IMPLEMENTAÇÃO EFICIENTE DE UMA ARQUITETURA LOAD/STORE EM VHDL

F.G.Moraes, N.L.V.Calazans, E.H.Ferreira, D.C. Liedke

Faculdade de Informática – PUCRS
Av. Ipiranga, 6681 – Porto Alegre – CEP: 90619-900
{moraes, calazans, ferreira, liedke}@inf.pucrs.br

Abstract: This work presents the specification, design, implementation and test of a simple, 16-bit load/store instruction set architecture in the VHDL hardware description language. During the description of the whole process the paper establishes a discussion about how to incrementally tune the initial VHDL description to obtain an efficient hardware implementation. The merits of several VHDL constructs with regard to their cost in hardware and their code compactness are compared. As a result, some guidelines are developed to employ during the design in order to evaluate tradeoffs between the size of the implementation against the ease of description.

Keywords: computer organization and architecture, VHDL, FPGA, prototyping.

1 Introdução

Este trabalho apresenta a implementação de um núcleo processador simples (core processor) com arquitetura Load/Store. O projeto foi descrito utilizando unicamente a linguagem VHDL. Descreve-se também sua integração a um módulo controlador. Este por sua vez disponibiliza a comunicação com a memória externa e outros componentes da plataforma de prototipação XS-40/Xstend da empresa Xess [1]. Durante a descrição do processo de projeto, implementação e teste, discute-se o efeito do uso de determinadas estruturas VHDL na implementação, comparando possíveis alternativas equivalentes do ponto de vista funcional.

Inicialmente, esta arquitetura foi proposta como trabalho final da disciplina de Organização de Computadores na Faculdade de Informática da PUCRS (terceiro semestre) [2]. Os alunos devem implementar a especificação em VHDL simulável, sem maior preocupação com a implementação física.

A partir da especificação da arquitetura, foi realizada a implementação da mesma em uma plataforma de prototipação [1]. As motivações para o desenvolvimento deste trabalho são: (*i*) dispor de um núcleo processador reconfigurável que possa ser facilmente modificado de acordo com a aplicação do usuário (ou seja, um ASIP – application specific instruction-set processor), trabalho este em andamento; (*ii*) utilizar este processador como o módulo executor de software em projetos integrados de sistemas computacionais compostos por hardware-software para aplicações embarcadas [3]; (*iii*) capacitar a equipe do GAPH à implementação de sistemas digitais completos em um único FPGA (systems-on-a-chip - SOC). Atualmente, o grupo de pesquisa dos autores deste artigo dispõe de diversas plataformas de prototipação de hardware (com capacidade para implementar entre 5.000 e 300.000 portas lógicas equivalentes cada [4]), o que dará suporte a trabalhos subseqüentes. Este trabalho será desenvolvido sobre uma plataforma contendo aproximadamente 10.000 portas lógicas equivalentes.

A Seção 2 apresenta a descrição da arquitetura, sendo sua implementação em VHDL sintetizável apresentada na Seção 3. A Seção 4 descreve o ambiente de prototipação empregado e a descrição do circuito de interface entre o processador e a memória externa. A Seção 5 apresenta resultados relativos à implementação no FPGA e finalmente, a Seção 6 detalha conclusões e trabalhos futuros.

2 Descrição da arquitetura

O processador implementado, denominado ***R3***, é uma organização Von Neumann (memória de dados/instruções unificada), *Load/Store*, com CPI igual a 2 [5]. Esta arquitetura é praticamente uma máquina RISC, faltando contudo algumas características que existem em qualquer máquina RISC, tal como *pipeline*.

A arquitetura contém, no bloco de dados, 15 registradores de uso geral; registradores para: armazenamento da instrução corrente (IR), endereço da próxima instrução a executar (PC) e controle de endereços de retorno de subrotina (SP); uma ULA (Unidade Lógica e Aritmética) com 16 operações e 4 qualificadores de estado (Z-zero, N- negativo, C- vai-um , V- transbordo).

O bloco de controle é uma máquina de estados responsável por gerar os comandos para o bloco de dados a partir da informação da instrução corrente contida no IR e dos qualificadores de estado, usados para tomadas de decisão em instruções de desvio condicional.

A comunicação entre o processador e o meio externo compreende: sinais de temporização, *clock* e *reset*; sinais de acesso à memória, *ce*, *rw*, *address*, *datain* e *dataout*; e sinal que indica término de execução, *e_halt*.

No ciclo de busca são realizadas duas operações em um único ciclo de relógio: (*i*) leitura da posição de memória apontada por PC e (*ii*) auto-incremento do PC.

As instruções realizadas durante o ciclo de execução compreendem:

- Operações **binárias**, envolvendo 2 registradores fonte (Rs1 e Rs2) e um registrador destino (Rt). Realiza-se a função Rt ← Rs1 *opcode* Rs2. *Opcode* assume as funções soma, subtração, e lógico, ou lógico e ou-exclusivo.
- Operações **unárias**, envolvendo um registrador fonte (Rs1) e um registrador destino (Rt). Realiza-se a função Rt ← *opcode* Rs1. *Opcode* assume as funções: cópia de

valores entre registradores, deslocamentos, incremento, decremento, inversão e complemento de dois.

- Operações de **salto** com deslocamento curto, salto relativo a registrador e salto absoluto. Os saltos realizados são: *JN*, *JZ*, *JC*, *JV* (negativo, zero, vai-um e transbordo aritmético), *JMP* (incondicional) e *JSR* (incondicional para subrotina).
- Operações de **carga** de dados da memória, modo imediato (*load)*. Realiza-se as funções: *LDL* (carga da parte baixa do registrador) e *LDH* (carga da parte alta do registrador).
- Operações de **carga** de dados, modo indireto (*load)*. Um registrador *Rtarget* recebe o conteúdo de memória apontado pelo registrador *Rsource*.
- Operações de **escrita** de dados, modo indireto (*store)*. A posição de memória apontada por um *Rtarget* recebe o conteúdo de *Rsouce*.
- Operações **miscelâneas**: *LDSP* (carga do conteúdo de um registrador no registrador SP), *RTS* (retorno de subrotina), *NOP* e *HLT* (suspensão da execução de instruções pelo processador).

Os modos de endereçamento implementados são quatro: a registrador, registrador indireto, imediato, e relativo ao PC, não havendo modo direto de endereçamento.

Um simulador foi desenvolvido para testar os programas escritos em linguagem *de montagem.* O ambiente de desenvolvimento construído contém, além do simulador, um montador e um editor de texto para linguagem de montagem integrados. Além disto, o ambiente permite gerar um arquivo com o código objeto da aplicação, no formato adequado para carga na memória da plataforma de prototipação.

3 Implementação da arquitetura

A implementação da arquitetura em VHDL compreende a construção dos dois blocos principais: o bloco de controle e o bloco de dados. Estes blocos são unidos hierarquicamente em um terceiro bloco, constituindo assim o processador.

A implementação do bloco de dados foi feita em VHDL *estrutural* (registradores, multiplexadores, decodificadores, ULA). A implementação do bloco de controle foi feita em VHDL *comportamental* (utilização de comandos como *case* e *if*).

3.1 Bloco de Controle (BC)

O BC tem por função gerar os comandos para a busca da instrução (*fetch*) e depois enviar os comandos ao bloco de dados para que a instrução seja executada. Sua implementação é uma grande estrutura de seleção que avalia o registrador de instrução corrente e gera as microinstruções necessárias para que o bloco de dados realize as operações sobre dados. O BC é formado por dois processos. O primeiro é responsável por decodificar a instrução *halt* e gerar o sinal externo *e_halt,* o qual indica que o processador suspendeu a execução de instruções.

O segundo processo determina o funcionamento normal do BC. Após o *reset*, na primeira borda de subida do clock é lançada a microinstrução responsável pela busca de instrução. Na segunda borda do *clock*, a instrução que foi buscada é avaliada por uma grande estrutura do tipo *case* em VHDL, selecionando-se assim a microinstrução correspondente à instrução especificada. Uma vez terminada a execução, inicia-se novo ciclo de busca.

Na Figura 1 é mostrado o código VHDL que implementa as microinstruções relativas a duas instruções. A primeira é um salto condicional, JN. Observar que se o salto é tomado gera-se a microinstrução para alterar o PC, caso contrário, gera-se uma microinstrução relativa à instrução NOP para manter-se CPI igual a 2. Depois é mostrada a microinstrução para o XOR.

```
process(reset, ck)   -- processo responsável por gerar o sinal halt
 begin
    if reset='1' then
              e_halt <= '0';
           elsif ck'event and ck='1' then
              if ir=x"6230" then e_halt <= '1'; end if;
    end if;
end process;

process    -- processo responsável por gerar as microinstruções de busca e execução de cada instrução
 begin

  wait until ck'event and ck='1';     -- espera borda ascendente de clock para realizar a busca

  uins<=(cz,c1,cz,passaB,cz,cz,c1,c1,cz,cz,cz,c1,c1,cz,cz,cz,cz,cz,cz); -- busca da instrução

  wait until ck'event and ck='1';     -- espera borda ascendente de clock para realizar a execução

  case ir(15 downto 12) is            -- decodifica instrução e gera microinstrução para executá-la
    when d0 => if n=c1 then
      uins<=(cz,cz,c1,ext_sinal,cz,cz,cz,cz,cz,cz,cz,cz,c1,cz,cz,cz,cz,cz,cz);  --JN desloc
     else
      uins <= (cz,cz,cz,soma,cz,cz,cz,cz,cz,cz,cz,cz,cz,cz,cz,cz,cz,cz,cz);      -- NOP
      end if;
    ...
    when d14=> uins<=(cz,cz,cz,ouX,c1,c1,cz,cz,c1,cz,cz,cz,cz,cz,cz,cz,cz,cz,cz);    -- XOR
   ...
  end case;
end process;
```

Figura 1 - Código parcial do bloco de controle – cz é o bit constante 0 e c1 é o bit constante 1.

Esta solução tem as vantagens de ser simples de implementar e de fácil modificação (supressão ou inclusão de novas instruções). Uma segunda forma de implementação consistiu em escrever uma equação booleana para cada sinal da microinstrução (parte de controle *hardwired*). Esta solução pode parecer mais otimizada, pois suprime-se a grande estrutura de seleção. Comparações efetuadas sobre ambas implementações, no nível físico, mostraram que ambas implementações resultam praticamente no mesmo resultado (quanto a quantidade de hardware e atraso). Logo, a solução baseada em decodificação foi a adotada para este processador.

3.2 Bloco de Dados (BD)

A Figura 2 ilustra a estrutura do BD do processador. Há quatro blocos principais: (*i*) banco de registradores (15 de propósito geral e 1 para a instrução corrente), (*ii*) PC (contador de programa), (*iii*) SP (ponteiro para topo da pilha) e (*iv*) ULA (unidade lógico aritmético, sendo “F” os 4 qualificadores).

Banco de Registradores.

O banco de registradores, também mostrado na Figura 2, tem uma porta de acesso para escrita, proveniente do multiplexador *mux1*, e duas portas de leitura, respectivamente para os barramentos *BUS_A* e *BUS_B*. A seleção do registrador a ser alterado ou dos registradores a serem lidos é especificada implicitamente pela instrução corrente, a qual é armazenada no registrador 15.

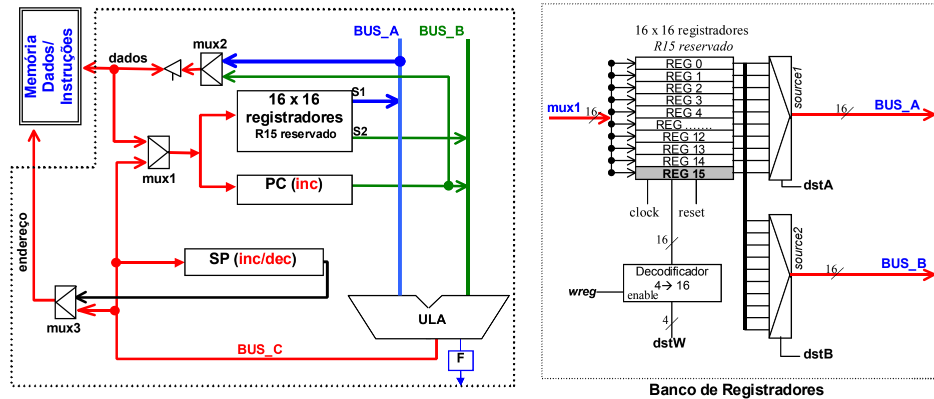

Figura 2 - Bloco de dados do processador R3 e banco de registradores.

Um dos desafios foi implementar eficientemente, em termos de gasto de hardware, o processo de duas leituras simultâneas (necessárias a cada operação binária, por exemplo). A Figura 3 apresenta as 3 soluções implementadas para uma dada decodificação.

```
(a) decodificador 16X1
process(dstA)
   begin
    case dstA (3 downto 0) is
      when d0 =>     busA <=  reg0;
      when d1 =>     busA <=  reg1;
      ...
      when d14=>     busA <=  reg14;
      when others => busA <=  reg15;
   end case;
end process;

(b) buffers tristate
busA <= reg0  when dstA=d0 else (others=>'Z');
busA <= reg1  when dstA=d1 else (others=>'Z');
....
busA <= reg13 when dstA=d13 else (others=>'Z');
busA <= reg14 when dstA=d14 else (others=>'Z');

(c) decodificador 2x1 com a buffers tristate
busA <= reg0 when dstA=d0 else
        reg1 when dstA=d1 else (others=>'Z');

busA <= reg2 when dstA=d13 else
        reg3 when dstA=d14 else (others=>'Z');
```

Figura 3 - Soluções para implementar o banco de registradores.

A primeira solução é baseada em decodificação, simples de ser implementada. Porém esta solução implica em dois decodificadores 16x1 com largura de palavra igual a 16, tornando o hardware demasiadamente grande para ser implementado no FPGA Xilinx XC4010 da plataforma [6] (utilização de CLBs em torno de 450 das 400 disponíveis).

A segunda solução requer menos hardware, porém necessita 512 buffers tristate (16 bits *x* 16 registradores *x* 2 acessos). Apesar de haverem 880 buffers tristate no XC4010, há vários outros módulos que utilizam estes recursos, e a ferramenta de roteamento não conseguiu gerar uma solução completa.

A terceira solução é um misto das duas anteriores, baseando-se no uso de decodificação 2x1 seguido de barramento tri-state. Esta solução resultou em um

hardware sintetizável, com 388 CLBs (dos 400 CLBs disponíveis).

Para justificar a implementação estrutural, implementou-se o mesmo banco de registradores utilizando-se um *array* de 16 palavras de 16 bits. Esta implementação tem código VHDL muito mais compacto, porém no momento da síntese obteve-se resultados com ocupação de 700 (setecentos) CLBs.

Este exemplo mostra como uma implementação cuidadosa no nível estrutural permite economia de área (número de CLBs), e eventualmente viabiliza a implementação da arquitetura com limitados recursos de hardware.

Registradores SP e PC.

O registrador SP tem a característica de ser auto-incrementável e auto-decrementável. Sua implementação está apresentada na Figura 4. A operação de incremento/decremento é combinacional e assíncrona. Se há habilitação de escrita (**wSP**=1) é feita ou a entrada do valor via *BUS_C* (no caso da instrução LDSP), ou do valor atual incrementado ou decrementado. O registrador **PC** tem implementação semelhante ao **SP**, não havendo entretanto auto-decremento (o **PC** pode apenas ser incrementado).

```
architecture reg16sp of reg16sp is
  signal cout, coutant, c0: std_logic;
  signal SSP, um, incdecsp: reg16;
begin
  incdecsp <= ssp+1 when (incsp = '1') else ssp-1;

   process (clock, reset)
     begin
      if RESET = '1' then
        ssp <= (others => '0');
      elsif clock'event and clock='0' then
       if WSP = '1' then
         if spin = '1' then SSP <= D else SSP <= incdecsp; end if;
        end if;
      end if;
   end process;

   Q <= ssp when spout='0' else incdecsp;   -- saída do registrador SP
end reg16sp;
```

Figura 4 – Implementação VHDL do registrador SP.

Unidade Lógico-Aritmética (ULA).

O desafio para implementar a ULA foi também reduzir o número de blocos lógicos configuráveis (CLBs). A implementação VHDL original consistia de uma estrutura de seleção do tipo *case*, que em função da microoperação de controle especificada, realizava uma determinada operação entre os barramentos A e B. O problema desta implementação é que, durante a síntese automatizada, ocorre a replicação da estrutura de soma tantas vezes quanto a mesma seja escrita, ocasionando um grande gasto em CLBs. A solução adotada foi primeiro selecionar os operandos, e depois somá-los. Desta forma, instrui-se o sintetizador a utilizar apenas uma única estrutura de soma de 16 bits, tornando factível a implementação em hardware de pequeno porte.

A Figura 5 ilustra a implementação VHDL da ULA, segundo a solução de minimizar o número de elementos de soma. A microoperação enviada à ULA está em *uins.ula*. Este comando possui 3 funções: selecionar os operandos *input_adder1* e *input_adder2* (que são entradas para um somador de 16 bits - *somaAB*) e selecionar qual o resultado de operação a ser colocado na saída da ULA - *out_ula*.

```
out_ula <=  (BusA or busB)                       when uins.ula=ou       else
            (BusA and busB)                      when uins.ula=e        else
            (BusA xor busB)                      when uins.ula=ouX      else
             busB(15 downto 8) & BusA(7 downto 0) when uins.ula=cte_low  else
             BusA(7 downto 0)  & busB(7 downto 0) when uins.ula=cte_high else
             BusA(14 downto 0) & cin              when uins.ula=sl0 or uins.ula=sl1 else
             cin & BusA(15 downto 1)              when uins.ula=sr0 or uins.ula=sr1 else
             output_adder;

 input_adder1 <=   (not busA)    when uins.ula=negA or uins.ula=notA else
    (others=>'0') when uins.ula=passaB else
    BusA(11) & BusA(11) & BusA(11) & BusA(11) & BusA(11 downto 0) when uins.ula=ext_sinal
    else busA;
 input_adder2 <= (not busB)    when uins.ula=AsubB else
   (others=>'0') when uins.ula=negA or uins.ula=notA or uins.ula=passaA or uins.ula=IncA else
   (others=>'1') when uins.ula=decA else
   busB;
 cin<='1' when uins.ula=AsubB or uins.ula=negA or uins.ula=incA or uins.ula=sl1  or uins.ula=sr1
         else '0';

 somaAB( input_adder1, input_adder2, cin, output_adder, coutant, cout);
 -- em função de cout, coutant,output_adder gerar os qualificadores
```

Figura 5 – Implementação da ULA.

3.3 Processador e validação inicial

Uma vez implementados os módulos principais do BD, esses são unidos em um par *entity-architecture,* que instancia os módulos, unindo-os via multiplexadores (em VHDL, na forma de atribuições condicionais) e barramentos. A validação inicial foi realizada via simulação funcional pelo uso do simulador Active-VHDL [7].

O *test_bench* para a simulação desempenha as seguintes funções: (*i*) instancia o processador; (*ii*) gera os sinais de controle e de temporização (*clock* e *reset*); (*iii*) simula o botão externo *spare*, o qual comanda a máquina de estados de controle (ver Seção 4); (*iv*) implementa os processos de interface com a memória para leitura e escrita; (*v*) realiza a leitura de arquivo texto contendo o código objeto da aplicação no momento do *reset* (código objeto gerado a partir do ambiente de desenvolvimento).

4 Controlador do processador

A plataforma de prototipação [1] consiste de: (*i*) uma placa XS40 que abriga o FPGA Xilinx XC4010E, um microcontrolador 80C51 da Intel (não usado), RAM de 32Kbytes, clock externo de 12MHZ; (*ii*) uma placa XStend contendo, entre outros recursos de entrada e saída, dip-switches, leds, displays 7-segmentos e comunicação via porta paralela com um computador hospedeiro do tipo PC.

O FPGA comunica-se com a memória através de dois barramentos, um para endereçamento (15 bits) e outro bidirecional para dados (8 bits), além dos sinais de controle *ce, oe* e *we*. Estes mesmos barramentos são utilizados para envio de valores aos leds e displays da plataforma para mostrar resultados, usando multiplexação.

O módulo controlador, ilustrado na Figura 6, é responsável pela comunicação entre o núcleo processador R3 e os componentes que fazem a interface com os dispositivos da plataforma de prototipação, como memória, chaves e *displays*.

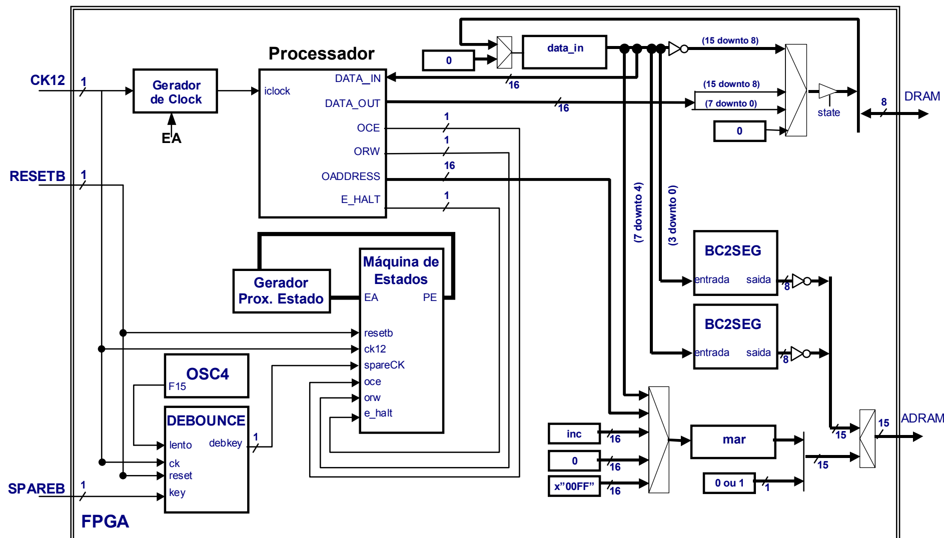

Figura 6 - Diagrama em blocos do controlador.

Os módulos que compõe o controlador são:

- **CPU**: processador R3, responsável pelo processamento do programa inicialmente colocado em memória no momento do download do projeto;
- **Debounce**: circuito que *(i)* realiza a filtragem na captura de eventos (pressionamento) das teclas *reset* e *spare* da plataforma de prototipação; *(ii)* realiza a sincronização do botão *spare* (sinal assíncrono, com ruído) ao *clock*, gerando-se um único pulso após a primeira borda de descida em *spare*;
- **Bcd2Seg**: circuito que realiza a conversão de valores binários para o correspondente em 7-segmentos para posteriormente ser apresentado nos displays da plataforma;
- **Máquina de estados:** para controle de escrita e leitura na memória RAM, e controle das etapas de processamento;
- **Registradores intermediários:** (mar, data_in) para armazenamento de conteúdos e endereços para acesso à memória.
- **Gerador de clock para o processador**: gera o *clock* para o processador. A largura dos pulsos, bem como seu ciclo de serviço são variáveis, conforme é mostrado na Figura 7, sendo este sinal de *clock* congelado após ativação de *e_halt* pela R3.

O controlador funciona conforme o estado da máquina de estados. Após gerado um pulso no sinal externo *RESETB,* a máquina entra no estado inicial **S0**, permanecendo neste estado até que se pressione o botão *SPAREB*. No estado seguinte, **S1**, o processador é inicializado e inicia-se a fase de processamento. Os estados seguintes correspondem às fases de escrita e leitura na memória. Identificada uma instrução *halt* no programa, o processador ativa o sinal *e_halt*, sinalizando ao controlador o término do processamento. O controlador passa então para o **estado S2**, que suspende o clock do processador. Os estados seguintes, **S3**, **S4** e **S5**, exibem a área de dados nos *displays* e *leds* da plataforma de prototipação, permitindo assim verificar se o processamento foi correto ou não.

A Figura 7 ilustra a execução de 3 instruções neste processador, considerando o ambiente de prototipação externa. **Observar na simulação:**

- A microinstrução (*uins*) é gerada na borda de subida de ck_R3;
- Ao final da busca (primeira borda de descida em ck_R3) R15 recebe a instrução corrente;
- Ao final da execução (segunda borda de descida em ck_R3) o registrador destino é alterado;
- A fase de busca consome **sempre** 3 ciclos de *clock* de referência (12 MHz), pois são necessários 2 acessos à memória e uma atualização de endereço;
- A fase de execução pode consumir 1 ciclo, caso a operação não envolva acesso à memória (como *soma*) ou 3 ciclos, como no caso da instrução *store*;
- A instrução ***store*** faz: PMEM(R4)← R3, ou seja, o conteúdo de memória apontado por R4 recebe o conteúdo de R3. O conteúdo de *reg3* é transferido para *data_out*. O conteúdo de *reg4* é D1H, é deslocado para a esquerda tornando-se 1A2H e 1A3H, conforme *adram*.

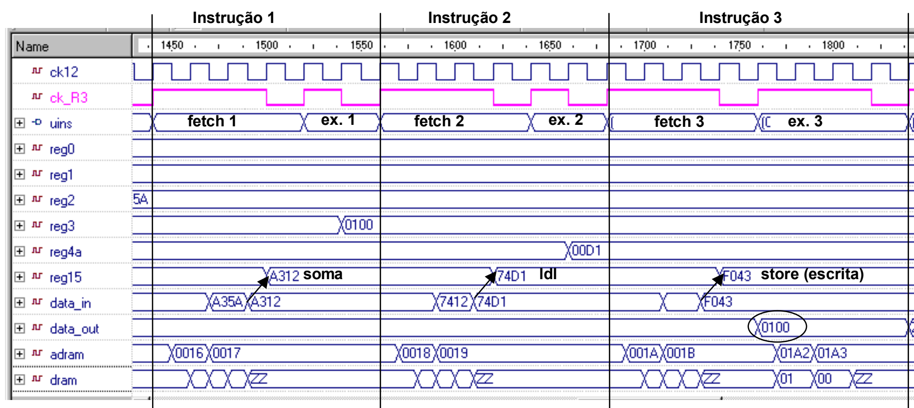

Figura 7 - Simulação do circuito completo, estado de funcionamento S1.

Uma vez o processador e o controlador validados funcionalmente (utilizou-se por exemplo, *bubble* e *quicksort*), a fase posterior do trabalho consistiu na sua implementação efetiva na plataforma de prototipação. É importante ressaltar que o desenvolvimento do código VHDL foi feito acompanhado da síntese física, como comentado na parte relativa ao bloco de dados. A cada novo módulo inserido no código era verificado seu impacto na área final do circuito. Um aprendizado importante é que código VHDL simulável não significa necessariamente código sintetizável.

5 Resultados

Como ferramenta de síntese foi utilizado o pacote Foundation 1.5i e 2.1i [8]. Este pacote compreende ferramentas de síntese lógica (FPGA Express - Synopsys [9]), síntese física, análise de *timing*, geração de arquivo binário para download, entre outras ferramentas.

A Figura 8 mostra o relatório (parcial) de síntese física para uma versão da arquitetura com 12 registradores, ao invés dos 16. A razão para tal corte foi a ocupação

de blocos lógicos. Como pode ser observado o tempo de CPU consumido pela síntese física (posicionamento e roteamento) foi de 11 minutos, em um PC Pentium III com sistema operacional NT, 450 MHz e 256 MB de RAM. Para a realização completa do roteamento foram necessários 28 passos de *rip-up and reroute.*

```
 Design Summary:
  Number of CLBs:  365 out of   400   91%
  CLB Flip Flops:     307
  4 input LUTs: 665 (2 used as route-throughs)
  3 input LUTs: 176 (94 used as route-throughs)

Number of bonded IOBs:  38 out of 61   62%

Number of clock IOB pads: 1 out of   8   12%
Number of primary CLKs:   1 out of   4   25%
Number of secondary CLKs: 2 out of   4   50%
Number of TBUFs:        240 out of 880   27%
Number of OSC:            1 out of   1  100%
Total equivalent gate count for design: 7196
…
placer score = 357762
Finished Constructive Placer.
REAL time: 4 mins 4 secs
Total CPU  time:  7 mins 41 secs
End of route.  3750 routed (100.00%);
               0 unrouted. Completely routed.
```

Figura 8 - Relatório de síntese física para 12 registradores.

Uma vez gerado o arquivo binário de configuração do FPGA, passou-se à fase de testes na plataforma de prototipação. No momento do *download* envia-se primeiro o programa para a memória e depois o arquivo de configuração. Foram executados programas de diferentes complexidades, todos apresentando comportamento conforme a simulação funcional.

Como ferramenta de teste, foi utilizado um analisador lógico HP1663E, para monitoramento dos linhas de endereço, dados e controle. Esta ferramenta foi muito útil para a depuração de versões iniciais da arquitetura, pois estas apresentavam erros de temporização no acesso à memória, apesar do funcionamento estar correto na simulação funcional VHDL. Observando o funcionamento dos sinais no analisador lógico, alterou-se o código VHDL, obtendo-se assim uma versão com funcionamento correto tanto no nível de simulação quanto na plataforma de prototipação.

6 Conclusão, estado atual e trabalhos futuros

Este trabalho apresentou o desenvolvimento completo de uma arquitetura com os passos de: (*i*) especificação da arquitetura; (*ii*) uso de simulador e tradutor de linguagem de montagem específicos para a arquitetura; (*iii*) simulação funcional em VHDL; (*iv*) implementação VHDL sintetizável; (*v*) síntese lógica e física; (*vi*) implementação física em plataforma de prototipação; (*vii*) validação do hardware com analisador lógico.

Dada a limitação de recursos disponíveis para a implementação do projeto na plataforma de prototipação escolhida, contendo um FPGA com capacidade aproximada de 10.000 portas lógicas equivalentes, foi necessário aguçar a capacidade dos projetistas de extrair o melhor das ferramentas de síntese automatizada. O processo de síntese foi compreendido em profundidade. A relação entre diversas estruturas de uso

geral na linguagem VHDL e seu efeito quantitativo sobre as dimensões da implementação foram avaliados e aproveitados durante as fases de projeto e implementação da arquitetura.

Este processador está disponível publicamente, e pode ser utilizado como um núcleo processador (*core processor*) em projetos conjuntos de hardware e software. Foi esta a motivação que conduziu os autores ao trabalho.

Como trabalhos em desenvolvimento, citam-se: (*i*) inserção de *pipeline* na arquitetura, visando execução com CPI=1. Atualmente, uma primeira versão em VHDL simulável muito simplificada demonstrou ganho inicial de 40% em termos de número de ciclos de relógio em relação à versão sem pipeline. Posteriormente será desenvolvida a versão sintetizável; (*ii*) versão ASIP do processador. O trabalho consiste em personalizar automaticamente o processador conforme a aplicação do usuário, modificando-se o bloco de controle (lembrar que este foi escrito visando fácil inserção/remoção de instruções), o banco de registradores e a ULA; (*iii*) migração do processador e controlador para a plataforma Virtex [4].

Como trabalhos futuros, citam-se: (*i*) geração do código *assembly* à partir de um compilador C, via uso de compiladores reconfiguráveis tais como *lcc* [10]; (*ii*) implementação de diferentes partições hardware/software; (*iii*) avaliação de desempenho destas partições e definição de critérios que auxiliem na partição de novos circuitos; (*iv*) integração do processador desenvolvido a um barramento de alto desempenho, tal como PCI.

7 Referências Bibliográficas

[1] http://www.xess.com/prod006.html. Describes the prototyping platform XS40 and Xstend.

[2] N.Calazans, F.Moraes. *"VLSI Hardware Design by Computer Science Students: How early can they start? How far can they go?".* 1999 Frontiers in Education Conference (FIE'99), 10-13/11/1999, Puerto Rico, p. 13c6-12 to 13c6-17.

[3] Rolf Ernst. "*Codesing of Embedded Systems: Status and Trends*". In: IEEE Design & Test of Computers, April-June 1998, p. 45-54.

[4] http://www.vcc.com/vw.html. "*The Virtual Workbench*". Describes the hardware prototyping platform with a Virtex XCV300.

[5] Hennessy, John, Patterson,David. "*Computer Organization and Design: the Hardware/ Softwarwe Interface*", Englewood Cliffs, 1994.

[6] http://www.xilinx.com/products/xc4000XLA.html. Describes de XC400 family.

[7] http://www.aldec.com/ActiveHDL/40XE/main.htm Contains references to the VHDL simulator.

[8] "*Foundation Series – Quick Start Guide 1.5*". Xilinx, 1998.

[9] http://www.synopsys.com/products/fpga/fpga_express.html Contains references to the VHDL synthesis toool.

[10] C. Fraser, D. Hanson. "A Retargetable C Compiler: Design and Implementation". Redwood City, CA:Benjamin/Cummings, 1995.

Agradecimentos: O autor Fernando G. Moraes agradece o suporte do CNPq (projeto integrado número 522939/96-1) e da FAPERGS (projeto número 96/50369-5). O autor Ney L. V. Calazans agradece o suporte do CNPq (projeto integrado número 520091/96-5) e da FAPERGS (projeto número 98/50970-6).