FACULDADE DE INFORMÁTICA

PUCRS – Brasil

http://www.inf.pucrs.br

IMPLEMENTAÇÃO DE UMA ARQUITETURA LOAD/STORE EM UM AMBIENTE DE PROTOTIPAÇÃO

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

TECHNICAL REPORT SERIES

Number 002
May, 2000

Contact:

moraes@inf.pucrs.br

http://www.inf.pucrs.br/~moraes

F. G. Moraes is a senior lecturer (associate professor) at the PUCRS/Brazil since 1996. His main research topics are digital systems design and fast prototyping, digital systems physical synthesis CAD, telecommunication applications, hardware-software codesign. He is a member of the Hardware Design Support Group (GAPH) at the PUCRS.

E. H. Ferreira is an undergraduate student at the PUCRS/Brazil. Mr. Ferreira is member of the GAPH group since 1999, and receives a federal undergraduate research grant from CNPq (Brazil).

N. L. V. Calazans works at the PUCRS/Brazil since 1986. He is a professor since 1999. His main research topics are digital systems design and fast prototyping, hardware-software codesign, telecommunication applications. He is the leader of the Hardware Design Support Group (GAPH) at the PUCRS.

Copyright © Faculdade de Informática – PUCRS

Published by the Campus Global – FACIN – PUCRS

Av. Ipiranga, 6681

90619-900 Porto Alegre – RS – Brazil

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 report 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.

Sumário

1 Introdução 1
2 Processador R3 3
  2.1 Formato de instrução (registrador reservado R15) 4
  2.2 Instruções 4
  2.3 Modos de endereçamento 7
  2.4 Ciclo de busca (1 ciclo de relógio) 7
  2.5 Ciclo de execução (1 ciclo de relógio) 7
  2.6 Ambiente de desenvolvimento de software R3 8
3 Implementação VHDL do Processador R3 9
  3.1 Bloco de Controle - BC 9
  3.2 Bloco de Dados - BD 11
    3.2.1 Banco de Registradores 11
    3.2.2 Registrador SP 14
    3.2.3 Registrador PC 15
    3.2.4 ULA 16
  3.3 Validação do Processador R3 17
4 Controle do processador na placa XS40 19
  4.1 Acesso à RAM 23
    Ciclo de leitura de uma palavra de 16 bits 23
    Ciclo de escrita de uma palavra de 16 bits 24
5 Implementação Física 26
6 Conclusão, estado atual e futuro 28
7 Referências Bibliográficas 29
8 Anexos – Programas Exemplo 30
  8.1 Programa BubbleSort 30
  8.2 Programa QuickSort 31
9 Anexos – Fontes VHDL do Processador 32
  9.1 Fontes do Processador 32
  9.2 Fontes do Controlador 37
  9.3 Fontes do Test_bench – processador 39
  9.4 Fontes do Test_bench – processador e controlador 40
  9.5 Arquivo com a pinagem para a placa de prototipação – UCF 42

Lista de Figuras

Figura 1 - Sinais de comunicação entre os Blocos de Controle e de Dados. 3
Figura 2 - Simulador para a arquitetura R3. 8
Figura 3 - Código parcial do bloco de controle. 10
Figura 4 - Fragmento de código da implementação que utiliza estruturas hardwired. 10
Figura 5 - Bloco de dados do processador R3. 11
Figura 6 - Banco de registradores. 12
Figura 7 - Soluções para implementar o banco de registradores. 13
Figura 8 - Banco de registradores implementado como array. 13
Figura 9 - Registrador de topo de pilha. 14
Figura 10 - Implementação VHDL do registrador SP. 14
Figura 11 - Registrador PC. 15
Figura 12 - Código fonte VHDL do registrador PC. 15
Figura 13 - Implementação da ULA. 16
Figura 14 - Janela de simulação do Processador R3. 18
Figura 15- Plataforma de prototipação utilizada no projeto. 19
Figura 16 - Multiplexação dos barramentos de endereços e dados. 20
Figura 17 - Diagrama em blocos do controlador. 20
Figura 18 - Simulação do circuito completo, estado de funcionamento S1. 22
Figura 19 - Passos de execução em 1 ciclo de leitura. 23
Figura 20 - Passos de execução em 1 ciclo de escrita. 24
Figura 21 - Relatório de síntese física (12 registradores). 26
Figura 22 - Janela do analisador lógico HP1663E. 27

Lista de Tabelas

Tabela 1 - Resumo dos formatos das instruções. 4
Tabela 2 - Tabela unificada das instruções do processador R3. 6
Tabela 3 - Ciclo de execução das instruções. 7
Tabela 4 - Instruções codificadas pelo somador 17

1 Introdução

A complexidade de sistemas digitais modernos conduz cada vez mais ao uso de altos níveis de abstração durante seu projeto. Linguagens de descrição de hardware poderosas como VHDL têm sido crescentemente empregadas para a captura das especificações em uma descrição formal precisa, embora ainda muito abstrata. Esta descrição pode ou não ser sintetizável automaticamente, mas em geral não o é, limitando-se a servir como uma descrição funcionalmente simulável. VHDL, a exemplo de outras linguagens, permite alterar de forma incremental a descrição abstrata inicial no sentido de torná-la sintetizável ou mesmo eficientemente sintetizável de forma automática. Contudo, alcançar altos níveis de eficiência via síntese automatizada requer um bom conhecimento dos mecanismos de implementação das estruturas sintáticas e semânticas da linguagem subjacente, bem como do poder da ferramenta de síntese e de suas limitações. Dado que o código fonte dos sintetizadores comerciais não se encontra disponível em geral, uma abordagem mais profunda de investir na compreensão dos algoritmos de síntese é quase sempre descartada em um ambiente de produção, além de ser muito menos acessível ao projetista de sistemas digitais.

Este trabalho apresenta a implementação de um núcleo processador (*core processor*) simples com arquitetura Load/Store. O projeto foi descrito unicamente via uso da linguagem VHDL. 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.

A CPU deste processador utiliza o modelo de von Neumann: bloco de controle e bloco de dados, ambos trabalhando com 16 bits. A descrição VHDL modela o bloco de dados estruturalmente (registradores, multiplexadores, barramentos e ULA) e o bloco de controle comportamentalmente (processos para decodificação da instrução e geração dos micro-comandos). Para simular esta descrição foi utilizado o software Active-HDL [1]. Para a validação do código, foi implementado um *test bench* que tem por objetivo avaliar o funcionamento da arquitetura através da instanciação da CPU e da modelagem de uma memória externa, contento o código de máquina do programa a ser executado.

Descreve-se posteriormente a integração do processador a um módulo controlador, o qual disponibiliza a comunicação do processador com a memória externa e outros componentes da plataforma de prototipação utilizada [2].

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, abordando o conceito de *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.

Atualmente o processador está operacional, executando programas armazenados na memória RAM da plataforma de prototipação. Estes programas são escritos em linguagem de montagem (assembly language), passam por um montador chamado *R3sim*, sendo posteriormente carregados na memória da plataforma de prototipação no momento do *download* do projeto.

Inicialmente, esta arquitetura foi proposta como trabalho de final da disciplina Organização de Computadores na Faculdade de Informática da PUCRS (terceiro semestre) [5], onde os alunos devem implementar a especificação em VHDL simulável, sem maior preocupação com a implementação física.

Este relatório está organizado da seguinte forma. O Capítulo 2 descreve a arquitetura do processador R3, o Capítulo 3 sua implementação em VHDL e o Capítulo 4 sua integração à plataforma de prototipação. O Capítulo 5 discute resultados da implementação física.

2 Processador R3

O processador implementado, denominado ***R3***, é uma organização von Neumann (memória de dados/instruções unificada), *Load/Store*, com CPI igual a 2 [6]. *Load/Store* implica que todas as operações lógico/aritméticas são executadas entre registradores, e as operações de acesso à memória reduzem-se a operações de leitura (*load*) ou escrita (*store*). Esta arquitetura é praticamente uma máquina RISC, faltando contudo algumas características que existem em qualquer máquina RISC, tal como *pipeline*.

O bloco de dados foi implementado de forma estrutural e contém 15 registradores para uso geral, um registrador para armazenamento da instrução corrente (**IR**), endereço da próxima instrução a executar (**PC**) e controle dos endereços 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), dois multiplexadores para envio dos valores armazenados nos registradores para os barramentos (dois) da arquitetura, e um decodificador para escrita.

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 bloco de controle e o bloco de dados é feita através de 36 sinais, sendo 16 sinais destinados à microinstrução, 4 sinais de qualificadores de estado (N,Z,C,V) e 16 sinais para transmitir a instrução corrente do bloco de dados para o bloco de controle (conteúdo do IR).

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*.

A Figura 1 ilustra a comunicação entre os blocos de controle e dados, assim como com o ambiente externo ao processador.

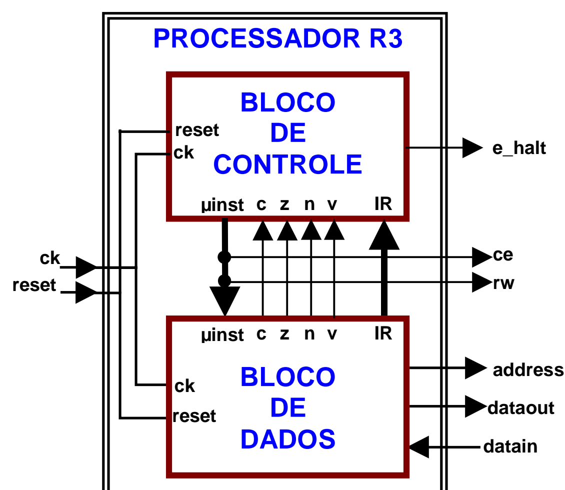


Figura 1 - Sinais de comunicação entre os Blocos de Controle e de Dados.

O processador opera com palavras de 16 bits para dados e endereços. Todas as instruções são executadas em exatamente 2 ciclos de relógio (CPI=2), possuem exatamente o mesmo tamanho, ou seja, 1 palavra de memória, contendo o código da operação e o(s) operando(s), caso este(s) exista(m).

2.1 Formato de instrução (registrador reservado R15)

Os 4 bits mais significativos da palavra de instrução contêm a primeira parte do código da operação (Opcode1, bits 15-12), e em função da instrução podem haver outros bits para a especificação completa do código de operação.

A Tabela 1 resume os formatos de instrução, todos ocupando exatamente 16 bits.

Formato	bits 15 a 12	bits 11 a 8	bits 7 a 4	bits 3 a 0
1	Opcode1	Deslocamento (12 bits)		
2	Opcode1	Rtarget	Constante low/high	
3	Opcode1	Rtarget ou Opcode2	Rsource2 ou Opcode2 ou Opcode3	Rsource1 ou Opcode4

Tabela 1 - Resumo dos formatos das instruções.

2.2 Instruções

As instruções realizadas durante o ciclo de execução compreendem:

- Operações **binárias**, envolvendo 2 registradores fonte (Rsource1 e Rsource2) e registrador destino (Rtarget). Realiza a função Rtarget ← Rsource1 *Op* Rsource2. O*p* assume as funções soma (*ADD*), subtração (*SUB*), e lógico (*AND*), ou lógico (*OR*) e ou-exclusivo (*XOR*), e é determinado pelo valor de *Opcode1*. Formato da instrução (cada campo com 4 bits):

Opcode1	Rtarget	Rsource2	Rsource1

- Operações **unárias**, envolvendo 1 registrador fonte (Rsource1), registrador destino (Rtarget) e dois códigos de operação. Realizando a função Rtarget ← O*p* Rsource1. *Op* assume as funções: cópia de valores entre registradores (*MOV*), deslocamentos lógicos (*SL0*, *SL1*, *SR0*, *SR1*), incremento (*INC*), decremento (*DEC*), inversão (*NOT*) e complemento de 2 (*NEG*). O *Opcode1* é a constante hexadecimal **9** e *Opcode2* é variável e determina a instrução específica. Formato da instrução (cada campo com 4 bits):

Opcode1	Rtarget	Opcode2	Rsource1

- Operações de **salto com Deslocamento curto**. O novo valor do registrador PC é dado pelo seu valor atual mais o valor da constante (o valor de 16 bits é obtido estendendo-se o sinal da constante de 12 bits). *Opcode1* define o tipo específico de salto: *JN* (salta se negativo), *JZ* (salta se zero), *JC* (salta se vai-um), *JV* (salta se transbordo aritmético), *JMP* (salto incondicional) e *JSR* (salto incondicional para subrotina, o conteúdo de PC é gravado na pilha cujo topo é apontado por SP previamente incrementado). Extensão de sinal do operando é automaticamente provida em hardware. Formato da instrução:

Opcode1	**Deslocamento (12 bits)**

- Operações de **salto relativo a registrador**. O novo valor do registrador PC é dado pelo seu valor atual somado ao conteúdo do registrador *Rsource1*. O campo *Opcode2* assume as mesma função que nos saltos com Deslocamento curto (*Opcode1* é a constante hexadecimal **6**). Formato da instrução:

Opcode1	**0H**	**Opcode2**	**Rsource1**

- Operações de **salto absoluto**. O novo valor do registrador PC é dado pelo conteúdo do registrador *Rsource1*. O campo *Opcode2* assume as mesma função que *Opcode1* nos saltos com Deslocamento curto (*Opcode1* aqui é a constante hexadecimal **6**). Formato da instrução:

Opcode1	**1H**	**Opcode2**	**Rsource1**

- Operações de **carga** de dados da memória, modo imediato (load). O campo *Opcode1* define as funções: LDL (carga da parte baixa do registrador) e LDH (carga da parte alta do registrador). Portanto, para carregar uma constante igual ao tamanho da palavra (16 bits), são necessárias duas instruções (LDL e LDH). Formato da instrução:

Opcode1	**Rtarget**	**Constante (8 bits)**

- Operações de **carga** de dados, modo indireto (load). O registrador *Rtarget* recebe o conteúdo de memória apontado pelo registrador *Rsource1*. Formato da instrução:

Opcode1	**Rtarget**	**1H**	**Rsource1**

- Operações de **escrita** de dados, modo indireto (store). A posição de memória apontada por Rsource2 recebe o conteúdo de Rsouce1. Formato da instrução:

Opcode 1	**0H**	**Rsource2**	**Rsource1**

- 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).

A Tabela 2 apresenta a codificação completa das instruções do Processador R3.

Instrução		FORMATO DA INSTRUÇÃO – Registrador R15				
		15 - 12	11 - 8	7 - 4	3 - 0	AÇÃO
JN	**desloc**	**0**	Deslocamento (12 bits)			if (N=1) PC ← PC + **ext_sinal &** desloc
JZ	**desloc**	**1**	Deslocamento (12 bits)			if (Z=1) PC ← PC + **ext_sinal &** desloc
JC	**desloc**	**2**	Deslocamento (12 bits)			if (C=1) PC ← PC + **ext_sinal &** desloc
JV	**desloc**	**3**	Deslocamento (12 bits)			if (V=1) PC ← PC + **ext_sinal &** desloc
JMP	**desloc**	**4**	Deslocamento (12 bits)			PC ← PC + **ext_sinal &** desloc
JSR	**desloc**	**5**	Deslocamento (12 bits)			PMEM(SP)←PC; SP←SP+1; PC← PC + **ext_sinal &** desloc
JN	**Rs1, R**	**6**	**0**	**0**	Rsource1	if (N=1) PC ← PC + Rs1
JZ	**Rs1, R**	**6**	**0**	**1**	Rsource1	if (Z=1) PC ← PC + Rs1
JC	**Rs1, R**	**6**	**0**	**2**	Rsource1	if (C=1) PC ← PC + Rs1
JV	**Rs1, R**	**6**	**0**	**3**	Rsource1	if (V=1) PC ← PC + Rs1
JMP	**Rs1, R**	**6**	**0**	**4**	Rsource1	PC ← PC + Rs1
JSR	**Rs1, R**	**6**	**0**	**5**	Rsource1	PMEM(SP)←PC; PC← PC+ Rs1; SP←SP+1
JN	**Rs1**	**6**	**1**	**0**	Rsource1	if (N=1) PC ← Rs1
JZ	**Rs1**	**6**	**1**	**1**	Rsource1	if (Z=1) PC ← Rs1
JC	**Rs1**	**6**	**1**	**2**	Rsource1	if (C=1) PC ← Rs1
JV	**Rs1**	**6**	**1**	**3**	Rsource1	if (V=1) PC ← Rs1
JMP	**Rs1**	**6**	**1**	**4**	Rsource1	PC ← Rs1
JSR	**Rs1**	**6**	**1**	**5**	Rsource1	PMEM(SP)←PC; PC← Rs1; SP←SP+1
LDSP	**Rs1**	**6**	**2**	**0**	Rsource1	SP ← Rs1 (inicializa o apontador de pilha)
RTS		**6**	**2**	**1**	0	PC ← PMEM(SP-1); SP←SP-1
NOP		**6**	**2**	**2**	0	Nenhuma ação
HALT		**6**	**2**	**3**	0	suspende seqüência de ciclos de busca e execução
LDL	**Rt, #cte**	**7**	Rtarget	constante(7-0)		Rt ← Rt_{high} & constante *(setar Z e N)*
LDH	**Rt, #cte**	**8**	Rtarget	constante(15-8)		Rt ← constante & Rt *(setar Z e N)*
MOV	**Rt, Rs1**	**9**	Rtarget	**0**	Rsource1	Rt ← Rs1 *(setar Z e N)*
LD	**Rt, Rs1**	**9**	Rtarget	**1**	Rsource1	Rt ← PMEM (Rs1)
SL0	**Rt, Rs1**	**9**	Rtarget	**2**	Rsource1	Rt ← Rs1[14:0] & '0'; C ← Rt(15) *(setar Z e N)*
SR0	**Rt, Rs1**	**9**	Rtarget	**3**	Rsource1	Rt ← '0' & Rs1 [15:1]; C ← Rt(0), *(setar Z e N)*
SL1	**Rt, Rs1**	**9**	Rtarget	**4**	Rsource1	Rt ← Rs1[14:0] & '1'; C ← Rt(15) *(setar Z e N)*
SR1	**Rt, Rs1**	**9**	Rtarget	**5**	Rsource1	Rt ← '1' & Rs1[15:1]; C ← Rt(0) *(setar Z e N)*
INC	**Rt, Rs1**	**9**	Rtarget	**6**	Rsource1	Rt ← Rs1+ 1 *(setar todos os flags)*
DEC	**Rt, Rs1**	**9**	Rtarget	**7**	Rsource1	Rt ← Rs1–1 *(setar todos os flags)*
NOT	**Rt, Rs1**	**9**	Rtarget	**8**	Rsource1	Rt ← not (Rs1) *(setar Z e N)*
NEG	**Rt, Rs1**	**9**	Rtarget	**9**	Rsource1	Rt ← not (Rs1) + 1 *(setar todos os flags)*
ADD	**Rt,Rs1,Rs2**	**A**	Rtarget	Rsource2	Rsource1	Rt ← Rs1 ADD Rs2 *(setar todos os flags)*
SUB	**Rt,Rs1,Rs2**	**B**	Rtarget	Rsource2	Rsource1	Rt ← Rs1 SUB Rs2 *(setar todos os flags)*
AND	**Rt,Rs1,Rs2**	**C**	Rtarget	Rsource2	Rsource1	Rt ← Rs1 AND Rs2 *(setar Z e N)*
OR	**Rt,Rs1,Rs2**	**D**	Rtarget	Rsource2	Rsource1	Rt ← Rs1 OR Rs2 *(setar Z e N)*
XOR	**Rt,Rs1,Rs2**	**E**	Rtarget	Rsource2	Rsource1	Rt ← Rs1 XOR Rs2 *(setar Z e N)*
ST	**Rs2, Rs1**	**F**	**0**	Rsource2	Rsource1	PMEM (Rs2) ← Rs1

Observação: Deve-se notar que estão sobrando códigos para expansões futuras (Ex: F320 não é instrução definida).

Tabela 2 - Tabela unificada das instruções do processador R3.

2.3 Modos de endereçamento

Os modos de endereçamento utilizados pelo processador R3 são:

- **A registrador**: operando é conteúdo do registrador indicado na instrução. É utilizado em todas as operações lógicas e aritméticas.
- **A registrador indireto**: endereço do operando está em registrador. É utilizado nas instruções *LD* e *ST*.
- **Imediato (#):** operando é o próprio dado. É utilizado nas instruções *LDL* e *LDH*.
- **Relativo ao PC**: operando é uma constante em complemento de 2 a ser somada ao conteúdo do PC para assim gerar o endereço do operando. É utilizado somente para saltos e chamada de subrotina.

Observações: (*i*) *n*ão há modo direto de endereçamento; (*ii*) pilha funciona com pós-incremento para empilhar e pré-decremento para desempilhar, ou seja, o SP aponta sempre para uma posição livre no topo da pilha.

2.4 Ciclo de busca (1 ciclo de relógio)

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*) incremento do **PC**. Para que isso se tornasse viável, foram respeitadas duas restrições no hardware. A primeira diz respeito ao endereçamento da memória que é feito pela saída da ULA e a segunda, a implementação do contador de programa (**PC**) com um mecanismo de auto-incremento.

2.5 Ciclo de execução (1 ciclo de relógio)

Todas as instruções tem a fase de execução realizada em um ciclo de relógio. Portanto, para carregar uma constante igual ao tamanho da palavra (16 bits), são necessárias duas instruções (LDL e LDH). Para ler/escrever um dado contido numa posição de memória especificada por um endereço de 16 bits são necessárias 3 instruções em linguagem de montagem: as duas primeiras carregam em um registrador a parte alta e baixa de um endereço (LDL e LDH) e a terceira instrução realiza a leitura/escrita (LD ou ST). A Tabela 3 mostra as micro-instruções para o ciclo de execução para a maior parte das instruções.

LOAD: carregar dados em registrador destino Rt	
Registrador indireto: *LD Rt, Rs1*	Rt ← PMEM(Rs1)
Imediato: LDL Rt, #cte e LDH Rt, #cte	$Rt_{high/low}$ ← constante $_{high/low}$
STORE: gravar conteúdo de um registrador Rs1 no endereço armazenando em Rs2	
Registrador indireto: *ST Rs2, Rs1*	PMEM(Rs2) ← Rs1
Move: MOV Rt, Rs1	Rt ← Rs1
Operações lógicas e aritméticas em modo registrador ou operações unárias (SL0 até NEG):	Rt ← [Rs1] op Rs2

Os campos Rt, Rs1, Rs2 e constante são extraídos dos campos do registrador R15, carregado no ciclo de busca.

Tabela 3 - Ciclo de execução das instruções.

2.6 Ambiente de desenvolvimento de software R3

Um ambiente de desenvolvimento de software foi implementado para testar os programas escritos em linguagem de montagem. A Figura 2 ilustra a janela do simulador desenvolvido para a R3, onde se exibe: o conteúdo da memória em formato de instruções em linguagem de montagem e em código objeto, o conteúdo dos registradores, o valor de todos os rótulos definidos no programa, e o estado dos qualificadores. Existem comandos para controlar a velocidade de execução (do passo a passo a execução rápida).

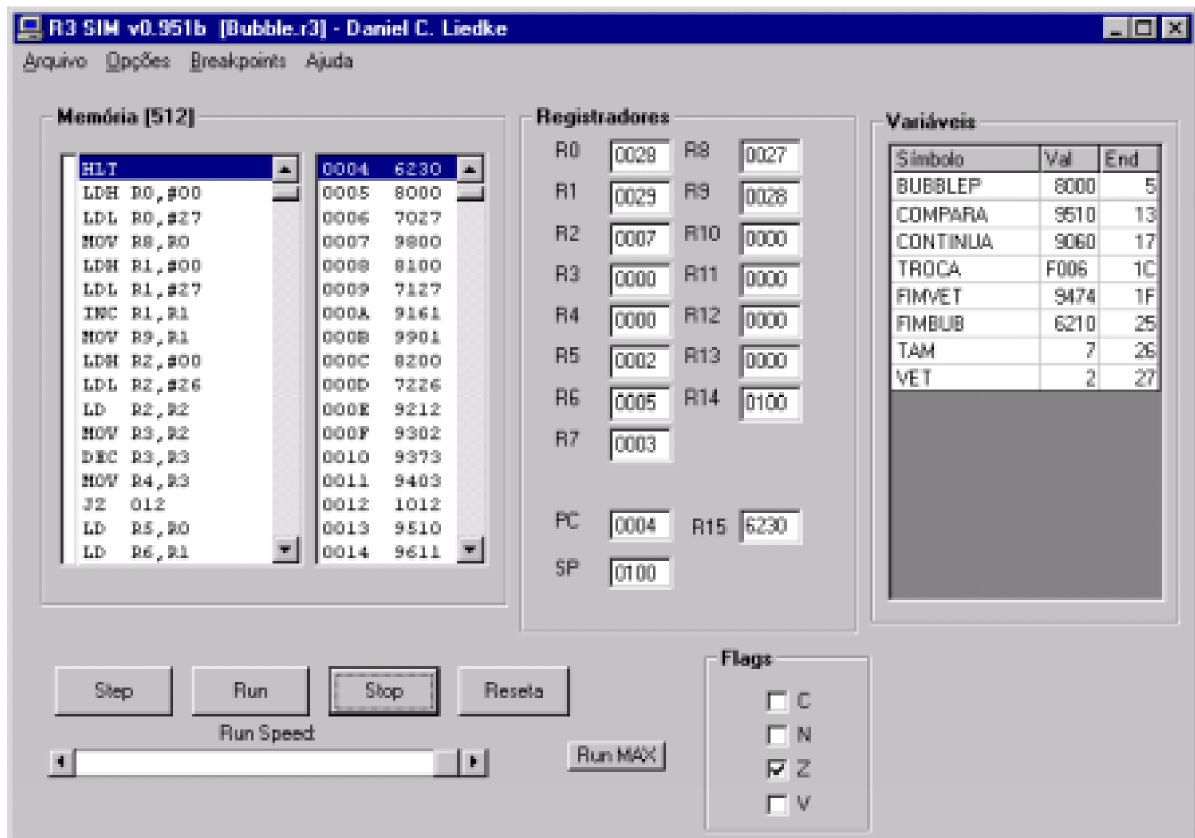


Figura 2 - Simulador para a arquitetura R3.

O ambiente de desenvolvimento construído contém, além do simulador, um montador e um editor de textos para linguagem de montagem integrados. Além disto, o ambiente pode gerar automaticamente um arquivo com o código objeto da aplicação, no formato adequado para carga na memória da placa de prototipação.

3 Implementação VHDL do Processador R3

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*. A implementação do bloco de controle foi feita em VHDL *comportamental*, sendo constituído basicamente por 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.

Ao longo do projeto foram utilizadas formas variadas de implementação que possibilitaram testar maneiras alternativas de desenvolvimento do projeto, objetivando: *(i)* redução dos recursos de hardware necessários à implementação; *(ii)* aumento de desempenho; *(iii)* clareza do código; *(iv)* otimização. De uma forma geral, todos os itens descritos anteriormente afetaram de forma significativa diversos módulos que compõem a arquitetura.

A seguir serão descritos os dois módulos que compõe o processador: bloco de controle e bloco de dados.

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.

A Figura 3 mostra o código (parcial) do BC. 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. Seu comportamento é determinado por um processo sincronizado com o clock e também dependente do sinal assíncrono reset. Logo no início do processamento, após uma borda de subida do sinal reset, halt recebe o valor zero, e permanece neste estado durante a execução do programa. Assim que for decodificada uma instrução halt, o sinal assume o valor 1 indicando suspensão do processamento.

O segundo processo contempla 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 pertinente. Uma vez terminada a execução da instrução, inicia-se novo ciclo de busca.

Na Figura 3 é mostrado o código VHDL que implementa as microinstruções relativas a três 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

  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,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);   -- 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;
```

observação: cz é o bit constante 0 e c1 é o bit constante 1

Figura 3 - Código parcial do bloco de controle.

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, apresentada na Figura 4, consiste em escrever uma equação Booleana para cada sinal da microinstrução (parte de controle *hardwired*). Esta solução a primeira vista parece mais otimizada, pois suprime-se a grande estrutura de seleção.

```
    if (ir(15 downto 12)=d0 and n='1') or
        (ir(15 downto 12)=d1 and z='1') or
        (ir(15 downto 12)=d2 and c='1') or
        (ir(15 downto 12)=d3 and v='1') or
        (ir(15 downto 12)=d4) or
        (ir(15 downto 12)=d5) or
        (ir(15 downto 12)=d7) or
        (ir(15 downto 12)=d8) then uins.rreg15  <= '1';
    else uins.rreg15  <= '0';
```

Em uma implementação hardwired cada sinal de controle é gerado independentemente, no exemplo temos a micro-instrução rreg15 sendo gerada

Figura 4 - Fragmento de código da implementação que utiliza estruturas hardwired.

Observando os testes realizados utilizando as duas formas alternativas descritas, verificou-se que ambas formas de implementações resultam nas mesmas freqüência de execução e taxa de ocupação do FPGA. A implementação que utiliza estruturas *case* realiza em uma única instrução VHDL a atribuição a todos os campos da micro-instrução, portanto é simples de modificar e fácil de implementar. Por outro lado, a implementação hardwired realiza atribuições diretas e condicionais a

cada campo da micro-instrução em separado, o que compromete os quesitos simplicidade de modificação e facilidade de implementação. Optou-se então pela implementação que utiliza estrutura *case*.

A atual bloco de controle ocupa um grande espaço no hardware implementado (em termos de blocos lógicos), comprometendo a freqüência de funcionamento, atualmente em 2MHz. Estudos futuros prevêem o desenvolvimento de estratégias de implementação de bloco de controle com ocupação reduzida de área.

3.2 Bloco de Dados - BD

A Figura 5 ilustra a estrutura do BD do processador. Como pode ser observado, 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, com os 4 qualificadores). Estes 4 módulos são conectados através de barramentos, multiplexadores e portas tristate.

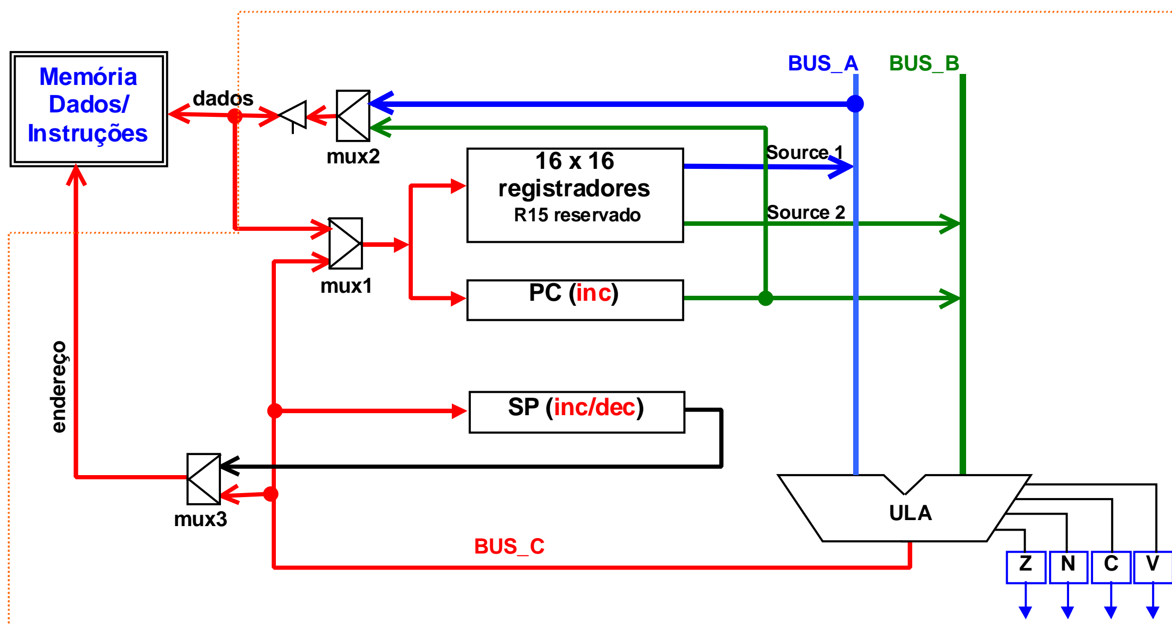


Figura 5 - Bloco de dados do processador R3.

As Seções seguintes detalham a implementação do banco de registradores, o registrador SP, o registrador PC e a Unidade Lógico Aritmética.

3.2.1 Banco de Registradores

A Figura 6 mostra a estrutura do banco de registradores. Existem 15 registradores de propósito geral (R0 a R14) e um registrador reservado para a instrução corrente (R15). R15 é responsável por armazenar o código da operação (opcode) da instrução atual e o(s) operando(s) desta. Ele referencia também implicitamente os registradores a serem lidos ou aquele a ser escrito.

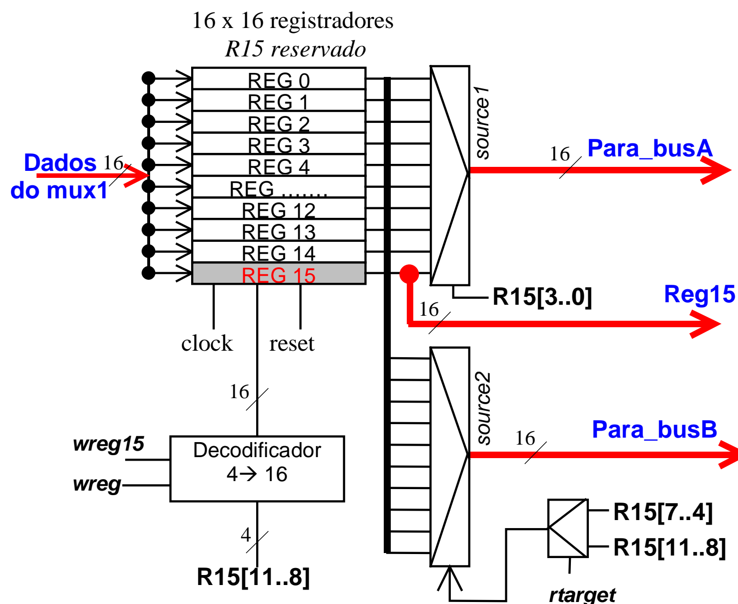


Figura 6 - Banco de registradores.

Componentes do banco de registradores:

(i) Decodificador de escrita 4x16: tem como entradas o sinal *wreg* (sinal de habilitação do decodificador - gerado pelo bloco de controle), *wreg15* (escrita no registrador 15 no momento da busca da instrução) e o os bits [11 a 8] do registrador reservado R15. Estes quatro bits do R15 selecionam qual registrador irá receber o conteúdo proveniente do multiplexador externo *mux1* (Figura 5). Há uma única porta de escrita, ou seja, é possível escrever em apenas um registrador por vez.

(ii) Multiplexadores de leitura: foi utilizada uma solução mista com 8 multiplexadores 2x1 para cada saída de leitura. Estes multiplexadores possuem como entrada todas as saídas dos registradores. O registrador cuja saída é enviada para o barramento *busA* é selecionado pelo bits [3 a 0] do R15, e no barramento *busB* pelo sinal bits [11 a 8] ou [7 a 4]. Estas duas portas de leitura possibilitam fazer duas leituras simultâneas, colocando o conteúdo de um registrador no *busA* e o conteúdo de outro registrador (ou o mesmo) no *busB*.

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 7 apresenta as 3 soluções implementadas e testadas, para uma dada decodificação.

A primeira solução é baseada em decodificação, sendo simples de ser codificada. Porém, esta solução implica em dois decodificadores 16x16:1x16 com largura de palavra igual a 16, tornando o hardware demasiadamente grande para ser implementado no FPGA Xilinx XC4010 da plataforma [7] (utilização de Blocos Lógicos - CLBs em torno de 450 das 400 disponíveis).

A segunda solução requer menos hardware, porém necessita 512 buffers tristate (16 bits * 16 registradores * 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 com sucesso.

A terceira solução é um misto das duas anteriores, baseando-se no uso de decodificação 2x1 seguido de barramentos tri-state. Esta solução resultou em um em hardware sintetizável, com 388 CLBs (dos 400 CLBs disponíveis). Este número de blocos lógicos refere-se à implementação **sem** o controlador.

```
(a) decodificador 16X1
process(dstA)
   begin
    case dstA (3 downto 0) is
      when d0 =>     busA <=  reg0;
      when d1 =>     busA <=  reg1;
      ...
      when d14=>     busA <=  reg10;
      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 7 - Soluções para implementar o banco de registradores.

Para justificar a implementação estrutural, implementou-se o mesmo banco de registradores utilizando-se um *array* de 16 palavras de 16 bits. A Figura 8 mostra como é feita a escolha de qual registrador vai ser lido. Esta implementação tem código VHDL muito mais compacto, porém no momento da síntese obteve-se resultados com ocupação de 700 CLBs.

```
architecture bcregs of bcregs is
  type banco is array (0 to 15) of reg16;
  signal rgt: banco;
  signal destA, destB: reg4;
 begin

 -- processo de escrita...

 -- leitura para o barramento A
 destA <= rgt(15)(3 downto 0) when rregA='1'
          else "1111";
 busA <=   rgt( CONV_INTEGER(destA) );

 -- leitura para o barramento B  ...
```

Figura 8 - Banco de registradores implementado como array.

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.

3.2.2 Registrador SP

O diagrama de blocos do registrador "*stack-pointer*" (apontador de pilha) **SP** é apresentado na Figura 9. Este registrador é responsável pelo armazenamento do endereço do topo da pilha, sendo utilizado na chamada e retorno de subrotinas. Deve ser inicializado a cada programa com a instrução *LDSP* (carrega endereço do topo da pilha). Ele é "auto-incrementável" e "auto-decrementável".

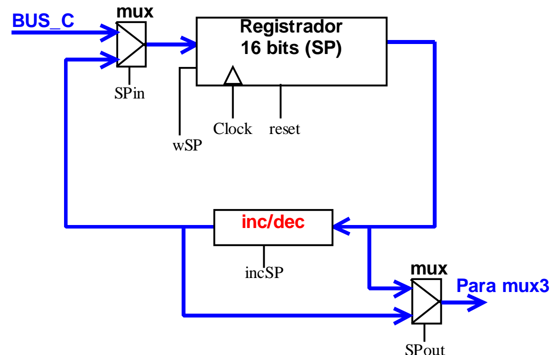


Figura 9 - Registrador de topo de pilha.

Sua implementação em VHDL está apresentada na Figura 10. 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.

```
library IEEE;
use IEEE.Std_Logic_1164.all;
use IEEE.Std_Logic_unsigned.all;
use work.r3.all;

entity reg16sp is
    port( clock, reset, wSP, incsp, spin, spout: in std_logic; D: in reg16; Q: out reg16 );
end reg16sp;

architecture reg16sp of reg16sp is
    signal cout, coutant, c0: std_logic;
    signal SSP, um, incdecsp: reg16; --Saida do SP
begin
    incdecsp <= ssp+1 when (incsp = '1')
                else ssp-1;
    process (clock, reset, wSP)
        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;
end reg16sp;
```

Figura 10 - Implementação VHDL do registrador SP.

3.2.3 Registrador PC

O registrador "Program Counter" (Contador de Programa) **PC** é responsável por armazenar o endereço da posição de memória da instrução corrente. Possui capacidade de "auto-incremento", ou seja, pode ser incrementado sem passar pela ULA.

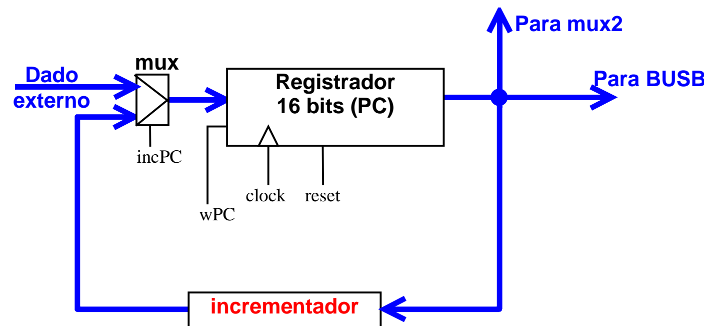


Figura 11 - Registrador PC.

A Figura 12 apresenta o código fonte VHDL do registrador PC.

```
library IEEE;
use IEEE.Std_Logic_1164.all;
use IEEE.Std_Logic_unsigned.all;
use work.r3.all;

entity reg16pc is
    port( clock, reset, ce, incPC: in std_logic; D: in reg16; outPC: out reg16 );
end reg16pc;

architecture reg16pc of reg16pc is
    signal sq: reg16;
begin
    process (clock, reset, ce)
            begin
            if reset = '1' then
                    sq <= (others => '0');
            elsif clock'event and clock = '0' then
                    if ce = '1' then
                            if incPC = '0' then
                                    sq <= D;
                            else
                                    sq <= sq + 1;
                            end if;
                    end if;
            end if;
    end process;

outPC <= sq;  -- Q e um sinal de SAIDA

end reg16pc;
```

Figura 12 - Código fonte VHDL do registrador PC.

3.2.4 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 realizável a implementação em hardware.

A Figura 13 apresenta o código VHDL da ULA. A microoperação enviada à ULA pelo bloco de controle é denominada *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*.

```
--
-- a saída da ula é função da operação realizada entre os barramentos A e B, em função
-- da micro-instrução proveniente do bloco de controle
--
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;

--
-- definição das entradas e vai um do somador
--
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';

--
-- SOMADOR UTILIZADO PARA IMPLEMENTAR DIVERSAS FUNÇÕES DA UNIDADE LOGICO ARITMÉTICA
--
somaAB( input_adder1, input_adder2, cin, output_adder, coutant, cout);

-- em função de cout, coutant,output_adder gerar os qualificadores
```

Figura 13 - Implementação da ULA.

A Tabela 4 apresenta as instruções realizáveis pelo somador *somaAB*, presente na ULA. Este somador executa 9 das 16 funções da ULA . A função do somador é também gerar os sinais de *vai um* (cout) e *vai um anterior* (coutant), para o cálculo dos flags de *carry* e *overflow*.

OPERAÇÃO	INPUT1	INPUT2	CIN
Neg A	~A	0000	1
Not A	~A	0000	0
Passa A	A	0000	0
Passa B	0000	B	0
Extensão de Sinal + B	Aext	B	0
A sub B	A	~B	1
Inc A	A	0000	1
Dec A	A	FFFF	0
Operação Default	A	B	0

Tabela 4 - Instruções codificadas pelo somador.

3.3 Validação do Processador R3

Utilizando o simulador funcional do Active-HDL e um test-bench desenvolvido especificamente para o processador R3, foram realizados inúmeros testes simples e executados alguns programas exemplo. Esta fase teve por objetivos testar: (i) a estrutura do banco de registradores (multiplexadores, decodificador); (ii) a estrutura que controla o endereço de acesso a pilha, o apontador de pilha (multiplexador, decodificador, incrementador/decrementador); (iii) a estrutura do contador de programa auto-incrementável (multiplexador, decodificador); (iv) as operações da ULA; (v) as instruções; (vi) a estrutura geral do bloco de controle; (vii) o funcionamento geral do processador bem como a interação com o controlador externo, memória e dispositivos da plataforma.

O test_bench para a simulação desempenha as seguintes funções:

- instancia o processador;
- gera os sinais de controle de temporização (clock e reset);
- simula o botão externo *spare*, a qual comanda a máquina de estados de controle (ver Seção 4);
- implementa os processos de interface com a memória para leitura e escrita;
- 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 - ver Figura 2).

A Figura 14 apresenta um exemplo de simulação de alguns ciclos do Processador R3.

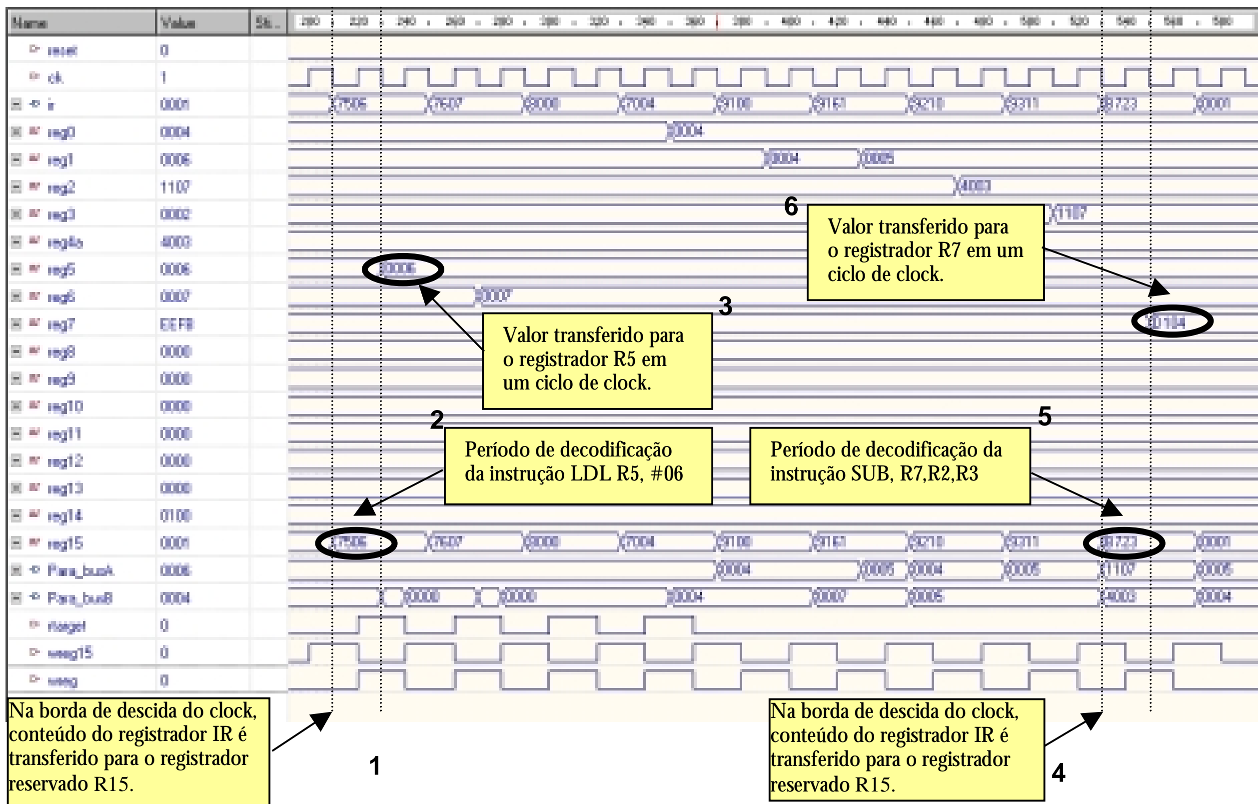


Figura 14 - Janela de simulação do Processador R3.

O *test_bench* lê o programa em código objeto gerado como saída do ambiente de desenvolvimento de software.

O arquivo com o código objeto contém *n* linhas, uma para cada instrução do programa. Cada linha é codificada com 9 caracteres, no formato “xxxx yyyy”, onde xxxx é o endereço com 16 bits (4 dígitos hexadecimais) e yyyy é a instrução com 16 bits (4 dígitos hexadecimais). A memória é endereçada a palavra, e não a byte. Este arquivo deve ser carregado na memória quando o reset é ativado, no início da simulação

O anexo 8.1 apresenta o código objeto para o programa de classificação *bubblesort* e o anexo 8.2 para o programa de classificação *quicksort*.

4 Controle do processador na placa XS40

O ambiente de prototipação consiste de: (*i*) uma placa XS40 que abriga o FPGA Xilinx da família XC4010E com 400 blocos lógicos configuráveis, microcontrolador 80C51 da Intel (não utilizado no projeto), memória RAM de 64K, clock externo de 12MHZ; (*ii*) uma placa XStend contendo dip-switches, leds, displays 7-segmentos e comunicação paralela com o PC.

A Figura 15 mostra a plataforma de prototipação formada pelo conjunto XS40/Xstend.

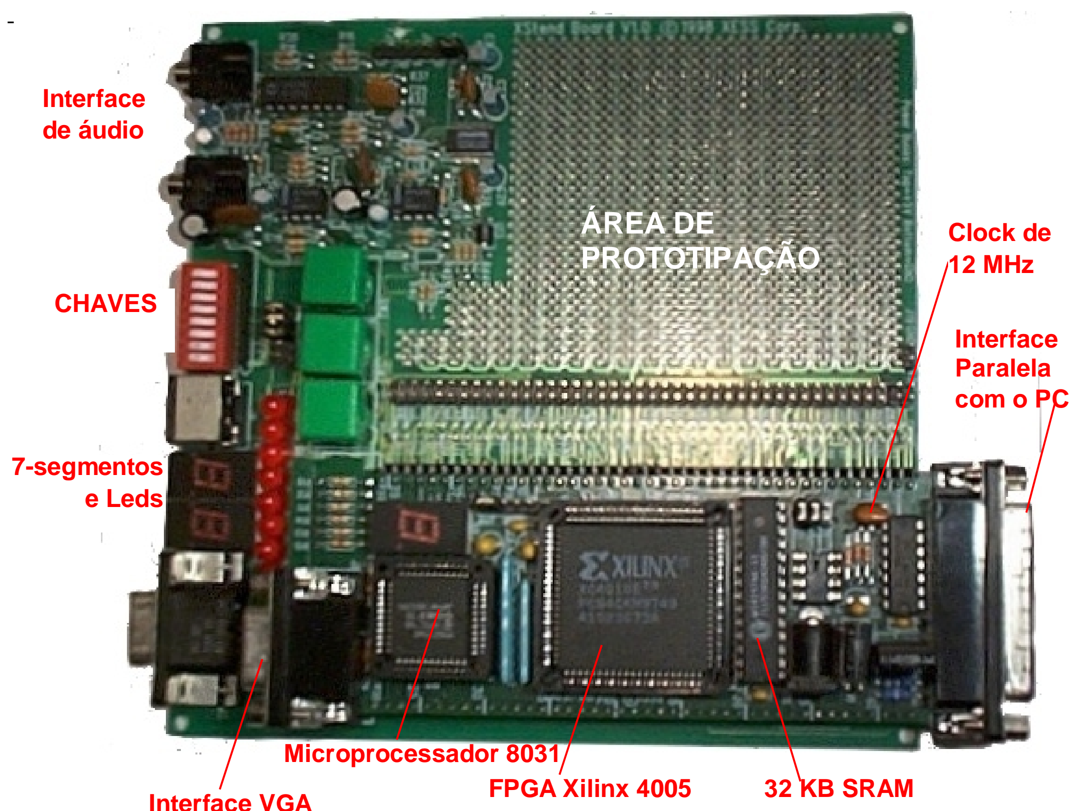


Figura 15- Plataforma de prototipação utilizada no projeto.

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. A Figura 16 ilustra a multiplexação dos sinais de endereços e de dados com os dispositivos de entrada e saída empregados neste projeto.

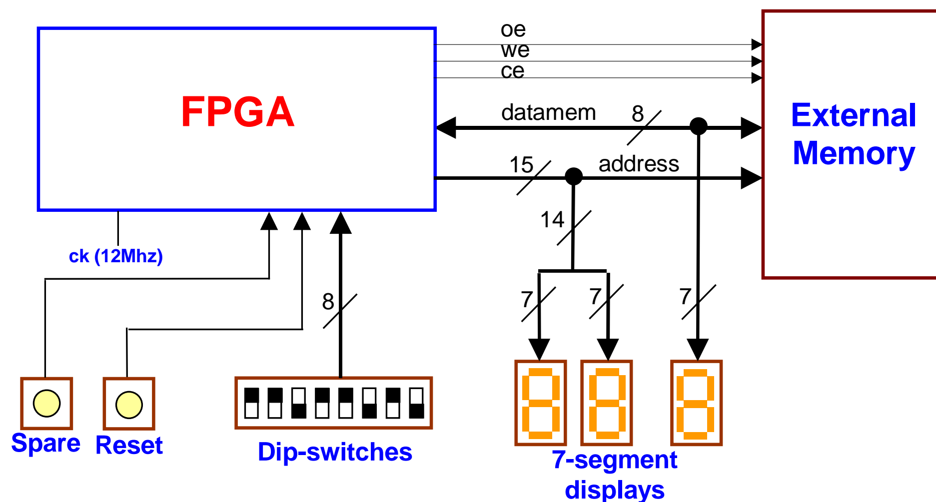
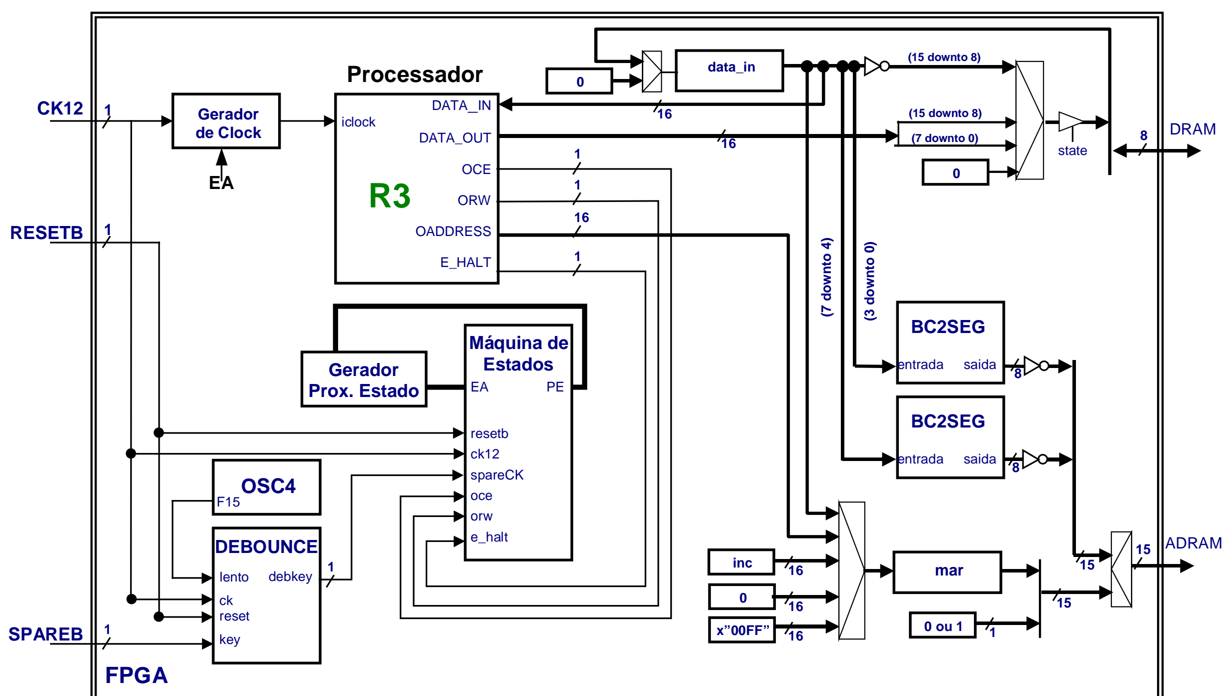


Figura 16 - Multiplexação dos barramentos de endereços e dados.

O módulo controlador, ilustrado na Figura 17, é responsável pela comunicação entre o núcleo processador R3 e os componentes que fazem a interface com os dispositivos da plataforma de prototipação, tais como: clock, botões, leds, displays de 7 segmentos e a memória RAM. Este módulo foi implementado, junto com o núcleo processador R3, no FPGA XC4010E.



(observação: os sinais de acesso à memória - rw, ow, ce não estão mostrados)

Figura 17 - Diagrama em blocos do controlador.

Descreve-se agora as partes do controlador:

- CPU: processador R3, responsável pelo processamento do programa colocado em memória no momento do download do projeto;
- Debounce: circuito que tem for função: (i) realizar a filtragem na captura de eventos

(pressionamento) das teclas *resetb* e *spareb* da plataforma de prototipação; (ii) realizar a sincronização do botão *spareb* (sinal assíncrono e "sujo") ao *clock*, gerando-se um único pulso após a primeira borda de descida em *spareb*;

- Bcd2Seg: circuito que realiza a conversão de 4 dígitos binários para o correspondente em 7 segmentos a serem apresentados nos mostradores 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 interno para o processador. A largura dos pulsos do clock interno, bem como seu ciclo de serviço são variáveis, conforme é mostrado na Figura 18, sendo o clock interno congelado após ativação do sinal *e_halt* pela R3.

Conforme mencionado acima, o funcionamento do controlador depende do estado corrente 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 *SPARE.*

No estado seguinte, **S1**, é enviado um pulso de *reset* ao processador, iniciando-se a fase de processamento. Os estados seguintes correspondem às fases de escrita e leitura na memória (estados **SW1**, **SW2**, **SRD1** e **SRD2**). A utilização de dois estados para escrita e leitura advém do fato de que o processador trabalha com 16 bits de dados e o barramento da plataforma é de 8 bits. O endereço fornecido pelo processador é deslocado à esquerda um bit, sendo o primeiro ciclo de leitura/escrita com 0 no bit menos significativo e o seguinte com 1 (ver Figura 18, ciclo de escrita).

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. Ainda neste estado, o controlador busca em um dado endereço (atualmente a posição de memória 100H) a posição inicial da área de dados do programa, armazenando-o no registrador **data_in**. No estado seguinte, **S3**, exibe-se no displays o endereço da área de dados, transfere-se o conteúdo de data_in para o registrador **mar** e aguarda-se por evento no botão *SPARE.*

Após pressionar-se o botão *SPARE,* o dado apontado pelo endereço da área de dados é lido. Passa-se para o **S4**, onde os dados lidos são exibidos (parte baixa nos *displays* de sete-segmentos e parte alta nos *leds*). Fica-se neste estado até pressionamento do botão *SPARE.* Após evento em *SPARE*, incrementa-se o registrador **mar**, passa-se para **S5**, iniciando-se outro ciclo de leitura de dados.

O controlador fica lendo-exibindo resultados da área de dados, até a pressão de *RESETB*.

A Figura 18 ilustra a execução de 3 instruções neste processador, sobre a plataforma de prototipação. **Observa-se 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 e ldl) 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 R3 é transferido para *data_out*. O conteúdo de R4 é D1H, é deslocado para a esquerda tornando-se 1A2H e 1A3H, conforme *adram*.

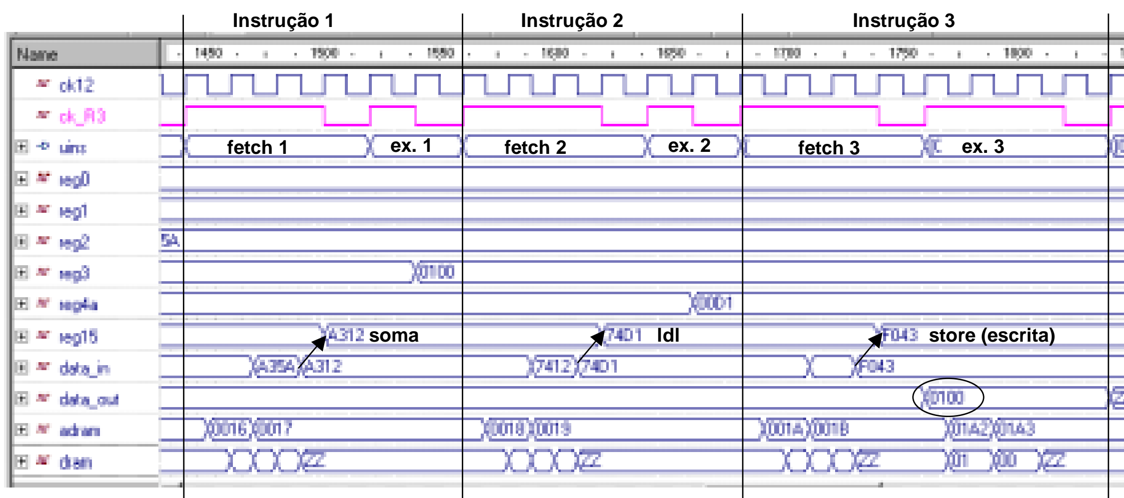


Figura 18 - Simulação do circuito completo, estado de funcionamento S1.

Uma vez o sistema processador e controlador validado (via simulação de diversos programas de teste, tais como rotinas de ordenamento *bubble* e *quicksort*), a fase posterior do trabalho consiste na sua implementação efetiva na placa 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. Uma conclusão importante deste trabalho foi que código VHDL simulável não significa necessariamente código sintetizável.

4.1 Acesso à RAM

Durante a fase de processamento, o processador tem controle sobre o acesso à memória. Quem provê os mecanismos de acesso é o controlador da arquitetura. Os endereços de memória gerados pelo processador são armazenadas no registrador *mar* durante a fase de processamento e mapeados para o sinal externo *adram*. Os conteúdos são mapeados para o sinal *dram* em duas palavras de 8 bits, portanto são necessários dois ciclos para armazenamento na memória. Os conteúdos lidos são armazenados no registrador *data_in* com sua saída mapeada para o sinal de mesmo nome no processador.

Observa-se uma limitação na arquitetura que advém da incompatibilidade de tamanho do barramento de acesso a memória da plataforma (8 bits) com o tamanho da palavra com que o processador trabalha (16 bits). Isso faz com que haja perda de desempenho do processador quando implementado sobre a plataforma XS40/XStend. Para o processador são necessários apenas dois ciclos de clock para execução de uma instrução. Isto não é suportado pela plataforma, que precisa acessar a memória em duas etapas. A fase de busca, por exemplo, consome **sempre** 3 ciclos de *clock* de referência (12 MHz), pois são necessários 2 acessos à memória e uma atualização de endereço. Também são necessários 3 ciclos para a fase de execução, caso seja feito um acesso à memória (load/store).

Ciclo de leitura de uma palavra de 16 bits

A Figura 19 mostra a simulação relativa aos ciclos de leitura na memória.

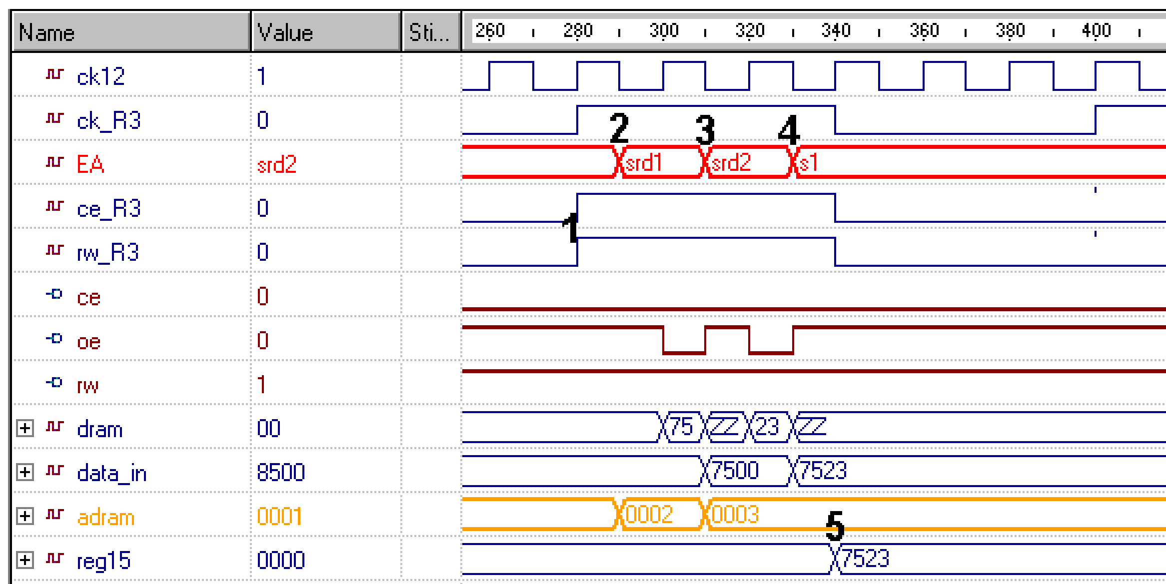


Figura 19 - Passos de execução em 1 ciclo de leitura.

- Passo 1: na borda de subida do processador (**ck_r3**) os sinais de acesso à memória do processador sobem (**ce_R3** e **rw_R3**)
- Passo 2: na primeira descida subsequente do clock mestre (**ck12**) aos comandos **ce_r3** e **rw_r3**

há a passagem para o estado de controle **srd1**. Na primeira borda de subida do clock mestre, os sinais **ce** e **oe** vão à 0, e o endereço da memória recebe o endereço do processador (deslocado à esquerda com concatenação de 0), permitindo assim que se faça a leitura da primeira palavra memória (sinal **dram**, proveniente da memória).

- Passo 3: ao final do primeiro ciclo de leitura passa-se para o estado de controle **srd2**, onde se armazena os 8 bits lidos (**dram**) em **data_in**, coloca-se novamente **ce** e **oe** em 1 (repouso) e o endereço da memória recebe o endereço do processador (deslocado à esquerda com concatenação de 1), permitindo assim que se faça a leitura do byte menos significativo. Em seguida, no momento da primeira borda de subida do clock mestre os sinais **ce** e **oe** vão para 0, permitindo assim que se faça a leitura da segunda palavra de memória (sinal **dram**, proveniente da memória).
- Passo 4: ao final do segundo ciclo de leitura armazena-se a parte baixa da palavra em **data_in** (palavra completa de 16 bits).
- Passo 5: na borda de descida do processador (ck_r3) armazena-se no reg15 (registrador de instrução corrente) o valor da instrução lida.

Ciclo de escrita de uma palavra de 16 bits

A Figura 20 mostra a simulação relativa aos ciclos de escrita na memória.

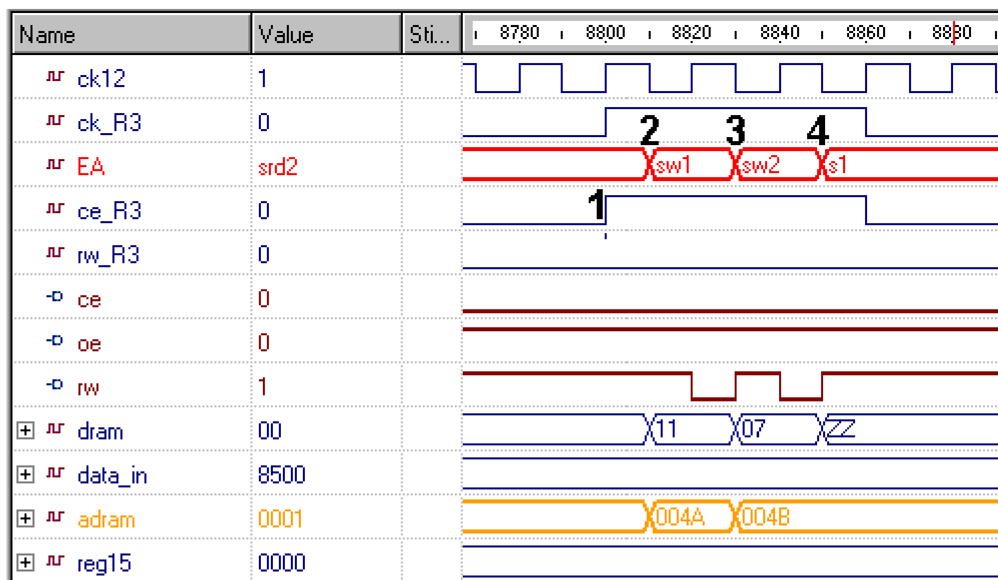


Figura 20 - Passos de execução em 1 ciclo de escrita.

- Passo 1: na borda de subida do processador (**ck_r3**) os sinais de acesso à memória do processador são ativados (**ce_R3** =1 e **rw_R3**=0)
- Passo 2: na primeira descida do clock mestre (**ck12**) subsequente aos comandos **ce_r3** e **rw_r3** o endereço da memória recebe o endereço do processador (deslocado à esquerda com concatenação de 0) e na primeira borda de subida do clock mestre, os sinais **ce** e **rw** vão para 0,

permitindo assim que se faça a escrita da memória (coloca-se o byte mais significativo de **data_out** em **dram**).

- Passo 3: no próximo estado de escrita os sinais **ce** e **rw** são colocados novamente em 1 (repouso) e o endereço da memória recebe o endereço do processador (deslocado à esquerda com concatenação de 1), e na primeira borda de subida do clock mestre, os sinais **ce** e **rw** vão para 0, permitindo assim que se faça a escrita da memória (coloca-se o byte mais significativo de **data_out** em **dram**).
- Passo 4: escrita encerrada.

5 Implementação Física

Como ferramenta de síntese foi utilizado o pacote Foundation nas versões 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 muitas outras ferramentas.

Os resultados preliminares obtidos indicaram uma taxa de utilização de blocos lógicos (CLBs) e linhas de roteamento superior ao disponível para implementar a arquitetura completa (processador R3 e controlador) no FPGA Xilinx da placa de prototipação XS40/XSTEND [2] (XC4010E, 400 CLBs). Para implementar uma versão funcional da arquitetura, composta pelo conjunto processador R3 e controlador, se fez necessária a supressão de 4 registradores de um total de 16 e 6 instruções de um total de 39, reduzindo-se assim a quantidade de CLBs ao número de 351, tornando a síntese física possível.

A **Erro! A origem da referência não foi encontrada.** mostra o relatório (parcial) de síntese física para uma versão da arquitetura com 12 registradores, ao invés dos 16 da especificação. 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.*

Uma vez gerado o arquivo binário de configuração do FPGA, passou-se à fase de testes na placa de prototipação. Para isto, no momento do *download* envia-se primeiro o programa para a memória e depois o arquivo de configuração. Foram executados programas de diferentes complexidades, e todos apresentaram o comportamento previsto na simulação.

```
 Design Summary:
  Number of errors:        0
  Number of warnings:      4
  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%
IOB Flops:            0
IOB Latches:          0

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 21 - Relatório de síntese física (12 registradores).

Como ferramenta de validação do protótipo, foi utilizado um analisador lógico HP1663E, para monitoramento dos linhas de endereço, dados e controle. A Figura 22 ilustra uma janela do analisador lógico, onde monitora-se uma leitura no endereço 000FH e a ativação do sinal “*e-halt*”.

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 a nível de simulação quanto da placa de prototipação.

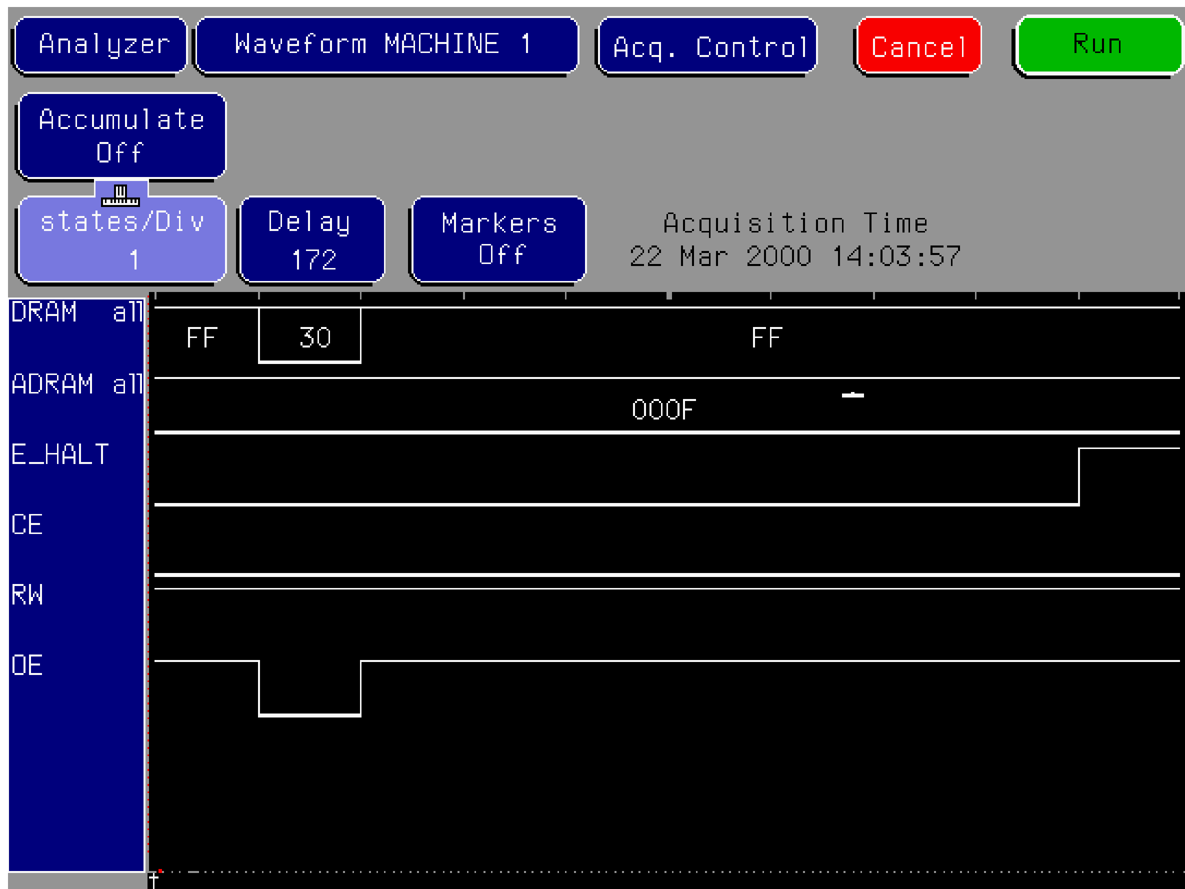


Figura 22 - Janela do analisador lógico HP1663E.

6 Conclusão, estado atual e futuro

Este trabalho apresentou o desenvolvimento completo de uma arquitetura com os passos de:

- especificação da arquitetura;
- uso de simulador e tradutor de linguagem de montagem específicos para a arquitetura;
- simulação funcional em VHDL;
- implementação VHDL sintetizável;
- síntese lógica e física;
- implementação física em plataforma de prototipação;
- validação do hardware com analisador lógico.

Dados os recursos limitados disponíveis para a implementação do projeto, 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. Abaixo são citados os trabalhos em andamento atualmente, bem como os trabalhos relacionados a serem iniciados a curto e médio prazo no PPGCC (Programa de Pós-Graduação em Ciência da Computação da Faculdade de Informática) da PUCRS.

Como trabalhos em desenvolvimento no momento, citam-se:

- inserção de técnicas 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;
- 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;
- migração do processador e controlador para a plataforma Virtex [4] (maior capacidade em termos de blocos lógicos – 300.000 portas equivalentes).

Como trabalhos futuros, citam-se:

- geração do código assembly à partir de um compilador C, via uso de compiladores reconfiguráveis tais como *lcc* [10];
- implementação de diferentes partições hardware/software;
- avaliação de desempenho destas partições e definição de critérios que auxiliem na partição de novos circuitos;
- integração do processador desenvolvido a plataforma conectável a barramento de alto desempenho, tal como PCI. Plataforma esta em aquisição.

---

7 Referências Bibliográficas

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

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

[3] Rolf Ernst. "*Codesign 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] 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.
Available in: ftp://ftp.inf.pucrs.br/pub/moraes/papers/99fie.pdf

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

[7] http://www.xilinx.com/products/xc4000XLA.html. Describes de XC4000 Family of FPGAs.

[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.

8 Anexos – Programas Exemplo

8.1 Programa BubbleSort

```
0000 8500    ;LOAD R5,#0023H
0001 7523    ;LOAD R5,#0023H
0002 8600    ;LOAD R6,#0023H
0003 7623    ;LOAD R6,#0023H
0004 8000    ;LOAD R0,#0022H
0005 7022    ;LOAD R0,#0022H
0006 9010    ;LD R0,R0
0007 9070    ;DEC R0,R0
0008 A606    ;ADD R6,R6,R0
0009 A505    ;ADD R5,R5,R0
000A 9575    ;DEC R5,R5
000B 8000    ;LOAD R0,#0023H
000C 7023    ;LOAD R0,#0023H
000D 9100    ;MOV R1,R0
000E 9161    ;INC R1,R1
000F 9210    ;LD  R2,R0
0010 9311    ;LD  R3,R1
0011 B423    ;SUB R4,R3,R2
0012 0001    ;JN  #0014H
0013 4005    ;JMP #0019H
0014 9402    ;MOV R4,R2
0015 9203    ;MOV R2,R3
0016 9304    ;MOV R3,R4
0017 F002    ;ST  R0,R2
0018 F013    ;ST  R1,R3
0019 B316    ;SUB R3,R6,R1
001A 1002    ;JZ  #001DH
001B 9161    ;INC R1,R1
001C 4FF2    ;JMP #000FH
001D B305    ;SUB R3,R5,R0
001E 1002    ;JZ  #0021H
001F 9060    ;INC R0,R0
0020 4FEC    ;JMP #000DH
0021 6230    ;HLT
0022 0007    ;DW  #0007H
0023 4003    ;DW  #4003H, #1107H, #0002H , #5206H , #7011H, #2000H, #0005H
0024 1107    ;DW  #4003H, #1107H, #0002H , #5206H , #7011H, #2000H, #0005H
0025 0002    ;DW  #4003H, #1107H, #0002H , #5206H , #7011H, #2000H, #0005H
0026 5206    ;DW  #4003H, #1107H, #0002H , #5206H , #7011H, #2000H, #0005H
0027 7011    ;DW  #4003H, #1107H, #0002H , #5206H , #7011H, #2000H, #0005H
0028 2000    ;DW  #4003H, #1107H, #0002H , #5206H , #7011H, #2000H, #0005H
0029 0005    ;DW  #4003H, #1107H, #0002H , #5206H , #7011H, #2000H, #0005H
```

8.2 Programa QuickSort

```
0000 400F    ;JMP #0010H
0001 0006    ;DW  #0006H,#0044H,#0015H,#0032H,#0009H,#0033H,#0011H,#0021H
0002 0044    ;DW  #0006H,#0044H,#0015H,#0032H,#0009H,#0033H,#0011H,#0021H
0003 0015    ;DW  #0006H,#0044H,#0015H,#0032H,#0009H,#0033H,#0011H,#0021H
0004 0032    ;DW  #0006H,#0044H,#0015H,#0032H,#0009H,#0033H,#0011H,#0021H
0005 0009    ;DW  #0006H,#0044H,#0015H,#0032H,#0009H,#0033H,#0011H,#0021H
0006 0033    ;DW  #0006H,#0044H,#0015H,#0032H,#0009H,#0033H,#0011H,#0021H
0007 0011    ;DW  #0006H,#0044H,#0015H,#0032H,#0009H,#0033H,#0011H,#0021H
0008 0021    ;DW  #0006H,#0044H,#0015H,#0032H,#0009H,#0033H,#0011H,#0021H
0009 0004    ;DW  #0004H,#0008H,#0003H,#0022H,#0005H,#0014H,#0002H
000A 0008    ;DW  #0004H,#0008H,#0003H,#0022H,#0005H,#0014H,#0002H
000B 0003    ;DW  #0004H,#0008H,#0003H,#0022H,#0005H,#0014H,#0002H
000C 0022    ;DW  #0004H,#0008H,#0003H,#0022H,#0005H,#0014H,#0002H
000D 0005    ;DW  #0004H,#0008H,#0003H,#0022H,#0005H,#0014H,#0002H
000E 0014    ;DW  #0004H,#0008H,#0003H,#0022H,#0005H,#0014H,#0002H
000F 0002    ;DW  #0004H,#0008H,#0003H,#0022H,#0005H,#0014H,#0002H
0010 8000    ;LOAD R0,#0002H
0011 7002    ;LOAD R0,#0002H
0012 8100    ;LOAD R1,#009DH
0013 719D    ;LOAD R1,#009DH
0014 8200    ;LOAD R2,#00ADH
0015 72AD    ;LOAD R2,#00ADH
0016 8900    ;LOAD R9,#00BDH
0017 79BD    ;LOAD R9,#00BDH
0018 8E01    ;LOAD R14,#0100H
0019 7E00    ;LOAD R14,#0100H
001A 6209    ;LDSP R9
001B 8300    ;LOAD R3,#0001H
001C 7301    ;LOAD R3,#0001H
001D 8400    ;LOAD R4,#000FH
001E 740F    ;LOAD R4,#000FH
001F 5001    ;JSR #0021H
0020 6230    ;HLT
0021 F0E3    ;PUSH R3
0022 9E7E    ;PUSH R3
0023 F0E4    ;PUSH R4
0024 9E7E    ;PUSH R4
0025 501F    ;JSR #0045H
0026 F0E7    ;PUSH R7
0027 9E7E    ;PUSH R7
0028 F0E6    ;PUSH R6
0029 9E7E    ;PUSH R6
002A BD37    ;SUB R13,R7,R3
002B 0007    ;JN  #0033H
002C 1006    ;JZ  #0033H
002D F0E4    ;PUSH R4
002E 9E7E    ;PUSH R4
002F 9407    ;MOV R4,R7
0030 5FF0    ;JSR #0021H
0031 9E6E    ;POP R4
0032 941E    ;POP R4
```

9 Anexos – Fontes VHDL do Processador

9.1 Fontes do Processador

```
------------------------------------------------------------------------
-- R3 versao implementada na plataforma xs40/xstend
------------------------------------------------------------------------

library IEEE;
use IEEE.Std_Logic_1164.all;

package r3 is
        subtype reg16  is std_logic_vector(15 downto 0);
        subtype reg15  is std_logic_vector(14 downto 0);
        subtype reg8   is std_logic_vector(7 downto 0);
        subtype reg5   is std_logic_vector(4 downto 0);
        subtype reg4   is std_logic_vector(3 downto 0);

        type mem1 is array (0 to 65534) of std_logic_vector(15 downto 0);
        constant TAMaddress : integer := 65534;

        type microinstrucao is record
                rtarget: std_logic;                  -- LeRtarget na BUS B, para LDL e LDH
                wreg15: std_logic;                   -- Gravar no registrador 15 (para Fetch)
                rreg15: std_logic;                   -- Ler registrador 15 na bus A
                ula: reg5;                           -- Operacao da ULA
                rregb: std_logic;                    -- Tristate para bus B
                wreg: std_logic;                     -- Escrever ou nao nos registradores(enable do decodificador)
                ce,rw: std_logic;                    -- Chip enable e R_Wnegado
                nz, c ,v: std_logic;                 -- Load Flags de estado
                incPC,wPC : std_logic;               -- Controles do reg PC
                Spin,wSp,incSP,Spout: std_logic;     -- Controles do reg SP
                mux2: std_logic;                     -- Controle do mux2 (escreve bus A ou pc)
                mux3: std_logic;                     -- Controle do mux3 (endereco da ula ou do SP)
        end record;

        constant d0        : reg4 := "0000";         constant d1          : reg4 := "0001";
        constant d2        : reg4 := "0010";         constant d3          : reg4 := "0011";
        constant d4        : reg4 := "0100";         constant d5          : reg4 := "0101";
        constant d6        : reg4 := "0110";         constant d7          : reg4 := "0111";
        constant d8        : reg4 := "1000";         constant d9          : reg4 := "1001";
        constant d10       : reg4 := "1010";         constant d11         : reg4 := "1011";
        constant d12       : reg4 := "1100";         constant d13         : reg4 := "1101";
        constant d14       : reg4 := "1110";         constant d15         : reg4 := "1111";

        constant soma      : reg5 := "00000";        constant incA        : reg5 := "00001";
        constant notA      : reg5 := "00010";        constant AsubB       : reg5 := "00011";
        constant PassaB    : reg5 := "00100";        constant Ou          : reg5 := "00101";
        constant E         : reg5 := "00110";        constant PassaA      : reg5 := "00111";
        constant OuX       : reg5 := "01000";        constant decA        : reg5 := "01001";
        constant negA      : reg5 := "01010";        constant ext_sinal   : reg5 := "01011";
        constant cte_low   : reg5 := "01100";        constant cte_high    : reg5 := "01101";
        constant sl0       : reg5 := "01110";        constant sr0         : reg5 := "01111";
        constant sl1       : reg5 := "10000";        constant sr1         : reg5 := "10001";

        procedure somaAB  (  signal A, B: in  reg16;  signal Cin: in STD_LOGIC;  signal S: out reg16;  signal CoutAnt, Cout: out STD_LOGIC);
end r3;

package body r3 is
        ------------------------------------------------------------------------
        -- Somador 16 bits
        ------------------------------------------------------------------------
        procedure somaAB  (  signal A, B: in  reg16;  signal Cin: in STD_LOGIC;  signal S: out reg16;  signal CoutAnt, Cout: out STD_LOGIC) is
                variable carry : STD_LOGIC;
        begin
                for w in 0 to 15 loop
                        if w=0 then carry:=Cin;  end if;
                        S(w) <= A(w) xor B(w) xor carry;
                        carry := (A(w) and B(w)) or (A(w) and carry) or (B(w) and carry);
                        if w=14 then CoutAnt <= carry;   end if;
                end loop;
                Cout <= carry;
        end somaAB;
end r3;

------------------------------------------------------------------------
-- Registrador de uso geral
------------------------------------------------------------------------
library IEEE;
use IEEE.Std_Logic_1164.all;
use work.r3.all;

entity reg16clear is
        port( clock,reset,ce:in std_logic;   D:in reg16;  Q:out reg16 );
end reg16clear;

architecture reg16clear of reg16clear is
begin
        process (clock, reset, ce)
                begin
                if RESET = '1' then
                        Q <= (others => '0');
                elsif clock'event and clock = '0' then
                        if ce = '1' then  Q <= D;     end if;
                end if;
        end process;
end reg16clear;

------------------------------------------------------------------------
-- Registrador PC
------------------------------------------------------------------------
```

```
library IEEE;
use IEEE.Std_Logic_1164.all;
use IEEE.Std_Logic_unsigned.all;
use work.r3.all;

entity reg16pc is
        port( clock, reset, ce, incPC: in std_logic; D: in reg16; outPC: out reg16 );
end reg16pc;

architecture reg16pc of reg16pc is
        signal sq: reg16;
begin
        process (clock, reset, ce)
                begin
                if reset = '1' then
                        sq <= (others => '0');
                elsif clock'event and clock = '0' then
                        if ce = '1' then
                                if incPC = '0' then sq <= D;    else sq <= sq + 1;  end if;
                        end if;
                end if;
        end process;

        outPC <= sq;  -- Q e um sinal de SAIDA

end reg16pc;

------------------------------------------------------------------------
-- Registrador SP
------------------------------------------------------------------------
library IEEE;
use IEEE.Std_Logic_1164.all;
use IEEE.Std_Logic_unsigned.all;
use work.r3.all;

entity reg16sp is
        port( clock, reset, ce, incsp, spin, spout: in std_logic; D: in reg16; Q: out reg16 );
end reg16sp;

architecture reg16sp of reg16sp is
        signal cout, coutant, c0: std_logic;
        signal SSP, um, incdecsp: reg16; --Saida do SP (ver folha)
begin
        incdecsp <= ssp+1 when (incsp = '1')
        else ssp-1;
        process (clock, reset, ce)
                begin
                if RESET = '1' then
                        ssp <= (others => '0');
                elsif clock'event and clock = '0' then
                        if ce = '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;
end reg16sp;

------------------------------------------------------------------------
-- Banco de registradores (R0 .. R15)
------------------------------------------------------------------------
library IEEE;
use IEEE.Std_Logic_1164.all;
use work.r3.all;
entity bcregs is
        port( reset, ck: std_logic;  datamux: in reg16;  wreg15, wreg: in std_logic;  rtarget: in std_logic;  Para_busA, Para_busB, ir: out reg16);
end entity;

architecture bcregs of bcregs is
        component reg16clear      port( clock, reset, ce: in std_logic; D: in reg16; Q: out reg16 ); end component;

        signal reg0, reg1,  reg2,  reg3,  reg4a, reg5,  reg6, reg7, reg8,  reg9, reg10, reg11, reg12, reg13, reg14, reg15: reg16;
        signal wenables: std_logic_vector  (14 downto 0);
        signal destB, destA: reg4;
  begin
        r0:  reg16clear port map(clock=>ck,reset=>reset,ce=>wenables(0) ,d=>datamux,q=>reg0 );
        r1:  reg16clear port map(clock=>ck,reset=>reset,ce=>wenables(1) ,d=>datamux,q=>reg1 );
        r2:  reg16clear port map(clock=>ck,reset=>reset,ce=>wenables(2) ,d=>datamux,q=>reg2 );
        r3:  reg16clear port map(clock=>ck,reset=>reset,ce=>wenables(3) ,d=>datamux,q=>reg3 );
        r4:  reg16clear port map(clock=>ck,reset=>reset,ce=>wenables(4) ,d=>datamux,q=>reg4a );
        r5:  reg16clear port map(clock=>ck,reset=>reset,ce=>wenables(5) ,d=>datamux,q=>reg5 );
        r6:  reg16clear port map(clock=>ck,reset=>reset,ce=>wenables(6) ,d=>datamux,q=>reg6 );
        r7:  reg16clear port map(clock=>ck,reset=>reset,ce=>wenables(7) ,d=>datamux,q=>reg7 );
        r8:  reg16clear port map(clock=>ck,reset=>reset,ce=>wenables(8) ,d=>datamux,q=>reg8 );
        r9:  reg16clear port map(clock=>ck,reset=>reset,ce=>wenables(9) ,d=>datamux,q=>reg9 );
        r10: reg16clear port map(clock=>ck,reset=>reset,ce=>wenables(10),d=>datamux,q=>reg10);
        r11: reg16clear port map(clock=>ck,reset=>reset,ce=>wenables(11),d=>datamux,q=>reg11);
        r12: reg16clear port map(clock=>ck,reset=>reset,ce=>wenables(12),d=>datamux,q=>reg12);
        r13: reg16clear port map(clock=>ck,reset=>reset,ce=>wenables(13),d=>datamux,q=>reg13);
        r14: reg16clear port map(clock=>ck,reset=>reset,ce=>wenables(14),d=>datamux,q=>reg14);
        r15: reg16clear port map(clock=>ck,reset=>reset,ce=>wreg15,d=>datamux,q=>reg15);

        ir <= reg15;   -- o registrador R15 é reservado e utilizado como IR

        -- seleção de escrita nos registradores
        destA <= reg15(11 downto 8) when wreg='1' else "1111";              -- 15 nunca ocorre!
        wenables(0)  <=  '1' when  destA=d0  else '0';                  wenables(1)  <=  '1' when  destA=d1  else '0';
        wenables(2)  <=  '1' when  destA=d2  else '0';                  wenables(3)  <=  '1' when  destA=d3  else '0';
        wenables(4)  <=  '1' when  destA=d4  else '0';                  wenables(5)  <=  '1' when  destA=d5  else '0';
        wenables(6)  <=  '1' when  destA=d6  else '0';                  wenables(7)  <=  '1' when  destA=d7  else '0';
        wenables(8)  <=  '1' when  destA=d8  else '0';                  wenables(9)  <=  '1' when  destA=d9  else '0';
        wenables(10) <=  '1' when  destA=d10 else '0';                  wenables(11) <=  '1' when  destA=d11 else '0';
        wenables(12) <=  '1' when  destA=d12 else '0';                  wenables(13) <=  '1' when  destA=d13 else '0';
        wenables(14) <=  '1' when  destA=d14 else '0';
```

```
        -- solucao mista, com mux 2x1 ---
        para_busA <=  reg0  when reg15(3 downto 0)=d0        else (others=>'Z');
        para_busA <=  reg1  when reg15(3 downto 0)=d1        else    reg2  when reg15(3 downto 0)=d2        else (others=>'Z');
        para_busA <=  reg3  when reg15(3 downto 0)=d3        else    reg4a when reg15(3 downto 0)=d4        else (others=>'Z');
        para_busA <=  reg5  when reg15(3 downto 0)=d5        else     reg6  when reg15(3 downto 0)=d6       else (others=>'Z');
        para_busA <=  reg7  when reg15(3 downto 0)=d7        else    reg8  when reg15(3 downto 0)=d8        else (others=>'Z');
        para_busA <=  reg9  when reg15(3 downto 0)=d9        else    reg10 when reg15(3 downto 0)=d10       else (others=>'Z');
        para_busA <=  reg11 when reg15(3 downto 0)=d11       else  reg12 when reg15(3 downto 0)=d12         else (others=>'Z');
        para_busA <=  reg13 when reg15(3 downto 0)=d13       else  reg14 when reg15(3 downto 0)=d14         else (others=>'Z');
        para_busA <=  reg15  when reg15(3 downto 0)=d15      else (others=>'Z');

          -- solucao mista, com mux 2x1 ---
        destB <= reg15(7 downto 4) when rtarget = '0'  else reg15(11 downto 8);  --reg15
        Para_busB <= reg0  when destB=d0                     else  (others=>'Z');
        Para_busB <= reg1  when destB=d1                     else   reg2  when destB=d2        else  (others=>'Z');
        Para_busB <= reg3  when destB=d3                     else   reg4a when destB=d4        else  (others=>'Z');
        Para_busB <= reg5  when destB=d5                     else   reg6  when destB=d6        else  (others=>'Z');
        Para_busB <= reg7  when destB=d7                     else   reg8  when destB=d8        else  (others=>'Z');
        Para_busB <= reg9  when destB=d9                     else   reg10 when destB=d10       else  (others=>'Z');
        Para_busB <= reg11 when destB=d11                    else   reg12 when destB=d12       else  (others=>'Z');
        Para_busB <= reg13 when destB=d13                    else   reg14 when destB=d14       else  (others=>'Z');
        Para_busB <= reg15 when destB=d15                    else  (others=>'Z');
end bcregs;

--------------------------------------------------------------------------
--  Descricao estrutural do bloco de dados
--------------------------------------------------------------------------
library IEEE;
use IEEE.Std_Logic_1164.all;
use IEEE.Std_Logic_unsigned.all;
use work.r3.all;

entity datapath is
        port( uins: in microinstrucao;                    ck, reset: in std_logic; ir, endereco : out reg16; datain: in reg16; dataout: out reg16;  n, z, c, v: out std_logic);
end datapath;

architecture datapath of datapath is

        component bcregs          port(ck, reset: std_logic; datamux: in reg16; wreg15, wreg, rtarget: in std_logic; Para_busA, Para_busB, ir: out reg16);  end component;

        component reg16sp         port(clock, reset, ce, spin, spout, incsp: in std_logic; D: in reg16; Q: out reg16 );    end component;

        component reg16pc         port( clock, reset, ce, incPC: in std_logic; D: in reg16; outPC: out reg16 );   end component;

        signal input_adder1, input_adder2, output_adder,  reg15: reg16;
        signal out_ula, mux1, mux2, source1, source2: reg16;
        signal busA, busB, pc, sp: reg16;
        signal cin, cout, coutant: std_logic;
begin

        REGS: bcregs port map(ck=>ck,reset=>reset,datamux=>mux1,wreg=>uins.wreg,rtarget=>uins.rtarget,para_busA=>source1,para_busB=>source2,ir=>reg15,wreg15=>uins.wreg15);
        RPC: reg16pc port map(clock=>ck,reset=>reset,ce=>uins.wpc,incpc=>uins.incpc,d=>mux1, outPC=>pc);
        RSP: reg16sp port map(clock=>ck,reset=>reset,ce=>uins.wsp,spin=>uins.spin,spout=>uins.spout,incsp=>uins.incsp,d=>out_ula,q=>sp);

        ir <= reg15;  -- IR

        busA <= source1 when uins.rreg15 = '0' else reg15;    -- BUS A

        busB <= source2 when uins.rregb = '1' else pc;   -- BUS B

        endereco <= out_ula when uins.mux3 = '0' else sp;       -- ENDERECO para a memória

        mux1 <= datain when uins.ce = '1' and uins.rw = '1' else out_ula; -- MUX 1

        -- DADOS que serao escritos na memoria
        dataout <= busA when uins.mux2='1' and uins.ce='1' and uins.rw ='0' else
              pc   when uins.mux2='0' and uins.ce='1' and uins.rw ='0' else
              (others => 'Z');

        ---  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;        -- IMPORTANTE: A SAIDA DA ULA E' A INSTRUCAO DEFAULT

        --- somador da ULA, utilizado para 9 instrucoes: soma  AsubB  negA  incA  decA  ext_sinal notA passaA passaB   //  ** cin é usado pelo sl e sr
        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);

        -- flip flops para armazenamento dos QUALIFICADORES
        process(ck,reset,uins)
              begin
              if (reset = '1') then
                     c <= '0';  n <= '0';                   z <= '0';  v <= '0';
              elsIf ck'event and ck = '0' then
                     if uins.c = '1' then c <= cout;  end if;
                     if uins.nz = '1' then
                              n <= out_ula(15);
                              if  out_ula="0000000000000000" then z <= '1';  else z<='0'; end if;
                       end if;
```

```
                    if uins.v = '1' then  v <= cout xor coutant;  end if;
               end if;
          end process;
end datapath;

--------------------------------------------------------------------------
--  Descricao comportamental do bloco de controle
--------------------------------------------------------------------------
library IEEE;
use IEEE.Std_Logic_1164.all;
use work.r3.all;
entity controle is
          port (uins: out microinstrucao;    reset,ck: in std_logic;   n,c,z,v:in std_logic;  ir: in reg16;       e_halt: out std_logic );
end controle;

architecture controle of controle is
          signal c1,cz:std_logic;
 begin
   c1 <= '1';
   cz <= '0';

  -- DECODIFICACAO DO HALT, o sinal halt
  -- O 'e-halt' desce sempre que vier um reset.
  process(reset, ck)
   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;

  -- DECODIFICACAO DAS INSTRUCOES, EXCETO O HALT QUE E' DECODIFICADO ACIMA
  process
   begin

     wait until ck'event and ck='1';

     uins <= (cz,c1,cz,passaB,cz,cz,c1,c1,cz,cz,cz,c1,c1,cz,cz,cz,cz,cz,cz);      -- Fetch

     wait until ck'event and ck='1';

     case ir(15 downto 12) is

          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 d1 => if z=c1 then
                        uins <= (cz,cz,c1,ext_sinal,cz,cz,cz,cz,cz,cz,cz,cz,c1,cz,cz,cz,cz,cz,cz); -- JZ 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 d2 => if c=c1 then
                        uins <= (cz,cz,c1,ext_sinal,cz,cz,cz,cz,cz,cz,cz,cz,c1,cz,cz,cz,cz,cz,cz); -- JC 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 d3 => if v=c1 then
                        uins <= (cz,cz,c1,ext_sinal,cz,cz,cz,cz,cz,cz,cz,cz,c1,cz,cz,cz,cz,cz,cz);  -- JV 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 d4 => uins <= (cz,cz,c1,ext_sinal,cz,cz,cz,cz,cz,cz,cz,cz,c1,cz,cz,cz,cz,cz,cz);    -- JMP desloc
          when d5 => uins <= (cz,cz,c1,ext_sinal,cz,cz,c1,cz,cz,cz,cz,cz,c1,cz,c1,c1,c1,cz,c1);    -- JSR desloc
          when d6 => case ir(11 downto 8) is
                      when d0 =>
                         case ir(7 downto 4) is
                          when d0 => if n=c1 then
                                                  uins <= (cz,cz,cz,soma,cz,cz,cz,cz,cz,cz,cz,cz,c1,cz,cz,cz,cz,cz,cz); -- JN Rs1,R
                                 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 d1 =>  if z=c1 then
                                                  uins <= (cz,cz,cz,soma,cz,cz,cz,cz,cz,cz,cz,cz,c1,cz,cz,cz,cz,cz,cz); -- JZ Rs1,R
                                 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 d2 =>  if c=c1 then
                                                  uins <= (cz,cz,cz,soma,cz,cz,cz,cz,cz,cz,cz,cz,c1,cz,cz,cz,cz,cz,cz); -- JC Rs1,R
                                 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 d3 =>  if v=c1 then
                                                  uins <= (cz,cz,cz,soma,cz,cz,cz,cz,cz,cz,cz,cz,c1,cz,cz,cz,cz,cz,cz); -- JV Rs1,R
                                 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 d4 => uins <= (cz,cz,cz,soma,cz,cz,cz,cz,cz,cz,cz,cz,c1,cz,cz,cz,cz,cz,cz);   -- JMP Rs1,R
                          when d5 => uins <= (cz,cz,cz,soma,cz,cz,c1,cz,cz,cz,cz,cz,c1,cz,c1,c1,c1,cz,c1);   -- JSR Rs1,R
                          when others => null;
                         end case;
                      when d1 =>
                           case ir(7 downto 4) is
                                when d0 => if n=c1 then
                                                   uins <= (cz,cz,cz,passaA,cz,cz,cz,cz,cz,cz,cz,cz,c1,cz,cz,cz,cz,cz,cz);  -- JN Rs1
                                                  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 d1 => if z=c1 then
                                                   uins <= (cz,cz,cz,passaA,cz,cz,cz,cz,cz,cz,cz,cz,c1,cz,cz,cz,cz,cz,cz);  -- JZ Rs1
                                                  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 d2 => if c=c1 then
                                        uins <= (cz,cz,cz,passaA,cz,cz,cz,cz,cz,cz,cz,cz,c1,cz,cz,cz,cz,cz,cz);  -- JC Rs1
                                      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 d3 => if v=c1 then
                                        uins <= (cz,cz,cz,passaA,cz,cz,cz,cz,cz,cz,cz,cz,c1,cz,cz,cz,cz,cz,cz);  -- JV Rs1
                                      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 d4 => uins <= (cz,cz,cz,passaA,cz,cz,cz,cz,cz,cz,cz,cz,c1,cz,cz,cz,cz,cz,cz);      -- JMP Rs1
                    when d5 => uins <= (cz,cz,cz,passaA,cz,cz,c1,cz,cz,cz,cz,cz,c1,cz,c1,c1,c1,cz,c1);      -- JSR Rs1
                    when others => null;
                end case;
            when d2 =>
                case ir(7 downto 4) is
                    when d0 => uins <= (cz,cz,cz,passaA,cz,cz,cz,cz,cz,cz,cz,cz,cz,c1,c1,cz,cz,cz,cz);      -- LDSP Rs1
                    when d1 => uins <= (cz,cz,cz,passaA,cz,cz,c1,c1,cz,cz,cz,cz,c1,cz,c1,cz,cz,cz,c1);      -- RTS
                    when d2 => uins <= (cz,cz,cz,soma,  cz,cz,cz,cz,cz,cz,cz,cz,cz,cz,cz,cz,cz,cz,cz);      -- NOP
                    when others => null;   -- LOGICA DE HALT ESTA EM OUTRO PROCESSO
                   end case;
            when others => null;
        end case;

    when d7 => uins <= (c1,cz,c1,cte_low,c1,c1,cz,cz,c1,cz,cz,cz,cz,cz,cz,cz,cz,cz,cz);              -- LDL Rt,#cte
    when d8 => uins <= (c1,cz,c1,cte_high,c1,c1,cz,cz,c1,cz,cz,cz,cz,cz,cz,cz,cz,cz,cz);             -- LDH Rt,#cte
    when d9 =>
      case ir(7 downto 4) is
        when d0 => uins <= (cz,cz,cz,passaA,cz,c1,cz,cz,c1,cz,cz,cz,cz,cz,cz,cz,cz,cz,cz);            -- MOV Rt,Rs1
        when d1 => uins <= (cz,cz,cz,passaA,cz,cz,c1,c1,c1,cz,cz,cz,cz,cz,cz,cz,cz,cz,cz);            -- LD Rt,Rs1
        when d2 => uins <= (cz,cz,cz,sl0   ,cz,c1,cz,cz,c1,cz,cz,cz,cz,cz,cz,cz,cz,cz,cz);            -- SL0 Rt,Rs1
        when d3 => uins <= (cz,cz,cz,sr0   ,cz,c1,cz,cz,c1,cz,cz,cz,cz,cz,cz,cz,cz,cz,cz);            -- SR0 Rt,Rs1
        when d4 => uins <= (cz,cz,cz,sl1   ,cz,c1,cz,cz,c1,cz,cz,cz,cz,cz,cz,cz,cz,cz,cz);            -- SL1 Rt,Rs1
        when d5 => uins <= (cz,cz,cz,sr1   ,cz,c1,cz,cz,c1,cz,cz,cz,cz,cz,cz,cz,cz,cz,cz);            -- SR1 Rt,Rs1
        when d6 => uins <= (cz,cz,cz,incA  ,cz,c1,cz,cz,c1,c1,c1,cz,cz,cz,cz,cz,cz,cz,cz);            -- INC Rt,Rs1
        when d7 => uins <= (cz,cz,cz,decA  ,cz,c1,cz,cz,c1,c1,c1,cz,cz,cz,cz,cz,cz,cz,cz);            -- DEC Rt,Rs1
        when d8 => uins <= (cz,cz,cz,notA  ,cz,c1,cz,cz,c1,cz,cz,cz,cz,cz,cz,cz,cz,cz,cz);            -- NOT Rt,Rs1
        when d9 => uins <= (cz,cz,cz,negA  ,cz,c1,cz,cz,c1,c1,c1,cz,cz,cz,cz,cz,cz,cz,cz);            -- NEG Rt,Rs1
        when others => null;
      end case;
    when d10 => uins <= (cz,cz,cz,soma,c1,c1,cz,cz,c1,c1,c1,cz,cz,cz,cz,cz,cz,cz,cz);                 -- ADD Rt,Rs1,Rs2
    when d11 => uins <= (cz,cz,cz,AsubB,c1,c1,cz,cz,c1,c1,c1,cz,cz,cz,cz,cz,cz,cz,cz);                -- SUB Rt,Rs1,Rs2
    when d12 => uins <= (cz,cz,cz,e,c1,c1,cz,cz,c1,cz,cz,cz,cz,cz,cz,cz,cz,cz,cz);                    -- AND Rt,Rs1,Rs2
    when d13 => uins <= (cz,cz,cz,ou,c1,c1,cz,cz,c1,cz,cz,cz,cz,cz,cz,cz,cz,cz,cz);                   -- OR  Rt,Rs1,Rs2
    when d14 => uins <= (cz,cz,cz,ouX,c1,c1,cz,cz,c1,cz,cz,cz,cz,cz,cz,cz,cz,cz,cz);                  -- XOR Rt,Rs1,Rs2
    when d15 => uins <= (cz,cz,cz,passaB,c1,cz,c1,cz,cz,cz,cz,cz,cz,cz,cz,cz,cz,c1,cz);               -- ST  Rt,Rs2,Rs1
    when others => null;
  end case;

  end process;

end controle;

------------------------------------------------------------------------
-- Uniao dos componentes da R3
------------------------------------------------------------------------
library IEEE;
use IEEE.Std_Logic_1164.all;
use work.r3.all;

entity processador is
     port(iclock,ireset: in std_logic;
          datain: in reg16;
          dataout: out reg16;
          oAddress: out reg16;
          oce,orw: out std_logic;
          e_halt: out std_logic
          );
end processador;

architecture processador of processador is

  component controle
          port (uins: out microinstrucao; reset,ck: in std_logic; n,c,z,v: in std_logic;  ir: in reg16;  e_halt: out std_logic );
  end component;

  component datapath
          port( uins: in microinstrucao;
                ck, reset: in std_logic;
                ir, endereco : out reg16;
                datain: in reg16;
                dataout: out reg16;
                n, z, c, v: out std_logic
                );
  end component;

  signal n,z,c,v: std_logic;
  signal uins: microinstrucao;
  signal ir: reg16;

 begin

  dp: datapath port map(uins=>uins,ck=>iclock,reset=>ireset,ir=>ir,endereco=>oAddress,datain=>datain,dataout=>dataout,c=>c,z=>z,n=>n,v=>v);
  ctrl: controle port map(uins=>uins,reset=>ireset,ck=>iclock,n=>n,c=>c,z=>z,v=>v,ir=>ir,e_halt=>e_halt);

  oce <= uins.ce and iclock;  -- voltou a gambiarra !
  orw <= uins.rw and iclock;

end processador;
```

9.2 Fontes do Controlador

```
library IEEE;
use IEEE.Std_Logic_1164.all;
use work.R3.all;

package R3xs is
        type Type_STATE is (S0,S1,S2,S22,S3,S4,S5,S6,SW1,SW2,SRD1,SRD2,SRET, SiniciaR3);

        component processador is port(iclock, ireset: in std_logic; datain:   in reg16;  dataout:  out reg16;  oAddress: out reg16;  oce,orw:  out std_logic; e_halt:   out std_logic ); end component;

        component Bcd2Seg is     port ( entrada: in std_logic_vector(4 downto 1); saida: out reg8 );  end component;

        component debounce is port (lento, clock, reset, key: in STD_LOGIC;          debkey: out STD_LOGIC); end component;

        component OSC4 is port (F15: OUT std_logic);  end component;
end R3xs;

--------------------------------------------------------------------------------
-- Entidade Debounce: realiza a filtragem das teclas. O Debounce serve de
-- intermediario entre as teclas e a entidade principal R3xs40.
--------------------------------------------------------------------------------
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

entity debounce is
  port (lento, clock, reset, key: in STD_LOGIC; debkey: out STD_LOGIC);
end;

architecture debounce_arch of debounce is
   type State_type is (S1, S2, S3);
   signal EA: State_type;
   signal cont: STD_LOGIC_VECTOR(1  downto 0);
 begin
        process (reset, clock)
          begin
           if reset='0'then  EA <= S1;
           elsif clock'event and clock='0' then              -- borda de descida do clock da maquina
                case EA is
                  when S1 =>
                       if key='1' then  EA <= S1;  elsif key='0' then   EA <= S2;  end if;
                  when S2 =>
                       EA <= S3;
                  when S3 =>
                       if cont="11" then EA <= S1; else  EA <= S3 ;  end if;
                end case;
          end if;
        end process;

        debkey<= '0' when EA=S2 else '1';                              -- LOGICA NEGADA

        process (lento, EA)
        begin
           if EA=S1 then cont <= "00";
             elsif lento'event and lento='1' then
                  cont <= cont + 1;
          end if;
        end process;

 end debounce_arch;

--------------------------------------------------------------------------------
-- Entidade Bcd2Seg
-- Este circuito realiza a conversao binario para 7 segmentos para
-- posteriormente ser mostrado nos displays da placa XS40
--------------------------------------------------------------------------------
library IEEE;
use IEEE.std_logic_1164.all;
use work.R3.all;

entity Bcd2Seg is
        port ( entrada: in std_logic_vector(3 downto 0); saida: out reg8);
end Bcd2Seg;

architecture Bcd2Seg_arch of Bcd2Seg is
begin
        --segment encoding
        --          6
        --         ---
        --      5 |   | 4
        --         ---  <- 3
        --      2 |   | 1
        --         ---
        --          0
        with entrada (3 downto 0) select
        saida <=      "00010010" when "0001", --1              "01011101" when "0010", --2
                      "01011011" when "0011", --3              "00111010" when "0100", --4
                      "01101011" when "0101", --5              "01101111" when "0110", --6
                      "01010010" when "0111", --7              "01111111" when "1000", --8
                      "01111011" when "1001", --9              "01111110" when "1010", --A
                      "00101111" when "1011", --b              "01100101" when "1100", --C
                      "00011111" when "1101", --d              "01101101" when "1110", --E
                      "01101100" when "1111", --F              "01110111" when others; --0
end Bcd2Seg_arch;

--------------------------------------------------------------------------------
-- Entidade R3xs40
--------------------------------------------------------------------------------
-- Esta entidade constitui-se no bloco principal do projeto. Todos as
-- funcionalidades, sao abaixo descritas:
-- Realiza a filtragem da tecla SPARE atraves do componente Debounce
-- Sincronizacao do sinal assincrono SPARE
```

```
-- Codificacao e atribuicao ao display da XS40 do estado atual
-- Geracao do proximo estado
-- Escrita e leitura na memoria (Ram)
-- Logica de cola com o core da R3
--------------------------------------------------------------------------------
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;
use work.R3xs.all;
use work.R3.all;

entity R3xs40 is
  port(     spareb: in std_logic;          -- SPARE pushbutton
            resetb: in std_logic;          -- RESET pushbutton
            display: out reg8;             -- XS Board LED diit
            dram: inout reg8;              -- XS Board Ram data
            adram: out reg15;              -- XS Board Ram Address
            rw: out std_logic;             -- Read/Write (negado)
            oe: out std_logic;             -- Output enable for all RAMs (negado)
            ce: out std_logic;             -- Chip enable (negado)
            halt_out: out std_logic;
            ck12: in std_logic             -- clock para a R3(12Mhz)
        );
end R3xs40;

architecture R3xs40_arch of R3xs40 is

    signal EA, SRET: Type_STATE;           -- sinais de Estado Atual e Proximo Estado

    signal spareCK, F15, ck_interno,       -- tecla SPARE sincronizada ao clock
           ck_R3, reset_R3, e_halt, ce_R3, rw_R3: std_logic;

    signal address_c, data_in, mar, data_out: reg16;
    signal out_BCD1, out_BCD2: reg8;
    signal cycles, contador : std_logic_vector(1 downto 0);

begin

    -- COMPONENTES DO CIRCUITO CONTROLADOR: cpu, OSC4, debounce e Bcd2Seg

    C1: processador port map (iclock=>ck_R3, ireset=>reset_R3, datain=>data_in, dataout=>data_out, oAddress=>address_c, oce=>ce_R3, orw=>rw_R3, e_halt=>e_halt);

    halt_out <= e_halt;

    C2 : OSC4 port map (F15);

    C3: debounce port map (lento=>F15, clock=>ck12, reset=>resetb, key=>spareb, debkey=>spareCK);

    C4 : Bcd2Seg port map(data_in(7 downto 4), out_BCD2);
    C5 : Bcd2Seg port map(data_in(3 downto 0), out_BCD1);

    -- Reset da R3
    reset_R3 <= '1' when EA=S0 else '0';

    --------------------------------------------------------------------------------
    -- CLOCK ASSIMETRICO PARA A R3: ciclo de busca/execucao mais longa para ler/escrever na memoria
    -- ELE NAO DEPENDE MAIS DO SINAL HALT - PODIA ESTAR HAVENDO UM "DEAD-LOCK"
    --------------------------------------------------------------------------------
    process
      begin
        ck_interno <= '0';
        wait until  ck12'event and ck12='1';
        wait until  ck12'event and ck12='1';
        wait until  ck12'event and ck12='1';
        ck_interno <= '1';
        wait until  ck12'event and ck12='1';
        wait until  ck12'event and ck12='1';
        wait until  ck12'event and ck12='1';
    end Process;

    ck_R3<= ck_interno when SRET=S1 else '0';

    ---- Processo que codifica o estado atual para visualizacao no display da XS40
    ---- ****  apenas os estados S0, S3, S5 e S6 sao visiveis pelo usuario.
    process(EA)
     begin
        case(EA) is
            when S0 =>  display<="01110111";
            when S3 =>  display<="01011011";
            when S5 =>  display<="01101011";
            when S6 =>  display<="00101111";
            when others=> display<="11111111";
        end case;
    end process;

    -- Processo que gera o proximo estado - BORDA DE DESCIDA
    -- Armazena o estado de retorno
    -- CONTROLA o registrador de endereco
    process(ck12,resetb)
    begin
     if (resetb='0') then EA<=S0; SRET<=SiniciaR3; mar<=(others => '0');
     elsif ck12'event and ck12='0' then

         case(EA) is
         -- Estado Inicial espera pelo reset
         when S0 =>  if spareCK='0' then  EA<=SiniciaR3;  else EA<=S0;  end if;
         when SiniciaR3 => EA<=S1; SRET<=S1;

         -- Atribui reset a R3, esperando pelo fim do processamento (e_halt='1')
         when S1 =>   SRET <= S1;
                  if e_halt='1' then EA<=S2;
                     elsif ce_R3='1' then mar <= address_c;
                     if rw_R3='0' then  EA<=SW1; else  EA<=SRD1; end if;
                   end if;
```

```
            -- Pega no endereco 00FFh o inicio da area de dados
            when S2 =>  SRET <= S3;
                        mar <= x"00FF";
                        EA<=S22;
            when S22 => EA<=SRD1;

            -- Mostra no display fim de processamento
            -- Estado Inicial de leitura da memoria (espera borda de subida de spare)
            when S3 =>  SRET <= S5;
                       mar <= data_in;
                        if spareCK='0' then EA<=SRD1; else EA<=S3; end if;

             --  Mostra no display e leds o conteudo de mdr (espera borda de subida de spare)
            when S5 =>  if  spareCK='0' then EA<=S4; else EA<=S5; end if;

            -- Incrementa o endereco da area de dados e testa fim de exibicao
            when S4 =>  mar <= mar + 1;
                       EA<=SRD1;

            -- Mostra lido OK e fica preso neste estado, saindo apenas em caso de reset
            when S6 =>   EA<=S6;

            when SW1   =>  EA<=SW2;                  -- estados de escrita na memoria
            when SW2   =>  EA<=SRET;

            when SRD1  => EA<=SRD2;                  -- estados de leitura na memoria
            when SRD2  => EA<=SRET;

            when others => EA <= S0;

           end case;

      end if;
  end process;

    ------------------------------------------------------------------------------
    -- dado VINDO da memoria  DEVE SER NA BORDA DE DESCIDA DE CK12
    ------------------------------------------------------------------------------
    process(ck12,resetb,EA)
    begin
          if resetb='0' then data_in<=(others=>'0');
          elsif ck12'event and ck12='0' then
             if EA=SRD1 then data_in(15 downto 8)<=dram;
             elsif EA=SRD2 then data_in(7 downto 0)<=dram;
              end if;
          end if;
    end process;

    ------------------------------------------------------------------------------
    -- DADO QUE SERA' ESCRITO NA MEMORIA (ou leds), nao esquecer da multiplexacao
    -- dado da R3 em SW1 ou SW2, parte alta dos dados lidos (S5 ou S6),
    -- ou todos os leds em S3
    ------------------------------------------------------------------------------
    dram <=  data_out (15 downto 8) when (EA=SW1)
            else data_out (7 downto 0)  when (EA=SW2)
            else not (data_in(15 downto 8)) when (EA=S5 or EA=S6)
            else (others=>'0') when (EA=S3)
            else "ZZZZZZZZ";

    ------------------------------------------------------------------------------
    -- ENDERECAMENTO DA RAM EXTERNA OU 7 SEGMENTOS
    ------------------------------------------------------------------------------
    adram<= not(out_BCD1(6 downto 0)) & not(out_BCD2(7 downto 0)) when (EA=S3 or EA=S5 or EA=S6)      -- Mostra no display o conteudo de data_in
              else mar(13 downto 0) & '0' when (EA=SRD1 or EA=SW1)                              -- Endereco a ser lido
              else mar(13 downto 0) & '1';

    ------------------------------------------------------------------------------
    -- ACESSO A RAM - Versao simplificada
    -- de acordo com o data-sheet da memoria, esta pode estar sempre habilitada (em '0')
    ------------------------------------------------------------------------------
    oe <=  '0' when ck12='1' and (EA=SRD1 or EA=SRD2) else '1' ;
    rw <=  '0' when ck12='1' and (EA=SW1  or EA=SW2)  else '1' ;
    ce <=  '0' ;
end R3xs40_arch;
```

9.3 Fontes do Test_bench – processador

```
library IEEE;
use IEEE.Std_Logic_1164.all;
use IEEE.Std_Logic_unsigned.all;
use STD.TEXTIO.all;
use work.R3.all;

entity r3_tb is
end r3_tb;

architecture r3_tb of r3_tb is
component processador is port(iclock,ireset: in std_logic;  datain: in reg16; dataout: out reg16; oAddress: out reg16; oce,orw: out std_logic; e_halt: out std_logic); end component;

    signal ck, reset, ce, rw, go: std_logic;
    signal address, datain, dataout: reg16;
    file   ARQ : TEXT open READ_MODE is "st.txt";
    signal RAM : mem1 := mem1'(others=>"0000000000000000");
    signal endereco  : integer;
    signal ins       : reg16;

begin
    -- Inclui a R3 no Test_Bench
    cpu : processador port map (iclock=>ck, ireset=>reset, datain=>datain, dataout=>dataout, oAddress=>address, oce=>ce, orw=>rw);
```

```
	-- le da memoria
	datain <= RAM(CONV_INTEGER(address)) when address<(TAMaddress + 1) and ce='1' and rw='1' else "ZZZZZZZZZZZZZZZZ";

	-- escreve na memoria, de maneira sincrona, como nos registradores
	process(go, ce, rw, ck)
	begin
		-- borda de subida do go - escreve na memoria
		if go'event and go='1' then
			if endereco>=0 and endereco<=TAMaddress then RAM(endereco) <= ins; end if;
		-- borda de descida do go - escreve na memoria
		elsif ck'event and ck='0' then
			if ce='1' and rw='0' then
			if endereco>=0 and endereco<=TAMaddress then RAM(CONV_INTEGER(address)) <= dataout; end if;
		end if;
		end if;
	end process;

	-- gera o reset
	reset <= '1', '0' after 5ns;

	-- gera o clock
	process
	begin
		ck <= '1',  '0' after 10ns;
		wait for 20ns;
	end process;

	-- realiza a carga na memoria quando acontece o reset
	process
		variable LINHA_ARQ : LINE;
		variable linha     : string(1 to 9);
		variable base, num : integer;
		variable bin       : STD_LOGIC_VECTOR(1 to 4);

	begin
		wait until reset = '1';

		while NOT (endfile(ARQ)) loop           -- end file checking
			readline(ARQ, LINHA_ARQ);       -- read line of a file
			read(LINHA_ARQ, linha);

			endereco <= 0;
		   wait for 1ps;

			base := 1;
			for w in 4 downto 1 loop
				if w /= 4 then base := base * 16; end if;
				case linha(w) is
					when '0' => num := 0;		when '1' => num := 1;
					when '2' => num := 2;		when '3' => num := 3;
					when '4' => num := 4;		when '5' => num := 5;
					when '6' => num := 6;		when '7' => num := 7;
					when '8' => num := 8;		when '9' => num := 9;
					when 'A' => num := 10;		when 'B' => num := 11;
					when 'C' => num := 12;		when 'D' => num := 13;
					when 'E' => num := 14;		when 'F' => num := 15;
					when others => null;
				end case;

				endereco <= endereco + num * base;
				wait for 1ps;
			end loop;  -- end loop da conversao p/ decimal

			for w in 6 to 9 loop
				case linha(w) is
					when '0' => bin := d0;		when '1' => bin := d1;
					when '2' => bin := d2;		when '3' => bin := d3;
					when '4' => bin := d4;		when '5' => bin := d5;
					when '6' => bin := d6;		when '7' => bin := d7;
					when '8' => bin := d8;		when '9' => bin := d9;
					when 'A' => bin := d10;		when 'B' => bin := d11;
					when 'C' => bin := d12;		when 'D' => bin := d13;
					when 'E' => bin := d14;		when 'F' => bin := d15;
					when others => null;		end case;

				case w is
					when 6 => ins(15 downto 12) <= bin;
					when 7 => ins(11 downto 8)  <= bin;
					when 8 => ins(7 downto 4)   <= bin;
					when 9 => ins(3 downto 0)   <= bin;
				end case;

			end loop;  -- end loop da conversao p/ binario

			wait for 1ps;
			go <= '1';    -- este sinal de go habilita a escrita na memoria

			wait for 1ps;
			go <= '0';    -- tira o go
		end loop; -- end loop da leitura do arquivo
	end process;

end r3_tb;
```

9.4 Fontes do Test_bench – processador e controlador

```
library ieee;
use ieee.std_logic_1164.all;

entity OSC4 is
	 port (F15: OUT std_logic);
end OSC4;
```

```
architecture OSC4 of OSC4 is
begin
	process
	begin
		F15 <= '1', '0' after 20ns;
		wait for 40ns;
		end process;
end OSC4;

------------------------------------------------------------------------
--  TEST_BENCH PARA SIMULACAO DO CONTROLE MAIS CORE
------------------------------------------------------------------------
library ieee;
use ieee.STD_LOGIC_UNSIGNED.all;
use ieee.std_logic_1164.all;
use STD.TEXTIO.all;

entity r3xs40_tb is
end r3xs40_tb;

architecture TB_ARCHITECTURE of r3xs40_tb is

	component R3xs40 port(spareb : in std_logic; resetb : in std_logic; display : out std_logic_vector(7 downto 0); dram : inout std_logic_vector(7 downto 0); adram : out std_logic_vector(14 downto 0); rw : out std_logic;
oe : out std_logic;  ce : out std_logic;
				ck12 : in std_logic );  end component;

	type mem1 is array (0 to 1023) of std_logic_vector(7 downto 0);
	constant TAMaddress : integer := 1023;

	signal go, spareb, resetb, ck12, rw, oe, ce : std_logic;
	signal dram : std_logic_vector(7 downto 0);
	signal display : std_logic_vector(7 downto 0);
	signal adram :  std_logic_vector(14 downto 0);

	signal RAM : mem1;
	signal endereco, endereco1, endereco2 : integer;
	signal ins : std_logic_vector(15 downto 0);

	file ARQ : TEXT open READ_MODE is "bubblefull.r3"   ;
begin

	UUT : R3xs40 port map (spareb => spareb, resetb => resetb, display => display, dram => dram, adram => adram, rw => rw, oe => oe, ce => ce, ck12 => ck12 );

	-- gera a tecla spareb
	process
	  begin
		resetb <= '0', '1' after 5ns;
		spareb <= '1', '0' after 30ns, '1' after 60ns;   -- primeiro spare

		wait for 50000ns;  -- tempo de processamento

		for x in 0 to 16 loop -- envia  16 pulsos nas chaves
			spareb <= '0', '1' after 30ns;
			wait for 150ns;
		end loop;

		wait for 500ns;		-- aguarda 500 ns para inicio de novo ciclo

	end process;

	-- le da memoria
	dram <= RAM(CONV_INTEGER(adram)) when adram<(TAMaddress + 1) and ce='0' and oe='0' else "ZZZZZZZZ";

	-- escreve na memoria, de maneira sincrona, como nos registradores
	process(go, ce, rw, ck12)
	begin
		-- borda de subida do go - escreve na memoria
		if go'event and go='1' then
			if endereco1>=0 and endereco2<=TAMaddress then RAM(endereco1) <= ins(15 downto 8); RAM(endereco2) <= ins(7 downto 0); end if;
		-- borda de descida DE RW
		elsif rw'event and rw='0' then
			if CONV_INTEGER(adram)>=0 and CONV_INTEGER(adram)<=TAMaddress then  RAM(CONV_INTEGER(adram)) <= dram;
			end if;
		end if;
	end process;

	-- gera o clock
	process
	begin
		ck12 <= '1', '0' after 10ns;
		wait for 20ns;
	end process;

	-- realiza a carga na memoria quando acontece o reset
	process
		variable LINHA_ARQ : LINE;
		variable linha     : string(1 to 9);
		variable base, num : integer;
		variable bin       : STD_LOGIC_VECTOR(1 to 4);

	begin
		wait until resetb = '0';

		while NOT (endfile(ARQ)) loop		-- end file checking
			readline(ARQ, LINHA_ARQ);	-- read line of a file
			read(LINHA_ARQ, linha);
			endereco <= 0;
		  wait for 1ps;

			base := 1;
			for w in 4 downto 1 loop
				if w /= 4 then base := base * 16; end if;
				case linha(w) is
					when '0' => num := 0;			when '1' => num := 1;
```

```
                    when '2' => num := 2;                when '3' => num := 3;
                    when '4' => num := 4;                when '5' => num := 5;
                    when '6' => num := 6;                when '7' => num := 7;
                    when '8' => num := 8;                when '9' => num := 9;
                    when 'A' => num := 1                 when 'B' => num := 11;
                    when 'C' => num := 1                 when 'D' => num := 13;
                    when 'E' => num := 14;               when 'F' => num := 15;
                    when others => null;
                end case;

                endereco <= endereco + num * base;
                wait for 1ps;
            end loop;  -- end loop da conversao p/ decimal

            for w in 6 to 9 loop
                case linha(w) is
                    when '0' => bin := "0000";           when '1' => bin := "0001";
                    when '2' => bin := "0010";           when '3' => bin := "0011";
                    when '4' => bin := "0100";           when '5' => bin := "0101";
                    when '6' => bin := "0110";           when '7' => bin := "0111";
                    when '8' => bin := "1000";           when '9' => bin := "1001";
                    when 'A' => bin := "1010";           when 'B' => bin := "1011";
                    when 'C' => bin := "1100";           when 'D' => bin := "1101";
                    when 'E' => bin := "1110";           when 'F' => bin := "1111";
                    when others => null;
                end case;

                case w is
                    when 6 => ins(15 downto 12) <= bin;
                    when 7 => ins(11 downto 8)  <= bin;
                    when 8 => ins(7 downto 4)   <= bin;
                    when 9 => ins(3 downto 0)   <= bin;
                end case;

            end loop;  -- end loop da conversao p/ binario

            endereco1 <= 2*(endereco + num * base);
            endereco2 <= 2*(endereco + num * base)+1;

            wait for 1ps;
            go <= '1';    -- este sinal de go habilita a escrita na memoria

            wait for 1ps;
            go <= '0';    -- tira o go

        end loop; -- end loop da leitura do arquivo
    end process;

end TB_ARCHITECTURE;
```

9.5 Arquivo com a pinagem para a placa de prototipação – UCF

```
NET display<0>        LOC=P25;
NET display<1>        LOC=P26;
NET display<2>        LOC=P24;
NET display<3>        LOC=P20;
NET display<4>        LOC=P23;
NET display<5>        LOC=P18;
NET display<6>        LOC=P19;

NET ck12              LOC=P13;    # CLOCK
NET spareb            LOC=P67;
NET resetb            LOC=P37;
NET ce                LOC=P65;
NET oe                LOC=P61;
NET rw                LOC=P62;

#
# SRAM DATA
NET dram<0>           LOC=P41;
NET dram<1>           LOC=P40;
NET dram<2>           LOC=P39;
NET dram<3>           LOC=P38;
NET dram<4>           LOC=P35;
NET dram<5>           LOC=P81;
NET dram<6>           LOC=P80;
NET dram<7>           LOC=P10;

#
# SRAM ADDRESS
NET adram<0>     LOC=P3;
NET adram<1>     LOC=P4;
NET adram<2>     LOC=P5;
NET adram<3>     LOC=P78;
NET adram<4>     LOC=P79;
NET adram<5>     LOC=P82;
NET adram<6>     LOC=P83;
NET adram<7>     LOC=P84;
NET adram<8>     LOC=P59;
NET adram<9>     LOC=P57;
NET adram<10>    LOC=P51;
NET adram<11>    LOC=P56;
NET adram<12>    LOC=P50;
NET adram<13>    LOC=P58;
NET adram<14>    LOC=P60;

# para debug
# NET saida_ck_r3    LOC=P66;
NET halt_out     LOC=P77;
```