

# Ambiente de Desenvolvimento de Processador Embarcado para Aplicações de Codesign

Fernando Moraes, Aline Vieira de Mello, Ney Calazans

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

**Abstract:** *This paper presents an environment to compile, simulate and optimize a configurable processor architecture adequate to embedded applications. The configurable parts of the architecture are the instruction set and the data path organization. The instruction set parameterization allows to fit the instruction set to the user application. The internal organization, composed by a data-path and a control unit reflects the instruction set choices, as the number of internal registers, the arithmetic and logic unit functions and the instructions of the control block. The availability of a flexible, low cost, high performance processor is crucial in system-on-a-chip designs. The environment includes five tools: architecture configuration mapper, assembler, simulator, C compiler and VHDL code generator.*

*Keywords: core processors, embedded processors, SoC.*

## 1. Introdução

Sistemas embarcados ou sistemas embutidos são sistemas computacionais que executam uma função específica. Eles possuem a mesma estrutura geral de um computador, mas a especificidade de suas tarefas faz com que não sejam nem usados nem percebidos como um computador [1][2]. Exemplos típicos de sistemas embarcados são chaves ATM, roteadores de rede IP com exigência de Qualidade de Serviço e estações base para comunicação móvel [3] e sistemas de controle automotivo.

O mercado de produtos computacionais força a redução de custos e tempo de projeto, o que leva os projetistas a desejar deslocar a maior parte da funcionalidade dos sistemas embarcados para o *software*, deixando os elementos de *hardware* dedicados apenas às funcionalidades que necessitam de alto desempenho [4]. Considerando estas restrições, De Micheli [1] define *projeto integrado de hardware e software* (em inglês, *hardware/software codesign*, ou simplesmente *codesign*) como: “*a busca do alcance dos objetivos em nível de sistema do produto pela exploração da sinergia entre hardware e software, através do projeto concorrente destas entidades*”.

No processo de *codesign*, o projetista parte da especificação de um sistema, analisando o conjunto de requisitos e restrições para a obtenção de uma ou mais descrições abstratas que representem o sistema computacional. Obtidas as descrições, devem ser extraídas informações destas que permitam direcionar a implementação do mesmo para particionar o sistema em componentes de hardware e software. O particionamento gera novas descrições, tanto para os componentes de hardware, quanto para os componentes de software. Ambas descrições passam pelo processo de síntese, que gera três elementos distintos: o software, o hardware e a interface de comunicação entre o hardware e o software. Para todas as etapas do projeto podem existir estágios de validação, seja via verificação formal, seja via simulação, que permitem avaliar o quão correta está a descrição obtida.

De acordo com vários autores, entre estes [5] e [6], em menos de 7 anos já existirão no mercado circuitos integrados compostos por mais de um bilhão de transistores. Com esta capacidade de integração, pode-se imaginar a inclusão de um sistema computacional completo em um único *chip*, o que cria o conceito de SOC (*System On a Chip*). Dispositivos reconfiguráveis tais como os FPGAs no estado da arte já possuem hoje capacidade para implementar circuitos com cerca de 10 milhões de portas lógicas equivalentes. Isto permite que estes dispositivos sirvam de suporte à prototipação de SOCs e mesmo à construção de SOCs de uso em produtos acabados. Exemplos deste tipo de utilização de dispositivos reconfiguráveis são as iniciativas Excalibur da Altera [7] e Empower! da Xilinx [8]. Estas iniciativas disponibilizam processadores embarcados do tipo soft core (Nios e Microblaze) ou hard core (ARM, MIPS, e PowerPC), juntamente com periféricos e área de lógica reconfigurável de grande porte dentro de um único dispositivo integrado. Isto torna tais dispositivos muito atraentes no desenvolvimento de trabalhos de *codesign* de SOCs. Uma desvantagem clara dos hard cores, e dos soft cores disponíveis no momento é a questão da reconfigurabilidade. Nenhum dos processadores disponíveis apresenta capacidade de adaptação, mesmo parcial, do conjunto de instruções à aplicação. A adaptação do conjunto de instruções à aplicação gera o conceito denominado em inglês de Application Specific Instruction set Processor ou ASIP.

Este artigo tem o objetivo de apresentar uma arquitetura do tipo ASIP e um conjunto de ferramentas que permite configurar a arquitetura para aplicações embarcadas em FPGAs. A arquitetura é parametrizável em termos de módulos funcionais (número de registradores e operações da ULA) e conjunto de instruções. Este processador é utilizado como um *core processor* que implementa as funções do *software*, enquanto que as demais partes do FPGA são utilizadas para a implementação dos componentes de *hardware* e da interface de comunicação. O ganho potencial desta abordagem em relação aos produtos existentes para FPGAs é o aumento do desempenho do processador, devido a sua personalização, e à economia de recursos, sobretudo de área, do FPGA.

Uma possível deficiência está associada a limitações dos FPGAs atuais, os quais dispõem de uma pequena quantidade de memória interna, de forma que sistemas de média e grande complexidade devem ser implementados com uma memória externa, perdendo assim algumas das vantagens inerentes de SOCs.

Este artigo está organizado da seguinte forma. A seção 2 apresenta brevemente a arquitetura básica do processador. A seção 3 apresenta o ambiente de desenvolvimento proposto para a geração de processador embarcado, assim como o configurador da arquitetura. As Seções 4 e 5 descrevem a ferramenta que permite a montagem/simulação e otimização do código VHDL, respectivamente. Finalmente, são apresentadas algumas conclusões e trabalhos futuros.

## 2. Descrição da arquitetura

A arquitetura, denominada **R6**, é uma organização Von Neumann, *Load/Store*, com CPI igual a 2 [9], barramento de dados e endereços de 16 bits. Esta arquitetura é praticamente uma máquina RISC, faltando contudo algumas características gerais de máquina RISC, tal como *pipeline* e módulos de entrada/saída, como tratamento de interrupções. Estas deficiências devem-se ao fato desta arquitetura ter sido originalmente desenvolvida visando o ensino de Organização de Computadores [10] em cursos de graduação. A inclusão de recursos de *pipeline* na arquitetura é hoje um dos tópicos de uma outra disciplina Arquitetura de Computadores I, subsequente à Organização de Computadores na instituição dos autores. No futuro, pretende-se incluir a construção e integração de módulos de entrada/saída no escopo de uma terceira disciplina de graduação (Arquitetura de Computadores II).

Processadores embarcados comerciais simples, tais como o NIOS da Altera e Microblaze da Xilinx, possuem organização semelhante à R6, diferenciando-se principalmente por já disporem de estruturas de entrada/saída, tais como tratamento de interrupções e *timers*. Uma vantagem daqueles em relação à R6 é a possibilidade de dimensionar o soft core para dados e endereços de 16 ou 32 bits.

A comunicação entre o processador e o meio externo compreende: sinais de temporização e inicialização, *clock* e *reset*; sinais de controle do acesso à memória, *ce*, *rw*, *address*, *datain* e *dataout*; e um sinal que indica que o processador não está executando instruções, *e\_halt*.

O projeto da arquitetura, desenvolvido em VHDL parametrizável [11], compreende a descrição de dois blocos principais: o bloco de controle e o bloco de dados. O bloco de controle tem por função gerar os comandos para a busca da instruções e envio de 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 bloco de dados contém 15 registradores de uso geral (o registrador índice 0 é a constante 0); registradores para armazenamento da instrução corrente (*IR*), endereço da próxima instrução a executar (*PC*) e ponteiro de pilha (*SP*); uma ULA (Unidade Lógica e Aritmética) com 15 operações e 1 qualificador de estado, utilizado para os saltos. A Figura 1 ilustra a organização do bloco de dados deste processador, assim como a interface com a memória externa. Os sinais em *itálico*, conectados a cada módulo do processador (registradores, ULA, multiplexadores, *tri-states*) representam os comandos provenientes do bloco de controle. Discussão sobre diferentes estratégias de implementação do bloco de dados pode ser encontrada em [12].

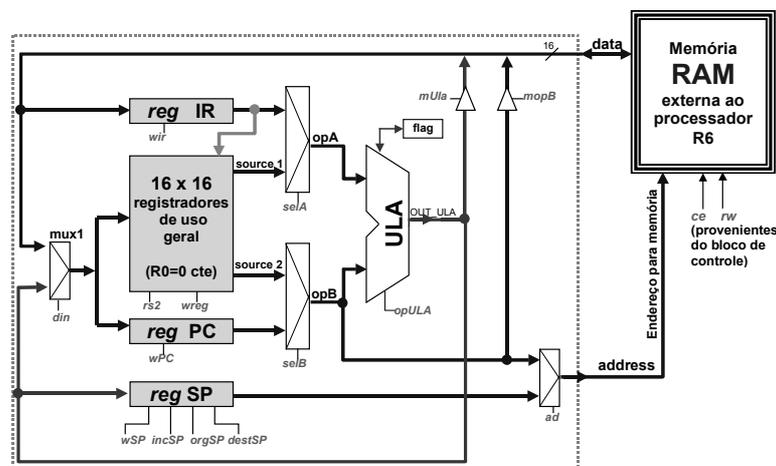


Figura 1 - Bloco de dados da Arquitetura R6, e interface com a memória.

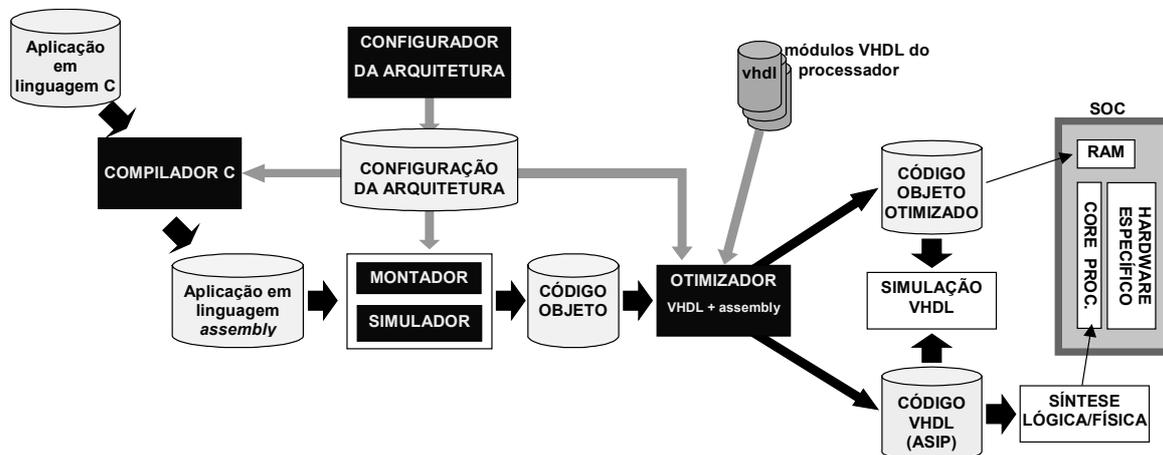
### 3. Ambiente de desenvolvimento para a geração do processador embarcado

Seja dado um sistema composto de uma parte de *hardware* e de uma parte de *software*. Um sistema interessante para o tratamento da parte *software* seria aquele onde, a partir de uma aplicação descrita, por exemplo, em linguagem C, se pudesse obter de forma automatizada o código executável otimizado para esta aplicação, assim como o ASIP, a ser implementado no circuito integrado que executasse este código executável. A parte *hardware* seria implementada a partir de, por exemplo, descrições parametrizáveis em VHDL ou Verilog, adaptadas à aplicação.

Para que tal sistema opere conforme o esperado, o usuário deve possuir uma forma de configurar a arquitetura. No sistema descrito aqui, isto é feito pela geração de um arquivo de

configuração. Esta configuração permite ao compilador C gerar a descrição em linguagem de montagem (*assembly*) da parte *software*, a qual é utilizada para a geração do código objeto pelo montador. Este código objeto, a configuração da arquitetura e um banco de dados com módulos VHDL permitem a geração do código executável otimizado e do código VHDL que representará a descrição personalizada do processador (ASIP). Este ASIP é sintetizado, resultando no *core processor* que executará o código objeto armazenado em memória.

A Figura 2 ilustra o ambiente de desenvolvimento implementado. Este ambiente possui cinco ferramentas: (i) configurador da arquitetura, (ii) montador, (iii) simulador, (iv) compilador C e (v) otimizador. O montador e o simulador permitem gerar o código objeto de uma dada aplicação e simular funcionalmente o processador, em função da configuração da arquitetura. O montador e o simulador são vistos como uma única ferramenta, pois estão integrados, funcionando sob uma única interface gráfica. O otimizador modifica a descrição VHDL, e o próprio código *assembly*, de forma que o hardware do processador só venha a ter os recursos estritamente necessários para executar a aplicação a que se destina. O configurador da arquitetura está implementado para montagem, simulação e otimização. O compilador C encontra-se em desenvolvimento, através do uso de compiladores configuráveis, como GCC [13] ou LCC [14].



**Figura 2 - Ambiente de desenvolvimento para a geração de processador embarcado.**

Uma importante ferramenta deste ambiente é o configurador da arquitetura, o qual permite gerar uma descrição da arquitetura, sendo utilizado para montagem/simulação de aplicações *assembly* e otimização do código *assembly* e VHDL.

O *configurador* de arquiteturas contém os seguintes parâmetros: (i) o nome da arquitetura; (ii) o número de registradores; (iii) o tamanho da memória de dados e programa; (iv) o tamanho da palavra; (v) o funcionamento da pilha; (vi) os qualificadores de estado; e (vii) as instruções da arquitetura. O número de registradores pode variar entre 1 e 16. O tamanho da memória pode ter de 255 a 65535 posições. A atual versão em desenvolvimento tem tamanho fixo de palavra igual a 16 bits. A pilha pode ser pós-incrementada (crescendo da posição de memória com endereço menor para a posição de memória com endereço maior) ou pré-decrementada (crescendo da posição de memória com endereço maior para a posição de memória com endereço menor).

Existe nesta ferramenta, o *configurador*, um conjunto de instruções pré-definidas. O objetivo é associar mnemônicos e campos de operandos a estas instruções pré-definidas, de forma a permitir a leitura do código *assembly* de uma dada aplicação e gerar corretamente o código objeto. É permitido ao usuário definir uma instrução cujo comportamento não esteja especificado no configurador. Caso isto ocorra, a montagem pode ser executada, porém a

simulação funcional não pode ser feita sem uma alteração do código fonte do simulador. Esta nova instrução deverá ser definida na organização do processador, em linguagem VHDL, para que seja possível sintetizá-la.

#### 4. Ambiente de montagem e simulação

O ambiente de montagem e simulação é parametrizável, permitindo a geração e a validação de diferentes arquiteturas. O simulador, desenvolvido em linguagem Java, possibilita a simulação de programas descritos em *assembly*, para a arquitetura definida pelo configurador descrito na Seção anterior.

A Figura 3 mostra a janela principal do simulador. A esquerda desta figura está apresentada a tabela de memória, contendo em cada linha a instrução em *assembly*, o endereço da posição da memória e o código objeto. Ao centro há a tabela de símbolos, onde são apresentados o nome do símbolo, seu endereço de memória e o seu valor. A direita da figura estão localizados os registradores de uso geral e os registradores IR, PC e SP. Na parte inferior são ilustrados os botões de controle *Step*, *Run*, *Pause*, *Stop* e *Reset* e as opções de velocidade de simulação *Lento*, *Normal* e *Rápido*. Os qualificadores de estado encontram-se na parte inferior à direita.

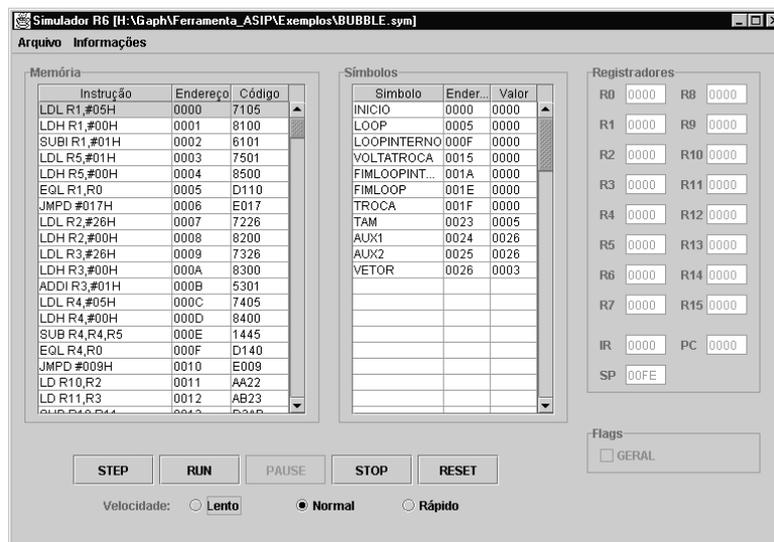


Figura 3 – Janela do Simulador.

A ferramenta de montagem é transparente para o usuário, pois a mesma está integrada ao simulador como um método nativo [15]. A interface com usuário é apenas a interface do simulador. Um trabalho em desenvolvimento para este ambiente integrado é um editor de texto para a descrição *assembly*. Esta integração ocorre devido ao fato de que o simulador não processa instruções diretamente em *assembly*, mas sim em código objeto. O montador, escrito em linguagem C, é responsável por converter uma aplicação em *assembly* para o código objeto determinado na configuração da arquitetura. Para realizar a conversão de uma aplicação em *assembly* para código objeto, o montador passa por três fases: análise sintática, substituição de texto e criação de arquivos.

- A fase de análise sintática consiste em verificar se as linhas de instruções correspondem às instruções descritas pela arquitetura, ou seja, identifica qual instrução está sendo referenciada, quantos registradores, quantas variáveis e quantos *labels* são relacionados à mesma. Nesta fase de análise sintática é criada a tabela de símbolos, as variáveis e as constantes com seus endereços respectivos. Quando é encontrado um identificador não presente na tabela, este é chamado de *pseudo-label*, definição dada às palavras que podem

estar declaradas mais adiante no programa.

- A fase de substituição de texto consiste em substituir *labels*, *pseudo-labels*, *variáveis* e *constantes*, por seus respectivos códigos binários, respeitando o tamanho da palavra reservado a eles. Por exemplo, suponha a instrução **LDH R1,#x** (carregar a parte alta do registrador **R1** com a constante “x”) e que seu código objeto conforme o arquivo de configuração seja “7 Rt Cte Cte” (opcode 7, seguido de registrador destino e dois campos de constantes de 8 bits). Se a constante “x” for A3, o código objeto final será “71A3”.
- A terceira etapa do montador é responsável pela criação de três arquivos: (i) arquivo no formato Intel “hex”, utilizado para fazer *download* do código objeto na memória da plataforma de prototipação; (ii) “sym” utilizado pelo simulador; e (iii) o arquivo opcional “txt” utilizado no *test\_bench* de simulações VHDL do processador e pelo usuário para verificar o código montado para cada instrução.

Erros encontrados durante a execução de uma das fases do montador são salvos em um arquivo de mensagens. Este arquivo é lido pelo simulador após a execução do montador a fim de que os erros sejam apresentados ao usuário e não se prossiga a simulação. Erros na execução do montador não possibilitam a simulação, porque os mesmos indicam que as instruções da aplicação *assembly* não condizem com as instruções existentes na arquitetura.

Uma vez o código objeto gerado (montagem), a simulação do programa pode ser realizada. A capacidade de simular programas para diferentes arquiteturas é possível devido a forma que as ferramentas de montagem e simulação foram escritas. Na ferramenta de configuração da arquitetura relaciona-se os mnemônicos com as instruções pré-definidas, conforme explicado anteriormente. Por exemplo, é possível associar o *mnemônico* ADIÇÃO à instrução pré-definida ADD, e desta forma permitir ao simulador encontrar o código objeto da instrução ADIÇÃO, para que o mesmo seja interpretado e executado como sendo a instrução ADD.

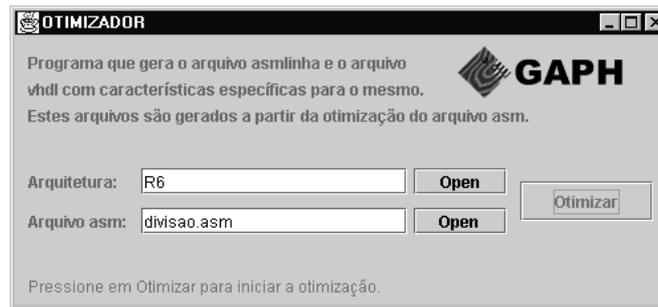
Como as instruções podem afetar qualificadores diferentes e independentes de qualquer outra instrução, cada instrução contém uma lista dos qualificadores que são afetados com a sua execução. Esta lista também está presente no arquivo de configuração e é feito acesso a esta ao final da execução de cada instrução para que os qualificadores sejam atualizados conforme a mesma.

As instruções que fazem acesso à pilha dependem do registrador SP, o qual deve ser inicializado pela aplicação. Esta inicialização é feita pela instrução LDSP. Caso não seja inicializado, o simulador assume valores *default* para o endereço inicial da pilha. Se a pilha funciona com pós-incremento, a posição inicial da pilha será o endereço de memória que corresponde a 90% do tamanho da memória. Caso a pilha funcione com pré-decremento, a posição inicial desta será o último endereço de memória.

## 5. Otimizador

A ferramenta de otimização, permite gerar o código *assembly* otimizado de uma aplicação descrita para arquitetura R6 e o código VHDL do processador desta arquitetura. A otimização é realizada visando ter um processador dedicado à aplicação, apresentando uma redução no custo em termos de área e um aumento em seu desempenho.

A Figura 4 ilustra a interface gráfica do otimizador, que contém o nome da arquitetura descrita pelo configurador de arquiteturas e o nome do arquivo que contém o código *assembly* da aplicação para tal arquitetura.



**Figura 4 – Interface gráfica da ferramenta de otimização do programa *assembly* e geração do ASIP.**

A otimização consiste na redução do número de registradores, do número de operações da ULA e do número de instruções utilizadas pelo processador da arquitetura. A otimização é realizada a partir do arquivo de configuração da arquitetura, do código *assembly* da aplicação e do código VHDL do processador da arquitetura. O processo de otimização ocorre em três fases: inicialização do otimizador, otimização do código *assembly* e otimização do código VHDL.

- Na fase de *inicialização do otimizador* é gerada a lista de registradores, a lista de operações da ULA e a lista de instruções do processador com base no arquivo de configuração da arquitetura.
- Na fase de *otimização do código assembly* é gerado um arquivo que contém o código *assembly* da aplicação otimizado. No código *assembly* otimizado, os registradores são substituídos pelo primeiro registrador livre, ou seja, ao encontrar a instrução ADD R6,R2,R10 o otimizador substituirá a mesma por ADD R1,R2,R3, identificando na lista de registradores do otimizador que o registrador R1 corresponde ao registrador R6 e assim sucessivamente. Também durante a otimização do código *assembly* são verificadas quais instruções do processador são utilizadas selecionando as mesmas da lista de instruções. Cada instrução do processador tem vinculada a si uma operação da ULA. Logo, ao selecionar o conjunto de instruções utilizadas na aplicação, tem-se a lista de operações da ULA correspondente. Ao término desta fase, o otimizador sabe o número de registradores, a lista de operações da ULA e as instruções realmente utilizadas pela aplicação.
- Na fase de *otimização do código VHDL* é gerado um arquivo que contém o código VHDL do processador otimizado. A otimização é realizada com base no arquivo que contém o código VHDL original do processador e nas informações obtidas na fase anterior. Estas são o total de registradores utilizados, a lista de operações da ULA e a lista de instruções utilizadas.

## 5.1 Otimização do código VHDL

O código VHDL possui três trechos de código que são otimizados: o bloco de registradores, a ULA e o bloco de controle. Nestes trechos de códigos foram colocadas palavras-chaves que indicam porções de código que são substituídos durante a otimização.

- Na arquitetura R6, o bloco de registradores é implementado na forma de um *array* de registradores, e o número de registradores é determinado pelo valor da variável **nregs** Na entidade do banco de registradores, mostrado na Figura 5, a palavra-chave **\$nreg\$** é substituída pelo número de registradores realmente utilizados pela aplicação. A parte *architecture* não é modificada.

```

entity bcregs is
  generic(nregs: integer := $nreg$); -- indica o número de registradores da arq.
  port( ck, rst, wreg, rs2: in std_logic;
        ir, inREG: in reg16;
        sourcel, source2: out reg16
        );
end entity;
architecture bcregs of bcregs is
  type banco is array (0 to 15) of std_logic_vector(15 downto 0);
  signal reg: banco;
  signal wen: std_logic_vector(15 downto 0);
  signal destA, destB: std_logic_vector(3 downto 0);
begin
  destB <= ir(3 downto 0) when rs2 = '0' else ir(11 downto 8);
  destA <= ir(7 downto 4);
  sourcel <= reg( CONV_INTEGER(destA) );
  source2 <= reg( CONV_INTEGER(destB) );
  g1:for i in $nregs$ downto 0 generate --** nregs é parâmetro externo *
    ri: reg16clear port map(ck=>ck, rst=>rst, ce=>wen(i), d=>inREG, q=>reg(i) );
    wen(i) <= '1' when CONV_INTEGER( ir(11 downto 8) ) = i and wreg='1' else '0';
  end generate g1;
end bcregs;

```

Figura 5 – Código VHDL do banco de registradores.

- A otimização da ULA consiste em substituir as operações da arquitetura R6, apresentada na Figura 6, pela lista de operações da ULA correspondente às instruções do código *assembly* da aplicação. A correspondência entre a instrução encontrada no programa e a respectiva operação que deve ser feita em VHDL é definida também no configurador da arquitetura.

```

out_ula <=      opB + '1'
               when uins.ula=incB else
               opA + opB           when uins.ula=add   else
               opA - opB           when uins.ula=sub   else
               opA or opB          when uins.ula=ou    else
               opA and opB         when uins.ula=e     else
               opA xor opB         when uins.ula=ouX   else
               opB + extended8     when uins.ula=addi  else
               opB - extended8     when uins.ula=subi  else
               opB(15 downto 8) & opA(7 downto 0) when uins.ula=cte_L else
               opA(7 downto 0) & opB(7 downto 0) when uins.ula=cte_H else
               opB(14 downto 0) & iflag when uins.ula=s1  else
               iflag & opB(15 downto 1) when uins.ula=sr  else
               not opB(15 downto 0) when uins.ula=notB else
               opB + extended4     when uins.ula=jmp   else
               opA;                -- DEFAULT: passaA

```

Figura 6 – Código VHDL da ULA da Arquitetura R6.

O código VHDL gerado contém apenas as operações da ULA utilizadas no programa *assembly*. Entretanto existem duas operações que independem da aplicação e são sempre incluídas, *passaA* e *incB*, que são utilizadas para busca de instrução.

- No trecho de código corresponde ao bloco de controle, mostrado na Figura 7, a palavra-chave **\$blococontrole\$** é substituída pela descrição VHDL das instruções selecionadas durante a otimização do código *assembly* da aplicação. Na arquitetura R6, a instrução NOP é sempre incluída, correspondendo à operação *default*.

```

-- GERACAO DAS MICROINSTRUcoes
process(estado)
begin
  case estado is
    -- gera a microinstrucao relativa ao ciclo de fetch
    when FETCH => uins <= (c1, c0,c0, c1, c0,c0,c0,c0, c0,c0, c1,c0,c0,c0, c1,c1, incB );
    when EXECUTA => $blococontrole$
    when others => uins <= (c0, c0, c0, c0, c0,c0,c0,c0, c0,c0, c0,c0,c0,c0, c0,c0, passaA );
  end case;
end process;

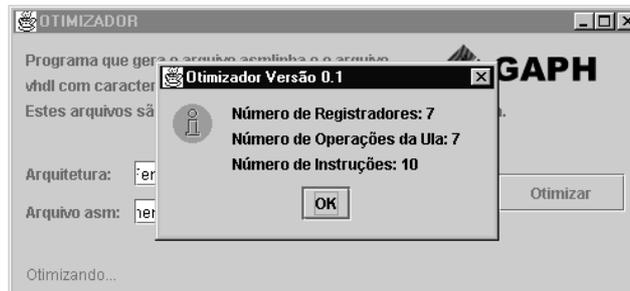
```

Figura 7 - Código VHDL (parcial) do bloco de controle.

Cada instrução tem definida sua correspondente micro-instrução no arquivo de configuração, por exemplo, a instrução ADD tem seguinte código VHDL:

```
if op1=x"0" then
  uins <= (c0, c0,c1, c0, c0,c0,c0,c0, c1,c1, c1,c0,c0,c0, c0,c0, add );    -- add
```

Ao término da execução do otimizador é apresentado o total de registradores, de operações da ULA e o número de instruções utilizadas pela aplicação, como ilustrado na Figura 8.



**Figura 8 - Resultados do Otimizador**

## 6. Resultados

Foram analisadas três aplicações descritas para a arquitetura R6 com o objetivo de verificar se houve aumento do desempenho e redução de área no momento de sintetizar o processador, em relação à arquitetura completa original. As aplicações consistiram em:

- Aplicação 1 - divisão de dois números de 16 bits, através de subtrações sucessivas.
- Aplicação 2 - conversão de texto de minúsculas, para maiúsculas, em um vetor de caracteres.
- Aplicação 3 – ordenamento de um vetor de tamanho  $n$ , através do algoritmo *Bubble Sort*.

A Tabela 1 apresenta os resultados obtidos durante a síntese do processador, para cada um dos programas embarcados.

**Tabela 1 - Análise da síntese de aplicações embarcadas.**

Software Foundation 3.1. Dispositivo Xilinx Virtex XCV300 (total de *slices* disponíveis 3072). Esforço máximo de síntese lógica e física.

	Nº Reg.	Nº Instruções	Nº Oper. ULA	SLICES	Flip Flops	Atraso (ns)
R6 original	16	31	15	555	307	40.608
Aplicação 1	7	10	7	265	147	39.262
Aplicação 2	6	9	5	250	163	42.065
Aplicação 3	8	10	6	261	163	41.804

Os resultados obtidos mostram que os objetivos foram parcialmente alcançados. A área utilizada no FPGA, medida através do número de *slices* (bloco lógico na família Virtex), realmente teve grande redução, na ordem de 50%. Esta redução de área permite ou inserir o processador em um FPGA de menor complexidade, reduzindo os custos, ou liberar mais área para a aplicação que se quer implementar na parte hardware, ou ainda, permitido replicar o processador para permitir multiprocessamento.

O atraso do caminho crítico do circuito, obtido através do uso da ferramenta de análise

de *timing*, permaneceu relativamente constante, na ordem de 40 ns. Isto se deve provavelmente à não modificação da profundidade lógica máxima do circuito. Observando-se os relatórios de *timing*, a profundidade dos caminhos críticos fica em torno de 19-23 blocos lógicos (incluído, neste total o roteamento).

## 7. Conclusão e Trabalhos Futuros

Este trabalho apresentou uma série de ferramentas que disponibilizam ao projetista um processador personalizado à aplicação que este executa. Processadores personalizados (ASIPs) são importantes em aplicações embarcadas, pois estes processadores executarão apenas o programa para o qual foram destinados, não operando como processadores genéricos, e este tipo de produto ainda não se encontra disponível comercialmente para dispositivos do tipo FPGA. Contudo é necessário, para viabilizar a abordagem, o desenvolvimento de ferramentas que automatizem a personalização do processador, de forma que não haja desperdício de hardware em funções que jamais serão utilizadas, e que isto não coloque uma carga adicional sobre o projetista do sistema. Esta automação da personalização permite reduzir custos ou liberar mais área para o hardware do restante do sistema, bem como garantir que figuras de mérito tais como *time-to-market*, não sejam prejudicadas.

Como trabalho em andamento temos o desenvolvimento de um compilador C reconfigurável para a arquitetura parametrizável, o qual está sendo desenvolvido com o auxílio de *retargetabel compilers*, como GCC [13] ou LCC [14].

## 8. Referências

- [1] DE MICHELI, G. - **Hardware/Software Codesign: Application Domains and Design Technologies**. Proceedings of the NATO Advanced Study Institute in Hardware/Software Co-Design, Kluwer Academic Publishers, Italy, 1995.
- [2] LAVAGNO, L., SANGIOVANNI-VICENTELLI A. and HSIEH H. - **Embedded System Codesign: Synthesis and Verification**. Proceedings of the NATO Advanced Study Institute in Hardware/Software Co-Design, Kluwer Academic Publishers, City Italy, 1995.
- [3] SLOMKA, F., DORFEL, M., MUNZENBERGER, R. and HOFMANN, R.- **Hardware/Software Codesign and Rapid Prototyping of Embedded Systems**. IEEE Design & Test of Computers, April 2000.
- [4] SCIUTO, Donatella - **Guest Editor's Introduction: Design Tools for Embedded Systems**. IEEE Design & Test of Computers, April 2000.
- [5] PATT, Y. N., PATEL, S. J., EVERS, M., FRIENDLY, D. H., STARK, J. – **One Billion Transistors, One Uniprocessor, One Chip**. IEEE Computer, pages 51-57, September 1997.
- [6] HAMMOND, L., NAYFEH, B. A., OLUKOTUN, K. - **A Single-Chip Multiprocessor**. IEEE Computer, pages 79-85, September 1997.
- [7] ALTERA, Inc. – **Excalibur Series FPGAs**. available at the homepage: <http://www.altera.com/products/devices/excalibur/exc-index.html>. USA, 2001.
- [8] XILINX, Inc. – **MicroBlaze Soft Processor Overview**. available at the homepage: [http://www.xilinx.com/ipcenter/processor\\_central/microblaze.htm](http://www.xilinx.com/ipcenter/processor_central/microblaze.htm). USA, 2001.
- [9] Hennessy, John, Patterson, David - **Computer Organization and Design: the Hardware/ Software Interface**, Englewood Cliffs, 1994.
- [10] Ney Calazans, Fernando Moraes. **VLSI Hardware Design by Computer Science Students: How early can they start? How far can they go?** In: *1999 Frontiers in Education Conference*, San Jose, PR, pp. 13c6-12 to 13c6-17, November, 1999. Available at: <http://fairway.ecn.purdue.edu/~fie/>.
- [11] MAZOR, S., LANGSTRAAT, P - **A Guide to VHDL**. Kluwer Academic Publishers, Norwell, MA, 1992.
- [12] F. MORAES, N. CALAZANS, E. FERREIRA, D. LIEDKE. **Implementação eficiente de uma arquitetura load/store em VHDL**. CORE 2000, p.2-13.
- [13] **Using and Porting the GNU Compiler Collection (GCC)** - <http://gcc.gnu.org/onlinedocs/gcc.html>
- [14] W. FRASER, C., HANSON R. **A Retargetable Compiler : Design and Implementation**. Adison – Wesley Publishing Company (1995).
- [15] MORGAN, MIKE -**Using Java 1.2**. Editora QUE (1998), pages 82-86.