Modelação de um Processador em VHDL

André Neto

***Abstract* – This paper describes the development of a MIPS based processor using VHDL language. It ilustrastes the main components of a single cycle processor and how to describe them in VHDL. The target technology is an FPGA of Spartan3 family.**

***Resumo* – Este artigo mostra o desenvolvimento de um processador baseado na arquitectura MIPS usando a linguagem VHDL. Apresenta os principais componentes de um processador *single cycle* e como os descrever usando VHDL. A tecnologia alvo é uma FPGA da família Spartan3.**

***Keywords* – VHDL(Very High Speed Integrated Circuits Hardware Description Language), Reconfigurable Systems, Embedded Systems**

***Palavras chave* – VHDL(*Very High Speed Integrated Circuits Hardware Description Language*), Sistemas Reconfiguráveis, Sistemas Embutidos**

I. INTRODUÇÃO

No que se refere ao desenvolvimento e concepção de sistemas, um conceito cada vez mais comum e aceite é o de Sistemas Embutidos. Este é um tipo de sistema específico e plenamente dedicado ao seu objectivo. Sendo assim, é possível que os siatemas sejam optimizados para determinadas tarefas, aumentando assim o seu desempenho. Além disso, visto que possuem apenas características estritamente necessárias, tornam-se mais compactos e baratos (quando produzidos em grande escala).

Uma tecnologia que tem contribuído bastante para este conceito de sistemas são as FPGAs (*Field Programmable Gate Arrays*). As FPGAs consistem, de um modo simples, numa rede de elementos lógicos programáveis unidos por uma rede de ligações configurável. Algumas FPGAs mais recentes atingem uma complexidade bastante elevada, com um número significativo de elementos lógicos bem como de elementos mais específicos como memórias (RAM e ROM), multiplexadores, ALUs, etc. Desta forma é possível modelar no seu interior os diversos componentes de um mesmo sistema. Obtém-se assim um sistema dedicado a um dado objectivo e encapsulado num mesmo circuito integrado.

Indo ao encontro desta tecnologia (FPGA) é de interesse a linguagem de descrição de hardware VHDL (*Very High Speed Integrated Circuits Hardware Description Language*). Esta é uma das linguagens que nos permite, de um modo relativamente simples, fazer a modelação dos diversos componentes do sistema. Depois disso, com uma ferramenta de síntese apropriada poder-se-á configurar a FPGA para que implemente o sistema pretendido.

O trabalho aqui descrito aborda a modelação de um processador em VHDL para funcionamento numa FPGA. Baseado na arquitectura MIPS e com uma implementação *single cycle*, este processador possui apenas um pequeno conjunto de instruções, mas que pode ser tanto expandido como reduzido.

Neste artigo, apresenta-se uma breve descrição do processador (secção II), junto com alguns dos seus elementos. Na secção III são apresentadas algumas das características das instruções utilizadas num processador MIPS. Com respeito ao projecto em si, a secção IV apresenta alguns aspectos da modelação em VHDL e a secção V contém alguns resultados obtidos pela síntese. A conclusão encontra-se na secção VI.

Foi necessário efectuar alguma pesquisa para se efectuar este trabalho [1] [2] [3].

II. O PROCESSADOR

Um processador é uma máquina lógica que permite a execução de um conjunto de instruções de uma forma sequencial e de processar informação. Esta definição torna-se muito abrangente mas permite ter uma ideia do que se pretende neste estudo. Para que possa cumprir com este objectivo, existem alguns elementos essenciais que necessitam de ser considerados. A figura 1 apresenta alguns destes elementos para um processador simples.

A. PC - program counter

O *program counter* consiste num registo que é actualizado no início de cada ciclo. É o valor deste contador que determina qual o endereço da instrução que será executada nesse mesmo ciclo. Conforme será abordado em II-C, este contador só poderá assumir valores múltiplo de 4, em binário corresponde aos dois bits menos significativos serem iguais a zero.

B. Somadores

Na estrutura do processador a modelar são necessários dois somadores. Ambos são usados para determinar o endereço da próxima instrução a executar. Um deles soma o valor constante de 4 ao endereço da instrução actual (PC - *program counter*), para sequência de instruções seguida. O outro permite somar um valor arbitrário ao endereço da instrução seguinte (PC+4).

C. Memória

Um dos elementos essenciais de um processador é a memória ou memórias. No caso de uma implementação *single cycle*, como é o caso de estudo, é necessário a existência de duas memórias, uma para as instruções e outra para dados.

No caso da arquitectura MIPS, em que as instruções são de tamanho fixo e igual a 32 bits, a memória de instruções

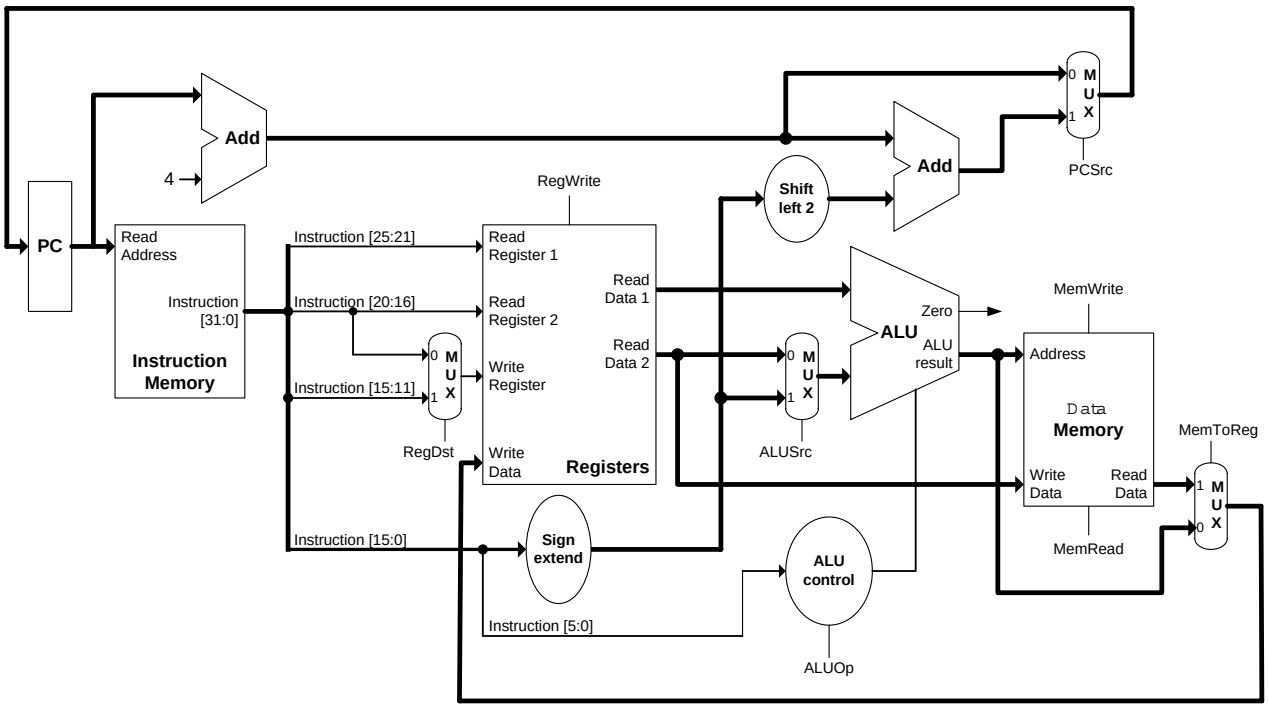

Fig. 1 - Data Path do processador (simplificado).

(*instruction memory*) é *word addressable*, isto é, permite o acesso apenas a palavras de tamanho fixo. Este tamanho é definido pelo tamanho da instrução, 32 bits. Além disso, visto que o programa ou conjunto de instruções não é alterado durante a sua execução, esta memória é apenas de leitura (*read-only*). Desta forma temos uma memória da qual se podem ler palavras de 32 bits que correspondem às instruções a executar.

A segunda memória necessária, é usada para dados (*data memory*). É nesta memória que se encontram os dados a processar. Torna-se assim necessário que seja possível tanto ler como escrever nesta memória. Em alguns casos esta memória pode não ser uma memória real como uma RAM ou ROM. Encontra-se em alguns sistemas componentes que podem ser mapeados na memória por meio de um descodificador, assim quando se lê no endereço especificado, na realidade está-se a aceder a esse mesmo componente. Isto permite a troca de informação entre o processador e outros componentes, quer para aquisição de dados do exterior, para controlo ou configuração. No presente caso de estudo, esta memória é também endereçada apenas à palavra (32 bits).

A memória de dados é assíncrona na leitura e síncrona na escrita. Isto permite que quando se pretende ler informação da memória ela esteja disponível bem cedo no ciclo de processamento. Os elementos de atraso ou espera pela informação correspondem apenas ao tempo de propagação dos sinais e do tempo de resposta da memória. Por outro lado, a escrita síncrona com o fim do ciclo de processamento, procura garantir que a informação esteja estável quando é escrita na memória.

Operação	**Descrição**
AND	Realiza a operação lógica *and* bit a bit entre os dois operandos e apresenta o resultado na saída
OR	Realiza a operação lógica *or* bit a bit entre os dois operandos e apresenta o resultado na saída
ADD	Adiciona dois operandos e apresenta o resultado na saída.
SUB	subtrai o valor do segundo operando ao do primeiro
set on less then	Compara o valor entre os dois operandos (por subtracção), o resultado é 1 se o primeiro operando for inferior ao segundo

TABELA I
DESCRIÇÃO DAS OPERAÇÕES NA ALU

D. ALU - arithmetic logic unit

É neste elemento que se concentram as principais capacidades de processamento. Neste caso de estudo encontram-se na ALU as operações descritas na tabela I.

Este elemento é na realidade um circuito combinatório, cujo resultado apresentado na saída depende de um conjunto de sinais de controlo que lhe são aplicados.

Estes sinais de controlo são, para o presente caso de estudo, *bInvert* - faz a negação bit a bit do segundo operando, *carryIn* - bit de *carry*, *operation* - define a operação base a ser realizada. A conjugação destes vários sinais permitem obter as operações descritas na tabela I.

E. Registos

A arquitectura MIPS tem por base um número elevado de registos, sobre os quais são realizadas as diversas operações. É destes registos que são obtidos os valores para os operandos da ALU e é neles que são guardados os resultados das diversas operações.

Nesta arquitectura são usados 32 registos numerados de 0 (zero) a 31 (trinta e um). Apenas o registo zero tem a particularidade de ter um valor fixo e predefinido que é 0 (zero), sendo um registo apenas de leitura. Todos os outros podem ser usados como leitura e escrita.

No que se refere à troca de informação com a memória de dados, são usados os registos. Este tipo de arquitectura não permite a operação directa sobre a memória. Assim sendo é necessário recorrer a operações de *load* e *store* para operar sobre dados da memória.

Os registos estão inseridos num dos elementos do processador. O interface deste elemento determina como os registos podem ser acedidos para leitura e para escrita. Este módulo permite o acesso de leitura a dois dos registos por vez, e escrita em um deles. A escrita neste registo é determinada por um sinal de controlo (*RegWrite*).

Embora a maior parte dos outros elementos seja apenas combinatória, a escrita num registo é feita sincronizada com o fim do ciclo de execução, determinado pelo sinal de relógio. Isto visa garantir que o valor a escrever bem como os sinais de controlo estejam estáveis e bem definidos.

F. Sign extend

O elemento *sign extend* permite extender um número de 16 bits para 32 bits de acordo com as regras de representação em complemento para 2. Neste tipo de representação, faz-se isso por colocar os 16 bits originais nos 16 bits menos significativos do número resultante, preenchendo os restantes bits com o valor do bit mais significativo do número original.

G. Shift left 2

Este elemento simples executa na realidade uma multiplicação por 4. Visto que o valor do *program counter* é sempre múltiplo de 4 os dois bits menos significativos deste número são sempre zero. Desta forma, qualquer valor que lhe seja adicionado deve respeitar esta condição. Conciliando esta propriedade ao elemento *shift left 2*, pode-se omitir os dois bits menos significativos deste valor. Visto que valor do "multiplicando"é proveniente do *sign extend* pode-se deste modo ampliar a gama valores a somar ao PC+4 de -2^{15} a $2^{15}-1$ para -2^{17} a $2^{17}-1$ (ver II-A, II-B e II-F).

H. Multiplexadores

Em muitos dos elementos que compõem o processador, existe a necessidade de ter acesso a mais que uma fonte de informação. Por exemplo, para se escrever no *program counter* existem dois valores possíveis conforme considerado em II-B. Para que não haja conflito nos barramentos e se tenha controlo sobre o valor que realmente é utilizado, é necessário recorrer a multiplexadores que permitem fazer esta mesma selecção. Os sinais de controlo do multiplexer são determinados de acordo com a instrução por uma unidade de controlo.

Tipo	31 26	25 21	20 16	15 11	10 6	5 0
R	*opcode (6bits)*	*rs (5bits)*	*rt (5bits)*	*rd (5bits)*	*shamt (5bits)*	*function (6bits)*
I	*opcode (6bits)*	*rs (5bits)*	*rt (5bits)*	*immediate (16bits)*		
J	*opcode (6bits)*	*address (26bits)*				

Fig. 2 - Tipos de Instruções e respectivos campos.

I. Unidades de controlo

Para além da unidade de controlo da ALU, que determina qual a operação a ser realizada (ver: II-D e tabela I), existe uma unidade de controlo para o processador. Esta unidade determina o valor dos vários sinais de controlo no interior do processador. São este sinais de controlo que por sua vez determinam qual a operação que está a ser realizada pelo processador. Observando a figura 1, pode-se detectar a existência destes mesmos sinais em várias das unidades do processador (multiplexadores, registos, memória de dados e unidade de controlo da ALU).

É com base na instrução lida, que a unidade de controlo determina o valor dos vários sinais de controlo. Esta unidade, funciona na realidade como um descodificador das instruções.

III. AS INSTRUÇÕES

As instruções utilizadas na arquitectura MIPS, estão divididas em três tipos: R, I e J. Todas as instruções começam com um campo de 6 bits denominado *opcode*. Para além do primeiro campo, as instruções tipo R, possuem 3 campos de 5 bits que contêm o índice de 3 registos, um campo para instruções de *shift* e um campo para escolher uma função. As instruções do tipo I, possuem apenas 2 campos para seleccionar registos, seguidos de um campo de 16 bits para um valor "imediato"(*immediate value*). Instruções do tipo J, para além do primeiro campo possuem apenas um campo de 26 bits usado para instruções de *jump*. Os diferentes tipos de instrução estão resumidos na figura 2.

A. Tipo R

Este grupo de instruções é composto por todas as instruções que não necessitam de constantes, *offsets* para *branchs* ou saltos na sequência do programa, endereços de memória ou conteúdos de memória. Fazem parte dele, operações aritméticas com o conteúdo dos registos, operações de *shift* e saltos directos para endereço guardado em registo (jalr e jr).

A.1 Opcode

Todas estas instruções usam o *opcode* 000000(2).

A.2 rs, rt, rd

A maior parte de instruções do tipo R, são operações realizadas sobre o conteúdo de registos. Assim estes três campos permitem escolher quais os registos a usar para efectuar a operação, bem como o seu papel na operação. Dois destes são usados como operandos, rs define o primeiro operando e rt o segundo operando. O terceiro registo, rd, é usado

Instrução	Operação	opcode	function
add	rd = rs + rt	000000(bin)	100000(bin)
sub	rd = rs - rt	000000(bin)	100010(bin)
and	rd = rs & rt	000000(bin)	100100(bin)
or	rd = rd | rt	000000(bin)	100101(bin)
slt	rd = 1 when rd < rt else 0	000000(bin)	101010(bin)

TABELA II
CAMPOS OPCODE E FUNCTION DAS INSTRUÇÕES TIPO R.

como registo destino onde é guardado o resultado. Nada obriga a que os registos usados sejam distintos. É de notar que a escrita sobre o registo zero resulta na perda de informação.

A.3 shamnt

Este campo é usado em operações de *shift*, permitindo determinar o número de bits a deslocar o operando.

A.4 function

Visto que as diversas instruções do tipo R partilham o mesmo *opcode*, este campo é usado para distinguir as diversas operações possíveis.

B. Tipo I

Este grupo inclui todas as operações que usem constantes, operações de salto relativo (*branch*) e instruções de *load* e *store*.

B.1 opcode

Este tipo de instrução usa todos os *opcodes* excepto 000000(2), 00001x(2), e 0100xx(2).

B.2 rs, rt

Algumas das instruções do tipo I incluem operações entre um registo e uma constante, cujo resultado é armazenado num outro registo. Neste caso rs define o operando e rt o registo destino do valor resultante.

Quanto às operações de *load* e *store*, o valor no registo rs define um endereço de memória ao qual será adicionado um valor de *offset*, o que resultará no endereço de memória a ser lido ou escrito. O registo rt define de onde deve ser lido o valor a escrever na memória, no caso de *store*, ou onde deve ser guardado o valor lido da memória, no caso de um *load*.

No caso da instrução BEQ (*branch if equal*), o conteudo dos registos definidos por rs e rt, são comparados para verificar a condição de salto relativo.

B.3 immediate

Este campo da instrução define o valor de uma constante. Este valor pode servir como operando ou como *offset* numa instrução de *load* ou *store*. No caso de instruções de salto relativo, é este campo que define o número de instruções a avançar ou recuar.

C. Tipo J

O grupo de instruções do tipo J consistem em duas instruções de salto directo (*jump*). Estas são: j e jal. Ambas requerem um endereço de memória como seu operando.

C.1 opcode

Instruções do tipo J usam os *opcodes* 00001x(2).

C.2 address

Este campo da instrução define qual o endereço de memória onde se encontra a próxima instrução a ser executada.

IV. MODELAÇÃO COM VHDL

Uma das características apreciadas na linguagem VHDL é a possibilidade de dividir um projecto complexo em módulos mais simples. Isto facilita o desenvolvimento do projecto, bastando que no fim se interliguem os diversos módulos. Isto permite também que um mesmo módulo seja replicado várias vezes pelo projecto.

Esta característica foi utilizada no desenvolvimento deste trabalho.

A. Memórias

São necessários dois blocos de memória para este processador. Tendo em conta que estas duas memórias têm características distintas torna-se necessário criar um módulo distinto para cada uma delas.

A.1 Instruções

Esta é apenas uma memória de leitura, onde estão armazenadas as instruções a realizar. Cada instrução corresponde a uma palavra de 32 bits. Teremos assim um *array* de palavras de 32 bits. O interface desta memória é bastante simples, exige apenas um porto de entrada para o endereço da instrução e outro de saída para a instrução respectiva.

No seu comportamento, temos o endereço da instrução que vai servir como índice para o *array* de instruções. Visto que esta memória é endereçada à palavra (de 32 bits), os 2 bits menos significativos podem ser ignorados pois deverão ser sempre iguais a zero. Ao fornecer o endereço da instrução no respectivo porto, será disponibilizado o respectivo valor da instrução.

É também possível na modelação da memória iniciar o seu valor. Embora existam outras possibilidades para o fazer, devido a questões de simplicidade a ”programação” desta memória é feita junto com a sua modelação. Também por uma questão de simplicidade, a memória modelada neste trabalho possui apenas 256 palavras disponíveis para instruções, sendo necessário uma palavra de apenas 8 bits para endereçar todas elas.

A modelação desta memória (*Instruction Memory*) está apresentada na figura 3.

A.2 Dados

A memória de dados é tanto de leitura como de escrita. Além disso pode ter características tanto síncronas como assíncronas, consoante se leia ou se escreva. Por uma

```
entity InstructionMemory is
 Port (
  imRAddress: in std_logic_vector(31 downto 0);
  instruction: out std_logic_vector (31 downto 0)
 );
end InstructionMemory;

architecture Behavioral of InstructionMemory is

type TRAM is array(0 to 255) of
                std_logic_vector(31 downto 0);

signal ram_block : TRAM :=
(0 => (others => '0'),
 1 => "00100000000000010111010000000000",
 2 => "00110100000000110001000000011011",
 others => x"00000000");

begin
 instruction <=
  ram_block(conv_integer(imRAddress(9 downto 2)));
end Behavioral;
```

Fig. 3 - Modelação da memória de instruções.

questão de simplicidade de projecto e síntese do mesmo, optou-se por ter leitura e escrita síncronas. A escrita é feita no fim do ciclo de processamento, síncrona no flanco ascendente do sinal de relógio do processador. Por outro lado a leitura é feita a meio do ciclo de processamento. Isto permite que os sinais de controlo e de endereço estabilizem antes de se fazer a leitura, e não compromete a execução da instrução. Esta operação é assim síncrona ao flanco de relógio contrário ao da escrita.

A estrutura de dados é similar à memória de instruções. Embora alguns processadores tenham memória endereçável ao byte, mais uma vez por questões de simplicidade, neste projecto a memória é endereçável à palavra (32 bits).

A modelação desta memória (*Data Memory*) está apresentada na figura 4.

B. Registos

Embora seja também um módulo distinto, devido à sua preponderância no projecto, a modelação dos registos foi feita no módulo de mais alto nível do sistema, aquele que reúne e interliga todos os outros.

Nesta modelação foi necessário ter em conta os sinais de controlo para a escrita e que o registo zero tem valor fixo e não pode ser alterado.

Contrário às memórias que se pretende que sejam sintetizadas na forma de memória RAM ou ROM, conforme o seu objectivo, não se pretende isso com os registos. Visto que se está a utilizar VHDL ao nível comportamental, a escolha dos elementos a utilizar na síntese fica ao cuidado da ferramenta utilizada. No entanto, há alguns aspectos na descrição comportamental do elemento que podem influenciar a escolha a realizar pela ferramenta de síntese. No presente caso temos o uso de um sinal de *reset* e de se ter simultaneamente um interface síncrono e outro assíncrono, o que condiciona à escolha de LUTs para a síntese deste elemento.

Nesta modelação pode-se notar a invocação de um elemento multiplexador. A sua função pode ser facilmente

```
entity dataMemory is
 Port (
  clk       : in  std_logic;
  dmAddress : in  std_logic_vector (31 downto 0);
  wData     : in  std_logic_vector (31 downto 0);
  rData     : out std_logic_vector (31 downto 0);
  memRead   : in  std_logic;
  memWrite  : in  std_logic);
end dataMemory;

architecture Behavioral of dataMemory is
 type TRAM is array(0 to 255) of
                 std_logic_vector(31 downto 0);

 signal ram_block : TRAM;

begin

process(clk)
begin
 if falling_edge(clk) then
  if memRead = '1' then
   rData <= ram_block(conv_integer(
                        dmAddress(7 downto 0)));
  end if;
 end if;

 if rising_edge(clk) then
  if memWrite = '1' then
   ram_block(conv_integer(
              dmAddress(7 downto 0))) <= wData;
  end if;
 end if;
end process;

end Behavioral;
```

Fig. 4 - Modelação da memória de dados.

depreendida por consultar a figura 1.

A modelação destes registos está apresentada na figura 5.

C. ALU

Este elemento é o centro de todo o sistema, é em torno dele que toda a informação (dados e controlo) circula. Colocado num modulo individual, este possui duas entradas e uma saída de dados de 32 bits cada, 4 bits de controlo e 2 bits de estado. Na sua modelação, são usados alguns sinais internos. Estes elementos podem ser observados na figura 6 (Nesta figura a descrição está simplificada por questões de apresentação).

Visto que a ALU é um circuito combinatório, as operações base são efectuadas em paralelo. Os 2 bits de *operation* fazem na realidade a multiplexagem dos resultados das diversas operações. Também os sinais de estado estão sempre disponíveis. A descrição comportamental deste módulo pode ser observada na figura 7 (Nesta figura a descrição está simplificada por questões de apresentação).

D. Unidades de controlo

Existem duas unidades de controlo, uma delas encontra-se explicita na figura 1, a outra está implícita.

A unidade de controlo da ALU, como o nome indica, tem como objectivo determinar o valor dos sinais de controlo da ALU, de acordo com a instrução a realizar. De um modo geral, esta unidade está dependente da unidade de controlo

```
architecture Behavioral of general is
 type TREGISTERS is array(0 to 31) of
                 std_logic_vector(31 downto 0);

 signal s_registers  : TREGISTERS :=
                       (others => x"00000000");
 signal s_regData1:std_logic_vector(31 downto 0);
 signal s_regData2:std_logic_vector(31 downto 0);
 signal s_wReg    :std_logic_vector(4  downto 0);
 signal s_wData   :std_logic_vector(31 downto 0);

 signal s_instRS : std_logic_vector( 4 downto 0);
             -- nº do reg com 1º operando fonte
 signal s_instRT : std_logic_vector( 4 downto 0);
             -- nº do reg com 2º operando fonte
 signal s_instRD : std_logic_vector( 4 downto 0);
             -- nº do reg destino do resultado
begin
 s_instRS    <= s_instruction(25 downto 21);
 s_instRT    <= s_instruction(20 downto 16);
 s_instRD    <= s_instruction(15 downto 11);

 s_regData1<=s_registers(conv_integer(s_instRS));
 s_regData2<=s_registers(conv_integer(s_instRT));

 write_proc : process(s_regWriteCtrl, s_pcClk)
 begin
  if s_rst = '0' then
   s_registers <= (others => (others => '0'));
  elsif (rising_edge(s_pcClk)) then
   if (s_regWriteCtrl = '1') then
    if (s_wReg /= "00000") then
     s_registers(conv_integer(s_wReg))<=s_wData;
    end if;
   end if;
  end if;
 end process;

 wRegSourceMux : entity work.muxSIZE
 generic map (5)
 port map (  scr     => s_regDestCtrl,
             inZero  => s_instRT,
             inOne   => s_instRD,
             mux     => s_wReg );

end Behavioral;
```

Fig. 5 - Modelação dos registos.

do processador. No presente caso, o valor dos sinais de controlo da ALU são também determinados pelos campos da instrução, no caso de instruções do tipo R.

A unidade de controlo do processador, é a unidade responsável por determinar o valor de cada um dos sinais de controlo (excepto os da ALU). Torna-se na realidade um descodificador da instrução. Este descodificador poderá ser tanto um circuito lógico, como uma memória ROM. Recorrendo a VHDL, essa é uma decisão que fica do lado da ferramenta de síntese utilizada.

Desta forma apresentamos na figura 8 a declaração da entidade da unidade de controlo da ALU e na figura 9 a declaração da unidade de controlo do processador. Na figura 10 um exemplo de uma descrição comportamental a usar neste tipo de circuitos, este exemplo não tem correspondência com a figura 9, é apenas ulustrativo.

```
entity ALU is
 Port (
  a        : in  std_logic_vector (31 .. 0);
  b        : in  std_logic_vector (31 .. 0);
  operation: in  std_logic_vector (1  .. 0);
  bInvert  : in  std_logic;
  carryIn  : in  std_logic;
  result   : out std_logic_vector (31 .. 0);
  zero     : out std_logic;
  overflow : out std_logic);
end ALU;

architecture Behavioral of ALU is
 signal s_operand1  : std_logic_vector (31 .. 0);
 signal s_operand2  : std_logic_vector (31 .. 0);
 signal s_addResult : std_logic_vector (32 .. 0);
 signal s_orResult  : std_logic_vector (31 .. 0);
 signal s_andResult : std_logic_vector (31 .. 0);
 signal s_lessResult: std_logic_vector (31 .. 0);

begin
  ...
end Behavioral;
```

Fig. 6 - Modelação da ALU (interface).

```
entity ALU is
 Port ( ... );
end ALU;

architecture Behavioral of ALU is
  ...
begin
 s_operand1 <= a;
 -- INVERT --
 s_operand2 <= b when (bInvert = '0')
                 else (not b);
 -- ADDER --
 s_addResult <=
            ('0'& s_operand1) + ('0'& s_operand2)
          + (x"0000000"&"000"&carryIn);
 -- AND --
 s_andResult<= s_operand1 and s_operand2;
 -- OR --
 s_orResult <= s_operand1 or  s_operand2;
 -- SET_ON_LESS_THAN --
 s_lessResult(0) <= s_addResult(31);
 s_lessResult(31 downto 1) <= (others => '0');
 -- ZERO --
 zero <= '1' when (s_addResult(31 downto 0) =
                                  (x"00000000"))
             else '0';
 -- MUX --
 result <= s_andResult when (operation="00") else
    s_orResult         when (operation="01") else
    s_addResult(31 downto 0)
                       when (operation="10") else
    s_lessResult;
end Behavioral;
```

Fig. 7 - Modelação da ALU (descrição comportamental).

V. SÍNTESE

Para efectuar a síntese deste projecto, recorreu-se à ferramenta de síntese disponibilizada pela Xilinx, XST.

Para este projecto, foi utilizado como dispositivo alvo uma FPGA Spartan3 xc3s400-4pq208 (*device*: XC3S400; *speed grade*: 4; *package*: pq208).

Como resultado da síntese deste projecto destacam-se algumas das opções feitas pela ferramenta.

```
entity ALUctrlUnit is
 Port (
  ALUop1   : in  std_logic;
  ALUop2   : in  std_logic;
  ALUop3   : in  std_logic;
  funct    : in  std_logic_vector (5 downto 0);
  ALUctrl  : out std_logic_vector (2 downto 0);
  opcode   : in  std_logic_vector (5 downto 0));
end ALUctrlUnit;

architecture Behavioral of ALUctrlUnit is
begin
  ...
end Behavioral;
```

Fig. 8 - Modelação da unidade de controlo da ALU (interface).

```
entity ctrlUnit is
 Port (
  reset    : in  std_logic;
  opcode   : in  std_logic_vector (31 downto 26);
  regDst   : out std_logic;
  ALUsrc   : out std_logic;
  mem2reg  : out std_logic;
  regWrite : out std_logic;
  memRead  : out std_logic;
  memWrite : out std_logic;
  branch   : out std_logic;
  ALUop1   : out std_logic;
  ALUop2   : out std_logic;
  ALUop3   : out std_logic);
end ctrlUnit;

architecture Behavioral of ctrlUnit is
begin
  ...
end Behavioral;
```

Fig. 9 - Modelação da unidade de controlo do processador (interface).

A. *Memória de dados*

Para a síntese desta memória, é usada uma *block-ram* "256x32-bit single-port distributed RAM". Com base na descrição comportamental da memória, a ferramenta de síntese pôde depreender que se tratava de um bloco de memória RAM, de 256 palavras (visto que se utiliza apenas 8 bits de endereço), com o *write enable* activo ao sinal lógico "1" e síncrona com flanco ascendente do sinal de relógio.

B. *Unidade de Controlo*

Conforme já mencionado em IV-D, a síntese deste tipo de unidades pode ser feita por lógica ou por recurso a memórias do tipo ROM que actuam como descodificadores.

Neste projecto, a síntese da unidade de controlo do processador, foi feita com recurso a memórias ROM. Estas efectuam assim a descodificação da instrução, determinando qual o valor para cada um dos sinais de controlo. Não é explicito no relatório de síntese, mas é usada alguma lógica adjacente nesta unidade. Isso fica claro por recorrer ao *View RTL Schematic* disponível na ferramenta de síntese.

Por outro lado, pode-se do mesmo modo observar que a unidade de controlo da ALU é toda ela sintetizada por meio de LUTs, não fazendo recurso a blocos de memória.

```
architecture Behavioral of CtrlUnit is
begin
  process
   if (reset = '0') then
     ctrlOut1 <= '0';
     ctrlOut2 <= '0';
     ctrlOut3 <= '0';
   else
     case (instruction) is
       when "0000" =>
         ctrlOut1 <= '1';
         ctrlOut2 <= '0';
         ctrlOut3 <= '1';
       when "0001" =>
         ctrlOut1 <= '1';
         ctrlOut2 <= '1';
         ctrlOut3 <= '1';
       when "0010" =>
         ctrlOut1 <= '0';
         ctrlOut2 <= '0';
         ctrlOut3 <= '1';
       ...
       when others =>
         ctrlOut1 <= '0';
         ctrlOut2 <= '0';
         ctrlOut3 <= '0';
end Behavioral;
```

Fig. 10 - Modelação tipo de uma unidade de controlo.

VI. CONCLUSÃO

Embora com um conjunto de instruções limitado, este é um exemplo válido de um processador baseado na arquitectura MIPS. As suas instruções implementadas são: ADD, ADDI, AND, ANDI, BEQ, LW, OR, ORI, SLT, SLTI, SUB, SW (baseado nas instruções *assembly* do MIPS). Estas instruções básicas permitem já efectuar algumas rotinas que poderiam ser úteis num sistema simples. A existência de instruções para lidar com memória, para além de permitir armazenar e ler valores da memória, tornam possível a interacção com outros dispositivos de interface compatível. As instruções BEQ, SLT e SLTI podem ser usadas para criar ciclos. Neste artigo consideramos apenas a modelação do processador. No entanto, esta modelação pode ser utilizada em outros projectos.

Com o objectivo de testar este processador, a sua memória de instruções foi inicializada, e no projecto foi inserido um módulo VGA, que permitiu observar o conteúdo dos registos. Desta forma foi possível observar o bom funcionamento do processador.

Este trabalho no âmbito dos Sistemas Digitais Reconfiguráveis, deixa claro a versatilidade dos modernos sistemas digitais com recurso a FPGAs.

REFERÊNCIAS

[1] Iouliia Skliarova, "Desenvolvimento de circuitos reconfiguráveis que interagem com um monitor vga", em *Electrónica e Telecomunicações*, 2005, vol. 4, pp. 626–631.

[2] Nuno Lau, *Arquitectura de Computadores - Slides*, DET-UA, Campus Universitário de Santiago, Aveiro, Portugal, 2006/07.

[3] D.A.Patterson J.Hennessy, "Computer organization and design - the hardware/software interface", 2004, Elsevier.