UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE ENGENHARIA DE COMPUTAÇÃO

GUSTAVO KAEFER ORTH

Implementação em Hardware da Arquitetura do Computador Hipotético CESAR

Trabalho de Diplomação.

Prof. Dr. Carlos Arthur Lang Lisbôa
Orientador

Porto Alegre, novembro de 2010.

AGRADECIMENTOS

Agradeço aos meus pais pela educação, compreensão e, sobretudo, pelo apoio nas dificuldades enfrentadas durante esta jornada. Sem vocês este trabalho não seria possível.

Agradeço também a todos os meus professores nestes anos de curso. Certamente o conhecimento e as lições que compartilham com os alunos, serão de grande importância nas nossas vidas e carreiras.

Agradeço especialmente ao professor Lisbôa que, não só me concedeu a honra e a responsabilidade de ser seu primeiro orientando, como também foi responsável pela escolha do tema deste trabalho. Muito obrigado pela oportunidade.

SUMÁRIO

LISTA DE ABREVIATURAS E SIGLAS

ASCII	American Standard Code for Information Interchange
CRT	Cathode Ray Tube
FPGA	Field Programmable Gate Array
FSM	Finite State Machine
LUT	Look Up Table
PC	Program Counter
RTL	Register Transfer Level
SP	Stack Pointer
UCF	User Constraints File
UFRGS	Universidade Federal do Rio Grande do Sul
ULA	Unidade Lógica e Aritmética
USB	Universal Serial Bus
VGA	Video Graphics Array
VHDL	Very High Speed Integrated Circuit Hardware Description Language

LISTA DE FIGURAS

LISTA DE TABELAS

RESUMO

O contínuo desenvolvimento de ferramentas de síntese lógica, em conjunto com o aumento da capacidade de dispositivos de hardware programável como FPGAs, permitiu o desenvolvimento de processadores *soft-core*, projetados especificamente para rodar nestes dispositivos. Ao mesmo tempo, linguagens de descrição de hardware, como VHDL, permitem a descrição de sistemas digitais em diferentes níveis de abstração.

Este trabalho apresenta duas possíveis implementações em VHDL da arquitetura do computador Cesar, um processador hipotético utilizado no Instituto de informática da UFRGS como ferramenta de auxílio no ensino de arquitetura e organização de computadores. O computador Cesar é baseado na arquitetura da família de processadores PDP-11, da Digital Equipment Corporation. Além da descrição do processador, foram desenvolvidos circuitos que implementam a interface deste processador com um monitor de vídeo e um teclado, ambos conectados a uma placa FPGA.

Palavras-Chave: VHDL, processadores *soft-core*, FPGA, processador hipotético Cesar.

Hardware Implementation of the CESAR hypothetical computer architecture

ABSTRACT

The continuous development of logic synthesis tools, in conjunction with the increase in capacity of programmable hardware devices such as FPGAs, allowed the development of soft-core processors, designed specifically to run on these devices. At the same time, hardware description languages, such as VHDL, allow the description of digital systems in different levels of abstraction.

This paper presents two possible implementations in VHDL of the Cesar computer architecture, an hypothetical processor used in the Institute of Informatics at UFRGS as an aid tool in the teaching of computer architecture and organization. The Cesar computer is based on the architecture of the PDP-11 processor family, manufactured by Digital Equipment Corporation. Besides the description of the processor, circuits that perform the interface of this processor with a video monitor and a keyboard, both connected to an FPGA Board, have been developed.

Keywords: VHDL, soft-core processors, FPGA, Cesar hypothetical processor.

1 INTRODUÇÃO

1.1 Contextualização

Em seus 25 anos de existência, os dispositivos de hardware programável FPGAs (*Field Programmable Gate Arrays)* têm tido um papel de fundamental importância no desenvolvimento de sistemas digitais. Se no início estes dispositivos eram usados apenas para a prototipação de circuitos relativamente simples, permitindo que fossem testados antes do processo de fabricação, o aumento da capacidade dos FPGAs, aliado ao desenvolvimento contínuo das ferramentas de síntese lógica, possibilita que sistemas de alta complexidade sejam desenvolvidos para estes dispositivos.

Neste contexto, existem hoje vários processadores desenvolvidos especificamente para rodar em hardware programável. Estes processadores, denominados usualmente de processadores *soft-core*, permitem que sistemas sejam implementados em hardware programável seguindo uma metodologia muito semelhante à usada no desenvolvimento de um programa de computador comum.

Processadores *soft-core*, assim como outros sistemas digitais feitos para rodar em FPGAs, são desenvolvidos com o auxílio de linguagens de descrição de hardware. A linguagem VHDL (*Very High Speed Integrated Circuit Hardware Description Language*) é uma das mais populares linguagens de descrição de hardware existentes. VHDL é uma linguagem extremamente complexa e sofisticada, permitindo que um sistema digital seja descrito de várias maneiras, com diferentes níveis de abstração (BROWN, 2005).

1.2 Objetivos e Motivação

Ao longo de vários anos, as disciplinas introdutórias sobre arquitetura e organização de computadores, dos cursos de Ciência da Computação e Engenharia de Computação da UFRGS, têm feito uso de computadores hipotéticos, concebidos com o claro propósito de facilitar a aprendizagem dos conceitos referentes à arquitetura de computadores. Dentre estes computadores, o Cesar é o último a ser estudado e sua arquitetura, inspirada na família de processadores PDP-11, que fez muito sucesso durante os anos setenta e inicio dos anos oitenta, é consideravelmente mais complexa que a dos demais.

Por esta maior complexidade, o Cesar é o único computador que ainda não teve sua arquitetura implementada em hardware por alunos da graduação. Assim, este trabalho propõe duas possíveis implementações VHDL do conjunto de instruções da arquitetura Cesar. A descrição do hardware do computador será então programada em uma placa FPGA, de forma que seja possível a execução de programas desenvolvidos especificamente para este processador, incluindo programas que façam uso de entrada e saída de dados através de um teclado e um monitor VGA, respectivamente.

1.3 Organização

O capítulo 2 descreve o conjunto de instruções e as demais características básicas do computador Cesar. Também são descritas brevemente algumas características da placa de desenvolvimento utilizada no trabalho, bem como o conjunto de ferramentas de software utilizadas no processo de descrição do circuito.

No capítulo 3 são mostrados os dois modelos de implementação do processador, comportamental e estrutural, detalhando a organização de memória utilizada e o funcionamento do sistema de entrada e saída. O capítulo descreve também a máquina de estados utilizada na versão comportamental, bem como os diversos componentes que formam o circuito da versão estrutural. Além de simulações das duas versões, que exemplificam os passos envolvidos na execução de uma instrução do processador, é feita uma breve comparação entre os dois modelos de implementação.

O capítulo 4 mostra como foram construídos os circuitos de interface de vídeo e do teclado, que compõem o circuito completo do computador Cesar, utilizado nos testes de funcionamento realizados na placa.

Já o capítulo 5 relata os objetivos alcançados e dificuldades encontradas no desenvolvimento deste trabalho. Por fim, são dadas sugestões de trabalhos futuros que poderiam ser desenvolvidos tendo este trabalho como base.

2 FUNDAMENTAÇÃO TEÓRICA E FERRAMENTAS UTILIZADAS

2.1 Arquitetura Cesar

Cesar é um computador hipotético cuja arquitetura foi inspirada no PDP-11, um processador de pequeno porte muito popular no início dos anos 80. A seguir são listadas as características básicas da arquitetura do computador Cesar. Uma descrição mais detalhada desta arquitetura, bem como exemplos de programas desenvolvidos para este processador, podem ser vistos em (WEBER, 2008).

- Largura de dados e de endereços de 16 bits.
- Memória de 64 KB, com dados armazenados na memória usando a disposição *big endian*, na qual o byte mais significativo de uma palavra é armazenado em um byte e o menos significativo no byte seguinte.
- Endereçamento da memória a byte, uma vez que algumas instruções são codificadas em apenas um byte.
- Oito registradores de uso geral (R0 a R7). Dentre estes registradores, R6 é usado como ponteiro da pilha (SP) e R7 como apontador de programa (PC).
- Registrador de estado com quatro códigos de condição: negativo (N), zero (Z), carry (C) e overflow (V).
- Dados representados em complemento de dois.
- Possui um total de 41 instruções, incluindo:
  - 15 instruções de desvio condicional.
  - 12 instruções aritméticas de um operando.
  - 6 instruções de dois operandos.
  - 2 instruções para chamada e retorno de sub-rotinas.
  - 1 instrução de controle de laço.

A arquitetura Cesar possui oito modos de endereçamento, descritos a seguir:

- Registrador: o operando encontra-se no registrador especificado na instrução.
- Registrador Pós-incrementado: o endereço do operando é dado pelo registrador especificado na instrução. Após o acesso, o valor do registrador é incrementado em duas unidades.
- Registrador Pré-decrementado: o conteúdo do registrador especificado na instrução é decrementado em duas unidades. Após esta operação o registrador contém o endereço do operando.
- Indexado: a palavra que segue a instrução é somada ao conteúdo do registrador especificado, formando o endereço do operando.
- Registrador indireto: o registrador especificado na instrução contém o endereço do operando.

- Pós-incrementado indireto: o endereço do endereço do operando é dado pelo registrador especificado na instrução. Após o acesso, o valor do registrador é incrementado em duas unidades.
- Pré-decrementado indireto: o conteúdo do registrador especificado na instrução é decrementado em duas unidades. Após esta operação o registrador contém o endereço do endereço do operando.
- Indexado indireto: a palavra que segue a instrução é somada ao conteúdo do registrador especificado, formando o endereço do endereço do operando.

A tabela 2.1 lista os códigos usados nas instruções de cada um dos modos de endereçamento, bem como o nome utilizado na descrição VHDL do computador.

Tabela 2.1: Modos de endereçamento do processador Cesar

Código	Sigla	Modo
000	REG	Registrador
001	RPI	Registrador pós-incrementado
010	RPD	Registrador pré-decrementado
011	INX	Indexado
100	IND	Registrador indireto
101	PII	Pós-incrementado indireto
110	PDI	Pré-decrementado indireto
111	IXI	Indexado indireto

Além destes modos de endereçamento, como o registrador R7, que é utilizado como PC, pode ter seu conteúdo alterado livremente, outros modos de endereçamento podem ser obtidos. Aqui é importante ressaltar que, após a busca e a decodificação de uma instrução que utilize qualquer um dos oito modos de endereçamento listados, o valor de R7 é automaticamente incrementado em duas unidades.

- Endereçamento Imediato: utilizando-se o modo de endereçamento pós-incrementado em conjunto com o registrador R7, o operando localiza-se na palavra que segue a instrução. Após a busca e decodificação da instrução, R7, já incrementado em duas unidades, aponta para o operando. Após o acesso, R7 é incrementado novamente em duas unidades, apontando para o endereço da próxima instrução.
- Endereçamento Absoluto: de maneira semelhante ao modo imediato, utilizando-se o modo de endereçamento pós-incrementado indireto em conjunto com o registrador R7, o endereço do operando localiza-se na palavra que segue a instrução.
- Endereçamento relativo ao PC: utilizando-se o modo indexado ou indexado indireto, em conjunto com o registrador R7, a palavra que segue a instrução é somada ao valor de R7 para fornecer o endereço do operando ou o endereço do endereço do operando, respectivamente.

Na arquitetura Cesar, os códigos das instruções são formados por um ou dois bytes. A tabela 2.2 classifica as instruções em dez tipos diferentes, de acordo com o valor dos primeiros quatro bits do código da instrução.

Tabela 2.2: Códigos das instruções

Código	Tipo
0000	Instrução NOP
0001 e 0010	Instruções sobre códigos de condição
0011	Instruções de desvio condicional
0100	Instrução de desvio incondicional (JMP)
0101	Instrução de controle de laço (SOB)
0110	Instrução de desvio para sub-rotina (JSR)
0111	Instrução de retorno de sub-rotina (RTS)
1000	Instruções de um operando
1001 e 1110	Instruções de dois operandos
1111	Instrução de parada (HLT)

Fonte: WEBER, 2008. p. 183.

A seguir são descritas todas as instruções da arquitetura Cesar. As figuras mostram o formato utilizado no código da instrução, onde MMM representa o código do modo de endereçamento utilizado e RRR representa um registrador. Nas instruções de dois operandos, $R_1R_1R_1$ e $M_1M_1M_1$ representam, respectivamente, o registrador e o modo de endereçamento do primeiro operando, enquanto $R_2R_2R_2$ e $M_2M_2M_2$ representam o mesmo para o segundo operando. X significa que o bit pode ter qualquer valor e C indica o código específico das instruções de um e dois operandos e das instruções de desvio condicional.

A instrução NOP ocupa um byte e não realiza nenhuma operação. A execução desta instrução apenas incrementa o valor do registrador R7 em uma unidade, fazendo com que aponte para a próxima instrução.

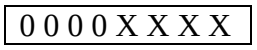


Figura 2.1: Instrução NOP

As instruções SCC e CCC ocupam um byte e apenas alteram os códigos de condição. A instrução SCC liga os códigos de condição especificados pelos valores dos últimos quatro bits do código da instrução. A instrução CCC funciona da mesma forma, porém desligando os códigos de condição especificados na instrução.

Figura 2.2: Instrução CCC

Figura 2.3: Instrução SCC

As instruções de desvio condicional ocupam dois bytes. Os últimos quatro bits do primeiro byte do código da instrução especificam a condição testada pela instrução. O segundo byte do código da instrução contém um deslocamento que é somado ao registrador R7, em complemento de dois, se o desvio for realizado, ou seja, se a condição testada pela instrução for verdadeira. A tabela 2.3 especifica os códigos das instruções, bem como as condições de desvio testadas por cada instrução.

Figura 2.4: Instruções de desvio condicional

Tabela 2.3: Instruções de desvio condicional

CCCC	Instrução	Condição de desvio
0000	BR (always)	Sempre verdadeira
0001	BNE (Not Equal)	Z=0
0010	BEQ (EQual)	Z=1
0011	BPL (PLus)	N=0
0100	BMI (MInus)	N=1
0101	BVC (oVerflow Clear)	V=0
0110	BVS (oVerflow Set)	V=1
0111	BCC (Carry Clear)	C=0
1000	BCS (Carry Set)	C=1
1001	BGE (Greater or Equal)	N=V
1010	BLT (Less Than)	N<>V
1011	BGT (Greater)	N=V e Z=0
1100	BLE (Less or Equal)	N<>V ou Z=1
1101	BHI (Higher)	C=0 e Z=0
1110	BLS (Lower or Same)	C=1 ou Z=1

Fonte: WEBER, 2008. p. 184.

A instrução de desvio incondicional JMP ocupa dois bytes. Sua execução carrega em R7 o endereço do operando, calculado a partir do modo de endereçamento e do registrador especificados no código da instrução. O modo de endereçamento registrador não deve ser usado nesta instrução. Se for usado, a instrução é interpretada como uma instrução NOP.

0 1 0 0 X X X X	X X M M M R R R

Figura 2.5: Instrução JMP

A instrução de controle de laço SOB ocupa dois bytes. A execução desta instrução decrementa em uma unidade o registrador indicado pelos últimos 3 bits do primeiro byte do código da instrução e, se o resultado for diferente de zero, o deslocamento especificado no segundo byte da instrução é subtraído do registrador R7. Se o resultado do decremento do registrador for igual a zero, o processador busca a instrução seguinte na memória.

0 1 0 1 X R R R	D D D D D D D D

Figura 2.6: Instrução SOB

A instrução de desvio para sub-rotina JSR ocupa dois bytes. Os últimos três bits do primeiro byte do código da instrução representam um registrador. Neste registrador, denominado registrador de ligação, será salvo o conteúdo de R7 antes do desvio para o endereço da sub-rotina, calculado através do modo de endereçamento e do registrador especificados no segundo byte do código da instrução. Assim como na instrução JMP, o modo registrador não deve ser utilizado nesta instrução.

0 1 1 0 X R R R	X X M M M R R R

Figura 2.7: Instrução JSR

A execução da instrução JSR dispara a seguinte sequência de transferências: um registrador temporário recebe o endereço da sub-rotina; o conteúdo do registrador de ligação é salvo na pilha; o registrador de ligação recebe o conteúdo do registrador R7; R7 recebe o endereço da sub-rotina armazenado no registrador temporário.

A instrução de retorno de sub-rotina RTS possui apenas um byte e carrega em R7 o conteúdo salvo no registrador de retorno, que corresponde ao registrador de ligação utilizado na instrução JSR. Isto significa que R7 recebe o endereço da instrução seguinte à instrução JSR. Esta operação é seguida pela atualização deste registrador com o seu conteúdo original, salvo na pilha pela instrução JSR.

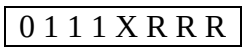


Figura 2.8: Instrução RTS

As instruções de um operando ocupam no mínimo dois bytes, que correspondem ao código da instrução mais o registrador e o modo de endereçamento utilizados. A tabela 2.4 resume o formato e o significado das instruções de um operando, bem como a forma como estas instruções afetam os códigos de condição. Para esta tabela, T indica que um código de condição é testado e ajustado conforme o resultado da operação, B indica que o carry é carregado com o valor do borrow-out, XOR para overflow indica que este recebe o valor da operação "ou exclusivo" entre os bits N e C, já atualizados pela execução da instrução, e MSB e LSB referem-se, respectivamente, ao bit mais significativo e menos significativo do operando.

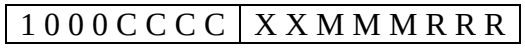


Figura 2.9: Instruções de um operando

Tabela 2.4: Instruções de um operando

CCCC	Instrução	Significado	N	Z	V	C
0000	CLR	op←0	T	T	0	0
0001	NOT	op←NOT op	T	T	1	0
0010	INC	op←op+1	T	T	T	T
0011	DEC	op←op-1	T	T	B	T
0100	NEG	op←-op	T	T	B	T
0101	TST	op←op	T	T	0	0
0110	ROR	op←SHR(C & op)	T	T	LSB	XOR
0111	ROL	op←SHL(op & C)	T	T	MSB	XOR
1000	ASR	op←SHR(MSB & op)	T	T	LSB	XOR
1001	ASL	op←SHL(op&0)	T	T	MSB	XOR
1010	ADC	op←op+C	T	T	T	T
1011	SBC	op←op-C	T	T	T	T

Fonte: WEBER, 2008. p. 186.

As instruções de dois operandos ocupam no mínimo dois bytes, podendo chegar até seis bytes dependendo dos modos de endereçamento utilizados no cálculo dos endereços dos operandos. Da mesma forma que a tabela 2.4, a tabela 2.5 resume o formato e o significado das instruções de dois operandos, bem como os códigos de condição afetados.

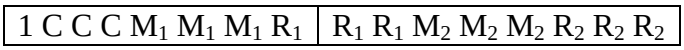


Figura 2.10: Instruções de dois operandos

Tabela 2.5: Instruções de dois operandos

CCC	Instrução	Significado	N	Z	V	C
001	MOV	op2←op1	T	T	0	-
010	ADD	op2←op2+op1	T	T	T	T
011	SUB	op2←op2-op1	T	T	T	B
100	CMP	op1-op2	T	T	T	B
101	AND	op2←op2 AND op1	T	T	0	-
110	OR	op2←op2 OR op1	T	T	0	-

Fonte: WEBER, 2008. p. 187.

Por fim, a instrução de parada HLT ocupa um byte e não realiza nenhuma operação.

Figura 2.11: Instrução HLT

A arquitetura Cesar possui também um sistema de entrada e saída rudimentar que mapeia os últimos 38 endereços da memória para os registradores de interface com dois periféricos: um teclado (endereços 65498 e 65499) e um visor alfanumérico de 36 posições (endereços 65500 a 65535). Na comunicação com o teclado, o byte 65499 armazena o código ASCII do último caractere digitado, enquanto o byte 65498 indica o recebimento do caractere do teclado com o valor 128. Um novo caractere é aceito somente depois do byte 65498 ter seu conteúdo zerado.

É importante ressaltar que qualquer operação envolvendo estes endereços de memória considera operandos de apenas um byte.

2.2 Ferramentas utilizadas

2.2.1 Placa FPGA

No desenvolvimento deste trabalho foi utilizada a placa Virtex-II Pro XC2VP30 da fabricante Xilinx. A figura 2.12 mostra uma imagem da placa, com seus principais componentes. Além do FPGA, foram utilizadas a saída de vídeo XSGA, uma das portas PS/2, para a conexão de um teclado, e dois botões. Para a programação do circuito na placa foi utilizada a porta USB.

A escolha deste modelo de FPGA se deu principalmente pela disponibilidade da mesma no Laboratório de Sistemas Embarcados do Instituto de Informática da UFRGS. As especificações desta placa vão muito além das necessárias para o desenvolvimento do trabalho.

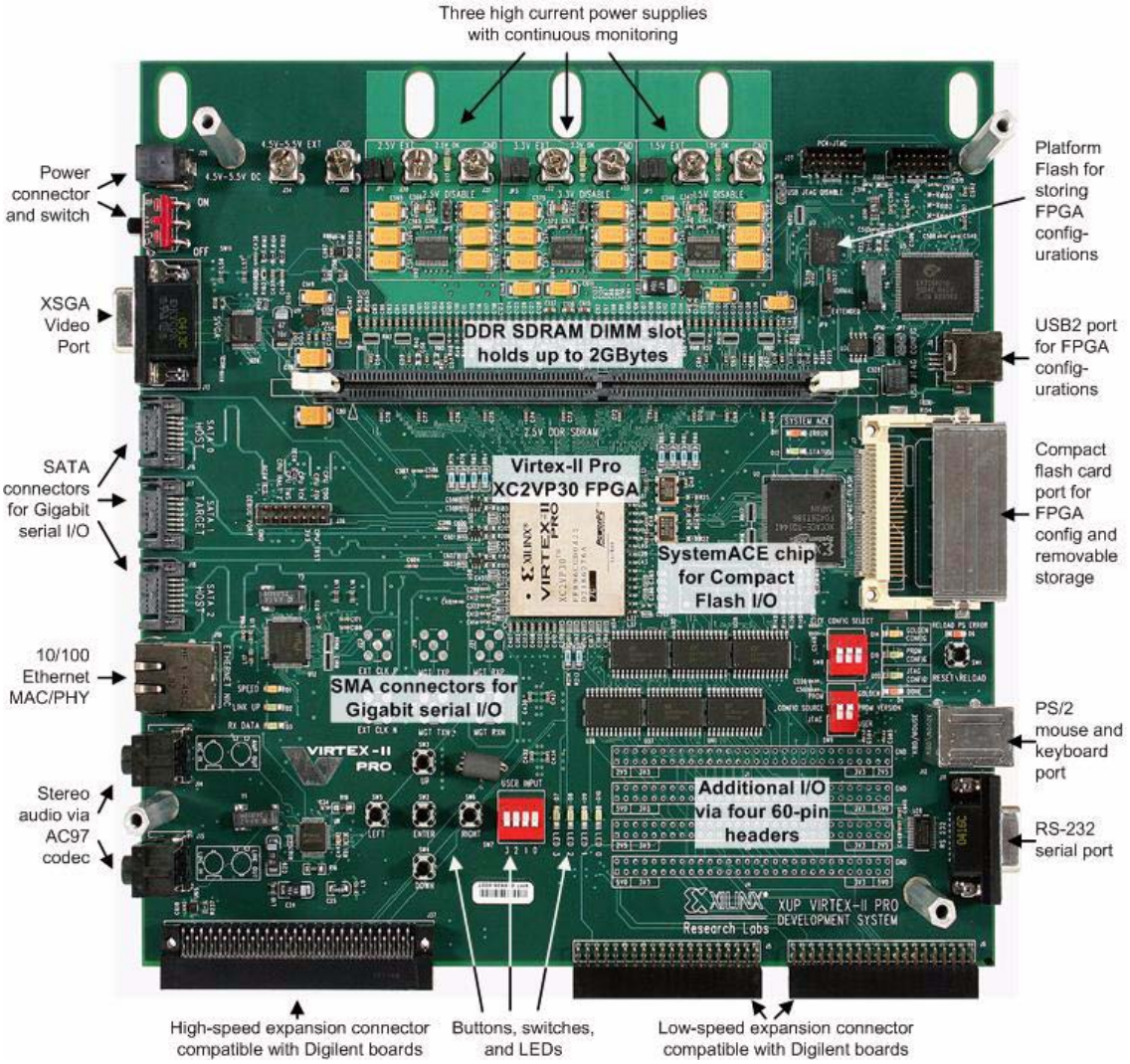


Figura 2.12: Placa Virtex-II Pro XC2VP30 (XILINX, 2009a)

A tabela 2.6 resume as principais características do FPGA que são relevantes na implementação do processador. A figura 2.13 mostra os elementos que compõem cada um dos Slices da placa. Embora no manual da placa conste que o FPGA contém 30816 Logic Cells, apenas 27392 estão disponíveis para a programação do circuito, ou seja, apenas as Logic Cells pertencentes aos Slices estão disponíveis. Mais detalhes sobre os recursos disponíveis na placa utilizada podem ser encontrados em (XILINX, 2007a), (XILINX, 2007b) e (XILINX, 2009a).

Tabela 2.6: Características do FPGA

Logic Cells	Slices	Distributed RAM (Kb)	BLOCK SelectRAM (18Kb blocks)	DCM	User I/O Pads
30816	13696	428	136	8	644
Logic Cell = (1) 4-input LUT + (1) Flip-Flop + Carry Logic.					

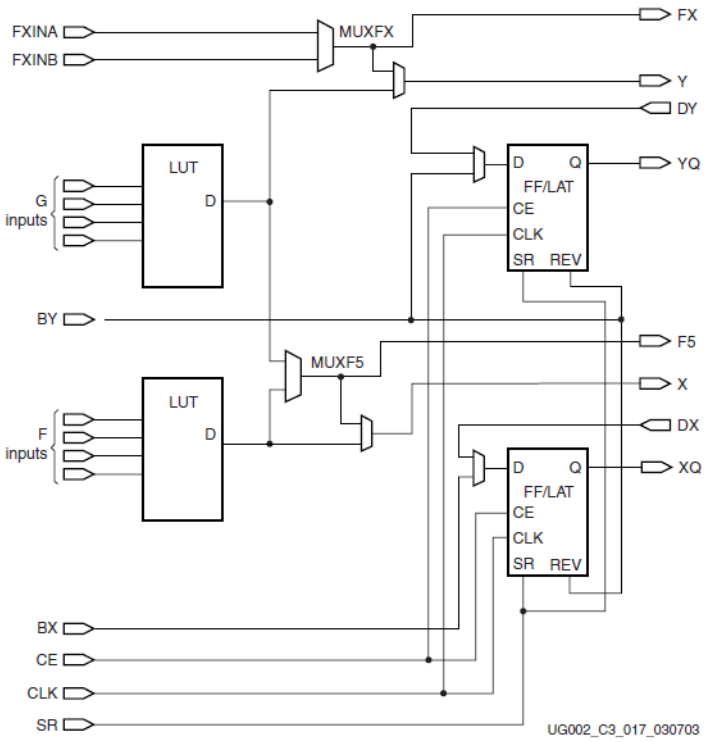


Figura 2.13: Componentes de um dos Slices disponíveis na placa (XILINX, 2007b)

2.2.2 Software

A programação em VHDL do processador foi feita com o auxílio da ferramenta ISE (Integrated Software Environment) versão 10.1, disponível para download no site da Xilinx (XILINX, 2009b). Embora este software já esteja disponível na sua versão 12.1, esta versão não manteve a compatibilidade com a placa utilizada neste trabalho, que é um modelo mais antigo. Toda a implementação do Cesar, desde a síntese lógica, que examina todos os arquivos VHDL em conjunto e otimiza o circuito para o modelo do FPGA utilizado (PERRY, 2002), até o roteamento e posicionamento do circuito na placa, é feita com o uso desta ferramenta. Os primeiros testes funcionais foram realizados com a ferramenta ISE Simulator (XILINX, 2008b).

Após a síntese lógica do circuito, o mapeamento das entradas e saídas do circuito para os pinos corretos, correspondentes aos botões, teclado e vídeo, foi feito com o software PACE (Pinout and Area Constraints Editor) que cria um arquivo .UCF (User Constraint File) (XILINX, 2002) e (XILINX, 2008a). A partir da síntese lógica em conjunto com o arquivo UCF, a ferramenta ISE faz a implementação do projeto criado, através da tradução, mapeamento, posicionamento e roteamento do circuito.

Ao final deste processo, a ferramenta gera um arquivo .BIT que é utilizado pelo software IMPACT 10.1 para a programação efetiva do circuito no FPGA. Esta programação é feita com o auxílio de um cabo USB, conectado ao computador e à placa.

Além dos softwares da Xilinx, para o estudo e descrição RTL da arquitetura Cesar, foram utilizados o Montador Daedalus Multi Assembler versão 1.0.2.4 e o Simulador WCesar16 versão 1.1.2.0. Mais informações sobre estes softwares podem ser encontradas em (WEBER, 2008).

3 IMPLEMENTAÇÃO

O processo de implementação em hardware do processador CESAR começou com a análise do conjunto de instruções e dos modos de endereçamento da arquitetura. A partir desta análise, foi realizada a descrição do processador em nível de transferência de registradores (*Register Transfer Level* ou RTL). Esta descrição se constitui na etapa mais importante no projeto do hardware de um processador, uma vez que todo o funcionamento do sistema é especificado através de operações e transferências de dados entre registradores. A descrição RTL do processador Cesar foi realizada ainda no TG1 e pode ser vista nas tabelas localizadas no artigo do apêndice A.

A partir da descrição RTL, a implementação em hardware de um processador consiste na tradução desta descrição RTL para uma descrição VHDL. No caso do Cesar, isto implica na codificação de uma máquina de estados (Finite State Machine ou FSM) que especifica quando e como cada uma das operações e transferências entre registradores deve ser realizada. A respeito disto, (HWANG, 2005) apresenta duas metodologias distintas para a implementação de um processador em VHDL: descrição estrutural e descrição comportamental.

Na descrição estrutural, todos os elementos necessários para o funcionamento do computador, como registradores, multiplexadores, decodificadores, unidade aritmética e lógica, bem como as interconexões entre estes elementos, devem ser projetados à mão. O circuito resultante deste processo é conhecido como caminho de dados (datapath). Além disto, deve ser projetado um módulo de controle que gere os sinais necessários para que as operações e transferências de dados do datapath ocorram corretamente.

Na descrição comportamental, ao invés de projetar-se manualmente o datapath do circuito, as operações e transferências entre registradores são codificadas explicitamente, em conjunto com a máquina de estados.

Mais adiante serão apresentadas duas implementações do processador Cesar: comportamental e estrutural. A figura 3.1 revela as etapas mais importantes do projeto realizado. Como se pode perceber, primeiro foi realizada a implementação comportamental. A partir desta implementação, foi projetado um datapath e um bloco de controle para a versão estrutural. Para ambas as versões, os sistemas de memória e de entrada e saída, descritos a seguir, são conceitualmente idênticos.

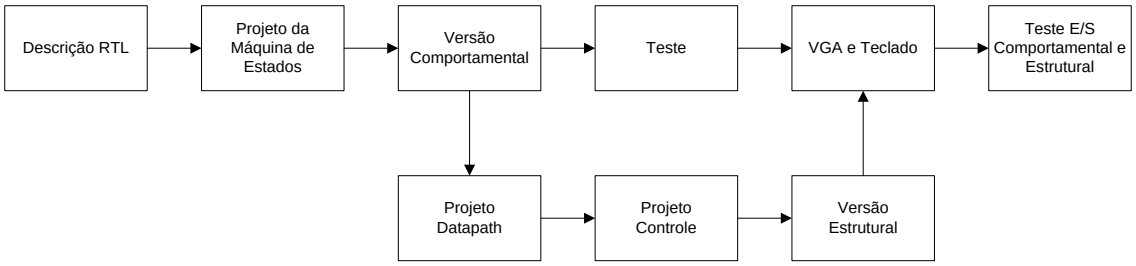


Figura 3.1: Etapas da implementação

3.1 Sistema de Memória

A figura 3.2 mostra um dos blocos de memória, denominados de Block SelectRAM, disponíveis na placa da Xilinx. No total a placa dispõe de 136 blocos de 18 Kbits, cada um contendo duas portas de acesso totalmente independentes. Cada porta, por sua vez, possui sua própria entrada de clock, entrada e saída de dados independentes, além dos sinais de Enable, Write Enable e Set/Reset. As operações de leitura ou escrita da memória são realizadas em uma única borda do clock.

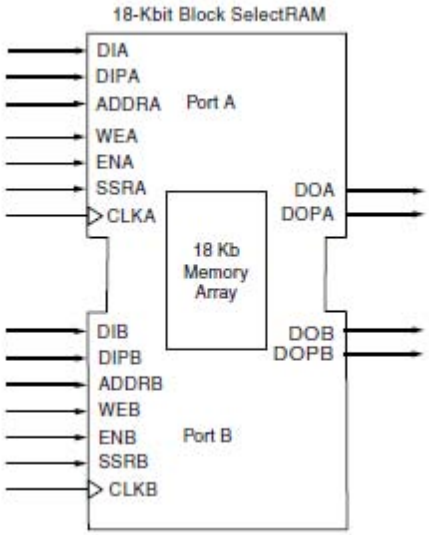


Figura 3.2: Um dos blocos de memória disponíveis na placa

Para a memória do Cesar são utilizados 32 blocos de memória concatenados. Com a utilização da ferramenta Core Generator, o processo de criação da memória foi bastante simples e direto. Esta ferramenta possibilita a criação de diferentes tipos de memória, através das várias opções existentes. Foi escolhida uma memória do tipo True Dual Port RAM, com duas portas de acesso independentes, largura de dados de um byte e barramento de endereços de 16 bits. Assim, a memória possui no total 65536 posições de um byte, já que, embora utilize operandos de 16 bits, o Cesar endereça a memória a byte. Também foi utilizada a opção Always Enabled, que dispensa o sinal de Enable.

A ferramenta também possibilita a escolha de três modos de operação, cuja diferença está apenas no valor colocado na saída da memória durante uma operação de escrita. Foi utilizada a opção Write First. Neste modo de operação, um dado escrito numa posição de memória determinada pelo registrador de endereços, também é colocado simultaneamente no registrador de saída da memória. Na verdade o modo de operação não é relevante para a memória do Cesar, uma vez que jamais são realizadas operações de escrita e leitura num mesmo intervalo de clock.

Em uma operação de leitura as duas portas da memória são utilizadas para a busca de dois bytes de cada vez. Isto permite que um código ou um operando de uma instrução seja buscado através de um único acesso à memória. O mesmo vale para operações de escrita na memória. No caso de um acesso a um dos endereços de entrada e saída, quando um operando é lido da memória em um endereço maior do que 65497, o multiplexador MUX_RDM formata adequadamente o dado para ser usado pela ULA. Já em uma operação de escrita na memória em um desses endereços, apenas o primeiro byte é escrito na memória, na posição apontada pelo registrador REM1. Isto é possível porque existem dois sinais de escrita W1 e W2 independentes, um para cada porta de acesso à memória.

3.2 Sistema de Entrada e Saída

A opção por implementar o sistema de entrada e saída do Cesar adicionou uma considerável complexidade ao projeto, como já era esperado. O maior desafio, na verdade, residiu no fato de que as operações realizadas sobre operandos localizados nas últimas 38 posições de memória devem ser feitas a byte, ou seja, considerando operandos de apenas um byte, ao contrário do padrão do processador, que utiliza operandos de 16 bits. Primeiramente se pensou em implementar todas as operações com operandos de um byte em separado, isto é, cada operação realizada pelo processador teria também uma versão para operandos de um byte. Esta opção, contudo, geraria um circuito consideravelmente maior, além de ser desnecessária.

A solução encontrada foi a utilização de um multiplexador especial, que é responsável pela interface de leitura e escrita de dados do processador Cesar com a memória. Este multiplexador, denominado MUX_RDM, é responsável pela formatação adequada de um dado a ser lido ou escrito na memória, de acordo como o endereço em que este dado se localiza. Assim, embora possua apenas duas entradas, ligadas às portas de saída da memória e ao registrador temporário RT, existem quatro possíveis saídas para este multiplexador, selecionadas através do sinal SEL_RDM.

Para os valores de SEL_RDM iguais a 0 (“00”) e 1 (“01”) a saída do multiplexador simplesmente seleciona os conteúdos de uma de suas entradas, da memória, ou do registrador temporário RT, respectivamente. No caso de um operando de um byte lido da memória, ou seja, o operando está num endereço maior do que 65497, o multiplexador formata o operando que será utilizado pela ULA, concatenando os oito bits mais significativos do registrador de saída da memória a oito bits com valor zero. Assim, o operando utilizado pela ULA terá o valor equivalente a 00000000&MEM(15:8). Esta opção é ativada com SEL_RDM igual a 2 (“10”).

Já quando o endereço onde será escrito um dado for um dos endereços de entrada e saída, o multiplexador formata o dado concatenando os oito bits menos significativos do registrador RT, a oito bits com valor zero. Assim a memória recebe o dado no formato RT(7:0)&00000000, onde RT, como já mencionado, é o registrador temporário, onde são armazenados os valores das operações aritméticas e lógicas realizadas pelo processador. Neste caso, apenas o sinal W1 é ativado, escrevendo na memória apenas o primeiro byte.

Além disto, não é possível fazer um mapeamento direto de posições de memória para uma interface de entrada e saída. Como já havia sido mencionado no artigo em anexo, foram utilizados 38 registradores de 8 bits que espelham o conteúdo das últimas

38 posições de memória do Cesar. Assim, qualquer operação de escrita numa destas posições de memória, também atualiza o valor armazenado em um destes registradores.

Para a criação destes registradores foi utilizada uma estrutura de array na versão comportamental. Desta forma, o registrador RES(0) representa o conteúdo da posição de memória 65498 e assim por diante. Os registradores RES(2) a RES(37) representam os dados que devem ser exibidos no display do Cesar e, portanto, são mapeados para as saídas V00 a V35, respectivamente.

Outra questão importante se refere ao processamento da entrada de dados através do teclado. Da especificação da arquitetura Cesar, descrita no capitulo anterior, verifica-se que, enquanto o conteúdo do endereço de memória 65498 estiver zerado, o processador deve monitorar constantemente a presença de um dado vindo do teclado, isto é, se alguma tecla foi pressionada. Este monitoramento é feito independentemente das instruções que estejam sendo executadas. Já o processamento de um dado vindo do teclado, consiste em colocar o código ASCII da tecla pressionada no endereço 65499 e o valor 128 no endereço 65498. Isto impede que outro dado seja recebido do teclado até que o conteúdo armazenado no endereço 65498 seja zerado.

O processamento de um dado vindo do teclado foi implementado através de 3 sinais de controle e um mecanismo de interrupção. O sinal de entrada T2 recebe o código ASCII da tecla pressionada, enquanto o sinal de entrada T1 indica que uma tecla foi pressionada. O terceiro sinal utilizado é o sinal de saída ACK_TCL, que corresponde ao bit mais significativo do registrador RES(0). Este sinal indica que o processamento do dado vindo do teclado foi realizado, evitando que uma mesma tecla seja processada várias vezes.

O mecanismo de interrupção funciona da seguinte forma. No intervalo entre a execução de duas instruções, no estado BUSCA1, o processador testa duas condições. Se as duas condições forem verdadeiras simultaneamente, isto é, T1=1 e RES(0)= 0, o fluxo de execução desvia para o estado INT1. Neste estado, o registrador RES(0) recebe o valor 128, enquanto RES(1) recebe o valor da entrada T2. Estes valores também são escritos na memória, nos endereços 65498 e 65499 respectivamente. Após a realização destas operações, o processador retorna ao estado BUSCA1, para buscar o código da instrução seguinte.

Há ainda dois outros sinais de entrada, que são mapeados para dois dos botões disponíveis na placa. O sinal de RESET, quando ativado, faz com que o processador volte ao estado INICIO. Assim, o PC(R7), bem como o conteúdo de todos os outros 7 registradores, é zerado. Já o sinal de RESUME faz com que o processador, caso esteja no estado de HALT, saia deste estado e busque a próxima instrução. O PC (R7), neste caso, aponta para o endereço de memória seguinte ao do endereço da instrução que fez com que o processador entrasse no estado HALT. A figura 3.3 mostra todas as entradas e saídas do processador.

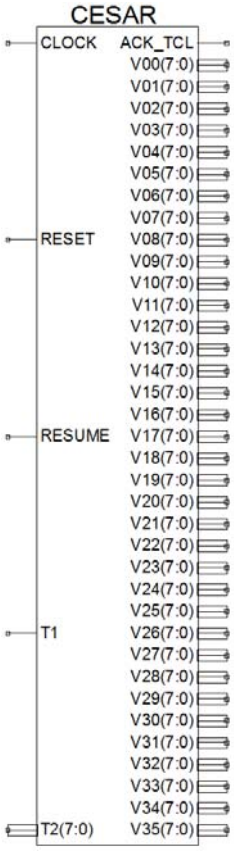


Figura 3.3: Portas de Entrada e Saída do processador Cesar

3.3 Implementação Comportamental (Cesar 1)

A implementação comportamental do processador Cesar na linguagem VHDL está codificada, basicamente, em três entidades: uma memória, um multiplexador, denominado MUX_RDM, que realiza a interface do processador com a memória, e a entidade principal CESAR1, que interconecta as outras duas e implementa a máquina de estados que controla o funcionamento do processador. Um esboço desta máquina de estados é mostrado na figura 3.4.

Como a arquitetura CESAR possui diferentes formatos e tamanhos de instruções, além de oito modos de endereçamento, existem várias possibilidades para a construção desta máquina de estados. O modelo utilizado segue os moldes da descrição do processador hipotético Neander, encontrada em (Weber, 2008). Embora a fase de busca da instrução na memória seja comum a todas as instruções, já no estado de decodificação as operações que devem ser realizadas diferem de acordo com a instrução sendo executada. Isto pode ser visto nas tabelas 3.1 e 3.2.

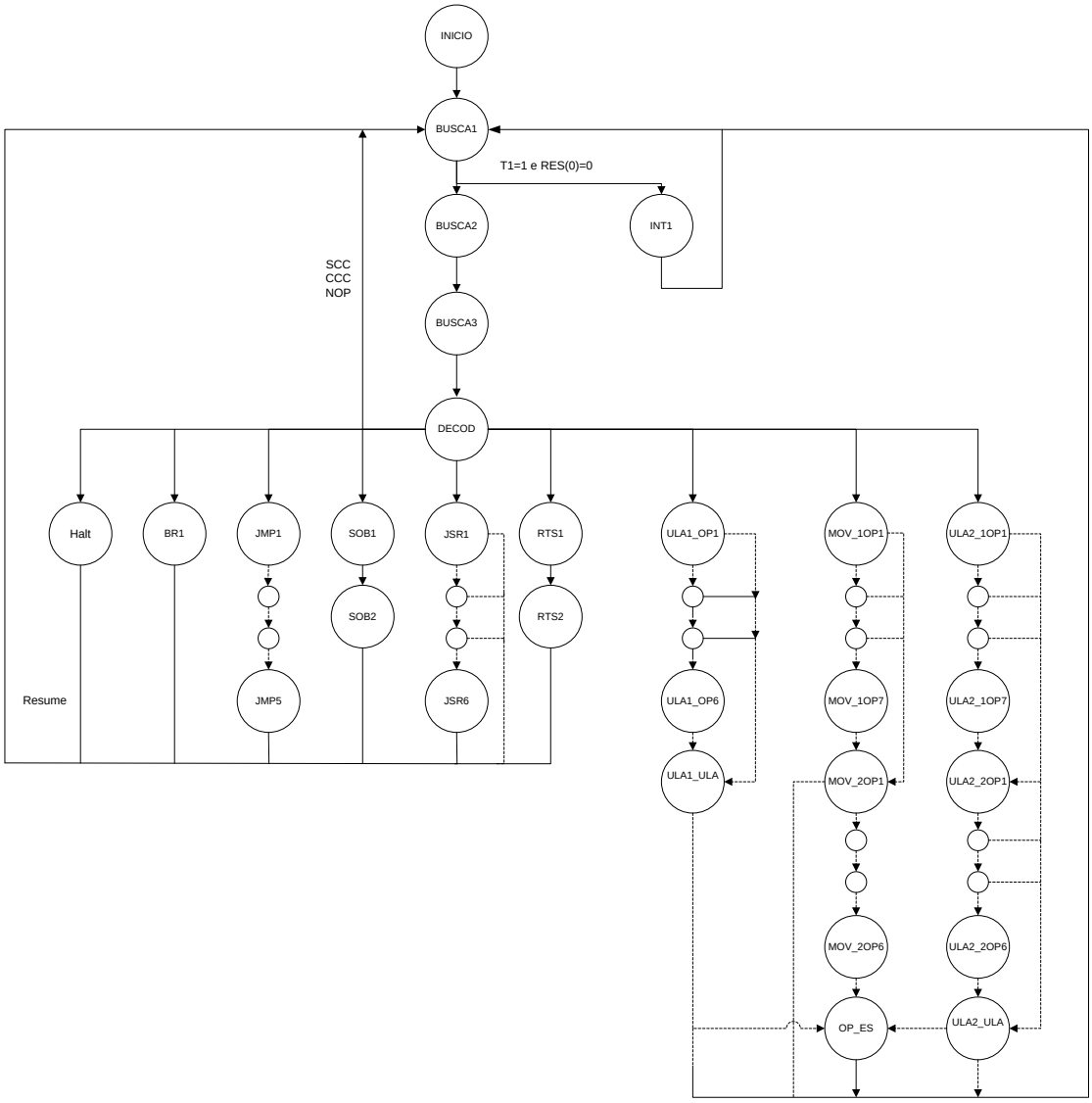


Figura 3.4: Esboço da Máquina de Estados

Em todas as tabelas, *RDM←MEM* ou *RDM←0&MEM* significam, respectivamente, uma leitura de 16 bits ou de 1 byte da memória. Da mesma forma, *RDM←RT* ou *RDM←RT&0* representam a escrita de 16 bits ou 1 byte na memória. Os códigos de condição são atualizados no estado MOV_2OP1 na instrução MOV e nos estados ULA1_ULA e ULA2_ULA nas instruções de um e dois operandos, respectivamente.

Tabela 3.1: Estados da busca de uma instrução.

BUSCA1	REM1←R7 REM1←R7+1
BUSCA2	*RDM←MEM*
BUSCA3	RI←RDM

Tabela 3.2: Estado da decodificação de uma instrução

DECOD		
INSTRUÇAO	**OPERAÇÃO**	**PRÓXIMO ESTADO**
NOP	R7←R7+1	BUSCA1
CCC	R7←R7+1 CC←CC AND (NOT (RDM(11:8)))	BUSCA1
SCC	R7←R7+1 CC←CC OR RDM(11:8)	BUSCA1
HLT	R7←R7+1	HALT
MOV	R7←R7+2	MOV_1OP1
ADD, SUB, CMP, AND, OR	R7←R7+2	ULA2_1OP1
INST_ 1OP	R7←R7+2	IF REG ULA1_ULA ELSE ULA1_OP1
INST_BRANCH	R7←R7+2	BR1
JMP	R7←R7+2	JMP1
SOB	R7←R7+2	SOB1
JSR	R7←R7+2	JSR1
RTS	R7←R7+2	RTS1

Na máquina de estados projetada, a subdivisão de estados foi feita, principalmente, de acordo com a similaridade dos códigos das instruções. Assim, as instruções de um operando, por exemplo, são executadas através de uma única sequência de estados, como mostrado nas tabelas 3.3 e 3.4. Os estados de ULA1_OP1 até ULA1_OP6 realizam a busca do operando e as operações e transferências de dados realizadas em cada um destes estados difere de acordo com o modo de endereçamento utilizado. No estado ULA1_ULA, comum a todos os modos de endereçamento, é efetivamente executada a operação codificada na instrução. O estado OP_ES atualiza o conteúdo dos registradores RES, correspondentes ao visor, se uma atualização for necessária.

Tabela 3.3: Instruções de um operando nos modos REG, RPI, RPD e INX

ESTADO	REG	RPI	RPD	INX
ULA1_OP1		REM1←R REM2←R+1	R←R-2	REM1←R7 REM2←R7+1
ULA1_OP2		R←R+2 IF REM1>65497 *RDM←0&MEM* ELSE *RDM←MEM*	REM1←R REM2←R+1	R7←R7+2 *RDM←MEM*
ULA1_OP3			IF REM1>65497 *RDM←0&MEM* ELSE *RDM←MEM*	REM1←R+RDM REM2←R+RDM+1
ULA1_OP4				IF REM1>65497 *RDM←0&MEM* ELSE *RDM←MEM*
ULA1_ULA	R←RT($_{OP}$)R	RT←OP(RDM) IF REM1>65497 *RDM←RT&0* ELSE *RDM←RT*	RT←OP(RDM) IF REM1>65497 *RDM←RT&0* ELSE *RDM←RT*	RT←OP(RDM) IF REM1>65497 *RDM←RT&0* ELSE *RDM←RT*
OP_ES		IF REM1>65497 RES←RT(7..0)	IF REM1>65497 RES←RT(7..0)	IF REM1>65497 RES←RT(7..0)

Tabela 3.4: Instruções de um operando nos modos IND, PII, PDI e IXI

ESTADO	IND	PII	PDI	IXI
ULA1_OP1	REM1←R REM2←R+1	REM1←R REM2←R+1	R←R-2	REM1←R7 REM2←R7+1
ULA1_OP2	IF REM1>65497 *RDM←0&MEM* ELSE *RDM←MEM*	R←R+2 *RDM←MEM*	REM1←R REM2←R+1	R7←R7+2 *RDM←MEM*
ULA1_OP3		REM1←RDM REM2←RDM+1	*RDM←MEM*	REM1←R+RDM REM2←R+RDM+1
ULA1_OP4		IF REM1>65497 *RDM←0&MEM* ELSE *RDM←MEM*	REM1←RDM REM2←RDM+1	*RDM←MEM*
ULA1_OP5			IF REM1>65497 *RDM←0&MEM* ELSE *RDM←MEM*	REM1←RDM REM2←RDM+1
ULA1_OP6				IF REM1>65497 *RDM←0&MEM* ELSE *RDM←MEM*
ULA1_ULA	RT←OP(RDM) IF REM1>65497 *RDM←RT&0* ELSE *RDM←RT*	RT←OP(RDM) IF REM1>65497 *RDM←RT&0* ELSE *RDM←RT*	RT←OP(RDM) IF REM1>65497 *RDM←RT&0* ELSE *RDM←RT*	RT←OP(RDM) IF REM1>65497 *RDM←RT&0* ELSE *RDM←RT*
OP_ES	IF REM1>65497 RES←RT(7..0)	IF REM1>65497 RES←RT(7..0)	IF REM1>65497 RES←RT(7..0)	IF REM1>65497 RES←RT(7..0)

Para as instruções de dois operandos foi feita ainda uma subdivisão de acordo com a sequência de operações necessárias para a execução da instrução. Desta forma, a instrução MOV é executada por uma sequência de estados diferente da utilizada nas demais instruções de dois operandos. A instrução MOV é mostrada nas tabelas 3.5 e 3.6. Assim como nas tabelas referentes às demais instruções de dois operandos R_F é o registrador utilizado no cálculo do endereço do operando fonte e R_D do operando destino.

Tabela 3.5: Instrução MOV nos modos REG, RPI, RPD e INX

ESTADO	REG	RPI	RPD	INX
MOV_1OP1	RT←R_F	REM1←R_F REM2←R_F+1	R_F←R_F-2	REM1←R7 REM2←R7+1
MOV_1OP2		R_F←R_F+2 IF REM1>65497 *RDM←0&MEM* ELSE *RDM←MEM*	REM1←R_F REM2←R_F+1	R7←R7+2 *RDM←MEM*
MOV_1OP3		RT←RDM	IF REM1>65497 *RDM←0&MEM* ELSE *RDM←MEM*	REM1←R_F+RDM REM2←R_F+RDM+1
MOV_1OP4			RT←RDM	IF REM1>65497 *RDM←0&MEM* ELSE *RDM←MEM*
MOV_1OP5				RT←RDM
MOV_2OP1	R_D←RT	REM1←R_D REM2←R_D+1	R_D←R_D-2	REM1←R7 REM2←R7+1
MOV_2OP2		R_D←R_D+2 IF REM1>65497 *RDM←RT&0* ELSE	REM1←R_D REM2←R_D+1	R7←R7+2 *RDM←MEM*

		RDM←RT		
MOV_2OP3			IF REM1>65497 *RDM←RT&0* ELSE *RDM←RT*	REM1←R_D+RDM REM2←R_D+RDM+1
MOV_2OP4				IF REM1>65497 *RDM←RT&0* ELSE *RDM←RT*
MOV_2OP5				
MOV_2OP6				
OP_ES		IF REM1>65497 RES←RT(7..0)	IF REM1>65497 RES←RT(7..0)	IF REM1>65497 RES←RT(7..0)

Tabela 3.6: Instrução MOV nos modos IND, PII, PDI e IXI

ESTADO	IND	PII	PDI	IXI
MOV_1OP1	REM1←R_F REM2←R_F+1	REM1←R_F REM2←R_F+1	R_F←R_F-2	REM1←R7 REM2←R7+1
MOV_1OP2	IF REM1>65497 *RDM←0&MEM* ELSE *RDM←MEM*	R_F←R_F+2 *RDM←MEM*	REM1←R_F REM2←R_F+1	R7←R7+2 *RDM←MEM*
MOV_1OP3	RT←RDM	REM1←RDM REM2←RDM+1	*RDM←MEM*	REM1←R_F+RDM REM2←R_F+RDM+1
MOV_1OP4		IF REM1>65497 *RDM←0&MEM* ELSE *RDM←MEM*	REM1←RDM REM2←RDM+1	*RDM←MEM*
MOV_1OP5		RT←RDM	IF REM1>65497 *RDM←0&MEM* ELSE *RDM←MEM*	REM1←RDM REM2←RDM+1
MOV_1OP6			RT←RDM	IF REM1>65497 *RDM←0&MEM* ELSE *RDM←MEM*
MOV_1OP7				RT←RDM
MOV_2OP1	REM1←R_D REM2←R_D+1	REM1←R_D REM2←R_D+1	R_D←R_D-2	REM1←R7 REM2←R7+1
MOV_2OP2	IF REM1>65497 *RDM←RT&0* ELSE *RDM←RT*	R_D←R_D+2	REM1←R_D REM2←R_D+1	R7←R7+2
MOV_2OP3		REM1←RDM REM2←RDM+1	*RDM←MEM*	REM1←R+RDM REM2←R+RDM+1
MOV_2OP4		IF REM1>65497 *RDM←RT&0* ELSE *RDM←RT*	REM1←RDM REM2←RDM+1	*RDM←MEM*
MOV_2OP5			IF REM1>65497 *RDM←RT&0* ELSE *RDM←RT*	REM1←RDM REM2←RDM+1
MOV_2OP6				IF REM1>65497 *RDM←RT&0* ELSE *RDM←RT*
OP_ES	IF REM1>65497 RES←RT(7..0)	IF REM1>65497 RES←RT(7..0)	IF REM1>65497 RES←RT(7..0)	IF REM1>65497 RES←RT(7..0)

Nas demais instruções de dois operandos, vistas nas tabelas 3.7 e 3.8, o estado ULA2_ULA, comum a todos os modos de endereçamento, executa efetivamente a operação codificada na instrução. Embora não seja mostrado, a instrução CMP não altera a memória, ou seja, o resultado não é salvo.

Tabela 3.7: Instruções de dois operandos nos modos REG, RPI, RPD e INX

ESTADO	REG	RPI	RPD	INX
ULA2_1OP1	RT←R_F	REM1←R_F REM2←R_F+1	R_F←R_F-2	REM1←R7 REM2←R7+1
ULA2_1OP2		R_F←R_F+2 IF REM1>65497 *RDM←0&MEM* ELSE *RDM←MEM*	REM1←R_F REM2←R_F+1	R7←R7+2 *RDM←MEM*
ULA2_1OP3		RT←RDM	IF REM1>65497 *RDM←0&MEM* ELSE *RDM←MEM*	REM1←R_F+RDM REM2←R_F+RDM+1
ULA2_1OP4			RT←RDM	IF REM1>65497 *RDM←0&MEM* ELSE *RDM←MEM*
ULA2_1OP5				RT←RDM
ULA2_2OP1		REM1←R_D REM2←R_D+1	R_D←R_D-2	REM1←R7 REM2←R7+1
ULA2_2OP2		R_D←R_D+2 IF REM1>65497 *RDM←0&MEM* ELSE *RDM←MEM*	REM1←R_D REM2←R_D+1	R7←R7+2
ULA2_2OP3			IF REM1>65497 *RDM←0&MEM* ELSE *RDM←MEM*	REM1←R_D+RDM REM2←R_D+RDM+1
ULA2_2OP4				IF REM1>65497 *RDM←0&MEM* ELSE *RDM←MEM*
ULA2_2OP5				
ULA2_2OP6				
ULA2_ULA	R←RT($_{OP}$)R	RT←RT($_{OP}$)RDM IF REM1>65497 *RDM←RT&0* ELSE *RDM←RT*	RT←RT($_{OP}$)RDM IF REM1>65497 *RDM←RT&0* ELSE *RDM←RT*	RT←RT($_{OP}$)RDM IF REM1>65497 *RDM←RT&0* ELSE *RDM←RT*
OP_ES		IF REM1>65497 RES←RT(7..0)	IF REM1>65497 RES←RT(7..0)	IF REM1>65497 RES←RT(7..0)

Tabela 3.8: Instruções de dois operandos nos modos IND, PII, PDI e IXI

ESTADO	IND	PII	PDI	IXI
ULA2_1OP1	REM1←R_F REM2←R_F+1	REM1←R_F REM2←R_F+1	R_F←R_F-2	REM1←R7 REM2←R7+1
ULA2_1OP2	IF REM1>65497 *RDM←0&MEM* ELSE *RDM←MEM*	R_F←R_F+2 *RDM←MEM*	REM1←R_F REM2←R_F+1	R7←R7+2 *RDM←MEM*
ULA2_1OP3	RT←RDM	REM1←RDM REM2←RDM+1	*RDM←MEM*	REM1←R_F+RDM REM2←R_F+RDM+1
ULA2_1OP4		IF REM1>65497 *RDM←0&MEM* ELSE *RDM←MEM*	REM1←RDM REM2←RDM+1	*RDM←MEM*
ULA2_1OP5		RT←RDM	IF REM1>65497 *RDM←0&MEM* ELSE *RDM←MEM*	REM1←RDM REM2←RDM+1

ULA2_1OP6			RT←RDM	IF REM1>65497 *RDM←0&MEM* ELSE *RDM←MEM*
ULA2_1OP7				RT←RDM
ULA2_2OP1	REM1←R_D REM2←R_D+1	REM1←R_D REM2←R_D+1	R_D←R_D-2	REM1←R7 REM2←R7+1
ULA2_2OP2	IF REM1>65497 *RDM←0&MEM* ELSE *RDM←MEM*	R_D←R_D+2 *RDM←MEM*	REM1←R_D REM2←R_D+1	R7←R7+2 *RDM←MEM*
ULA2_2OP3		REM1←RDM REM2←RDM+1	*RDM←MEM*	REM1←R_D+RDM REM2←R_D+RDM+1
ULA2_2OP4		IF REM1>65497 *RDM←0&MEM* ELSE *RDM←MEM*	REM1←RDM REM2←RDM+1	*RDM←MEM*
ULA2_2OP5			IF REM1>65497 *RDM←0&MEM* ELSE *RDM←MEM*	REM1←RDM REM2←RDM+1
ULA2_2OP6				IF REM1>65497 *RDM←0&MEM* ELSE *RDM←MEM*
ULA2_ULA	RT←RT($_{OP}$)RDM IF REM1>65497 *RDM←RT&0* ELSE *RDM←RT*	RT←RT($_{OP}$)RDM IF REM1>65497 *RDM←RT&0* ELSE *RDM←RT*	RT←RT($_{OP}$)RDM IF REM1>65497 *RDM←RT&0* ELSE *RDM←RT*	RT←RT($_{OP}$)RDM IF REM1>65497 *RDM←RT&0* ELSE *RDM←RT*
OP_ES	IF REM1>65497 RES←RT(7..0)	IF REM1>65497 RES←RT(7..0)	IF REM1>65497 RES←RT(7..0)	IF REM1>65497 RES←RT(7..0)

As tabelas 3.9 e 3.10 mostram os estados envolvidos na execução da instrução JMP. O modo de endereçamento registrador (REG) não é válido para esta instrução e, se usado, a instrução funciona como um NOP.

Tabela 3.9: Instrução JMP nos modos REG, RPI, RPD e INX

ESTADO	REG	RPI	RPD	INX
JMP1		R7←R R←R+2	R←R-2	REM1←R7 REM2←R7+1
JMP2			R7←R	R7←R7+2 *RDM←MEM*
JMP3				R7←R+RDM

Tabela 3.10: Instrução JMP nos modos IND, PII, PDI e IXI

ESTADO	IND	PII	PDI	IXI
JMP1	R7←R	REM1←R REM2←R+1 R←R+2	R←R-2	REM1←R7 REM2←R7+1
JMP2		*RDM←MEM*	REM1←R REM2←R+1	R7←R7+2 *RDM←MEM*
JMP3		R7←RDM	*RDM←MEM*	REM1←R+RDM REM2←R+RDM+1
JMP4			R7←RDM	*RDM←MEM*
JMP5				R7←RDM

As tabelas 3.11 e 3.12 mostram os estados envolvidos na execução da instrução JSR. Aqui também o modo de endereçamento registrador (REG), não é válido. R_x representa o registrador de ligação, que é salvo na pilha e recebe o conteúdo do registrador R7(PC)

antes do desvio. O registrador temporário RT2 é usado apenas nesta instrução e recebe temporariamente o endereço da sub-rotina, antes do endereço ser copiado para R7.

Tabela 3.11: Instrução JSR nos modos REG, RPI, RPD e INX

ESTADO	REG	RPI	RPD	INX
JSR1		RT2←R R←R+2	R←R-2	REM1←R7 REM2←R7+1
JSR2		R6←R6-2	RT2←R R6←R6-2	R7←R7+2 *RDM←MEM*
JSR3		REM1←R6 REM2←R6+1 RT←R_X *RDM←RT* R_X←R7 R7←RT2	REM1←R6 REM2←R6+1 RT←R_X *RDM←RT* R_X←R7 R7←RT2	RT2←R+RDM R6←R6-2
JSR4				REM1←R6 REM2←R6+1 RT←R_X *RDM←RT* R_X←R7 R7←RT2

Tabela 3.12: Instrução JSR nos modos IND, PII, PDI e IXI

ESTADO	IND	PII	PDI	IXI
JSR1	RT2←R R6←R6-2	REM1←R REM2←R+1 R←R+2	R←R-2	REM1←R7 REM2←R7+1
JSR2	REM1←R6 REM2←R6+1 RT←R_X *RDM←RT* R_X←R7 R7←RT2	*RDM←MEM*	REM1←R REM2←R+1	R7←R7+2 *RDM←MEM*
JSR3		RT2←RDM R6←R6-2	*RDM←MEM*	REM1←R+RDM REM2←R+RDM+1
JSR4		REM1←R6 REM2←R6+1 RT←R_X *RDM←RT* R_X←R7 R7←RT2	RT2←RDM R6←R6-2	*RDM←MEM*
JSR5			REM1←R6 REM2←R6+1 RT←R_X *RDM←RT* R_X←R7 R7←RT2	RT2←RDM R6←R6-2
JSR6				REM1←R6 REM2←R6+1 RT←R_X *RDM←RT* R_X←R7 R7←RT2

Na tabela 3.13, D representa o deslocamento, dado pelo segundo byte do código da instrução, que é somado ao valor de R7 ou não, conforme a condição testada seja verdadeira ou falsa, respectivamente.

Tabela 3.13: Instruções de desvio condicional

ESTADO	OPERAÇÕES
BR1	IF TRUE R7← R7+D ELSE R7←R7

As tabelas 3.14, 3.15 e 3.16 mostram, respectivamente, as operações realizadas pelas instruções SOB, RTS e por uma interrupção do teclado.

Tabela 3.14: Instrução de controle de laço SOB

ESTADO	OPERAÇÕES
SOB1	R← R-1
SOB2	IF R=0 R7← R7 ELSE R7←R7-RDM(7..0)

Tabela 3.15: Instrução de retorno de sub-rotina RTS.

ESTADO	OPERAÇÕES
RTS1	R6← R6+2
RTS2	R7←R_X R_X←RDM

Tabela 3.16: Interrupção do teclado.

ESTADO	OPERAÇÕES
INT1	REM1←65498 REM1←65499 RT←128&T2 *RDM←RT*

3.3.1 Simulação

A simulação da figura 3.5 mostra todo o fluxo de execução de uma instrução de soma (ADD) de um operando imediato (#5000) a um dos endereços de memória (65500) correspondentes ao visor do César. A instrução ocupa 6 bytes na memória e leva um total de 13 pulsos de clock para ser executada. Além disto, como a instrução está localizada na posição zero da memória, sendo a primeira a ser executada, a figura também mostra o estado INICIO, que zera os registradores. É importante esclarecer que, neste modelo de codificação, com um só processo, as operações acontecem efetivamente na transição do estado atual, onde a operação está codificada, para o próximo estado. Por exemplo, a transferência de R7 para REM1, codificada no estado BUSCA1 ocorre, na verdade, na borda de clock que realiza a transição do estado BUSCA1 para o estado BUSCA2. A seguir são descritas as operações e transferências de dados que ocorrem em cada um dos estados necessários para a execução desta instrução.

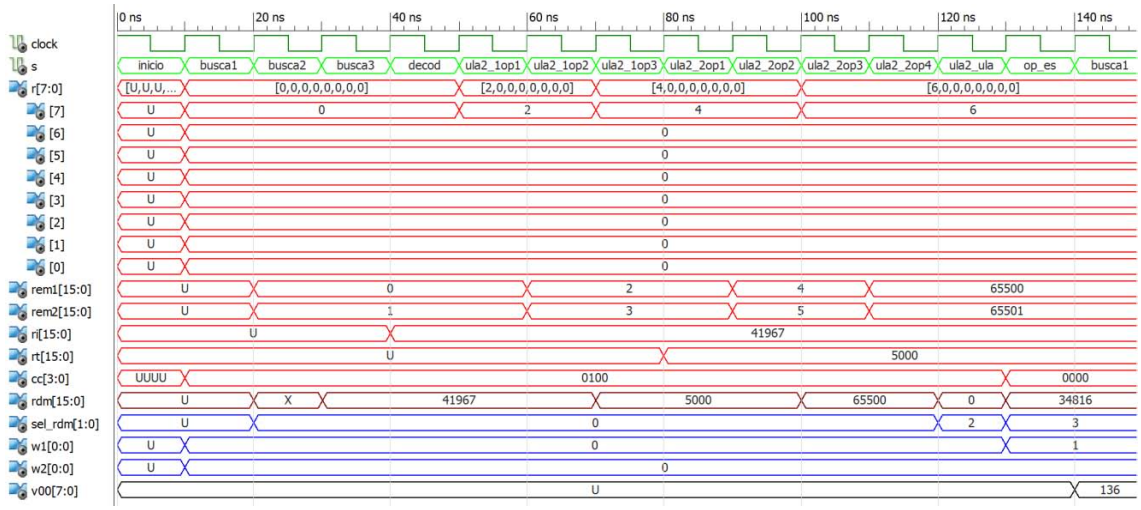


Figura 3.5: Simulação de execução de instrução (Versão Comportamental)

- BUSCA1 – neste estado o conteúdo do PC (R7) é transferido para o registrador de endereços da memória REM1. O registrador REM2 recebe o mesmo valor, incrementado de uma unidade, ou seja, R7+1. O sinal SEL_RDM recebe o valor 0, indicando que a saída do multiplexador MUX_RDM, denominada RDM recebe o conteúdo que será lido da memória.
- BUSCA2 – neste estado é lido o conteúdo da memória, cujo valor é disponibilizado pelo sinal RDM.
- BUSCA3 – é o estado onde o código da instrução lido da memória é copiado para o registrador de instruções RI. Como já explicado, esta transferência ocorre na transição do estado BUSCA3 para o estado DECOD.
- DECOD – neste estado é feito o incremento de duas unidades no registrador R7, já que o código da instrução possui dois bytes. O próximo estado é escolhido de acordo com o tipo da instrução.
- ULA2_1OP1 – aqui começa a busca do primeiro operando. O modo de endereçamento imediato corresponde ao modo de endereçamento pós-incrementado com o registrador R7. Neste caso o operando está no endereço de memória apontado por R7, já incrementado no estado anterior, ou seja, o operando está no endereço seguinte ao código da instrução. A operação realizada neste estado é a transferência do conteúdo de R7 para o registrador REM1. REM2 recebe R7+1.
- ULA2_1OP2 - neste estado é feita a leitura do operando da memória. O registrador especificado no código da instrução, no caso R7, é incrementado em duas unidades.
- ULA2_1OP3 – o operando lido da memória (#5000), disponível em RDM, é copiado para o registrador temporário RT.
- ULA2_2OP1 – aqui começa a busca do segundo operando. O modo de endereçamento absoluto corresponde ao modo de endereçamento pós-incrementado indireto com o registrador R7. Neste caso o endereço do operando está no endereço de memória apontado por R7. A operação realizada neste estado é a transferência do conteúdo de R7 para o registrador REM1. REM2 recebe R7+1.
- ULA2_2OP2 - neste estado é feita a leitura do endereço do operando, da memória. O registrador especificado no código da instrução, no caso R7, é incrementado em duas unidades.

- ULA2_2OP3 – o endereço do operando lido da memória (65500), disponível no sinal RDM, é copiado para o registrador REM1. REM2 recebe RDM+1.
- ULA2_2OP4 – é feita a leitura do operando da memória. Como o endereço do operando é maior do que 65497, o sinal de seleção SEL_RDM recebe o valor 2 (“10”) indicando que este é um operando de um byte e fazendo com que o sinal RDM, da saída do multiplexador, receba o valor 00000000&MEM(65500), ou seja, oito bits com valor 0 são concatenados ao valor do byte contido no endereço de memória 65500, formando o operando de 16 bits que será usado na soma realizada pela instrução da simulação.
- ULA2_ULA – neste estado é realizada a soma entre o primeiro operando (#5000), armazenado no registrador RT, com o segundo operando (#0), dado pelo sinal RDM. O resultado é armazenado no próprio RT. Também os códigos de condição têm seu valor atualizado neste estado. Como o endereço de destino, onde será escrito o resultado, é um endereço de memória maior do que 65497, o sinal SEL_RDM recebe o valor 3 (“11”), indicando que o sinal RDM recebe o valor RT(7:0)&00000000. Apenas o sinal W1 recebe o valor 1, ou seja, apenas o byte menos significativo do resultado da soma é escrito no endereço de memória 65500.
- OP_ES – se o endereço de destino for maior do que 65497, o registrador RES correspondente ao endereço de memória utilizado recebe o conteúdo do byte menos significativo de RT. No caso desta simulação, o endereço de destino é 65500, que corresponde ao registrador RES(2) e à primeira posição do visor do Cesar. Daí a saída V00 receber o valor (#136), que corresponde exatamente ao byte menos significativo do resultado da soma realizada pela instrução.

3.3.2 Exemplo de Código

A figura 3.6 mostra um trecho do código VHDL correspondente ao estado ULA2_1OP1. As transferências de dados são codificadas explicitamente. MEF refere-se ao modo de endereçamento do operando fonte. Também é usada a função CONV_INTEGER (RF) que converte para inteiro o número correspondente ao registrador (R0 a R7) que será utilizado. Isto é necessário porque os registradores foram implementados através de uma estrutura de array, simplificando o código.

```
WHEN ULA2_1OP1 =>
   CASE MEF IS
      WHEN REG =>
         RT<=R(CONV_INTEGER (RF));
         S<=ULA2_2OP1;
      WHEN RPI|IND|PII =>
         REM1<=R(CONV_INTEGER (RF));
         REM2<=R(CONV_INTEGER (RF))+1;
         S<=ULA2_1OP2;
      WHEN RPD|PDI =>
         R(CONV_INTEGER(RF))<=R(CONV_INTEGER (RF))-2;
         S<=ULA2_1OP2;
      WHEN INX|IXI =>
         REM1<=R(7);
         REM2<=R(7)+1;
         S<=ULA2_1OP2;
      WHEN OTHERS =>
         NULL;
   END CASE;
```

Figura 3.6: Exemplo de código (Versão Comportamental)

3.4 Implementação Estrutural (Cesar 2)

As figuras 3.7 e 3.8 mostram o circuito da versão estrutural do Cesar. Esta versão é dita estrutural na medida em que a entidade principal, CESAR2, apenas conecta as entidades CONTROLE e DATAPATH, esta última correspondendo também apenas à interconexão dos outros elementos do circuito. Neste caso, o sistema é visto como dividido em um subsistema de dados, o datapath, e um subsistema de controle. (ERCEGOVAC, 2002).

Este circuito é composto basicamente por um conjunto de registradores, multiplexadores, uma memória de 64 KB, uma ULA e diversos módulos que realizam funções importantes do processador. Enquanto na versão comportamental as operações realizadas em cada etapa são codificadas explicitamente, nesta versão estrutural todas as operações são efetuadas a partir dos sinais gerados pelo módulo de controle.

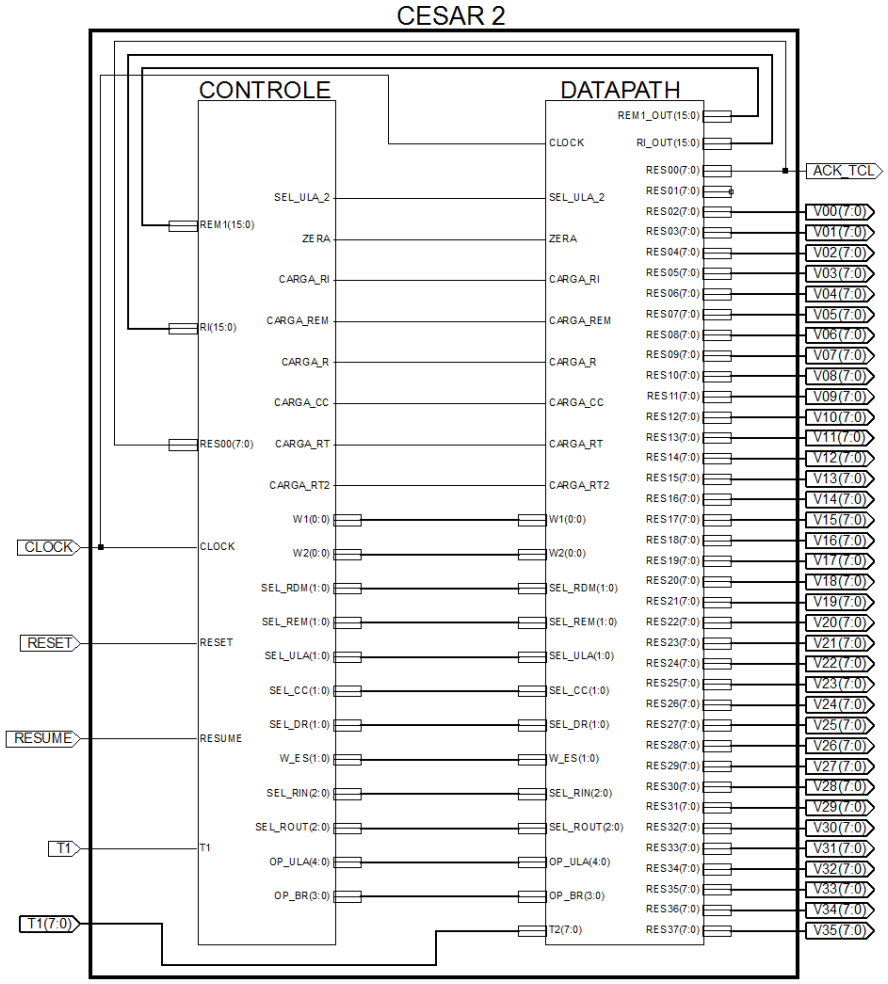


Figura 3.7: Versão estrutural do processador Cesar

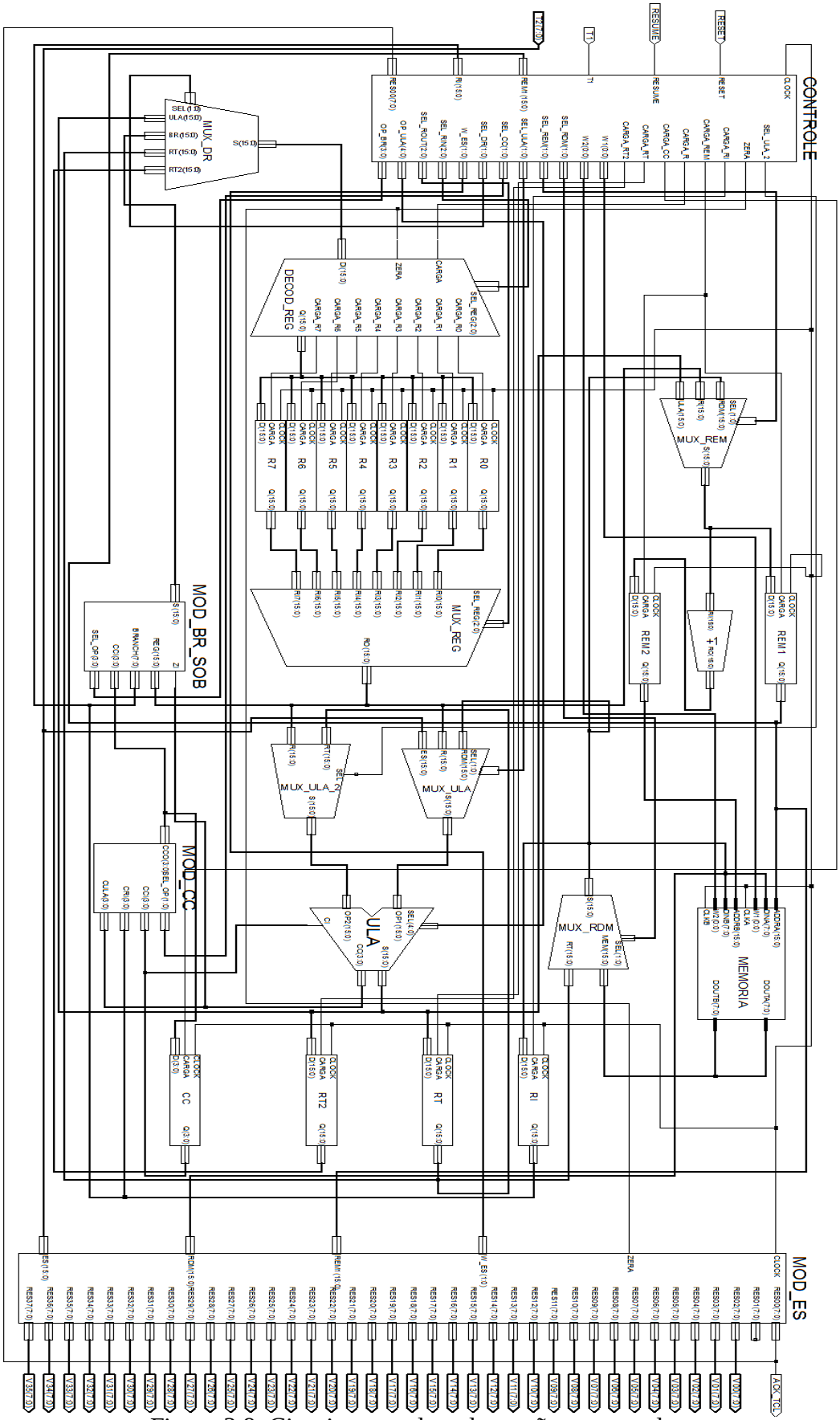


Figura 3.8: Circuito completo da versão estrutural

3.4.1 Componentes do Circuito

A seguir são descritos os componentes do circuito que implementa a versão estrutural do processador Cesar.

- R0 a R7 são os 8 registradores do Cesar.
- RT e RT2 são registradores temporários. RT armazena o valor do primeiro operando nas instruções de dois operandos e também o resultado da operação realizada pela ULA, quando este deve ser escrito na memória, ou seja, o modo de endereçamento utilizado não é o modo registrador. Já o registrador RT2 é utilizado apenas para a instrução JSR.
- REM1 e REM2 são os registradores de endereço da memória. REM2 recebe o valor de REM1 incrementado de uma unidade. Desta forma são acessados dois bytes em cada operação de leitura ou escrita da memória.
- CC guarda os códigos de condição N, Z, C e V do processador.
- RI guarda o código da instrução. No caso de instruções de um byte, os valores do byte menos significativo deste registrador podem ser ignorados, pois faz parte da instrução seguinte.
- MUX_DR seleciona qual valor será carregado em um dos registradores a partir do sinal SEL_DR. Este valor poder vir da ULA, do módulo BR_SOB ou de um dos registradores temporários RT e RT2.
- MUX_ULA e MUX_ULA_2 selecionam os operandos para a ULA a partir dos sinais SEL_ULA e SEL_ULA_2, respectivamente. O primeiro seleciona o valor do operando 1, que pode vir de um registrador, da memória, ou de uma interrupção do teclado. Já o segundo seleciona o valor do operando 2, que ou vem do registrador temporário RT, o que ocorre na maioria dos casos, ou de um dos oito registradores do processador, o que só ocorre nas somas realizadas para o cálculo do endereço nos modos de endereçamento indexado.
- MUX_REM seleciona o valor que será carregado no registrador de endereços da memória, a partir do sinal SEL_REM. Este valor pode vir da memória, de um dos registradores R0 a R7, ou da ULA, o que também só ocorre nos modos de endereçamento indexado. No caso de uma interrupção do teclado, é selecionado o valor 65498.
- MUX_REG seleciona o conteúdo de um dos oito registradores do Cesar a partir do sinal SEL_ROUT.
- MUX_RDM funciona da mesma forma que na versão comportamental, realizando a formatação adequada do dado a ser lido ou escrito na memória, conforme o endereço de memória acessado.
- O decodificador DECOD_REG carrega em um dos registradores, especificado pelo sinal SEL_RIN, um dado proveniente do multiplexador MUX_DR, quando o sinal CARGA_R é ativado. Já quando o sinal ZERA é ativado, todos os registradores tem seu valor zerado.
- O módulo CC calcula o valor a ser carregado no registrador dos códigos de condição, a partir do sinal SEL_CC. Este valor é escolhido conforme a instrução executada. Nas instruções de um ou dois operandos o valor dos códigos de condição é calculado pela ULA. Já nas instruções SCC e CCC os códigos de condição são atualizados a partir de uma operação lógica feita sobre o valor atual carregado no registrador CC com os bits 11 a 8 do registrador de instrução RI.
- A ULA realiza a maioria das operações do processador. Além de todas as instruções de um e dois operandos, a ULA também executa as operações de incremento e decremento de duas unidades, necessárias aos modos de

endereçamento pós-incrementado e pré-decrementado, bem como o incremento automático do registrador R7, nas instruções de dois ou mais bytes (estas operações de incremento e decremento de registradores não afetam os códigos de condição). A ULA possui quatro entradas: um seletor da operação a ser realizada, dois operandos de 16 bits, sendo que as instruções de um operando são realizadas sobre o operando 1, e o valor atual do bit de CARRY, utilizado por algumas instruções. Na implementação da ULA é utilizada uma variável denominada R17B que, assim como na versão comportamental, possui dezessete bits. Este bit a mais, armazena os valores do Carry ou do Borrow nas operações que envolvem uma soma ou uma subtração, respectivamente.

- O módulo BR_SOB realiza a soma e a subtração necessárias às instruções de desvio condicional e à instrução SOB, respectivamente. Embora as operações realizadas por este módulo até pudessem ser realizadas pela ULA, o circuito gerado pela ferramenta de síntese utilizava uma maior número de LUTs, motivo pelo qual se optou pela implementação das instruções de desvio condicional e SOB em um circuito separado. Este módulo recebe o valor atualizado do registrador R7, um deslocamento, dado pelo segundo byte da instrução, cujo valor é armazenado nos bits menos significativos do registrador de instrução RI, um seletor, que assim como na ULA, fornece o código da operação a ser realizada, e o código de condição Z fornecido pela ULA. Este código de condição é necessário para a instrução SOB, que executa ou não um desvio, dependendo do resultado de um decremento de uma unidade do registrador especificado pela instrução. Como esta operação de decremento é realizada pela ULA e a instrução SOB não atualiza os valores dos códigos de condição armazenados no registrador CC, é utilizado o valor de Z calculado no momento desta operação.
- O módulo ES faz a atualização dos valores dos registradores de entrada e saída, que espelham o conteúdo dos últimos 38 bytes da memória do CESAR, conforme o endereço de memória fornecido pela entrada REM1. O dado a ser escrito no registrador é selecionado a partir da entrada W_ES e pode tanto vir da entrada RDM, no caso de uma instrução que acesse um dos endereços de entrada e saída, como da entrada ES, no caso de uma interrupção do teclado.
- O módulo Controle é responsável pela geração dos sinais que controlam todas as transferências entre registradores necessárias à implementação do conjunto de instruções do processador e, assim como na versão comportamental, é implementado através de uma máquina de estados. Enquanto na versão comportamental, apenas um processo controla a lógica do próximo estado e as transferências entre registradores, na versão estrutural há dois processos no módulo de controle. O primeiro controla a lógica de transição dos estados, enquanto o segundo é responsável pela geração dos sinais de controle. No total, o módulo de controle emite vinte sinais que controlam as transferências de dados entre os registradores Os sinais de controle utilizados são os seguintes:
    - SEL_CC, SEL_DR, SEL_RDM, SEL_REM, SEL_RIN, SEL_ROUT, SEL_ULA e SEL_ULA2 selecionam as saídas adequadas dos multiplexadores do circuito em cada estado.
    - OP_ULA e OP_BR selecionam respectivamente as operações realizadas pela ULA e pelo módulo BR_SOB.
    - W1 e W2 são os sinais que determinam uma operação de escrita na memória. Em uma operação de escrita normal, ambos os sinais recebem

o valor lógico 1. No caso de uma escrita em um dos endereços de entrada e saída, apenas W1 é ativado.

- O sinal ZERA é responsável por zerar o valor dos registradores no estado inicial.
- O valor do sinal W_ES determina a escrita ou não em um dos registradores de entrada e saída, quando uma instrução faz referência aos últimos endereços de memória, ou quando ocorre uma interrupção do teclado.
- Os demais sinais referem-se à carga dos registradores.

Como se pode perceber pelas descrições do decodificador e do multiplexador ligados aos registradores R0 a R7, apenas um destes registradores pode ter seu conteúdo acessado ou alterado a cada pulso de clock. Isto diminui o número de sinais necessários no módulo de controle, pois só há um sinal de carga para os oito registradores do computador, e também diminui consideravelmente o número de ligações necessárias entre os diferentes módulos do circuito. A desvantagem, porém, reside na necessidade de um maior número de estados necessários à execução da instrução JSR, que exige quatro estados a mais na versão estrutural do que os necessários para a versão comportamental. Como as operações de incremento e decremento de duas unidades, bem como as somas necessárias ao modo de endereçamento indexado, são feitas pela ULA, uma operação de salvar no registrador temporário RT o conteúdo de um registrador, não pode ser feita ao mesmo tempo em que se realiza o incremento de duas unidades deste registrador. Isto ocorre, por exemplo, no modo de endereçamento pós-incrementado. Neste modo de endereçamento, o endereço da sub-rotina está em um registrador que, depois de ter o seu conteúdo copiado para o registrador temporário RT, deve ter seu valor incrementado em duas unidades. Enquanto na versão comportamental estas duas operações são feitas no estado JSR1, ou seja, em um único pulso de clock, na versão estrutural são necessários dois pulsos de clock, e a operação de incremento do registrador terá de ser feita no próximo estado, no caso JSR2.

Tabela 3.17: Instrução JSR nos modos REG, RPI, RPD e INX

ESTADO	REG	RPI	RPD	INX
JSR1		RT2←R	R←R-2	REM1←R7 REM2←R7+1
JSR2		R←R+2	RT2←R	R7←R7+2 *RDM←MEM*
JSR3		R6←R6-2	R6←R6-2	RT2←R+RDM
JSR4		REM1←R6 REM2←R6+1	REM1←R6 REM2←R6+1	R6←R6-2
JSR5		RT←R_X	RT←R_X	REM1←R6 REM2←R6+1
JSR6		*RDM←RT* R_X←R7	*RDM←RT* R_X←R7	RT←R_X
JSR7		R7←RT2	R7←RT2	*RDM←RT* R_X←R7
JSR8				R7←RT2

Tabela 3.18: Instrução JSR nos modos IND, PII, PDI e IXI

ESTADO	IND	PII	PDI	IXI
JSR1	RT2←R	REM1←R REM2←R+1 R←R+2	R←R-2	REM1←R7 REM2←R7+1
JSR2	R6←R6-2	*RDM←MEM*	REM1←R REM2←R+1	R7←R7+2 *RDM←MEM*
JSR3	REM1←R6 REM2←R6+1	RT2←RDM	*RDM←MEM*	REM1←R+RDM REM2←R+RDM+1
JSR4	RT←R_X	R6←R6-2	RT2←RDM	RDM←MEM
JSR5	*RDM←RT* R_X←R7	REM1←R6 REM2←R6+1	R6←R6-2	RT2←RDM
JSR6	R7←RT2	RT←R_X	REM1←R6 REM2←R6+1	R6←R6-2
JSR7		*RDM←RT* R_X←R7	RT←R_X	REM1←R6 REM2←R6+1
JSR8		R7←RT2	*RDM←RT* R_X←R7	RT←R_X
JSR9			R7←RT2	*RDM←RT* R_X←R7
JSR10				R7←RT2

3.4.2 Simulação

A simulação da figura 3.9 mostra todo o fluxo de execução de uma instrução de soma (ADD) de um operando imediato (#5000) a um dos endereços de memória (65500) correspondentes ao visor do César. Para efeito de comparação, esta é a mesma instrução utilizada na simulação da versão comportamental. A instrução leva os mesmos 13 pulsos de clock para ser executada.

Aqui também, como a instrução está localizada na posição zero da memória, a figura também mostra o estado INICIO, que zera os registradores ativando o sinal ZERA. Os sinais cujos valores são irrelevantes em determinado estado estão indicados pelo valor X, para que seja mais fácil verificar quais são os sinais necessários para que a operação ou transferência de dados seja efetuada. Na prática, quando um sinal é desnecessário, ele assume o valor 0.

Na descrição do módulo de controle, foi dito que o mesmo utiliza dois processos. O motivo para implementar desta forma é a temporização. Por exemplo, se na versão comportamental, com apenas um processo, uma transferência entre registradores descrita em um determinado estado, ocorre na verdade no momento da transição deste estado para o próximo, na versão estrutural, com dois processos, o sinal de controle recebe o valor codificado no estado já na transição que leva a este estado. Isto pode ser visto na comparação das simulações das duas versões.

Como mostrado na versão comportamental, a transferência do conteúdo de R7 para o registrador de endereços da memória REM1, codificada no estado BUSCA1, ocorre na transição do estado BUSCA1 para o estado BUSCA2. Na versão estrutural, por sua

vez, esta mesma operação é efetuada através do sinal de CARGA_REM e dos sinais de seleção adequados nos multiplexadores de saída dos registradores e da entrada do registrador REM1. Estes sinais assumem os valores corretos já no estado BUSCA1, ou seja, na transição do estado anterior para o estado atual. Desta forma, REM1 receberá o valor de R7 na transição do estado BUSCA1 para o estado BUSCA2, assim como na versão comportamental. Se os sinais de controle só assumissem os valores corretos na transição para o próximo estado, seria necessária a adição de mais um estado na fase de busca da instrução.

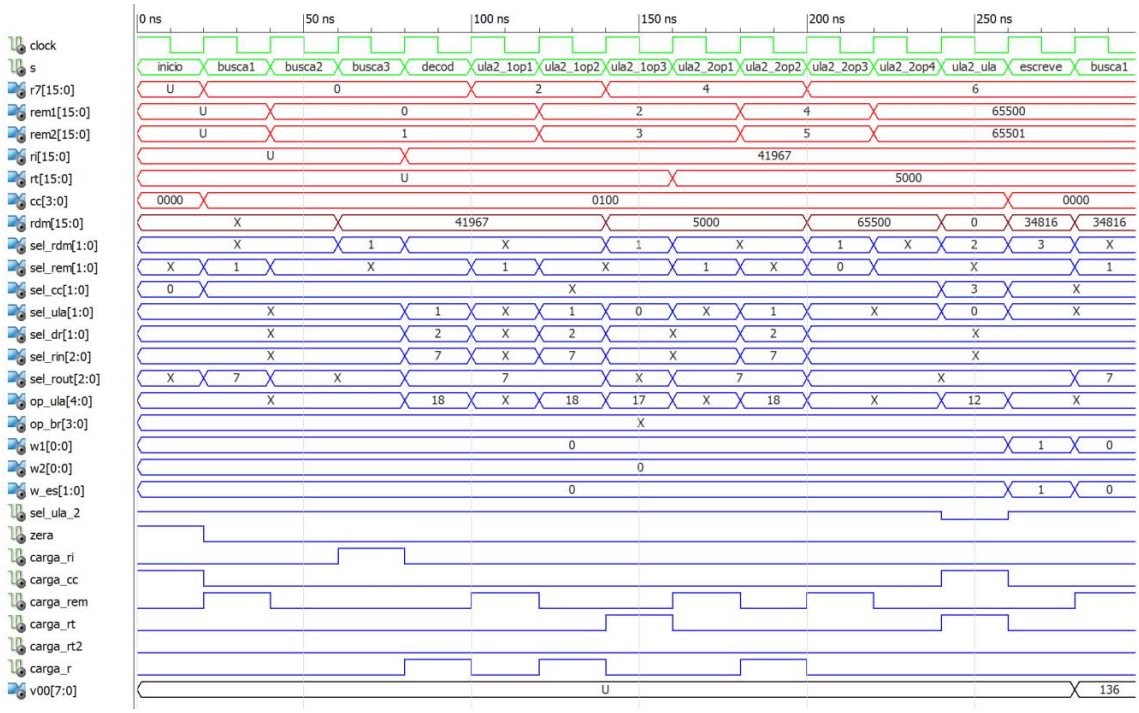


Figura 3.9: Simulação de execução de instrução (Versão Estrutural)

A seguir são descritas as operações e os sinais ativados em cada um dos estados necessários para a execução desta instrução.

- BUSCA1 – neste estado o conteúdo do PC (R7), é transferido para o registrador de endereços da memória REM1. O registrador REM2 recebe o mesmo valor incrementado de uma unidade, ou seja, R7+1. Esta operação exige que a saída do multiplexador MUX_REG receba o conteúdo de R7 e que a saída do multiplexador MUX_REM selecione a entrada vinda do MUX_REG. Isto é feito através dos valores adequados para os sinais SEL_ROUT (7) e SEL_REM (1), além da ativação do sinal CARGA_REM.
- BUSCA2 – neste estado é lido o conteúdo da memória. Nenhum sinal é necessário neste estado.
- BUSCA3 – é o estado onde o código da instrução lido da memória é copiado para o registrador de instruções RI. Para isto, SEL_RDM (0) seleciona a entrada vinda da memória e o sinal CARGA_RI é ativado.
- DECOD – neste estado é feito o incremento de duas unidades no registrador R7, já que o código da instrução possui dois bytes. Esta operação necessita de vários sinais de controle. SEL_ROUT (7) seleciona o registrador R7, SEL_ULA (1) seleciona a entrada vinda do registrador, OP_ULA (18) seleciona a operação incremento de dois, SEL_DR (2) seleciona a entrada vinda da ULA, SEL_RIN (7) seleciona o registrador R7 e CARGA_R carrega o novo valor em R7. Isto

significa que o conteúdo lido do registrador R7 passa por dois multiplexadores (MUX_REG e MUX_ULA), é incrementado pela ULA, passa por mais um multiplexador (MUX_DR) e, finalmente, pelo decodificador (DECOD_REG) para ser, então, atualizado através do sinal CARGA_R. O próximo estado é escolhido de acordo com o tipo da instrução.

- ULA2_1OP1 – aqui começa a busca do primeiro operando. O modo de endereçamento imediato corresponde ao modo de endereçamento pós-incrementado com o registrador R7. Neste caso o operando está no endereço de memória apontado por R7, já incrementado no estado anterior. A operação realizada neste estado é a transferência do conteúdo de R7 para o registrador REM1. Esta é a mesma operação feita no estado BUSCA1, sendo realizada exatamente pelos mesmos sinais de controle.
- ULA2_1OP2 - neste estado é feita a leitura do operando da memória. O registrador especificado no código da instrução, no caso R7, é incrementado em duas unidades. A mesma sequência de sinais gerados no estado de decodificação é gerada também neste estado.
- ULA2_1OP3 – o sinal SEL_RDM recebe o valor 1 fazendo com que o operando lido da memória (#5000) seja disponibilizado no sinal RDM, correspondente à saída do multiplexador MUX_RDM. Para que seja salvo no registrador RT, são necessários os sinais SEL_ULA (0), que seleciona a entrada RDM, OP_ULA (17), que seleciona a operação MOV na ULA e, finalmente, CARGA_RT.
- ULA2_2OP1 – aqui começa a busca do segundo operando. O modo de endereçamento absoluto corresponde ao modo de endereçamento pós-incrementado indireto com o registrador R7. Neste caso o endereço do operando está no endereço de memória apontado por R7. A operação realizada neste estado é a transferência do conteúdo de R7 para o registrador REM1. Esta é a mesma operação feita nos estados BUSCA1 e ULA2_1OP1, sendo realizada exatamente pelos mesmos sinais de controle.
- ULA2_2OP2 - neste estado é feita a leitura do endereço do operando, da memória. O registrador especificado no código da instrução, no caso R7, é incrementado em duas unidades. A mesma sequência de sinais gerada nos estados DECOD e ULA2_1OP2 é gerada também neste estado.
- ULA2_2OP3 – o endereço do operando lido da memória (65500), disponível no sinal RDM, é copiado para o registrador REM1. REM2 recebe RDM+1. Esta operação é feita com os sinais SEL_RDM (1), SEL_REM (0) e com a ativação do sinal CARGA_REM.
- ULA2_2OP4 – é feita a leitura do operando da memória. Nenhum sinal é necessário neste estado.
- ULA2_ULA – neste estado é realizada a soma entre o primeiro operando (#5000), armazenado no registrador RT, com o segundo operando (#0). Como o endereço do operando é maior do que 65497, o sinal de seleção SEL_RDM recebe o valor 2, indicando que este é um operando de um byte e fazendo com que o sinal RDM, da saída do multiplexador, receba o valor 00000000&MEM(65500). A soma é realizada através dos sinais SEL_ULA (0), SEL_ULA_2 (0) e OP_ULA (12), que seleciona a operação ADD. O resultado é armazenado no registrador RT, através do sinal CARGA_RT. Também os códigos de condição têm seu valor atualizado neste estado. Isto é feito com os sinais SEL_CC (3), que seleciona a entrada vinda da ULA nos módulos MOD_CC e CARGA_RT.

- ESCREVE – Como o endereço de destino, onde será escrito o resultado, é um endereço de memória maior do que 65497, o sinal SEL_RDM recebe o valor 3 (“11”), indicando que o sinal RDM recebe o valor RT(7:0)&00000000. Apenas o sinal W1 recebe o valor 1, ou seja, apenas o byte menos significativo do resultado da soma é escrito no endereço de memória 65500. O sinal W_ES igual a 1 indica que o registrador RES02, correspondente ao endereço 65500, e, por conseqüência, V00 recebem o conteúdo do byte menos significativo de RT, ou (#136).

3.4.3 Exemplo de Código

A descrição da máquina de estados em dois processos diferentes também determina que em cada estado deva ser obrigatoriamente atribuído um valor a todos os sinais de controle. Se em algum estado um destes sinais de controle não tiver um valor especificado no código, a ferramenta de síntese irá emitir um alerta de que foi utilizado um latch para este sinal, ao invés de um flip-flop.

As figuras 3.10 e 3.11 mostram trechos do código VHDL correspondentes à lógica de próximo estado e aos sinais de controle, respectivamente, para o estado ULA2_1OP1 da versão estrutural.

```
WHEN ULA2_1OP1 =>
   CASE MEF IS
      WHEN REG =>
         S<=ULA2_2OP1;
      WHEN OTHERS =>
         S<=ULA2_1OP2;
   END CASE;
```

Figura 3.10: Exemplo de código da lógica do próximo estado (Versão Estrutural)

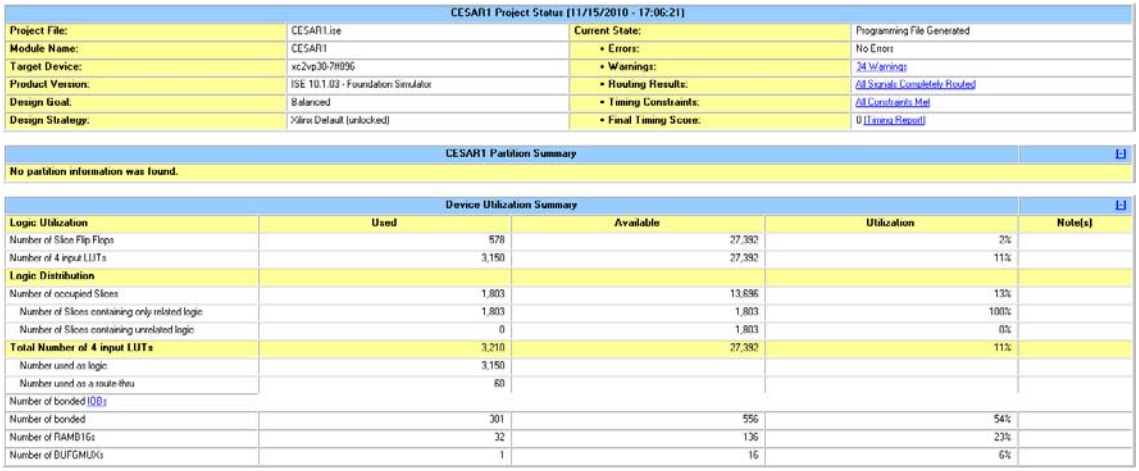
```
WHEN ULA2_1OP1 =>
   CASE MEF IS
      WHEN REG =>
         SEL_REM<=DONT_CARE_2B;
         SEL_ULA<=ER;
         SEL_DR<=DONT_CARE_2B;
         SEL_RIN<=DONT_CARE_3B;
         SEL_ROUT<=RF;
         OP_ULA<=ULA_MOV;
         CARGA_REM<='0';
         CARGA_R<='0';
         CARGA_RT<='1';
      WHEN RPI|IND|PII =>
         SEL_REM<=ER;
         SEL_ULA<=DONT_CARE_2B;
         SEL_DR<=DONT_CARE_2B;
         SEL_RIN<=DONT_CARE_3B;
         SEL_ROUT<=RF;
         OP_ULA<=DONT_CARE_5B;
         CARGA_REM<='1';
         CARGA_R<='0';
         CARGA_RT<='0';
      WHEN RPD|PDI =>
         SEL_REM<=DONT_CARE_2B;
         SEL_ULA<=ER;
```

```
            SEL_DR<=EULA;
            SEL_RIN<=RF;
            SEL_ROUT<=RF;
            OP_ULA<=ULA_DEC2;
            CARGA_REM<='0';
            CARGA_R<='1';
            CARGA_RT<='0';
        WHEN INX|IXI =>
            SEL_REM<=ER;
            SEL_ULA<=DONT_CARE_2B;
            SEL_DR<=DONT_CARE_2B;
            SEL_RIN<=DONT_CARE_3B;
            SEL_ROUT<=R7;
            OP_ULA<=DONT_CARE_5B;
            CARGA_REM<='1';
            CARGA_R<='0';
            CARGA_RT<='0';
        WHEN OTHERS =>
            SEL_REM<=DONT_CARE_2B;
            SEL_ULA<=DONT_CARE_2B;
            SEL_DR<=DONT_CARE_2B;
            SEL_RIN<=DONT_CARE_3B;
            SEL_ROUT<=DONT_CARE_3B;
            OP_ULA<=DONT_CARE_5B;
            CARGA_REM<='0';
            CARGA_R<='0';
            CARGA_RT<='0';
    END CASE;
    SEL_RDM<=DONT_CARE_2B;
    SEL_ULA_2<=DONT_CARE_1B;
    SEL_CC<=DONT_CARE_2B;
    W1<="0";
    W2<="0";
    OP_BR<=DONT_CARE_4B;
    ZERA<='0';
    CARGA_RI<='0';
    CARGA_CC<='0';
    CARGA_RT2<='0';
    W_ES<="00";
```

Figura 3.11: Exemplo de código dos sinais de controle (Versão Estrutural)

3.5 Comparação das Versões

Embora não seja este o objetivo do trabalho, é inevitável fazer uma comparação entre os dois métodos de implementação. As figuras 3.12 e 3.13 mostram os resultados apresentados pela ferramenta de síntese para as duas implementações.



CESAR1 Project Status (11/15/2010 - 17:06:21)

Project File:	CESAR1.ise	Current State:	Programming File Generated
Module Name:	CESAR1	• Errors:	No Errors
Target Device:	xc2vp30-7ff896	• Warnings:	34 Warnings
Product Version:	ISE 10.1.03 - Foundation Simulator	• Routing Results:	All Signals Completely Routed
Design Goal:	Balanced	• Timing Constraints:	All Constraints Met
Design Strategy:	Xilinx Default (unlocked)	• Final Timing Score:	0 (Timing Report)

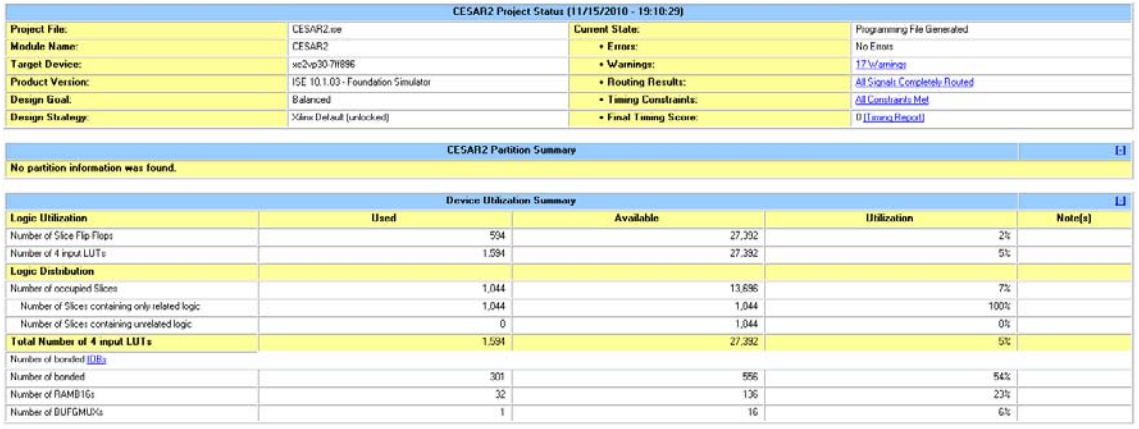
CESAR1 Partition Summary

No partition information was found.

Device Utilization Summary

Logic Utilization	Used	Available	Utilization	Note(s)
Number of Slice Flip Flops	578	27,392	2%	
Number of 4 input LUTs	3,150	27,392	11%	
Logic Distribution				
Number of occupied Slices	1,803	13,696	13%	
Number of Slices containing only related logic	1,803	1,803	100%	
Number of Slices containing unrelated logic	0	1,803	0%	
Total Number of 4 input LUTs	3,210	27,392	11%	
Number used as logic	3,150			
Number used as a route-thru	60			
Number of bonded IOBs				
Number of bonded	301	556	54%	
Number of RAMB16s	32	136	23%	
Number of BUFGMUXs	1	16	6%	

Figura 3.12: Resultado da síntese da versão comportamental



CESAR2 Project Status (11/15/2010 - 19:10:29)

Project File:	CESAR2.xse	Current State:	Programming File Generated
Module Name:	CESAR2	• Errors:	No Errors
Target Device:	xc2vp30-7ff896	• Warnings:	17 Warnings
Product Version:	ISE 10.1.03 - Foundation Simulator	• Routing Results:	All Signals Completely Routed
Design Goal:	Balanced	• Timing Constraints:	All Constraints Met
Design Strategy:	Xilinx Default (unlocked)	• Final Timing Score:	0 (Timing Report)

CESAR2 Partition Summary

No partition information was found.

Device Utilization Summary

Logic Utilization	Used	Available	Utilization	Note(s)
Number of Slice Flip Flops	594	27,392	2%	
Number of 4 input LUTs	1,594	27,392	5%	
Logic Distribution				
Number of occupied Slices	1,044	13,696	7%	
Number of Slices containing only related logic	1,044	1,044	100%	
Number of Slices containing unrelated logic	0	1,044	0%	
Total Number of 4 input LUTs	1,594	27,392	5%	
Number of bonded IOBs				
Number of bonded	301	556	54%	
Number of RAMB16s	32	136	23%	
Number of BUFGMUXs	1	16	6%	

Figura 3.13: Resultado da síntese da versão estrutural

Como se pode ver, a versão comportamental, ainda que apresente um código menor e muito mais simples de ser compreendido, ocupa um número de elementos lógicos (LUTs) que representa praticamente o dobro do número utilizado pela versão estrutural. Além disto, é quase impossível de se prever como será o circuito sintetizado pela ferramenta.

Entretanto, além do menor tempo necessário na codificação, este circuito também é capaz de operar em uma freqüência consideravelmente maior, como pode ser visto na tabela 3.19, que mostra o resultado obtido pela ferramenta de síntese para diferentes estratégias de otimização oferecidas por ela. Também o número de estados necessários para a execução de algumas instruções é menor na versão comportamental e, no caso de uma eventual alteração na arquitetura, alterações podem ser feitas com relativa facilidade.

Tabela 3.19: Comparação das versões comportamental e estrutural

Versão	**Estratégia**	**LUTs**	**Frequência Máxima em MHz (aproximada)**
Comportamental	Área	3148	91
Estrutural	Área	1300	55
Comportamental	Velocidade	3210	100
Estrutural	Velocidade	1594	67

Na tabela 3.19 vale considerar que, independentemente da estratégia de otimização utilizada, a ferramenta de síntese seleciona, por default, a codificação one-hot para a implementação da máquina de estados. Esta codificação, que utiliza um flip-flop por estado, simplifica a lógica do circuito, além de ser a mais rápida (PEDRONI, 2004).

As vantagens e desvantagens da versão estrutural não podem, porém, ser consideradas como inerentes a este método de implementação, pois são diretamente dependentes do circuito projetado para o datapath. No caso deste trabalho, o circuito foi projetado já com o objetivo de que fosse o mais simples possível, em termos de estruturas necessárias para o seu funcionamento. Isto foi feito na tentativa de se verificar qual seria um valor mínimo de elementos lógicos necessários para a implementação deste processador em uma FPGA, até para comparar com outros processadores *soft-core* existentes, como os citados na figura 3.14.

De qualquer forma, a implementação estrutural, por exigir o projeto de um circuito com antecedência, embora possibilite maior controle sobre o circuito sintetizado pela ferramenta, torna qualquer alteração posterior muito mais onerosa, além de demandar um tempo consideravelmente maior para ser realizada.

CPU core	Architecture	Bits	License	Pipeline depth	Cycles per instruction[1]	MMU[2]	MUL[3]	FPU[4]	Area (LEs[5])	Comments
S1 Core	SPARC-v9	64	Open-source (GPL)	6	1	+	+	+	37000 - 60000	Single-core version of UltraSPARC T1
LEON3	SPARC-v8	32	Open-source (GPL)	7	1	+	+	+	3500	
LEON2	SPARC-v8	32	Open-source (LGPL)	5	1	+	+	ext	5000	Unmaintained in favor of LEON3
OpenRISC 1200	OpenRISC 1000	32	Open-source (LGPL)	5	1	+	+	-	6000	
MicroBlaze	MicroBlaze	32	Proprietary	3, 5	1	opt	opt	opt	1324	Limited to Xilinx devices
aeMB	MicroBlaze	32	Open-source (LGPL)	3	1	-	opt	-	2536	Open-source clones of MicroBlaze
OpenFire	MicroBlaze	32	Open-source (MIT)	3	1	-	opt	-	1928	
Nios II/f	Nios II	32	Proprietary	6	1	+	+	opt	1800	Limited to Altera devices
Nios II/s	Nios II	32	Proprietary	5	1	-	+	opt	1170	
Nios II/e	Nios II	32	Proprietary	no	6	-	-	opt	390	
LatticeMico32	LatticeMico32	32	Open-source	6	1	-	opt	-	1984	Not limited to Lattice devices (can be used elsewhere)
Cortex-M1	ARMv6	32	Proprietary	3	1	-	+	-	2600	
DSPuva16	DSPuva16	16	Open-source	no	4	-	+	-	510	
PicoBlaze	PicoBlaze	8	Proprietary, zero-cost	no	2	-	-	-	192	Limited to Xilinx devices
PacoBlaze	PicoBlaze	8	Open-source (BSD)	no	2	-	-	-	204	An open-source clone of PicoBlaze
LatticeMico8	LatticeMico8	8	Open-source	no	2	-	-	-	200	Not limited to Lattice devices (can be used elsewhere)

Figura 3.14: Exemplos de processadores *soft-core* (http://www.1-core.com/library/digital/soft-cpu-cores/)

4 TESTE DO PROCESSADOR (VGA + TECLADO)

Embora tenham sido mostradas no capítulo anterior simulações das duas versões do processador, para realmente testar o seu funcionamento no FPGA, foi necessária a implementação de componentes adicionais. As seções a seguir explicam como foram concebidos os circuitos de vídeo, que utiliza a porta VGA disponível na placa para simular o visor do Cesar, e de teclado, que permite que sejam testados programas com entrada de dados através de um teclado, ligado a uma das portas PS/2 da placa.

4.1 Módulo VGA

O módulo VGA mostrado na figura 4.1 recebe os conteúdos das saídas correspondentes ao visor do Cesar e gera os sinais necessários para a exibição destes valores em um monitor de vídeo. As saídas de dados de cor RGB (red, green e blue) possuem oito bits cada e passam por um conversor digital/analógico ligado à porta VGA da placa. Além disto, são utilizados o sinal de BLANK que desabilita o sinal de vídeo, sobrepondo-se às saídas RGB, o sinal PIXEL_CLOCK, que define o número de pixels processados em um segundo, e os sinais de sincronização horizontal e vertical H_SYNC e V_SYNC. Embora apareça no circuito, o sinal SYNC possui um valor constante e não é utilizado.

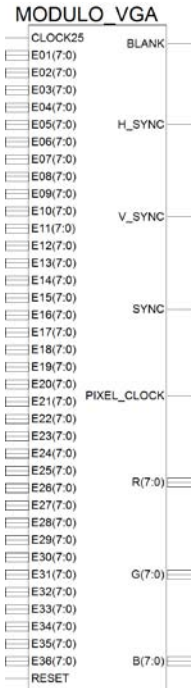
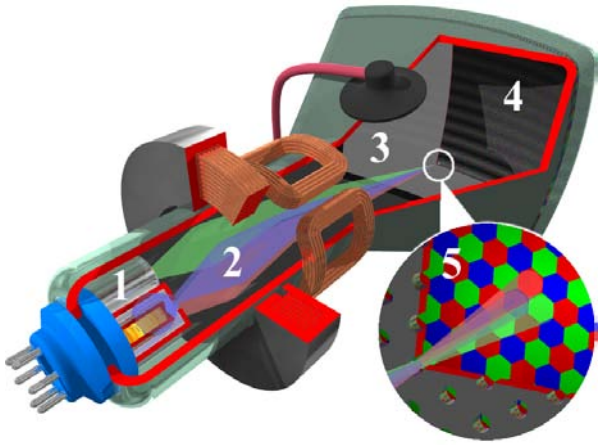


Figura 4.1: Módulo VGA

Para entender o funcionamento destes sinais é necessária uma breve explicação do funcionamento dos monitores CRT. O principal componente de um monitor CRT é o tubo de raios catódicos. Nele um canhão de elétrons, na parte de trás do tubo, gera um ou mais feixes de elétrons que, guiados por um campo magnético, varrem uma tela coberta com uma fina camada de elementos fosforescentes, denominados fósforos. Existem três tipos de fósforos, um para cada uma das cores verde, azul e vermelha. Estes fósforos emitem luz quando atingidos pelo feixe de elétrons, criando os pixels que formam a imagem. A informação contida nos sinais RGB é usada para controlar a força dos feixes de elétrons, produzindo, assim, todas as cores vistas na imagem.



1. Canhões de Elétrons.
2. Feixes de Elétrons
3. Máscara que separa os feixes vermelho, verde e azul.
4. Camada de fósforos para as cores vermelha, verde e azul.
5. Detalhe da camada de fósforos na parte interna da tela.

Figura 4.2: Componentes de um monitor CRT (Wikipedia)

A varredura da tela pelos feixes de elétrons é feita da direita para a esquerda e de cima para baixo. Os sinais de sincronização H_SYNC e V_SYNC representam, respectivamente, os tempos necessários para percorrer uma linha e a tela inteira. A figura 4.3 mostra o esquema de varredura da tela pelo feixe de elétrons. Já as figuras 4.4 e 4.5 mostram os diagramas de tempo dos sinais de sincronização horizontal e vertical. As três figuras consideram uma imagem de 640 X 480 pixels com taxa de atualização de 60 Hz.

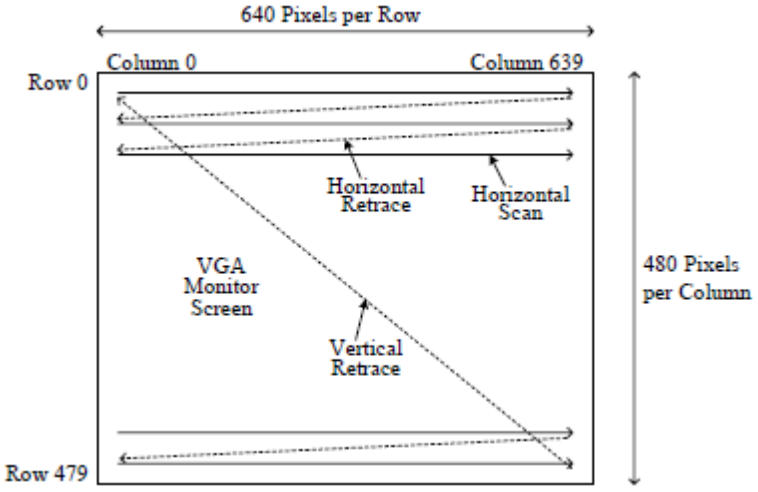


Figura 4.3: Varredura da Tela (HWANG, 2005)

É interessante ressaltar que, embora sejam 640 pixels em cada linha da imagem, um período do sinal H_SYNC contém um total de 800 pixels divididos nas quatro regiões mostradas na figura 4.4. Isto também ocorre com relação ao sinal V_SYNC que possui um período de 525 linhas, embora apenas 480 linhas sejam efetivamente exibidas na tela. A freqüência de clock utilizada para esta resolução de tela, correspondente à saída PIXEL_CLOCK, é de 25 MHz. Seu cálculo vem do valor aproximado da seguinte multiplicação: $800\ \frac{pixels}{linha} \times 525\ linhas\ \times 60\ Hz$. Mais informações sobre os sinais que controlam o VGA podem ser encontrados em (CHU, 2008) e (WILSON, 2007).

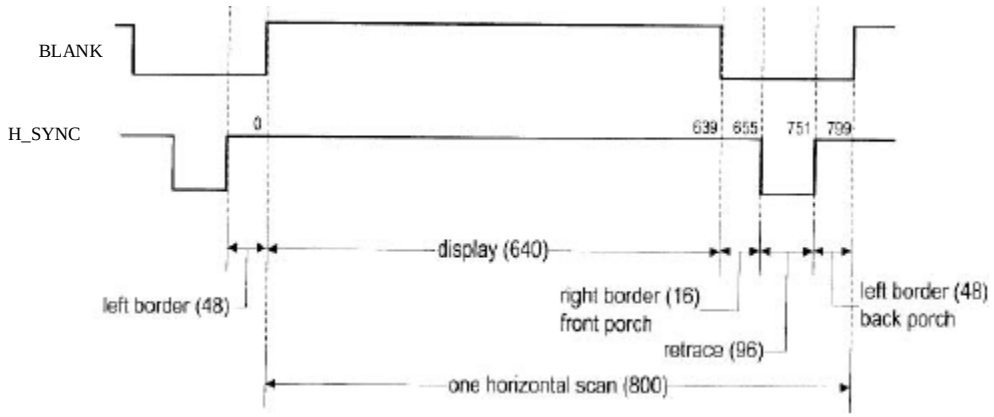


Figura 4.4: Sinal de sincronização horizontal (CHU, 2008)

Na figura 4.4 as seguintes regiões podem ser identificadas:

- Display: é a região que contém os 640 pixels que são exibidos na tela e ajudam a formar a imagem. É a única região onde o sinal de vídeo está habilitado, ou seja, o sinal de BLANK possui nível lógico 1. Nesta região a informação de cor do pixel é dada pelo valor das saídas RGB.
- Retrace: é a região onde o feixe de elétrons retorna à margem esquerda e é marcada por um pulso negativo do sinal de sincronização H_SYNC, ou seja, H_SYNC recebe o valor lógico 0. Neste intervalo, que corresponde ao tempo de varredura de 96 pixels, o sinal de BLANK também recebe o valor lógico 0, o que desabilita o sinal de vídeo, fazendo com que as saídas RGB sejam ignoradas.
- Right Border: representa a borda direita da região de display. Possui 16 pixels e também é denominada Front Porch (porch before retrace). Desabilita o sinal de vídeo.
- Left Border: é a região que forma a borda esquerda da região de display. Possui 48 pixels e também é denominada Back Porch (porch after retrace). Desabilita o sinal de vídeo.

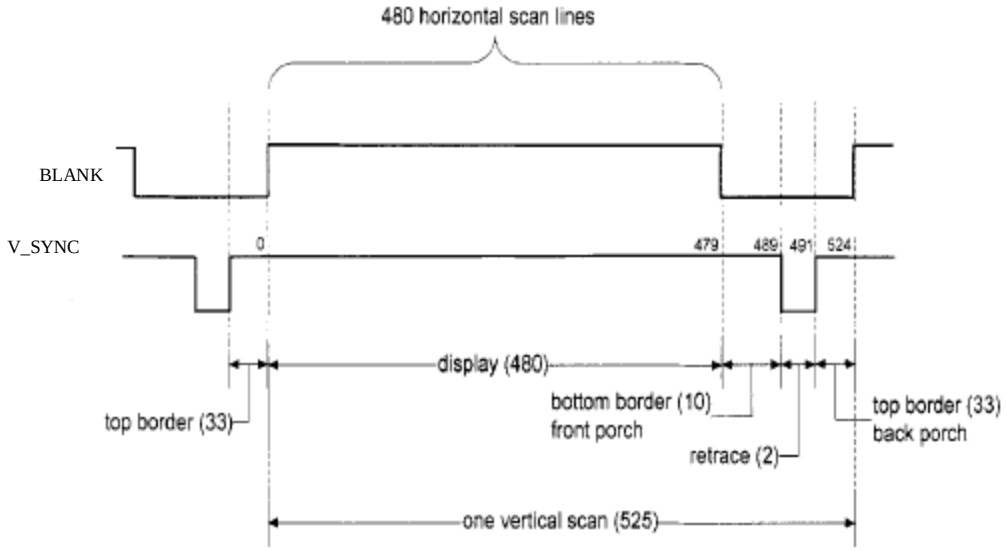


Figura 4.5: Sinal de sincronização vertical (CHU, 2008)

Na figura 4.5 as seguintes regiões podem ser identificadas:

- Display: é a região que contém as 480 linhas que são exibidas na tela e formam a imagem. Nesta região o sinal de vídeo está habilitado, ou seja, BLANK = 1.
- Retrace: é a região onde o feixe de elétrons retorna à posição inicial de varredura da tela, na borda superior esquerda, e é marcado por um pulso negativo do sinal V_SYNC. Neste intervalo, que dura duas linhas, o sinal de vídeo é desabilitado com BLANK = 0.
- Bottom Border: representa a borda inferior da região de display. Este intervalo tem o tamanho de 10 linhas e também é denominado Front Porch. Também desabilita o sinal de vídeo.
- Top Border: região que forma a borda esquerda da região de display. Possui 48 pixels e também é denominada Back Porch. Também desabilita o sinal de vídeo.

A figura 4.6 mostra o circuito completo do módulo VGA. O módulo GERADOR_VGA é responsável pela geração dos sinais de sincronismo H_SYNC e V_SYNC, a partir de dois contadores V_COUNT e H_COUNT que contam, respectivamente, as 525 linhas com 800 pixels cada. Estes dois contadores também são utilizados na geração dos sinais PIXEL_X e PIXEL_Y, conectados às saídas PIXEL_X_OUT e PIXEL_Y_OUT e que, em conjunto, fornecem a coordenada do pixel exibido na tela. Estes dois sinais contam apenas os pixels que realmente fazem parte da imagem, ou seja, as 480 linhas de 640 pixels da região do display.

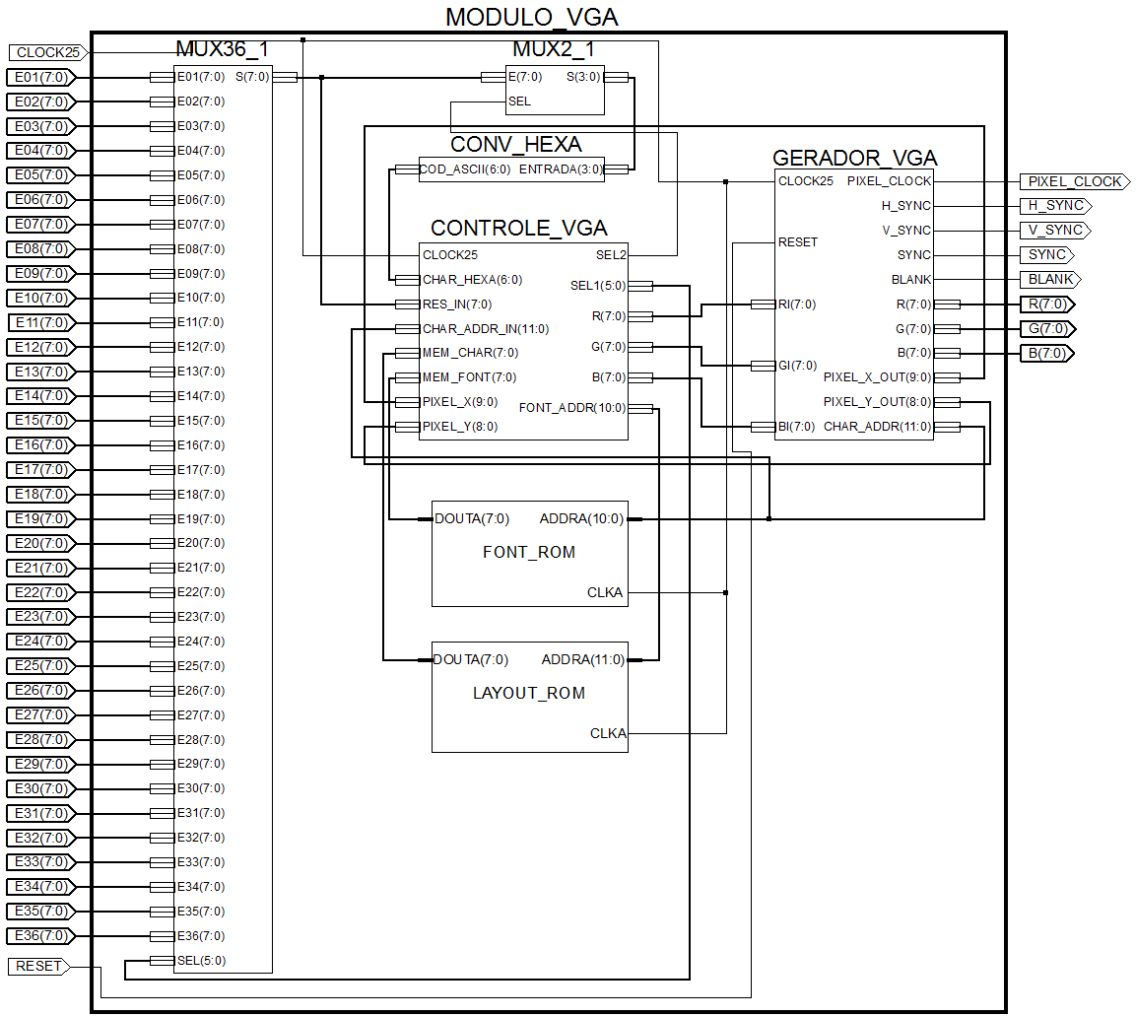


Figura 4.6: Circuito do módulo VGA

Outro sinal importante fornecido por este módulo é o sinal CHAR_ADDR. Este sinal, obtido a partir do contador CONT_CHAR e do sinal PIXEL_Y, fornece a posição do caractere exibido na tela. Um caractere corresponde a um conjunto de bits na forma de um padrão de 0s e 1s que pode ser interpretado como o desenho de um símbolo a ser exibido na tela. Para armazenar os padrões de bits dos 128 símbolos do padrão ASCII é utilizada uma memória apenas de leitura denominada FONT_ROM. Como foi utilizado um formato de caractere de 8x16 (8 pixels de largura por 16 pixels de altura), é necessária uma memória de 2048 bytes, ou 2 KB. A figura 4.7 mostra o padrão de bits para as letras A, B e C.

00000000	00000000	00000000
00000000	00000000	00000000
00010000	11111100	00111100
00111000	01100110	01100110
01101100	01100110	11000010
11000110	01100110	11000000
11000110	01111100	11000000
11111110	01100110	11000000
11000110	01100110	11000000
11000110	01100110	11000010
11000110	01100110	01100110
11000110	11111100	00111100
00000000	00000000	00000000
00000000	00000000	00000000
00000000	00000000	00000000
00000000	00000000	00000000

Figura 4.7: Padrão de bits dos caracteres A, B e C

Considerando o formato do caractere utilizado, a imagem na tela é formada por 2400 caracteres, ou 30 linhas com 80 caracteres cada. Para armazenar estes 2400 caracteres, é utilizada outra memória denominada LAYOUT_ROM. Parte do conteúdo desta memória, mais exatamente a parte correspondente à região central da imagem, pode ser vista na figura 4.8.

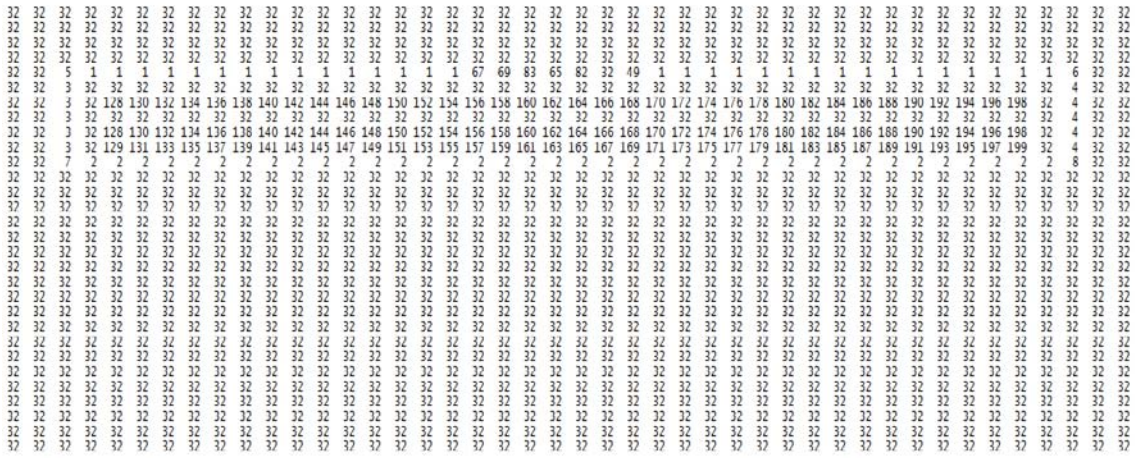


Figura 4.8: Parte do conteúdo da memória referente ao visor do Cesar

O módulo CONTROLE_VGA utiliza uma série de comparadores em conjunto com os sinais fornecidos pelo módulo Gerador, mais as duas memórias, para calcular os valores RGB de cada um dos pixels que formam a imagem. Assim, este módulo utiliza o conteúdo da posição da memória LAYOUT_ROM, correspondente a determinado caractere, para definir o que será exibido na tela. Na maior parte do tempo o valor acessado nesta memória contém um código ASCII. Este código ASCII é utilizado em conjunto com os bits menos significativos do sinal PIXEL_Y na formação do endereço utilizado para acessar um byte na memória FONT_ROM. Cada byte fornece o padrão de bits de uma das 16 linhas que formam cada um dos símbolos representados pelo código ASCII. Para que seja possível visualizar um caractere na tela, basta que sejam atribuídos, para um bit com valor 0, valores RGB diferentes dos que são utilizados para um bit com valor 1.

Como se pode ver na figura 4.8, três linhas contêm valores superiores a 128. Estes valores indicam que o conteúdo exibido na tela vem de uma das 36 entradas correspondentes ao visor do Cesar. Na primeira linha, os bits (6:1), do valor lido da memória LAYOUT_ROM, são utilizados para determinar qual das entradas deve ser selecionada no multiplexador MUX36_1. O código ASCII dado pelo conteúdo da entrada escolhida determina qual será o endereço acessado na memória FONT_ROM e, por conseqüência, qual o símbolo ASCII exibido. Se o valor armazenado na entrada não corresponder ao código ASCII de um símbolo visível, é exibido um espaço em branco na posição referente a esta entrada. Na prática, esta linha representa o visor do César.

Como cada saída do Cesar pode conter 256 valores diferentes, nem todos os valores correspondem a um símbolo visível na codificação ASCII. Para que se possa visualizar o conteúdo de todas as 36 saídas do Cesar, independentemente do valor, as outras duas linhas exibem o conteúdo destas saídas em hexadecimal. Para exibição destas linhas, os bits (6:1) do valor lido da memória LAYOUT_ROM, também são utilizados para selecionar uma das entradas, mas neste caso, o bit menos significativo deste valor é utilizado para selecionar, no multiplexador MUX2_1, os quatro primeiros ou os quatro últimos bits da entrada. O valor destes quatro bits é utilizado pelo módulo CONV_HEXA, que converte este valor no código ASCII correspondente ao símbolo adequado na representação hexadecimal. Este código ASCII determina qual será o endereço acessado na memória FONT_ROM.

A figura 4.9 mostra um exemplo da imagem exibida pelo MODULO_VGA.

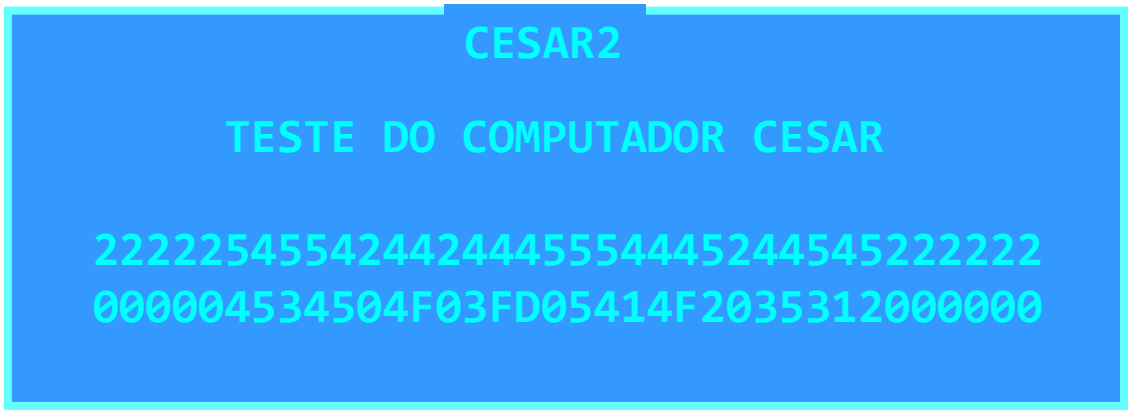


Figura 4.9: Visor do Cesar, como exibido no monitor

4.2 Módulo Teclado

O circuito TECLADO, mostrado na figura 4.10, tem a função de enviar um dado recebido através do teclado para o processador, gerando uma interrupção, conforme visto no capítulo anterior. O teclado é conectado à porta PS/2 da placa e comunica-se com o circuito através das entradas CLK_TECLADO e TCL_IN. Toda vez que uma tecla é pressionada, um código de varredura (*scan code*) é enviado pelo teclado, sendo recebido pela entrada TCL_IN.

Figura 4.10: Módulo TECLADO

Os *scan codes* são enviados serialmente como uma sequência de 11 bits, como mostrado na figura 4.11. O teclado controla tanto a linha de dados como a de clock. Para enviar um dado, primeiro o teclado prepara o envio do bit de *start*, colocando a linha de dados no nível lógico 0. Logo após, o teclado baixa o sinal de clock por aproximadamente 35 µs. Os outros 10 bits do código são então enviados com um período de clock de aproximadamente 70 µs. Os oito bits correspondentes ao *scan code* são enviados, do bit menos significativo até o mais significativo, e são sucedidos por um bit de paridade e por um bit de *stop*. Mais informações sobre o funcionamento do teclado, bem como um exemplo de código VHDL para receber dados do teclado, podem ser vistas em (HAMBLEN, 2002).

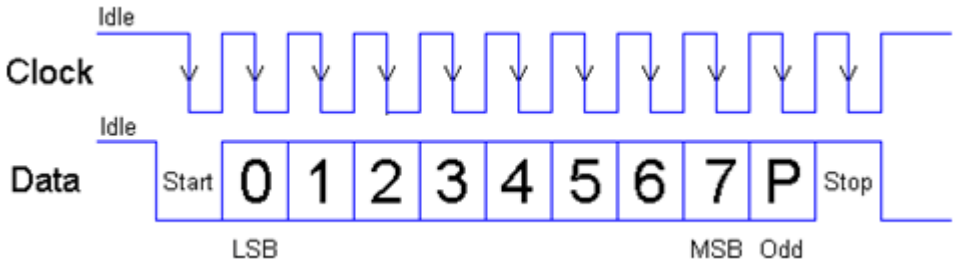


Figura 4.11: Formato do sinal enviado pelo teclado ao ser pressionada uma tecla

Para cada tecla existe um código para quando ela é pressionada, denominado de *make code*, e um código para quando ela é solta, denominado *break code*. Na verdade, o *break code* consiste no código F0 seguido do *make code* da tecla. Assim, quando a tecla SPACE é pressionada, por exemplo, o teclado envia o código 29 em hexadecimal. Já quando a mesma tecla é solta, o código “F0 29” é enviado pelo teclado. A figura 4.12 mostra as teclas com seus respectivos *scan codes*.

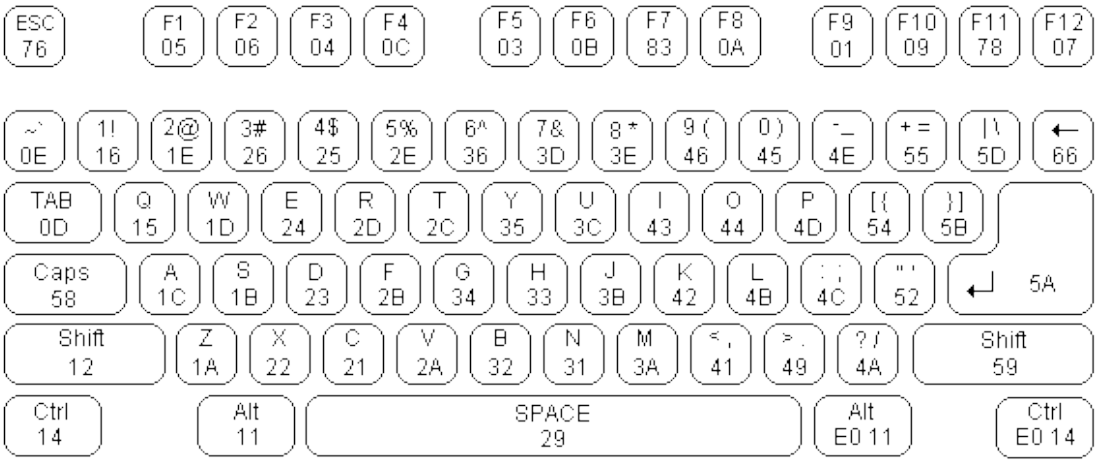


Figura 4.12: Principais teclas com seus respectivos *scan codes* (http://www.beyondlogic.org/keyboard/keybrd.htm)

O código VHDL do circuito que implementa a interface do teclado é composto basicamente por três processos. O primeiro processo utiliza o clock de 25 MHz, em conjunto com o registrador de deslocamento CLK_COUNT, para filtrar eventuais oscilações no sinal de clock vindo do teclado. Já o segundo processo, tem a função de sinalizar, através do sinal T1, quando um novo dado foi recebido do teclado. Um sinal de confirmação (ACK) de que o dado foi processado, enviado pelo processador, desativa o sinal T1, evitando que uma tecla pressionada no teclado seja processada várias vezes.

Por fim, o terceiro processo utiliza outro registrador de deslocamento para receber os oito bits referentes ao *make code* enviado pelo teclado. Quando uma tecla é pressionada, este processo sinaliza para o segundo processo que um dado foi recebido, através do sinal READY_TCL. Este processo também converte o código recebido para seu valor correspondente na codificação ASCII. É este valor que é passado ao processador. Como o teclado envia um código tanto quando a tecla é pressionada como quando ela é solta, utiliza-se o sinal F0_FLAG, que sinaliza quando o código F0 é recebido, fazendo com que o *break code*, gerado quando uma tecla é solta, seja ignorado.

Nem toda a tecla pressionada, no entanto, resulta em um dado enviado para o processador. Além das teclas ignoradas pelo processador, por não possuírem um código ASCII correspondente, a tecla SHIFT, por exemplo, só envia um sinal ao teclado se vier acompanhada de outra tecla. São utilizados os sinais SHIFT_FLAG e SHIFT_HOLD para controlar o dado enviado ao teclado no caso da utilização da tecla SHIFT. Quando a tecla SHIFT é utilizada em conjunto com outra tecla, o *make code* recebido é composto pelo código da tecla SHIFT seguido pelo código da outra tecla. Assim, se a tecla SHIFT for pressionada em conjunto com a tecla “1”, por exemplo, o *make code* recebido será “12 16”, e o código ASCII enviado para o processador corresponde ao caractere “!”, ao invés de “1”. O controle de qual código ASCII é enviado ao processador neste caso, é feito pelo sinal SHIFT_FLAG. Já o sinal SHIFT_HOLD controla se a tecla SHIFT continua sendo pressionada, quando a outra tecla é solta.

4.3 Circuito Completo

A figura 4.13 mostra o circuito completo utilizado para o teste do processador no FPGA. Além dos módulos do processador, do VGA e do teclado, também são utilizados dois circuitos *debouncers*, que filtram o sinal dos botões de RESET e RESUME, e dois DCMs (*Digital Clock Manager*) para dividir a frequência de clock de 100 MHz, disponível na placa.

DCM1 fornece a frequência de 25 MHz utilizada nos módulos Teclado e VGA. DCM2 fornece uma frequência de aproximadamente 60 MHz para a versão estrutural do processador. Na versão comportamental este segundo DCM não é necessário, pois é possível utilizar a frequência de 100 MHz.

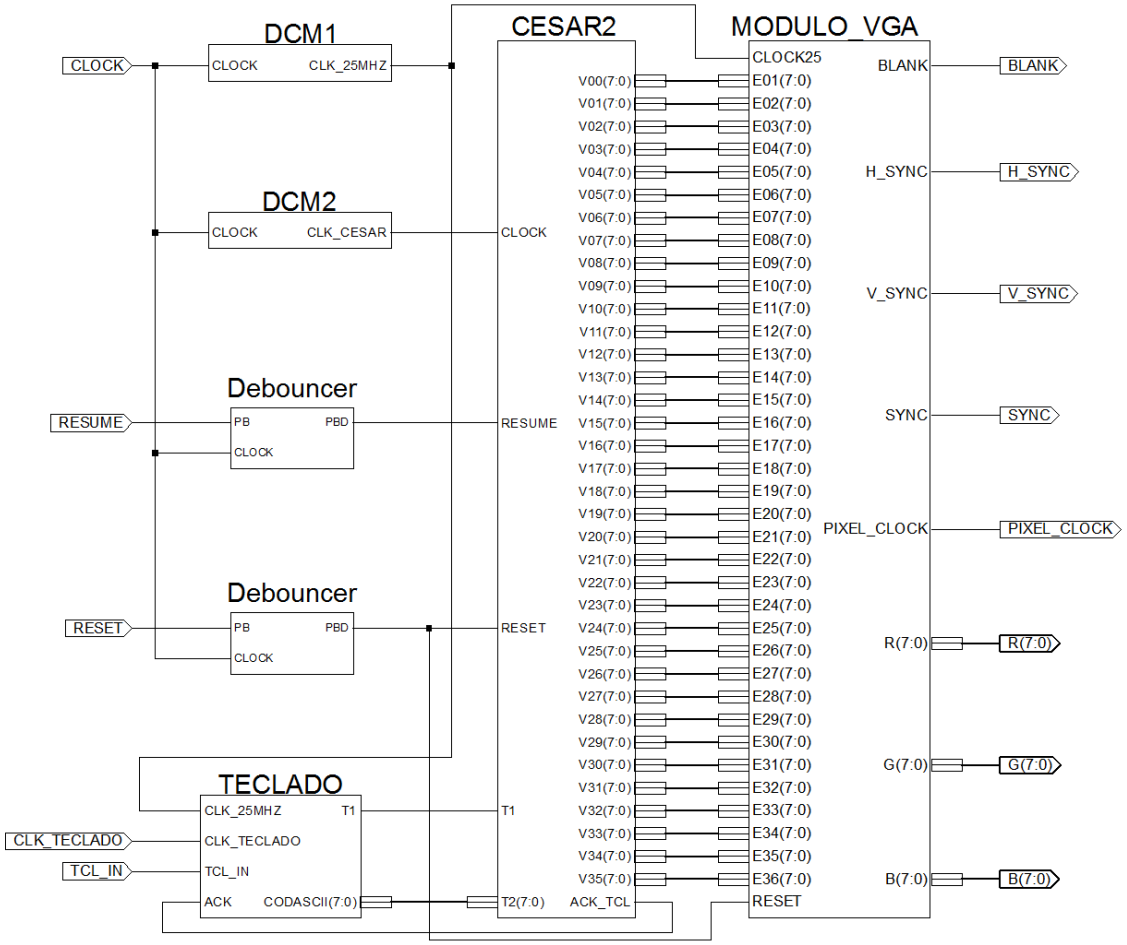


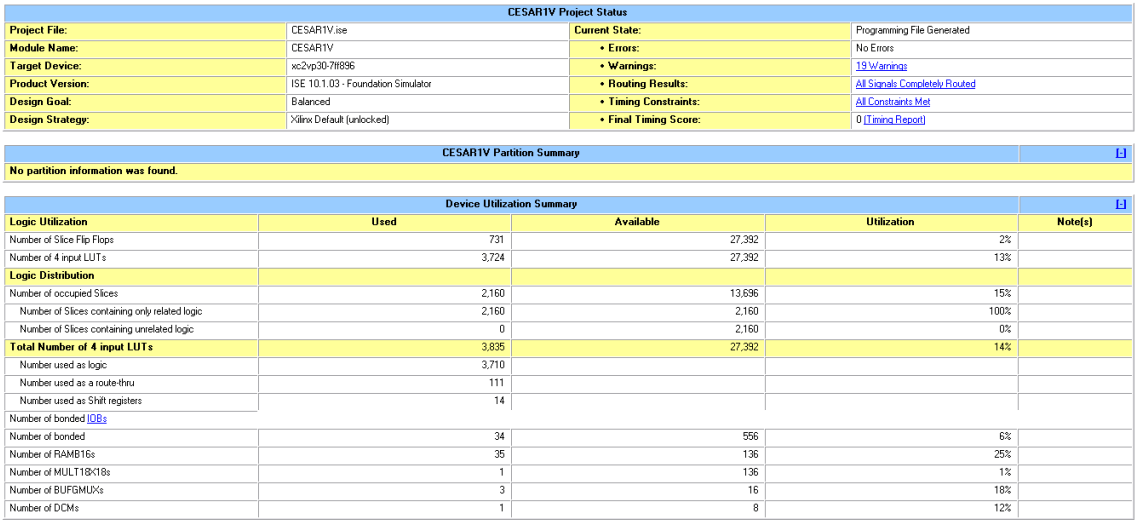
Figura 4.13: Circuito completo utilizado nos testes

Para o teste do processador na placa, além dos programas encontrados em (WEBER, 2008), foram utilizados programas que testam a entrada de dados pelo teclado, incluindo uma versão do programa implementado por alunos como um dos trabalhos obrigatórios deste semestre na disciplina de Arquitetura e Organização de Computadores I da UFRGS.

Todos os testes efetuados tiveram êxito, no que se refere a possíveis falhas do processador. Alguns problemas, entretanto, foram encontrados com o software IMPACT, que faz a programação do circuito na placa. Em algumas raras ocasiões, o programa gravado na memória parecia não estar sendo executado. A determinação da natureza do erro, se no circuito ou no software, não foi possível, mas a probabilidade de

ser uma falha do software não é pequena, levando-se em consideração a grande quantidade de problemas encontrados no software ISE, utilizado na síntese e roteamento do circuito.

As figuras 4.14 e 4.15 mostram os relatórios de utilização de elementos lógicos, fornecidos pela ferramenta de síntese, para o circuito completo nas versões estrutural e comportamental do processador. Foi utilizada a estratégia de otimização para velocidade.

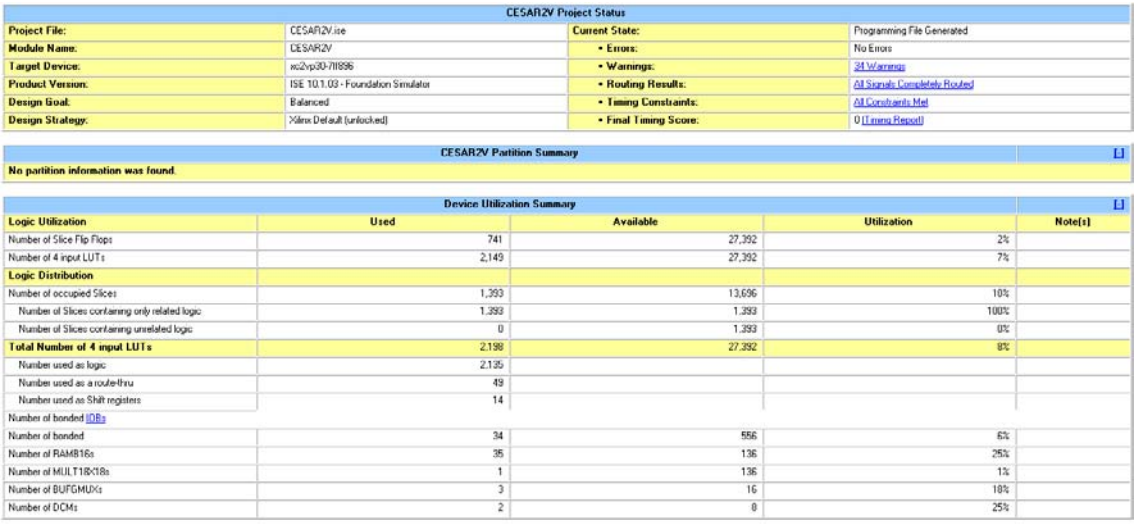


CESAR1V Project Status			
Project File:	CESAR1V.ise	**Current State:**	Programming File Generated
Module Name:	CESAR1V	**• Errors:**	No Errors
Target Device:	xc2vp30-7ff896	**• Warnings:**	19 Warnings
Product Version:	ISE 10.1.03 - Foundation Simulator	**• Routing Results:**	All Signals Completely Routed
Design Goal:	Balanced	**• Timing Constraints:**	All Constraints Met
Design Strategy:	Xilinx Default (unlocked)	**• Final Timing Score:**	0 (Timing Report)

CESAR1V Partition Summary
No partition information was found.

Device Utilization Summary				
Logic Utilization	**Used**	**Available**	**Utilization**	**Note(s)**
Number of Slice Flip Flops	731	27,392	2%	
Number of 4 input LUTs	3,724	27,392	13%	
Logic Distribution				
Number of occupied Slices	2,160	13,696	15%	
Number of Slices containing only related logic	2,160	2,160	100%	
Number of Slices containing unrelated logic	0	2,160	0%	
Total Number of 4 input LUTs	3,835	27,392	14%	
Number used as logic	3,710			
Number used as a route-thru	111			
Number used as Shift registers	14			
Number of bonded IOBs				
Number of bonded	34	556	6%	
Number of RAMB16s	35	136	25%	
Number of MULT18X18s	1	136	1%	
Number of BUFGMUXs	3	16	18%	
Number of DCMs	1	8	12%	

Figura 4.14: Resultado da síntese do circuito completo da versão comportamental



CESAR2V Project Status			
Project File:	CESAR2V.ise	**Current State:**	Programming File Generated
Module Name:	CESAR2V	**• Errors:**	No Errors
Target Device:	xc2vp30-7ff896	**• Warnings:**	34 Warnings
Product Version:	ISE 10.1.03 - Foundation Simulator	**• Routing Results:**	All Signals Completely Routed
Design Goal:	Balanced	**• Timing Constraints:**	All Constraints Met
Design Strategy:	Xilinx Default (unlocked)	**• Final Timing Score:**	0 (Timing Report)

CESAR2V Partition Summary
No partition information was found.

Device Utilization Summary				
Logic Utilization	**Used**	**Available**	**Utilization**	**Note(s)**
Number of Slice Flip Flops	741	27,392	2%	
Number of 4 input LUTs	2,149	27,392	7%	
Logic Distribution				
Number of occupied Slices	1,393	13,696	10%	
Number of Slices containing only related logic	1,393	1,393	100%	
Number of Slices containing unrelated logic	0	1,393	0%	
Total Number of 4 input LUTs	2,198	27,392	8%	
Number used as logic	2,135			
Number used as a route-thru	49			
Number used as Shift registers	14			
Number of bonded IOBs				
Number of bonded	34	556	6%	
Number of RAMB16s	35	136	25%	
Number of MULT18X18s	1	136	1%	
Number of BUFGMUXs	3	16	18%	
Number of DCMs	2	8	25%	

Figura 4.15: Resultado da síntese do circuito completo da versão estrutural

Por fim, para facilitar a identificação de eventuais falhas no processador, foi criada ainda uma terceira versão, que difere das outras apenas pela disponibilização dos valores dos registradores e códigos de condição como saídas do circuito. Também foi adicionado um sinal de status, que informa se o processador está executando ou encontra-se no estado HALT.

Esta terceira versão também necessitou de modificações no módulo VGA, para que fossem exibidas as informações extras em um monitor. A figura 4.16 mostra como estas informações são exibidas no monitor, em conjunto com o visor do Cesar mostrado na

figura 4.9. Já a figura 4.17 mostra o resultado da síntese da versão 3, tendo como base a implementação estrutural do processador. Como se pode perceber, este circuito utiliza um número consideravelmente maior de elementos lógicos.

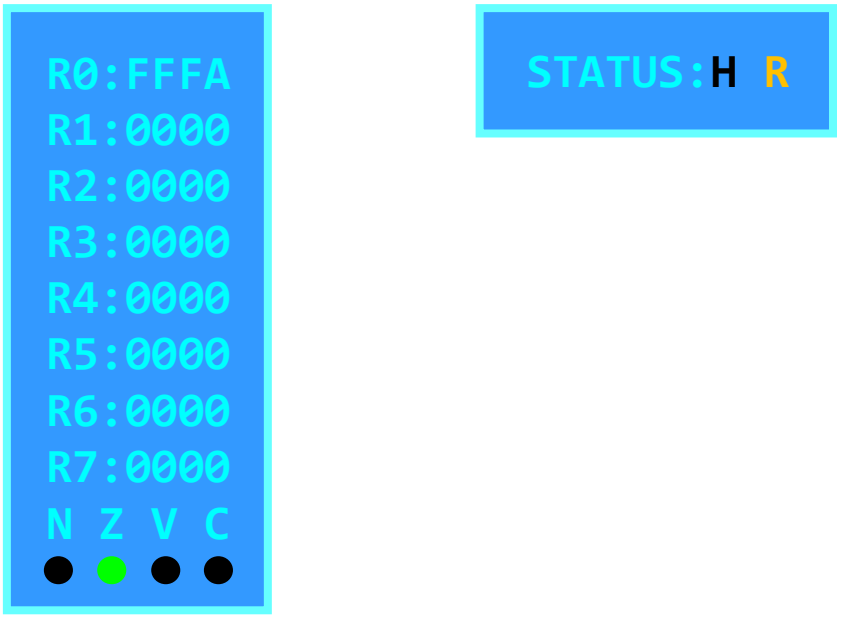


Figura 4.16: Informações adicionais exibidas pela versão 3 do processador

CESAR2V3 Project Status (11/17/2010 - 01:39:45)

Project File:	CESAR2V3.ise	**Current State:**	Programming File Generated
Module Name:	CESAR2V3	**• Errors:**	No Errors
Target Device:	xc2vp30-7ff896	**• Warnings:**	80 Warnings
Product Version:	ISE 10.1.03 - Foundation Simulator	**• Routing Results:**	All Signals Completely Routed
Design Goal:	Balanced	**• Timing Constraints:**	All Constraints Met
Design Strategy:	Xilinx Default (unlocked)	**• Final Timing Score:**	0 (Timing Report)

CESAR2V3 Partition Summary

No partition information was found.

Device Utilization Summary

Logic Utilization	Used	Available	Utilization	Note(s)
Number of Slice Flip Flops	743	27,392	2%	
Number of 4 input LUTs	2,485	27,392	9%	
Logic Distribution				
Number of occupied Slices	1,506	13,696	11%	
Number of Slices containing only related logic	1,506	1,506	100%	
Number of Slices containing unrelated logic	0	1,506	0%	
Total Number of 4 input LUTs	2,587	27,392	9%	
Number used as logic	2,469			
Number used as a route-thru	102			
Number used as Shift registers	16			
Number of bonded IOBs				
Number of bonded	34	556	6%	
Number of RAMB16s	35	136	25%	
Number of MULT18X18s	1	136	1%	
Number of BUFGMUXs	3	16	18%	
Number of DCMs	2	8	25%	

Figura 4.17: Resultado da síntese do circuito da versão 3

5 CONCLUSÃO E SUGESTÕES PARA TRABALHOS FUTUROS

Examinando os objetivos que foram traçados ao término do TG1, percebe-se que este trabalho foi bastante além do inicialmente previsto. Se no início havia dúvida a respeito da possibilidade de implementação do sistema de entrada e saída, logo ficou claro que, embora adicionasse grande complexidade ao trabalho, ficaria muito difícil a realização de qualquer teste prático no FPGA sem este sistema.

Outro aspecto importante foi a necessidade do desenvolvimento de circuitos separados para a interface de vídeo e de teclado. Esta etapa, embora trabalhosa, foi bastante gratificante, por permitir a visualização do funcionamento do processador com muito mais informações do que seria possível utilizando um simples conjunto de quatro LEDs disponíveis na placa. Além disto, puderam ser testados programas mais complexos, com entrada de dados, como os que são realizados nos trabalhos das disciplinas de arquitetura de computadores.

Também ao final do TG1, o planejado era escolher apenas um método de implementação VHDL para o trabalho. Entretanto, a realização da descrição comportamental do processador acabou por estimular a curiosidade de se projetar um circuito para a descrição estrutural. Embora este último modelo de descrição demande mais tempo, devido à necessidade de se projetar o circuito com antecedência, permite uma maior liberdade de projeto, uma vez que cada componente do processador pode ser desenvolvido separadamente.

De qualquer forma, analisando o desempenho de ambas as versões implementadas, em comparação com o desempenho dos processadores da família PDP-11 dos anos setenta e oitenta, percebe-se claramente a evolução tecnológica dos dispositivos de hardware programável. Se quando de sua criação, estes dispositivos serviam apenas para teste de sistemas digitais, hoje já é possível desenvolver processadores complexos que rodam com desempenho aceitável em FPGAs.

Claro que o fato do processador Cesar, desenvolvido no trabalho, rodar em hardware programável com um desempenho vinte a trinta vezes superior ao do PDP-11, não diz muita coisa, afinal são trinta anos de evolução da tecnologia. Mas o desempenho do processador rodando na FPGA também é muito superior ao desempenho do simulador utilizado no estudo da arquitetura.

Ainda sobre o desempenho, o processador Cesar, da forma como foi implementado, demanda um mínimo de quatro ciclos de clock para executar uma instrução, podendo chegar a até dezenove ciclos, dependendo do modo de endereçamento utilizado. Se comparado aos processadores atuais, mesmo alguns processadores *soft-core*, este não pode ser considerado um processador muito eficiente. Processadores modernos

conseguem executar até mais de uma instrução por ciclo de clock, empregando técnicas de paralelismo como *pipelines* e superescalaridade.

Uma tentativa de implementação da arquitetura Cesar com uma estrutura de *pipeline* seria um desafio bastante interessante em possíveis trabalhos futuros. O problema de como lidar com os diferentes modos de endereçamento neste caso, certamente não é trivial. Uma tarefa um pouco mais simples talvez fosse a tentativa de, nas instruções de dois operandos, tentar realizar a busca do segundo operando concomitantemente à busca do primeiro. Neste caso, o maior cuidado seria no acesso à memória, devido à possibilidade de acessos simultâneos.

Outro trabalho que poderia ser realizado se refere à programação do processador. Os processadores da família PDP-11 estiveram entre os primeiros a rodar o sistema operacional Unix. Poderia ser desenvolvido um esboço de sistema operacional, ainda que primitivo, que facilitasse a carga e execução de programas no processador Cesar. Isto exigiria a adição de outros dispositivos de entrada de dados, além do teclado. Também seriam necessárias alterações no código do processador para efetuar a comunicação com estes dispositivos. Embora complexas, estas alterações permitiriam que o processador Cesar pudesse ser utilizado mais facilmente em múltiplas aplicações, desde o aprendizado da arquitetura de computadores, até o desenvolvimento de sistemas digitais complexos com o auxílio deste processador.

REFERÊNCIAS

(WEBER, 2008) WEBER, R. F. **Fundamentos de Arquitetura de Computadores**. 3.ed. Porto Alegre: Instituto de Informática da UFRGS: Bookman, 2008. 320 p. (Série Livros Didáticos, n.8).

(ERCEGOVAC, 2002) ERCEGOVAC, M. et al. **Introdução aos Sistemas Digitais**. Porto Alegre: Bookman, 2002.

(BROWN, 2005) BROWN, S; VRANESIC, Z. **Fundamentals of Digital Logic with VHDL Design**. Second Edition. New York: McGraw-Hill, 2005.

(HWANG, 2005) HWANG, E. O. **Digital Logic and Microprocessor Design With VHDL**. CL-Engineering, 2005. 588 p.

(PEDRONI, 2004) PEDRONI, V. A. **Circuit Design with VHDL**. Cambridge, Massachusetts: The MIT Press, 2004. 375 p.

(HAMBLEN, 2002) HAMBLEN, J. O; FURMAN, M. D. **Rapid Prototyping of Digital Systems A Tutorial Aproach**. Second Edition. New York: Kluwer Academic Publishers, 2002.

(CHU, 2008) CHU, P. P. **FPGA Prototyping By VHDL Examples Xilinx Spartan-3 Version**. New Jersey: John Wiley & Sons, 2008. 440 p.

(WILSON, 2007) WILSON, P. **Design Recipes for FPGAs**. Oxford: Newnes, 2007. 289 p.

(PERRY, 2002) Perry, D. L. VHDL **Programming by Example.** Fourth Edition. New York: McGraw-Hill, 2002. 478 p.

(FAIRCHILD, 2002) FAIRCHILD SEMICONDUTOR. **FMS3818 Triple Video D/A Converter**, 2002. Disponível em: <http://www.datasheetcatalog.org/datasheet/fairchild/FMS3818.pdf>. Acesso em: setembro. 2010.

(XILINX, 2002) XILINX. **What is the Pinout Area Constarints Editor (PACE).** XAPP419 (v1.0), October 28, 2002. Disponível em: <http://www.xilinx.com/support/documentation/application_notes/xapp419.pdf>. Acesso em: maio. 2010.

(XILINX, 2007a) XILINX. **Virtex-II Pro and Virtex-II Pro X Platform FPGAs: Complete Data Sheet.** DS083 (v4.7), November 5, 2007. Disponível em: <http://www.xilinx.com/support/documentation/data_sheets/ds083.pdf>. Acesso em: maio. 2010.

(XILINX, 2007b) XILINX. **Virtex-II Pro and Virtex-II Pro X FPGA User Guide.** UG012 (v4.2), November 5, 2007. Disponível em:

<http://www.xilinx.com/support/documentation/user_guides/ug012.pdf>. Acesso em: maio. 2010.

(XILINX, 2008a) XILINX. **Constraints Guide 10.1**, 2008. Disponível em: <http://www.xilinx.com/itp/xilinx10/books/docs/cgd/cgd.pdf>. Acesso em: agosto. 2010.

(XILINX, 2008b) XILINX. **Synthesis and Simulation Design Guide 10.1**, 2008. Disponível em: <http://www.xilinx.com/itp/xilinx10/books/docs/sim/sim.pdf>. Acesso em: agosto. 2010.

(XILINX, 2009a) XILINX. **Xilinx University Program Virtex-II Pro Development System. Hardware Reference Manual.** UG069 (v1.2) July 21, 2009. Disponível em: <http://www.xilinx.com/univ/XUPV2P/Documentation/ug069.pdf>. Acesso em: maio. 2010.

(XILINX, 2009b) XILINX. **ISE In-Depth Tutorial.** UG695 (v 11.2) June 24, 2009. Último acesso em maio, 2010. Disponível em: <http://www.xilinx.com/support/documentation/sw_manuals/xilinx11/ise11tut.pdf>.

ANEXO A <ARTIGO TG1>

GUSTAVO KAEFER ORTH

Implementação em Hardware da Arquitetura do Computador Hipotético CESAR

Artigo Final do Trabalho de Graduação 1.

Prof. Dr. Carlos Arthur Lang Lisbôa
Orientador

Porto Alegre, junho de 2010.

1. Introdução

A crescente disponibilidade de ferramentas de síntese que permitem a prototipagem rápida de circuitos digitais tornou viável o desenvolvimento e implementação física de circuitos de relativa complexidade no ambiente acadêmico. Paralelamente, o surgimento de dispositivos de hardware programável que se tornam cada vez mais baratos e com maior capacidade, viabilizou a implementação de tais circuitos com baixo custo, permitindo que os mesmos sejam não apenas projetados, mas também implementados e testados.

Como processador alvo deste trabalho, foi escolhido o do computador CESAR, uma máquina baseada na arquitetura dos processadores PDP-11, e que é usada no ensino de arquitetura e organização de computadores nos cursos de Bacharelado em Ciência da Computação e Engenharia de Computação. Quanto à implementação, optou-se pelo uso de uma linguagem de descrição de hardware (VHDL) e de placas de desenvolvimento para FPGAs (*Field Programmable Gate Arrays*) disponíveis nos laboratórios de pesquisa do Instituto de Informática.

Este trabalho está estruturado como segue: na Seção 2, é justificada a opção pelo processador do CESAR. A Seção 3 detalha a etapa de planejamento da implementação e a Seção 4 discute alternativas de implementação usando VHDL. Na Seção 5 são descritas as atividades desenvolvidas durante o semestre e definido o cronograma das atividades para o Trabalho de Graduação II.

2. Motivação: o computador CESAR

Dentre os computadores hipotéticos criados para facilitar a aprendizagem da arquitetura de computadores, nas disciplinas introdutórias sobre arquitetura e organização de computadores dos cursos de Ciência da Computação e Engenharia de Computação da UFRGS, o computador CESAR é o último a ser estudado e sua arquitetura é consideravelmente mais complexa que a dos demais. Esta maior complexidade decorre do fato dele ser inspirado na arquitetura de um processador real de grande sucesso comercial. O CESAR compartilha suas características básicas com a família de processadores PDP-11, da Digital, que fez muito sucesso durante os anos setenta e inicio dos anos oitenta (WEBER, 2008).

Embora seu simulador já seja utilizado há muitos anos, ele é o único dos computadores hipotéticos cuja implementação em hardware ainda não foi realizada por alunos da graduação.

A seguir são listadas as características básicas da arquitetura do computador CESAR:

- Largura de dados e de endereços de 16 bits.
- Memória de 64kB, sendo os dados armazenados na memória usando a disposição *big endian*, na qual o byte mais significativo de uma palavra é armazenado em um byte e o menos significativo no byte seguinte. Embora seja uma arquitetura de 16 bits, o CESAR endereça a memória a byte, uma vez que algumas instruções são codificadas em apenas um byte.
- Oito registradores de uso geral (R0 a R7). Dentre estes registradores, R6 é usado como ponteiro da pilha (SP) e R7 como apontador de programa (PC).

- Oito modos de endereçamento, sendo quatro modos básicos (registrador, pós-incrementado, pré-decrementado e indexado) e quatro versões dos modos básicos com endereçamento indireto. Outros modos de endereçamento (direto e imediato) podem ser obtidos com o uso do registrador R7 na formação do endereço do operando na memória.
- Registrador de estado com quatro códigos de condição: negativo (N), zero (Z), carry (C) e overflow (V).
- Dados representados em complemento de dois.
- Possui um total de 41 instruções, incluindo:
  - 15 instruções de desvio condicional.
  - 12 instruções aritméticas de um operando.
  - 6 instruções de dois operandos.
  - 2 instruções para chamada e retorno de sub-rotinas.
- Sistema de entrada e saída rudimentar que mapeia os últimos 38 endereços da memória para os registradores de interface com dois periféricos: um teclado (endereços 65.498 e 65.499) e um visor alfanumérico de 36 posições (endereços 65500 a 65.535).

3. Planejamento da implementação.

A primeira etapa da implementação do processador do CESAR incluiu a análise de sua especificação (WEBER, 2008) e a descrição do seu funcionamento em nível de transferência de registradores (*Register Transfer Level* ou RTL). Esta descrição é a etapa inicial e ao mesmo tempo a mais importante do trabalho de projeto do hardware do processador CESAR, uma vez que requer a elaboração de uma descrição detalhada de todas as operações de transferência de informações (dados e endereços) necessárias para a execução de cada uma das instruções que compõem a arquitetura do conjunto de instruções (ISA – *Instruction Set Architecture*) do processador (ERCEGOVAC, 2002).

Para melhor compreensão das tabelas, é necessário antecipar que uma das decisões de projeto já tomadas foi o uso de uma memória capaz de ler 16 bits (uma palavra) de cada vez. Assim sendo, o registrador de dados da memória (RDM) precisa ter 16 bits. Considerando isto, nas tabelas é utilizada a seguinte notação para indicar os elementos da arquitetura:

- E: registrador de códigos de condição, formado pelos bits N, Z, C e V.
- R: qualquer registrador (R0 a R7).
- RI: registrador de instruções.
- RDM: registrador de dados da memória.
- RDM_{cc}: os bits 8 a 11 do RDM, que são usados para especificar os códigos de condição que devem ser ligados ou desligados nas instruções SCC e CCC, respectivamente.
- RDM(2): segundo byte do RDM. Contém o deslocamento (-128 a +127) que é somado ao R7 (PC) nas instruções de desvio condicional quando a condição testada é verdadeira.
- REM: registrador de endereços da memória.
- Rt: registrador temporário.
- Na instrução JSR, Rx corresponde ao registrador de ligação, que recebe o endereço de retorno da sub-rotina, enquanto Re é o registrador usado no

cálculo do endereço da sub-rotina. Rx e Re podem ser quaisquer registradores (R0 a R7).

As etapas de Busca e Decodificação são executadas para todas as instruções.

Busca e Decodificação
REM←R7
Read
RI←RDM

Na etapa de Execução são executadas transferências específicas para cada instrução, mostradas nas tabelas seguintes.

NOP
R7←R7+1

SCC	**CCC**
E←E OR RDM_{cc} R7←R7+1	E←E AND NOT(RDM_{cc}) R7←R7+1

JSR			
Registrador	Registrador PI	Registrador PD	Indexado
R7←R7+2	R7←R7+2	R7←R7+2	R7←R7+2
	Rt←Re	Re←Re-2	REM←R7
	Re←Re+2	Rt←Re	Read R7←R7+2
	R6←R6-2	R6←R6-2	Rt ←RDM+Re
	REM←R6 RDM←Rx	REM←R6 RDM←Rx	R6←R6-2
	Write Rx←R7	Write Rx←R7	REM←R6 RDM←Rx
	R7←Rt	R7←Rt	Write Rx←R7
			R7←Rt
Registrador Indireto	PI Indireto	PD Indireto	Indexado Indireto
R7←R7+2	R7←R7+2	R7←R7+2	R7←R7+2
Rt←Re	REM←Re	Re←Re-2	REM←R7
R6←R6-2	Re←Re+2 Read	REM←Re	Read R7←R7+2
REM←R6 RDM←Rx	Rt←RDM	Read	REM←RDM+Re
Write Rx←R7	R6←R6-2	Rt←RDM	Read
R7←Rt	REM←R6 RDM←Rx	R6←R6-2	Rt←RDM
	Write Rx←R7	REM←R6 RDM←Rx	R6←R6-2
	R7←Rt	Write Rx←R7	REM←R6 RDM←Rx
		R7←Rt	Write Rx←R7
			R7←Rt

RTS
REM←R6
R6←R6+2 Read
R7←R
R←RDM

MOV			
Registrador	Registrador PI	Registrador PD	Indexado
Primeiro Operando			
R7←R7+2	R7←R7+2	R7←R7+2	R7←R7+2
Rt←R Atualiza E	REM←R	R←R-2	REM←R7
	Read R←R+2	REM←R	Read R7←R7+2
	Rt←RDM Atualiza E	Read	REM←R+RDM
		Rt←RDM Atualiza E	Read
			Rt←RDM Atualiza E
Segundo Operando			
R←Rt	REM←R RDM←Rt	R←R-2	REM←R7
	R←R+2 Write	REM←R RDM←Rt	Read R7←R7+2
		Write	REM←R+RDM RDM←Rt
			Write
Registrador Indireto	PI Indireto	PD Indireto	Indexado Indireto
Primeiro Operando			
R7←R7+2	R7←R7+2	R7←R7+2	R7←R7+2
REM←R	REM←R	R←R-2	REM←R7
Read	Read R←R+2	REM←R	Read R7←R7+2
Rt←RDM Atualiza E	REM←RDM	Read	REM←R+RDM
	Read	REM←RDM	Read
	Rt←RDM Atualiza E	Read	REM←RDM
		Rt←RDM Atualiza E	Read
			Rt←RDM Atualiza E
Segundo Operando			
REM←R RDM←Rt	REM←R	R←R-2	REM←R7
Write	Read R←R+2	REM←R	Read R7←R7+2
	REM←RDM RDM←Rt	Read	REM←R+RDM
	Write	REM←RDM RDM←Rt	Read
		Write	REM←RDM RDM←Rt
			Write

CMP			
Registrador	Registrador PI	Registrador PD	Indexado
Primeiro Operando			
R7←R7+2	R7←R7+2	R7←R7+2	R7←R7+2
Rt←R	REM←R	R←R-2	REM←R7
	Read R←R+2	REM←R	Read R7←R7+2
	Rt←RDM	Read	REM←R+RDM
		Rt←RDM	Read
			Rt←RDM
Segundo Operando			
Rt-R Atualiza E	REM←R	R←R-2	REM←R7
	Read R←R+2	REM←R	Read R7←R7+2
	Rt-RDM Atualiza E	Read	REM←R+RDM
		Rt-RDM Atualiza E	Read
			Rt-RDM Atualiza E
Registrador Indireto	PI Indireto	PD Indireto	Indexado Indireto
Primeiro Operando			
R7←R7+2	R7←R7+2	R7←R7+2	R7←R7+2
REM←R	REM←R	R←R-2	REM←R7
Read	Read R←R+2	REM←R	Read R7←R7+2
Rt←RDM	REM←RDM	Read	REM←R+RDM
	Read	REM←RDM	Read
	Rt←RDM	Read	REM←RDM
		Rt←RDM	Read
			Rt←RDM
Segundo Operando			
REM←R	REM←R	R←R-2	REM←R7
Read	Read R←R+2	REM←R	Read R7←R7+2
Rt-RDM Atualiza E	REM←RDM	Read	REM←R+RDM
	Read	REM←RDM	Read
	Rt-RDM Atualiza E	Read	REM←RDM
		Rt-RDM Atualiza E	Read
			Rt-RDM Atualiza E

BNE, BEQ, BPL, BMI, BVC, BVS, BCC, BCS, BGE, BLT, BGT, BLE, BHI, BLS	
Condição testada verdadeira	**Condição testada falsa**
R7←R7+2	R7←R7+2
R7←R7+RDM(2)	

BR
R7←R7+2
R7←R7+RDM(2)
JMP

Registrador	Registrador PI	Registrador PD	Indexado
R7←R7+2	R7←R7+2	R7←R7+2	R7←R7+2
	R7←R	R←R-2	REM←R7
	R←R+2	R7←R	Read R7←R7+2
			R7 ←RDM+R
Registrador Indireto	PI Indireto	PD Indireto	Indexado Indireto
R7←R7+2	R7←R7+2	R7←R7+2	R7←R7+2
R7←R	REM←R	R←R-2	REM←R7
	R←R+2 Read	REM←R	Read R7←R7+2
	R7←RDM	Read	REM←RDM+R
		R7←RDM	Read
			R7←RDM

SOB	
R7←R7+2	
R←R-1	
R ≠ 0	**R = 0**
R7←R7-RDM(2)	

CLR, NOT, INC, DEC, NEG, TST, ROR, ROL, ASR, ASL, ADC, SBC			
Registrador	Registrador PI	Registrador PD	Indexado
R7←R7+2	R7←R7+2	R7←R7+2	R7←R7+2
R←OP(R) Atualiza E	REM ←R	R←R-2	REM←R7
	R←R+2 Read	REM←R	Read R7←R7+2
	RDM←OP(RDM) Atualiza E	Read	REM←R+RDM
	Write	RDM←OP(RDM) Atualiza E	Read
		Write	RDM←OP(RDM) Atualiza E
			Write
Registrador Indireto	PI Indireto	PD Indireto	Indexado Indireto
R7←R7+2	R7←R7+2	R7←R7+2	R7←R7+2
REM←R	REM←R	R←R-2	REM←R7
Read	Read R←R+2	REM←R	Read R7←R7+2
RDM←OP(RDM) Atualiza E	REM←RDM	Read	REM←R+RDM
Write	Read	REM←RDM	Read
	RDM←OP(RDM) Atualiza E	Read	REM←RDM
	Write	RDM←OP(RDM) Atualiza E	Read
		Write	RDM←OP(RDM) Atualiza E
			Write

ADD, SUB, AND, OR			
Registrador	Registrador PI	Registrador PD	Indexado
Primeiro Operando			
R7←R7+2	R7←R7+2	R7←R7+2	R7←R7+2
Rt←R	REM←R	R←R-2	REM←R7
	Read R←R+2	REM←R	Read R7←R7+2
	Rt←RDM	Read	REM←R+RDM
		Rt←RDM	Read
			Rt←RDM
Segundo Operando			
R←R(op)Rt Atualiza E	REM←R	R←R-2	REM←R7
	Read R←R+2	REM←R	Read R7←R7+2
	RDM←Rt(op)RDM Atualiza E	Read	REM←R+RDM
	Write	RDM←Rt(op)RDM Atualiza E	Read
		Write	RDM←Rt(op)RDM Atualiza E
			Write
Registrador Indireto	PI Indireto	PD Indireto	Indexado Indireto
Primeiro Operando			
R7←R7+2	R7←R7+2	R7←R7+2	R7←R7+2
REM←R	REM←R	R←R-2	REM←R7
Read	Read R←R+2	REM←R	Read R7←R7+2
Rt←RDM	REM←RDM	Read	REM←R+RDM
	Read	REM←RDM	Read
	Rt←RDM	Read	REM←RDM
		Rt←RDM	Read
			Rt←RDM
Segundo Operando			
REM←R	REM←R	R←R-2	REM←R7
Read	Read R←R+2	REM←R	Read R7←R7+2
RDM←Rt(op)RDM Atualiza E	REM←RDM	Read	REM←R+RDM
Write	Read	REM←RDM	Read
	RDM←Rt(op)RDM Atualiza E	Read	REM←RDM
	Write	RDM←Rt(op)RDM Atualiza E	Read
		Write	RDM←Rt(op)RDM Atualiza E
			Write

HLT
R7←R7+1
Para processamento.

4. Considerações sobre a implementação em VHDL.

A partir da descrição RTL detalhada na Seção 2, a construção em VHDL da arquitetura do computador CESAR pode ser feita de várias maneiras (HWANG, 2005). Uma delas é a descrição estrutural de todos os módulos necessários para a implementação do computador. Neste caso, deve ser feito, à mão, o projeto de todo o circuito do computador. Este circuito, por sua vez, contém todas as estruturas necessárias para o funcionamento do computador, como registradores, multiplexadores, decodificadores, unidade aritmética e lógica e um módulo de controle. Neste circuito também deve aparecer o caminho de dados (*datapath*) do computador, com todas as interconexões entre os módulos, bem como todos os sinais que devem ser gerados pelo módulo de controle para o correto funcionamento do computador de acordo com a descrição RTL.

A codificação VHDL neste caso seria composta por várias entidades, cada qual descrevendo um componente do circuito. Assim, teríamos entidades para os registradores, decodificadores, multiplexadores, ULA e para o controle, que seria responsável por gerar os sinais necessários, a cada pulso de clock, para que sejam realizadas as transferências de dados e as operações da descrição RTL de cada instrução. Alguns exemplos de sinais de controle são carga e incremento/decremento de registrador, seleção de multiplexador e operação da ULA, e leitura ou escrita da memória. Por fim, ainda haveria uma entidade principal em que seriam declarados todos os componentes que formam o circuito e onde seria feito o mapeamento das portas de entrada e saída destes componentes, o que, em termos práticos, representa as interconexões entre os módulos do circuito.

A opção pela descrição estrutural do computador, se realizada, levanta algumas outras questões de projeto. Entre elas a implementação das instruções de desvio, com a soma realizada pela ULA, ou por um módulo somador próprio e a implementação dos registradores, com ou sem sinais de incremento e decremento.

A principal vantagem da descrição estrutural está na capacidade de se prever com um pouco mais de exatidão o circuito que será gerado no processo de síntese, realizado pela ferramenta de CAD. Além disto, conforme testes realizados com a arquitetura Neander, mais simples, este estilo de programação tende a gerar um circuito menor, ou seja, são necessários menos LUTs para descrever o respectivo circuito.

Uma das desvantagens da descrição estrutural é o tamanho do código VHDL gerado, uma vez que, em cada estado do módulo de controle devem ser gerados todos os sinais necessários ao funcionamento do circuito. Outra desvantagem está na dificuldade para se efetuar qualquer modificação no funcionamento do computador. Se, por exemplo, alguém desejasse adicionar alguma nova funcionalidade ao computador, isto exigiria um reprojeto do circuito, que, por sua vez, exigiria uma mudança nos sinais de controle, tornando necessária a alteração, não só do módulo de controle, como, muito provavelmente, de outras entidades componentes do circuito, além de exigir um remapeamento das portas nos componentes declarados na entidade principal do código VHDL do computador.

A segunda opção é a descrição comportamental do computador. Neste caso, o código VHDL é composto basicamente de uma entidade em cuja arquitetura as operações listadas na descrição RTL, como transferências de dados entre registradores, são explicitamente codificadas em VHDL.

Entre as vantagens deste tipo de descrição está a geração de um código VHDL mais enxuto e fácil de ser compreendido. Além disto, este tipo de descrição reduz significativamente o tempo gasto no desenvolvimento de qualquer projeto de hardware, uma vez que elimina a necessidade do projeto completo do circuito, com todos os sinais de controle e interconexões, deixando isto a cargo exclusivamente da ferramenta de síntese. Por fim, eventuais alterações são mais fáceis de ser implementadas, pois não envolvem necessariamente uma modificação completa no projeto.

As desvantagens residem principalmente na dificuldade de se prever o circuito que será sintetizado pela ferramenta de CAD. Isto tende a gerar um circuito mais complexo, com a utilização de um maior número de LUTs no FPGA, o que não significa, entretanto, que o desempenho do circuito gerado, em termos de velocidade máxima de operação, seja pior. Mesmo que a ferramenta de CAD gere um circuito com maior utilização de LUTs, este pode ser mais rápido do que um circuito gerado a partir de uma descrição estrutural.

Uma terceira opção é a utilização dos dois tipos de descrição. Até certo ponto esta opção é inevitável, uma vez que a descrição estrutural do computador é formada por entidades cuja arquitetura é descrita de forma comportamental. VHDL é uma linguagem de descrição de hardware que visa facilitar e agilizar o projeto de circuitos, então não faria sentido codificar uma ULA, por exemplo, em nível de portas lógicas. Seria quase como utilizar código de máquina em uma linguagem de programação de alto nível. O projeto puramente comportamental, por sua vez, também é impossível, uma vez que, pelo menos a memória será descrita em uma entidade independente, obrigando também a utilização de pelo menos um multiplexador na entrada do registrador de dados da memória.

De qualquer forma, a intenção do trabalho não é a de comparar vários métodos de implementação, nem de buscar o circuito com melhor desempenho, até porque isto depende muito da placa utilizada. O objetivo principal do trabalho é o de implementar a arquitetura CESAR de maneira bem documentada, a fim de facilitar os testes de funcionamento, para que contenha o menor número de falhas possível.

O sistema de entrada e saída do CESAR, embora rudimentar, adiciona uma considerável carga extra de complexidade à lógica do circuito do computador. Isto ocorre porque todas as operações que envolvam um operando localizado nos últimos 38 endereços da memória, devem ser realizadas sobre um operando de um byte, ao contrário da operação normal do CESAR, que realiza as operações sobre operandos de 16 bits, ou seja, dois bytes. Além disto, deve ser considerado que não é trivial determinar se o operando está ou não localizado nestas posições de memória, uma vez que o momento em que o endereço efetivo do operando é determinado, varia de acordo com o modo de endereçamento utilizado. Desta forma, a implementação ou não do sistema de entrada e saída dependerá da disponibilidade de tempo e das dificuldades encontradas ao longo do trabalho.

Embora se deseje que a descrição VHDL do CESAR seja o mais genérica possível, isto é, não seja especificamente projetada para rodar em um FPGA específico, algumas características do projeto terão que, obrigatoriamente, se adequar às características da placa escolhida. Por exemplo, embora na descrição da arquitetura do CESAR esteja especificado que os últimos 38 bytes de memória são mapeados para o sistema de entrada e saída, este mapeamento não é possível de ser realizado com a estrutura de

memória disponível na placa. Desta forma, o sistema de entrada e saída, se implementado, utilizará 38 registradores genéricos na sua implementação.

Outra característica relacionada ao sistema de memória disponível na placa, que acabou por influenciar no projeto do computador, é a possibilidade de se realizar dois acessos simultâneos, sejam leituras ou escritas. Como dito anteriormente, o CESAR, embora seja uma arquitetura de 16 bits, endereça a memória a byte. Como um acesso à memória dura pelo menos dois ciclos de clock, serão utilizados dois registradores de endereço, o segundo contendo o endereço do byte seguinte ao primeiro. Isto é especialmente importante no caso do CESAR, que possui vários modos de endereçamento e em que a execução de uma única instrução pode realizar muitos acessos à memória. Um exemplo que ilustra uma situação extrema é o de uma instrução ADD com os dois operandos na memória endereçados através do modo indexado indireto.

Independentemente do modelo de implementação a ser escolhido, o módulo de controle do processador, como já citado anteriormente, tem o comportamento descrito através de uma máquina de estados. Como o CESAR possui diferentes formatos e tamanhos de instruções, a partir do estado em que é feita a decodificação da instrução, o próximo estado é decidido de acordo com o tipo de instrução em execução. O principal motivo para esta divisão é o de facilitar a lógica de transição do estado atual para o próximo.

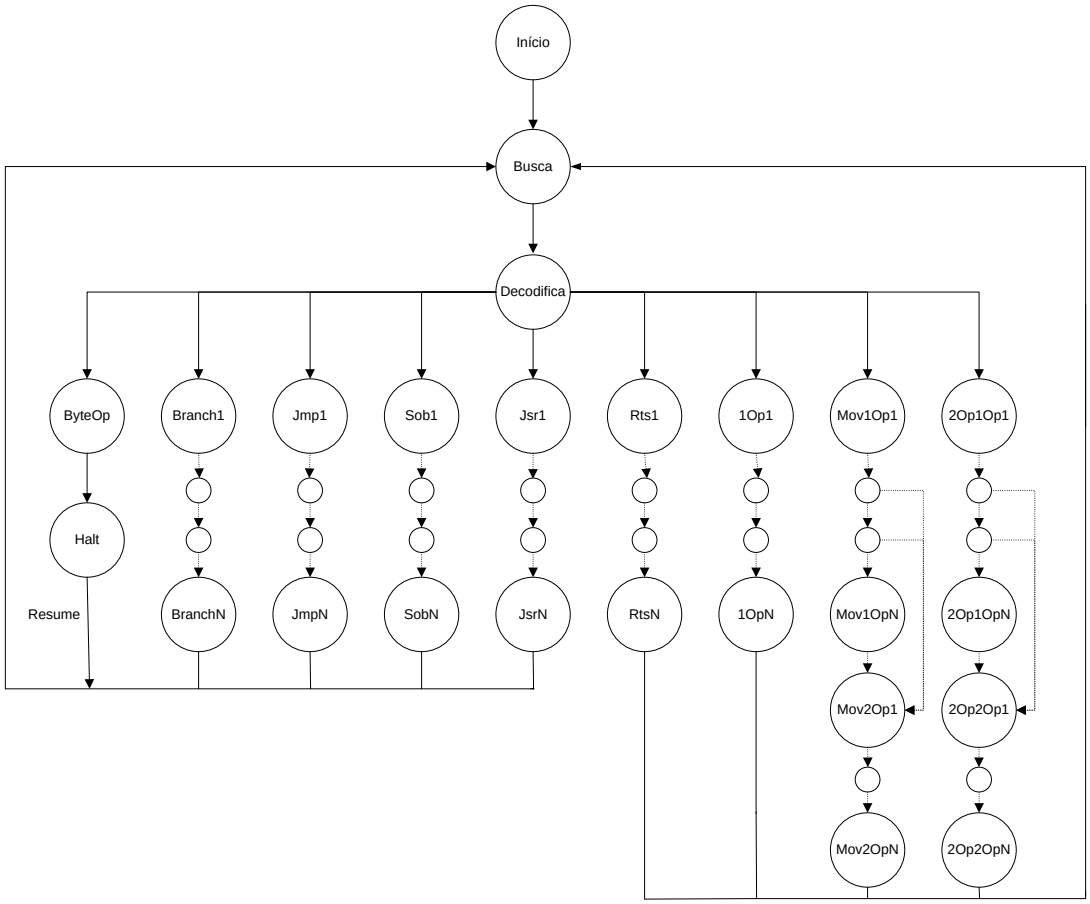


Figura 1. Esboço da máquina de estados a ser implementada

A Figura 1 representa um esboço da máquina de estados que se pretende implementar para o módulo de controle do CESAR. Futuras alterações poderão ser feitas, como uma segunda subdivisão dos tipos de instrução conforme o modo de endereçamento, o que multiplicaria o número de estados, porém simplificaria a lógica necessária em cada estado, além de facilitar os testes. A fase de busca da instrução compreende dois ou mais estados, conforme o modelo de implementação escolhido.

No projeto da máquina de estados, as instruções foram agrupadas de acordo com o formato do código da instrução e com a similaridade na sequência de transferências entre registradores necessárias para a sua execução. Desta forma, as instruções NOP, HLT, SCC e CCC, todas de um byte, foram agrupadas. A instrução RTS, embora também de um byte, apresenta uma descrição RTL mais complexa e, por isso, é descrita por uma sequência de estados diferente.

As instruções de desvio condicional (*branch*), que possuem um formato e descrição similares, foram também agrupadas em uma única sequência. O mesmo também ocorre com as instruções de um operando, cuja única diferença na descrição RTL, se dá na operação realizada pela ULA. Já as instruções JMP, SOB e JSR são executadas, cada uma, por uma sequência de estados própria, devido às diferenças de formato e passos necessários para a sua execução.

Por fim, as instruções de dois operandos são implementadas por duas sequências de estados. A primeira contém os passos para a busca do primeiro operando, enquanto a segunda contém os passos para a busca do segundo operando e para a execução da operação realizada pela instrução.

Ferramentas Utilizadas.

Abaixo estão listadas as principais ferramentas que serão utilizadas no desenvolvimento deste trabalho. Descrições mais detalhadas podem ser encontradas em (XILINX, 2007a), (XILINX, 2007b) e (XILINX, 2009a).

- Placa FPGA modelo Virtex-II Pro XC2VP30 da fabricante Xilinx. Dentre as características básicas deste FPGA, que serão importantes para a implementação do computador, destacam-se:
  - Um total de 30.816 Logic Cells.
  - Logic Cell = (1) 4-input LUT + (1) Flip-Flop + Carry Logic.
  - Um total de (136) 18 Kb Blocks of Block SelectRAM+.
  - True Dual-Port RAM.
- Software Xilinx ISE versão 10.1.
- Montador Daedalus Multi Assembler versão 1.0.2.4.
- Simulador WCesar16 versão 1.1.2.0.

5. Metodologia e cronograma de atividades.

Ao longo do semestre foram realizadas diversas reuniões com o orientador, para o esclarecimento de dúvidas, decisões de projeto e verificação do andamento do trabalho. Uma vez que estas reuniões foram realizadas a cada duas semanas, o cronograma das atividades já realizadas no TG1 foi subdividido desta mesma forma. Na segunda parte desta seção, é também apresentado o cronograma previsto para o TG2, que será desenvolvido no segundo semestre de 2010.

Cronograma de atividades realizadas no TG1:

Período de 23 de março a 4 de abril.

- Revisão dos conceitos básicos de arquitetura de computadores.
- Estudo da organização de arquiteturas mais simples: Neander e Ramses.
- Estudo detalhado da arquitetura do computador CESAR.

Período de 4 de abril a 20 de abril.

- Testes de funcionamento de todo o conjunto de instruções do computador CESAR, com o auxílio do simulador.
- Início da descrição RTL das instruções do computador CESAR.

Período de 20 de abril a 4 de maio.

- Término e verificação da descrição RTL das instruções do computador CESAR.
- Revisão de conceitos de Sistemas Digitais: Unidades Lógicas e Aritméticas e Máquinas de Estado (BROWN, 2005).
- Início do estudo da linguagem de descrição de hardware VHDL (PEDRONI, 2004), (BROWN, 2005) e (HWANG, 2005).

Período de 4 de maio a 18 de maio.

- Continuação do estudo da linguagem de descrição de hardware VHDL.
- Definição do modelo de placa FPGA a ser utilizado.
- Estudo das características básicas da placa FPGA da Xilinx através de manuais e data sheets disponíveis na internet.
- Instalação e testes da ferramenta ISE da Xilinx (XILINX, 2009b).

Período de 18 de maio a 1º de junho.

- Testes de programação VHDL: elementos básicos e arquitetura Neander.
- Projeto inicial da máquina de estados do controle da arquitetura CESAR.
- Testes de implementação de partes do conjunto de instruções da arquitetura CESAR.
- Início da redação do Artigo Final do TG1
- Início do projeto de implementação das operações da ULA do CESAR.

Período de 1º de junho a 8 de junho.

- Conclusão do Artigo Final do TG1.

Restante do semestre.

- Finalização do projeto da ULA do CESAR.
- Continuação dos testes de programação VHDL do conjunto de instruções da arquitetura CESAR.

Cronograma de atividades previstas para o TG2:

Agosto.

- Início dos testes de programação com a placa da Xilinx, conforme disponibilidade da mesma.
- Decisão final da forma de implementação em VHDL da arquitetura CESAR.

- Início da atividade de programação em VHDL do conjunto completo de instruções da arquitetura CESAR.

Setembro.

- Finalização da programação em VHDL do conjunto completo de instruções da arquitetura CESAR.
- Testes e simulações para a verificação do funcionamento correto do computador.
- Testes de funcionamento na placa FPGA.

Outubro.

- Redação da monografia final do TG2.

Novembro.

- Entrega e apresentação do trabalho.

6. Referências Bibliográficas.

(WEBER, 2008) WEBER, R. F. **Fundamentos de Arquitetura de Computadores**. 3.ed. Porto Alegre: Instituto de Informática da UFRGS: Bookman, 2008. 320 p. (Série Livros Didáticos, n.8).

(ERCEGOVAC, 2002) ERCEGOVAC, M. et al. **Introdução aos Sistemas Digitais**. Porto Alegre: Bookman, 2002.

(BROWN, 2005) BROWN, S; Vranesic, Z. **Fundamentals of Digital Logic with VHDL Design**. Second Edition. New York: McGraw-Hill, 2005.

(HWANG, 2005) HWANG, E. O. **Digital Logic and Microprocessor Design With VHDL**. CL-Engineering, 2005. 588 p.

(PEDRONI, 2004) PEDRONI, V. A. **Circuit Design with VHDL**. Cambridge, Massachusetts: The MIT Press, 2004. 375 p.

(XILINX, 2007a) XILINX. Virtex-II Pro and Virtex-II Pro X Platform FPGAs: Complete Data Sheet. DS083 (v4.7), November 5, 2007. Disponível em: <http://www.xilinx.com/support/documentation/data_sheets/ds083.pdf>. Acesso em: maio. 2010.

(XILINX, 2007b) XILINX. Virtex-II Pro and Virtex-II Pro X FPGA User Guide. UG012 (v4.2), November 5, 2007. Disponível em: <http://www.xilinx.com/support/documentation/user_guides/ug012.pdf>. Acesso em: maio. 2010.

(XILINX, 2009a) XILINX. Xilinx University Program Virtex-II Pro Development System. Hardware Reference Manual. UG069 (v1.2) July 21, 2009. Disponível em: <http://www.xilinx.com/univ/XUPV2P/Documentation/ug069.pdf>. Acesso em: maio. 2010.

(XILINX, 2009b) XILINX. ISE In-Depth Tutorial. UG695 (v 11.2) June 24, 2009. Último acesso em maio, 2010. Disponível em: <http://www.xilinx.com/support/documentation/sw_manuals/xilinx11/ise11tut.pdf>.