Cleo-LIRMM: Um Experimento de Implementação de Processadores Dedicados em Plataformas de Prototipação de Sistemas Embarcados

Fernando Gehm Moraes [1]
Ney Laert Vilar Calazans [2]
Felipe Rocha da Silva [3]
Maurício da Silva Barrios [3]

Pontifícia Universidade Católica do Rio Grande do Sul (PUCRS)
Instituto de Informática
Av. Ipiranga, 6681 - Prédio 30 / BLOCO 4
Telefone: +55 51 320-3611 - Fax: +55 51 320-3621
CEP 90619-900 - Porto Alegre - RS - BRASIL

Abstract

We describe the implementation of the Cleópatra core processor on the LIRMM embedded systems' prototyping platform. The processor core, together with the modules that allow it to communicate with the rest of the platform were implemented on a single FPGA, starting from their complete description in the VHDL language. The communication modules perform three main tasks: object code loading into the core processor external memory, firing of the object code execution by the core and output data retrieval after object code execution.

This work is a first complete case study developed by the authors on the construction of dedicated architectures aimed at embedded systems, as well as its integration on digital systems' fast prototyping and/or emulation platforms.

Resumo

Descreve-se a implementação do processador didático Cleópatra sobre a plataforma de prototipação de sistemas embarcados LIRMM. O núcleo do processador e os módulos de comunicação com o restante da plataforma foram implementados sobre um único FPGA, a partir de suas descrições completas em VHDL. Os módulos de comunicação desempenham as funções de controle de carga do código objeto na memória, disparo da execução do código objeto pelo processador e recuperação dos dados de saída após a execução do programa.

O trabalho aqui descrito constitui o primeiro estudo de caso completo desenvolvido pelos autores sobre a construção de arquiteturas dedicadas para sistemas embarcados e sua integração em plataformas de prototipação e/ou de emulação de sistemas digitais.

---

1. Doutor em Informática, opção Microeletrônica (LIRMM, França, 1994), Engenheiro Eletrônico (UFRGS, 1987), Professor Adjunto do Instituto de Informática/PUCRS.
   E-mail: moraes@andros.inf.pucrs.br
2. Doutor em Ciências Aplicadas, opção Microeletrônica (UCL, Bélgica, 1993), Engenheiro Eletrônico (UFRGS, 1985), Professor Adjunto do Instituto de Informática/PUCRS.
   E-mail: calazans@andros.inf.pucrs.br
3. Alunos do Bacharelado em Informática da PUCRS

1. INTRODUÇÃO

O avanço tecnológico na construção de sistemas digitais (SDs) complexos é tal que permite dispor hoje de um único circuito integrado (CI) contando com cerca de 10^7 transistores. Aliado a tempos médios de vida de produtos eletrônicos abaixo de dois anos, este fato impõe que o processo de projeto de tais componentes resulte em dispositivos satisfatoriamente funcionais já na primeira rodada de fabricação. Assim, há uma aguda necessidade de métodos de validação de projeto eficientes e viáveis de serem aplicados ao projeto global. Como métodos de simulação elétrica e/ou lógica são inviáveis para lidar com as dimensões de tais SDs, apela-se cada vez mais freqüentemente para técnicas tais como simulação funcional, verificação formal, aceleração de hardware, emulação e prototipação, num espectro de atividades de validação com desempenho relativo entre 1/1 (o desempenho do produto final) e $1/10^9$(o desempenho de uma simulação funcional em VHDL) [1][2].

Neste contexto, a validação de SDs via prototipação rápida mediante uso de FPGAs assume crescente importância, pois alia alto desempenho (tipicamente com um valor relativo de 1/10 na escala acima introduzida) a uma flexibilidade sem precedentes, com FPGAs baseados em RAM tornando estruturas de hardware tão fáceis de serem alteradas quanto software. Contudo, um determinado CI específico para uma dada aplicação (ASIC) nunca constitui *per se* um produto. Para garantir a validação do ASIC em um contexto realista, emprega-se ou um protótipo completo do produto (exceto pelo ASIC em projeto que é substituído por um FPGA) ou uma plataforma de prototipação standard sob algum aspecto e contendo FPGAs sobre os quais o ASIC é implementado. Uma amostra do sucesso desta última abordagem é a grande quantidade de plataformas de prototipação, comerciais e acadêmicas disponíveis. Steve Guccione disponibiliza uma lista de quase uma centena de máquinas baseadas em FPGAs em http://www.io.com/~guccione/HW_list.html, muitas das quais são plataformas de prototipação de SDs.

Este artigo descreve o desenvolvimento de um processador para uso em sistemas embarcados (embedded systems) e sua implementação e validação sobre uma plataforma de prototipação baseada em processadores DSP e FPGAs, desenvolvida no laboratório LIRMM da Universidade de Montpellier e em uso pelo Grupo de Apoio ao Projeto de Hardware (GAPH) da PUCRS. A Seção 2 apresenta a arquitetura do processador Cleópatra de forma resumida. Segue-se uma discussão da implementação em VHDL e a validação funcional do projeto, nas Seções 3 e 4, respectivamente. A Seção 5 discute a plataforma de prototipação LIRMM, bem como a implementação de Cleópatra sobre esta, com ênfase na adição de estruturas de hardware para garantir a correta interação com o computador hospedeiro(onde hardware e software foram desenvolvidos), com o processador DSP da plataforma LIRMM e com a memória de Cleópatra. A Seção 6 apresenta algumas conclusões e trabalhos futuros.

2. DESCRIÇÃO DA ARQUITETURA CLEÓPATRA

O processador Cleópatra é usado como estudo de caso inicial em disciplinas de Organização de Computadores (teórica e prática) do curso de Bacharelado em Informática da PUCRS. Trata-se de uma arquitetura simples com registrador acumulador, onde endereços e dados são de 8 bits, existem 13 códigos de operação e 4 modos de endereçamento, combinados em um total de 37 instruções distintas. A implementação da Unidade Central de Processamento (CPU) possui dois blocos (modelo clássico de von Neumann): o bloco de controle (BC) e o bloco de dados (BD). Programas e os dados são armazenados em uma memória externa. A comunicação entre BC e BD é feita através de 24 sinais, sendo 13

destinados a codificar a palavra de controle (microinstrução), 3 implementando qualificadores gerados pela última operação relevante da Unidade Lógico-Aritmética (ULA) (vai-um de operações aritméticas - c, resultado zero - z, resultado negativo – n) e 8 sinais para transmitir a código da instrução em execução, armazenado no registrador de instruções (IR) do BD. A Figura 1 mostra o diagrama de blocos Cleópatra e sua comunicação com a memória externa.

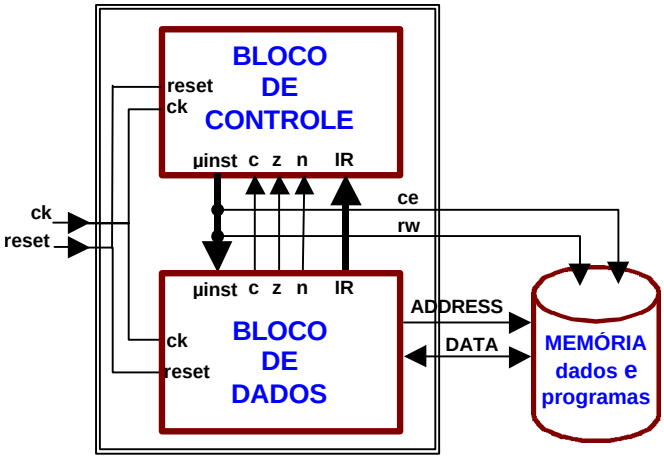

Figura 1 – Diagrama de blocos do processador Cleópatra e comunicação com a memória externa.

O BD é composto por 6 registradores, uma ULA de 8 bits e decodificadores de escrita e leitura. A ULA realiza as operações aritméticas de soma e incremento e as operações lógicas de negação, E-lógico e OU-lógico, além das operações de passagem dos barramentos aos quais estão conectados os registradores. Os registradores têm as seguintes funções:

- *MAR (Memory Address Register*) - responsável pelo endereçamento da memória.
- *MDR* (*Memory Data Register*) - recebe informações lidas da memória (dados ou instruções) e é também o registrador que intermedia a escrita na memória.
- *IR* (*Instruction Register*) - armazena a instrução corrente. É a partir dele que o BC gera as microinstruções a serem executadas pelo BD após a busca da instrução.
- *PC* (*Program Counter*) é o contador de programa, incrementado a cada nova busca de instrução e/ou operando.
- *AC* (*Acumulador*) o único registrador de dados da arquitetura Cleópatra.
- *RS* (*Registrador de Subprograma*) - guarda o endereço de retorno de subrotinas.

A microinstrução gerada pelo BC a cada *clock* é responsável pela determinação de quais registradores serão escritos (3 bits), de quais serão fonte de dados (3 bits), qual será a operação a realizar na ULA (3 bits), determinar se os qualificadores devem ser armazenados ou não (2 bits) e também determinar uma operação de leitura ou escrita na memória (2 bits).

O funcionamento do BC, bastante simplificado, baseia-se na instrução corrente e nos valores instantâneos dos qualificadores (c,n,z). A partir destes valores, determina-se o conjunto de microinstruções, a serem executadas seqüencialmente pelo BD a cada *clock*. A descrição completa desta arquitetura, e detalhes de implementação são encontrados em [3].

3. IMPLEMENTAÇÃO EM VHDL

Foi utilizado o simulador Active-VHDL da ALDEC para validar a descrição VHDL da arquitetura. A descrição compreende 3 entidades principais: BD, BC e CPU, que instancia e conecta o BD com o BC. O texto VHDL aparece abaixo, sem incluir alguns detalhes.

No BD foram instanciados registradores, criados os sinais e barramentos de conexão, um processo para implementar a ULA e dois processos para determinar qualificadores.

Descrição estrutural do BD

```
library IEEE;
use IEEE.Std_Logic_1164.all;
use work.cleo.all;

entity datapath is
        port( uins : in microinstrucao;
              ck, reset: in std_logic;
              ir, address   :  out regsize;
              datamem       :  inout regsize;
              n, z, c       :  out std_logic );
end datapath;

architecture datapath of datapath is

   component reg8clear
        port( clock,reset,ce : in std_logic;   D : in regsize; Q : out regsize );
   end component;

   signal zero, cout, sel1,sel2,rmdr,rir,rpc,rac,rrs,wmar,wmdr,wir,wpc,wac,wrs : std_logic;
   signal um, mdr, reg_ir, pc, ac, rs, outmux, busA, busB, out_ula : regsize;

begin

    ir <= reg_ir;    -- Instrução atual, a ser usada pelo BC

    sel1 <= uins.ce and uins.rw;
    outmux <= out_ula  when sel1='0' else datamem;

    sel2 <= uins.ce and (not uins.rw);
    datamem <= out_ula when sel2='1' else "ZZZZZZZZ";

    -- Registradores de dados: R1=mar, R2=mdr, R3=ir, R4=pc, R5=ac, R6=rs
    R1: reg8clear port map ( clock=>ck, reset=>reset, ce=>wmar, D=>out_ula,  Q=>address);
    R2: reg8clear port map ( clock=>ck, reset=>reset, ce=>wmdr, D=>outmux,  Q=>mdr);
    R3: reg8clear port map ( clock=>ck, reset=>reset, ce=>wir,  D=>out_ula, Q=>reg_ir);
    R4: reg8clear port map ( clock=>ck, reset=>reset, ce=>wpc,  D=>out_ula, Q=>pc);
    R5: reg8clear port map ( clock=>ck, reset=>reset, ce=>wac,  D=>out_ula, Q=>ac);
    R6: reg8clear port map ( clock=>ck, reset=>reset, ce=>wrs,  D=>out_ula, Q=>rs);

    -- Tristates para controlar qual registrador comanda barramentos
    busB <= mdr     when rmdr='1' else "ZZZZZZZZ";
    busB <= reg_ir  when rir='1'  else "ZZZZZZZZ";
    busA <= pc      when rpc='1'  else "ZZZZZZZZ";
    busA <= ac      when rac='1'  else "ZZZZZZZZ";
    busA <= rs      when rrs='1'  else "ZZZZZZZZ";

    -- Codificação de escrita:0 MAR, 1 MDR, 2 IR, 3 PC, 4 AC, 5 RS, 6 PC e MDR, 7 nenhum registrador
    wmar <= '1' when uins.w=d0              else '0';
    wmdr <= '1' when uins.w=d1 or uins.w=d6 else '0';
    wir  <= '1' when uins.w=d2              else '0';
    wpc  <= '1' when uins.w=d3 or uins.w=d6 else '0';
    wac  <= '1' when uins.w=d4              else '0';
    wrs  <= '1' when uins.w=d5              else '0';

    -- Codificação de leitura: 1 MDR, 2 IR, 3 PC, 4 AC, 5 RS, 6 MDR e AC, 7 MDR e PC
    rmdr <= '1' when uins.r=d1 or uins.r=d6 or uins.r=d7 else '0';
    rir  <= '1' when uins.r=d2                           else '0';
    rpc  <= '1' when uins.r=d3 or uins.r=d7              else '0';
    rac  <= '1' when uins.r=d4 or uins.r=d6              else '0';
    rrs  <= '1' when uins.r=d5                           else '0';

    --  Códigos da ULA: 0 soma, 1 inc, 2 not, 4 passaB, 5 ou, 6 and, 7 passaA
    um <= "00000001";
    zero <= '0';

    process (uins,busA,busB,um,zero)  -- descrição da ULA
     begin
           case uins.u is
                when d0 => somaAB( busA, busB, zero, out_ula, cout);
                when d1 => somaAB( busA, um,   zero, out_ula, cout);
                when d2 => out_ula <= not busA;
                when d4 => out_ula <= busB;
                when d5 => out_ula <= busA or busB;
                when d6 => out_ula <= busA and busB;
                when d7 => out_ula <= busA;
                when others => null;
           end case;
     end process;
```

```
process(ck,reset,uins)              -- Qualificador vai-um
      begin
        if (reset='1') then c <= '0';
          else if (uins.c='1') then
                    if(ck'event and ck='0') then  c <= cout;  end if;
               end if;
          end if;
         end process;

    process (ck,reset,uins)            -- Qualificadores zero e negativo
      begin
        if (reset='1') then n <= '0'; z <= '0';
          else if (uins.nz='1') then
                 if(ck'event and ck='0') then n <= out_ula(7); z <= is_zero(out_ula);  end if;
               end if;
          end if;
         end process;
end datapath;
```

O BC é composto por um único processo, que executa primeiro as 3 microinstruções, correspondentes ao ciclo de busca. Uma vez executadas estas, um comando *case* seleciona o conjunto de microinstruções relativas à instrução corrente. Ao término da execução do *case* inicia-se novo ciclo de busca, até o programa alcançar a instrução *halt*, que suspende o processador até novo sinal de *reset*. A função *selop*, no início da descriçao do BC, visa simplificar o código, pois permiter agrupar um conjunto de instruções realizadas na ULA na mesma cláusula do *case*.

O sincronismo é realizado por instruções *wait* após cada microinstrução. Cada microinstrução é executada na borda de subida do sinal de relógio (*clock*). O BD realiza a transferência dos dados entre os registradores e a unidade lógico-aritmética na borda de descida do relógio, garantindo-se assim a correta temporização do sistema.

Descrição comportamental do BC (parcial)

```
library IEEE;
use IEEE.Std_Logic_1164.all;
use work.cleo.all;

entity control is
        port( ir :         in regsize;
              reset,n, z, c, ck: in std_logic;
              inicio: out std_logic;
              uins : out microinstrucao);
end control;

architecture control of control is

  signal cz,c1,jc,jn,jz: std_logic;
  signal zero,um,dois,tres,quatro,cinco,seis,sete : opcode;

 procedure selop (signal ir: in regsize; signal uins : out microinstrucao ) is
  begin
      case ir is
            when ldaim =>  microi (uins,quatro,um,quatro,cz,cz,c1,cz);   -- AC <- MDR
            when andim =>  microi (uins,quatro,seis,seis,cz,cz,c1,cz);   -- AC <- AC and MDR
            when orim =>   microi (uins,quatro,seis,cinco,cz,cz,c1,cz);  -- AC <- AC or MDR
            when addim =>  microi (uins,quatro,seis,zero,cz,cz,c1,c1);   -- AC <- AC + MDR
            when others => null;
         end case;
  end selop;

begin
     cz<='0'; c1<='1';
     zero <= d0;   um <= d1;    dois <= d2; tres <= d3;
     quatro <= d4; cinco <= d5; seis <= d6; sete <= d7;

     jc <= (ir(7) and (not(ir(6))) and (not(ir(5))) and ir(4));
     jn <= (ir(7) and (not(ir(6))) and ir(5) and (not(ir(4))));
     jz <= (ir(7) and (not ir(6))) and ir(5) and ir(4);

     process
      begin

       wait until  ck'event and ck='1';
       -- busca --
       microi (uins,zero,tres,sete,cz,cz,cz,cz);     wait until  ck'event and ck='1';
```

```
      microi (uins,seis,tres,um,c1,c1,cz,cz);       wait until  ck'event and ck='1';
      microi (uins,dois,um,quatro,cz,cz,cz,cz);      wait until  ck'event and ck='1';

      case ir is

       when ldaim | andim | orim | addim => -- LDA, AND, OR, ADD imediato
                microi (uins,zero,tres,sete,cz,cz,cz,cz);   wait until  ck'event and ck='1';
                microi (uins,seis,tres,um,c1,c1,cz,cz);     wait until  ck'event and ck='1';
                selop (ir,uins);

           when ldarel | addrel | andrel | orrel =>  -- LDA, AND, OR, ADD relativo
                microi (uins,zero,tres,sete,cz,cz,cz,cz);   wait until  ck'event and ck='1';
                microi (uins,um,zero,tres,c1,c1,cz,cz);     wait until  ck'event and ck='1';
                microi (uins,zero,sete,zero,cz,cz,c1,c1);   wait until  ck'event and ck='1';
                microi (uins,seis,tres,um,c1,c1,cz,cz);     wait until  ck'event and ck='1';
                selop (ir,uins);

           when stadir =>   -- STA direto
                microi (uins,zero,tres,sete,cz,cz,cz,cz);   wait until  ck'event and ck='1';
                microi (uins,seis,tres,um,c1,c1,cz,cz);     wait until  ck'event and ck='1';
                microi (uins,zero,um,quatro,cz,cz,cz,cz);   wait until  ck'event and ck='1';
                microi (uins,um,quatro,sete,cz,cz,cz,cz);   wait until  ck'event and ck='1';
                microi (uins,sete,um,quatro,c1,cz,cz,cz);   -- PMEM(MAR) <- MDR
           -- ... Várias instruções omitidas

           when nota => -- NOT AC
                microi (uins,quatro,quatro,dois,cz,cz,c1,cz);  -- AC <- NOT(AC)

           when hlt =>
                while reset='0' loop wait until  ck'event and ck='1';
                end loop;

         when others => null ;
       end case;
     end process;
end control;
```

Para a implementação da CPU foram instanciados como componentes o BD e o BC e criados os sinais para as conexões entre estes blocos.

Instruções como "*c<='0' when reset='1' else cout when ck'event and ck='0' and uins.c='1';*" não puderam ser utilizadas, apesar de corretas para o simulador, não são aceitas pela ferramenta de síntese lógica (observar a implementação dos qualificadores no BD). Outros casos de restrições impostas à sintaxe da linguagem foram encontradas, como a atribuição de sinais a *records*.

A descrição completa deste pequeno processador pode ser encontrada em *http://www.inf.pucrs.br/~moraes/FMvhdl.html.*

4. SIMULAÇÃO DA ARQUITETURA CLEÓPATRA

Para validar o código gerado, foi implementado um *test bench* que tem como objetivo avaliar o funcionamento da arquitetura implementada. Este *test bench* é composto por uma instância da CPU e de um componente chamado *my_ram*, que simula a memória externa. O *test_bench* é responsável por gerar os sinais de *clock* e *reset* para a CPU e carregar a memória externa com um programa como o da Figura 2, obtido a partir da leitura de um arquivo texto.

```
00 48 61 ; lda @ 61H                12 44 63 ; lda 63H                  60 05    ; Número de elementos
02 58 62 ; add @ 62H                14 50 01 ; add #1                   61 80    ; ponteiros para o
04 28 63 ; sta @ 63H                16 24 63 ; sta 63H                           ; início dos vetores
06 44 61 ; lda 61H                  18 44 60 ; lda 60H                  62 90
08 50 01 ; add #1                   1A 50 FF ; add #ffH                 63 A0
0A 24 61 ; sta 61H                  1C 24 60 ; sta 60H                  80 77 45 15 10 07    ; Vetor 1
0C 44 62 ; lda 62H                  1E B4 22 ; jz 22H (fim)             90 12 18 45 67 FF    ; Vetor 2
0E 50 01 ; add #1                   20 84 00 ; jump 00 (início)
10 24 62 ; sta 62H                  22 E0    ; halt
```

Figura 2 - Código objeto do test_bench - Mnemônicos indicam a função do código objeto apenas. Primeira coluna hexadecimal contém endereço, uma ou duas seguintes conteúdos deste e da posição subseqüente. O símbolo @ indica modo de endereçamento indireto, # indica endereçamento imediato, ausência de símbolo indica endereçamento direto, modo relativo não usado no exemplo.

Os resultados obtidos com o *test bench* podem ser observados na Figura 3. Nesta simulação estão representadas as três instruções iniciais do programa objeto, que são:

- Carga da primeira posição do Vetor 1 no acumulador (77H);
- Soma deste valor com a primeira posição do Vetor 2 e carga do resultado no acumulador AC (89H);
- Gravação do resultado no endereço de memória contido na posição 63H (A0H) da memória externa.

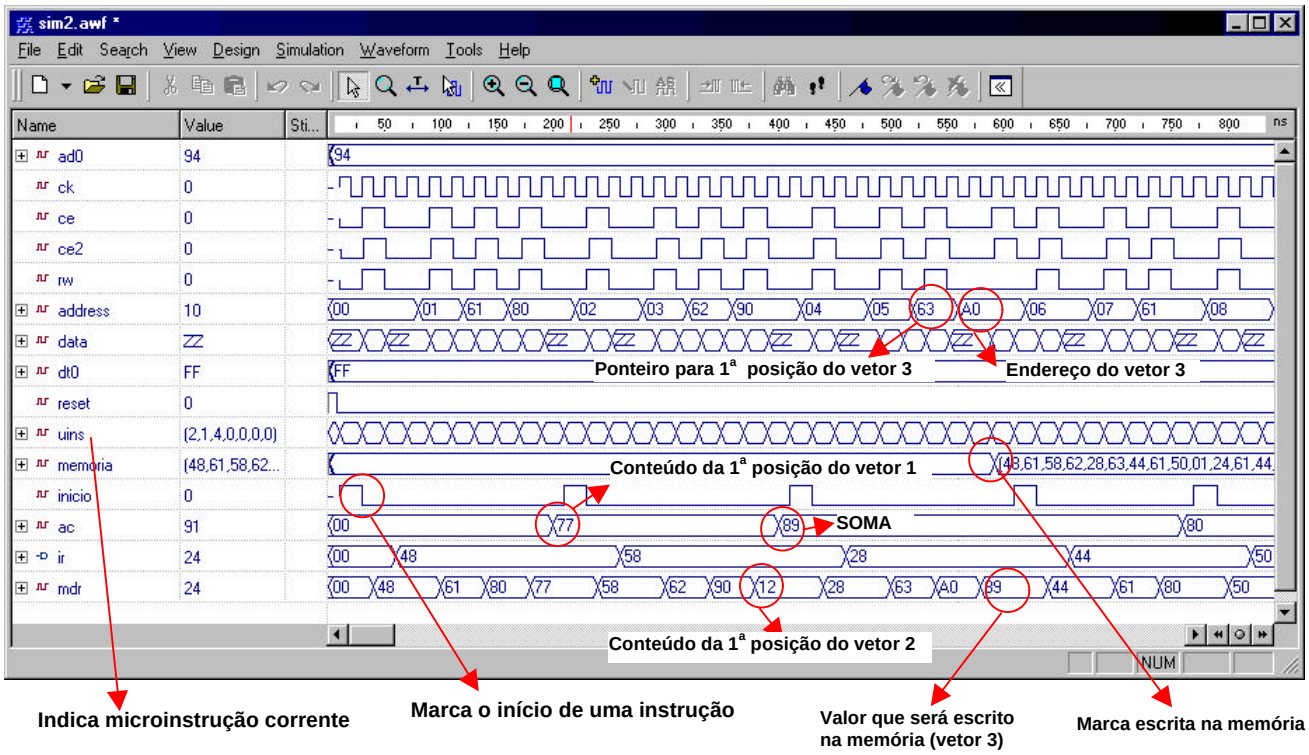

Figura 3 - Simulação de três instruções do processador Cleópatra.

5. A PLATAFORMA LIRMM E IMPLEMENTAÇÃO CLEÓPATRA

A plataforma utilizada para implementar este circuito é a placa LIRMM [4], composta por um processador DSP C40 (Texas), 2 FPGAS Xilinx 4013, 128 KBytes de memória RAM para cada FPGA e uma placa mãe para conectar estes dispositivos ao barramento hospedeiro.

A comunicação entre o DSP e FPGAs é feita por 4 canais de 8 bits, 2 por FPGA . O envio de dados ao FPGA é feito por palavras de 32 bits, sendo o FPGA responsável por receber os 4 bytes e controlar a comunicação. O envio de dados ao DSP também é feito em pacotes de 32 bits. O DSP interage com o computador hospedeiro.

As memórias conectadas aos FPGAs não têm comunicação com o barramento do DSP. Ou seja, o controle de leitura/escrita nas memórias também é feito internamente. no FPGA.

Um dos objetivos do presente estudo de caso é implementar um núcleo processador (*core processor*), visando estabelecer um método para futuros trabalhos em projeto integrado de software e hardware (hardware/software codesign). Para implementar um núcleo processador na plataforma LIRMM, como de resto em qualquer plataforma de prototipação, é necessário estabelecer os protocolos de comunicação entre o computador hospedeiro, onde hardware e software do núcleo são desenvolvidos, e o núcleo. No caso da plataforma LIRMM, esta interação é feita indiretamente, através do processador DSP, que sob comando do hospedeiro configura o(s) FPGA(s), descarrega código objeto e dados de entrada para o núcleo, libera a execução do programa e recupera os dados de saída.

O procedimento para utilizar o processador compreende 3 etapas:

1. Envio do programa em código objeto Cleópatra do DSP para o FPGA. Nesta etapa, Cleópatra deve estar inativo, e um bloco do FPGA recebe os dados provenientes do DSP e os armazena na memória externa.
2. Uma vez terminado o envio do programa, o DSP sinaliza ao FPGA que deve ser iniciado o processamento. Neste momento Cleópatra inicia a execução do programa enviado no passo anterior.
3. O DSP solicita dados. Se o processamento terminou, outro bloco envia os dados do FPGA para o DSP.

A figura 4 ilustra o circuito implementado no FPGA.

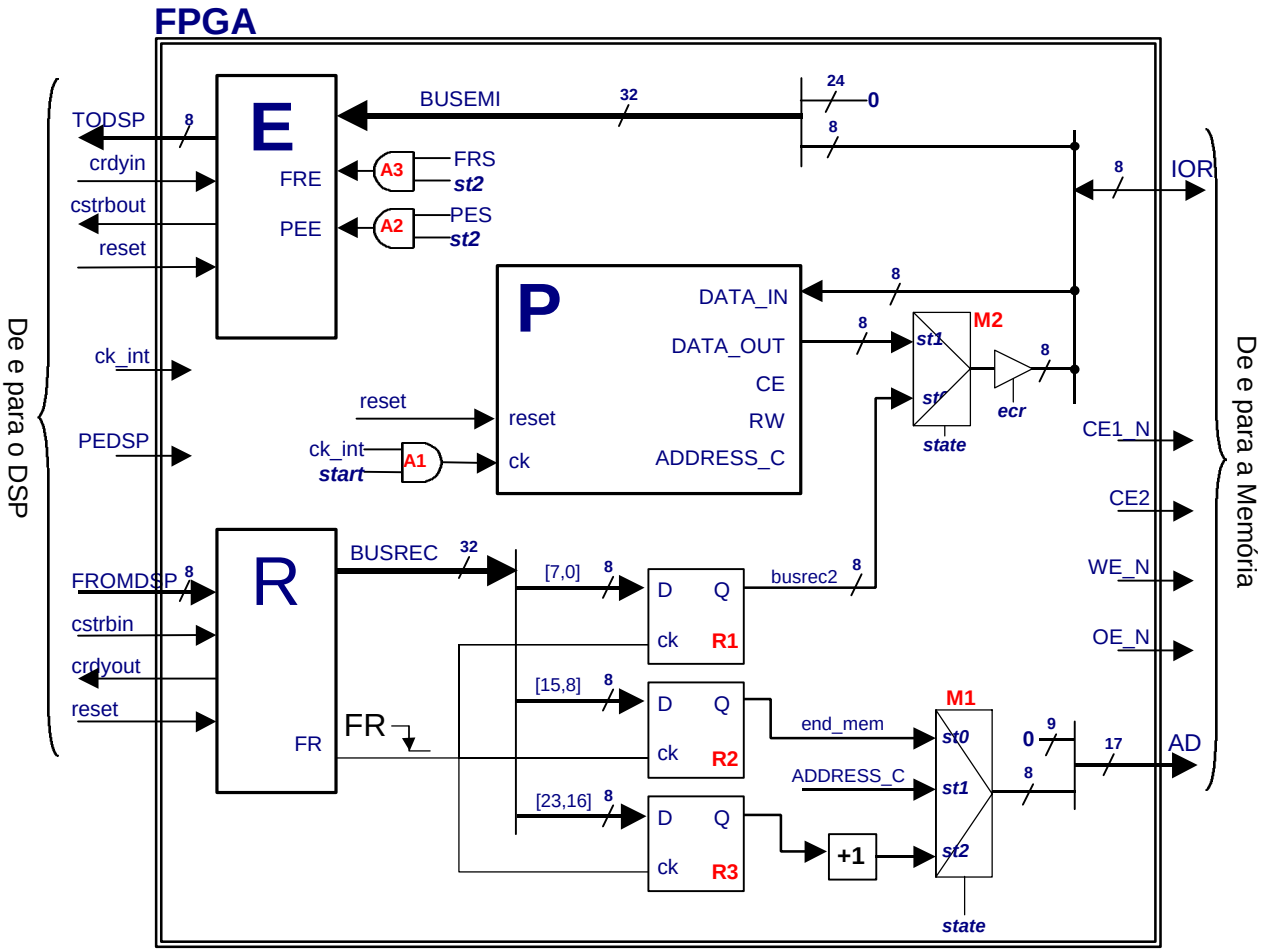

Figura 4 – Implementação do processador (*core processor*) Cleópatra e circuitos de controle para recebimento do programa, sua execução e envio dos dados no FPGA.

O bloco ***R*** (recepção) tem por função receber os dados do DSP. O barramento *fromdsp* contém os 8 bits provenientes do DSP. O sinal *cstrbin* sinaliza a este bloco que há dados válidos. Após recebido o quarto byte, haverá um borda de descida no sinal ***FR***. Esta borda de descida indica que há uma palavra válida de 32 bits no barramento *BUSREC*.

O FPGA é inicializado por um pulso no pino *reset*. Neste momento, existe uma máquina de estados de controle ***F*** (não mostrada) que estará no estado *st0*. A primeira palavra enviada pelo DSP conterá o endereço da área de dados do programa no mapa de memória da Cleópatra, onde esta deverá armazenar os resultados da execução. Este endereço será armazenado em *R3*. Os registradores *R1* e *R2* armazenam o valor a ser gravado e o endereço, respectivamente. Desta forma, a memória estará sendo endereça por *R2* (mux *M1*) e os dados

enviados estarão sendo enviados ao barramento de dados da memória (mux *M2*). O sinal *ecr* estará então ativo, controlado por ***F***.

Ao primeiro pulso PEDSP, ***F*** passa para o estado *st1*, e libera-se o clock da Cleópatra (porta lógica *A1*). A partir deste momento, o endereçamento da memória será feito pelo processador (mux *M1*, entrada *address_c*). Observa-se que o resultado do processamento deve ser armazenado em uma área de dados, com o endereçamento definido no estado *st0*.

Ao próximo pulso PEDSP, ***F*** passa para o estado *st2*, o endereço de memória é gerado inicialmente por *R3*, e o bloco entre *R3* e *M1*, (um registrador incrementador) produz endereços consecutivos para acesso à área de dados. Estas palavras são enviadas ao DSP através da máquina de emissão ***E***.

O bloco ***E*** (emissão) tem por função enviar dados ao DSP. O barramento *BUSEMI* contém 32 bits provenientes do FPGA. O sinal *PEE*, gerado internamente no FPGA, inicializa este bloco. A cada sinal *crdyin* do DSP, é colocado 1 byte no barramento *TODSP*.

Este circuito foi validado por simulação, antes de sua implementação no FPGA.

O programa escrito em linguagem C, executado no DSP, e que controla o circuito implementado no FPGA é mostrado abaixo. O protocolo de comunicação utiliza duas portas do DSP para envio e recepção dos dados (envio e recepção em 32 bits) e sinas de controle mapeados no barramento global de endereços do DSP: INIT1, para reset; PE1, para mudança de estado no DSP; RAZ para fazer com que estes sinais voltem ao estado original (zero).

Descrição em C do controle do circuito implementado no FPGA (lado DSP, parcial)

```
void main()
{

     *INIT1=0;                          /* Envia reset para FPGA – FPGA vai para st0 */
     *RAZ=0;

     *comm_port1_output=0x00f00000;     /* Envia o endereço da área de dados de saída */

     in=fopen("prog","rt");             /* Lê o programa e envia-o para o FPGA */
     do{
       fscanf(in,"%c",&indata);
       *comm_port1_output=indata;
     } while(!feof(in));
     fclose(in);

    *PE1=0;                             /* Envia primeiro PEDSP – FPGA vai para st1 */
    *RAZ=0;

    wait(300);                          /* Espera um pouco enquanto Cleópatra executa */

    for(j=0; j<10; j++)
     {
       *PE1=0;                          /* Envia segundo PEDSP – FPGA vai para st2 */
       *RAZ=0;

       b =  *comm_port4_input;          /* Lê dados do FPGA */
     }
}
```

5.1. Implementação do Núcleo Cleópatra e Comunicação com Mundo Externo

Este processo envolve 4 passos:

1. Descrição do circuito em linguagem VHDL e simulação funcional (Active VHDL);
2. Síntese lógica (FPGA Express);
3. Síntese física, envolvendo mapeamento tecnológico, posicionamento, roteamento, análise de *timing* e geração do arquivo binário para *download* (XACT 1.5);
4. Descrição do programa em linguagem C para o controle do circuito implementado no FPGA e compilação do mesmo.

O resultado dos passos 1 a 4 descritos acima resultam em dois arquivos principais: o código objeto a ser executado no DSP e o arquivo binário a ser descarregado para configurar o FPGA. Para validar o sistema, escreveu-se diversos programas para este processador, e todos foram executados com sucesso no FPGA.

Apresenta-se abaixo alguns dados numéricos da implementação em termos de blocos lógicos da família Xilinx 4000 (CLBs) utilizados.

```
Number of CLBs:           270 out of   576   46%
      CLB Flip Flops:     192
      4 input LUTs:       390 (2 used as route-throughs)
      3 input LUTs:       153 (65 used as route-throughs)
   Number of bonded IOBs: 52 out of   160   32%
      IOB Flops:          0
      IOB Latches:        0
   Number of global buffers: 1 out of     8   12%
   Number of primary CLKs:   1 out of     4   25%
   Number of secondary CLKs: 3 out of     4   75%
   Number of TBUFs:          40 out of  1248    3%
   Number of RPM macros:     1
Total equivalent gate count for design: 4045
```

Como indicado, foi utilizada apenas 46% da área de um dos FPGAs XC4013EPQ208-4. A utilização por módulos foi, aproximadamente: 70 CLBs para os módulos de emissão e recepção, 20 CLBs para o controle e 180 CLBs para o processador Considerando-se que há dois FPGAs na plataforma, pode-se implementar pelo menos 4 processadores em paralelo.

6. CONCLUSÃO E TRABALHOS FUTUROS

Este trabalho mostrou a implementação de um núcleo processador simples sobre uma plataforma de prototipação de sistemas embarcados baseada em FPGAs. O trabalho foi totalmente desenvolvido em VHDL, a partir da especificação Cleópatra [3]. Além da aplicação natural em atividades de ensino de graduação, foi definido um método razoavelmente genérico de integração de núcleos processadores para prototipação. Isto ocorre porque a placa LIRMM é um representante típico em termos de estrutura de blocos destas plataformas, formadas por processador(es) comercial(ais), memória, dispositivo(s) reconfigurável(eis) e conexão a um computador hospedeiro.

Os trabalhos em andamento a este relacionados são: o desenvolvimento de um processador embarcado mais complexo e moderno do tipo RISC; a implementação de múltiplos processadores Cleópatra para execução em paralelo, usando a capacidade ociosa atual dos FPGAs; a avaliação de critérios de particionamento entre software e hardware na implementação de processadores embarcados, com o intuito de garantir o melhor compromisso entre custo e desempenho de sistemas digitais voltados para aplicações específicas, e a busca de aplicações específicas para validar os critérios em desenvolvimento, tais como tratamento de imagens em tempo real.

7. BIBLIOGRAFIA

[1] Aptix Corporation. Block-Based Prototyping Methodology. White Paper disponível a partir da página http://www.aptix.com/News/BBP%20White%20Paper.html.

[2] Calazans, N. *Projeto Lógico Automatizado de Sistemas Digitais Seqüenciais*, capítulo 6: Análise Seqüencial. Imprinta Gráfica e Editora Ltda, Rio de janeiro, RJ, 1998. pp 259-289. Disponível a partir de ftp://ftp.inf.pucrs.br/pub/calazans/pubs/prjlog/v1.0/Cap6.ps.

[3] Moraes, F & Calazans, N. Arquitetura Cleópatra (Versão 1.8). *Notas de Aula*, 1998. 26 páginas. Disponível a partir de ftp://ftp.inf.pucrs.br/pub/calazans/undergrad/orgcomp/cleopatra/o_cleo18.doc.

[4] Maillet-Contoz, L., Pillement, S. & Sallantin, J. A unified Workbench for Designing Hardware/Software Systems, In: *International Workshop on Logic and Architecture Synthesis* 96, Grenoble, 16-18 Dec. 1996.