Universidad Tecnológica Nacional

Técnicas Digitales IV

Trabajo Práctico 3: CPU softcore en FPGA

Autores:
Matías Iván Bucca
Ignacio Rodríguez Grandi
Jonás Gabriel Ramírez Torres
Federico Nicolas Elizondo

Profesore/s:
Pablo Cayuela
Sergio Olmedo

June 22, 2021

Contents

1 Introduccion

En este practico se llevara a cabo la implementacion de un microcontrolador Nanoblaze (descripcion y compilador otorgado por la catedra) en la FPGA.

El objetivo es cargar en la *ROM* del CPU una secuencia de código en formato **.vhd**, a partir de un algoritmo realizado en *Assembly* y poder comprobar el correcto funcionamiento del microcontrolador, utilizando los recursos de la FPGA.

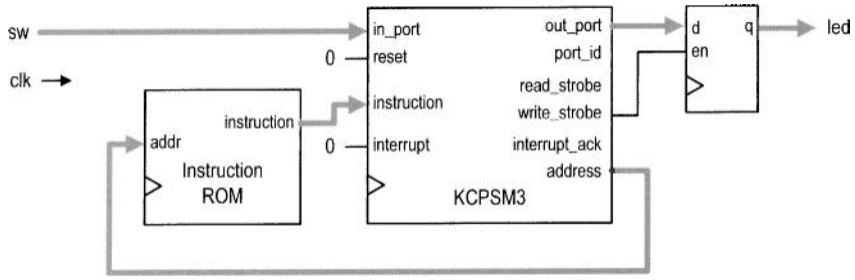

Figure 1: Nanoblaze con una simple interfaz I/O.

El Nanoblaze es un microcontrolador muy simple orientado inicialmente para FPGA's Spartan 3, y se encuentra optimizado para estas ultimas. Este se puede conseguir como softcore, provisto por la comunidad.

Su principal característica es su reducido consumo de recursos y que carece de una interfaz de entradas y salidas, pudiendo ser descriptas por el usuario según su necesidad.

2 Implementacion en Quartus

Luego de generar el archivo **.vhd** para el bloque *ROM*, podemos sintetizar todo el sistema en la FPGA. Este microcontrolador no tiene periféricos de entrada y salida incluidos, esta es creada a la necesidad del usuario. Dado que el objetivo en este caso es comprobar el correcto funcionamiento del Nanoblaze, solo usaremos una interfaz simple.

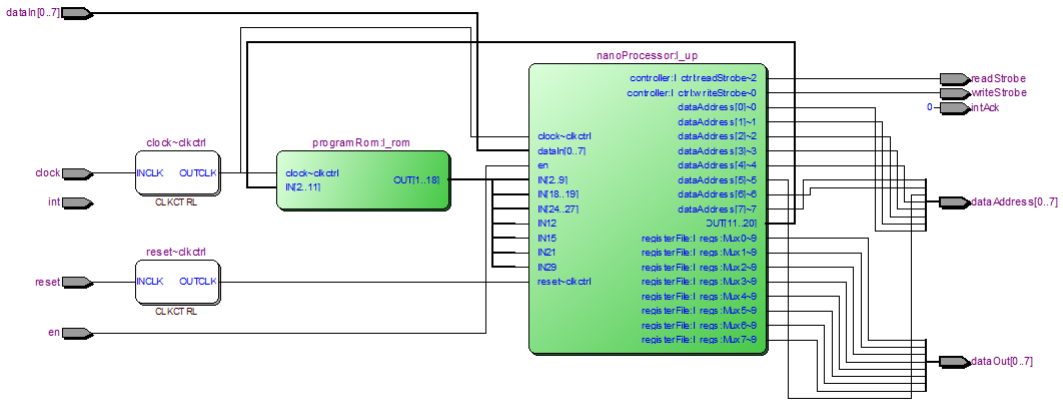

Figure 2: Nanoblaze implementación en Quartus

El diseño final quedaria integrado por este bloque y el adicional de un registro de salida, quedando sintetizado de la siguiente manera:

```
library ieee;
use ieee.std_logic_1164.all ;
use ieee.numeric_std.all ;
entity out_regs is
GENERIC(
         addressBitNb           : positive := 8;
         registerBitNb          : positive := 8;
         programCounterBitNb    : positive := 10;
         stackPointerBitNb      : positive := 5;
         registerAddressBitNb   : positive := 4;
         scratchpadAddressBitNb : natural  := 6
  );
port(
        clk, reset: in std_logic;
        sw: in std_ulogic_vector (7 downto 0) ;
        led: out std_ulogic_vector (7 downto 0)
);
end out_regs;
architecture arch of out_regs is

signal salida : std_ulogic_vector (7 downto 0) ;
signal entrada : std_ulogic_vector (7 downto 0) ;
signal led_reg : std_ulogic_vector (7 downto 0) ;
signal sWriteStorbe : std_logic;

COMPONENT nanoblaze
  PORT(
    reset       : IN     std_ulogic;
    clock       : IN     std_ulogic;
    en          : IN     std_ulogic;
```

```
        dataAddress : OUT    unsigned(addressBitNb-1 DOWNTO 0);
        dataOut     : OUT    std_ulogic_vector(registerBitNb-1 DOWNTO 0);
        dataIn      : IN     std_ulogic_vector(registerBitNb-1 DOWNTO 0);
        readStrobe  : OUT    std_uLogic;
        writeStrobe : OUT    std_uLogic;
        int         : IN     std_uLogic;
        intAck      : OUT    std_ulogic
      );
    END COMPONENT;




    begin

    inst_nanoblaze: nanoblaze
    port map(
        reset  => reset,
        clock  => clk,
        en     => '1',
        dataAddress => open,
        dataOut     => salida,
        dataIn      => entrada,
        readStrobe  => open,
        writeStrobe => sWriteStorbe,
        int         => '0',
        intAck      => open
    );
    --================================================
    -- output interface
    --================================================
    process (clk)
    begin
    if rising_edge(clk) then
            if sWriteStorbe ='1' then
                    led_reg <= salida;
            end if;
    end if;
    end process;
    led <= led_reg;

    --================================================
    -- input interface
    --================================================

    entrada <= sw;

    end arch;
```

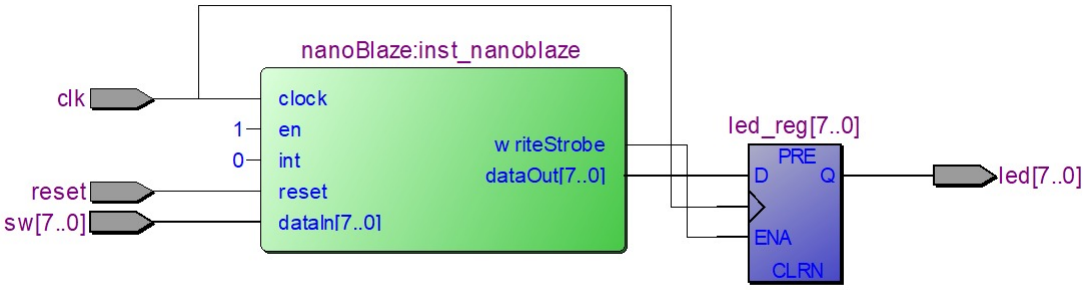

Figure 3: Nanoblaze con una simple interfaz I/O y registro de salida de *8bits*.

3 Secuencias experimentales

3.1 *Blink*

Para probar el microprocesador se procedió a programar un *Blinkg Led* en *Assembly* siguiendo el ejemplo del set de instrucciones.

3.1.1 Asembly

```
            LOAD    s2, 01
loop:       LOAD    s0, F0
            OUTPUT  s0, (s2)
            CALL    delay
            LOAD    s0, 0F
            OUTPUT  s0, (s2)
            CALL    delay
            JUMP    loop
delay:      LOAD    s1, 00
for:        ADD     s1, s2
            LOAD    s3, 00
for2:       ADD     s3, s2
            LOAD    s4, 00
for3:       ADD     s4, s2
            COMPARE s4, FF
            JUMP    NZ, for3
            COMPARE s3, FF
            JUMP    NZ, for2
            COMPARE s1, FF
            JUMP    NZ, for
            RETURN
```

3.1.2 Program Rom

El compilador dió como resultado el siguiente mapa de rom:

```
ARCHITECTURE mapped OF programRom IS
  subtype opCodeType is std_ulogic_vector(5 downto 0);
  constant opLoadC   : opCodeType := "000000";
  constant opLoadR   : opCodeType := "000001";
  constant opInputC  : opCodeType := "000100";
  constant opInputR  : opCodeType := "000101";
  constant opFetchC  : opCodeType := "000110";
  constant opFetchR  : opCodeType := "000111";
  constant opAndC    : opCodeType := "001010";
  constant opAndR    : opCodeType := "001011";
  constant opOrC     : opCodeType := "001100";
  constant opOrR     : opCodeType := "001101";
  constant opXorC    : opCodeType := "001110";
  constant opXorR    : opCodeType := "001111";
```

```
constant opTestC   : opCodeType := "010010";
constant opTestR   : opCodeType := "010011";
constant opCompC   : opCodeType := "010100";
constant opCompR   : opCodeType := "010101";
constant opAddC    : opCodeType := "011000";
constant opAddR    : opCodeType := "011001";
constant opAddCyC  : opCodeType := "011010";
constant opAddCyR  : opCodeType := "011011";
constant opSubC    : opCodeType := "011100";
constant opSubR    : opCodeType := "011101";
constant opSubCyC  : opCodeType := "011110";
constant opSubCyR  : opCodeType := "011111";
constant opShRot   : opCodeType := "100000";
constant opOutputC : opCodeType := "101100";
constant opOutputR : opCodeType := "101101";
constant opStoreC  : opCodeType := "101110";
constant opStoreR  : opCodeType := "101111";

subtype shRotCinType is std_ulogic_vector(2 downto 0);
constant shRotLdC : shRotCinType := "00-";
constant shRotLdM : shRotCinType := "01-";
constant shRotLdL : shRotCinType := "10-";
constant shRotLd0 : shRotCinType := "110";
constant shRotLd1 : shRotCinType := "111";

constant registerAddressBitNb : positive := 4;
constant shRotPadLength : positive
  := dataOut'length - opCodeType'length - registerAddressBitNb
   - 1 - shRotCinType'length;
subtype shRotDirType is std_ulogic_vector(1+shRotPadLength-1 downto 0);
constant shRotL : shRotDirType := (0 => '0', others => '-');
constant shRotR : shRotDirType := (0 => '1', others => '-');

subtype branchCodeType is std_ulogic_vector(4 downto 0);
constant brRet  : branchCodeType := "10101";
constant brCall : branchCodeType := "11000";
constant brJump : branchCodeType := "11010";
constant brReti : branchCodeType := "11100";
constant brEni  : branchCodeType := "11110";

subtype branchConditionType is std_ulogic_vector(2 downto 0);
constant brDo : branchConditionType := "000";
constant brZ  : branchConditionType := "100";
constant brNZ : branchConditionType := "101";
constant brC  : branchConditionType := "110";
constant brNC : branchConditionType := "111";

subtype memoryWordType is std_ulogic_vector(dataOut'range);
type memoryArrayType is array (0 to 2**address'length-1) of memoryWordType;

signal memoryArray : memoryArrayType := (
  16#000# => opLoadC   & "0010" & "00000001",
```

```
    16#001# => opLoadC   & "0000" & "11110000",
    16#002# => opOutputR & "0000" & "0010----",
    16#003# => brCall    & brDo   & "0000001000",
    16#004# => opLoadC   & "0000" & "00001111",
    16#005# => opOutputR & "0000" & "0010----",
    16#006# => brCall    & brDo   & "0000001000",
    16#007# => brJump    & brDo   & "0000000001",
    16#008# => opLoadC   & "0001" & "00000000",
    16#009# => opAddR    & "0001" & "0010----",
    16#00A# => opLoadC   & "0011" & "00000000",
    16#00B# => opAddR    & "0011" & "0010----",
    16#00C# => opLoadC   & "0100" & "00000000",
    16#00D# => opAddR    & "0100" & "0010----",
    16#00E# => opCompC   & "0100" & "11111111",
    16#00F# => brJump    & brNZ   & "0000001101",
    16#010# => opCompC   & "0011" & "11111111",
    16#011# => brJump    & brNZ   & "0000001011",
    16#012# => opCompC   & "0001" & "11111111",
    16#013# => brJump    & brNZ   & "0000001001",
    16#014# => brRet     & brDo   & "----------",
    others => (others => '0')
  );

BEGIN

  process (clock)
  begin
    if rising_edge(clock) then
      if en = '1' then
        dataOut <= memoryArray(to_integer(address));
      end if;
    end if;
  end process;

END ARCHITECTURE mapped;
```

3.2 *Contador*

Este programa muestra la cuenta por medio de 3 leds conectados a las salidas del microcontrolador embebido dentro de la fpga y se resetea cada vez que la cuenta iguala el numero ingresado en las entradas del mismo.

3.2.1 Asembly

```
                LOAD    s2, 01
sta:            INPUT   s5, 01
                LOAD    s0, 00
loop:           OUTPUT  s0, (s2)
                CALL    delay
```

```
                COMPARE   s0, s5
                JUMP      Z , sta
                ADD       s0, s2
                JUMP      loop
        delay:  LOAD      s1, 00
        for:    ADD       s1, s2
                LOAD      s3, 00
        ford:   ADD       s3, s2
                LOAD      s4, 00
        fort:   ADD       s4, s2
                COMPARE   s4, FF
                JUMP      NZ , fort
                COMPARE   s3, FF
                JUMP      NZ , ford
                COMPARE   s1, FF
                JUMP      NZ , for
                RETURN
```

3.3 Program Rom

```
ARCHITECTURE mapped OF programRom IS
  subtype opCodeType is std_ulogic_vector(5 downto 0);
  constant opLoadC   : opCodeType := "000000";
  constant opLoadR   : opCodeType := "000001";
  constant opInputC  : opCodeType := "000100";
  constant opInputR  : opCodeType := "000101";
  constant opFetchC  : opCodeType := "000110";
  constant opFetchR  : opCodeType := "000111";
  constant opAndC    : opCodeType := "001010";
  constant opAndR    : opCodeType := "001011";
  constant opOrC     : opCodeType := "001100";
  constant opOrR     : opCodeType := "001101";
  constant opXorC    : opCodeType := "001110";
  constant opXorR    : opCodeType := "001111";
  constant opTestC   : opCodeType := "010010";
  constant opTestR   : opCodeType := "010011";
  constant opCompC   : opCodeType := "010100";
  constant opCompR   : opCodeType := "010101";
  constant opAddC    : opCodeType := "011000";
  constant opAddR    : opCodeType := "011001";
  constant opAddCyC  : opCodeType := "011010";
  constant opAddCyR  : opCodeType := "011011";
  constant opSubC    : opCodeType := "011100";
  constant opSubR    : opCodeType := "011101";
  constant opSubCyC  : opCodeType := "011110";
  constant opSubCyR  : opCodeType := "011111";
  constant opShRot   : opCodeType := "100000";
  constant opOutputC : opCodeType := "101100";
  constant opOutputR : opCodeType := "101101";
  constant opStoreC  : opCodeType := "101110";
```

```
constant opStoreR  : opCodeType := "101111";

subtype shRotCinType is std_ulogic_vector(2 downto 0);
constant shRotLdC : shRotCinType := "00-";
constant shRotLdM : shRotCinType := "01-";
constant shRotLdL : shRotCinType := "10-";
constant shRotLd0 : shRotCinType := "110";
constant shRotLd1 : shRotCinType := "111";

constant registerAddressBitNb : positive := 4;
constant shRotPadLength : positive
  := dataOut'length - opCodeType'length - registerAddressBitNb
   - 1 - shRotCinType'length;
subtype shRotDirType is std_ulogic_vector(1+shRotPadLength-1 downto 0);
constant shRotL : shRotDirType := (0 => '0', others => '-');
constant shRotR : shRotDirType := (0 => '1', others => '-');

subtype branchCodeType is std_ulogic_vector(4 downto 0);
constant brRet  : branchCodeType := "10101";
constant brCall : branchCodeType := "11000";
constant brJump : branchCodeType := "11010";
constant brReti : branchCodeType := "11100";
constant brEni  : branchCodeType := "11110";

subtype branchConditionType is std_ulogic_vector(2 downto 0);
constant brDo : branchConditionType := "000";
constant brZ  : branchConditionType := "100";
constant brNZ : branchConditionType := "101";
constant brC  : branchConditionType := "110";
constant brNC : branchConditionType := "111";

subtype memoryWordType is std_ulogic_vector(dataOut'range);
type memoryArrayType is array (0 to 2**address'length-1) of memoryWordType;

signal memoryArray : memoryArrayType := (
  16#000# => opLoadC   & "0010" & "00000001",
  16#001# => opInputC  & "0101" & "00000001",
  16#002# => opLoadC   & "0000" & "00000000",
  16#003# => opOutputR & "0000" & "0010----",
  16#004# => brCall    & brDo   & "0000001001",
  16#005# => opCompR   & "0000" & "0101----",
  16#006# => brJump    & brZ    & "0000000001",
  16#007# => opAddR    & "0000" & "0010----",
  16#008# => brJump    & brDo   & "0000000011",
  16#009# => opLoadC   & "0001" & "00000000",
  16#00A# => opAddR    & "0001" & "0010----",
  16#00B# => opLoadC   & "0011" & "00000000",
  16#00C# => opAddR    & "0011" & "0010----",
  16#00D# => opLoadC   & "0100" & "00000000",
  16#00E# => opAddR    & "0100" & "0010----",
  16#00F# => opCompC   & "0100" & "11111111",
  16#010# => brJump    & brNZ   & "0000001110",
```

```
        16#011# => opCompC   & "0011" & "11111111",
        16#012# => brJump    & brNZ   & "0000001100",
        16#013# => opCompC   & "0001" & "11111111",
        16#014# => brJump    & brNZ   & "0000001010",
        16#015# => brRet     & brDo   & "----------",
        others => (others => '0')
    );

BEGIN

    process (clock)
    begin
      if rising_edge(clock) then
        if en = '1' then
          dataOut <= memoryArray(to_integer(address));
        end if;
      end if;
    end process;

END ARCHITECTURE mapped;
```

4 Videos de funcionamiento

Video 1: *Blink* **Clic aquí para ver el video.**
Video 2: *Contador* **Clic aquí para ver el video.**

5 Conclusión

En este trabajo practico nos permitió entender con mas a profundidad el funcionamiento del microcontrolador, además obtuvimos las bases para poder llegar a implementar nuestra propia arquitectura, o de otros, de microprocesador.

Aprendimos los conceptos básicos para la implementación de micros o CPU's, producidos por la comunidad, dándonos pie para poder emprender otros desarrollos.

Refrescamos conocimientos sobre programación a bajo nivel (*Assembly*) y resolvimos errores sobre el compilador.

En este caso en particular podemos observar la diferencia entre un microprocesador y una FPGA con mayor claridad. Mientras que con un CPU es multi-proposito y nos permite resolver tareas generales mas directamente, la FPGA permite resolver tareas precisas con una mayor eficiencia (siendo en este caso mas compleja la implementación).

Para consideración:

- Hay que tener en cuenta no repetir caracteres en cadena de las etiquetas del código assembly, ya que el compilador puede confundirse entre ellas.
- Si se desean implementar más registros en la interfaz I/O, la salida del *Nanoblaze* "port id" nos sirve como señal selectora para estos (figura 4).

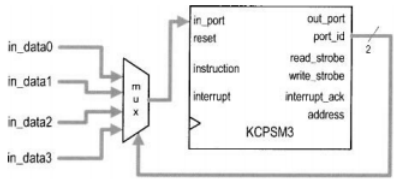

Figure 16.4 Block diagram of four continuous-access ports.

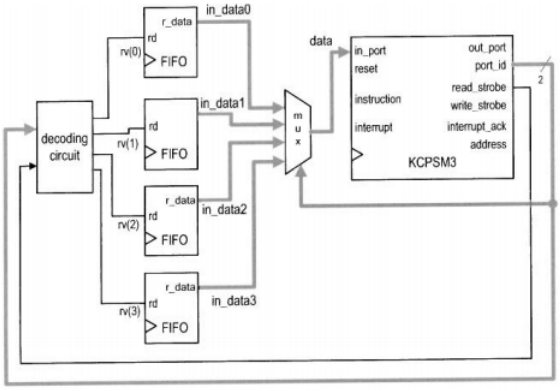

Figure 4: Referencia para implementar más registros en la interfaz I/0 utilizando port id