**Natalius II 8-bit RISC Processor in Verilog**

Adapted from Natalius by Fabio Andrés Guzmán Figueroa (fabioandres.guzman@opencores.org)  
  
**1. Description**Natalius II is a compact, capable and fully embedded 8-bit RISC processor core described 100% in Verilog. It occupies about 370 LEs, 130 registers, and 33,000bits internal memory in a Altera Cyclone II EP2C8Q208C8N (around 5% LEs, 20% memory).

Natalius II offers a python assembler. The instruction memory is implemented using internal FPGA memeory it stores 2048 instructions; each instruction has a width of 16 bits (2048x16). Each instruction takes 3 clock cycles to be executed.

**Features:**

* 8-Bit ALU
* 8x8 register file (r0…r7)
* 2048x16 instruction (ROM) memory
* 32x8 RAM memory
* 16x11 stack memory for PC
* 3 CLK cycles per instruction
* carry ( C ) and zero ( Z) flags
* 8-bit Address Port (224 I/O peripheral addresses, 32 RAM addresses, or 256 I/O if no RAM)
* Can retrieve data bytes from highest 256 instruction memory addresses

**2. Instruction Set**Natalius II contains the typical instruction set on a processor. These are: memory access, arithmetic, logical and flow control, as described in Table 1.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Instr.** | **Description** | **Type** | **Instruction** | **Description** | **Type** |
| **ldi** | load immediate | memory access | **ret** | return subrutine | flow control |
| **ldm** | load from memory | memory access | **adi** | add with imm | arithmetic |
| **stm** | store to memory | memory access | **csz** | csr if zero | flow control |
| **cmp** | compare | arithmetic | **cnz** | csr if no zero | flow control |
| **add** | addition | arithmetic | **csc** | csr if carry | flow control |
| **sub** | subtraction | arithmetic | **cnc** | csr if no carry | flow control |
| **and** | logic and | logical | **sl0** | shift left zero fill | logical |
| **oor** | logic or | logical | **sl1** | shift left one fill | logical |
| **xor** | logic xor | logical | **sr0** | shift right zero fill | logical |
| **jmp** | jump | flow control | **sr1** | shiftright one fill | logical |
| **jpz** | jump if zero | flow control | **rrl** | rotary register left | logical |
| **jnz** | jump if no zero | flow control | **rrr** | rotay register right | logical |
| **jpc** | jump if carry | flow control | **not** | logic not | logical |
| **jnc** | jump if no carry | flow control | **nop** | no operation | nop |
| **csr** | call subrutine | flow control | **ldr** | load from ROM | memory access |
| **reti** | return from interrupt | flow control |  |  |  |

*Table 1. Natalius Instruction Set*

**3. Natalius Interface Signals**

The top-level interface signals to the Natalius Processor appear in Figure 1. and are described in  
Table 2.

A picture containing diagram

Description automatically generated  
 *Figure 1. Natalius interface connections*

|  |  |  |
| --- | --- | --- |
| **Signal** | **Direction** | **Description** |
| **CLK** | input | **clock input:** All Natalius II registers are clocked from the rising clock edge |
| **RTS** | input | **reset input:** To reset the Natalius II processor, this rst is asynchronous input and it sets PC=0x0000 |
| **data\_in[7:0]** | input | **Input data port:** The data is captured on the rising edge of CLK (used in ldm instruction) |
| **data\_out[7:0]** | output | **Output data port:** Output data appears on this port for three CLK cycles during a stm instruction, capture this data when write\_e is high (uded in stm instruction) |
| **port\_addr[7:0]** | output | **Port address:** This addresses the peripheral port to the input or output by instruction ldm or stm |
| **write\_e** | output | A write is being performed to the device on port\_addr[7:0]. Use for I/O decoding |
| **read\_e** | output | A read is being performed to the device on port\_addr[7:0]. Use for I/O decoding |

*Table 2. Interface signals descriptions*

**4. Organization of source code**

The hierarchy of the files are shown in Figure 2.

Text

Description automatically generated with medium confidence

*Figure 2. Source files hierarchy*

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Opcode** | **Instr** | **Description** | **Z/C flag  affected?** | **Use** |
| 2 | ldi | load immediate |  | ldi rd,imm (rd=imm) |
| 3 | ldm | load from memory |  | ldm rd,port\_addr (rd=data\_in <= mem[port\_addr]) |
| 4 | stm | store to memory |  | stm rd,port\_addr (rd=data\_out => mem[port\_addr]) |
| 5 | cmp | compare | ü | cmp rd,rs |
| 6 | add | addition | ü | add rd,rs (rd=rd+rs) |
| 7 | sub | subtraction | ü | sub rd,rs (rd=rd-rs) |
| 8 | and | logic and | ü | and rd,rs (rd=rd and rs) |
| 9 | oor | logic or | ü | oor rd,rs (rd=rd or rs) |
| 10 | xor | logic xor | ü | xor rd,rs (rd=rd xor rs) |
| 11 | jmp | jump |  | jmp inst\_addr (pc=inst\_addr) |
| 12 | jpz | jump if zero |  | jpz inst\_addr (pc=inst\_addr if zero) |
| 13 | jnz | jump if no zero |  | jnz inst\_addr (pc=inst\_addr if no zero) |
| 14 | jpc | jump if carry |  | jpc inst\_addr (pc=inst\_addr if carry) |
| 15 | jnc | jump if no carry |  | jnc inst\_addr (pc=inst\_addr if no carry) |
| 16 | csr | call subrutine |  | csr inst\_addr (pc=inst\_addr) save pc+1 in stack |
| 17 | ret | return subrutine |  | ret (pc=value stored in stack) |
| 18 | adi | add with imm | ü | add rd,imm (rd=rd+imm) |
| 19 | csz | csr if zero |  | csr inst\_addr (pc=inst\_addr if zero) affects stack |
| 20 | cnz | csr if no zero |  | csr inst\_addr (pc=inst\_addr if no zero) affects stack |
| 21 | csc | csr if carry |  | csr inst\_addr (pc=inst\_addr if carry) affects stack |
| 22 | cnc | csr if no carry |  | csr inst\_addr (pc=inst\_addr if no carry) affects stack |
| 23 | sl0 | shift left zero fill |  | sl0 rd (rd <= {rd[6:0],0}) |
| 24 | sl1 | shift left one fill |  | sl1 rd (rd <= {rd[6:0],1}) |
| 25 | sr0 | shift right zero fill |  | sr0 rd (rd <= {0,rd[7:1]}) |
| 26 | sr1 | shiftright one fill |  | sr1 rd (rd <= {1,rd[7:1]) |
| 27 | rrl | rotary register left |  | rrl rd (rd <= {rd[6:0],rd[7]}) |
| 28 | rrr | rotay register right |  | rrr rd (rd <= {rd[0],rd[7:1]) |
| 29 | not | logic not |  | sub rd,rs (rd=rd-rs) |
| 30 | nop | no operation |  | no operation (take 3 clk ) |
| 31 | ldr | load from ROM |  | ldr rd,ROM\_addr (rd=ROM[addr][7:0])  addr=0x00-0xFF. Bits [15:11] of the ROM location cannot be accessed. |

*Table 3. Detailed Instruction Set*

**5. Assembler script**The assembler script was written in Python, to be run as follows:

./assembler.py [-s] code.asm (Linux)  
python assembler.py [-s] code.asm (Windows cmd terminal)

You can view the assembler output by adding the optional flag -s, and "code.asm" is the file that will be assembled. The output file is named: " instructions.mem". You can pass the pass to this file as a parameter to the top level ‘natalius\_processor’ module, for example:  
  
parameter ROM\_FILE = "./assembler/instructions.mem";  
natalius\_processor #(.PATH\_TO\_PROG\_CODE(ROM\_FILE)) nat2, clk, rst, p\_addr, r\_e, w\_e, d\_in,d\_out);

**Example:**Consider the following code:

**; these are explanatory comments**

**; to help with understanding the code operation**

**.org 10 ; code starts at address 10 (0x000A)**

**csr negro ; show black screen**

**csr loop**

**end jmp end**

**negro stm r0,32**

**ldi r1,80**

**ldi r2,60**

**ldi r3,0**

**ldi r4,0**

**nextc cmp r4,r1**

**jpz inc\_fil**

**stm r3,64**

**stm r4,32**

**adi r4,1**

**jmp nextc**

**inc\_fil ldi r4,0**

**cmp r3,r2**

**jpz fneg**

**adi r3,1**

**jmp nextc**

**fneg ret**

**.org 0x100 ; another origin example**

**loop ret**

The above code is saved in the file "example.asm". When running the line:  
./assembler.py -s example.asm  
the console displays the following:  
  
line no. addr inst asm

-----------------------------------------------------

0 @10 .org 10

1 8004 csr 4

2 8016 csr 22

3 5803 jmp 3

4 2020 stm r0,32

5 1150 ldi r1,80

6 123C ldi r2,60

7 1300 ldi r3,0

8 1400 ldi r4,0

9 2C20 cmp r4,r1

10 600F jpz 15

11 2340 stm r3,64

12 2420 stm r4,32

13 9401 adi r4,1

14 5809 jmp 9

15 1400 ldi r4,0

16 2B40 cmp r3,r2

17 6014 jpz 20

18 9301 adi r3,1

19 5809 jmp 9

20 8800 ret

21 @256 .org 0x100

22 8800 ret

**6. Reference Design**The reference design is the implementation of the classic video game (pong) using the processor  
Natalius II. The architecture of this design is shown in Figure 3.

Diagram, schematic

Description automatically generated

*Figure 3. Hardware architecture of reference design (pong video game)*

*END OF DOCUMENT*