "FPGA", encoded as `0x4650474100`, when concatenated. Since each character is encoded using 8 bits, we can store up to four characters in a 32-bits word.

Write a procedure labelled `concatenate` which takes three inputs: the starting addresses of two null-terminated input strings in `a1` and `a2`, and the starting address of the output string in `a0`. The procedure should write the output of the concatenation to the memory starting at address `a0`.

The following table shows the relevant part of memory after the `concatenate` procedure is called with `a1` = `0x1002`, `a2` = `0x1011` and `a0` = `0x1023`.

Address	+0	+4	+8	+C
0x1000	0x????FFE5	0x3A1200??	0x????????	0x????????
0x1010	0x??A10500	0x????????	0x????????	0x????????
0x1020	0x??????FF	0xE53A12A1	0x0500????	0x????????

Pay attention that all values that are not part of the output string should be preserved. In the example above, all the values indicated as '?' or the input strings should not be modified.

Hint: *It might be useful to implement two functions to load and store one byte from a given address and implement the string concatenation operation using these functions.*

5 Submission

Submit the three files `count.asm`, `modifyFormat.asm` and `concatenate.asm` mentioned in Section 4.1, 4.2 and 4.3, respectively.

 Version 1.8 of 16th August 2012, EPFL ©2012