FUNDAÇÃO EURÍPIDES SOARES DA ROCHA
CENTRO UNIVERSITÁRIO EURÍPIDES DE MARÍLIA – UNIVEM
BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

DENISON MENEZES

ERNESTO KOJI OSHIRO

DESENVOLVIMENTO DE MICROCONTROLADORES UTILIZANDO FPGA E VHDL

MARÍLIA
2009

DENISON MENEZES
ERNESTO KOJI OSHIRO

DESENVOLVIMENTO DE MICROCONTROLADORES UTILIZANDO FPGA E VHDL

Trabalho de Conclusão de Curso apresentado ao curso de Bacharelado em Ciência da Computação do Centro Universitário Eurípides de Marília - UNIVEM, mantido pela Fundação de Ensino Eurípides Soares da Rocha, como requisito parcial para obtenção do grau de Bacharel em Ciência da Computação.

Orientador:
Prof. Ms. Ildeberto de Genova Bugatti

MARÍLIA
2009

Menezes, Denison; Oshiro, Ernesto Koji

Desenvolvimento de microcontroladores utilizando FPGA e VHDL / Denison Menezes; Ernesto Koji Oshiro; orientador: Ildeberto de Genova Bugatti. Marilia, SP: [s.n], 2009.

173 f.

Trabalho de Curso (Bacharelado em Ciência da Computação) - Curso de Ciência da Computação, Fundação de Ensino "Eurípides Soares da Rocha", mantenedora do Centro Universitário Eurípides de Marília – UNIVEM, Marília, 2009.

CENTRO UNIVERSITÁRIO EURÍPIDES DE MARÍLIA
BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

TRABALHO DE CONCLUSÃO DE CURSO – AVALIAÇÃO FINAL

Denison Menezes

DESENVOLVIMENTO DE MICROCONTROLADORES UTILIZANDO FPGA E VHDL

Banca examinadora da monografia apresentada ao Curso de Bacharelado em Ciência da Computação do UNIVEM/F.E.E.S.R., para obtenção do Título de Bacharel em Ciência da Computação.

Nota: 10,0 (Dez)

Orientador: Ildeberto de Gênova Bugatti

1°. Examinador: Leonardo Castro Botega

2°. Examinador: Fábio Dacêncio Pereira

Marília, 03 de dezembro de 2009.

CENTRO UNIVERSITÁRIO EURÍPIDES DE MARÍLIA
BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

TRABALHO DE CONCLUSÃO DE CURSO – AVALIAÇÃO FINAL

Ernesto Koji Oshiro

DESENVOLVIMENTO DE MICROCONTROLADORES UTILIZANDO FPGA E VHDL

Banca examinadora da monografia apresentada ao Curso de Bacharelado em Ciência da Computação do UNIVEM/F.E.E.S.R., para obtenção do Título de Bacharel em Ciência da Computação.

Nota: 9,5 (Nove e meio)

Orientador: Ildeberto de Gênova Bugatti

1°. Examinador: Leonardo Castro Botega

2°. Examinador: Fábio Dacêncio Pereira

Marília, 03 de dezembro de 2009.

MENEZES, Denison; OSHIRO, Ernesto Koji. **Desenvolvimento de Microcontroladores Utilizando FPGA e VHDL**. 2009. 173 f. Monografia (Bacharelado em Ciência da Computação) – Centro Universitário Eurípedes de Marília, Fundação de Ensino "Eurípedes Soares da Rocha", Marília, 2009.

RESUMO

A área de desenvolvimento de circuitos integrados digitais possui como uma das características a rapidez no desenvolvimento e pesquisas, o que possibilita a construção de componentes cada vez mais velozes. No entanto, não é somente a tecnologia de microeletrônica que viabiliza a construção de processadores com grande capacidade e velocidade de processamento de informações, a área de organização e arquitetura de computadores também exerce papel relevante nesse desenvolvimento. Além disso, a existência de componentes eletrônicos programáveis do tipo FPGA (*Field Programmable Gate Arrays*), permite a construção de componentes eletrônicos de alta integração que embasados em técnicas de síntese, projeto, implementação e validação de subsistemas digitais, permitindo ao Bacharel em Ciência da Computação atuar na área de desenvolvimento de *hardware* que satisfaz a necessidade do mercado na área em questão. O objetivo geral deste projeto é dominar as técnicas de síntese, projeto, implementação e validação de subsistemas digitais de média e alta integração para gerar um protótipo de um microcontrolador de 32 bits utilizando um FPGA com apoio de ferramentas de descrição de *hardware* como a linguagem HDL (*Hardware Description Language*) e conceitos de arquiteturas reconfiguráveis.

Palavras-chave: Arquitetura reconfigurável. CISC. FPGA. Microcontroladores. RISC. VHDL.

MENEZES, Denison; OSHIRO, Ernesto Koji. **Desenvolvimento de Microcontroladores Utilizando FPGA e VHDL**. 2009. 173 f. Monografia (Bacharelado em Ciência da Computação) – Centro Universitário Eurípedes de Marília, Fundação de Ensino "Eurípedes Soares da Rocha", Marília, 2009.

ABSTRACT

The area of development of digital integrated circuits has as one of the features to quickness development and research, enabling the construction of components each time faster. However, it isn't only the microelectronics technology that enables the construction of processors with high capacity and speed of information processing, the area of organization and computer architecture also plays an important role in this development. Furthermore, the existence of electronic components like programmable FPGA (Field Programmable Gate Arrays), allows the construction of electronic components, high integration grounded in synthesis techniques, design, implementation and validation of digital subsystems allowing the Bachelor of Computer Science to operate in the area of hardware development that satisfies the market need in the area in question. The goal of this work is to dominate the techniques of synthesis, design, implementation and validation of digital subsystems of medium and high integration, with the purpose of creating a prototype of a 32-bit microcontroller using a FPGA with support of hardware description tools as the HDL (Hardware Description Language) and concepts of reconfigurable architectures.

Keywords: Reconfigurable architecture. CISC. FPGA. Microcontrollers. RISC. VHDL.

LISTA DE ILUSTRAÇÕES

Figura 1 – Diagrama de blocos do PIC16F8X 23
Figura 2 – Microcontrolador PIC16F84 24
Figura 3 – Descrição dos pinos de entrada/saída do PIC16F8x 24
Figura 4 – Diagrama de blocos do 8051 31
Figura 5 – Microcontrolador Intel 8051 32
Figura 6 – Identificação dos pinos de entrada/saída do 8051 32
Figura 7 - Processo simplificado de execução de programas em processadores CISC 37
Figura 8 - Processo simplificado de execução de programas em processadores RISC 39
Figura 9 – Diagrama de blocos Spartan IIE 44
Figura 10 – Xilinx Spartan IIE 45
Figura 11 – Exemplo de uma descrição estrutural 52
Figura 12 – Hierarquia de tipos em VHDL 54
Figura 13 - Placa de Desenvolvimento D2SB 59
Figura 14 – Diagrama de Blocos da placa D2SB 60
Figura 15 - Placa de Entradas/Saídas DIO4 61
Figura 16 – Diagrama de Blocos da placa DIO4 62
Figura 17 - Fluxo de Projeto 66
Figura 18 – Organização interna 69
Figura 19 – 1º formato de instrução 71
Figura 20 – 2º formato de instrução 71
Figura 21 – 3º formato de instrução 71
Figura 22 – 4º formato de instrução 72
Figura 23 – 5º formato de instrução 72
Figura 24 – Diagrama de Blocos do Microcontrolador 84
Figura 25 – Organização interna da UCP 85
Figura 26 – Instrução contida no RC 85
Figura 27 – Representação da RAM 90
Figura 28 – Representação da ROM 91
Figura 29 – Representação do Multiplex_32_1 92
Figura 30 – Representação do Multiplex32_2_1 93
Figura 31 – Representação do Demultiplex_1_32 94
Figura 32 – Representação do Registrador32_c 95
Figura 33 – Representação do Registrador32_acesso_bit_c 96
Figura 34 – Representação do Registrador32_contador_c 97
Figura 35 – Representação do Timer_p 98
Figura 36 – Representação da Pilha_RISC 99

Figura 37 – Representação do RCCEI 100
Figura 38 – Representação do Flip_flop_d 101
Figura 39 – Representação da ULA_bit 105
Figura 40 – Representação da ULA_risc 106
Figura 41 – Representação dos estados da UC 109
Figura 42 – Primeiro contato com a ferramenta 132
Figura 43 – Criando novo projeto 133
Figura 44 – *Add items to the Project* 133
Figura 45 – *Add file to the Project* 133
Figura 46 – Compilando arquivos 134
Figura 47 – Abrir Simulação 135
Figura 48 – Iniciar Simulação 136
Figura 49 – Janela de simulação *Wave* 136
Figura 50 – *Wave Properties* 137
Figura 51 – Selecionar valor de entrada 137
Figura 52 – Definir *clock* 138
Figura 53 – Exemplo de simulação 139
Figura 54 – Finalizar simulação 139
Figura 55 – Primeiro contato com a ferramenta 140
Figura 56 – Criando novo projeto 141
Figura 57 – Configurando o Projeto 141
Figura 58 – *New Source* 142
Figura 59 – *Define VHDL Source* 143
Figura 60 – *New Project Information* 143
Figura 61 – *Add file to the Project* 144
Figura 62 – Definindo os pinos da FPGA 145
Figura 63 – Executando o *Synthesize* 146
Figura 64 – Programa para transferência de arquivos .Bit *iMPACT* 147
Figura 65 – Escrita Memória RAM 149
Figura 66 – Leitura Memória RAM 149
Figura 67 – Leitura Memória ROM 150
Figura 68 – Erro Memória RAM 151
Figura 69 – Erro Memória ROM 151
Figura 70 – Teste do *Demultiplex* 1x32 152
Figura 71 – Teste do *Demultiplex* 1x32 153
Figura 72 – Teste do *Multiplex* 32x1 154
Figura 73 – Teste do *Multiplex* 2x1 155

Figura 74 – Teste do *Multiplex* 32x1 155
Figura 75 – Flip Flop D.....156
Figura 76 – Escrita na Pilha 157
Figura 77 – Pilha Cheia.....157
Figura 78 – Pilha Vazia.....158
Figura 79 – RCCEI Escrita Modo Normal.....159
Figura 80 – RCCEI Modo Bit 159
Figura 81 – RCCEI Modo Set.....160
Figura 82 – RCCEI Modo Zerar 160
Figura 83 – RCCEI Modo Limpar 161
Figura 84 – Registrador32 Modo Acesso Bit com Modo Bit.....162
Figura 85 – Registrador32 Modo Limpar 162
Figura 86 – Registrador32 Modo Normal 163
Figura 87 – Registrador32 Contador.....163
Figura 88 – Registrador32 Leitura 164
Figura 89 – Timer Leitura 165
Figura 90 – Timer Limpar.....165
Figura 91 – Teste do módulo Somador 166
Figura 92 – Teste do módulo Subtrator.....167
Figura 93 – Teste do módulo Multiplicador.....167
Figura 94 – Multiplicação por número negativo 168
Figura 95 – Teste do módulo Divisor.....168
Figura 96 – Rotação a Esquerda.....169
Figura 97 - Rotação direita sem sinal.....169
Figura 98 - Deslocamento à Esquerda.....170
Figura 99 – Operação de negação 170
Figura 100 - Operação E 171
Figura 101 – Operando_1 maior 171
Figura 102 – Operação Igual.....172
Figura 103 – Montador implementado em JAVA.....173

LISTA DE QUADROS

Quadro 1 – Declaração de entidade....51
Quadro 2 – Declaração de uma arquitetura....52
Quadro 3 – Exemplo de declaração de objetos....54
Quadro 4 – Exemplo de atribuição de valores....54
Quadro 5 – Declaração de um componente....56
Quadro 6 – Declaração de uma biblioteca....57
Quadro 7 – Declaração de um pacote....58
Quadro 8 – Exemplo de código em *Assembly*....82
Quadro 9 – Código binário resultante....82
Quadro 10 – Código do seletor de programa....110
Quadro 11 – Código do programa multiplicador....111
Quadro 12 – Código do programa piscar....111
Quadro 13 – Código do programa Ler e Escrever memória....112
Quadro 14 – Procedimentos e funções auxiliares....112
Quadro 15 – Exemplo de atribuição de valores para sinal....121
Quadro 16 – Construção "*when else*"....121
Quadro 17 – Construção "*with select*"....122
Quadro 18 – Construção "*block*"....122
Quadro 19 – Construção "*process*"....123
Quadro 20 – Construção "*process*"....123
Quadro 21 – Comando *for*....124
Quadro 22 – Comando *if*....124
Quadro 23 – Comando seqüenciais....125
Quadro 24 – Comando *if else* e *elsif*....126
Quadro 25 – Comando *case when*....127
Quadro 26 – Comando *for loop*....127
Quadro 27 – Comando *wait*....128
Quadro 28 – Comando *null*....128
Quadro 29 – Declaração de Função....130
Quadro 30 – Declaração de Procedimento....131

LISTA DE ABREVIATURAS E SIGLAS[1]

ABINEE: Associação Brasileira da Indústria Elétrica e Eletrônica

ALADI: Associação Latino-Americana de Integração

AMD: Advanced Micro Devices

ASIC: Application-Specific Integrated Circuit

A/D: Analógico-Digital

CEITEC: Centro Nacional de Tecnologia Avançada S.A.

CI: Circuito Integrado

CISC: Complex Instruction Set Computer

CLB: Configurable Logic Block

CLRWDT: Clear Watchdog Timer

CPU: Central Processing Unit

DARPA: Defense Advanced Research Projects Agency

DRC: Design Rule Check

DSP: Digital Signal Processing

EEPROM: Electrically Erasable Programmable Read-Only Memory

EPIC: Explicitly Parallel Instruction Computing

EPROM: Erasable Programmable Read-Only Memory

FPGA: Field Programmable Gate Array

GPR: General Purpose Register

GPU: Graphics Processing Unit

HP: Hewlett-Packard

HDL: Hardware Description Language

IBM: International Business Machines

IEEE: Institute of Electrical and Electronics Engineers

ICSP: In-Circuit Serial Programming

I/O: Input/Output

IOB: In/Out Block

ISE: Integrated Software Environment

---

[1] O assunto do trabalho tem como característica a utilização de uma grande quantidade de abreviaturas (mnemônico), assim, foram relacionadas nesta lista as abreviaturas consideradas mais relevantes para a interpretação adequada do texto gerado.

JTAG: Joint Test Action Group

KB: KiloByte(1024 bytes)

LAS: Laboratório de Arquitetura de Sistemas

LED: Light-Emitting Diode

LIW- Long Instruction Word

LVS: Layout Versus Schematic

MBA: Master of Business Administration

MCT: Ministério da Ciência e Tecnologia

MHz: MegaHertz

MIPS: Microprocessor without Interlocked Pipeline Stages

MMX: MultiMedia Extensions

OPCODE: Operation Code

PA-RISC: Precision Architecture - Reduced Instruction Set Computer

PDIP: Plastic Dual Inline Package

PIC: Peripheral Interface Controller

PROTEC: Pró-Inovação Tecnológica

PS/2: Personal System/2

PSEN: Program Store Enable

PWM: Pulse Width Modulation

RAM: Random Access Memory

RC: Resistive-Capacitive

RCA: Radio Corporation of América

RISC: Reduced Instruction Set Computer

ROM: Read Only Memory

RST: Reset

RTL: Register Transfer Level

RTR: Run Time Reconfiguration

SB: Switch Box

SFR: Special Function Register

UC: Unidade de Controle

UCP: Unidade Central de Processamento

ULA: Unidade Lógica Aritmética

VAX: Virtual Address eXtension

VGA: Video Graphics Array

VHDL: VHSIC Hardware Description Language

VHSIC: Very High Speed Integrated Circuit

VLIW: Very Long Instruction Word

XTAL: Crystal

LISTA DE TABELAS

Tabela 1 - Balança Comercial do Setor Eletroeletrônico 16
Tabela 2 - Funções do PORT P3 30
Tabela 3 - Principais diferenças entre as arquiteturas RISC e CISC 40
Tabela 4 – Tipos de modos das portas de uma entidade 51
Tabela 5 – Descrição dos tipos de objetos 53
Tabela 6 – Tipos predefinidos no pacote padrão 55
Tabela 7 - Tipos compostos predefinidos no pacote padrão 56
Tabela 8 – Relação *Assembly Opcode* 80
Tabela 9 – Relação Sigla do registrador e código 81
Tabela 10 – Classes e operadores 118

SUMÁRIO

INTRODUÇÃO 16

CAPÍTULO 1 - MICROCONTROLADORES 19
1.1. Breve histórico 20
1.2. Organização Interna 20
1.3. Microcontrolador PIC16F84 21
1.4. Microcontrolador 8051 27

CAPÍTULO 2 - ARQUITETURAS 35
2.1. Arquitetura CISC 35
2.2. Arquitetura RISC 37
2.3. Comparativo entre RISC x CISC 39
2.4. Superescalar 42
2.5. Superpipelined 42

CAPÍTULO 3 - FPGA 43
3.1. Introdução 43
3.2. Histórico 43
3.3. Características 44
3.4. Roteamento 45
3.5. Reconfiguração 46
3.6. Vantagens do FPGAs 47

CAPÍTULO 4 - VHDL 49
4.1. Introdução 49
4.2. Histórico 49
4.3. Estrutura da linguagem VHDL 50

CAPÍTULO 5 - MATERIAIS E FERRAMENTAS 59
5.1. Placa de Desenvolvimento – Digilent D2SB 59
5.2. Placa de Entradas/Saídas Digitais - Digilent DIO4 61
5.3. Ambiente de Desenvolvimento Integrado ISE Xilinx 7.1i 62
5.4. Ambiente de Simulação ModelSim Xilinx Edition III 6.0a 64

CAPÍTULO 6 - METODOLOGIA DE DESENVOLVIMENTO 66
6.1. Especificação 67
6.2. Projeto Digital 67
6.3. Validação 68

CAPÍTULO 7 - ESTUDO DE CASO 69
7.1. Requisito da arquitetura proposta 69
7.2. Instruções 71
7.3. Relação *Assembly Opcode* 79

7.4. Interrupções 83
7.5. Definição da organização do microcontrolador, incluindo componentes internos 83
7.6. Implementação 90

CONCLUSÃO 113

REFERÊNCIAS 115

APÊNDICE A – Operadores, comandos seqüenciais e concorrentes da linguagem VHDL . 118

APÊNDICE B - Passo a passo de simulação utilizando o ModelSim 132

APÊNDICE C – Passo a passo da criação de projeto no ISE 140

APÊNDICE D – Testes e Validação 148

APÊNDICE E – Montador 173

INTRODUÇÃO

A área de desenvolvimento de circuitos digitais combinacionais e seqüenciais sofre rápida evolução, isso gera uma crescente demanda de profissionais na área de desenvolvimento de circuitos integrados, e para satisfazer essa necessidade de mercado faz-se necessário capacitar os egressos deste curso com o conhecimento do funcionamento e implementação de subsistemas digitais com média e alta integração.

> O MCT estima, por exemplo, que existam apenas 400 projetistas de circuitos integrados no país. A meta é formar 1.500 até 2010. O projetista é um profissional altamente qualificado. Além da formação em engenheira eletrônica ou num curso equivalente, ele é submetido a cursos específicos – e escassos – de especialização. O governo está estimulando a especialização por meio da criação de "*design-houses*", isto é, centros especializados na formulação (ou desenho) de projetos de CI (circuito integrado). (Protec, Brasil busca reduzir vácuo tecnológico em microeletrônica, 19/05/2008).

Segundo o gerente de economia da Abinee (Associação Brasileira da Indústria Elétrica e Eletrônica), Luiz Cezar Elias Rochel, em 2008, "o déficit comercial de produtos eletroeletrônicos foi de US$ 22,14 bilhões, 50% acima do ocorrido no ano anterior (US$ 14,75 bilhões)".

Na tabela 1 pode-se observar o resultado da balança comercial do setor eletroeletrônico, no período de Janeiro a Dezembro de 2008. A sigla Aladi significa Associação Latino-Americana de Integração (Ministério do Desenvolvimento, 2009).

Tabela 1 - Balança Comercial do Setor Eletroeletrônico

Janeiro-Dezembro 2008

Regiões	**US$ Milhões**		
	Exportação	**Importação**	**Saldo**
Estados Unidos	1.776,3	4.055,0	-2.278,7
Aladi (Total)	5.335,7	1.137,7	4.198,0
- Argentina	2.293,2	318,5	1.974,6
- Outros Aladi	3.042,5	819,1	2.223,4
União Européia	1.146,2	5.726,6	-4.580,4
Sudeste da Ásia (Total)	427,0	20.029,5	-19.602,4
- China	130,9	9.808,3	-9.677,4
- Outros Sudeste Ásia	296,1	10.221,1	-9.925,1
Demais Países do Mundo	1.205,6	1.084,1	121,5
Total	**9.890,8**	**32.032,9**	**-22.142,2**

Fonte: Abinee, 2009

Visando reduzir o déficit na balança comercial do setor eletroeletrônico, bem como a dependência de outros países para o desenvolvimento e produção de circuitos integrados, o governo federal criou, através do decreto Nº6638, de 07 de novembro de 2008 (BRASIL, 2008), a empresa pública CEITEC - Centro Nacional de Tecnologia Avançada S.A. Com investimentos iniciais de R$42.000.000,00 (quarenta e dois milhões de reais), a CEITEC tem como um de seus principais objetivos, inserir o Brasil no mercado global como produtor de semicondutores através da implantação de uma empresa competitiva em microeletrônica, atuando como líder e estimulador na formação e capacitação de mão-de-obra especializada e na modernização de outros setores industriais na América Latina (CEITEC, 2009).

Com o grande avanço ocorrido nas últimas décadas na área de circuitos eletrônicos programáveis e linguagens de descrição de *hardware*, surgiram, respectivamente, o FPGA e VHDL (VHSIC *Hardware Description Language*), o que tornou possível o desenvolvimento de sistemas digitais com alta integração de forma rápida e com custos extremamente baixos. O que permitiu aos projetistas implementar circuitos e arquiteturas cada vez mais complexas, sem a necessidade de uso de grandes recursos de fundição de silício (Ordonez, 2003).

A utilização de circuitos eletrônicos programáveis do tipo FPGA é uma alternativa superior aos circuitos ASIC (*Application-Specific Integrated Circuit*) que são programados de fábrica, pois o FPGA reduz os custos iniciais de investimento, elimina os ciclos de desenvolvimento longos, e o risco inerente dos ASICs convencionais. Além disso, o FPGA permite a sua atualização em campo sem a necessidade de substituição do *hardware*, algo impossível com ASICs (Xilinx, 2008).

O objetivo deste estudo é dominar as seguintes técnicas: síntese, projeto, implementação e validação de um subsistema digital de alta integração, em um componente FPGA utilizando linguagem VHDL. Para atingir este objetivo, foi realizada um estudo de dois microcontroladores distintos, bem como suas respectivas arquiteturas, com o propósito de selecionar as melhores características de cada uma das arquiteturas para a implementação de microcontrolador em FPGA utilizando a linguagem VHDL. Após a realização deste estudo optou-se por desenvolver um microcontrolador de uso geral de 32 bits com arquitetura RISC (*Reduced Instruction Set Computer*), utilizando a configuração proposta por Von Neumann, onde os dados e os programas são armazenados na mesma memória e compartilham o mesmo barramento de comunicação. Para isto foram utilizadas as seguintes ferramentas disponíveis no LAS (Laboratório de Arquitetura de Sistemas): o FPGA da fabricante Xilinx modelo Spartan IIE-200, o ambiente de programação da Xilinx ISE 7.1i (*Integrated Software Environment*), e a ferramenta de simulação ModelSim XE III 6.0a.

No capítulo 1 são apresentadas as arquiteturas dos microcontroladores 8051 e PIC (*Peripheral Interface Controller*), utilizados para estudos das arquiteturas CISC (*Complex Instruction Set Computer*) e RISC, foram úteis para definir as características na implementação do microcontrolador realizado neste trabalho.

O capítulo 2 aborda as arquiteturas CISC e RISC, apresentando um breve histórico e suas respectivas características e no final um comparativo entre as duas arquiteturas.

O capítulo 3 aborda o FPGA, onde é apresentado um breve histórico e as características do modelo utilizado no trabalho.

O capítulo 4 apresenta a linguagem VHDL através de um breve histórico da linguagem e sua estrutura.

No capítulo 5 são apresentados os materiais e ferramentas utilizados na codificação, síntese e implementação do microcontrolador.

O capítulo 6 apresenta a metodologia de desenvolvimento empregada na construção do microcontrolador.

No capítulo 7 é abordado o requisito da arquitetura, seguido da descrição do conjunto de instruções implementadas bem como uma tabela relacionado o *opcode* (o*peration code*) e a instrução *assembly*, em seguida é apresentado também uma definição da organização do microcontrolador incluindo seus componentes internos, seguido de como os módulos estão implementados, é apresentado também um pequeno programa em *assembly* como uma proposta de aplicação do microcontrolador para fins de validação do sistema, seguido da conclusão e referências bibliográficas.

CAPÍTULO 1 - MICROCONTROLADORES

Os microcontroladores possuem características de um sistema computacional completo em um único *chip*, e são encontrados, embutidos, em uma grande variedade de dispositivos. Eles possuem internamente processador, memória para armazenar dados do tipo RAM (*Random Access Memory*) e memória para armazenamento de programa do tipo EEPROM (*Electrically Erasable Programmable Read-Only Memory*), capacidade de entrada e saída, e de comunicação, *timer* gerador de sinais de interrupção, conversor A/D (Analógico-Digital), entre outras características específicas de cada fabricante (Ordonez, 2003).

Podem ser encontrados em versões de 4, 8, 16, e 32 bits, e podem ser de propósito geral ou propósito específico. Os de propósito gerais podem ser considerados processadores de menor porte, os de uso específico possuem características voltadas geralmente para controle de processo (Tanenbaum, 2007).

Na maioria dos casos o *software* está incorporado no *chip* em uma memória do tipo somente leitura (ROM - *Read Only Memory*) com conteúdo definido durante a fabricação do microcontrolador.

Um ponto de grande importância, nos microcontroladores, está relacionado ao custo, já que empresas que vão comprar milhares ou milhões deles podem optar por um e não pelo outro, motivados financeiramente por uma diferença de centavos no custo unitário. Outro ponto no desenvolvimento de um projeto de microcontrolador, além do preço final, é o seu consumo de energia, que depende da aplicação e da capacidade de processamento desejada. Esta relação de baixo custo e baixo consumo de energia, é o que permite que sejam utilizados em produtos com custo e desempenho competitivo que funcione a bateria inclusive.

Lembrando sempre que, as restrições impostas aos microcontroladores, dependem da aplicação na qual eles serão utilizados.

Nos subitens seguintes é apresentado um breve histórico dos microcontroladores, em seguida há uma descrição simplificada dos componentes comumente encontrados nos microcontroladores, e por último um estudo sobre dois dos principais microcontroladores utilizados no mercado, o PIC da Microchip e o 8051 da Intel. Estes microcontroladores apesar de apresentarem arquiteturas distintas e características próprias, podem ser utilizados em uma mesma aplicação.

1.1. Breve histórico

Sua história começa na década de 70, quando fabricantes, de eletrodomésticos e aparelhos em geral, começavam a incorporar em seus produtos o processador 8080 da Intel, porém para construir um sistema completo eles precisavam de memória externa e dispositivos de entrada e saída, o custo desses três *chips* mais as interconexões, tornava seu uso restrito a equipamentos de alto valor agregado. Por isto vários fabricantes pediram para Intel colocar um computador inteiro em um único *chip*, assim reduzindo seu custo. A Intel atendeu ao pedido, produzindo o *chip* 8748, que embora primitivo foi um sucesso de vendas, o que levou a Intel a lançar o 8051 em 1980, ele faz parte da família de microcontroladores da Intel denominada MCS-51, que são os mais populares dentre os microcontroladores de 8 bits.

1.2. Organização Interna

Unidade Central de Processamento

A Unidade Central de Processamento (UCP) desempenha a função de execução dos programas armazenados na memória principal buscando as instruções, examinando-as e executando-as uma após a outra. Ela é composta por várias partes distintas, sendo que as principais são:

- **Registradores e Contadores** – possuem a função de armazenamento de variáveis locais fundamentais e resultados intermediários de cálculos;
- **Unidade Lógica e Aritmética (ULA) –** circuito responsável pela execução das operações de cálculo como adição e subtração, e lógica como operações AND e OR, da UCP;
- **Unidade de Controle e Sincronização** – circuito responsável pelo gerenciamento dos demais circuitos tendo como lógica de base, microinstruções binárias, cuja função é o acionamento de cada bloco através de um vetor de bits de controle, sincronizando o funcionamento de todos os circuitos e retornando a solução do problema imposto pela instrução;

Unidade de Entrada e Saída

Responsável pela interface com o mundo exterior, projetada para permitir um controle sistemático da interação com o mundo exterior e fornecer a UCP as informações de que necessita para gerenciar a atividade de entrada/saída de maneira efetiva;

Memórias

Local onde dados e programas ficam armazenados para serem utilizados pela UCP, pode ser do tipo volátil ou não.

1.3. Microcontrolador PIC16F84

Este microcontrolador pertence à família PIC, lançado pela Microchip em meados de 1999, se popularizou rapidamente entre os estudantes e entusiastas em eletrônica devido ao seu baixo custo e a facilidade tanto de programação como de gravação (Souza, 2000). Atualmente o modelo PIC16F84 não é mais fabricado, em seu lugar a Microchip recomenda a utilização do PIC16F84A que é totalmente compatível ao seu antecessor, e com a vantagem que este modelo possuí o dobro da velocidade de *clock* (20MHz), o que permite um ciclo de instrução de 200ns.

Atualmente o uso deste modelo de microcontrolador é cada vez menor devido ao lançamento de novas famílias que possuem mais recursos, e compatíveis tanto em *hardware* quanto em *software* e em determinados casos com custo menor. Para efeito comparativo pode-se citar o PIC16F628 que possui o dobro de memória de programa, 224 bytes de RAM, 128 bytes de EEPROM, 3 *timers* e PWM (*Pulse Width Modulation*) e mais alguns outros recursos que o PIC16FF84 não possuí, mesmo assim o seu custo é menor (R$14,00 para o PIC16F628 e R$22,00 para o PIC16F84A – consulta realizada dia 10/10/09 em www.farnell.com.br).

1.3.1. Características do microcontrolador PIC16F84

O microcontrolador PIC16F84 é fabricado pela empresa Microchip e assim como o 8051 é considerado um marco na historia dos microcontroladores devido a algumas de suas características, únicas na época em que foi lançado, como o uso de arquitetura RISC e memória de dados e programas independentes, assim como os seus respectivos barramentos. As famílias de microcontroladores da Microchip que possuem a letra “F” em seus nomes indicam que a memória de programa do microcontrolador é do tipo Flash (EEPROM), onde as informações permanecem gravadas mesmo na ausência de energia elétrica. Este tipo de memória permite o usuário gravar a programação do microcontrolador diversas vezes, sem a necessidade de apagá-lo previamente com luz ultravioleta (memória EPROM - *Erasable*

Programmable Read-Only Memory). O processo era pouco prático, a memória necessitava ser totalmente apagada para depois ser reprogramada também inteiramente, demandando tempo e equipamentos auxiliares (dispositivo de componentes programáveis).

Este recurso juntamente com o sistema de gravação em série ICSP (*In-Circuit Serial Programming*) permite a gravação do programa no microcontrolador utilizando somente 2 pinos e sem que fosse necessário a sua retirada do circuito onde está aplicado, e graças ao ICSP a produção de gravadores e programadores de dispositivos tornou-se extremamente simples e barato.

Este microcontrolador foi escolhido para estudos porque além do fato de ser bastante popular, apresenta todas as características clássicas de um microcontrolador RISC, como conjunto reduzido de instruções, processamento *pipeline*, maior parte das instruções consumindo apenas um ciclo de máquina e poucos modos de endereçamento (direto, indireto e relativo) dentre outras.

A arquitetura Harvard empregada na construção dessa família de microcontroladores permite que o programa e os dados sejam acessados separadamente através de barramentos independentes próprios, no caso, o 16F84 possui um barramento de dados de 8 bits e o de instruções 14 bits. Essa independência dos barramentos permite que uma instrução seja executada enquanto outra é "buscada" na memória, agilizando o processamento. Outro detalhe importante é o fato de que o barramento de instruções é maior do que 8 bits, permitindo assim que o OPCODE da instrução já inclua o dado e o local onde irá operar (se necessário), o que economiza bastante memória de programa porque cada instrução vai ocupar somente uma posição de memória.

1.3.2. Descrição do PIC16F84

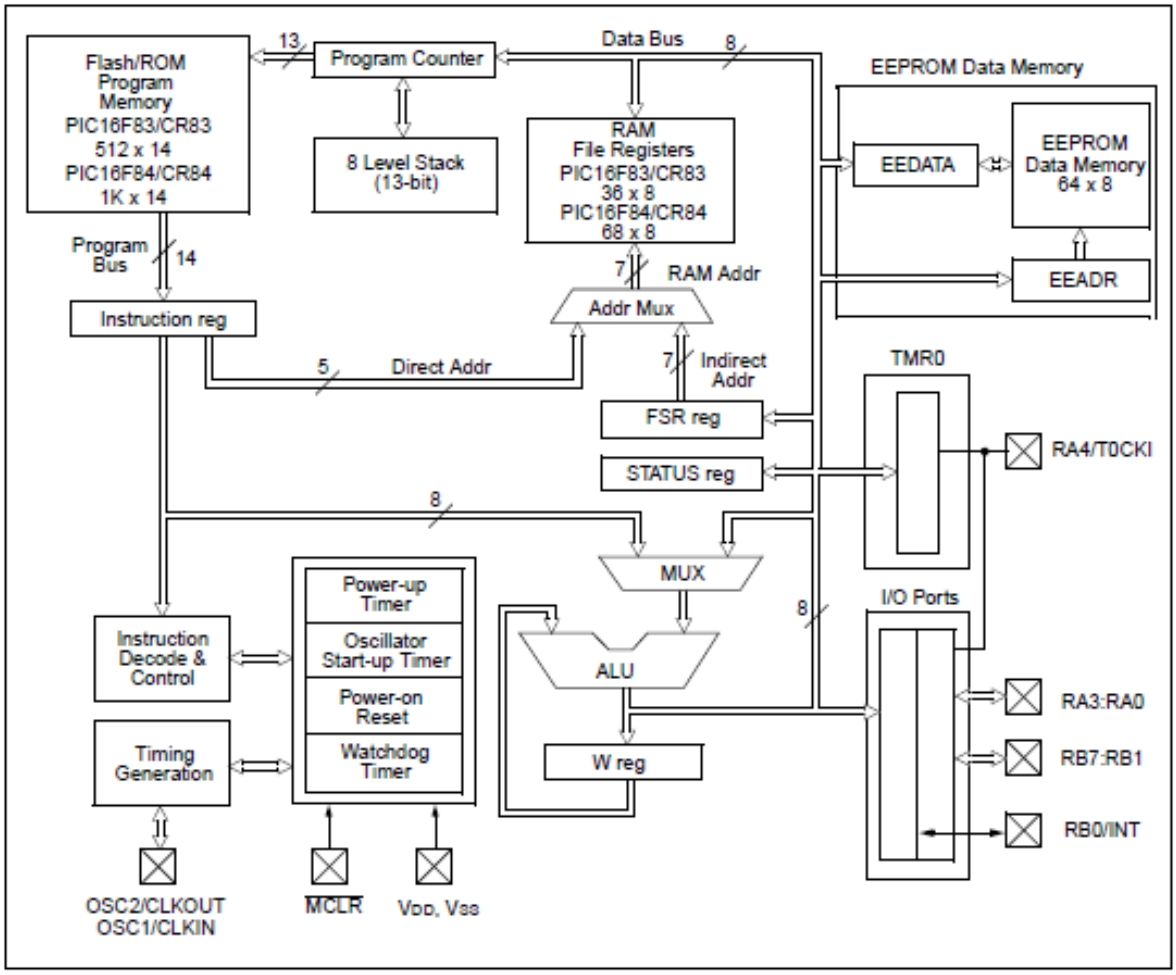
O PIC16F84 é um microcontrolador RISC, segue a arquitetura Harvard, como pode ser observado na figura 1, onde os barramentos de dados e programas estão separados, e possuí as seguintes características:

- Encapsulamento tipo PDIP (*Plastic Dual Inline Package*) de 18 pinos;
- 13 I/Os (*Input*/*Output*) configuráveis como entrada ou saída;
- *Pipeline* de 2 estágios;
- Velocidade de *clock* de até 10MHz (400ns por ciclo de instrução);
- 4 tipos de interrupções (Timer0, Externa, Mudança de Estado e EEPROM);

- Memória de programação EEPROM FLASH com 1024 bytes de palavras (*word*) de 14 bits cada;
- Pilha (*stack*) com capacidade de 8 níveis;
- 64 bytes de memória EEPROM;
- 68 bytes de memória RAM;
- Conjunto de 35 instruções com tamanho 14 bits cada;
- ICSP – *In-Circuit Serial Programming* que permite a gravação do *chip* através de 2 pinos;
- *Watchdog* Timer com circuito oscilador RC (*Resistive-Capacitive*) interno;

Através da figura 1 pode-se observar a organização interna do microcontrolador PIC16F8X, que está representado na forma de diagrama de blocos.

Figura 1 – Diagrama de blocos do PIC16F8X



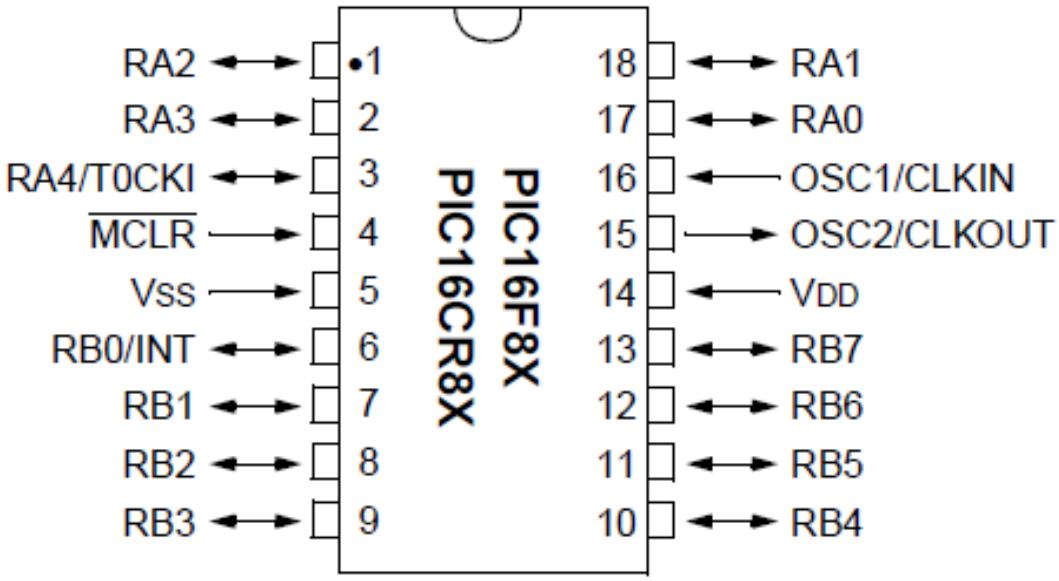
Fonte: Microchip, 2000

Na figura 2 pode-se observar uma imagem de um microcontrolador PIC16F84.

Figura 2 – Microcontrolador PIC16F84

Na figura 3 pode-se observar a disposição pinos de entrada/saída do microcontrolador PIC16F84.

Figura 3 – Descrição dos pinos de entrada/saída do PIC16F8x



Fonte: Microchip, 2000

O microcontrolador 16F84 possui em seu encapsulamento 18 pinos ao todo, sendo dois pinos destinados a alimentação do microcontrolador correspondentes ao Vdd (+5V) e Vss (gnd), dois pinos para ligação de oscilador externo, um pino para o recurso de *Master Clear Reset* e os treze pinos restantes são utilizadas como Entradas ou Saídas e estão divididos em dois conjuntos de portas distintas PORT A e PORT B.

O PORT A utiliza cinco pinos e recebem os nomes de RA0, RA1, RA2, RA3 e RA4, já o PORT B utiliza oito pinos e recebem os nomes de RB0, RB1, RB2, RB3, RB4, RB5, RB6 e RB7. Além das funções de entrada ou saída algumas destas portas possuem funções especiais, como por exemplo, a porta RA4 que pode ser configurado para incrementar o temporizador Timer 0 (TMR0); a porta RB0 pode ser utilizado para gerar interrupção externa e as portas RB4 a RB7 podem ser utilizadas para gerar interrupção por mudança de estado.

O *pipeline* de dois estágios possibilita que as instruções do 16F84 sejam executadas em apenas um ciclo de máquina, exceto as instruções que provoquem algum desvio no programa, neste caso são necessário dois ciclos de máquina para que sejam executadas.

A memória é dividida em dois blocos, sendo um bloco reservado a memória de programação e o outro para memória de dados. A memória de programação, onde ficam armazenadas as instruções do programa, possui capacidade de 1024 palavras (*words*) com 14 bits de extensão cada. Nessa memória estão localizadas duas posições especiais que recebem o nome de vetor de Reset e vetor de Interrupção. O vetor de Reset está localizado no endereço 000h e é para onde o programa vai sempre que o microcontrolador é inicializado, o vetor de interrupção está no endereço 004h e é para este endereço que o programa é desviado quando há ocorrência de alguma interrupção. Interrupções são eventos que podem ocorrer a qualquer momento (assíncrono) e que requerem algum tipo de atenção ou tratamento especial do programa, para que isso seja possível, assim que uma interrupção ocorra o fluxo do programa é imediatamente desviado para o endereço do vetor de interrupção onde é efetuado o tratamento adequado da interrupção ocorrida.

A memória de dados é o do tipo RAM, ou seja, volátil e é utilizada para armazenar temporariamente os dados utilizados pelo programa em execução. Está dividida em duas partes: registradores especiais (SFR – S*pecial Function Register*) e registradores de uso geral (GPR – *General Purpose Register*). O ponteiro de memória possui somente 7 bits de extensão, o que permite acessar até 128 posições de memória de cada vez, sendo assim, a memória de dados do PIC16F84 está dividida em dois bancos (*Bank* 0 e *Bank* 1). Para que o acesso a uma determinada posição dessa memória seja possível é necessário que o banco de memória esteja previamente selecionado através de um *bit* específico de controle localizado no registrador STATUS, que faz parte do SFR.

O registrador STATUS serve para indicar o estado atual da ULA (Unidade Lógica Aritmética), configurar a página de programação atual, se necessário, bem como a forma do último *reset*. O registrador OPTION serve para configurar uma série de opções de operação do microcontrolador, como ajustar o valor do PRESCALER e habilitar os Pull-Ups internos para o PORT B caso eles estejam configurados como entrada. O registrador INTCOM é utilizado para configurar e identificar as interrupções. Os registradores TRIS A e TRIS B servem para configurar os pinos das portas como entrada ou saída. Quando o valor “1” é colocado em um dos bits do registrador TRIS, o pino correspondente é configurado como entrada, caso o valor seja “0”, o pino é configurado como saída.

No 16F84 há um contador de uso geral de 8 bits, que recebe o nome de TMR0, e pode ser acessado diretamente na memória tanto para escrita como para leitura. Esse contador é incrementado automaticamente pelo *clock* ou por um sinal externo e toda vez que a ocorre um transbordo na contagem é gerado uma interrupção. Caso o TMR0 esteja sendo incrementado automaticamente, o seu transbordo vai ocorrer por volta dos 18ms, para aumentar este tempo basta configurar o PRESCALER.

O PRESCALER é um divisor de freqüência programável que pode ser utilizado para o TMR0 ou para WATCHDOG, ele divide o sinal que incrementa de um desses contadores. Os valores dessa divisão segue a potência de 2, sendo o valor para divisão máxima de 1:256 para o TMR0 e 1:128 para o WATCHDOG.

O *WATCHDOG*, cuja tradução literal seria "cão-de-guarda", é um recurso implementado em *hardware* cuja finalidade é manter o sistema funcionando no caso de falhas através da geração de um *Reset* no programa que está sendo executado caso este venha a falhar por alguma interferência externa (ruídos) ou por algum outro motivo. O *watchdog* é basicamente um contador que é incrementado por um sinal de relógio (*clock*) independente do resto do microcontrolador. Sempre que este contador atinge o seu valor máximo, ele reinicia a contagem a partir do zero e provoca uma reinicialização de todo o sistema. Para que isso não ocorra, dentro do programa deverá haver, no mínimo, uma chamada ao comando *CLEAR WATCHDOG TIMER* (CLRWDT) cuja função é zerar este contador evitando que chegue ao máximo e por conseqüência um *Reset* seja gerado. Recomenda-se cautela quanto ao uso indiscriminado desse comando, pois a sua má aplicação pode fazer com que o sistema fique travado diante de uma falha, um exemplo seria o programa entrar em um estado de *loop* infinito e dentro desse *loop* existir um comando *clear watchdog timer* que seria executado indefinidamente, causando assim o travamento do microcontrolador.

A linguagem de programação utilizada no 16F84 é composta de 35 instruções *Assembly* e podem ser classificadas em quatros grupos:

- Instruções que manipulam registradores;
- Instruções que manipulam constantes (literais);
- Instruções que manipulam bits;
- Instruções de controle;

Pode-se descrever de forma resumida as principais instruções utilizadas neste microcontrolador e seus respectivos exemplos de aplicação:

- ADD: Operação de soma. Ex: ADDWF FSR, 0

- AND: Operação booleana “E”. Ex: ANDWF FSR, 1
- CLR: Operação limpar ou zerar. Ex: CLRW
- COM: Operação complemento. Ex: COMF REG1, 0
- DEC: Operação decremento. Ex: DECF CNT, 1
- INC: Operação incremento. Ex: INC CNT, 1
- IOR: Operação booleana “OU”. Ex: IORWF RESULT, 0
- MOV: Operação mover, transferir. Ex: MOVF FSR, 0
- RL: Operação de rotação de 1 *bit* para a esquerda. Ex: RLF REG1, 0
- RR: Operação de rotação de 1 *bit* para a direita. Ex: RRF REG1, 0
- SUB: Operação de subtração. Ex: SUBWF REG1, 1
- SWAP: Operação de inversão entre partes do registrador. Ex: SWAPF REG, 0
- XOR: Operação “OU exclusivo”. Ex: XORWF REG 1

1.4. Microcontrolador 8051

Conforme já foi citado no item 1.1, o microcontrolador 8051 é o resultado de sucessivas melhorias aplicadas ao processador 8080 da Intel, que após agregar diversos recursos passou a ser denominado de microcontrolador sob o código 8748, que serviu de base para a construção do 8051. Este microcontrolador é integrante da família MCS-51 da Intel e graças ao seu sucesso se tornou em um dos microcontroladores mais populares no mundo.

1.4.1. Características do microcontrolador 8051

O termo MCS-51 se refere a uma família de microcontroladores criada pela Intel e é composta por diversos outros microcontroladores que segue a mesma arquitetura básica do 8051, porém com pequenas diferenças. Alguns exemplos dessas variantes são: o 8751 que possuí EPROM interna que podia ser programada pelo usuário, o 8031 possui EPROM externa e o 8052 possui um *timer* a mais.

A Intel não é a única fabricante deste *chip*, outras empresas como Philips, Siemens, AMD (*Advanced Micro Devices*) dentre outras, também fabricam este microcontrolador bem como as suas variantes (Nicolosi, 1998).

O 8051 é um microcontrolador de 8 bits e seu conjunto de instruções é composto por aproximadamente 250 comandos que podem ser classificados pelos seguintes tipos:

- Instruções aritméticas;
- Instruções lógicas;
- Instruções de transferência de dados;
- Instruções booleanas;
- Instruções de desvio;

Segundo Nicolosi (2005) os modos de endereçamento do 8051 podem ser classificados como:

- Registrador;
- Direto;
- Indireto;
- Imediato;
- Relativo;
- Absoluto;
- Longo;
- Indexado;

Com relação aos tipos de interrupções que podem ocorrer no 8051, pode-se classificar da seguinte forma:

- Interrupção externa INT0 – é um pino físico de interrupção, habilitado via *software*;
- Interrupção externa INT1 – semelhante ao INT0;
- Interrupção interna gerada pelo TIMER/COUNTER 0 – é um tipo de interrupção gerada internamente pelo TIMER0;
- Interrupção interna gerada pelo TIMER/COUNTER 1 – semelhante ao TIMER0;
- Interrupção pela serial – interrupção gerada pelo periférico serial;

Estas interrupções por sua vez podem possuir as seguintes propriedades:

- Vetorada ou não vetorada – quando ocorre uma interrupção, o fluxo do programa é desviado para o endereço de interrupção solicitada, se o desvio de interrupção possuir um endereço fixo então ela é chamada de interrupção não vetorada, caso o microcontrolador permite que, ao ser interrompido, o

dispositivo que o interrompeu também envie o endereço de desvio da interrupção, então ela é chamada de interrupção vetorada;

- Mascaramento – é a propriedade de o microcontrolador permitir ou não que determinado dispositivo o interrompa, seja por *software* ou por *hardware*;
- Prioridade – permite definir a prioridade para cada evento de interrupção, evitando possíveis conflitos caso estes venham ocorrer de forma concorrente;
- Origem – permite definir se a interrupção é interna ou externa ao *chip*;
- Tipo de disparo – é uma propriedade de interrupção externa ao *chip*, permite a interrupção do microcontrolador ocorra por nível (0 ou 1), borda de subida ou descida ou combinação de ambos;

1.4.2. Descrição do microcontrolador 8051

O microcontrolador 8051 possui as seguintes características:

- RAM interna de uso geral de 128 bytes e 128 bytes para registradores especiais;
- ROM interna de 4 Kbytes;
- 4 Ports de entrada/saída;
- 2 Timers de 16 bits;
- 1 Interface serial;
- Capacidade de endereçamento externo de ROM de até 64 Kbytes;
- Capacidade de endereçamento externo de RAM de até 64 Kbytes;
- Processador booleano (operação com bits);
- Ciclos de instrução típicos de 1 e 2 µs a 12 MHz;
- Instrução direta para divisão e multiplicação;
- Entradas de interrupção externa;

1.4.2.1. **Descrição do pinos do 8051**

No 8051 há 4 PORTS (P0, P1, P1 e P3), cada um com oito linhas, destinados a comunicação externa. P0 e P2 são destinados para o gerenciamento das vias dos dados e endereços de comunicação com as memórias ROM, RAM ou periféricos do tipo I/O mapeado.

P1 e P3 são destinados às vias de comunicação externa, sendo normalmente utilizadas como interface com o mundo externo.

PORT P0: Porta de 8 bits de propósito geral, se não for utilizado RAM nem ROM externa. Caso seja utilizada alguma memória externa esta porta é multiplexada, no tempo, entre dados e endereços (somente os endereços menos significativos), este artifício foi utilizado para diminuir o número de pinos no *chip*, tornando ele mais barato. Quando utilizado para dados e endereço ele funciona da seguinte forma: é utilizado um *chip latch*, que ligado ao pino ALE (*Address Latch Enable*) permite multiplexar externamente os dados dos endereços, ele funciona de forma transparente ao programador, ou seja, não é necessário se preocupar com este pino, pois ele é gerenciado pelo microcontrolador, no primeiro passo o valor do endereço contido em P0 é carregado no *latch*, depois é disponibilizado o P0 para dados.

PORT P1: Porta com 8 vias de propósito geral para I/O através de *software*.

PORT P2: Porta de propósito geral desde que não seja utilizado nenhuma memória externa seja ela RAM, ROM ou EPROM.

PORT P3: Porta de 8 bits de propósito geral para I/O se não for utilizado memória externa, interrupção externa, ou transmissão serial, pois nestes casos os pinos desta porta são utilizados conforme tabela 2.

Nome	Número Pino	Função Especial	Descrição
P3.0	10	RXD	Receptor de dados seriais
P3.1	11	TXD	Transmissor de dados seriais
P3.2	12	INT0	Interrupção externa 0
P3.3	13	INT1	Interrupção externa 1
P3.4	14	T0	Utilizado quando Timer 0 é um contador de eventos externos
P3.5	15	T1	Utilizado quando Timer 1 é um contador de eventos externos
P3.6	16	WR	Sinaliza que será realizada escrita na RAM (externa)
P3.7	17	RD	Sinaliza que será realizada leitura na RAM (externa)

Tabela 2 - Funções do PORT P3

PSEN (*Program Store Enable*): Ele aciona a ROM externa quando o microcontrolador vai realizar uma busca de código, para depois executá-lo. Ele também é acionado caso seja efetuada uma consulta a alguma tabela fixa, gravada na ROM. Ele é ativo em nível lógico 0, e é ativado automaticamente pelo microcontrolador.

ALE: Como já foi comentado na apresentação do P0, o ALE é um pino que comanda a demultiplexação de dados e endereço do port P0, sendo gerado automaticamente pelo microcontrolador.

EA (*External Access*): Pino de comando externo, determina qual memória ROM é utilizada, quando em nível lógico 0, leitura somente da ROM externa; em nível lógico 1, leitura da ROM interna passando automaticamente para a ROM externa, caso exista, após acabar o espaço da ROM interna.

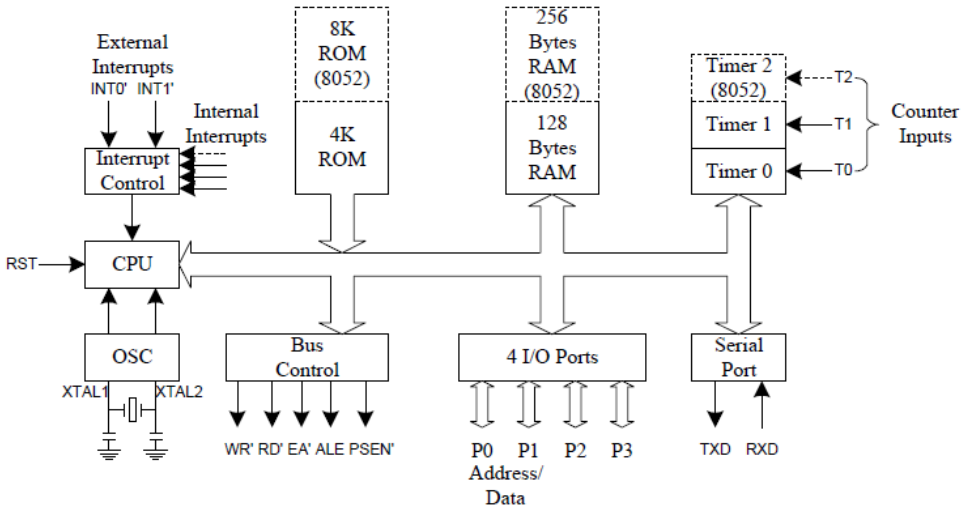
RST (*Reset*): Este pino "reseta" o microcontrolador, ou seja, organiza os valores internos do *chip*, para que possa ser iniciado o trabalho adequadamente e sempre da mesma forma. Para ser ativado é necessário ficar em nível 1 por, ao menos, dois ciclos de máquina.

XTAL1 e XTAL2(Cristal/Oscilador): Esse *chip* possui um sistema oscilador interno, que necessita de um cristal e dois capacitores externos, para gerar a oscilação (*clock*), que se torna o padrão de tempo para o microcontrolador trabalhar.

Vcc e Vss (Alimentação do *chip*): Vcc é o pino de alimentação do *chip* (+5 Volts), e o Vss é o pino terra.

Na figura 4 pode-se observar a organização interna de um 8051 básico, na forma de diagrama de blocos.

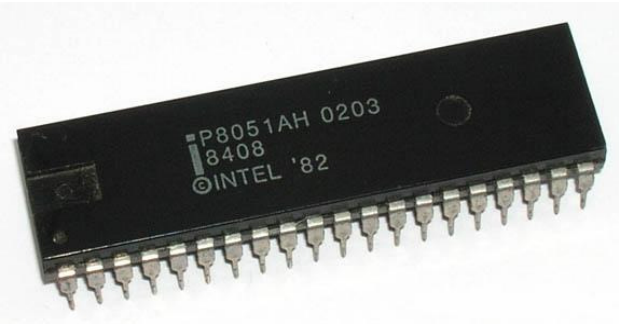
Figura 4 – Diagrama de blocos do 8051



Fonte: Nicolosi, 1998

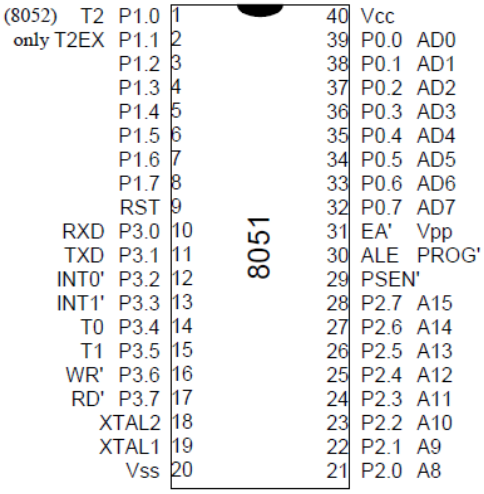
Na figura 5 pode-se observar a aparência física de um 8051.

Figura 5 – Microcontrolador Intel 8051



Na figura 6 pode-se observar a disposição de seus pinos de entrada/saída bem como as suas respectivas identificações.

Figura 6 – Identificação dos pinos de entrada/saída do 8051



Fonte: Nicolosi, 1998

1.4.2.2. **Organização da memória**

Neste microcontrolador há memória RAM interna, e pode gerenciar RAM externa. O mesmo acontecendo para ROM/EPROM.

O 8051 possui uma RAM interna dividida em registradores especiais e registradores de propósito geral. A capacidade máxima de endereçamento de RAM e ROM é de 64 kbytes cada, no caso da memória RAM ser externa, os pinos RD e WR serão utilizados para o controle de leitura e escrita.

- **RAM Interna**

É um espaço de 256 bytes e está dividida em:

- Posições com nomes para seus endereços, são divididas em quatro bancos cada uma com seus nomes indo de R0 até R7. Estas posições podem ser acessadas por seus nomes ou pela posição absoluta. Por exemplo, R0 no banco 0 tem endereço absoluto 00h. Este endereços vão do 00h até o 1Fh.
- Posições da RAM sem nomes, portanto acessível apenas pelo endereço absoluto, onde cada *bit* de uma posição de memória também tem seu endereço particular, ou seja, ela é *bit* endereçável assim como é byte endereçável. Vão da posição de byte 20h até 2Fh.
- Registradores de funções especiais: são aqueles que possuem nomes, além do seu endereço absoluto. Possuem uma função específica dentro do microcontrolador.

A memória está organizada da seguinte maneira: 256 bytes de RAM interna, dividida em 128 bytes de registradores de função especial, e 128 de registradores comuns. Os registradores de funções especiais são:

- ACC – Acumulador – é bastante utilizado em conjunto com outras instruções;
- P0, P1, P2 e P3 – registradores dos PORTS de comunicação;
- B – registrador utilizado nas instruções de multiplicação e divisão;
- PSW (*Program Status Word*) – registrador de estado da última operação lógica e aritmética realizado no acumulador;
- IP e IE (*Interrupt Priority* e *Interrupt Enable*) – registradores que permitem programar a parte de interrupções do microcontrolador;
- DPH e DPL (*Interrupt Priority* e *Interrupt Enable*) – estes dois registradores formam um registrador de 16 bits chamado DPTR (*Data Pointer*), permitindo a criação de um endereço de acesso às memórias externas;
- SP (*Stack Pointer*) – registrador utilizado como pilha de endereços de retorno de sub-rotinas;
- PCON (*Power Control Register*) – é utilizado para alterar os modos de funcionamento do *chip* com relação ao consumo de potência;

Os registradores que controlam os periféricos internos do microcontrolador são:

- SBUF (*Serial Buffer*) – é um registrador de comunicação com o periférico interno serial, responsável pelo *chip* de comunicação;
- SCON (*Serial Port Control Register*) – permite programar e controlar todas as atividades do periférico serial em conjunto com o SBUF;
- TH0, TL0, TH1 e TL1 (*Timer High* e *Timer Low*, 0 e 1) – são registradores de 8 bits que, em conjunto de dois (*high* e *low*), formam para cada *timer* (*Timer* 0 e *Timer* 1) seu registrador contador de até 16 bits;
- TCON (*Timer Control Register*) – registrador que controla as atividades dos timers;
- TMOD (*Timer Mode Register*) – permite programar os modos de atuação de cada *timer*;

CAPÍTULO 2 - ARQUITETURAS

2.1. Arquitetura CISC

O termo CISC (*Complex Instruction Set Computer*) computador com conjunto de instruções complexo, foi criado juntamente ao RISC (abordado no item 2.2) e faz referência aos computadores existentes até o surgimento dos RISC, e a tendência do aumento do número de instruções executado por seus processadores (Tanenbaum, 2007).

Para entendê-lo melhor é necessário observar alguns aspectos da história da evolução dos computadores e dos conceitos da microprogramação e do interpretador.

2.1.1. Microprogramação

Proposta em 1951 por Maurice Wilkes, pesquisador da Universidade de Cambridge, é um nível, a mais, implementado no *hardware*, com a finalidade de simplificá-lo. Neste nível estaria um interpretador para um conjunto de instruções imutáveis, o microprograma.

2.1.2. Interpretador

"Um interpretador subdivide as instruções da máquina em questão em pequenas etapas" (Tanenbaum, 2007, p. 31). Um processador construído com um interpretador tem o desenvolvimento mais simples, portanto mais rápido e barato, do que um processador de *hardware* para o mesmo computador proposto. No entanto, um processador com um nível a mais, no caso o de interpretação, tende a ser mais lento do que um sem este nível.

2.1.3. Histórico

No final da década de 1950, quando a IBM (*International Business Machines*) era líder do mercado de computadores, foi constatado que prestar suporte a uma única família de computadores, portanto que executam as mesmas instruções, contem muitas vantagens, tanto para a empresa quanto para seus clientes, neste contexto que foi criado o termo de arquitetura

para descrever tal nível de compatibilidade. Assim uma família de computadores teria a mesma arquitetura, mas cada modelo desta família teria uma organização diferente com custo e desempenho distintos. Para tornar possível a execução de programa com instruções complexas, executadas por computadores de alto desempenho, em computadores de baixo custo foi empregada a interpretação sugerida por Wilkes em 1951. O resultado foi a família IBM System/360, em que os computadores mais baratos eram interpretados, enquanto os mais caros e mais velozes usavam uma implementação de *hardware* direto.

Com o avanço do mercado de computadores na década de 1970, a capacidade de computação crescia rapidamente, junto com a demanda por computadores de baixo custo, que em conjunto com as memórias rápidas somente leitura (ROM), denominadas memória de controle, favorecia projetos de computadores que utilizassem interpretadores ao ponto de se tornar o modo padrão de se projetar computadores, naquela época.

Os projetistas notaram que poderiam adicionar novas funcionalidades apenas ampliando o microprograma, isto fez com que varias instruções fossem criadas, algumas até desnecessárias, pois poderiam ser obtidas através das já existentes, mas eram implementadas por executar mais rapidamente deste modo.

Esta tendência chegou ao seu ponto mais alto com o computador VAX (*Virtual Address eXtension*) da Digital Equipment Corporation, com varias centenas de instruções, que desde o inicio foi concebida para ser implementada com um interpretador, sem dar atenção a uma implementação de alto desempenho. O que resultou em uma arquitetura com muitas instruções que dificilmente seria executada diretamente por *hardware*, o que tornou esta arquitetura obsoleta no molde atual de arquiteturas, ao ponto de ser descontinuada.

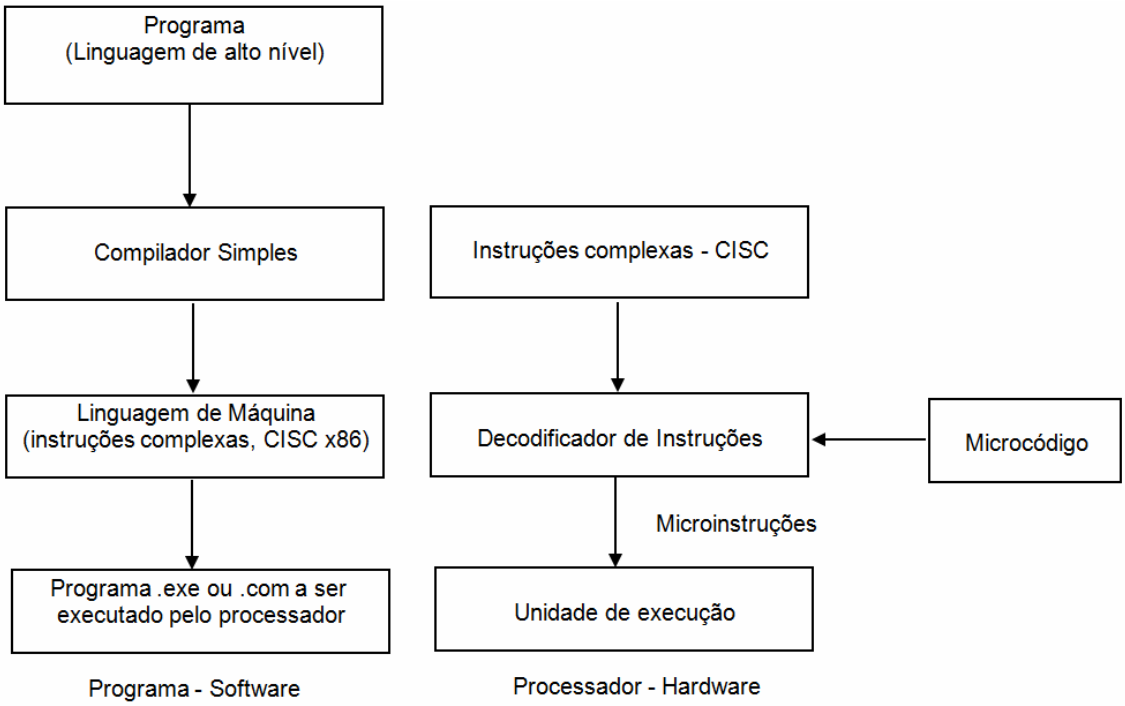
2.1.4. Características

Suas principais características são:

- Grande conjunto de instruções
- Uso de interpretador
- Várias instruções e modos de acessar a memória
- Pouco uso de *pipeline* e uso de paralelismo de instruções
- Vários formatos de instruções

A execução de um programa em um processador CISC é demonstrada de forma sintética na figura 7.

Figura 7 - Processo simplificado de execução de programas em processadores CISC



Fonte: TORRES, 2001

2.2. Arquitetura RISC

O termo RISC (*Reduced Instruction Set Computer*), é um acrônimo para computador com conjunto de instruções reduzido, e foi introduzido por um grupo de Berkeley, liderado por David Patterson e Carlo Séquin, em 1980, quando eles projetaram uma CPU (*Central Processing Unit*) que não utilizava interpretação e possuía um conjunto de instruções reduzido, com o nome de RISC I (Patterson, 1985; Patterson e Séquin, 1982). O conjunto de instruções simples de uma máquina com arquitetura RISC favorece o uso eficiente de *pipelines*, pois o número de operações por instrução é menor e mais previsível se comparado com uma máquina com arquitetura CISC. A arquitetura com um conjunto reduzido de instruções favorece também o uso da técnica de atraso de instruções de desvio (*delayed branch*), na qual instruções de desvio são trocadas de posição com outras instruções para aumentar a eficiência de uso da *pipeline* (Stallings, 2002).

2.2.1. Histórico

Como já mencionado anteriormente, no final da década de 1970 os processadores possuíam um grande conjunto de instruções, com muitas de grande complexidade, e utilizavam interpretadores, os projetistas não pensavam em projetos de máquinas mais simples, porém um grupo da IBM se opôs a esta tendência, o que resultou no minicomputador denominado 801, que embora não tenha sido lançado comercialmente inspirou outras pesquisas no mesmo sentido como a do grupo de Berkeley, e a do *chip* MIPS (*Microprocessor without Interlocked Pipeline Stages*) projetado por John Hennessy em Stanford.

Este ramo de pesquisa foi tomado devido a alguns fatores, pode-se citar, por exemplo: O fato dos programas implementados, para as máquinas existentes, não utilizarem muitas das instruções disponíveis, o que tornava nulo o ganho de desempenho obtido com estas instruções. O aumento da velocidade da memória principal, que tornava a penalidade imposta pelo uso da interpretação muito maior.

Como os projetos de computadores RISC não eram compatíveis com os existentes no mercado da época, os projetistas tiveram liberdade para escolher conjuntos de instruções que aumentasse o desempenho total do sistema, e também foi observado que mais importante do que ter instruções que fossem executadas rapidamente, era ter instruções que pudessem ser iniciadas mesmo sem ter terminado de executar as instruções anteriores (processamento *pipeline*), mas o que chamou mais a atenção foi o baixo numero de instruções que rodava a casa dos 50, que era bem menor que os 100 a 200 encontradas nos computadores existentes.

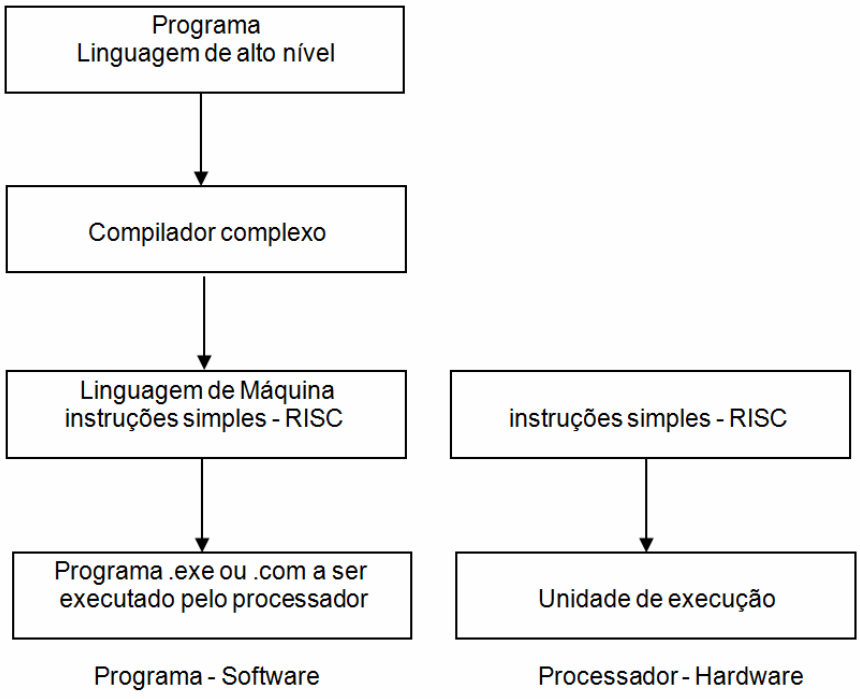
2.2.2. Características

As principais características da arquitetura RISC são:

- Conjunto reduzido, e simples de instruções;
- Não utiliza interpretador;
- Acesso à memória apenas por instruções específicas, e de modo simples;
- Elevado uso de *pipeline*;
- Formato regular, e simples de instruções;
- Operações realizadas apenas com registradores;
- Uma operação por ciclo de máquina;

A figura 8 descreve, de forma sintética, o fluxo de execução de um programa em um processador RISC.

Figura 8 - Processo simplificado de execução de programas em processadores RISC



Fonte: TORRES, 2001

2.3. Comparativo entre RISC x CISC

Antes o tamanho do conjunto de instruções era um fator determinante para diferenciar as duas arquiteturas, atualmente o uso dessa característica deixou de ser relevante quando o assunto é a comparação das duas arquiteturas. Um exemplo disso é o processador RISC PowerPC 601, utilizado nos primeiros computadores Apple Macintosh, ele possuí um conjunto de 272 instruções (Appendix G, 2009), contra aproximadamente 260 instruções (Intel, 1999) do Pentium MMX (*MultiMedia Extensions*) que segue a arquitetura CISC e foi lançando alguns anos após o PowerPC 601. Pode-se notar que o tamanho do conjunto de instruções de ambos são semelhantes, porém, a quantidade de instruções do PowerPC 601, foge da regra básica da arquitetura RISC, que era propor uma arquitetura com um conjunto reduzido de instruções.

Outro detalhe que também chama a atenção é fato dos processadores Pentium possuírem dois canais de execução de dados (*pipeline*) que lhe permitia a execução de mais de uma instrução por ciclo de *clock*, essa arquitetura recebeu o nome de Superescalar e era

dividida em um canal U que processava qualquer tipo de instrução e um canal V que processava somente as instruções mais simples.

Observa-se que os fabricantes de processadores não estão preocupados se um determinado recurso que está sendo implementado é inerente a uma arquitetura diferente, desde que promova algum ganho de desempenho no final.

Na tabela 3, pode-se comparar as principais características de cada uma das arquiteturas. Atualmente está cada vez mais difícil associar essas características como sendo únicas e exclusivas a uma arquitetura específica. A constante busca pelo desempenho cada vez maior na execução das instruções fez com que os fabricantes de processadores, sejam eles RISC ou CISC, utilizassem todos os recursos disponíveis para melhorar o desempenho e de alguma forma se sobressair em relação aos seus concorrentes.

Características das Arquiteturas	
Arquitetura RISC	Arquitetura CISC
Instruções simples de um ciclo	Instruções simples de vários ciclos
Somente *Load*/*Store* referenciam a memória	Qualquer instrução pode referenciar a memória
Elevado uso de *pipeline*	Ausência ou pouco uso de *pipeline*
Instruções executadas pelo *hardware*	Instruções interpretadas por microprograma
Instruções com formato fixo	Instruções com diversos formatos
Poucas instruções e modos endereçamento	Muitas instruções e modos de endereçamento
Complexidade no compilador	Complexidade no microprograma
Múltiplos conjuntos de registradores	Conjunto reduzido de registradores

Tabela 3 - Principais diferenças entre as arquiteturas RISC e CISC

Um exemplo dessa busca pelo desempenho é o processador Intel Merced (Stallings, 2002), que posteriormente ganhou o nome comercial de Itanium, esse processador foi desenvolvido pela Intel em conjunto com a HP (*Hewlett-Packard*), ambas desenvolveram a arquitetura IA-64 que era uma arquitetura 64 bits pura, ou seja, não era uma extensão da arquitetura atual de 32 bits x86 da Intel, muito menos uma adaptação da arquitetura PA-RISC (*Precision Architecture - Reduced Instruction Set Computer*) de 64 bits da HP. Essa nova arquitetura foi baseada em anos de pesquisa desenvolvida pelas duas companhias e de algumas universidades. O foco dessa arquitetura era a exploração do vasto conjunto de circuitos de alta velocidade disponíveis nas próximas gerações de circuitos integrados com o uso sistemático de paralelismo.

As principais características da arquitetura IA-64 são:

- Paralelismo de nível de instrução em tempo de execução explícito nas instruções de máquina, ao invés de ser determinado pelo processador em tempo de execução;
- Palavras de instrução longas ou muito longas (LIW- *Long Instruction Word* /VLIW – *Very Long Instruction Word*)
- Predicação de desvio;
- Carga especulativa;

A esse conjunto de conceitos deu-se o nome de EPIC (*Explicitly Parallel Instruction Computing*) computação de instrução explicitamente paralela, esse termo é usado pela Intel e HP para se referir à tecnologia ou a coleção de técnicas.

Apesar dos esforços das duas empresas em construir um processador revolucionário de 64 bits para o mercado corporativo, o Itanium não fez muito sucesso após o seu lançamento, que até então havia sido adiado por diversas vezes. Um dos fatores desse fracasso foi a sua incompatibilidade com a geração x86 atual de processadores. Para resolver essa incompatibilidade a Intel implementou um sistema de emulação por *hardware* o que permitia rodar os softwares existentes escritos para x86, porém, o seu desempenho era muito fraco. Outro fator que contribuiu para o fracasso do Itanium foi a falta de softwares específicos para este processador.

Percebendo essa "falha" da Intel, alguns anos depois a empresa AMD lança o seu processador K8, comercialmente conhecido por Opteron. Este processador, de 64 bits, voltado para o mercado corporativo (servidores) possuía a vantagem de ser totalmente compatível com as arquiteturas anteriores de 32 bits, o que garantiu a sua ampla aceitação no mercado.

Em 2006 a AMD adquiriu a companhia ATI, que junto com a nVidia dominavam o mercado de placas de vídeo. Após adquirir a ATI, a AMD agregou conhecimento e tecnologia das GPUs (*Graphics Processing Unit*) fabricadas pela ATI o que fortaleceu a sua atuação nessa área.

Atualmente o maior problema das arquiteturas *multicore* está sendo a falta de softwares que aproveitem melhor a quantidade de *core* disponíveis nos novos processadores. Para a AMD, a solução para esse problema se chama FUSION (AMD, 2008), um processador idealizado pela AMD, cujo codinome inicial era SWIFT (AMD, 2007), que ao invés de usar vários núcleos de propósito gerais, utilizaria núcleos de propósito específicos como GPU, *Chipset* e a CPU propriamente dita. A intenção da AMD é produzir este processador *multicore*, onde cada núcleo desempenharia somente uma função específica, como processar vídeo, gerenciar comunicações, manipular som, entre outros, eliminando assim os periféricos,

e tudo isso controlado por um único processador de propósito geral que estaria encapsulado junto aos demais processadores de função específica. Ao integrar todos estes processadores, eliminaria-se as dependências dos barramentos externos utilizados na comunicação dos vários processadores periféricos e com um processador de propósito geral gerenciando as instruções e distribuindo-as a cada processador específico, espera-se que o aumento no desempenho final seja enorme. Outra vantagem proporcionada por essa arquitetura seria a possibilidade de utilização dos softwares atuais sem a necessidade de reescrevê-los para esta nova arquitetura.

A estréia do FUSION estava previsto para este ano, 2009, porém a AMD adiou o seu lançamento e não definiu uma nova data.

2.4. Superescalar

O termo superescalar refere-se a máquinas projetadas para melhorar o desempenho da execução de instruções escalares. A implementação superescalar possibilita que instruções usuais possam ser iniciadas simultaneamente e executadas independentemente.

Esta arquitetura permite que mais de um *pipeline* seja executado ao mesmo tempo, ou seja, o ciclo de busca, decodificação, execução e armazenamento de mais de uma instrução no mesmo ciclo de execução da máquina (Stallings, 2002).

2.5. Superpipelined

O *superpipelined* é uma extensão do comprimento do *pipeline*, diminuindo assim a sua granulação e aumentando o número instruções executadas por ciclo, porém, com o aumento da quantidade de instruções se encontrarem simultaneamente dentro do *pipeline*, problemas com interdependência entre as instruções e oscilações de dados (atrasos cumulativos devido a erros no cachê, largura de faixa no bus e outro problemas de acesso a memória), aumentam o risco de travar o *pipeline*. Estes problemas podem ocorrer, em teoria, com qualquer *pipeline*, porém a probabilidade de ocorrer com o *superpipelined* é maior.

CAPÍTULO 3 - FPGA

3.1. Introdução

Os primeiros FPGAs foram introduzidos pela empresa Xilinx em 1984. Estes componentes consistiam de uma matriz 2-D de células lógicas simples que podiam ser conectadas arbitrariamente através de uma rede de interconexão.

Um FPGA é formado por um grande numero de arranjos de células configuráveis (ou bloco lógicos), contidos em um único *chip*, conforme pode ser observado na figura 9. Cada uma dessas células contém certa capacidade computacional para implementar funções lógicas, e/ou realizar roteamento para permitir a comunicação entre células. Essas operações podem acontecer simultaneamente no arranjo das células (Ribeiro, 2002).

Segundo Ordonez (2003), os FPGAs são circuitos programáveis formados por conjuntos de células lógicas em arranjo matricial. Cada célula contém capacidade computacional para implementar funções lógicas e realizar roteamento para comunicação entre elas. O roteamento e as funções lógicas das células são configuráveis via *software*.

3.2. Histórico

A Xilinx (Xilinx, 2009) foi fundada em 1984 por dois brilhantes engenheiros e um "guru" de marketing que até então trabalhavam no Vale do Silício e em diferentes empresas. Bernie Vonderschmitt, Ross Freeman e Jim Barnett sonhavam em criar uma empresa "diferente". Eles desejavam criar uma companhia que pudesse desenvolver e fabricar o estado-da-arte em tecnologia em um campo totalmente novo.

A tecnologia que impulsionou a Xilinx para a sua aparição foi considerada um conceito fora do comum em 1984. Idealizado pelo co-fundador da Xilinx, Ross Freeman, o novo dispositivo semicondutor, hoje conhecido como FPGA, foi uma forma completamente nova de programação lógica. Estes *chips* podiam ser personalizados pelos usuários para realizarem várias funções através de programação com a ajuda de um *software*.

Em meados de 1988, Ross Freeman registrou a patente número 4,870,302 que se referia a um *chip* de computador repleto de "portas abertas" que engenheiros poderiam

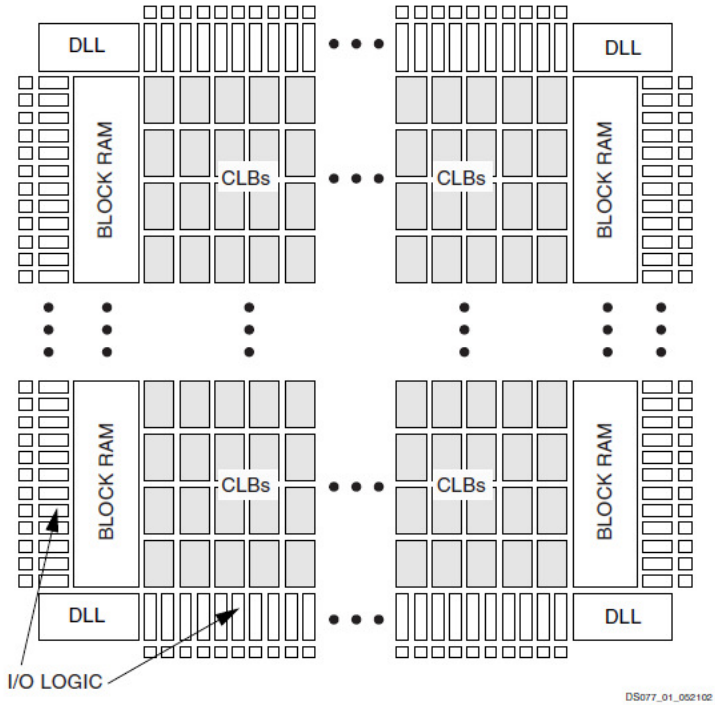
reprogramar tanto quanto necessário para adicionar novas funcionalidades, se adaptarem aos padrões ou especificações e fazer alterações no último minuto.

Bernie Vonderschmitt um engenheiro eletricista com graduação em MBA (*Master of Business Administration*) apareceu com modelo poderoso de negócio para sua nova empresa. Quando ele trabalhou como gerente geral da Divisão de Componentes de Estado Sólido da RCA (*Radio Corporation of América*) ficou convencido de que fábricas de semicondutores (fabs) eram algo caro e oneroso. Ele jurou que: "Se algum dia eu montar uma companhia de semicondutores, ela será fabless". "Nós vamos encontrar fornecedores que possam produzir para nós". Desde então a idéia tornou-se tão atraente e popular que atualmente existem 700 fabless de semicondutores ao redor do mundo.

3.3. Características

Um circuito reconfigurável do tipo FPGA inicialmente não possui nenhuma funcionalidade especifica, há a necessidade de uma programação inicial para que o circuito possa efetuar determinadas funções. Essa programação é realizada carregando um arquivo binário (*bitstream*) em um arranjo de memórias estáticas cujo conteúdo define o estado do elemento reconfigurável do FPGA ou parte dele.

Figura 9 – Diagrama de blocos Spartan IIE



Fonte – Xilinx, 2008

As FPGAs consistem de um elevado número de células lógicas simples, que podem ser conectadas ao se carregar um programa de configuração no dispositivo. Uma FPGA pode ser reconfigurada em questões de milisegundos, com um novo *hardware* e com aplicação muito diferente da configuração anterior. A estrutura geral de um FPGA pode ser vista na figura 9.

As principais partes que compõem um FPGA são:

- CLB (*Configurable Logic Block*): são blocos de unidade lógica configurável que estão dispostos em um arranjo 2-D, em forma de uma matriz;
- IOB (*In/Out Block*): unidade de entrada e saída;
- SB (*Switch Box*): unidade de conexão entre os diversos CLBs;
- Canais de roteamento: interligam as unidades de conexão para formar a rede de interconexão programável;

Pode-se observar na Figura 10 a imagem de um FPGA típico da Xilinx correspondente a família Spartan.

Figura 10 – Xilinx Spartan IIE

3.4. Roteamento

Roteamento é o processo de configuração das conexões internas de uma FPGA, no intuito de se obter o comportamento desejado, para uma determinada aplicação, utilizando os blocos lógicos. Segundo Ordonez (2003, p. 17) “Roteamento é a interconexão entre os blocos lógicos através de uma rede de camada de metal”, estas interconexões podem ser executadas através de transistores, controladas por bits de memória, ou através de chaves de interconexão (*switch matrix*).

3.5. Reconfiguração

Os sistemas reconfiguráveis são plataformas de *hardware* onde sua arquitetura pode ser modificada via *software*, em tempo de execução ou não, para melhor se adequar a uma determinada aplicação em um instante particular (Cardim, 2005).

Como já foi abordado no item 3.3, os circuitos são configurados de forma que a seqüência de bits responsáveis pela sua configuração é buscada na memória e usada diretamente na configuração do *hardware*, não sendo necessária uma fase de interpretação (Hauck, 2000).

Os métodos de reconfiguração de um dispositivo programável e reconfigurável são:

- Reconfiguração Total: o dispositivo reconfigurável é inteiramente alterado;
- Reconfiguração Parcial: somente uma parte do dispositivo é reconfigurada, pode ser disruptiva, onde a reconfiguração parcial afeta outras partes do sistema, forçando uma parada no funcionamento do sistema, ou não-disruptiva, onde as partes do sistema que não estão sendo reconfiguradas permanecem completamente funcionais durante o ciclo de reconfiguração;
- Reconfiguração Dinâmica: nesse tipo de reconfiguração não há necessidade de reiniciar o circuito ou remover elementos reconfiguráveis para programação;
- Reconfiguração Extrínseca: reconfigura parcialmente o sistema, mas somente considerado cada FPGA que o compõe como unidade atômica de reconfiguração;
- Reconfiguração Intrínseca: reconfigura parcialmente cada FPGA que compõe o sistema;

3.5.1. Computação Reconfigurável

As primeiras FPGAs demoravam vários segundos ou mais para que fossem completamente reconfiguradas. Atualmente, os novos modelos de FPGAs permitem ser reconfiguradas, em tempo de execução, de forma que o *hardware* se adapte as mudanças no sinal de entrada ou em resposta a sinais externos. Este processo de reconfiguração em tempo de execução é conhecido como RTR (*Run Time Reconfiguration*). Estas FPGAs podem ser reconfiguradas total ou parcialmente e são também chamadas de dispositivos dinamicamente reconfiguráveis.

Uma arquitetura reconfigurável processa as informações através de unidades configuradas no espaço por meio de diferentes blocos lógicos dentro do FPGAs, enquanto que um computador de uso geral processa as informações através de um conjunto de instruções no tempo.

Este método de configuração distingue os FPGAs dos processadores convencionais, pelo fato de não possuírem nenhuma programação interna, enquanto que os processadores são configurados durante o processo de fabricação.

Em uma FPGA a reconfiguração é implementada através da configuração dos CLBs e pela interligação dos elementos que formam a matriz de conexão. Essa característica permite, por exemplo, que uma determinada aplicação que envolva códigos complexos e extensos que necessitem ser carregados simultaneamente pelo hardware, possa ser implementada de forma que o *hardware* assuma configurações distintas solucionando partes da aplicação até o resultado final.

Existem três modelos de reconfiguração:

- Contexto Simples – o FPGA é configurado totalmente quando uma alteração é implementada;
- Contexto Múltiplo – várias configurações pré-realizadas podem ser chaveadas a qualquer momento constituindo vários contextos simples multiplexados;
- Reconfiguração Parcial – em determinados casos a configuração não ocupa todo o *hardware* disponível, nesses casos, a reconfiguração parcial permite a seleção de dados a serem alterados, de forma que uma parte dos dados originais não seja alterada, funcionando normalmente;

3.6. Vantagens do FPGAs

FPGAs podem ser utilizados em uma infinidade de aplicações, dentre elas uma que merece destaque é o desenvolvimento de protótipos de circuitos, uma vez que as características de reprogramação de um FPGA possibilita uma significativa redução no tempo de desenvolvimento através da agilização dos processos de simulação, teste, depuração e alteração do projeto.

Para este tipo de aplicação, o comportamento do FPGA pode ser definido por meio da programação em *schematic*, utilizando uma biblioteca para gerar o diagrama elétrico do

circuito ou uma linguagem HDL. Desta forma é gerado um arquivo de configuração, que ao ser carregado no FPGA, faz com que o mesmo se comporte como o circuito projetado.

Quando implementamos algoritmos em hardware onde ocorre a manipulação direta de bits, como por exemplo, compressão de dados, reconhecimento de padrões, criptografia, tratamento de sinais, dentre outros, há uma melhora significativa no desempenho, pois um FPGA é particularmente eficiente em aplicações orientadas a *bit*.

CAPÍTULO 4 - VHDL

4.1. Introdução

VHDL (*Very high speed integrated circuit Hardware Description Language*) é uma linguagem de descrição de *hardware*, que independe da tecnologia e metodologia utilizada para descrever o sistema, ao invés disto é utilizado uma abstração que inclui comportamento, temporização e características estruturais de sistemas digitais. Estas características permitem atuações em uma ampla faixa de opções para descrições de esquematização (Mourelle; et all, 2001).

VHDL é uma linguagem padronizada, segundo d'Amore (2005, p. 1) VHDL é "Uma linguagem de descrição de circuito padronizada que possibilita o intercâmbio de informações referentes ao comportamento de um circuito entre fabricantes, fornecedores de sistemas e empresas de projetos", evitando que os envolvidos no projeto criem seus próprios padrões, e reduz a incidência de má interpretação entre as partes envolvidas no projeto. A utilização de uma HDL (*Hardware Description Language*), também substitui o processo de desenvolvimento de circuito com captura de esquemático, o que é uma grande vantagem, já que tal técnica se torna muito onerosa em circuitos complexos.

Nos item 4.2 é abordada a história da linguagem, no item 4.3, é apresentada a estrutura da linguagem, contendo entidade de projeto, arquitetura, objetos, tipos, componentes, bibliotecas e pacotes, os comandos e operadores estão descritos no apêndice A.

4.2. Histórico

O VHDL surgiu em meados do ano de 1980 devido à necessidade de uma ferramenta de projeto e documentação padrão para o projeto VHSIC (*Very High Speed Integrated Circuit*) do Departamento de Defesa dos Estados Unidos (DARPA - *Defense Advanced Research Projects Agency*).

Devido à alta prioridade dada ao projeto VHSIC pelo Departamento de Defesa americano, havia uma crescente preocupação com as questões de portabilidade, documentação e usabilidade dos projetos, pois havia dezenas de fornecedores diferentes envolvidos que desenvolviam partes dos projetos ou fornecendo componentes que seriam

agregados a outros sistemas maiores. Pensando em uma forma de padronizar toda essa troca de informações tanto dos componentes ou de projetos, o Departamento de Defesa americano resolveu criar uma linguagem padronizada de descrição e documentação do circuito (*hardware*) que possibilitasse os diferentes fornecedores entender o funcionamento das outras partes envolvidas no projeto, não importando o formato original do circuito desenvolvido.

Inicialmente o VHDL foi utilizado para documentar os ASICs utilizados nos equipamentos que eram vendidos às Forças Armadas americana com o intuito de substituir os complexos manuais que descreviam o funcionamento destes dispositivos.

Em 1981 o Departamento de Defesa americano promoveu um encontro de especialistas para discutir os métodos para descrição de circuitos. Em 1983 definiu os requisitos de uma linguagem de descrição de circuitos padrão. Nesse mesmo ano foram concedidos as empresas IBM, Texas Instruments e Intermetrics um contrato de desenvolvimento da linguagem e suas respectivas ferramentas.

A linguagem foi colocada a domínio público e padronizada pelo IEEE (*Institute of Electrical and Electronics Engineers*) no ano de 1987, dando origem ao padrão IEEE 1076-1987. Como é natural ocorrer em qualquer processo de aprimoramento, em 1993 a linguagem passou por uma revisão, que acabou dando origem a IEEE 1076-1993. Passou por mais algumas revisões em 2000 e 2002, e mais recentemente em 2008, gerando assim a IEEE 1076-2008.

Dentre os diversos *package* (pacotes) existentes para a linguagem VHDL, pode-se destacar o IEEE 1164 e o IEEE 1076.3 que correspondem aos pacotes Std_logic_1164 e *Numeric_bit*/*Numeric_std*, respectivamente. Os *package* são locais onde estão armazenados as informações de uso comum, como constantes, tipos de dados, funções, procedimentos etc.

4.3. Estrutura da linguagem VHDL

A linguagem VHDL é composta basicamente por duas partes fundamentais, a primeira define a interface e a segunda o seu comportamento. A interface é definida pela declaração *entity*, e o comportamento é definido através da declaração *architecture*.

4.3.1. Entidade de projeto

Uma entidade de projeto (*design entity*), ela é composta de duas partes: declaração de entidade e de arquitetura. A declaração de entidade define a interface de comunicação com o ambiente externo, enquanto a arquitetura contém a especificação dos relacionamentos entre as entradas e saídas da entidade, conforme pode ser observado no quadro 1.

A simulação de uma entidade de projeto é realizada apenas após ele ser compilado e salvo em uma biblioteca (*library*), caso não se especifique qual biblioteca ela é salva na biblioteca de trabalho (*work*).

- **Declaração de entidade**

É a parte principal de qualquer projeto em VHDL, pois é nela onde é descrita a interface do sistema, e é onde se declaram o nome do componente e suas portas de entrada e saída. Tudo que for descrito na *entity* ficará visível a todas as outras unidades associadas com esta *entity*.

No quadro 1 têm-se o exemplo da declaração de uma entidade, observe que na cláusula "*GENERIC*" são declarados parâmetros estáticos na entidade. Na cláusula "*PORT*" é definido o modo e os tipos das portas de entrada e saída da descrição, os modos podem ser conforme tabela 4, e os tipos serão abordados no item 4.3.3.

```
entity nome_da_entidade is
     Generic (nome_da_constante : tipo := valor)
      Port (   nome_do_sinal_1 : in tipo;
               nome_do_sinal_2 : out tipo);
end nome_da_entidade;
```

Quadro 1 – Declaração de entidade

Modo	**Descrição**
In	Entrada simples
Out	Saída simples
Buffer	Saída, que pode ser referenciado internamente pela arquitetura.
Inout	Entrada e Saída

Tabela 4 – Tipos de modos das portas de uma entidade

4.3.1.1. **Arquitetura**

Como já mencionado a arquitetura descreve a relação entre as entradas e saídas da entidade, um exemplo de declaração pode ser visto no quadro 2, observe que a arquitetura possui um nome e o nome da entidade a qual ela implementa, o que possibilita ter mais de uma arquitetura para declarações de entidades diferentes, devendo atentar para o fato que cada par de declaração-arquitetura representa uma entidade distinta.

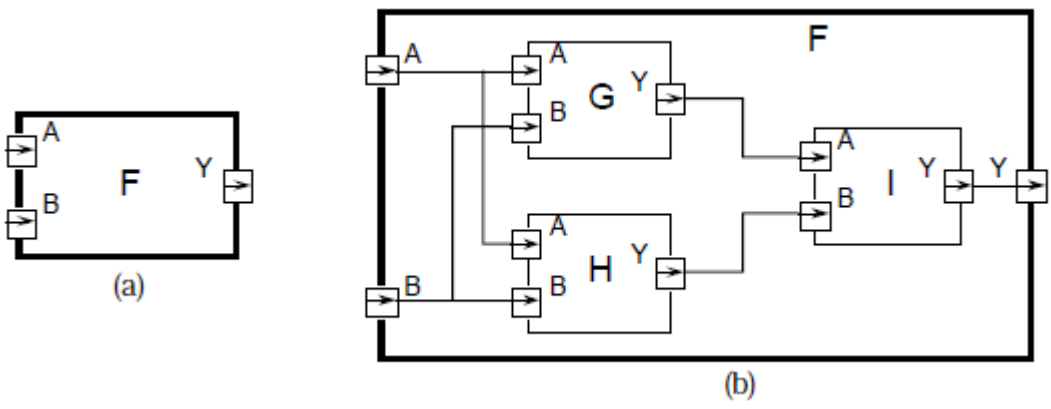
```
architecture nome_da_arquitetura of nome_da_entidade is
        --Região de declarações:
        -- declaração de sinais e constantes
        -- declaração de componentes referenciados
        -- declaração de subprogramas
        -- definição de tipos de dados locais
begin
        --Comandos concorrentes
end nome_da_arquitetura;
```

Quadro 2 – Declaração de uma arquitetura

- **Descrição Estrutural**

Um sistema digital eletrônico pode ser descrito como um módulo com entradas e/ou saídas (Ashenden,1990, p. 1-2).

Figura 11 – Exemplo de uma descrição estrutural



Fonte: Ashenden,1990

Descrição estrutural especifica como um sistema digital é composto, ou seja, como os subsistemas ou componentes são constituídos e de que forma estão interconectados. Permite a existência de vários níveis hierárquicos conforme mostra a figura 11(b), onde o módulo F é composto pelos módulos G, H e I. Note que os módulos G, H e I também possuem suas próprias descrições estruturais.

- **Descrição Comportamental**

A descrição comportamental especifica um sistema digital em termos de sua operação, ou seja, especifica as respostas nas portas de saída em termos dos estímulos aplicadas nas portas de entrada. Nos casos onde a descrição da estrutura interna de um componente não é necessária, efetua-se uma descrição que produza o mesmo resultado funcional, este tipo de descrição é chamada de: descrição funcional ou comportamental.

Para ilustrar isso, supõe-se que a entidade F na figura 11(a) seja uma função OU-Exclusivo. Então a descrição comportamental de F poderia ser demonstrada pela função Booleana: $Y = \overline{A} \cdot B + A \cdot \overline{B}$

4.3.2. Objetos

Os objetos são os elementos cabazes de armazenar valor, e estão descritos na tabela 5. No quadro 3 é exibido exemplos de declaração de objetos, e no quadro 4 exemplos de atribuição de valores.

Objeto	**Descrição**
constant	É uma constante, ou seja, um objeto de valor estático.
variable	A variável pode alterar seu valor durante o código, e é utilizada nas regiões de código seqüencial.
Signal	O sinal pode alterar seu valor durante o código, e é utilizada nas regiões de código seqüencial e concorrente.
File	Este objeto está associado a criação de arquivo.

Tabela 5 – Descrição dos tipos de objetos

```
--constante sem valor inicial
constant nome_constante       : tipo;
--sinal com valor inicial
signal nome_sinal             : tipo := valor_inicial;
--variáveis do mesmo tipo
variable nome_var_1, nome_var_2    : tipo;
```

Quadro 3 – Exemplo de declaração de objetos

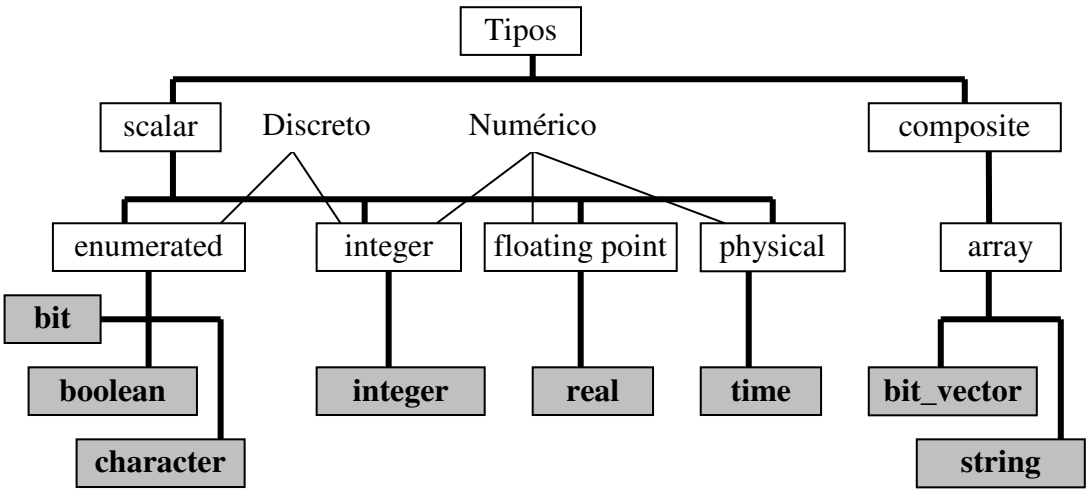
```
--atribuição de valor para sinal
nome_sinal     <=   nome_constante;
--atribuição de valor para variável
nome_var_1     :=   nome_signal;
nome_var_2     :=   nome_variável;
```

Quadro 4 – Exemplo de atribuição de valores

4.3.3. Tipos

Um tipo é um conjunto de valores com um conjunto de operações. Alguns tipos de dados em VHDL e sua hierarquia podem ser observados na figura 12, note que na parte inferior estão alguns dos tipos predefinidos da linguagem para declaração de objetos. Não é possível a transferência de valores entre objetos declarados com tipos diferentes.

Figura 12 – Hierarquia de tipos em VHDL



4.3.3.1. **Tipos escalares**

A tabela 6 descreve os tipos escalares padrões, eles são ordenados e representam valores únicos. Observe que "*natural*" e "*positive*" são subtipos do tipo "*integer*".

Valores representados no formato de ponto flutuante, normalmente, não são suportados pelas ferramentas de síntese devido à complexidade das estruturas envolvidas para realizar essas operações. Assim como o tipo "*real*" os tipos físicos também não são suportados pelas ferramentas de síntese, por exemplo, no tipo "*time*" a ferramenta de síntese ignora os intervalos de tempo definidos.

Tipo Predefinido	**Valor**	**Exemplo**
Bit	um, zero	1, 0
Boolean	Verdadeiro, falso	true, false
Character	Caracteres ASCII	a, b, C, D, (
Integer	$-2^{31}-1 <= x <= 2^{31}-1$	12, 2#1100#, 8#14#, 16#C#
Natural	$0<= x <= 2^{31}-1$	111, 2#110_1111#
Positive	$0< x <= 2^{31}-1$	16#6F#, 8#157#
Real	$-3.65*10^{47}<=x<=+3.65*10^{47}$	1.1, 1.22E+2
Time	ps=10^3fs;ns=10^3ps;us=10^3ns;ms=10^3us; sec=10^3ms;min=60sec;hr=60min	10 ns
Obs.: representação de outras bases "base#valor#" onde os "_" são ignorados		

Tabela 6 – Tipos predefinidos no pacote padrão

4.3.3.2. **Tipos compostos**

A tabela 7 descreve os tipos de "*array*" definidos na biblioteca padrão do VHDL, sendo o "*bit_vector*" um vetor de "*bit*", e o "*string*" um vetor de "*character*". Na declaração de um desses objetos, o número de elementos é declarado, utilizando as palavras reservadas "*downto*" (ordem decrescente nos índices) e "*to*" (ordem crescente nos índices).

Tipo Predefinido	Valor	Exemplo
bit_vector	tipo *bit*	“1100”
String	Tipo character	“palavra”

Tabela 7 - Tipos compostos predefinidos no pacote padrão

4.3.3.3. **Declaração de novos tipos**

A criação de novos tipos pode ser utilizada para facilitar a leitura de um código, atribuindo nomes específicos para determinadas condições. A declaração de um novo tipo é efetuada através do comando “*TYPE*”, seguida do nome escolhido para o novo tipo.

4.3.4. Componentes

Componentes são entidades de projeto empregadas na arquitetura de uma outra entidade. O uso desses elementos permite a interligação e múltiplas entidades de projeto, de modo a formar uma entidade mais complexa em um projeto hierárquico.

Um exemplo de declaração de um componente pode ser visualizado no quadro 5.

```
component nome_componente
--lista de genéricos
     generic        (n        : tipo_n  := valor);
--lista de portas
     port      (sinal_a  : modo_a  tipo_sinal_a;
                sinal_b  : modo_b  tipo_sinal_b;
                sinal_c  : modo_c  tipo_sinal_c);
end component;
```

Quadro 5 – Declaração de um componente

4.3.5. Bibliotecas e pacotes

Assim como os componentes e subprogramas, as bibliotecas e os pacotes provêem um auxílio na divisão e organização de um projeto.

As unidades de projeto compiladas são armazenadas em locais específicos, esses locais são as bibliotecas. É possível o usuário definir bibliotecas, o que possibilita o armazenamento de unidades de projeto afins num mesmo local. As bibliotecas podem ser empregadas como uma área comum que permite o emprego de uma mesma descrição por diferentes programadores.

A cláusula "*LIBRARY*" identifica as bibliotecas que podem ser referenciadas, sendo que, uma ou mais bibliotecas podem ser especificadas na lista dos nomes que segue a cláusula.

A biblioteca "*WORK*" é a biblioteca padrão utilizada para o armazenamento das unidades de projeto, e a biblioteca "*STD*" armazena os pacotes predefinidos "*STANDARD*" e "*TEXTIO*", sendo que, ambas podem ser referenciadas por qualquer unidade de projeto, sem a inclusão da cláusula "*LIBRARY STD*, *WORK*". No quadro 6 pode-se observar o uso da cláusula "*LIBRARY*".

```
-- referencia a biblioteca
library nome_biblioteca;
library biblioteca1, biblioteca2;
-- implícito em toda unidade de projeto
-- (exceto pacote Standard)
library STD, WORK;
```

Quadro 6 – Declaração de uma biblioteca

Um pacote permite sejam reunidas, declarações de constantes, tipos e subprogramas de uso freqüente. Os elementos contidos em um pacote podem ser compartilhados por mais de uma descrição, possibilitando assim a reutilização de um mesmo código em diversos projetos. Um pacote pode ser composto de duas partes: declaração do pacote e corpo do pacote (opcional).

A declaração de constantes, tipos, sinais, subprogramas e componentes devem estar dentro da declaração do pacote, enquanto que, caso o pacote contenha funções ou procedimentos, a descrição desses subprogramas deve ser apresentada no corpo do pacote.

No quadro 7 pode-se observar a declaração de um pacote e do seu corpo.

```
package nome_pacote is
--
-- declaração de:
--   subprogramas, constantes, componentes, tipos, subtipos,
-- sinais, variáveis compartilhadas
end nome_pacote;
```

```
package body nome_pacote is
-- declaração de:
--   subprogramas, constantes, componentes, tipos, subtipos,
-- sinais, variáveis compartilhadas
--  corpo de subprogramas
end nome_pacote;
```

Quadro 7 – Declaração de um pacote

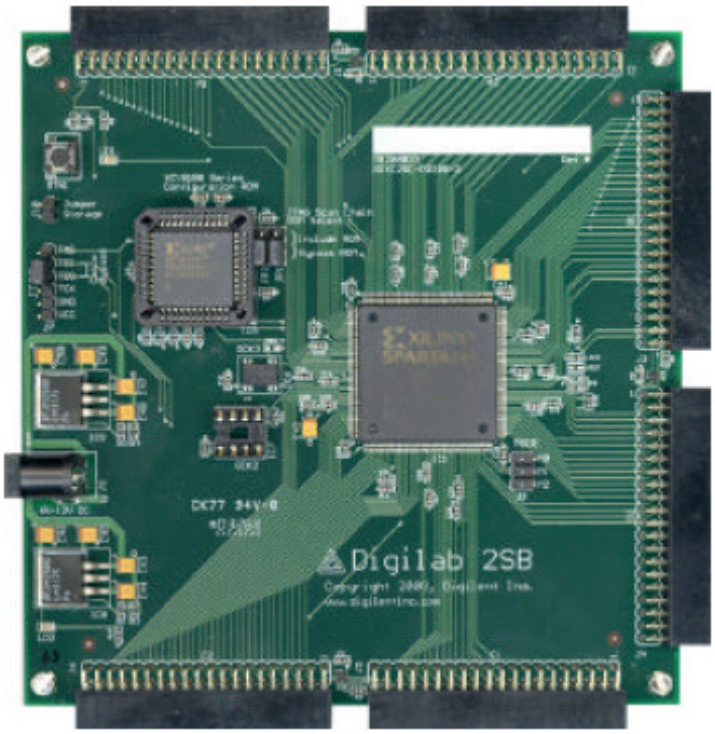
CAPÍTULO 5 - MATERIAIS E FERRAMENTAS

Este capítulo aborda os materiais e as ferramentas utilizadas no processo de desenvolvimento do microcontrolador de 32 bits proposto neste estudo. No item 5.1 há uma breve descrição da placa de desenvolvimento da marca Digilent, modelo D2SB, utilizada para sintetizar o código VHDL desenvolvido neste trabalho, o item 5.2 apresenta, de forma resumida, a placa de entradas e saídas DIO4, que em conjunto com a D2SB, permite que a interação do *software* com o mundo exterior ao *chip* seja efetuada de forma didática. O item 5.3 apresenta o ambiente de programação Xilinx ISE 7.1i utilizado na codificação do *software*, o item 5.4 apresenta o programa ModelSim XE III 6.0a utilizado na simulação.

5.1. Placa de Desenvolvimento – Digilent D2SB

A placa de desenvolvimento Digilent D2SB, que pode ser observada na figura 13, é baseada em FPGA e fornece um custo extremamente baixo e uma plataforma expansível que projeta e implementa circuitos digitais dos mais variados tipos. A placa D2SB possui um FPGA Xilinx Spartan IIE-200 com encapsulamento PQ208 que fornece portas e I/Os suficientes para implementar uma grande variedade de circuitos digitais.

Figura 13 - Placa de Desenvolvimento D2SB



Fonte: Digilent, 2004

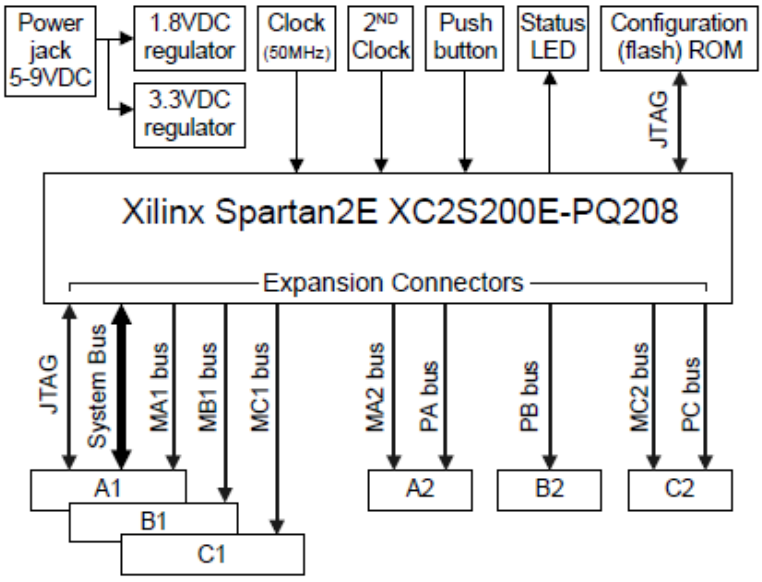
A placa D2SB fornece uma plataforma ideal para desenvolver controladores ou outros circuitos de uso geral com o mínimo esforço. A placa D2SB é totalmente compatível com o pacote gratuito WebPack da Xilinx, e ele acompanha uma fonte de alimentação e um cabo de programação.

5.1.1. Características

- Xilinx XC2S200E-200 FPGA com 200.000 portas;
- Todas as 143 I/Os disponíveis para o usuário são roteadas para 6 conectores de expansão de 40 pinos;
- Soquete para programador JTAG (*Joint Test Action Group*) 18V02 Flash Rom;
- Reguladores de tensão duplos de 1.5A (1.8V para o núcleo e 3.3V para as I/Os);
- Oscilador de 50MHz;
- Porta para programação JTAG e cabo;
- Led (*Light-Emitting Diode*) de Status e um botão para I/O básico;

Na figura 14 é possível observar algumas destas características pelo diagrama de bloco da placa.

Figura 14 – Diagrama de Blocos da placa D2SB



Fonte: Digilent, 2004

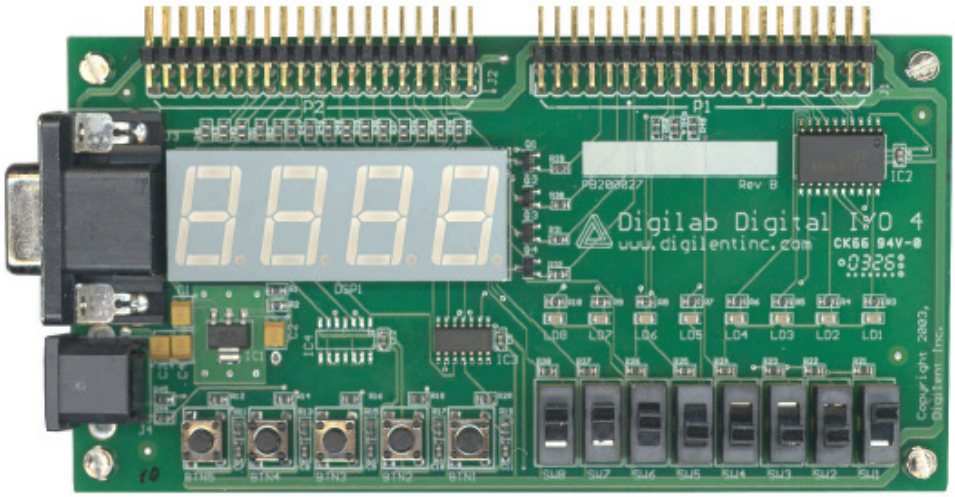
5.1.2. Aplicações

- Lógica em geral e circuitos seqüenciais de todos os tipos;
- Sistema geral digital e controladores;
- Aplicações específicas para soluções com DSP (*Digital Signal Processing*);

5.2. Placa de Entradas/Saídas Digitais - Digilent DIO4

A placa DIO4, apresentada na figura 15, fornece a um baixo custo, uma grande variedade de recursos de I/O mais comuns encontradas em sistemas digitais. Esta placa pode ser acoplada a placa Digilent D2SB para criar uma plataforma de desenvolvimento de circuitos.

Figura 15 - Placa de Entradas/Saídas DIO4



Fonte: Digilent, 2003

5.2.1. Características

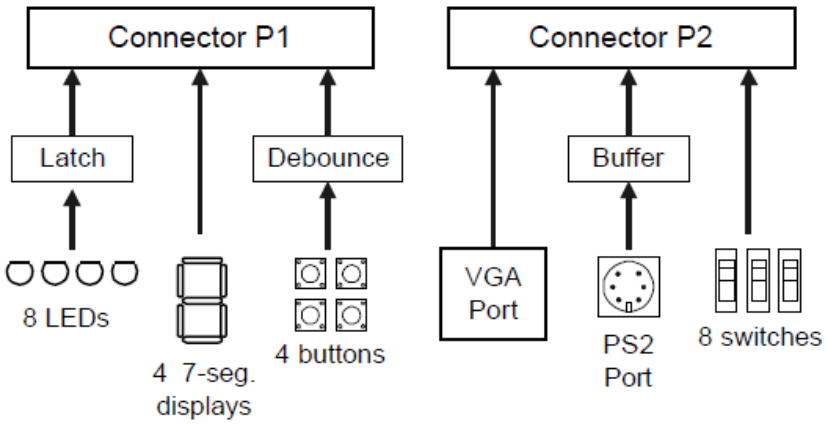
A placa DIO4 possuí as seguintes características:

- Display de LEDs de 7 segmentos com 4 dígitos;
- 8 Leds individuais;
- 4 botões pushbutton;
- 8 chaves deslizantes de 2 posições;

- Porta VGA (*Video Graphics Array*) de 3-bits;
- Porta PS/2 (*Personal System*/2) para teclado ou mouse;

Na figura 16 pode-se observar, as características citadas anteriormente, no diagrama de blocos da placa DIO4.

Figura 16 – Diagrama de Blocos da placa DIO4



Fonte: Digilent, 2003

5.2.2. Descrição funcional

A placa DIO4 pode ser acoplada a placa DS2B facilmente e adiciona vários dispositivos úteis de entrada/saída. A DIO4 é alimentada pela DS2B, e os sinais provenientes de todos os dispositivos de entradas/saídas são direcionados para pinos individuais no sistema de conexão da placa.

Todos os dispositivos da DIO4 utilizam uma alimentação de 3.3V que é fornecida pela DS2B, exceto a porta PS/2 que necessita de 5V.

5.3. Ambiente de Desenvolvimento Integrado ISE Xilinx 7.1i

A arquitetura do microcontrolador proposto e gerado neste trabalho foi descrita utilizando a linguagem VHDL. Toda codificação gerada em VHDL para sintetizar o microcontrolador foi executada dentro do ambiente de desenvolvimento integrado Xilinx ISE 7.1i (*Integrated Software Environment*). O ambiente de desenvolvimento integrado Xilinx ISE 7.1i é uma suíte de projeto de *software* que permite a criação de um projeto a partir da

entrada de um dispositivo de programação Xilinx. O ISE Project Navigator gerencia e processa o projeto através dos seguintes diversos módulos descritos a seguir (Xilinx, 2005).

No apêndice C há um pequeno tutorial explicando passo a passo como criar um projeto e depois sintetizá-lo.

5.3.1. Descrição dos Módulos do ISE

O Xilinx ISE é composto por vários módulos diferentes, cada módulo possui características e funções específicas, uma breve descrição destas características são apresentadas nos itens que seguem. Os principais módulos utilizados no desenvolvimento deste trabalho são: *Design Entry*, *Synthesis*, *Implementation* e *Verification*.

5.3.1.1. **Design Entry**

Este é o primeiro passo para criação de um novo projeto, durante a entrada do projeto, você cria seus arquivos baseados em seus objetivos. Pode-se criar o arquivo de projeto de nível superior usando uma linguagem de descrição de *hardware*, como o VHDL ou Verilog, ou utilizando um esquemático. Múltiplos formatos para os arquivos fontes de baixo nível no projeto podem ser utilizados.

5.3.1.2. **Synthesis**

Após o primeiro passo (*Design Entry*) e uma simulação opcional, pode-se executar a síntese. Durante este passo, VHDL ou Verilog, são convertidos em arquivos de *netlist* que serão utilizados como entrada na fase de implementação.

5.3.1.3. **Implementation**

Após a síntese, pode-se executar a implementação do projeto, que converte o projeto lógico em um arquivo de formato físico que pode ser carregado para o FPGA a ser programado. A partir do *Project Navigator*, pode-se executar o processo de implementação em um passo, ou pode-se executar cada um dos processos separadamente.

5.3.1.4. **Verification**

Pode-se verificar o funcionamento do projeto em vários pontos no fluxo de desenvolvimento. O programa de simulação é utilizado para verificar o funcionamento e a temporização do projeto ou uma parte dele. O simulador interpreta código VHDL ou Verilog dentro do circuito funcional e mostra resultados lógicos da descrição HDL para determinar a correta operação do circuito. A simulação permite a criação e a verificação de funções complexas em um tempo relativamente pequeno.

5.3.1.5. **Device Configuration**

Após a geração de um arquivo de programação, pode-se configurar o FPGA. Durante a execução do *Device Configuration*, são gerados os arquivos de configuração que em seguida podem ser enviados para o FPGA.

5.3.2. Características

Foram adicionadas a esta versão do ISE novas ferramentas como: *Technology Viewer*, *message filtering*, *design summary*, ISE Simulator e ModelSim Xilinx Edition III, bem como suporte para as seguintes famílias de FPGAs:

- Spartan-3E™ FPGA;
- Spartan-3L™ FPGA;
- Spartan-3™ FPGA XA (Xilinx Automotive);
- Spartan-IIE™ FPGA XA;
- CoolRunnner-II™ CPLD XA;
- XA9500XL™ CPLD XA;

5.4. Ambiente de Simulação ModelSim Xilinx Edition III 6.0a

O ModelSim XE III 6.0a é uma ferramenta utilizada na simulação e depuração de erros em programação VHDL ou Verilog, ele fornece um ambiente completo para simulação

HDL e permite verificar o funcionamento e as cartas de tempo do seu projeto, e seu código fonte HDL.

No apêndice B há um pequeno tutorial mostrando passo a passo a simulação de um programa.

5.4.1. Descrição

Existem vários recursos disponíveis dentro do ModelSim para auxiliar na simulação e correção de um código VHDL, dentre estes pode-se destacar os principais:

- Project – Criar um projeto→ Adicionar arquivos ao projeto → Compilar os arquivos de projeto → Executar simulação → Depurar Resultados;
- Simulation – Criar uma biblioteca de trabalho → Compilar arquivos de projeto → Executar simulação → Depurar resultados;
- Multiple Library – Criar uma biblioteca de trabalho → Compilar arquivos de projeto → Vincular as bibliotecas de recursos → Executar simulação → Depurar resultados;
- Debugging tools;

5.4.2. Características

As principais características do ModelSim Xilinx Edition são:

- Suporte para projetos com até 2 milhões de portas;
- Interface gráfica avançada que permite fácil acesso aos recursos e rápida depuração de erros;
- Suporte a linguagem Verilog 2001;
- Sistema totalmente integrado com o programa ISE proporcionando melhor controle dinâmico através da geração de um banco de testes gráfico automatizado;

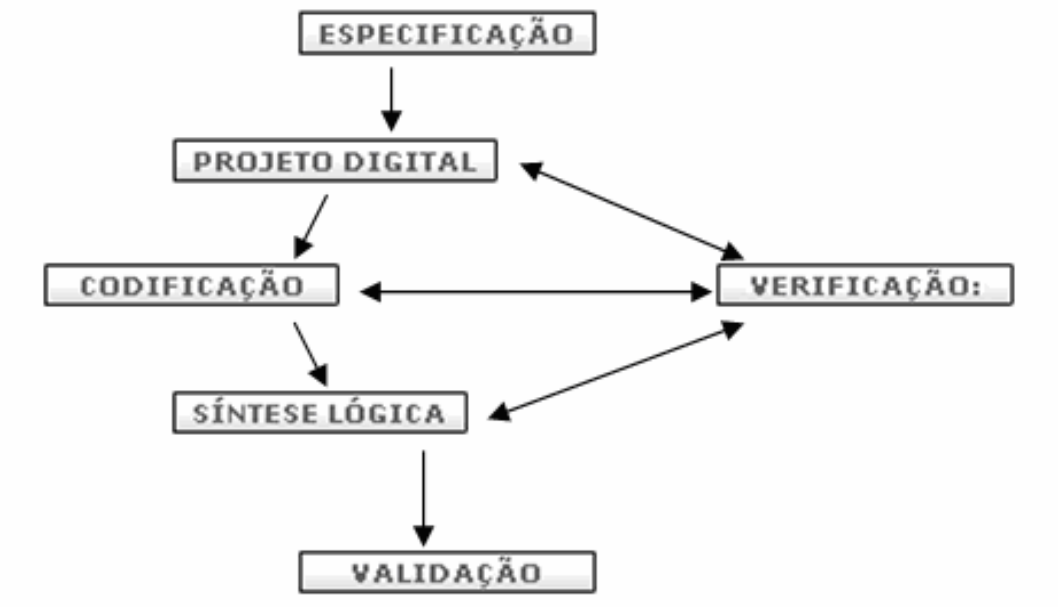
CAPÍTULO 6 - METODOLOGIA DE DESENVOLVIMENTO

Para projetar e implementar subsistemas eletrônicos digitais de alta integração além de dominar conceitos de área de arquitetura de sistemas digitais é também necessário: conhecer as características e potencialidades dos componentes programáveis (FPGA); gerar uma biblioteca de circuitos eletrônicos digitais combinacionais e seqüenciais para prover apoio ao projeto de subsistemas mais complexos e utilizar ferramentas de síntese, projeto, implementação e validação de circuitos digitais existentes no mercado.

O projeto de um Microcontrolador compreende diversas etapas que podem ser dispostas em diferentes fases para formar um fluxo de projeto. Este fluxo de projeto é composto, de forma simplificada, por três fases: especificação, projeto e validação (CEITEC, 2009).

A metodologia de desenvolvimento empregada no projeto foi a adotada pela empresa CEITEC, customizada para desenvolvimento com FPGA, cuja seqüência é descrita na figura 17. Neste capítulo está descrito os passos desta metodologia e a forma que foi utilizada.

Figura 17 - Fluxo de Projeto



Fonte: Ceitec, 2009

6.1. Especificação

É o primeiro passo dentro fluxo de projeto e é nesta fase que define a aplicação do produto, requisitos de comportamento, a sua arquitetura, tecnologia de implementação, condições de operação, disposição e características dos pinos, entre outros. Este processo envolve a especificação do comportamento esperado do projeto final. O projetista submete detalhes suficientes dentro da especificação de modo que o projeto possa ser construído. A especificação é geralmente escrita na linguagem nativa do projetista e especifica o comportamento e a função esperada do projeto utilizando descrição textual e elementos gráficos.

6.2. Projeto Digital

Após a etapa de especificação estar finalizada, pode-se iniciar o processo de implementação, que é a fase de projeto. A fase de projeto de circuitos digitais pode ser dividida, basicamente, em etapas de codificação, síntese lógica, síntese física e verificação, sendo que, por não ser um projeto de ASIC, a etapa de síntese física não é realizada.

6.2.1. Verificação

A fase de projeto é acompanhada por diferentes etapas de verificação, que deve ser executada durante toda a fase para garantir que o circuito projetado atinja as especificações, como exemplos de tipos de testes executados nesta etapa são: Funcionalidade, Atraso (STA), Testabilidade, DRC & LVS (*Design Rule Check*, *Layout Versus Schematic*).

Durante esta etapa foi realizado testes de funcionalidade, e validação dos componentes internos. Estes testes foram efetivados com o auxílio do ModelSim, e da placa de desenvolvimento Digilab 2SB.

6.2.2. Codificação

Projetos digitais são tipicamente descritos em uma linguagem de descrição de *hardware* (HDL) para emular o comportamento do circuito. Tal descrição, feita em nível de

registradores (RTL - *Register Transfer Level*), detalha exatamente o que vai acontecer com cada sinal em cada ciclo de *clock*.

No projeto a codificação foi feita em VHDL, com auxilio da ferramenta Xilinx ISE, já abordada no item 5.3.

6.2.3. Síntese Lógica

Síntese lógica é a etapa responsável por transformar a descrição textual do circuito, elaborada em linguagem HDL, em um "*netlist"* de portas lógicas funcionalmente equivalentes. No projeto esta fase é realizada automaticamente pelo ISE.

6.3. Validação

Esta é a fase responsável por garantir que todas as especificações foram atingidas. Ela pode ser realizada com o protótipo em separado, ou inserido no contexto da sua aplicação como produto.

CAPÍTULO 7 - ESTUDO DE CASO

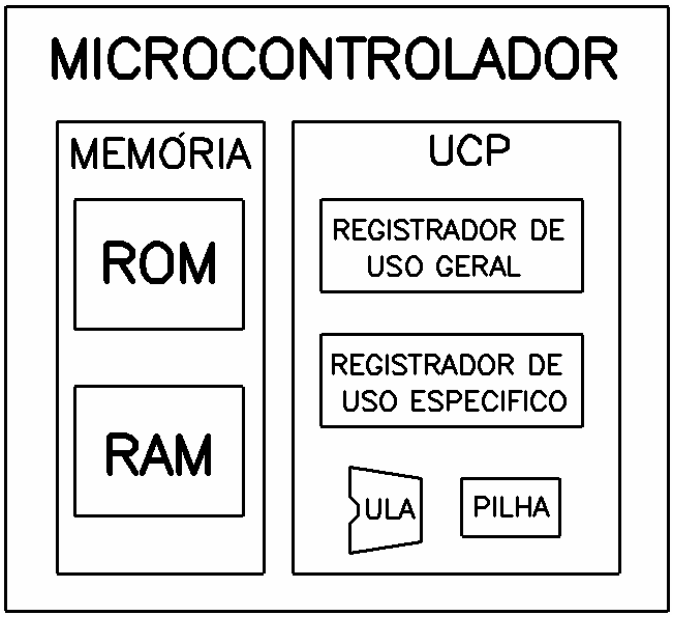
O estudo de caso é um microcontrolador de 32 bits com os conceitos da arquitetura RISC, e utilizando a configuração de Von Neumann, baseado em uma máquina de estados, que busca, decodifica e executa instruções, seus passos são:

- Busca instruções na memória ROM (pode ser mais de uma instrução).
- Decodifica instrução apontada pelo contador de programa.
- Executa a instrução.
- Reinicia o ciclo.

7.1. Requisito da arquitetura proposta

Sua organização interna possui memória ROM, RAM, ULA, multiplexadores para controlar o fluxo de dados, um conjunto de registradores de propósito geral, e outro de propósito específico, neste conjunto pode-se enquadrar o acumulador, e o registrador que contem códigos e informações da execução, a figura 18 representa de forma simplificada o microcontrolador.

Figura 18 – Organização interna



O ciclo de máquina deste microcontrolador é equivalente a 4 ciclos de clock.

A memória ROM foi emulada através de um arquivo em VHDL, com o conteúdo desejado já inserido nele, de forma que está disposto como algumas das posições iniciais da memória principal, onde o processador (UCP) busca cada instrução.

Dentre os registradores de propósito específicos, pode-se salientar o acumulador (RA) que é utilizado como entrada da ULA e local de armazenamento do resultado, e o registrador que contem os códigos e informações da execução e interrupção (RCCEI).

A seguir têm-se algumas de suas características:

- **Tipos de dados processados:**

  Números inteiros (complemento de dois). Palavras de 32 bits. Bit.

- **Funções:**

  Adição, subtração, divisão e multiplicação de números inteiros.

  Deslocamento e Rotação: direita e esquerda.

  Busca de dados: Registrador → Memória; Memória → Registrador; Registrador → Registrador;

  Saltos: simples e condicional.

  Interrupções: externa, *timer*, *software*.

  Pilha: empilha e desempilha.

  Contador de *timer* 32 bits.

  Chamada de sub-rotina, e retorno.

  Registrador de controle.

- **Desempenho:**

  Pipeline na busca de instrução.

  Ciclo de máquina de 4 ciclos de *clock*.

- **Interfaces Externas**

  32 pinos de entrada; 32 pinos de saída; Pino de *clock;* Pino de interrupção externa; Pino de *reset*

- **Restrições do projeto**

  O projeto teve limitações impostas pelas restrições de capacidade da FPGA existente no LAS, e o tempo disponível.

Para concluir a implementação foram seguido alguns passos bem definidos, a seguir está descrito a seqüência de atividades realizadas:

1. Levantamento das características do FPGA, e da placa de desenvolvimento.
2. Levantamento detalhado dos tipos de dados utilizados.

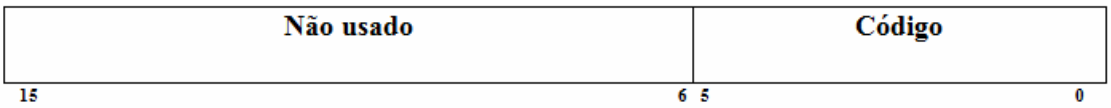
3. Escolha das instruções da arquitetura.
4. Definição da organização do microcontrolador, incluindo componentes internos.
5. Estudo aprofundado do VHDL.
6. Estudo detalhado das ferramentas disponíveis para o FPGA.
7. Implementação, teste, e validação dos componentes básicos.
8. Implementação, teste, e validação dos componentes complexos, ULA, UC.
9. Implementação, teste, e validação da integração dos componentes.

7.2. Instruções

Cada instrução tem tamanho fixo de 16 bits. O código da instrução tem tamanho de 6 bits, o que possibilita até 64 instruções distintas, eles são encontrados nos bits menos significativos da instrução, foram definidos cinco formatos de instruções descritos nas figuras 19, 20, 21, 22,23.

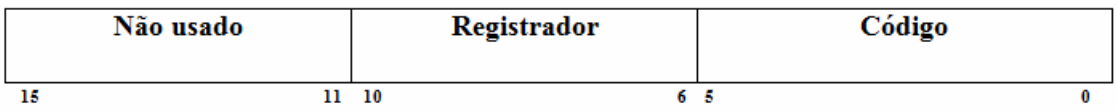
A figura 19 descreve as instruções que possuem apenas o código binário da instrução, e não utiliza os demais bits, como as operações aritméticas.

Figura 19 – 1º formato de instrução



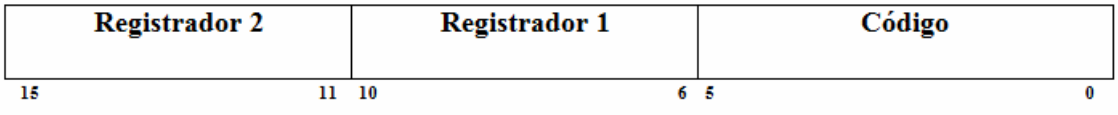
Na figura 20 as que referenciam um registrador com 5 bits, não utilizando os outros 5, como operações com a pilha.

Figura 20 – 2º formato de instrução



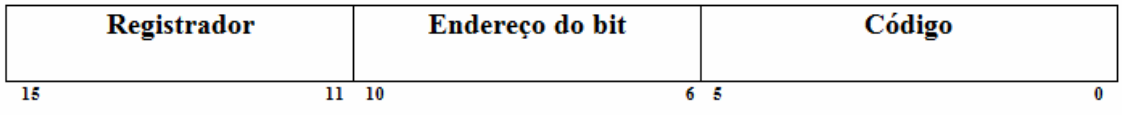
Na figura 21 é demonstrado o padrão das instruções que referenciam dois registradores como o mover.

Figura 21 – 3º formato de instrução



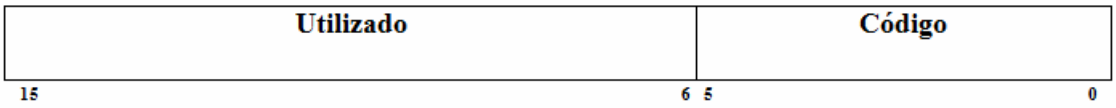
Na figura 22 é apresentado o formato das instruções que referencia um bit do registrador.

Figura 22 – 4º formato de instrução



A figura 23 descreve o formato das instruções que utilizam os 10 bits após o código, como a operação que salta a partir da posição atual, somando o valor destes 10 bits.

Figura 23 – 5º formato de instrução



Em seguida está descrito as instruções da arquitetura proposta, e suas características.

7.2.1. Aritméticas:

As instruções aritméticas disponíveis são Somar, Subtrair, Iniciar Multiplicação, Iniciar Divisão, Finalizar Multiplicação, Finalizar Divisão, e Resto. Elas seguem o formato de instrução da figura 19, e são executadas em um ciclo de máquina, o resultado da operação é armazenado no acumulador, e o registrador código de condição e erro é atualizado.

- **Somar;**

  A←A + B;

  Realiza a operação de adição, entre o registrador acumulador e o registrador 0B.

- **Subtrair;**

  A←A – B;

  Realiza a operação de subtração, entre o registrador acumulador e o registrador 0B.

- **Iniciar Multiplicação;**

  A * B;

  Inicia a operação de multiplicação, após esta instrução a ULA não pode ser utilizada durante 32 ciclos de *clock*, ou seja, deve ter pelo menos 8 instruções que não utilize a ULA, entre a operação de inicio e fim da multiplicação, como esta instrução apenas inicia a multiplicação o RA, e o RCCEI não são alterados.

- **Iniciar Divisão;**

  A **DIV** B;

  Inicia a operação de divisão, após esta instrução a ULA não pode ser utilizada durante 32 ciclos de *clock*, ou seja, deve ter pelo menos 8 instruções que não utilize a ULA, entre a operação de inicio e fim da divisão, como esta instrução apenas inicia a divisão o RA, e o RCCEI não são alterados.

- **Finalizar Multiplicação;**

  A←A * B

  Esta instrução armazena no acumulador o valor do resto da divisão, ela deve ser utilizada 32 ou mais ciclos de *clock* após iniciar a operação de multiplicação.

- **Finalizar Divisão;**

  A←A / B

  Esta instrução armazena no acumulador o valor da divisão, ela deve ser utilizada 32 ou mais ciclos de *clock* após iniciar a operação de divisão.

- **Resto;**

  A ← A % B;

  Esta instrução armazena no acumulador o valor do resto da divisão, ela deve ser utilizada 32 ou mais ciclos de *clock* após iniciar a operação de divisão. Esta instrução pode ser utilizada após a operação de finalizar a divisão.

7.2.2. Lógicas:

As instruções lógicas disponíveis são Negação, E, OU, e XOR. Elas seguem o formato de instrução da figura 19, e são executadas em um ciclo de máquina. O resultado da operação é armazenado no acumulador, e o registrador código de condição e erro é atualizado.

- **Negação;**

  A ← NOT(A);

  Realiza a operação lógica de negação, de todos os bits do registrador acumulador.

- **E;**

  A←A and B

  Realiza a operação lógica “AND”, bit a bit do registrador acumulador com o registrador 0B.

- **OU;**

  A←A OR B

  Realiza a operação lógica “OR”, bit a bit do registrador acumulador com o registrador 0B.

- **XOR;**

  A←A XOR B

  Realiza a operação lógica “ou exclusivo”, bit a bit do registrador acumulador com o registrador 0B.

7.2.3. Deslocamento e Rotação:

As instruções de Deslocamento e rotação disponíveis são Rotação Esquerda, Rotação Direita, Rotação Esquerda sem sinal, Rotação Direita sem sinal, Deslocamento Esquerdo, Deslocamento Direito. Elas seguem o formato de instrução da figura 19, e são executadas em um ciclo de máquina. O resultado da operação é armazenado no acumulador, e o registrador código de condição e erro é atualizado.

- **Rotação Esquerda;**

  (An + 1)←An para n de 0 até 30

  A(0)←A(31)

  Realiza a rotação do valor contido no registrador acumulador um bit para esquerda.

- **Rotação Direita;**

  An ← (An + 1) para n de 0 até 30

  A(31)←A(0)

  Realiza a rotação do valor contido no registrador acumulador um bit para direita.

- **Rotação Esquerda sem sinal;**

  (An + 1)←An para n de 0 até 30

  A(0)←A(31)

  Realiza a rotação do valor contido no registrador acumulador, exceto o bit da posição 31, um bit para esquerda.

- **Rotação Direita sem sinal;**

  An ← (An + 1) para n de 0 até 30

  A(31)←A(0)

Realiza a rotação do valor contido no registrador acumulador, exceto o bit da posição 31, um bit para direita.

- **Deslocamento Esquerdo;**

  (An + 1)←An para n de 0 até 30

  Realiza o deslocamento do valor contido no registrador acumulador um bit para esquerda, equivalente a operação de multiplicação por 2.

- **Deslocamento Direito;**

  An ← (An + 1) para n de 0 até 30

  Realiza o deslocamento do valor contido no registrador acumulador um bit para direita, equivalente a operação de divisão por 2.

7.2.4. Transferência de Dados

As instruções de Transferência de Dados disponíveis são Mover, Empilhar, Desempilhar, Ler Posição, Escrever Posição, Ler, Escrever. As instruções entre registradores são executadas em um ciclo de máquina, enquanto as que acessam a memória utilizam dois ciclos.

- **Mover [registrador destino] [registrador origem];**

  [Reg.1] ← [Reg.2]

  Transfere o valor armazenado no registrador de origem, para o registrador destino, esta instrução segue o formato descrito na figura 21.

- **Empilhar [Registrador];**

  Pilha(topo) ← [Reg.1] para topo de 0 até 7

  Armazena no topo da pilha o valor contido no registrador, a instrução segue o formato descrito na figura 20.

- **Desempilhar [Registrador];**

  [Reg.1] ← Pilha(topo) para topo de 0 até 7

  Armazena no registrador o valor contido no topo da pilha, a instrução segue o formato descrito na figura 20.

- **Ler Posição [Endereço];**

  RBMI ← [Enderço];

Realiza a leitura na memória e transfere o conteúdo para o registrador *buffer* de entrada da memória, sua instrução segue o formato descrito na figura 19, no entanto utiliza mais um endereço de memória, para indicar qual o endereço que será lido.

- **Escrever Posição [Endereço];**

  [Endereço] ← RBMO;

  Escreve no endereço da memória o conteúdo do registrador *buffer* de saída da memória, sua instrução segue o formato descrito na figura 19, no entanto utiliza mais um endereço de memória, para indicar qual endereço que será efetuado a escrita.

- **Ler;**

  RBMI ← @REM;

  Realiza leitura da memória na posição apontada pelo registrador endereço de memória, e salva o conteúdo no registrador *buffer* de entrada da memória, segue o formato da figura 19.

- **Escrever;**

  @REM ← RBMO;

  Armazena na memória, na posição apontada pelo registrador endereço de memória, o conteúdo do registrador *buffer* de saída da memória, a instrução segue o formato da figura 19.

7.2.5. Operações com bit

As operações com bit podem referenciar qualquer bit de qualquer registrador, porém só pode armazenar um resultado em um bit dos seguintes registradores: Acumulador, *buffer* de saída e o RCCEI. As instruções de Operações com bit disponíveis são Zerar, Setar, Negar bit, OU bit, Mover Carry, Mover bit. Elas seguem o formato de instrução da figura 22, e são executadas em um ciclo de máquina.

- **Zerar [Bit] [Registrador];**

  R[i]←0;

  Insere valor lógico ‘0’ no bit endereçável (R[i]).

- **Setar [Bit] [Registrador];**

  R[i]←1;

  Insere valor lógico ‘1’ no bit endereçável (R[i]).

- **Negar bit;**

  Not(RCCE[0]);

Inverte o valor do bit de transbordo da soma no registrador código de condição e erro.

- **E bit [Bit] [Reg.];**

  RCCE[0]←RCCE[0] and R[i];

  Realiza a operação lógica “and” entre o bit de transbordo da soma e o bit endereçável, salvando o resultado no bit de transbordo.

- **OU bit [Bit] [Reg.];**

  RCCE[0]←RCCE[0] R[i];

  Realiza a operação lógica “or” entre o bit de transbordo da soma e o bit endereçável (R[i]), salvando o resultado no bit de transbordo.

- **Mover Carry [Bit] [Reg.];**

  RCCE[0]←R[i];

  Transfere o valor do bit para o bit de transbordo de soma.

- **Mover bit [Bit] [Reg.];**

  R[i]←RCCE[0];

  Transfere o valor do bit de transbordo de soma para o bit endereçável (R[i]).

7.2.6. Controle de Fluxo de Execução

As instruções de Controle de Fluxo de Execução disponíveis são Salto Mais, Salto Menos, Salto, Chamada Mais, Chamada Menos, Chamada, Retorno. Elas são executadas em um ciclo de máquina.

- **Salto Mais [valor 10 bits];**

  RCP←RCP + Valor;

  Realiza o desvio no código para uma posição a partir da atual. O registrador contador de programa recebe o resultado da soma do seu próprio conteúdo mais o valor dos 10 bits mais significativos da instrução. Ela segue o formato descrito na figura 23.

- **Salto Menos [valor 10 bits];**

  RCP←RCP – Valor;

  Realiza o desvio no código para uma posição a partir da atual. O registrador contador de programa recebe resultado da subtração do seu próprio conteúdo menos o valor dos 10 bits mais significativos da instrução. Ela segue o formato descrito na figura 23.

- **Salto [Registrador];**

  RCP←Ri;

  Realiza o desvio no código para a posição apontada pelo registrador. O registrador contador de programa recebe o valor contido no registrador de origem, esta instrução segue o formato apresentado na figura 20.

- **Chamada Mais [valor 10 bits];**

  RCP←RCP + Valor;

  Realiza a chamada de um procedimento, desvia para uma posição a partir da atual, e salva a atual na pilha. Empilha o valor do registrador contador de programa, e ele recebe o resultado da soma de seu conteúdo mais o valor dos 10 bits mais significativos da instrução. Esta instrução segue o formato descrito na figura 23.

- **Chamada Menos [valor 10 bits];**

  RCP←RCP – Valor;

  Realiza a chamada de um procedimento, desvia para uma posição a partir da atual, e salva a atual na pilha. Empilha o valor do registrador contador de programa, e ele recebe o resultado da subtração de seu conteúdo menos o valor dos 10 bits mais significativos da instrução. Esta instrução segue o formato descrito na figura 23.

- **Chamada [Registrador];**

  RCP←Ri;

  Realiza a chamada de um procedimento, salva a posição atual na pilha e desvia para uma posição apontada pelo registrador. Empilha o valor do registrador contador de programa, e ele recebe o valor contido no registrador de origem, esta instrução segue o formato apresentado na figura 20.

- **Retorno;**

  RCP←Pilha(topo);

  Realiza um retorno de procedimento, desempilha conteúdo da pilha para o registrador contador de programa e busca próxima instrução, ela segue o formato da figura 19.

7.2.7. Saltos Condicionais

As instruções de Saltos Condicionais disponíveis são Se bit e Se não bit. Elas seguem o formato de instrução da figura 19, e são executadas em um ciclo de máquina.

- **Se Bit;**

    IF (Carry == ‘1’)

        Pc ← Pc+1;

    Realiza o salto de uma posição da memória de programa, se o bit de transbordo de soma for igual a ‘1’.

- **Se Não Bit;**

    IF (Carry == ‘0’)

    Pc ← Pc+1;

    Realiza o salto de uma posição da memória de programa, se o bit de transbordo de soma for igual a ‘0’.

7.2.8. Instruções Auxiliares

As demais instruções disponíveis são Atualiza RCCEI, Nada, e Fim. E são descritas a seguir:

- **Atualizar RCCEI;**

    RCCEI ← RCCEI

    Utiliza a ULA para atualizar os códigos de controle do registrador de código de condição e de erro e interrupção, a instrução segue o formato da figura 19.

- **Nada;**

    NOP

    Fica 4 ciclos de *clock*, ou seja, um ciclo de máquina sem realizar nenhuma ação, seu formato segue o da figura 19.

- **Fim;**

    Esta instrução faz o microcontrolador parar de buscar novas instruções, ela possui 32 bits e não segue nenhum dos formatos descrito anteriormente.

7.3. Relação *Assembly Opcode*

Após a definição do conjunto de instruções, foi definido um código binário para cada instrução, para gerar as instruções de montagem (Assembly) do microcontrolador. Na tabela 8 é apresentada todas as instruções, sua representação em *assembly* e o *opcode* correspondente.

Descrição das Instruções	*Assembly*	Instrução		
		Outros bits		Opcode
Somar RA RB	**Somar**			000000
Subtrair RA RB	**Subtrair**			000001
Iniciar multiplicação RA RB	**inic_mult**			000010
Iniciar divisão RA RB	**inic_divi**			000011
Negar RA	**Negar**			000100
E RA RB	**E**			000101
OU RA RB	**OU**			000110
XOR RA RB	**XOR**			000111
Rotação esquerda com sinal RA	**rot_esq**			001000
Rotação direita com sinal RA	**rot_dir**			001001
Rotação esquerda sem sinal RA	**rot_esq_n_sinal**			001010
Rotação direita sem sinal RA	**rot_dir_n_sinal**			001011
Deslocamento esquerdo RA	**desl_esq**			001100
Deslocamento direito RA	**desl_dir**			001101
Resto RA RB	**Resto**			001110
Atualizar RCCEI	**atu_RCCEI**			001111
Nada	**Nada**			010000
Mover [Registrador destino] [Registrador origem]	**Mover**	Reg_Ori	Reg_Dest	010001
Finalizar multiplicação RA RB	**fin_mult**			010010
Finalizar divisão RA RB	**fin_divi**			010011
Empilhar [Registrador origem]	**Empilhar**		Reg_Ori	010100
Desempilhar [Registrador destino]	**Desempilhar**		Reg_Dest	010101
Saltar posição atual mais #valor	**Saltar_mais**	Valor	Valor	010110
Saltar posição atual menos #valor	**Saltar_menos**	Valor	Valor	010111
Zerar [bit] do [registrador]	**Zerar**	Reg_Dest	Bit	011000
Setar [bit] do [registrador]	**Setar**	Reg_Dest	Bit	011001
Negar Transbordo da soma	**negar_bit**			011010
E Transbordo da soma [bit] do [registrador]	**E_bit**	Reg_Ori	Bit	011011
OU Transbordo da soma [bit] do [registrador]	**OU_bit**	Reg_Ori	Bit	011100
Mover bit Transbordo da soma [bit] do [registrador]	**mover_CY**	Reg_Ori	Bit	011101
Mover bit [bit] do [registrador] Transbordo da soma	**mover_bit**	Reg_Dest	Bit	011110
Saltar para posição [registrador]	**Saltar**		Reg_Ori	011111
Chamada de procedimento posição atual mais #valor	**Chamada_mais**	Valor	Valor	100000
Chamada de procedimento posição atual menos #valor	**Chamada_menos**	Valor	Valor	100001
Chamada de procedimento para posição [registrador]	**Chamada**		Reg_Ori	100010
Retorno de procedimento	**Retorno**			100011
Se não Transbordo da soma salta 1 posição	**Salta_se_zero**			100100
Se Transbordo da soma salta 1 posição	**Salta_se_um**			100101
Ler na posição #valor	**ler_pos**			100110
Escrever na posição #valor	**escrever_pos**			100111
Ler na posição do REM	**Ler**			101000
Escrever na posição do REM	**Escrever**			101001
Fim do programa	**Fim**			111111

Tabela 8 – Relação *Assembly Opcode*

Registradores	Sigla	Código
Registrador Acumulador	**RA**	00000
Registrador Buffer de Entrada	**RBE**	00001
Registrador Buffer de Saída	**RBS**	00010
Registrador Código de Condição e de Erro e Interrupção	**RCCEI**	00011
Timer 0	**T0**	00100
Registrador Endereço de Memória	**REM**	00101
Ponteiro de Pilha	**PP**	00110
Registrador Buffer Saída da Memória	**RBMS**	00111
Registrador Buffer Entrada da Memória	**RBME**	01000
Registrador 09	**R09**	01001
Registrador 0A	**R0A**	01010
Registrador 0B	**R0B**	01011
Registrador 0C	**R0C**	01100
Registrador 0D	**R0D**	01101
Registrador 0E	**R0E**	01110
Registrador 0F	**R0F**	01111
Registrador 10	**R10**	10000
Registrador 11	**R11**	10001
Registrador 12	**R12**	10010
Registrador 13	**R13**	10011
Registrador 14	**R14**	10100
Registrador 15	**R15**	10101
Registrador 16	**R16**	10110
Registrador 17	**R17**	10111
Registrador 18	**R18**	11000
Registrador 19	**R19**	11001
Registrador 1ª	**R1A**	11010
Registrador 1B	**R1B**	11011
Registrador 1C	**R1C**	11100
Registrador 1D	**R1D**	11101
Registrador 1E	**R1E**	11110

Tabela 9 – Relação Sigla do registrador e código

Nas instruções que referenciam explicitamente um registrador, no *assembly*, é utilizada sua sigla, na tabela 9, são apresentados todos os registradores do processador e seu código de referência.

Com a descrição do *assembly* e códigos, representada nas tabelas 8 e 9, é possível montar programas para o microcontrolador, por exemplo, o código descrito no quadro 8, realiza o soma da entrada com ela mesma e transfere resultado para saída, linhas iniciada com o caractere '#' indica comentário no código de uma linha, o código binário resultante é apresentado no quadro 9.

```
#Transferir conteúdo da entrada para o
acumulador
Mover RA RBE;
#Transferir conteúdo da entrada para o
registrador 0B
Mover R0B RBE;
#Realizar operação de soma
Somar;
#Disponibilizar resultado da soma na saída
Mover RBS RA;
Fim;
```

Quadro 8 – Exemplo de código em *Assembly*

```
0000100000010001
0000101011010001
0000000000000000
0000000010010001
00000000001111110000000000111111
```

Quadro 9 – Código binário resultante

Para facilitar testes e construções de programas para o microcontrolador, foi desenvolvido um montador, em JAVA, que está descrito no Apêndice E.

7.4. Interrupções

O microcontrolador possui 3 interrupções a interrupção externa, por tempo, e por *software* na respectiva ordem de prioridade. Elas devem ser habilitadas para funcionar.

A interrupção externa é gerada por um pino especifico do microcontrolador, a por tempo é gerada pelo *timer* interno e a por *software* deve ser controlada pelo programador.

Após a ocorrência de uma interrupção será executado o código a partir da seguinte posição da memória:

- Posição 1 interrupção externa.
- Posição 7 interrupção por tempo.
- Posição 15 interrupção por *software*.

Depois de atendido o pedido de interrupção é necessário ter um comando de retorno para voltar a posição que estava sendo executada antes da interrupção.

A seguir será descrito a forma de controle das interrupções, todos os bits referenciados são do registrador RCCEI

- Para que alguma interrupção possa ser habilitada deve-se setar o bit 16.
- Para habilitar a interrupção externa deve-se setar o bit 17.
- Para habilitar a interrupção por tempo deve-se setar o bit 18.
- Para habilitar a interrupção por *software* deve-se setar o bit 19.

A ocorrência de interrupção é detectada quando ela estiver habilitada e bit correspondente a ela valer ‘1’.

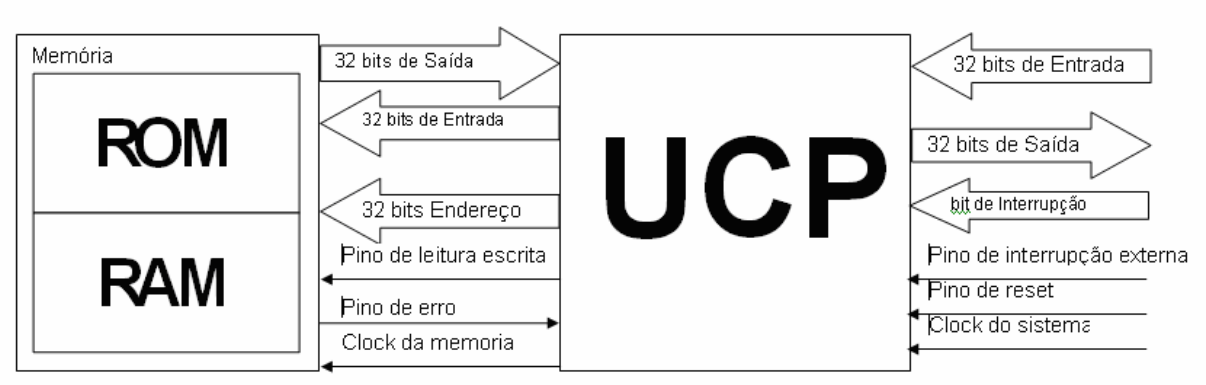
- O bit 20 detecta interrupção externa.
- O bit 21 detecta interrupção por tempo.
- O bit 22 detecta interrupção por *software*.

Após iniciar o atendimento da interrupção o bit que detecta é zerado.

7.5. Definição da organização do microcontrolador, incluindo componentes internos

Neste item abordar-se a organização do microcontrolador, bem como os seus componentes. Como foi citado anteriormente o microcontrolador é composto de um processador, e uma memória dividida em ROM e RAM, conforme pode ser observado na figura 24.

Figura 24 – Diagrama de Blocos do Microcontrolador



Os pinos externos ao microcontrolador são os de Entrada, para trazer dados de fora para dentro dele. Os pinos de saída para ele enviar dados externamente. O pino de interrupção externa que gera uma interrupção quando assumir o nível lógico alto (1), este pino deve permanecer em nível baixo (0), pois após ir para nível alto (1) já fica sinalizado a ocorrência de uma interrupção e se ele permanecer em nível alto será interpretado como a ativação de uma segunda ou mais interrupções. O sinal de *reset* reinicia o processador se permanecer em nível alto durante a busca de instrução, que ocorre após ele executar duas instruções.

7.5.1. Definição da memória

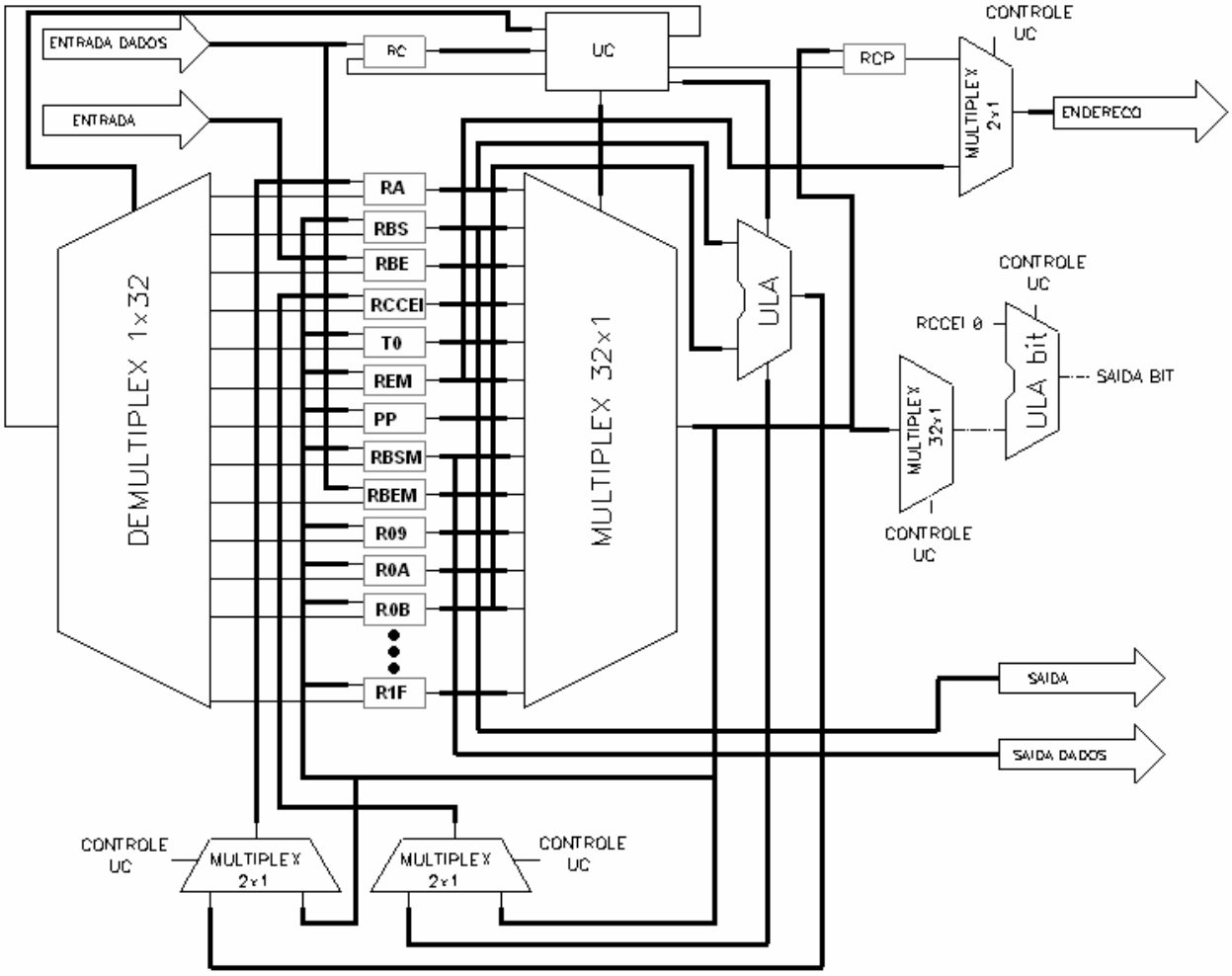
O sistema de memória do microcontrolador é formado por memória RAM e ROM. A memória total possui 128 palavras de 32 bits de comprimento, sendo as primeiras 64 palavras ROM, e as outras 64 RAM. A memória RAM tem função de armazenar os dados de programa e a ROM o programa propriamente dito. Na figura 24 pode-se observar a sua interligação com a UCP.

7.5.2. Definição da UCP

Neste item está descrito quais são os componentes da unidade central de processamento, e suas funções, enquanto no item 7.6 são descritos as entidades implementadas para obter os comportamentos desejados destes componentes. O processador possui 28 registradores que podem ser acessado diretamente pelo programador, e 2 registradores que o programador não pode acessar diretamente, uma unidade lógica e aritmética que manipula palavras de 32 bits e outra para bit, uma pilha, um *timer*, 5 *multiplex*,

um *demultiplex* e uma unidade de controle. Sua organização interna pode ser observada na figura 25.

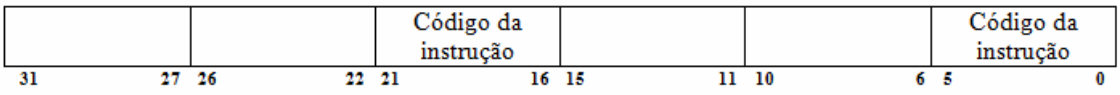
Figura 25 – Organização interna da UCP



- **Registrador de Código (RC):**

Contém 32 bits e armazena o valor da instrução que *está* sendo executada pelo processador. A cada ciclo de busca é carregado duas instruções da memória, conforme Figura 26, para este registrador, sendo que cada instrução tem 16 bits de comprimento.

Figura 26 – Instrução contida no RC



Nos bits de 0 até o 5 está o código da primeira instrução, e o da segunda nos bits de 16 até 21. Os demais bits são utilizados de acordo com o formato da instrução. O processador executa primeiro a instrução contida nos bits mais significativos do registrador.

- **Registrador Contador de Programa (RCP):**

Armazena o endereço da próxima instrução a ser executada, sendo também um contador que é incrementado a cada *clock*, ou assumindo endereço estabelecido por uma instrução de controle. Ele não é acessado diretamente pelo programador, exceto por meio de instrução de desvio de código.

- **Registrador Acumulador (RA):**

É um registrador de 32 bits de uso geral em que é utilizado nas operações da ULA, possui acesso aos bits individualmente, e controle de limpar, que atribui zero nele quando ocorrer um *reset*.

- **Registrador Buffer de Entrada (RBE):**

Este registrador armazena o valor contido nos 32 pinos de Entrada do microcontrolador.

- **Registrador Buffer de Saída (RBS):**

O valor armazenado neste registrador é transferido para os 32 pinos de Saída do microcontrolador.

- **Registrador Código de Condição e de Erro e Interrupção (RCCEI):**

É um registrador de 32 bits que guarda em seus bits informações, que indicam características do resultado de instruções que acabaram de ser executadas. E também armazena em seus bits informações a respeito das interrupções. Ele possui acesso a seus bits individualmente, e também tem o valor zerado no *reset*. A seguir são descritos os bits deste registrador e suas funções.

Descrição da função de seus bits:

[00] → Transbordo da soma (*carry*): Este bit é setado quando o resultado da operação de "Soma" é maior que 32 bits, vai 1 da subtração também.

[01] → Transbordo da multiplicação: Este bit é setado quando o resultado da operação de "Multiplicação" é maior que 32 bits.

[02] → Divisão quebrada: Este bit é setado quando o resto da operação de "Divisão" não for zero.

[03] → Divisão por zero: Este bit é setado quando for realizado operação de "Divisão" pelo número zero.

[04] → Zero: Este bit é setado quando o resultado de uma operação lógica aritmética ou de manipulação de dados é zero.

[05] → Negativo: Este bit é setado quando o resultado de uma operação lógica aritmética ou de manipulação de dados é negativo, lembrando que será utilizada a notação "Complemento de dois" para representar números negativos.

[06] → Par: Este bit é setado quando o acumulador estiver com um número par.

[07] → Impar: Este bit é setado quando o acumulador estiver com um mero impar.

[08] → Maior: analisar entrada da ULA e seta se entrada 1 for maior que a 2.

[09] → Menor: analisar entrada da ULA e seta se entrada 1 for menor que a 2.

[10] → Igual: analisar entrada da ULA e seta se entrada 1 for igual que a 2.

[11] → Pilha cheia (*Overflow*): Este bit é setado quando for inserir valor na pilha e ela estiver cheia.

[12] → Pilha vazia (*Underflow*): Este bit é setado quando for retirar valor da pilha e ela estiver vazia.

[13] → Erro de Leitura na Memória: Este bit é setado se for realizada a leitura de uma posição inexistente

[14] → Erro de Escrita na Memória: Este bit é setado se for tentar realizar escrita na ROM.

[16] → Habilitado: Deve permanece em nível alto para ser possível habilitar qualquer interrupção.

[17] → Habilita interrupção externa quando igual a '1';

[18] → Habilita interrupção por tempo quando igual a '1';

[19] → Habilita interrupção por *software* quando igual a '1';

[20] → Indica a ocorrência da interrupção externa 0, quando igual a '1';

[21] → Indica a ocorrência da interrupção por tempo 0, quando igual a '1';

[22] → Indica a ocorrência da interrupção por *software* 0, quando igual a '1';

- **Timer:**

É um contador decrescente de 32 bits que pode ser setado com o valor inicial da contagem, e fica em laço partindo do valor setado até zero, neste instante reinicia a contagem e seta o bit de interrupção.

- **Registrador de Endereço de Memória (REM):**

Nas operações de leitura e escrita na memória, utiliza-se este registrador de 32 bits para endereçar a memória.

- **Pilha (PP):**

Este componente é um conjunto de 8 registradores numerado de 0 a 7, no qual só é possível armazenar ou retirar valores do topo, a pilha é utilizado sempre que for feita uma

chamada de procedimento, e o programador não tem acesso direto a ele somente através das instruções de empilha e desempilha.

- **Registrador Buffer de Saída da Memória (RBSM):**

Quando é executada uma instrução de escrita na memória, o valor contido neste registrador é transferido para posição da memória apontada pelo REM.

- **Registrador Buffer de Entrada da Memória (RBEM):**

Quando é executada uma instrução de leitura na memória, o valor é transferido para este registrador.

- **Registradores de uso geral:**

O processador possui 21 registradores de uso geral, eles são registradores de 32 bits simples que possui apenas *clock* e entrada e saída paralela, estes registradores são representados por números em hexadecimal que estão no intervalo de 09 até 1E, destes apenas o registrador 0B possui função especifica, já que é utilizado como entrada da ULA, e possui um pino limpar para zerar seu valor, utilizado no *reset*.

- **Unidade Lógica Aritmética (ULA):**

É o circuito responsável por realizar operações aritméticas com números inteiros no processador, ela possui duas entradas de 32 bits e o código da operação de 4 bits, que determina a operação a ser executada, e disponibiliza o resultado com 32 bits e os códigos do Registrador de Código de Condição e de Erro.

A seguir as operações realizadas pela ULA:

Operações aritméticas com inteiros:

Soma → Realiza operação de adição.

Subtração → Realiza operação de subtração.

Multiplicação → Realiza operação de multiplicação.

Divisão → Realiza operação de divisão, retornando o resultado truncado.

Módulo → Realiza operação de divisão, retornando resto da divisão.

Operações lógicas:

Negação → Realiza operação lógica Negação em um registrador.

E → Realiza operação lógica E entre dois registradores.

OR → Realiza operação lógica OR entre dois registradores.

XOR → Realiza operação lógica XOR entre dois registradores.

Operações de rotação e deslocamento de bits:

Rotação Esquerda → Realiza a rotação do valor de entrada uma vez na direção esquerda.

Rotação Direita → Realiza a rotação do valor de entrada uma vez na direção direita.

Rotação Esquerda sem sinal → Realiza a rotação do valor de entrada uma vez na direção esquerda, exceto o bit da posição "31".

Rotação Direita sem sinal → Realiza a rotação do valor de entrada uma vez na direção direita, exceto o bit da posição "31".

Deslocamento Esquerdo → Realiza o deslocamento do valor de entrada uma vez na direção esquerda.

Deslocamento Direito → Realiza o deslocamento do valor de entrada uma vez na direção direita.

- **ULA_bit**

A ULA_bit é a unidade responsável pela execução das operações entre um bit e o bit de transbordo de soma do RCCEI, ela executa as operações lógicas "and", "or", "not"; retorna valor '1' ou '0'; e pode retornar o valor de uma de suas entradas, semelhante a um multiplexador.

- **Multiplexadores e Demultiplex:**

Os multiplexadores são controlados pela UC, para controlar o fluxo de dados entre os componentes contidos na arquitetura conforme demonstra a figura 25.

O componente *demultiplex* é utilizado pela UC para controlar os clocks dos registradores.

- **Unidade de controle (UC):**

Este componente é o resultado da implementação de uma máquina de estado, que realiza o ciclo de execução de uma instrução, leitura, decodificação e execução. A UC foi projetada para executar as instruções descritas no item 7.2, e reconhecer a ocorrência dos seguintes tipos de interrupções: externa, por tempo e por *software*; avaliando a prioridade das mesmas. Ela foi implementada com um nível de *pipeline*, ou seja, enquanto executa uma instrução realiza a busca da próxima. Para controlar o fluxo de dados dos registradores a UC controla os multiplexadores, para controlar o clock de cada registrador ela utiliza o demultiplexador, somente os registradores RC e RCP são controlados diretamente pela UC.

7.6. Implementação

A implementação do projeto foi feita de modo hierárquico, interligando múltiplas entidades para formar uma entidade mais complexa, neste item são demonstrados as entidades, implementadas, que juntas compõem o microcontrolador desenvolvido.

As entidades demonstradas foram divididas nos seguintes grupos:

- Memórias;
- *Multiplex* e Demultiplex;
- Registradores e contadores;
- ULA;
- Integração;
- Outros;

Todos os testes de simulação foram realizados no programa ModelSim XE III, e na placa de desenvolvimento Digilab 2SB. Os módulos foram testados e validados individualmente demonstrando que o funcionamento ocorre dentro de suas respectivas especificações e podem ser conferidos no apêndice D Testes e Validação.

7.6.1. Memórias

- **RAM**

A figura 27 descreve as entradas e saídas da "RAM", esta entidade representa uma memória do tipo RAM, contendo 64 palavras de 32 bits de comprimento.

Figura 27 – Representação da RAM

Dimensionamento e descrição das funções de suas entradas e saídas:

Entrada: porta de modo "IN", do tipo "std_logic_vector" de 32 bits, por onde é realizada a entrada de dados.

Saída: porta de modo “OUT”, do tipo “std_logic_vector” de 32 bits, esta porta retorna algum valor de uma posição de memória previamente definida.

Endereço: porta de modo “IN”, do tipo “std_logic_vector” de 6 bits, por onde é informado o endereço que se deseja realizar leitura ou escrita.

Leitura_Escrita: porta de modo “IN”, do tipo “std_logic”, indica se a operação a ser executada será de leitura, quando assume o valor ‘0’, ou escrita quando assume o valor ‘1’.

Clock: porta de modo “IN”, do tipo “std_logic”, a operação é executada quando ocorrer uma borda de subida do *clock*, ou seja, quando ocorre uma transição de nível lógico baixo (0) para nível lógico alto (1).

Comportamento: Quando houver uma borda de subida no *clock*, se for operação de leitura, a saída recebe valor contido na posição de memória apontada pelo endereço, se for escrita a palavra apontada pela entrada endereço recebe o conteúdo contido na entrada de dados.

- **ROM**

A figura 28 descreve os pinos da “ROM”, esta entidade representa uma memória do tipo ROM, com palavras de 32 bits e 64 posições de memória.

Figura 28 – Representação da ROM

Dimensionamento e descrição das funções de suas entradas e saídas:

Saída: porta de modo “OUT”, do tipo “std_logic_vector” de 32 bits, esta porta retorna o valor de uma posição de memória.

Endereço: porta de modo “IN”, do tipo “std_logic_vector” de 6 bits, por onde é informado o endereço da palavra cujo conteúdo se deseja retirar da memória.

Clock: porta de modo “IN”, do tipo “std_logic”, a operação é executada quando ocorrer uma borda de subida do *clock*.

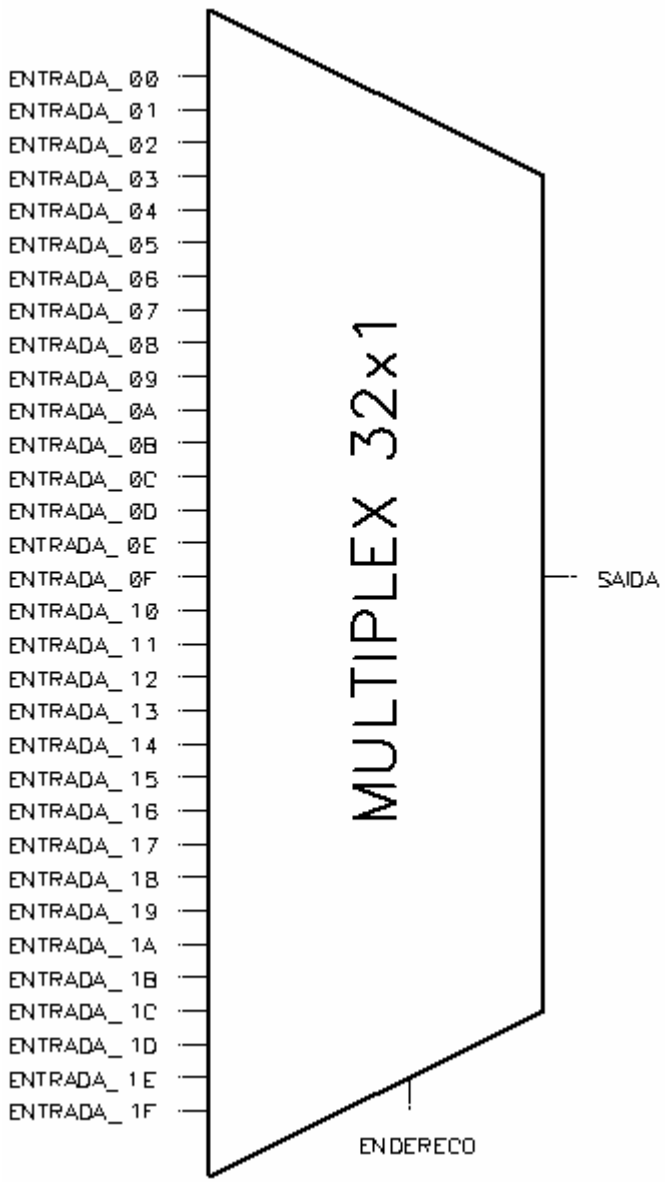
Comportamento: Quando houver uma borda de subida no *clock*, a saída recebe o valor contido na posição de memória apontada pelo endereço.

7.6.2. Multiplex e Demultiplex

- **Multiplex_32_1**

A entidade “multiplex_32_1” apresentada na figura 29 descreve um multiplexador de trinta e duas entradas e uma saída, contendo 1 bit de comprimento cada. Na arquitetura este multiplexador recebe a saída do “multiplex32_32_1”, para selecionar um bit para entrada da “ULA_bit”.

Figura 29 – Representação do Multiplex_32_1



Dimensionamento e descrição das funções de suas entradas e saídas:

Entrada_00 até Entrada_1F: portas de modo “IN”, do tipo “std_logic”, são todas as portas de entrada do multiplexador.

Saída: porta de modo “OUT”, do tipo “std_logic”, saída que recebe o valor de uma das entradas.

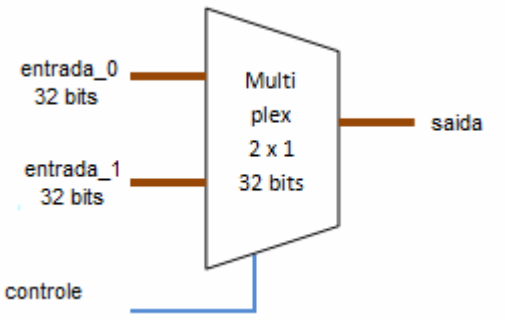
Endereço: porta de modo “IN”, do tipo “std_logic_vector” de 5 bits, que define qual entrada será transferida para saída.

Comportamento: o valor disposto na entrada, indicada pelo endereço, será transferido para a saída.

- **Multiplex32_2_1**

A entidade “multiplex32_2_1” apresentada na figura 30 descreve um multiplexador de duas entradas e uma saída, contendo 32 bits de comprimento cada. Na arquitetura este multiplexador é utilizado para selecionar qual entrada irá para os registradores, RA, RCCEI, e qual será à saída de endereço para memória.

Figura 30 – Representação do Multiplex32_2_1



Dimensionamento e descrição das funções de suas entradas e saídas:

Entrada_0: porta de modo “IN”, do tipo “std_logic_vector” de 32 bits, primeira entrada do multiplexador.

Entrada_1: porta de modo “IN”, do tipo “std_logic_vector” de 32 bits, segunda entrada do multiplexador.

Saída: porta de modo “OUT”, do tipo “std_logic_vector” de 32 bits, saída que recebe o valor de uma das entradas.

Controle: porta de modo “IN”, do tipo “std_logic”, define qual entrada será transferida para saída, entrada_0 (0), entrada_1 (1).

Comportamento: o valor disposto na entrada_0 será transferido para a saída quando controle valer ‘0’ caso contrário será transferido o valor da entrada_1.

- **Multiplex32_32_1**

A entidade “multiplex32_32_1” sua representação é semelhante ao da figura 29, porém com entradas e saída com 32 bits. Esta entidade descreve um multiplexador de trinta e duas entradas e uma saída, contendo 32 bits de comprimento cada. Na arquitetura este

multiplexador é interligado nas entradas e saídas da maioria dos registradores para controlar o fluxo de informações na UCP, a seguir estão descritos seus pinos e seu comportamento:

Entrada_00 até Entrada_1F: portas de modo “IN”, do tipo “std_logic_vector” de 32 bits, são todas as portas de entrada do multiplexador.

Saída: porta de modo “OUT”, do tipo “std_logic_vector” de 32 bits, saída que recebe o valor de uma das entradas.

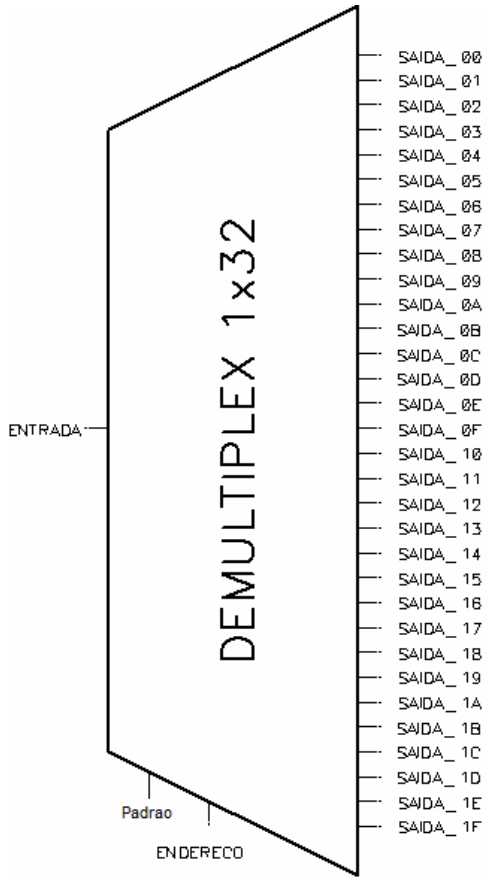
Endereço: porta de modo “IN”, do tipo “std_logic_vector” de 5 bits, que define qual entrada será transferida para saída.

Comportamento: o valor disposto na entrada de 32 bits, indicada pelo endereço, será transferido para a saída de 32 bits.

- **Demultiplex_1_32**

A entidade “demultiplex_1_32” apresentada na figura 31, implementa um circuito similar a um demultiplexador de trinta e duas saídas e uma entrada, contendo 1 bit de comprimento cada, na arquitetura este *demultiplex* é utilizado para controlar o *clock* da maioria dos registradores.

Figura 31 – Representação do Demultiplex_1_32



Dimensionamento e descrição das funções de suas entradas e saídas:

Entrada: porta de modo “IN”, do tipo “std_logic”, valor de entrada.

Saída_00 até Saída_1F: portas de modo “OUT”, do tipo “std_logic”, todas as saídas que podem receber o valor de entrada.

Padrão: porta de modo “IN”, do tipo “std_logic”, valor transferido para todas as saídas exceto a correspondente ao endereço.

Endereço: porta de modo “IN”, do tipo “std_logic_vector” de 5 bits, que define qual entrada será transferida para saída.

Comportamento: o valor disposto na entrada será transferido para a saída, indicada pelo endereço, todas as outras saídas recebem o valor padrão.

7.6.3. Registradores e contadores

- **Registrador32_c**

A entidade “registrador32_c” apresentada na figura 32, é a descrição de um registrador de 32 bits com *clear*, os registradores REM, e R0B são implementados com esta entidade.

Figura 32 – Representação do Registrador32_c

Dimensionamento e descrição das funções de suas entradas e saídas:

Entrada: porta de modo “IN”, do tipo “std_logic_vector” de 32 bits, por onde é realizada a entrada de dados.

Saída: porta de modo “OUT”, do tipo “std_logic_vector” de 32 bits, esta porta apresenta o valor contido no registrador.

Limpar: porta de modo “IN”, do tipo “std_logic”, saída recebe valor zero se este bit valer ‘1’.

Clock: porta de modo “IN”, do tipo “std_logic”, a operação é executada quando ocorrer uma borda de subida do *clock*.

Comportamento: Quando houver uma borda de subida no *clock*, a saída recebe o valor contido na entrada, e se bit limpar valer ‘1’ saída recebe valor zero independendo dos demais controles.

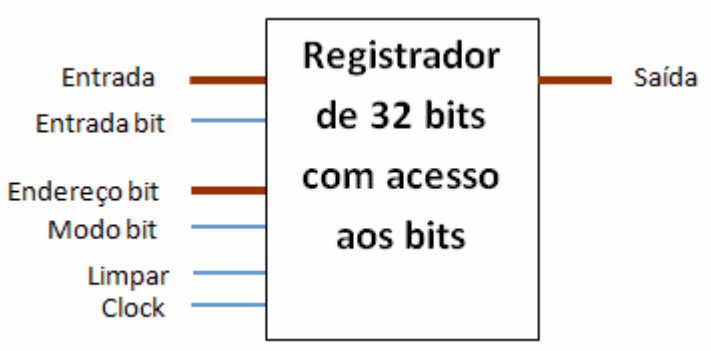
- **Registrador32**

A entidade “Registrador32” é semelhante à entidade “registrador32_c”, porém não possui o *clear*, representado na entidade anterior como limpar. Os registradores RBE, RC, RBMS, RBME, R09, R0A, e do R0C até o R1E são implementados com esta entidade

- **Registrador32_acesso_bit_c**

A entidade “registrador32_acesso_bit” apresentada na figura 33, é a descrição de um registrador de 32 bits, com um controle para modificar o valor de um único bit, e com controle de *clear*. O registrador RA é implementado com esta entidade.

Figura 33 – Representação do Registrador32_acesso_bit_c



Dimensionamento e descrição das funções de suas entradas e saídas:

Entrada: porta de modo “IN”, do tipo “std_logic_vector” de 32 bits, por onde é realizada a entrada de dados.

Saída: porta de modo “OUT”, do tipo “std_logic_vector” de 32 bits, esta porta apresenta o valor contido no registrador.

Modo_bit: porta de modo “IN”, do tipo “std_logic”, quando este bit valer ‘1’ o registrador funciona no modo bit.

Endereco_bit : porta de modo “IN”, do tipo “std_logic_vector” de 5 bits, por onde é informado o valor endereço do bit que se deseja acessar.

Entrada_bit: porta de modo “IN”, do tipo “std_logic”, pino de entrada de dado no modo bit.

Limpar: porta de modo “IN”, do tipo “std_logic”, saída recebe valor zero se este bit valer ‘1’.

Clock: porta de modo “IN”, do tipo “std_logic”, a operação é executada quando ocorrer uma borda de subida do *clock*.

Comportamento: Quando houver uma borda de subida no *clock*, se estiver no modo bit o valor da entrada_bit será transferido para o bit apontado pelo endereço_bit, caso contrário a saída recebe o valor contido na entrada, e se bit limpar valer ‘1’ saída recebe valor zero independendo dos demais controles.

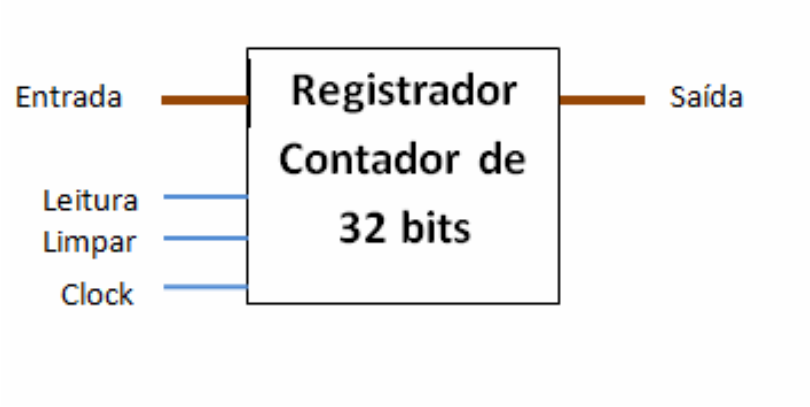
- **Registrador32_acesso_bit**

A entidade “registrador32_acesso_bit” é semelhante a entidade “registrador32_acesso_bit_c”, porém não possui o *clear*, representado na entidade anterior como limpar. O registrador RBS é implementado com esta entidade.

- **Registrador32_contador_c**

A entidade “registrador32_contador_c” apresentada na figura 34, é a descrição de um registrador contador crescente de 32 bits com *clear*. O registrador RCP é implementado com esta entidade.

Figura 34 – Representação do Registrador32_contador_c



Dimensionamento e descrição das funções de suas entradas e saídas:

Entrada: porta de modo “IN”, do tipo “std_logic_vector” de 32 bits, por onde é realizada a entrada de dados.

Saída: porta de modo “OUT”, do tipo “std_logic_vector” de 32 bits, esta porta apresenta o valor contido no registrador.

Leitura: porta de modo “IN”, do tipo “std_logic”, saída recebe valor da entrada se este bit valer ‘1’, após a borda de subida do clock.

Limpar: porta de modo “IN”, do tipo “std_logic”, saída recebe valor zero se este bit valer ‘1’.

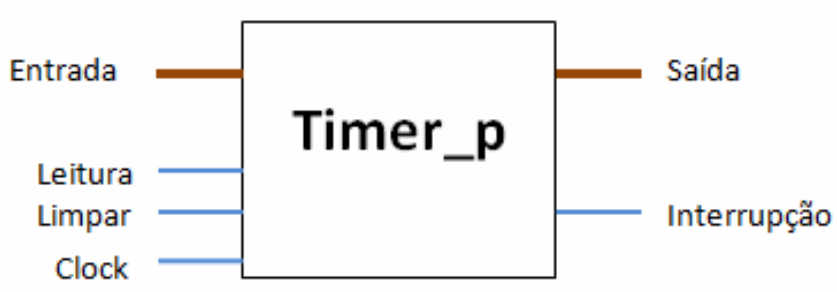
Clock: porta de modo “IN”, do tipo “std_logic”, a operação é executada quando ocorrer uma borda de subida do *clock*.

Comportamento: Quando houver uma borda de subida no *clock*, se leitura estiver com o valor ‘1’, a saída recebe o valor contido na entrada, se não saída recebe o valor anterior mais um, e se bit limpar valer ‘1’ saída recebe valor zero independendo dos demais controles.

- **Timer_p**

A entidade “timer_p” apresentada na figura 35 é semelhante a um registrador contador decrescente de 32 bits com *preset*.

Figura 35 – Representação do Timer_p



Dimensionamento e descrição das funções de suas entradas e saídas:

Entrada: porta de modo “IN”, do tipo “std_logic_vector” de 32 bits, por onde é realizada a entrada de dados.

Saída: porta de modo “OUT”, do tipo “std_logic_vector” de 32 bits, esta porta apresenta o valor contido no registrador.

Leitura: porta de modo “IN”, do tipo “std_logic”, saída recebe valor da entrada se este bit valer ‘1’, após a borda de subida do clock.

Limpar: porta de modo “IN”, do tipo “std_logic”,todos os bits da saída recebem valor ‘1’ se este bit estiver em nível alto.

Clock: porta de modo “IN”, do tipo “std_logic”, a operação é executada quando ocorrer uma borda de subida do *clock*.

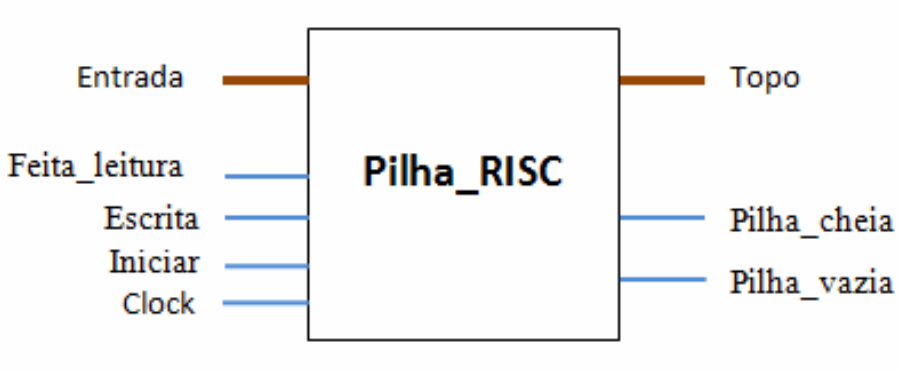
Interrupção: porta de modo “OUT”, do tipo “std_logic”, recebe valor ‘1’, quando valor de saída for zero.

Comportamento: Quando houver uma borda de subida no *clock*, se leitura estiver com o valor ‘1’, a saída recebe o valor contido na entrada, se não saída recebe o valor anterior menos um, e se bit limpar valer ‘1’ todos os bits da saída recebem ‘1’; quando o valor da saída for zero, o bit de interrupção é setado e o próximo valor de saída será o valor inicialmente informado.

- **Pilha_RISC**

A entidade “pilha_RISC” apresentada na figura 36, é a implementação de uma pilha composta por um conjunto de registradores, possui 8 níveis e 32 bits de comprimento

Figura 36 – Representação da Pilha_RISC



Dimensionamento e descrição das funções de suas entradas e saídas:

Entrada: porta de modo “IN”, do tipo “std_logic_vector” de 32 bits, por onde é realizada a entrada de dados.

Topo: porta de modo “OUT”, do tipo “std_logic_vector” de 32 bits, esta porta apresenta o valor contido no topo da pilha.

Feita_leitura: porta de modo “IN”, do tipo “std_logic”, deve permanecer em nível lógico alto durante a borda de subida de *clock* após ser realizada a leitura do topo, para poder mudar a posição de topo.

Escrita: porta de modo “IN”, do tipo “std_logic”, deve permanecer em nível lógico alto durante a borda de subida de *clock* para empilhar o valor de entrada na pilha.

Pilha_cheia: porta de modo “OUT”, do tipo “std_logic”, recebe valor ‘1’, quando todas as posições da pilha estiverem ocupadas.

Pilha_vazia: porta de modo “OUT”, do tipo “std_logic”, recebe valor ‘1’, quando não houver nenhum valor empilhado.

Iniciar: porta de modo “IN”, do tipo “std_logic”,seta pilha_vazia, e posição de topo 0.

Clock: porta de modo “IN”, do tipo “std_logic”, a operação é executada quando ocorrer uma borda de subida do *clock*.

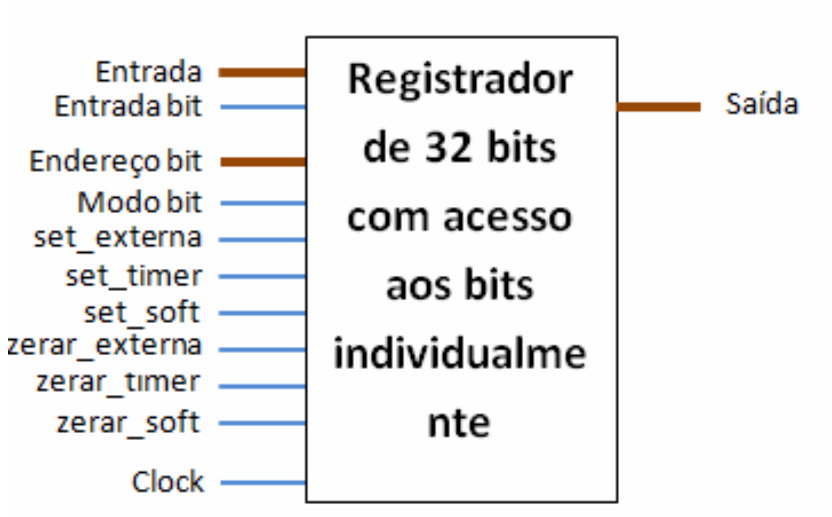
Comportamento: Na borda de subida de *clock* realiza uma operação de leitura, escrita ou inicializa os valores da pilha, de acordo com as entradas de controle Feita_leitura,Escrita,Iniciar. Na escrita ele incrementa posição do topo, indo de 0 até 7, e

armazena valor na pilha; na leitura ele decrementa posição do topo; e na inicialização seta posição de topo como 0 e informa que pilha está vazia.

- **RCCEI**

A entidade “RCCEI” apresentada na figura 37 é a descrição de um registrador de 32 bits, com um controle para modificar o valor de um único bit, com controle de *clear*, e controle especial para setar e zerar alguns bits. Este registrador foi implementado especificamente para realizar controles e interrupção.

Figura 37 – Representação do RCCEI



Seus pinos são do mesmo tipo e possuem as mesmas funções da entidade “registrador32_acesso_bit”, exceto os pinos descritos a seguir e o comportamento:

Set_externa: porta de modo “IN”, do tipo “std_logic”, quando em nível alto atribui valor ‘1’ ao bit correspondente a interrupção externa.

Set_timer: porta de modo “IN”, do tipo “std_logic”, quando em nível alto atribui valor ‘1’ ao bit correspondente a interrupção por tempo.

Set_soft: porta de modo “IN”, do tipo “std_logic”, quando em nível alto atribui valor ‘1’ ao bit correspondente a interrupção por *software*.

Zerar_externa: porta de modo “IN”, do tipo “std_logic”, quando em nível alto atribui valor ‘0’ ao bit correspondente a interrupção externa.

Zerar_timer: porta de modo “IN”, do tipo “std_logic”, quando em nível alto atribui valor ‘0’ ao bit correspondente a interrupção por tempo.

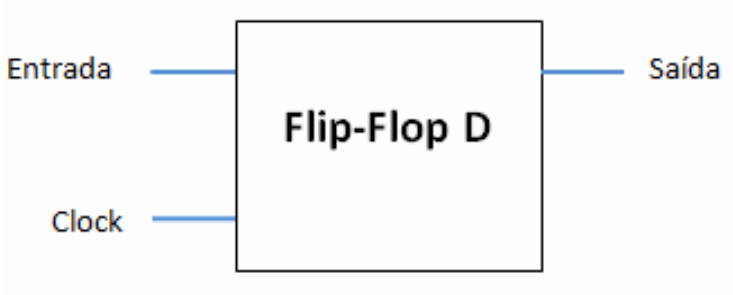
Zerar_soft: porta de modo “IN”, do tipo “std_logic”, quando em nível alto atribui valor ‘0’ ao bit correspondente a interrupção por *software*.

Comportamento: Quando houver uma borda de subida no *clock*, se estiver no modo bit o valor da entrada_bit será transferido para o bit apontado pelo endereço_bit, caso contrário a saída recebe o valor contido na entrada, e se bit limpar valer ‘1’ saída recebe valor zero. Caso uma das entradas, set e zerar, estiverem em nível alto irá atribuir ‘1’ ou ‘0’ a seu respectivo bit, que são: externa bit ‘20’, *timer* bit ‘21’, *software* bit ‘22’.

- **Flip_flop_d**

A entidade “flip_flop_d” apresentada na figura 38 é a descrição de um *flip flop* do tipo d.

Figura 38 – Representação do Flip_flop_d



Dimensionamento e descrição das funções de suas entradas e saídas:

Entrada: porta de modo “IN”, do tipo “std_logic”, por onde é realizada a entrada de dados.

Saída: porta de modo “OUT”, do tipo “std_logic”, esta porta apresenta o bit contido no *flip flop*.

Clock: porta de modo “IN”, do tipo “std_logic”, a operação é executada quando ocorrer uma borda de subida do *clock*.

Comportamento: Quando houver uma borda de subida no *clock*, a saída recebe o valor contido na entrada.

7.6.4. Unidades lógicas e aritméticas

Neste item são descritos os componentes pertencentes a ULA, e a ULA_bit que é um componente a parte da UCP.

- **Somador_subtrator**

A entidade “somador_subtrator” realiza as operações de adição e subtração de números inteiros. Dimensionamento e descrição das funções de suas entradas e saídas:

Operando_1: porta de modo “IN”, do tipo “std_logic_vector” de 32 bits, entrada do primeiro operando.

Operando_2: porta de modo “IN”, do tipo “std_logic_vector” de 32 bits, entrada do segundo operando.

Resultado: porta de modo “OUT”, do tipo “std_logic_vector” de 32 bits, esta porta apresenta o resultado da operação.

Operação: porta de modo “IN”, do tipo “std_logic”, indica qual operação é realizada.

Vem_1: porta de modo “IN”, do tipo “std_logic”, entrada de vem 1, quando está em nível alto subtrai ou adiciona ‘1’ ao resultado de acordo com a operação.

Vai_1: porta de modo “OUT”, do tipo “std_logic”, bit que indica o transbordo da operação, ou seja, quando o resultado é um número maior que 32 bits.

Comportamento: Quando operação estiver em nível baixo é realizada a adição: Operando_1 + Operando_2 + vem_1; caso contrário é realizado a subtração: . Operando_1 - Operando_2 - vem_1;

- **Multiplicador_clock**

A entidade “multiplicador_clock” realiza a operação de multiplicação de números inteiros, esta operação requer trinta e dois ciclos de *clock*. Dimensionamento e descrição das funções de suas entradas e saídas:

Operando_1: porta de modo “IN”, do tipo “std_logic_vector” de 32 bits, entrada do multiplicando.

Operando_2: porta de modo “IN”, do tipo “std_logic_vector” de 32 bits, entrada do multiplicador.

Resultado: porta de modo “OUT”, do tipo “std_logic_vector” de 32 bits, esta porta apresenta o produto da multiplicação.

Iniciar: porta de modo “IN”, do tipo “std_logic”, quando em nível alto indica que será iniciado uma nova operação e seta os valores iniciais.

Clock: porta de modo “IN”, do tipo “std_logic”, uma operação é executada quando ocorrer uma borda de subida do *clock*.

Transbordo: porta de modo “OUT”, do tipo “std_logic”, bit que indica o transbordo da operação, ou seja, quando o resultado é um número maior que 32 bits.

Comportamento: quando o iniciar está em nível alto, é armazenado os valores iniciais do circuito, e após trinta e dois ciclos de *clock* o produto é disponibilizado na saída resultado, permanecendo até que seja iniciado uma nova operação

- **Divisor_clock**

A entidade “divisor_clock” realiza as operações de divisão de números inteiros, requerendo trinta e dois ciclos de *clock* para concluir a operação. Dimensionamento e descrição das funções de suas entradas e saídas:

Operando_1: porta de modo “IN”, do tipo “std_logic_vector” de 32 bits, entrada do dividendo.

Operando_2: porta de modo “IN”, do tipo “std_logic_vector” de 32 bits, entrada do divisor.

Resultado: porta de modo “OUT”, do tipo “std_logic_vector” de 32 bits, esta porta apresenta o quociente.

Resto: porta de modo “OUT”, do tipo “std_logic_vector” de 32 bits, esta porta apresenta o resto.

Iniciar: porta de modo “IN”, do tipo “std_logic”, quando em nível alto indica que será iniciado uma nova operação e seta os valores iniciais.

Clock: porta de modo “IN”, do tipo “std_logic”, uma operação é executada quando ocorrer uma borda de subida do *clock*.

Transbordo: porta de modo “OUT”, do tipo “std_logic”, bit que indica o transbordo da operação, ou seja, quando o resultado é um número maior que 32 bits.

Comportamento: quando o iniciar está em nível alto, são armazenados os valores iniciais do circuito, com iniciar em nível baixo, após trinta e dois ciclos de *clock* o produto é disponibilizado o quociente e o resto nas suas respectivas saídas, permanecendo até que seja iniciada uma nova operação.

- **Deslocamento_1**

A entidade “deslocamento_1” realiza as operações de deslocamento e rotação a direita e esquerda, e rotação sem sinal. Dimensionamento e descrição das funções de suas entradas e saídas:

Operando_1: porta de modo “IN”, do tipo “std_logic_vector” de 32 bits, entrada do valor.

Resultado: porta de modo “OUT”, do tipo “std_logic_vector” de 32 bits, esta porta apresenta o resultado da operação.

Operação: porta de modo “IN”, do tipo “std_logic_vector” de 3 bits, indica qual operação é realizada.

Comportamento: Quando operação for equivalente a: “000” realiza: rotação esquerda;

"001" realiza: rotação direita;

"010" realiza: rotação esquerda sem sinal;

"011" realiza: rotação direita sem sinal;

"100" realiza: deslocamento esquerda;

"101" realiza: deslocamento direita;

- **Lógica**

A entidade “lógica” realiza as operações lógicas bit a bit entre duas entradas de 32 bits, disponibilizando um resultado de 32 bits. Dimensionamento e descrição das funções de suas entradas e saídas:

Operando_1: porta de modo “IN”, do tipo “std_logic_vector” de 32 bits, entrada do primeiro conjunto de bits.

Operando_2: porta de modo “IN”, do tipo “std_logic_vector” de 32 bits, entrada do segundo conjunto de bits.

Resultado: porta de modo “OUT”, do tipo “std_logic_vector” de 32 bits, esta porta apresenta o resultado da operação.

Operação: porta de modo “IN”, do tipo “std_logic_vector” de 2 bits, indica qual operação é realizada.

Comportamento: Quando operação for equivalente a: “00” realiza: negação (not) do operando_1;

"01" realiza: operação lógica and entre as duas entradas;

"10" realiza: operação lógica OR entre as duas entradas;

"11" realiza: operação lógica XOR entre as duas entradas;

- **Operador_relacional**

A entidade “operador_relacional” realiza operação relacional de números inteiros, no complemento de dois. Dimensionamento e descrição das funções de suas entradas e saídas:

Operando_1: porta de modo “IN”, do tipo “std_logic_vector” de 32 bits, entrada do primeiro operando.

Operando_2: porta de modo “IN”, do tipo “std_logic_vector” de 32 bits, entrada do primeiro operando.

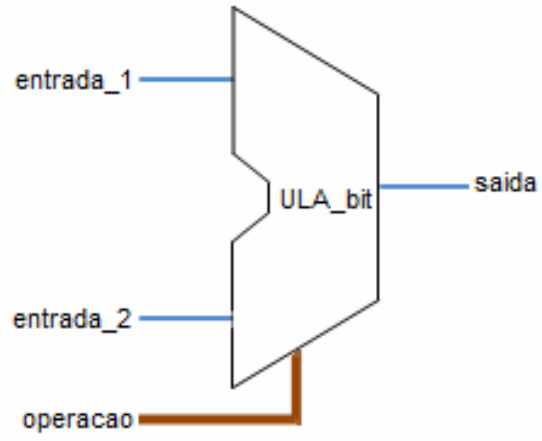
Resultado: porta de modo “OUT”, do tipo “std_logic_vector” de 2 bits, esta porta apresenta o resultado da operação.

Comportamento: resultado recebe “01” quando operando_1 for o maior, “10” caso operando 2 for o maior, e “00” se operando_1 igual ao operando_2.;

- **ULA_bit**

A entidade “ula_bit” apresentada na figura 39 é a implementação da unidade que realiza operações com entradas e saída de um bit.

Figura 39 – Representação da ULA_bit



Dimensionamento e descrição das funções de suas entradas e saídas:

Entrada_1: porta de modo “IN”, do tipo “std_logic”, entrada do primeiro operando.

Entrada_2: porta de modo “IN”, do tipo “std_logic”, entrada do segundo operando.

Operação: porta de modo “IN”, do tipo “std_logic_vector” de 3 bits, indica qual operação é realizada.

Saída: porta de modo “OUT”, do tipo “std_logic”, esta porta apresenta o resultado da operação.

Comportamento: Quando operação for equivalente a: “000” saída recebe ‘0’;"001" saída recebe ‘1’; “010” a saída recebe o valor da entrada1 negado; “011” realiza: operação lógica AND entre as duas entradas; “100” realiza: operação lógica OR entre as duas entradas; “101” saída recebe entrada_2; “111” saída recebe valor da entrada_1;

7.6.5. Integração

- **Memória**

A entidade “memória” realiza a integração entre os módulos de ROM e RAM, deixando a ROM nas posições mais baixa de memória. Dimensionamento e descrição das funções de suas entradas e saídas:

Entrada: porta de modo “IN”, do tipo “std_logic_vector” de 32 bits, por onde é realizada a entrada de dados.

Saída: porta de modo “OUT”, do tipo “std_logic_vector” de 32 bits, esta porta retorna algum valor de uma posição de memória previamente definida.

Endereço: porta de modo “IN”, do tipo “std_logic_vector” de 32 bits, por onde é informado o endereço que se deseja realizar leitura ou escrita.

Leitura_Escrita: porta de modo “IN”, do tipo “std_logic”, indica se a operação a ser executada será de leitura (bit ‘0’) ou escrita (bit ‘1’).

Clock: porta de modo “IN”, do tipo “std_logic”, a operação é executada quando ocorrer uma borda de subida do *clock*.

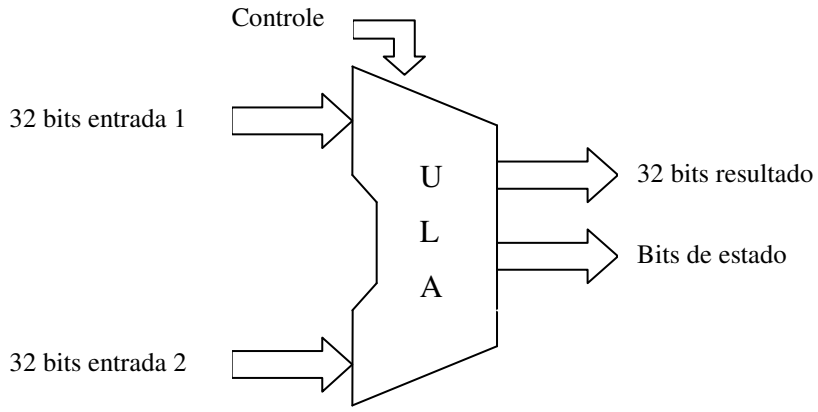
Erro: porta de modo “OUT”, do tipo “std_logic”, fica em nível alto se endereço não existir na memória ou tentar realizar escrita na ROM.

Comportamento: Transfere as entradas para ROM ou para RAM de acordo com o endereço informado, e retorna erro caso ocorra.

- **ULA_risc**

A entidade “ula_risc” é descrita na figura 40, ela utiliza todas as entidades descritas no item 7.6.4 exceto a ula_bit.

Figura 40 – Representação da ULA_risc



Dimensionamento e descrição das funções de suas entradas e saídas:

Operador_1: porta de modo “IN”, do tipo “std_logic_vector” de 32 bits, entrada para o primeiro valor da operação.

Operador_2: porta de modo “IN”, do tipo “std_logic_vector” de 32 bits, entrada para o segundo valor da operação.

Resultado: porta de modo “OUT”, do tipo “std_logic_vector” de 32 bits, esta porta apresenta o resultado da operação.

Comando: porta de modo “IN”, do tipo “std_logic_vector” de 4 bits, indica qual operação é realizada.

Vem_1: porta de modo “IN”, do tipo “std_logic”, entrada de vem ‘1’ para operação de adição e subtração.

Codigo_de_erro: porta de modo “OUT”, do tipo “std_logic_vector” de 11 bits, cada bit deste

Clock: porta de modo “IN”, do tipo “std_logic”, *clock* para o multiplicador e o divisor.

Comportamento:

Realiza as seguintes operações de acordo com o valor da entrada comando:

"0000" – Soma; "0001" – Subtração; "0010" - Multiplicação em 32 ciclos de *clock*; "0011" - Divisão em 32 ciclos de *clock*; "0100" - Operação lógica Negação; "0101" - Operação lógica E; "0110" - Operação lógica OU; "0111" - Operação lógica XOR; "1000" - Rotação esquerda; "1001" - Rotação direita; "1010" - Rotação esquerda sem sinal; "1011" - Rotação direita sem sinal; "1100" - Deslocamento esquerdo; "1101" - Deslocamento direito; "1110" - Resto da divisão; "1111" – Atualiza valores da porta codigo_de_erro, de acordo com valores da porta operador_1;

Após realizar a operação os bits da saída código_de_erro serão setados. A seguir está descrito o significado de cada bit do código_de_erro observe que o número do bit é indicado no texto:

Com relação à operação executada:

0 - Transbordo da soma; 1 - Transbordo da multiplicação; 2 - Divisão quebrada; 3 - Divisão por zero;

Com relação ao resultado:

4 – Zero; 5 – Negativo; 6 – Par; 7 – Impar;

Com relação às entradas:

8 – Maior; 9 – Menor; 10 – Igual.

- **processador_RISC**

A entidade “processador_RISC” realiza a integração entra os módulos de unidade de controle, registradores, multiplexadores e unidade lógicas e aritméticas, sua estrutura interna pode ser observada na figura 25. A seguir estão descritos seus pinos:

Entrada: porta de modo “IN”, do tipo “std_logic_vector” de 32 bits, são os pinos de entrada do processador.

Saída: porta de modo “OUT”, do tipo “std_logic_vector” de 32 bits, são os pinos de saída do processador.

Dados_in: porta de modo “IN”, do tipo “std_logic_vector” de 32 bits, são os pinos de entrada de dados provenientes da memória.

Dados_out: porta de modo “OUT”, do tipo “std_logic_vector” de 32 bits, são os pinos de saída de dados destinados a memória.

Endereco: porta de modo “OUT”, do tipo “std_logic_vector” de 32 bits, são os pinos de saída que indicam o endereço da memória.

Leitura_escrita: porta de modo “OUT”, do tipo “std_logic”, que indica sinal de leitura ou escrita para memória.

Clock_memoria: porta de modo “OUT”, do tipo “std_logic”, que indica sinal de leitura ou escrita para memória.

Erro_memo_leit: porta de modo “IN”, do tipo “std_logic”, entrada que indica a ocorrência de erro durante leitura na memória.

Erro_memo_esc: porta de modo “IN”, do tipo “std_logic”, entrada que indica a ocorrência de erro durante leitura na memória.

Int_externa: porta de modo “IN”, do tipo “std_logic”, pino que indica sinal de interrupção externa.

Reset: porta de modo “IN”, do tipo “std_logic”, pino que indica sinal para resetar o processador.

Clock: porta de modo “IN”, do tipo “std_logic”, *clock* do processador.

- **microcontrolador_RISC**

A entidade “microcontrolador_RISC” realiza a integração entra os módulos de memória, processador e divisor de *clock*, portanto a entidade final do projeto. A seguir estão descritos seus pinos:

Entrada: porta de modo “IN”, do tipo “std_logic_vector” de 32 bits, são os pinos de entrada do microcontrolador.

Saída: porta de modo “OUT”, do tipo “std_logic_vector” de 32 bits, são os pinos de saída do microcontrolador.

Int_externa: porta de modo “IN”, do tipo “std_logic”,pino que indica sinal de interrupção externa.

Reset: porta de modo “IN”, do tipo “std_logic”, pino que indica sinal para resetar o microcontrolador.

Clock: porta de modo “IN”, do tipo “std_logic”, *clock* externo, que na implementação foi dividido.

7.6.6. Outros

- **Divclk**

A entidade “divclk” realiza a divisão do sinal de *clock*, ela foi utilizada porque o *clock* disponível na placa de desenvolvimento é maior que o suportado pelo processador. Dimensionamento e descrição das funções de suas entradas e saídas:

Clock: porta de modo “IN”, do tipo “std_logic”, *clock* de entrada.

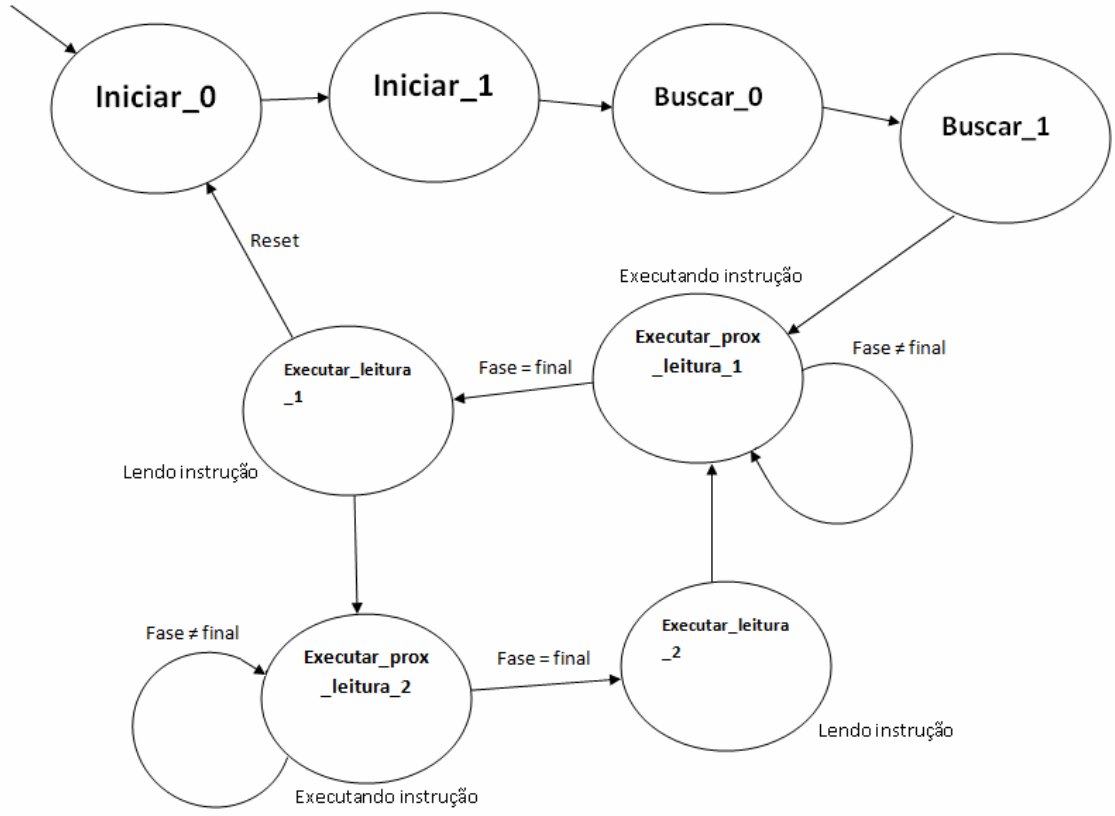
Sclock: porta de modo “OUT”, do tipo “std_logic”, sinal de saida.

Comportamento: A cada duas bordas de subida de *clock*, o valor da saída é alterado de ‘0’ para ‘1’ ou de ‘1’ para ‘0’.

- **UC_RISC**

A entidade “uc_risc” é uma máquina de estados que controla os componentes do processador para realiza a busca decodificação e execução das operações. A figura 41 demonstra os estados desta entidade:

Figura 41 – Representação dos estados da UC



Quando a UCP for ligada, ou ocorrer um *reset*, será necessário iniciar os valores de alguns registradores, esse passo é realizado pelos estados Iniciar_0 e Iniciar_1, em seguida o registrador de código recebe a primeira instrução da memória, que ocorre nos estados

Buscar_0 e Buscar_1. Concluído os passos descritos anteriormente inicia-se os ciclos de busca, decodificação e execução de instruções. No estado Executar_prox_leitura_2, é executada a instrução decodificada e realiza leitura na memória da próxima instrução, no Executar_prox_leitura_1 executa a instrução e paralelamente transfere a próxima instrução para o registrador de código. No estado Executar_leitura_1 é decodificado a instrução contida na parte mais significativa do registrador de código e verifica se houve alguma interrupção, e no estado Executar_leitura_2 decodifica a instrução contida na parte menos significativa do registrador de código.

7.7. Validação

Durante a fase de projeto foram feitos vários testes no intuito de validar os componentes da arquitetura, e finalizando o projeto foram realizados alguns testes com pequenos programas, o último teste realizado se encontra no módulo da memória ROM, ele é um programa que executa três funções distintas e seu código em *assembly* está descrito nos quadros 10, 11, 12, 13 e 14. Os testes efetuados apresentaram os resultados de acordo com o esperado.

```
#Posição 3
#Seletor de Programa
mover_CY [0] RBE;
salta_se_zero;
zerar [0] RCCEI;
saltar_mais #4;#Multiplicar
mover_CY [1] RBE;
salta_se_zero;
zerar [0] RCCEI;
saltar_mais #22;#Ler escrever
mover_CY [2] RBE;
salta_se_zero;
zerar [0] RCCEI;
saltar_mais #17;#Piscar
```

Quadro 10 – Código do seletor de programa

O Seletor de programa é utilizado ao iniciar a execução ou quando ocorre um *reset*, ele salta para uma posição de inicio dos programas implementados, de acordo com o valor contido nos três primeiros bits do RBE.

```
#posição 9 Executar Multiplicação
setar [16] RCCEI;
setar [17] RCCEI;
salta_se_um;
saltar_menos #1;
nada;
nada;
zerar [0] RCCEI;
mover RA R0A;
mover RBS RA;
salta_se_um;
saltar_menos #1;
nada;
```

```
zerar [0] RCCEI;
mover R0B R0A;
mover RBS R0B;
inic_mult;
chamada_mais
#4;#setar bit
nada;
fin_mult;
mover RBS RA;
desl_esq;
mover RBS RA;
desl_dir;
mover RBS RA;
fim;
```

Quadro 11 – Código do programa multiplicador

O programa multiplicar realiza a operação de multiplicação entre dois números informados pela entrada através de interrupção externa, foi utilizado para observar que a interrupção funciona corretamente e que poderia realizar operações enquanto a operação de multiplicação não for finalizada.

```
#posição 26 Piscar
chamada_menos #5; #Setar bit
nada;
chamada_mais #7;#Zerar bit
nada;
saltar_menos #2;
nada;
```

Quadro 12 – Código do programa piscar

O programa Piscar mantém um loop de acender e apagar os sete primeiros bits da saída através da chamada de dois procedimentos, durante os testes o microcontrolador estava com uma freqüência de *clock* bem baixo o que permitia observar o resultado, porém na velocidade máxima os leds de saída apareciam apenas acessos.

```
#Posição 29 Ler e escrever na memória
mover RBMS RBI;
escrever_pos;
#posição 30 contem valor 64 usado pela instrução escrever
ler_pos;
mover RBS RBME;
#posição 32 contem valor 64 usado pela instrução ler
nada;
nada;
fim;
```

Quadro 13 – Código do programa Ler e Escrever memória

O programa descrito no quadro 13 escreve o valor da entrada em uma posição da RAM, e posteriormente lê ele e transfere para saída, este programa foi utilizado para verificar o funcionamento da RAM.

```
#Posição 1
#Interrupcao externa
mover R0A RBE;
setar [0] RCCEI;
retorno;
nada;
```

```
#Posição 21
#Procedimento Setar bit
setar [0] RBS;
setar [1] RBS;
setar [2] RBS;
setar [3] RBS;
setar [4] RBS;
setar [5] RBS;
retorno;
nada;
```

```
#Posição 35
#Procedimento Zerar bit
zerar [0] RBS;
zerar [1] RBS;
zerar [2] RBS;
zerar [3] RBS;
zerar [4] RBS;
zerar [5] RBS;
retorno;
nada;
```

Quadro 14 – Procedimentos e funções auxiliares

A interrupção externa foi utilizada para ler um valor disponibilizado na entrada, enquanto os procedimentos setar e zerar bit foram utilizados para acender e apagar os bits da saída, sendo utilizado no programa de multiplicar para aguardar até a multiplicação ser concluída.

CONCLUSÃO

Conforme abordado na introdução a crescente demanda por produtos cada vez mais avançados tecnologicamente, a baixo custo, aumenta de forma exponencial (crescente) o uso de microcontroladores em todo tipo de equipamento agregando funcionalidades aos produtos sem elevar o seu custo final. Nas últimas décadas, enquanto os países desenvolvidos aumentavam, a nível global, a sua participação seja na fabricação quanto no projeto de microcontroladores, o Brasil negligenciava a sua entrada nesta área, ocasionando uma crescente dependência tecnológica. Iniciativas recentes do governo, através de políticas e grandes investimentos nesta área tenta reverter este quadro com a construção de fábricas de *chips* e criação de *design houses*.

Inicialmente efetuou-se o levantamento de informações sobre microcontroladores e arquiteturas de processadores, abordados nos capítulos 1 e 2, respectivamente.

Notoriamente as arquiteturas evoluem em busca da melhor performance utilizando a tecnologia mais atual disponível. Isso ocorreu com a arquitetura CISC, assim como na RISC, tornando-a o padrão de desenvolvimento, mas em todo projeto existem melhorias, e isto também ocorreu com a arquitetura RISC que recebeu aperfeiçoamentos desde a proposta inicial, e aprimorada como nas arquiteturas *Superpipelined*, e no Superescalar.

Enquanto processadores de mainframes, servidores, supercomputadores e microcomputadores são voltados ao alto desempenho, observou-se que os microcontroladores possuem outras métricas e suas características variam de acordo com a aplicação a qual ele é voltado, que pode ser a economia de energia e o preço final.

Depois de realizado o levantamento bibliográfico a respeito das arquiteturas e microcontroladores, foi definido a arquitetura RISC seria implementada.

Como resultados do trabalho foram obtidos o microcontrolador proposto, e um conjunto de instruções na forma de *assembly*, implementou-se também um montador em JAVA, agilizando o desenvolvimento de pequenos programas utilizados para testes.

Durante a fase de implementação foram encontradas várias dificuldades, que acarretaram modificações no projeto. As principais modificações foram motivadas após a implementação e validação dos algoritmos de multiplicação, divisão e deslocamento. A capacidade limitada da FPGA utilizada na implementação do microcontrolador foi insuficiente para conter os algoritmos inicialmente propostos. A limitação da síntese do VHDL para o FPGA, que não suportou todas as facilidades e funções disponíveis na linguagem, o que acabou sendo necessário alterar os algoritmos que utilizavam algumas

dessas funções não suportadas, e por último os problemas mais encontrados foram as falhas de execução do microcontrolador na FPGA. Apesar de funcionar corretamente na simulação, muitas vezes este problema era causado por falhas de sincronismo entre os componentes implementados ou de seus respectivos controles, estes problemas consumiram uma parcela significativa do tempo de implementação, sendo necessário refazer inúmeras vezes o mesmo componente, por exemplo, pode-se citar a unidade de controle que passou por dez versões até atingir um resultado final estável e que atendesse os requisitos.

Apesar dos esforços realizados na validação do microcontrolador na FPGA, não identificou-se a falha que ocorre na instrução de divisão, pois o módulo responsável pela divisão funciona corretamente na FPGA. A integração deste módulo ao restante do microcontrolador apresenta falha na placa, porém, funciona corretamente na simulação.

A falha encontrada, não inviabiliza o projeto, pois a instrução de divisão não é inerente da arquitetura RISC, por demorar vários ciclos de *clock* para ser executada, bastando identificar que o microcontrolador não tem suporte a divisão por *hardware*.

Como trabalho futuro, pode-se destacar a otimização do microcontrolador implementado, assim como a implementação de um microcontrolador em FPGA utilizando linguagem VHDL, baseado na arquitetura CISC reaproveitando a base e os módulos já elaborados neste trabalho, possibilitando assim uma comparação com o microcontrolador apresentado.

REFERÊNCIAS

ABINEE, **Balança Comercial – Janeiro - Dezembro 2008**. Disponível em: <http://www.abinee.org.br/abinee/decon/decon14.htm> Acesso em: 2 abr. 2009.

AMD, **The future is fusion – The Industry-Charging Impact of Accelerate Computing. 2008** Disponível em :< http://sites.amd.com/us/Documents/AMD_fusion_Whitepaper.pdf >. Acesso em: 20 out. 2009.

AMD, **Financial Analyst Day. 2007** Disponível em :<http://download.amd.com/Corporate/MarioRivasDec2007AMDAnalystDay.pdf>. Acesso em: 20 out. 2009

APPENDIX G. **PowerPC 601 Risc Microprocessor Instructions**, Disponível em:< http://publib.boulder.ibm.com/infocenter/systems/index.jsp?topic=/com.ibm.aix.aixassem/doc/alangref/instruction_set.htm> Acesso em: 15 out 2009.

BRASIL. Decreto-Lei no. 6.638, de 07 de novembro de 2008. Disponível em:< https://www.planalto.gov.br/ccivil_03/_ato2007-2010/2008/decreto/d6638.htm> Acesso em: 17 nov 2009.

CARDIM, Marcio Henrique Castilho. **RTRASSOC51: módulo de pipeline para um processador com arquitetura harvard superescalar embutido.** Marília, 71p. 2005.

CEITEC. **Centro de Excelência em Tecnologia Eletrônica Avançada** - Empresa. Disponível em <http://www.ceitecmicrossistemas.org.br/portal/empresa.php>. Acesso em: nov. 2009

CEITEC. **Centro de Excelência em Tecnologia Eletrônica Avançada** - Processo de Produção. Disponível em <http://www.ceitecmicrossistemas.org.br/portal/processo_p1.php>. Acesso em: abr. 2009

D´AMORE, Roberto. **VHDL - Descrição e Síntese de Circuitos Digitais**. 1ª Ed. Editora LTC. 2005.

Digilent, **D2-SB System Board Reference Manual - 2004**. Disponível em: < http://www.digilentinc.com/Data/Products/D2SB/D2SB-rm.pdf> Acesso em: 15 out 2009.

Digilent, **DIO4 Peripheral Board Reference Manual - 2003**. Disponível em: < http://www.digilentinc.com/Data/Products/DIO4/DIO4_rm.pdf> Acesso em: 15 out 2009.

Hauck, S – "**Reconfigurable Computing: A Survey of System and Software", to ACM Computing Surveys**, 2000**.**

Intel, **Intel Architecture Software Developer´s Manual – Volume 2 – Instruction Set Reference - 24319102**.1999.

Microchip, **PIC16F8X 18-pin Flash/EEPROM 8-Bit Microcontrollers**. 1998.

Ministério do Desenvolvimento, **Aladi – Associação Latino-Americana de Integração**. Disponível em: <http://www.mdic.gov.br/sitio/interna/interna.php?area=5&menu=404> Acesso em: 30 out 2009.

Mourelle, L. M.; et all. **Tradutor de C para VHDL – C2VHDL**. Revista Eletrônica de Iniciação Cientifica (REIC), ano I, número I, 2001.

Nicolosi, Denys E. C.; **Microcontrolador 8051 – Detalhado**. 6ª ed. Editora Érica. 1998.

Ordonez, E., D., M.. at all – "**Projeto, Desempenho e Aplicações de Sistemas Digitais em Circuitos Programáveis (FPGAs)".** Bless Gráfica e Editora Ltda. 2003.

Ordonez, E., D., M.; Penteado, C., G.; Silva, A., C., R.. **"Microcontroladores e FPGAs-Aplicações em Automação".** Novatec Editora. 2006.

Patterson, ., A.; Hennessy, J., L .. **Organização e Projeto de Computadores**. 3ª ed. Editora Campus, 2005.

Patterson, D. A. (1985). "**Reduced instruction set computers**". Commun. Of the ACM, v. 28, p. 8-21, jan.

Patterson, D. A. (1982); SEQUIN, C. H.. "**A VLSI RISC**". IEEE Computer Magazine, v. 15, p. 8-22, set.

Perry, D. – **" VHDL** " . McGrawHill, 1998.

Protec, **Brasil busca reduzir vácuo tecnológico em microeletrônica**, Disponível em:< http://www.protec.org.br/noticias.asp?cod=1390> Acesso em: 2 abr. 2009.

Ribeiro, A. A. L. "**Reconfigurabilidade dinâmica e remota de FPGAs**", 2002

Skahill, K. – **"VHDL for Programmable Logic"**. Addison Wesley, 1998.

Souza, D. J. – **"Desbravando o PIC"**. São Paulo: Érica, 2000.

Stallings, William. **Arquitetura e Organização de Computadores: Projeto para o Desempenho**. 5ª Ed. São Paulo: Prentice Hall, 2002.

Tanenbaum, A., S.. **Organização e Arquitetura de Computadores**. 5ª Ed. Pearson/Prentice Hall. 2007.

Tocci, J. R.. "**Sistemas Digitais - Princípios e Aplicações"**. 8ª Ed. Prentice Hall do Brasil. 2003.

TORRES, Gabriel. **Hardware: curso completo**. 4ª ed. Rio de Janeiro, Axcel Books do Brasil, 2001.

Weber, Raul Fernando. **Fundamentos de Arquitetura de Computadores**. 3ª ed. Porto Alegre: Sagra Luzzatto, 2004.

Xilinx. **DS077 – Spartan IIE FPGA Family Data Sheet**, 2008.

Xilinx. **Our History**. Disponível em:<http://www.xilinx.com/company/history.htm> Acesso em: 18 out. 2009

Xilinx. **ISE Overview - 2005** Disponível em:<http://www.xilinx.com/itp/xilinx7/help/iseguide/html/ise_b_overview.htm> Acesso em: 04 nov 2009.

Xilinx – **" The Practical Xilinx Designer Lab Book "**. V.1.3 Prentice Hall, 1998.

Xilinx – **"Quick Start Guide for Xilinx Alliance Series"** V.1.5. Xilinx, 1998.

Xilinx – **"Xilinx Student Edition-The Complete Programmable Logic Design Environment**" V.1.3 . Prentice Hall. 1998.

APÊNDICE A – Operadores, comandos seqüenciais e concorrentes da linguagem VHDL

Este apêndice descreve os operadores da linguagem VHDL e os comandos seqüenciais e concorrentes, no intuito de complementar as informações do capítulo 4.

7.7.1. Operadores

Na tabela 10 são apresentados, em ordem crescente de preferência, os operadores e suas classes, para alterar a ordem de preferência podem ser empregados os parênteses. Observe que o operador lógico "*not*" é classificado na classe "diversos" pelo fato de ter uma preferência mais alta.

Preferência	Classe	Operador
menor	Lógico	And, Or, Nand, Nor, Xor, Xnor
.	Relacional	=, /=, <, <=, >, >=
.	Deslocamento	Sll, Srl, Sla, Sra, Rol, Ror
.	Adição	+, -, &
.	Sinal	+, -
.	Multiplicação	*, /, Mod, Rem
maior	diversos	**, Abs, Not

Tabela 10 – Classes e operadores

A seguir está a descrição de cada operador:

And → Operador lógico "and", sendo os operandos do tipo bit ou boolean, ou vetores unidimensionais desses tipos.

Or → Operador lógico "or", sendo os operandos do tipo bit ou boolean, ou vetores unidimensionais desses tipos.

Nand → Operador lógico "and" negado, sendo os operandos do tipo bit ou boolean, ou vetores unidimensionais desses tipos.

Nor → Operador lógico "or" negado, sendo os operandos do tipo bit ou boolean, ou vetores unidimensionais desses tipos.

Xor → Operador lógico “xor” (ou-exclusivo), sendo os operandos do tipo bit ou boolean, ou vetores unidimensionais desses tipos.

Xnor → Operador lógico “xnor” (coincidência), sendo os operandos do tipo bit ou boolean, ou vetores unidimensionais desses tipos.

= → Operador relacional de igualdade, sendo os operandos do mesmo tipo.

/= → Operador relacional de desigualdade, sendo os operandos do mesmo tipo.

< → Operador relacional “menor”, os operandos devem ser do tipo escalar, ou vetor unidimensional com elementos do tipo inteiro ou enumerado, e ambos operandos do mesmo tipo.

<= →Operador relacional “menor igual”, os operandos devem ser do tipo escalar, ou vetor unidimensional com elementos do tipo inteiro ou enumerado, e ambos operandos do mesmo tipo.

> → Operador relacional “maior”, os operandos devem ser do tipo escalar, ou vetor unidimensional com elementos do tipo inteiro ou enumerado, e ambos operandos do mesmo tipo.

>= →Operador relacional “maior igual”, os operandos devem ser do tipo escalar, ou vetor unidimensional com elementos do tipo inteiro ou enumerado, e ambos operandos do mesmo tipo.

Sll → Operador shift left logical (deslocamento lógico a esquerda), realizado em vetores com um numero inteiro de posição a ser deslocadas.

Srl → Operador shift right logical (deslocamento lógico a direita), realizado em vetores com um numero inteiro de posição a ser deslocadas.

Sla → Operador shift left arithmetic (deslocamento aritmético a esquerda), realizado em vetores com um numero inteiro de posição a ser deslocadas.

Sra → Operador shift right arithmetic (deslocamento aritmético a direita), realizado em vetores com um numero inteiro de posição a ser deslocadas.

Rol → Operador rotate left logical (rotacionamento lógico a esquerda), realizado em vetores com um numero inteiro de posição a ser deslocadas.

Ror → Operador rotate right logical (rotacionamento lógico a direita), realizado em vetores com um numero inteiro de posição a ser deslocadas.

+ → Operador de soma entre elementos numéricos do mesmo tipo.

- → Operador de subtração entre elementos numéricos do mesmo tipo.

& → Operador de concatenação, sendo o resultado um vetor de tamanho igual a soma do tamanho de seus operandos.

+ → Sinal de identidade tem o mesmo significado do operador matemático de adição.

- → Sinal de negação tem o mesmo significado do operador matemático de subtração.

* → Operador aritmético de multiplicação, seus operandos podem ser do tipo integer, real e physical.

/ → Operador aritmético de divisão, seus operandos podem ser do tipo integer, real e physical.

Mod → Operador que retorna o módulo entre operandos integer.

Rem → Operador que retorna o resto da divisão entre operandos integer.

** → Operador de exponenciação pode ser aplicada em tipos inteiros e reais, com expoentes do tipo inteiro.

Abs → Operador valor absoluto, pode ser aplicado a qualquer tipo.

Not → Operação lógica de negação, pode ser aplicada em tipos bit boolean e vetores desses tipos.

7.7.2. Comandos concorrentes

Os comandos concorrentes do código são executados em conjunto, portanto a seqüência de apresentação é irrelevante.

7.7.2.1. **Atraso delta (Δ)**

Em regiões de código concorrente, a ordem dos comandos no código não devem interferir no resultado final da simulação. Para que isso ocorra, o simulador necessita de um atraso interno, denominado "Δ", de modo a coordenar a avaliação de eventos concorrentes, o valor padrão ou inicial do intervalo do tempo Δ é considerado nulo (Δ = 0). Durante uma simulação, a mudança de uma condição do circuito pode gerar a ativação um ou mais comandos do código. O resultado de todos os comandos ativados são transferidos 1Δ após a sua ativação, gerando uma nova condição que pode ativar novos comandos, repetindo o processo por sucessivas iterações até atingir uma condição estável.

7.7.2.2. **Atribuição de valores para um sinal**

A atribuição de valores em um sinal pode ocorrer em regiões de código concorrentes e seqüenciais.

Como pode ser visto no quadro 15 o delimitador para esta operação é "<=".

```
--tipos devem ser iguais
nome_sinal1    <=   nome_sinal_fonte;
nome_sinal2    <=   nome_signal1 or nome_sinal_fonte;
```

Quadro 15 – Exemplo de atribuição de valores para sinal

7.7.2.3. ***"when else"***

A construção "*when else*" permite transferir condicionalmente um sinal, conforme pode ser observado no quadro 16 a primeira condição que retornar verdadeiro definirá o valor a ser transferido para o sinal, caso nenhum retorne verdadeiro o sinal recebe o valor da última expressão.

```
                    --se condicao_1 = verdadeira
sinal_destino <=    expressao_1 when condicao_1 else
                    --se condicao_2 = verdadeira
                    expressao_2 when condicao_2 else
                    --nenhuma condição verdadeira
                    expressao_3;
```

Quadro 16 – Construção "*when else*"

7.7.2.4. ***"with select"***

A construção "*with select*" transfere um valor para um sinal segundo uma lista de opções, as quais devem ser mutuamente exclusivas, o quadro 17 mostra um exemplo dela. Nesta construção não há prioridade entre as opções diferentemente do "*when else*". Pode-se agrupar as opções com o delimitador "|" que funciona como uma condição "*or*", ou pelas palavras reservadas "*downto*" e "*to*" que delimita um intervalo de valores como opção.

```
--seleciona a expressão caso expressao_ou_variável seja igual
with expressao_ou_variável select
                    --condicao_1
sinal_destino <=    expressao_1 when condicao_1,
                    --condicao_2 ou condicao_3
                    expressao_2 when condicao_2 | condicao_3,
                    --condicao_4 até condicao_6
                    expressao_3 when condicao_4 to condicao_6,
                    --qualquer outro valor
                    expressao_4 when others;
```

Quadro 17 – Construção "*with select*"

7.7.2.5. **"*block*"**

A construção "*block*" (bloco) divide o código em regiões para facilitar o entendimento da descrição. O uso de sinais é permitido dentro de um *block*, porém, a sua visibilidade ficará restrita ao interior do bloco. O conteúdo de um bloco é executado de forma concorrente aos outros blocos existentes na descrição. Seu formato é apresentado no quadro 18.

```
nome_do_block: block expressao_de_guarda_opcional
     --declaração de sinais visíveis no bloco
begin
     --comandos
end block nome_do_block;
```

Quadro 18 – Construção "*block*"

7.7.2.6. **"*process*"**

Um processo define uma região de comando seqüencial, e segue o formato apresentado no quadro 19, o processo pode ter uma lista de sensibilidade, assim caso ocorra uma mudança no valor de um sinal contido nesta lista o processo é executado.

```
nome_do_processo: process (lista_de_sensibilidade)
     --Região de declarações:
          --definição de tipos de dados locais
          --declaração de variáveis e constantes
          --declaração de pseudônimo
--Obs.: não é permitido declarar sinais e variáveis compartilhadas
begin
     --comandos sequenciais
end process nome_do_processo;
```

Quadro 19 – Construção "*process*"

7.7.2.7. **"*unaffected*"**

A palavra reservada "*unaffected*" é utilizada quando não se deseja alterar o valor de um sinal, ela só é permitida em atribuições concorrentes, e nem todas as ferramentas de síntese tem suporte a ele. No quadro 20 é apresentado um exemplo de utilização.

```
with expressao select
sinal_destino <=    expressao_1 when condicao_1,
                expressao_3 when condicao_2,
                --valor do sinal não é alterado
                -- nas demais condições
                unaffected when others;
```

Quadro 20 – Construção "*process*"

7.7.2.8. **"*generate*"**

O comando "*generate*" permite a repetição de comandos concorrentes dentro dos esquemas de geração "*FOR*" e "*IF*". Esse comando é utilizando quando a geração automática de circuitos que possuem uma regularidade em sua estrutura, ou podem ser definidas a partir de um conjunto de células interconectadas segundo um padrão, um exemplo de aplicação seria em unidades lógicas e aritméticas, somadores e multiplicadores.

- **“*for*”**

O esquema de geração “*for*” possui um identificador local bem como os limites de geração. Não é necessária a declaração do identificador, os limites da geração são definidos dentro de um intervalo de valores inteiros ou enumerados e o tipo do identificador é determinado de acordo com a faixa de valores. A sua sintaxe está descrita no quadro 21.

```
nome_do_esquema: for identificador in limites_de_geracao generate
      -- comandos concorrente
end generate nome_do_ esquema;
```

Quadro 21 – Comando *for*

- **“*if generate*”**

O esquema “*if generate*” é normalmente utilizado no tratamento de casos de exceção dentro de uma estrutura, como as que ocorrem nos limites. Este esquema introduz uma réplica dos comandos contidos, após o “*if*”, caso a condição seja satisfeita. A sua sintaxe está descrita no quadro 22.

```
nome_do_esquema: if condição generate
     -- comando concorrente
     -- comando concorrente
end generate nome_do_ esquema;
```

Quadro 22 – Comando *if*

7.7.3. Comandos seqüenciais

Os comandos seqüenciais são utilizados dentro de regiões específicas de código como processos ou subprogramas. Quando uma região de código seqüencial é iniciada, os comandos são executados na seqüência em que se encontram no código, e não dependem da troca de valor nos sinais ou variáveis. Um exemplo de aplicação de comandos pode ser vista no quadro 23.

```
nome_do_esquema: process
begin
     comando_1
     comando_2
          •
          •
     comando_n
end process nome_do_esquema;
```

Quadro 23 – Comando seqüenciais

7.7.3.1. **Atribuição de sinais em regiões seqüenciais**

A atribuição incondicional de valor para um sinal pode ocorrer dentro de regiões de código concorrente ou seqüencial. Dentro de uma região de código concorrente, a atribuição de sinal ocorre no momento da alteração de valor de um sinal da expressão. Em uma região de código seqüencial, como um processo, por exemplo, a atribuição é agendada quando o comando for executado.

7.7.3.2. **Variáveis**

Variáveis são objetos utilizados em regiões de código seqüencial, e a sua declaração ocorre dentro dessas regiões. Um sinal é empregado na transferência de um valor armazenado em uma variável para região de código concorrente. Ao contrário do que ocorrem com os sinais, os valores das variáveis são atribuídos imediatamente após a avaliação do comando.

7.7.3.3. **"*if else*"**

A construção "*if else*" é empregada na execução condicional de uma ou mais comandos seqüenciais, de acordo com uma lista de condições. Estas condições são avaliadas na ordem em que são apresentadas no código.

O comando "*if*" pode ser seguido por cláusulas "*elsif*", que por sua vez, podem conter condições a serem analisadas. Caso nenhuma condição verdadeira não seja encontrada

e exista uma cláusula "*else*", então o conjunto de instruções que a segue é executado. Se não houver nenhuma condição verdadeira e não houver nenhuma cláusula "*else*" na construção, então nenhum comando é executado. O uso das cláusulas "*else*" e "*elsif*" são opcionais, e a quantidade de cláusulas "*elsif*" que podem ser utilizadas é ilimitada. A sua sintaxe está descrita no quadro 24.

```
if condição_1 then
     if condição_2 then
          comandos _sequenciais;
     end if;
elseif condição_3 then        -- elsif opcional
     comandos _sequenciais;
else                  -- else opcional
     comandos _sequenciais;
end if;
```

Quadro 24 – Comando *if else* e *elsif*

7.7.3.4. ***"case when"***

A construção "*case when*" é utilizada na execução condicional de um ou mais comandos seqüenciais, de acordo como valor de uma expressão. Cada condição define se um determinado conjunto de comandos é executado ou não.

As condições devem ser mutuamente exclusivas, ou seja, não deverá haver mais de uma condição verdadeira na relação de valores que expressão pode assumir e estas condições devem possuir a mesma prioridade.

É possível utilizar o delimitador "|" para agrupar mais de uma condição, caso a execução de um mesmo conjunto de comandos esteja condicionada a mais de uma condição. O seu uso equivale a uma operação "ou" entre as condições de escolha. As palavras reservadas "*to*" e "*downto*" podem ser empregadas para delimitar uma faixa de condições. A palavra reservada "*others*" pode ser empregada na última condição para agrupar condições que não estejam relacionadas na lista. A sua sintaxe está descrita no quadro 25.

```
case expressao_de_escolha is
     when condição_1     =>
               comandos_sequenciais;
     when condição_2 | condição_3       =>
               comandos_sequenciais;
     when condição_4 to  condição_8     =>
               comandos_sequenciais;
     when others         =>
               comandos_sequenciais;
end case;
```

Quadro 25 – Comando *case when*

7.7.3.5. ***"for loop"***

O esquema de iteração do comando "*for loop*" se assemelha ao esquema de geração do mesmo nome do comando "*generate*". Ele possui um identificador local e os limites da iteração, sendo que o identificador não precisa ser declarado e os limites são definidos por uma faixa discreta de valores. O tipo do identificador é determinado pela faixa de valores e nenhuma operação dentro do laço pode atribuir um valor ao identificador. A sua sintaxe está descrita no quadro 26.

```
nome_do_laco: for  identificador  in limites_da_interacao loop
     -- comandos_sequenciais;
end  loop nome_do_laco;
```

Quadro 26 – Comando *for loop*

7.7.3.6. ***"wait"***

O comando "*wait*" suspende a execução de um processo ou de um procedimento. O seu uso está condicionada a três cláusulas ou a uma combinação delas, conforme o quadro 18.

O comando "*wait on*" provê um mecanismo equivalente à lista de sensibilidade de um processo, ou seja, após a execução do comando "*wait*", o processo fica suspenso até a alteração de valor em um dos sinais relacionados na lista. Utilizando o comando "*wait until*",

o processo fica suspenso enquanto a expressão booleana contida na expressão não for satisfeita. A sua sintaxe está descrita no quadro 27.

```
wait on lista_de_sensibilidade;
wait until expressao_boolena;
wait on lista_de_sensibilidade until condição for expressao_de_tempo;
wait;
```

Quadro 27 – Comando *wait*

7.7.3.7. **"*null*"**

O comando "*null*" não realiza nenhuma operação, ou seja, a execução é apenas passada para a próxima instrução quadro 28.

```
entity nome_da_entidade is
     port: (x     :    in bit_vector(2 downto 0);
          sel   :   in bit_vector(1 downto 0);
            s     :   out bit);
end nome_da_entidade;
architecture  nome_da_arquitetura  of nome_da_entidade is
begin
     nome_do_processo: process (x, sel)
       begin
          case sel is
               when "00" =>  s  <= x(0)   and  x(1);
               when "10" =>  s  <= x(2)   xor  x(0);
               when others    =>   null;
          end case;
     end process nome_do_processo;
end nome_da_arquitetura;
```

Quadro 28 – Comando *null*

7.7.4. Subprogramas

Um subprograma é um recurso para isolar um conjunto de comandos utilizados com freqüência, com objetivo de melhorar o entendimento da descrição. Esse conjunto de comandos pode realizar uma operação mais abstrata na descrição, como por exemplo descrever o comportamento de um circuito. Funções e procedimentos são subprogramas que possuem comandos seqüenciais, sendo que uma função retorna somente um único valor, enquanto um procedimento pode retornar vários valores.

Os subprogramas podem ser chamados em regiões de código concorrente ou seqüencial e é permitido também o uso de chamada recursiva, porém, nem sempre é aceita pelas ferramentas de síntese. Quando um subprograma é invocado, todos os objetos declarados localmente são inicializados e ao final da execução são descartados.

7.7.4.1. **Função**

A invocação de uma função é feita por uma expressão, e retorna um único valor. A classe dos objetos declarada na lista de parâmetros pode ser "*Constant*", "*Signal*" ou "*File*", sendo que o modo do objeto deve ser obrigatoriamente do tipo "*IN*". Não é permitido o uso da classe "*Variable*" e uma função não altera os parâmetros que são passados a ela.

O argumento de retorno é identificado pelo comando "*Return*", sendo que uma função pode conter mais de comando "*Return*". Atribuições de valores empregam sempre variáveis, pois não é possível declarar um sinal em um subprograma. Não é permitido o uso do comando "*Wait*" em uma função.

A chamada de uma função pode ocorrer dentro de regiões de código concorrente ou seqüencial. Chamadas recursivas são permitidas, porém, não são suportadas pelas ferramentas de síntese.

O quadro 29 ilustra a declaração e o corpo de uma função.

```
-- declaração da funcao
function nome_funcao      (signal    a : in       tipo_a;
                           signal    b :          tipo_b;
                           constant  c : in       tipo_c;
                                     d :          tipo_c)
                          return                  tipo_r;
-------------------------------------------------------------------------
-- corpo da função
function  nome_funcao     (signal    a : in       tipo_a;
                           signal    b :          tipo_b;  -- modo in
                           variable  e : inout    tipo_e);
                                     d :          tipo_c) – classe Constant, modo in
                          return                  tipo_r;
-- declaração de tipo, constante, variável
begin
-- região de código seqüencial
--
return expressão;
end nome_funcao;
```

Quadro 29 – Declaração de Função

7.7.4.2. **Procedimento**

Os procedimentos são chamados por um comando, e permitem o retorno de mais de um valor. A classe de objetos declarados na lista de parâmetros pode ser "*Constant*", "*File*", "*Signal*" ou "*Variable*", e são permitidos os modos "*IN*", "*OUT*" e "*INOUT*". Como os parâmetros formais podem ser do modo "*INOUT*", um procedimento pode mudar o valor dos parâmetros passados a ele. O "*RETURN*" não é obrigatório e causa o término da execução do procedimento.

É possível atribuir valor para sinal no interior de um procedimento. Isso pode ocorrer na atribuição de valores em parâmetros formais da classe sinal, nos modos "*OUT*" ou "*INOUT*".

A chamada de um procedimento pode ocorrer em regiões de código concorrente ou seqüencial. Na região de código concorrente, a execução ocorre na alteração de valor em um dos parâmetros de modo "*IN*" ou "*INOUT*". Na região de código seqüencial, o procedimento é tratado como um comando seqüencial.

No quadro 30 pode-se observar o exemplo de uma declaração e um corpo de um procedimento.

```
-- declaração do procedimento
procedure nome_procedimento  (signal      a : in       tipo_a;
                              variable    b : out      tipo_c;
                              variable    e : inout    tipo_e);
-- corpo do procedimento
procedure nome_procedimento (signal       a : in       tipo_a;
                              variable    b : out      tipo_c;
                              variable    e : inout    tipo_e);
--
-- declaração de tipo, constante, variável
begin
-- região de código seqüencial
--
end nome_procedimento;
```

Quadro 30 – Declaração de Procedimento

APÊNDICE B - Passo a passo de simulação utilizando o ModelSim

ModelSim XE III 6.0a

O Software ModelSim XE III 6.0a, é uma ferramenta de simulação da Xilinx, a seguir descreveremos os passo básicos necessário para realizar uma simulação.

Ao abrir o aplicativo será mostrado a janela apresentada na figura 42.

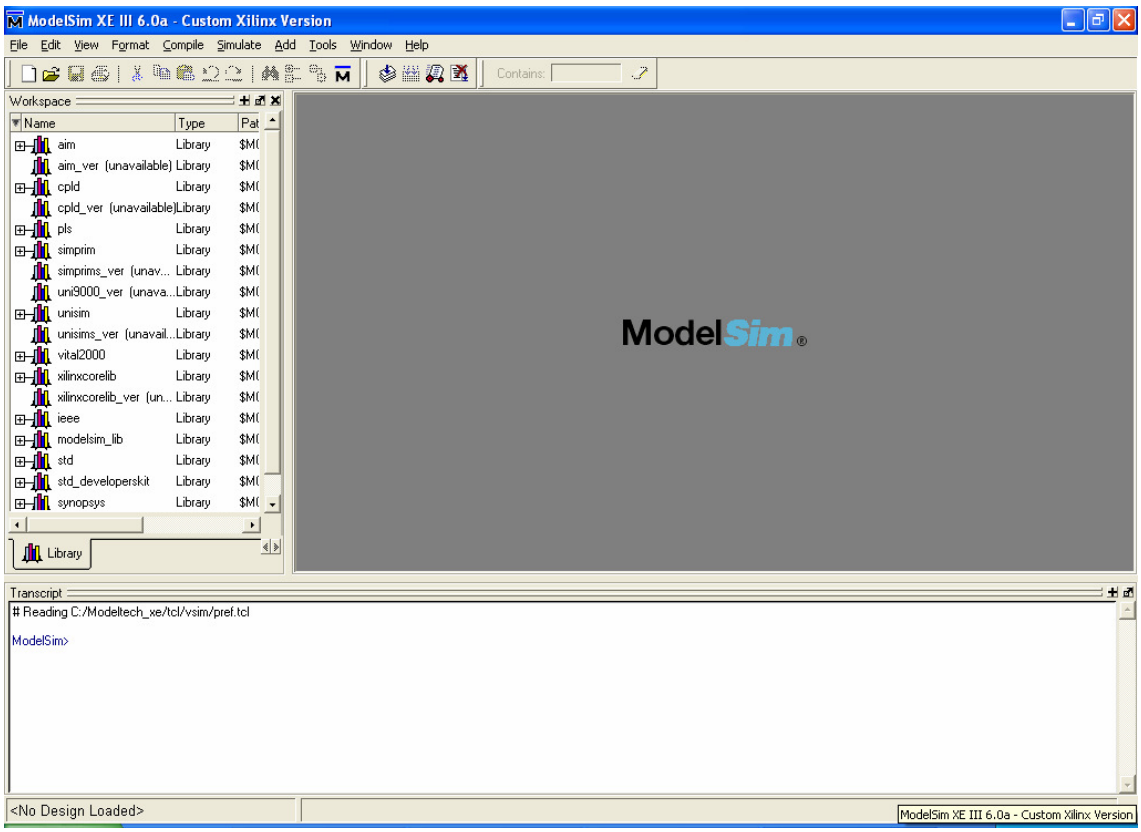


Figura 42 – Primeiro contato com a ferramenta

Criando um novo Projeto

O primeiro passo será criar um novo projeto, para tal siga a seqüência no menu: *File→New→Project...*.

Será aberta a janela "*Create Project*" que pode ser visualizada na figura 43.

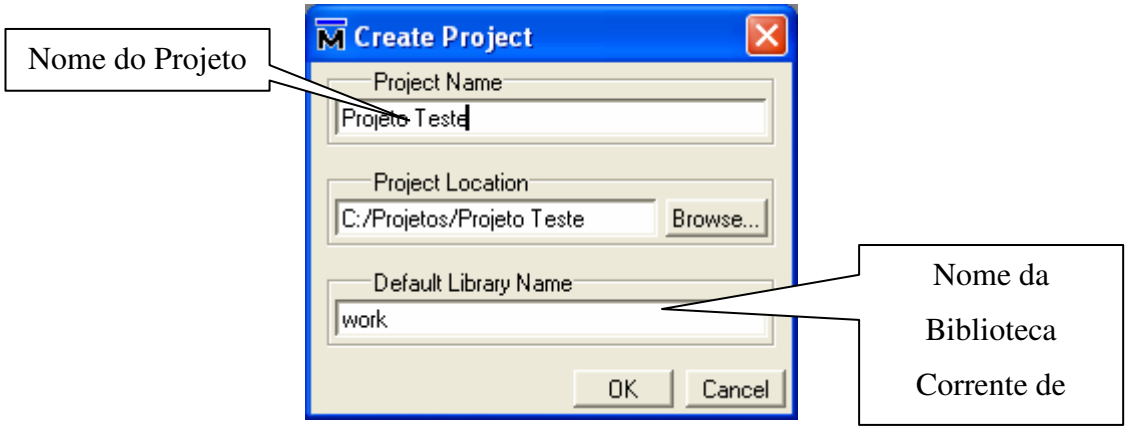


Figura 43 – Criando novo projeto

Será criado o projeto no caminho especificado e abrirá a janela da figura 44, por ela pode-se adicionar itens ao projeto, ou pelo caminho "*File→Add to Project→* (Opção)" onde a opção são as mesmas da figura 44.

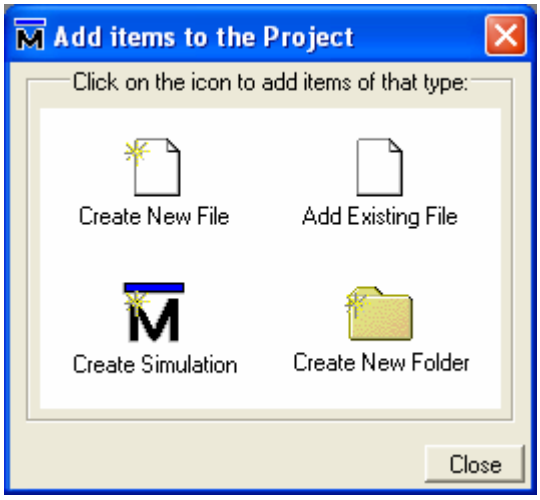


Figura 44 – *Add items to the Project*

Para adicionar um arquivo gerado através de outro *software*, basta clicar no item *"Add Existing File"* que será aberta janela correspondente a figura 45, e através do botão *"Browse..."* é possível selecionar tal arquivo.

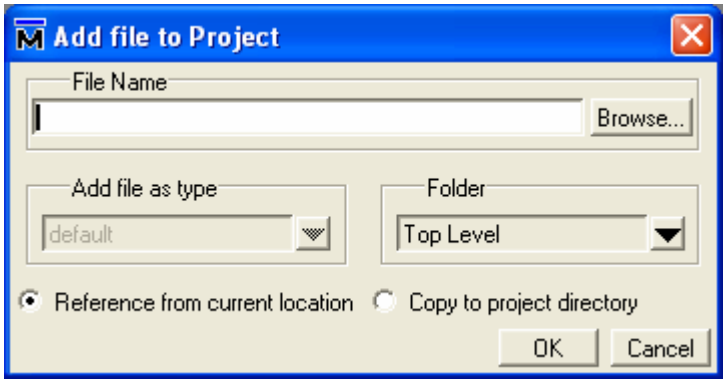


Figura 45 – *Add file to the Project*

Compilando

Selecione o arquivo que se deseja compilar na aba do projeto, com o botão direito do mouse selecione a opção *Compile*, nela contem o item para compilar todos os arquivos do projeto (conforme figura 46) ou o selecionado. Ou pode-se compilar um arquivo selecionado clicando no ícone , e todos pelo .

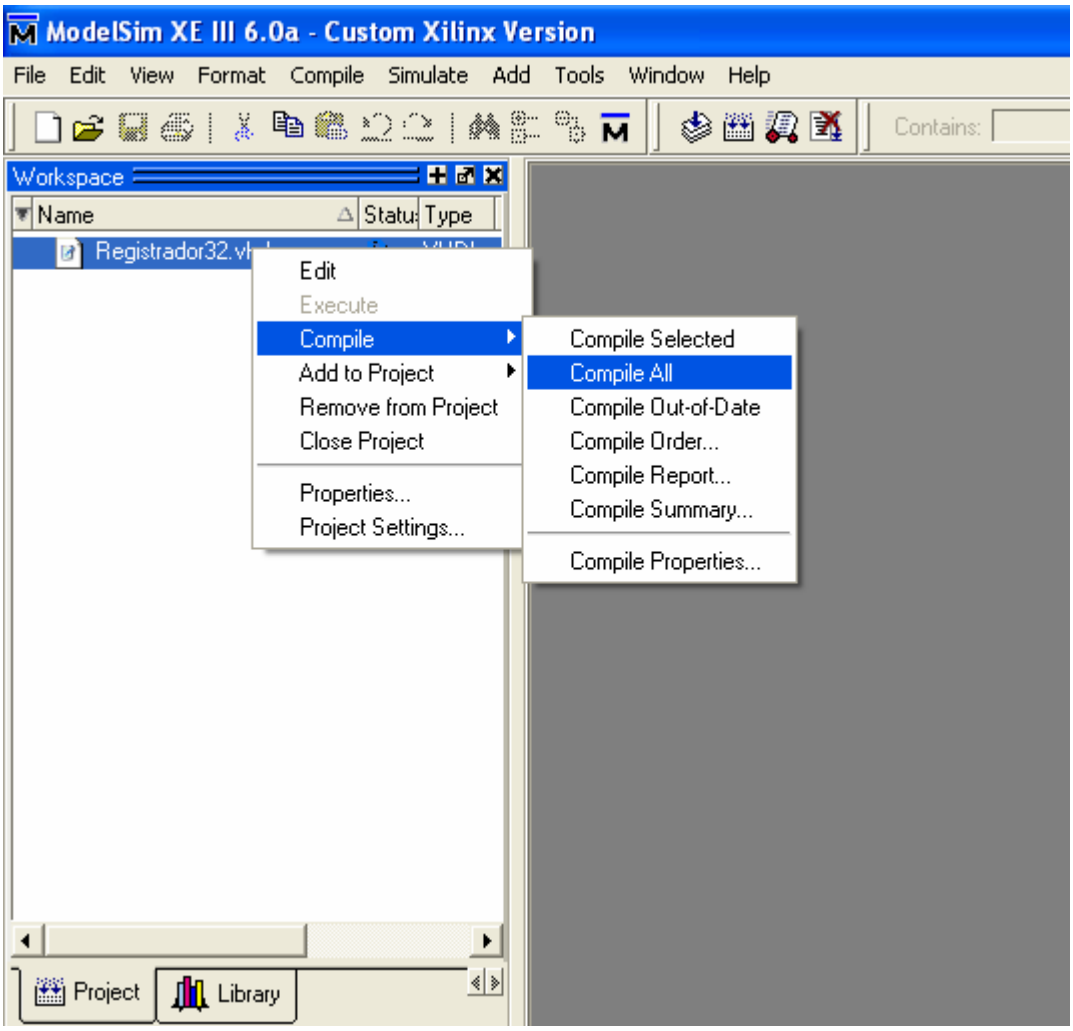


Figura 46 – Compilando arquivos

Simulando

Após compilar o projeto pode-se iniciar a simulação, através do ícone , ou pela aba de bibliotecas, na biblioteca corrente de trabalho, abrindo o arquivo que se deseja simular, conforme figura 47.

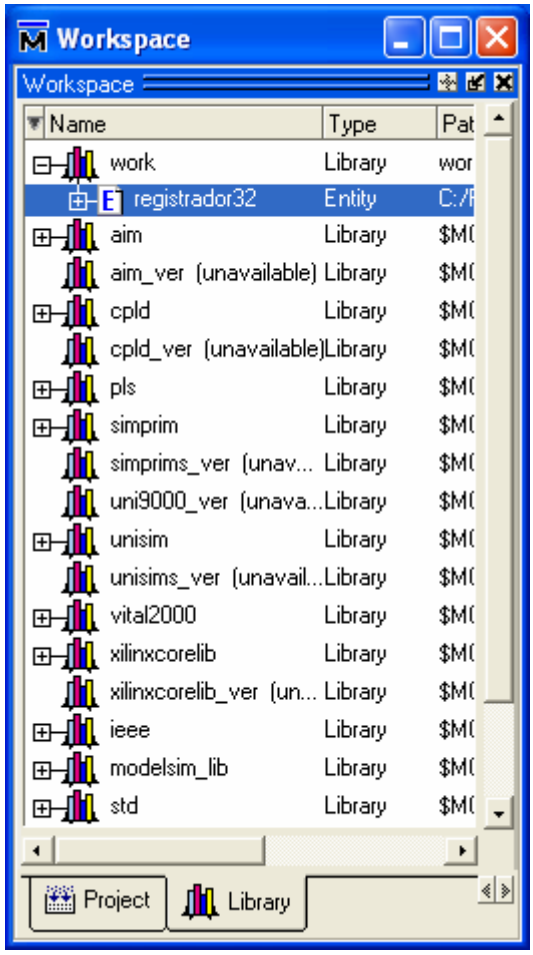


Figura 47 – Abrir Simulação

Para iniciar a simulação selecione o componente, aperte o botão direito, clique na opção *"Add→Add to Wave"* conforme figura 48, será aberta a janela *Wave* (figura 49) onde é realizada a simulação.

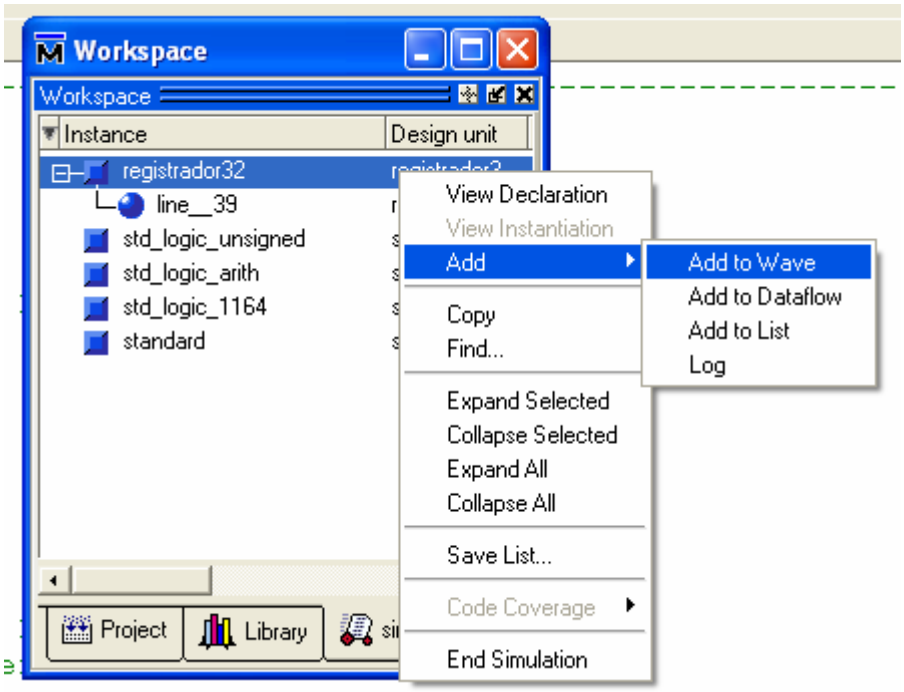


Figura 48 – Iniciar Simulação

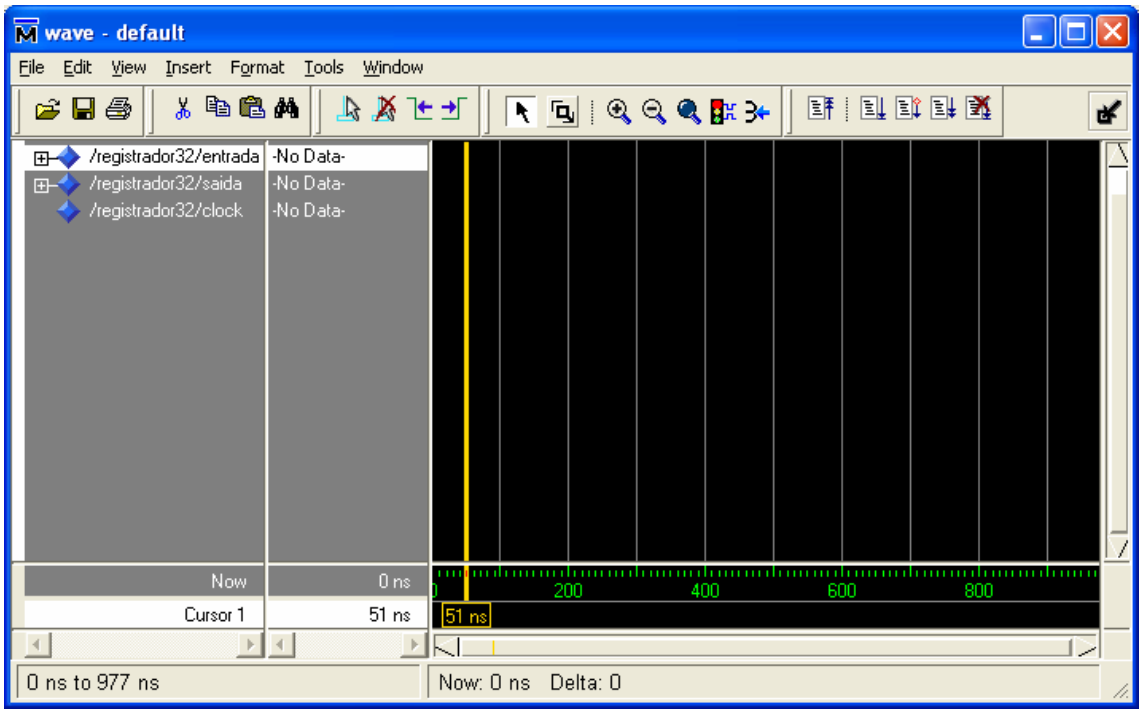


Figura 49 – Janela de simulação *Wave*

Nomear entradas/saídas

Com uma das entradas/saídas selecionadas, aperte botão direito do mouse e clique na opção *"Properties..."* abrirá janela conforme figura 50.

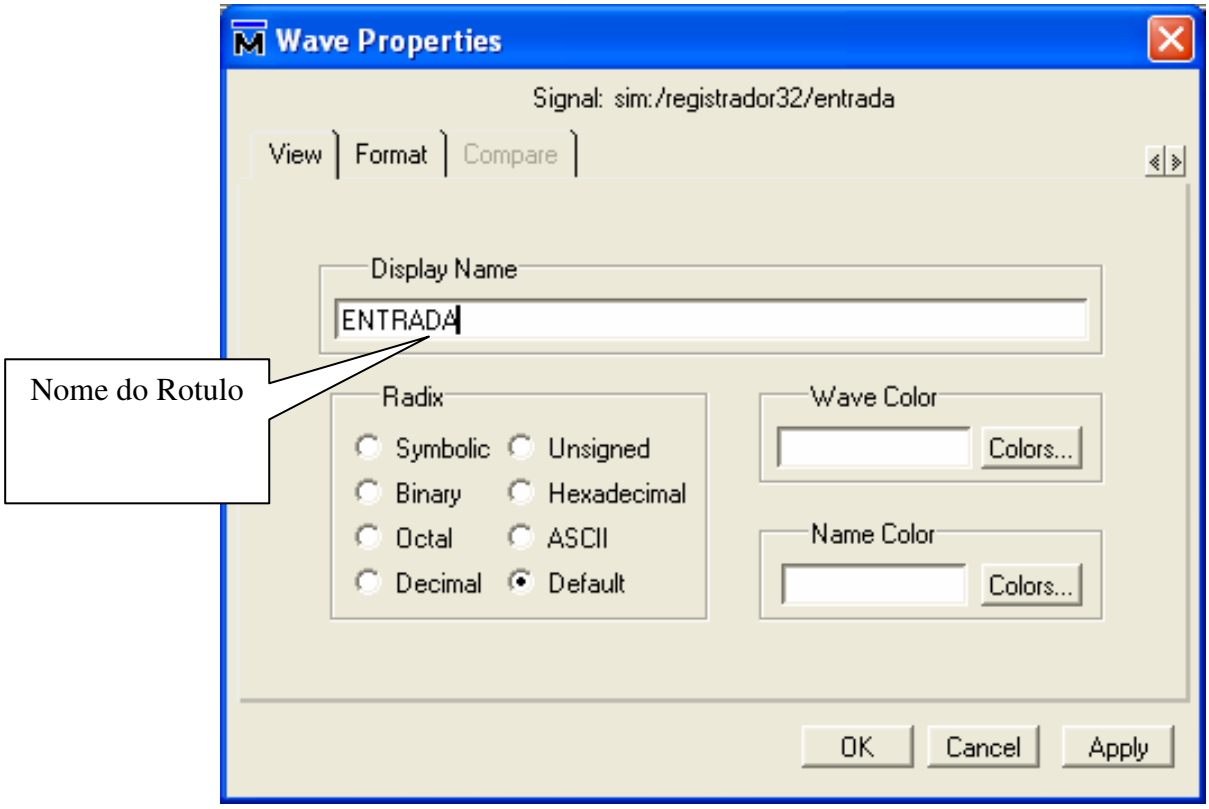


Figura 50 – *Wave Properties*

Selecionar valor das entradas

Com uma das entradas/saídas selecionadas, aperte botão direito do mouse e clique na opção *"Force..."* abrirá janela conforme figura 51. Este valor pode ser alterado durante a simulação.

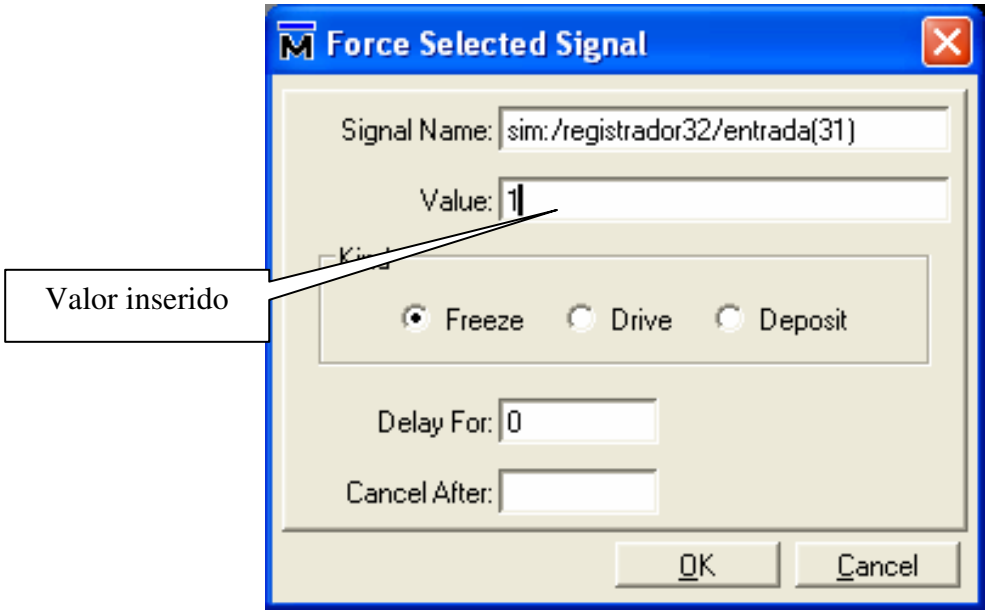


Figura 51 – Selecionar valor de entrada

Criar Clock

Com uma das entradas/saídas selecionadas, aperte botão direito do mouse e clique na opção *"Clock..."* abrirá janela conforme figura 52. Este valor pode ser alterado durante a simulação.

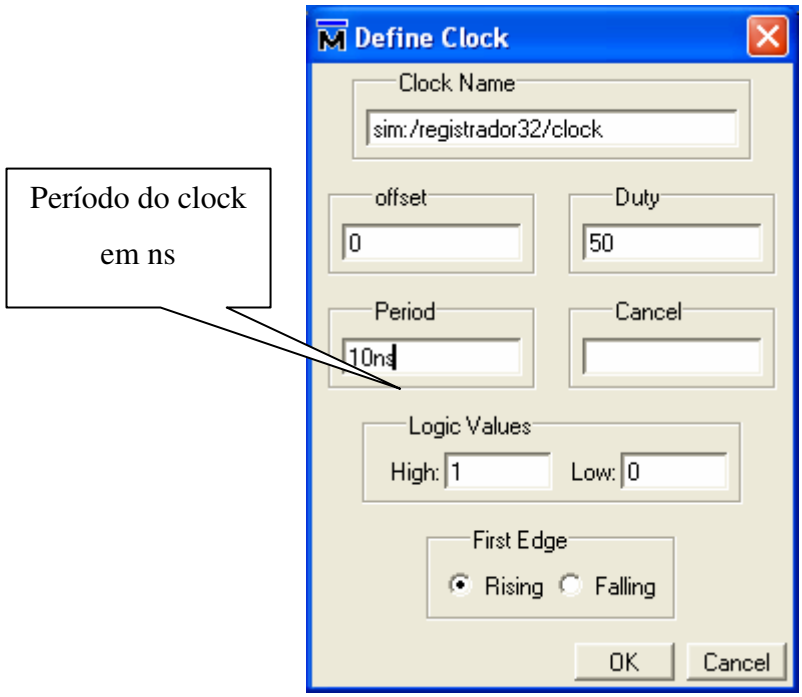


Figura 52 – Definir *clock*

Passos de simulação

Descrição dos recursos básicos para efetuar a simulação:

- 5 ns ***Run Length*:** Define o valor de cada passo de simulação.
- ***Zoom In*:** Aproxima a visualização da barra de tempo da simulação, de forma que possibilita visualizar intervalos de tempo menores.
- ***Zoom Out*:** Afasta a visualização da barra de tempo da simulação, de forma que possibilita visualizar intervalos de tempo maiores.
- ***Run*:** Executa cada passo da simulação.

Antes de iniciar a simulação deve ser configurado o tempo de cada passo, os valores das entradas, e *clock's* se necessário. Configurar a visualização dos valores no tempo de acordo com a necessidade, observe um exemplo da figura 53, do teste realizado com um registrador de 4 bits.

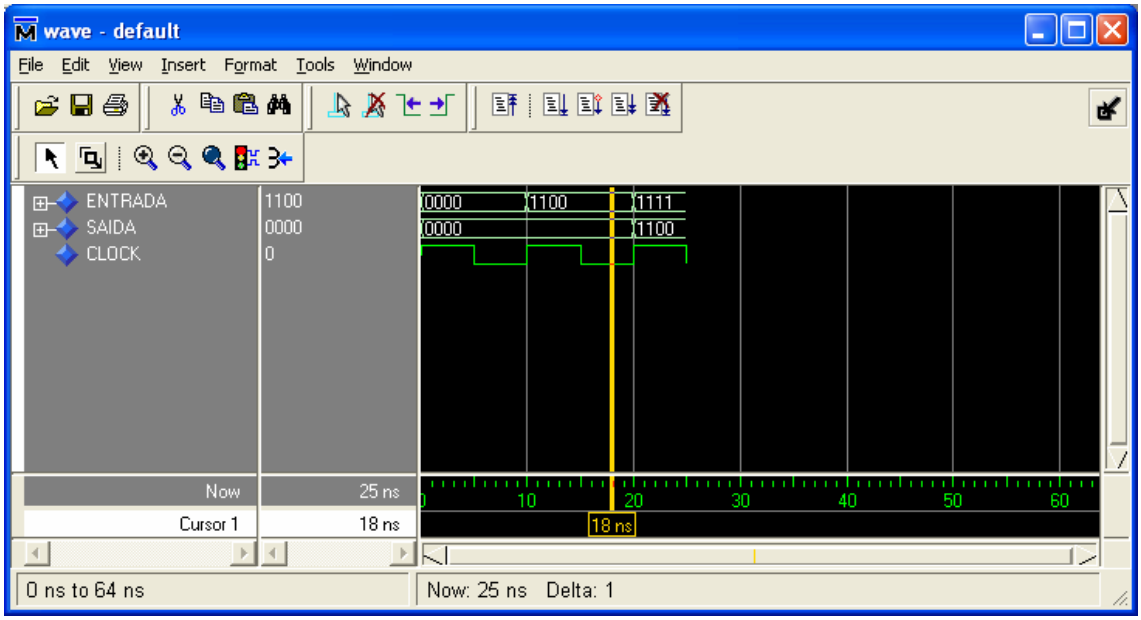


Figura 53 – Exemplo de simulação

Finalizar

Para finalizar a simulação selecione o componente, aperte o botão direito, clique na opção *"End Simulation"* conforme figura 54.

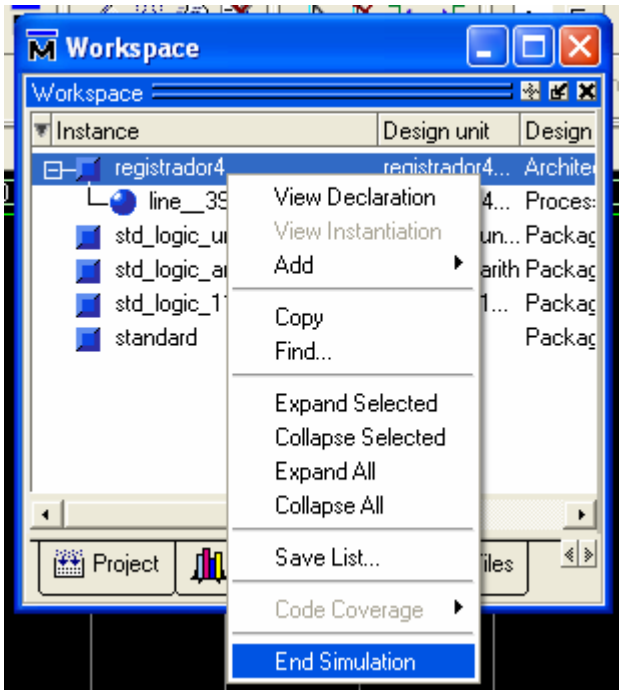


Figura 54 – Finalizar simulação

APÊNDICE C – Passo a passo da criação de projeto no ISE

ISE

O Software Xilinx ISE 7.1i, é um ambiente de desenvolvimento integrado da Xilinx, a seguir descreveremos os passos básicos necessários para construir um projeto nele.

Ao abrir o aplicativo será mostrada a janela apresentada na figura 55.

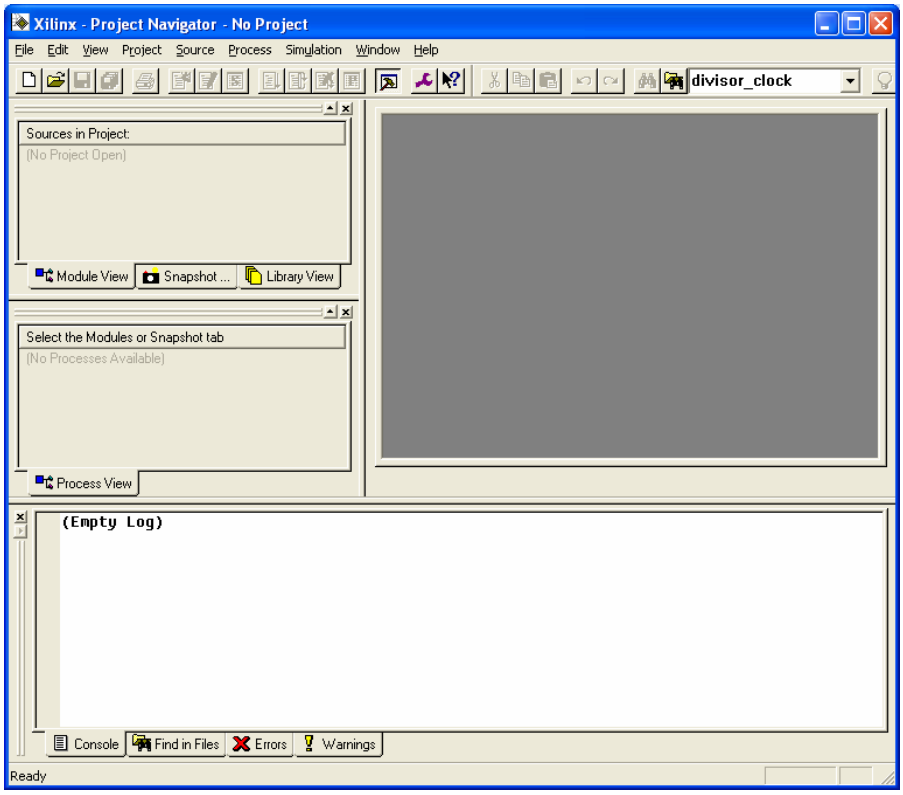


Figura 55 – Primeiro contato com a ferramenta

Criando um novo Projeto

O primeiro passo será criar um novo projeto, para tal siga a seqüência no menu: *File→New Project...*.

Será aberta a janela "*New Project*" que pode ser visualizada na figura 56.

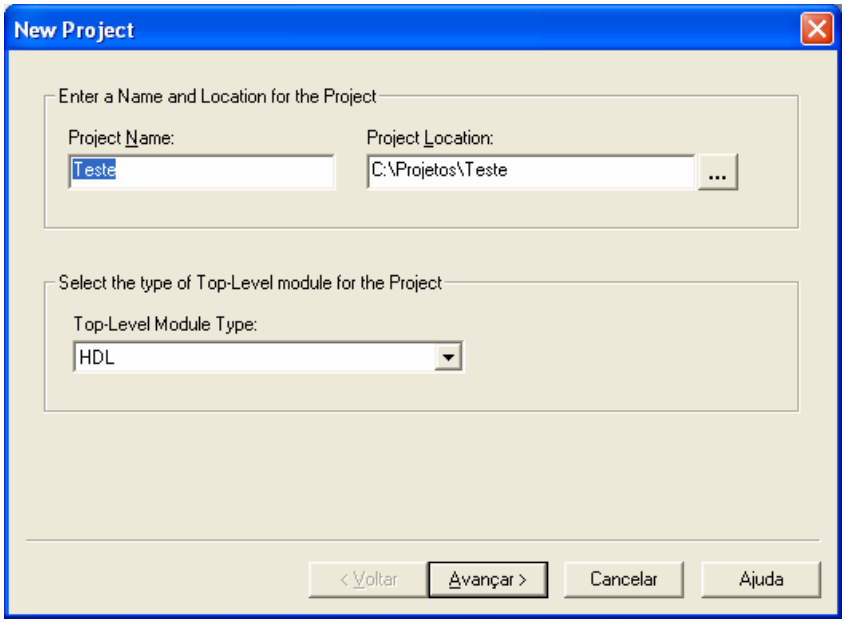


Figura 56 – Criando novo projeto

Será criado o projeto no caminho especificado, após apresentado a figura 56, em seguida irá surgir a janela da figura 57, onde são configurados os parâmetros do projeto, por exemplo, qual FPGA será utilizada e o simulador.

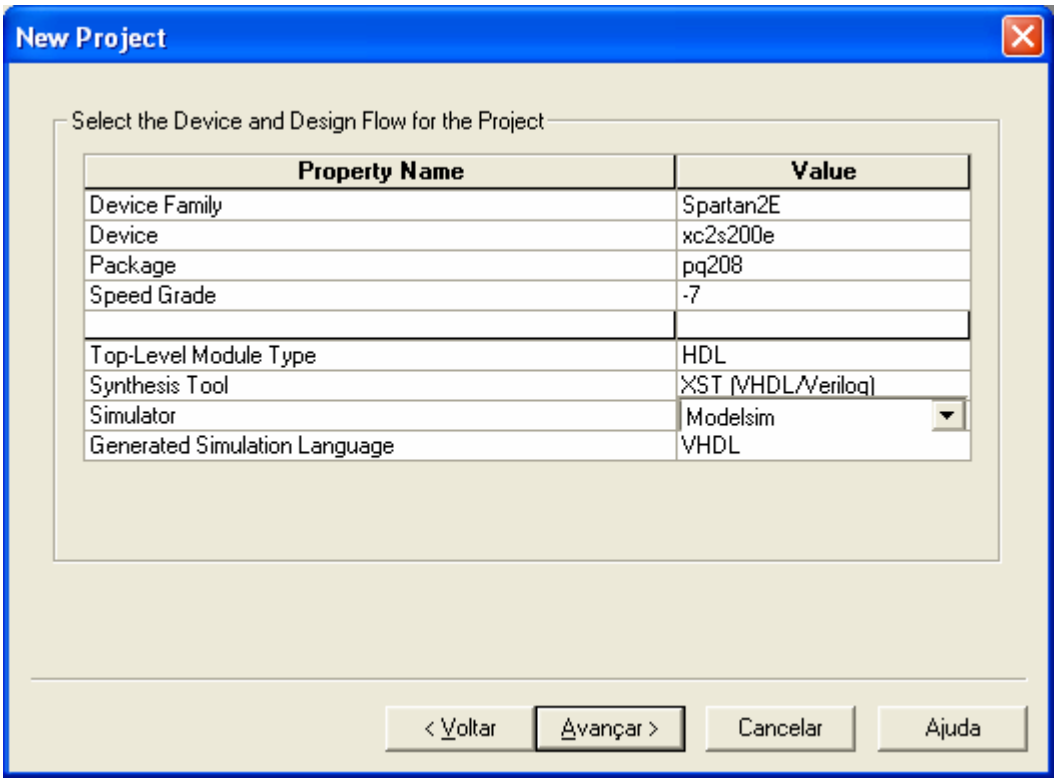


Figura 57 – Configurando o Projeto

Uma vez configurado o projeto abrirá uma janela na qual pode-se criar entidades novas entidades para o projeto através do botão *New Source*, caso seja necessário criar um item sugirá uma janela como a descrita na figura 58.

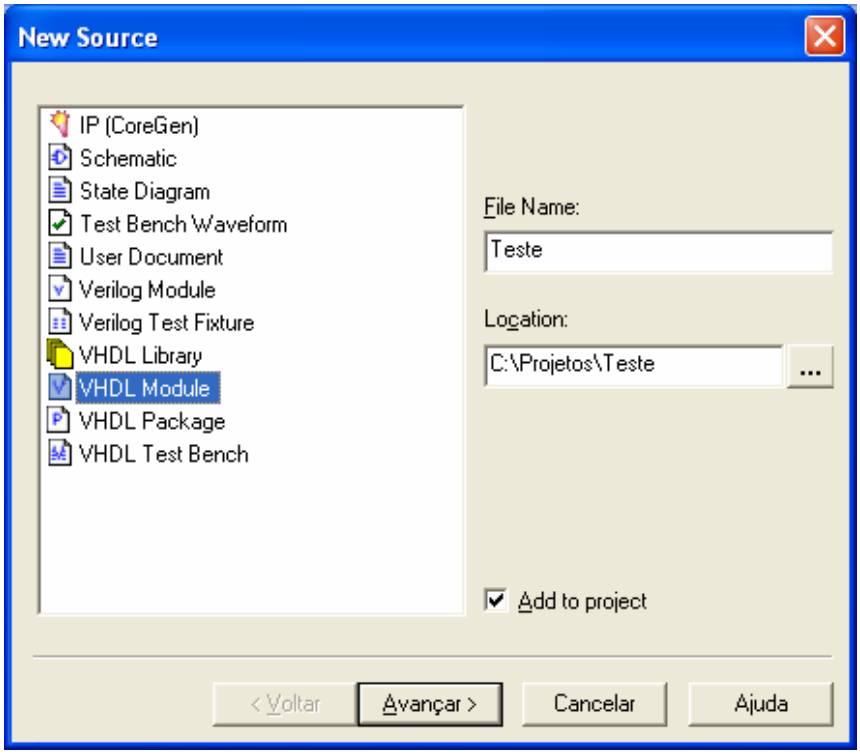


Figura 58 – *New Source*

Por exemplo, para adicionar um módulo em VHDL basta selecionar sua opção à esquerda e definir um nome para o arquivo. Após clicar no botão avançar sugirá a janela da figura 59, na qual pode-se definir a interface de entrada e saída da entidade a ser criada.

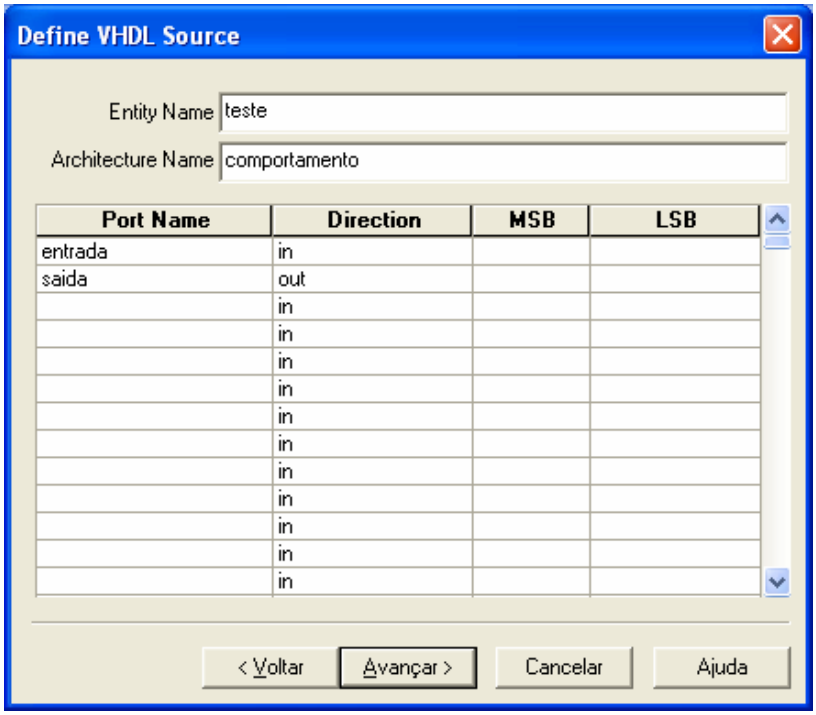


Figura 59 – *Define VHDL Source*

Uma vez definido as interfaces de entrada e saída aparecerá uma janela com as informações da entidade a ser criada bastando clicar no botão concluir. Clicando em voltar é possível adicionar novos itens ao projeto, se clicar em Avançar surgirá uma janela na qual é possível adicionar itens já existentes ao projeto. Avançando a janela de adicionar itens existentes ao projeto surgirá uma janela com informações do projeto, descrito na figura 60, bastando clicar em concluir para que o projeto seja criado.

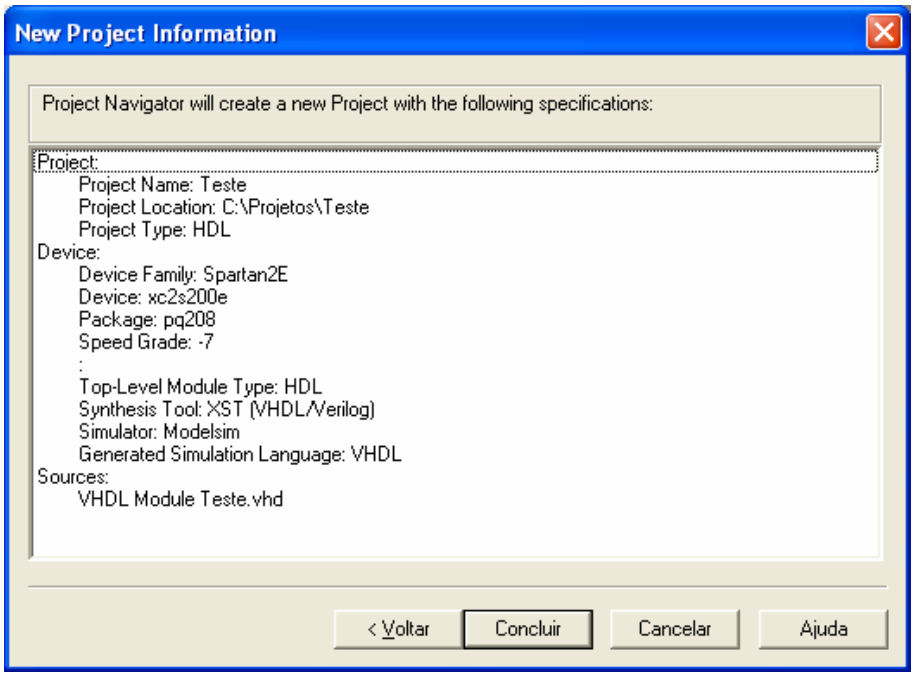


Figura 60 – *New Project Information*

Após o projeto ser criado aparecerá uma janela como da figura 61.

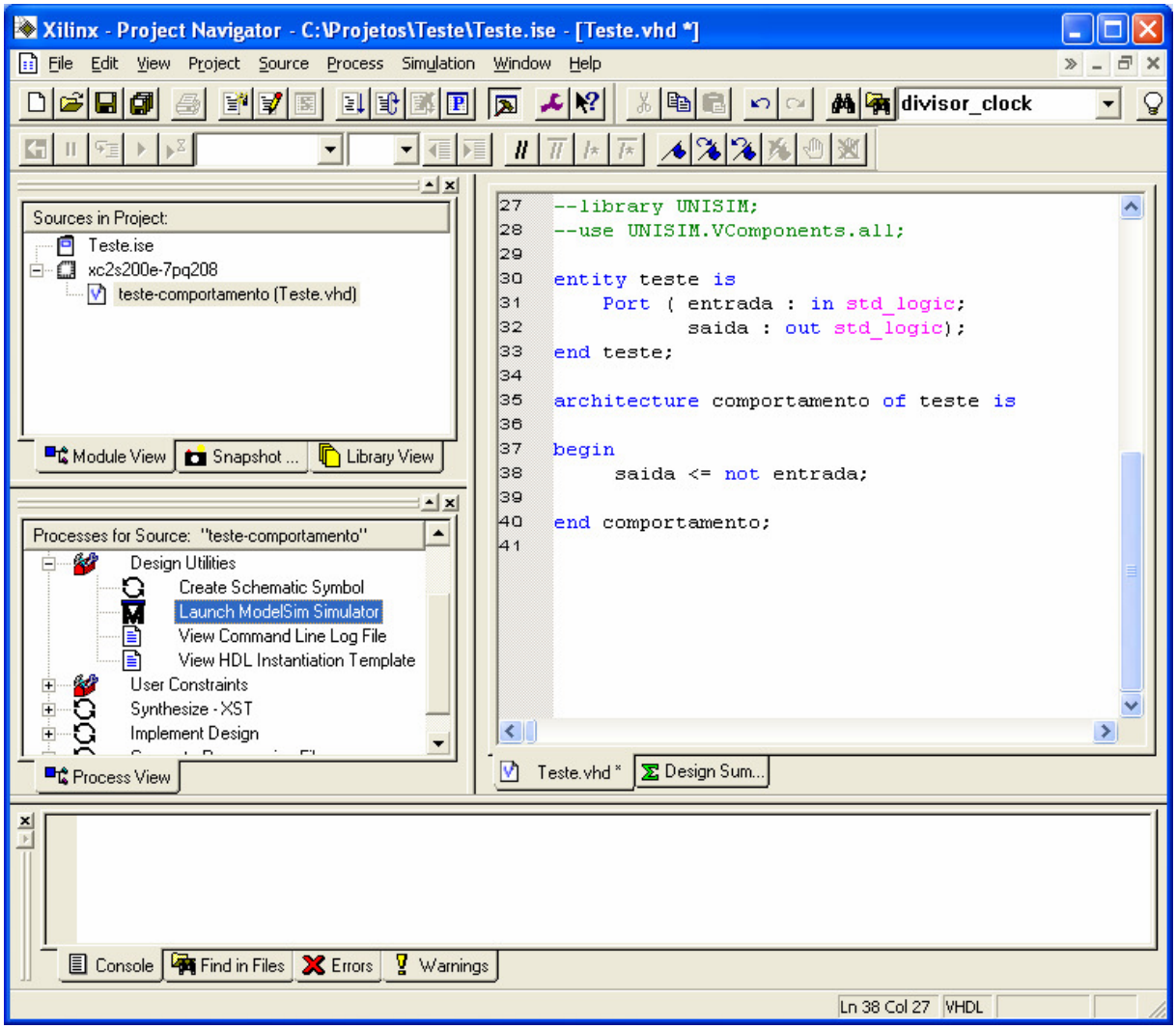


Figura 61 – *Add file to the Project*

Na aba *Sources in Project* dentro do *Module View* encontram-se as entidades do projeto. Enquanto no *Processess for Source* encontram-se os processos do item selecionado no *Sources in Project*, através dessa aba pode-se sintetizar o código em VHDL, ou abrir o simulador conforme mostra a figura 61, na qual *está* selecionado o simulador bastando um duplo clique para abri-lo ou clicando no ícone com o botão direto do mouse e selecionando a opção *Run*. O mesmo vale para as demais opções desta aba.

Através do ISE pode-se definir quais pinos serão utilizados na FPGA editando um arquivo de texto com a extensão .UCF, através do ícone *Edit Constraints* localizado na aba *Processess for Source* → *User Constraints* . Estes passos podem ser observados na figura 62.

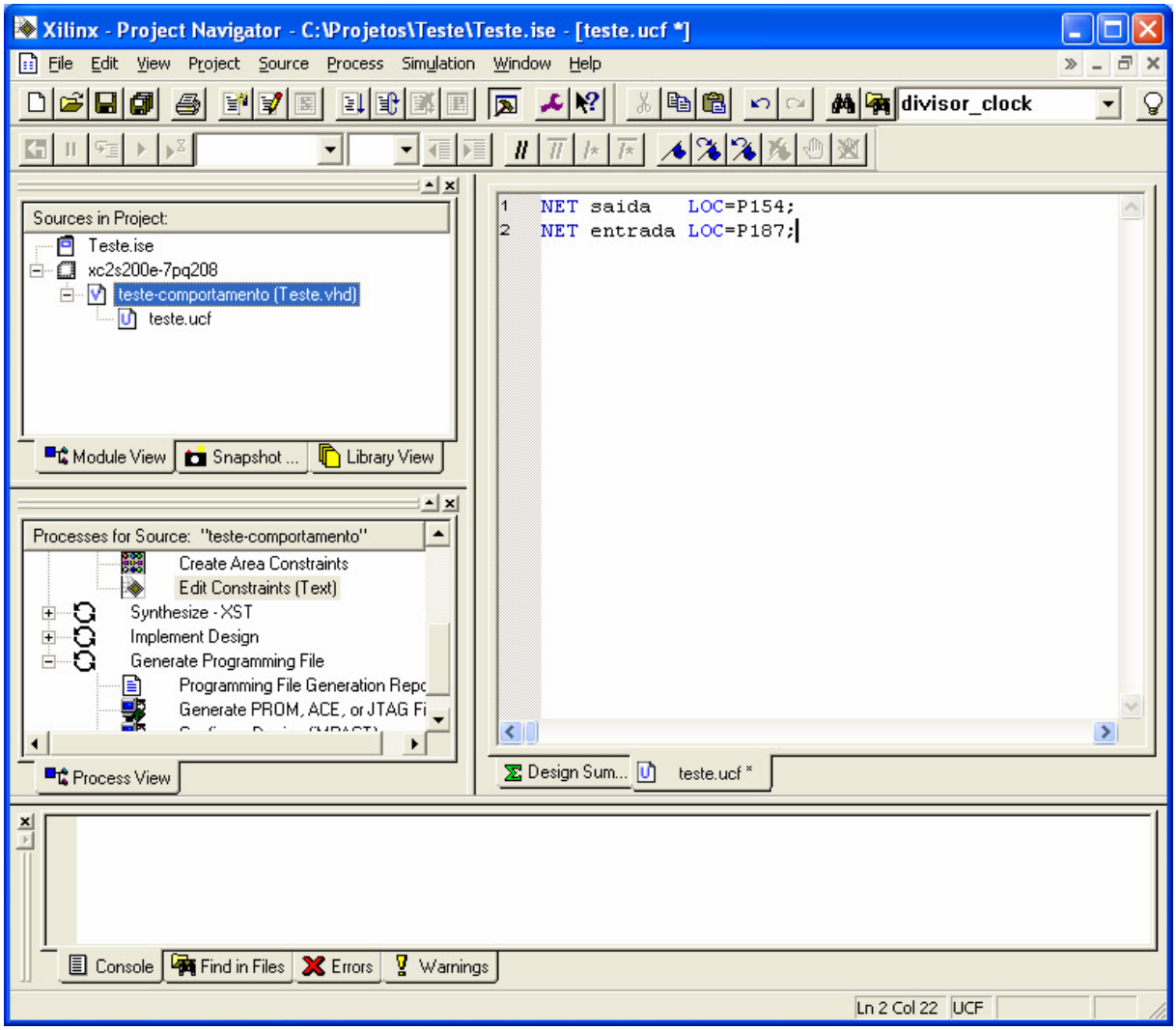


Figura 62 – Definindo os pinos da FPGA

Depois de realizar a programação da entidade e configurar os pinos da FPGA pode-se gerar o arquivo .BIT para transferência para FPGA, a síntese pode ser feita antes da geração do arquivo .BIT com duplo clique em *Synthesize – XST* com este processo pode-se observar se há algum erro na programação, e como pode ser observado na figura 63 as informações da síntese são descritas na aba *Console*.

Através do item *Generate Programming File* é possível gerar o arquivo .BIT e abrir o *iMPACT*, através do ícone *Configure Device (iMPACT)*.

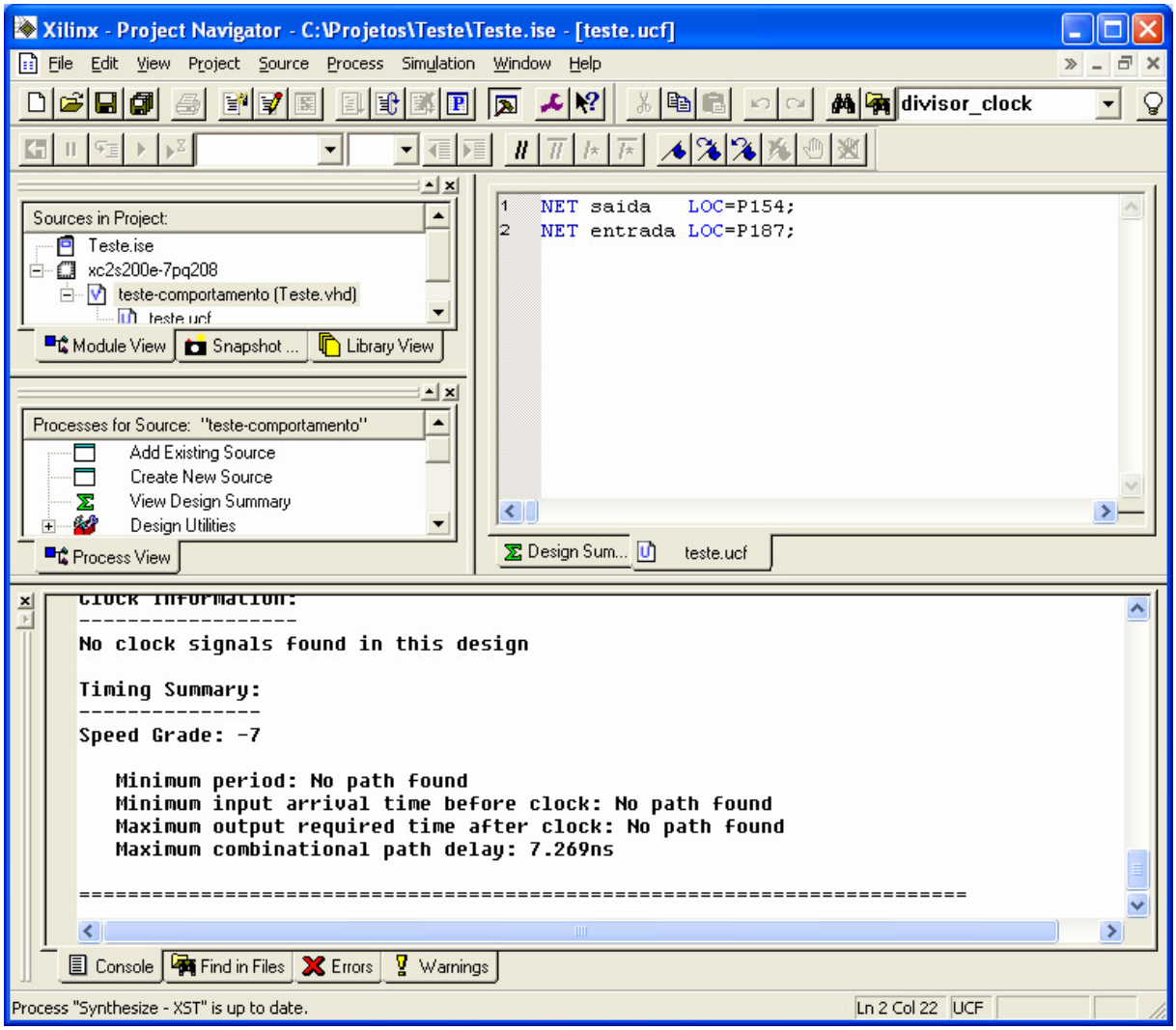


Figura 63 – Executando o *Synthesize*

Caso seja necessário transferir o arquivo .BIT do projeto para a FPGA isso ocorrerá através do *software iMPACT,* sua interface é descrita na figura 64, abrindo-o através do ISE a sua configuração ocorrerá automaticamente, sendo necessário apenas selecionar qual arquivo será transferido para FPGA, após selecionar o arquivo clicando com o botão direito do mouse na figura do FPGA aparecerá a opção *Program...* , através da qual é realizada a transferência.

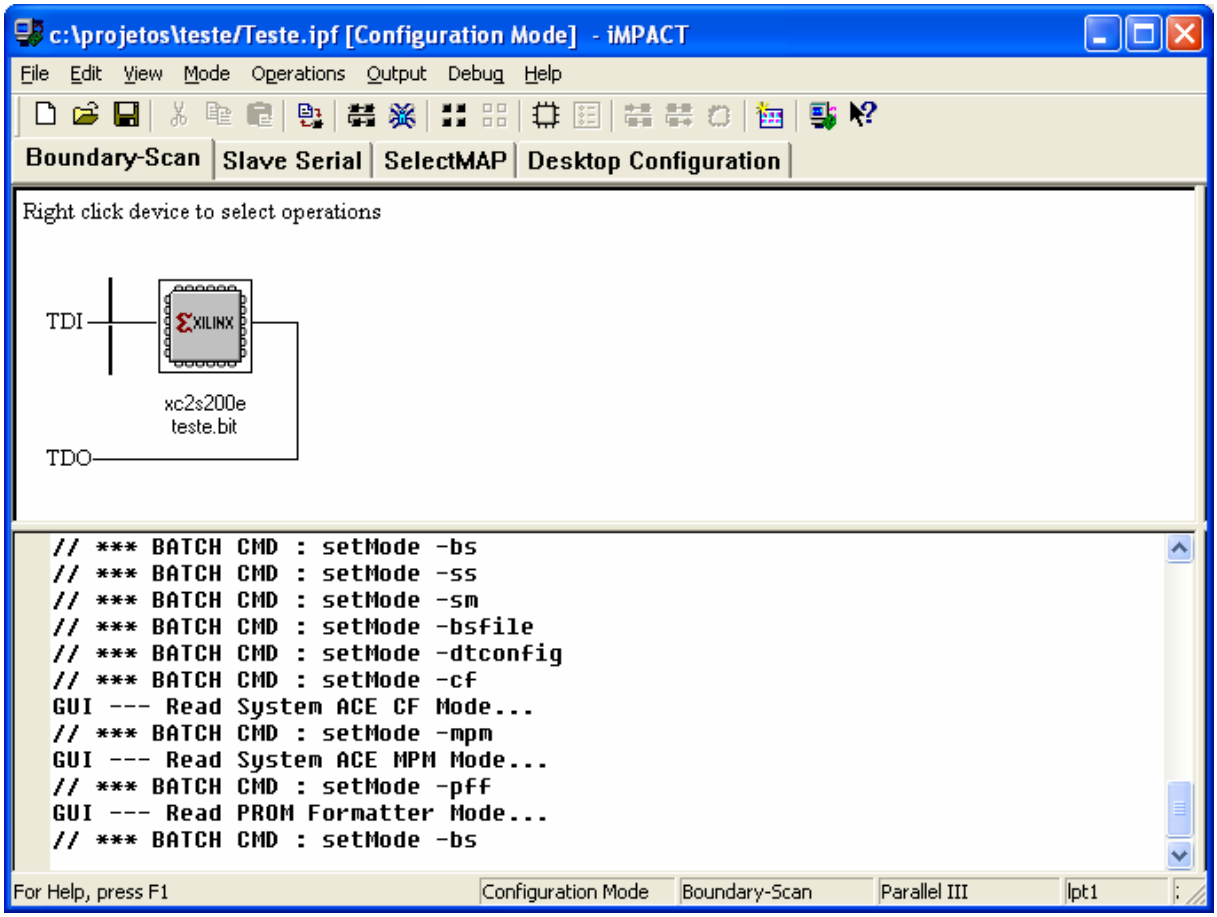


Figura 64 – Programa para transferência de arquivos .Bit *iMPACT*

APÊNDICE D – Testes e Validação

Neste apêndice são demonstrados os testes efetuados nos módulos que juntos compõem o microcontrolador desenvolvido neste trabalho.

Os módulos avaliados foram divididos nos seguintes grupos:

- Memórias;
- *Multiplex* e *Demultiplex*;
- Registradores e Contadores
- ULA;

Todos os testes de simulação foram realizados no programa ModelSim XE III. Os módulos foram testados individualmente demonstrando que o funcionamento ocorre dentro de suas respectivas especificações.

Memórias

Os módulos de memória possuem um tamanho de palavra de 32 bits e estão divididas em duas partes distintas, memória RAM e memória ROM, cuja função é armazenar os dados de programa e o programa propriamente dito, respectivamente. A memória RAM assim como a memória ROM possuem capacidade de 64 palavras, cada uma, totalizando 128 palavras.

Memória RAM

Escrita

Na figura 65 pode-se observar que com o sinal de controle leitura_escrita igual a ‘1’ o valor binário “1010101” inserido previamente na porta de entrada foi gravado na posição especificada na porta de endereço neste caso a posição zero, após a ocorrência da borda de subida do sinal de *clock*.

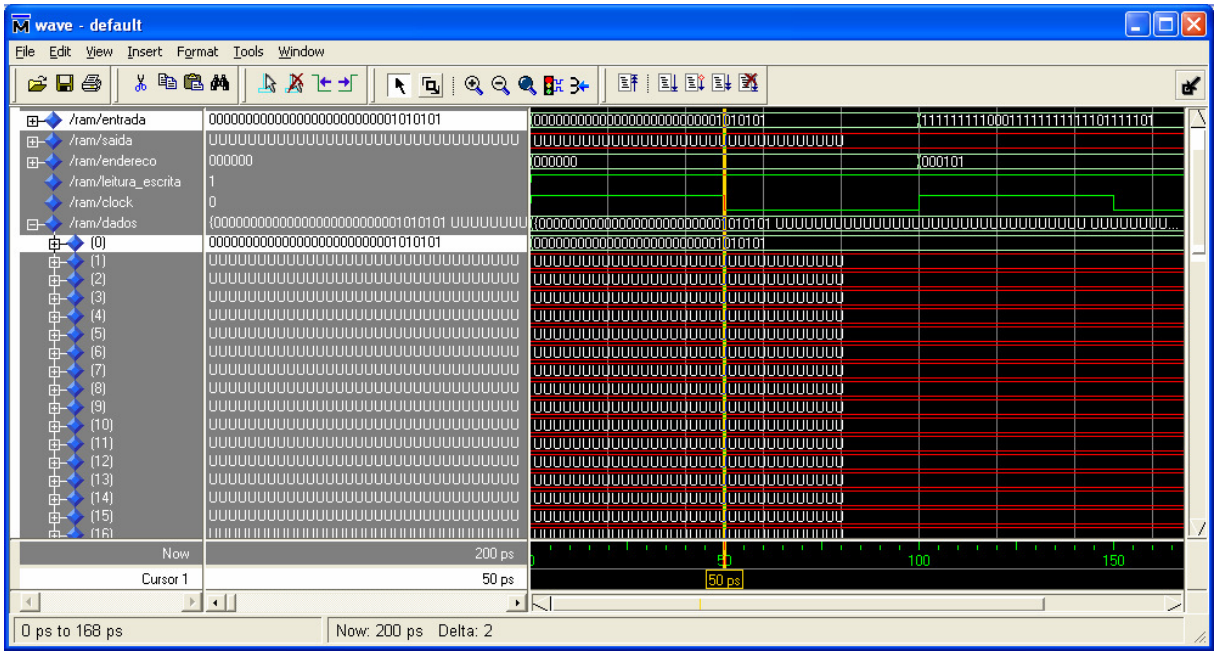


Figura 65 – Escrita Memória RAM

Leitura

A figura 66 mostra a leitura da posição zero da memória, pode-se observar que o valor contido na porta de saída e a posição zero são idênticos, para tal foi inserido o valor do endereço, que se deseja efetuar leitura, e indicado que a operação seria de leitura deixando o sinal leitura_escrita em nível baixo, a operação é concluída após a borda de subida do *clock*.

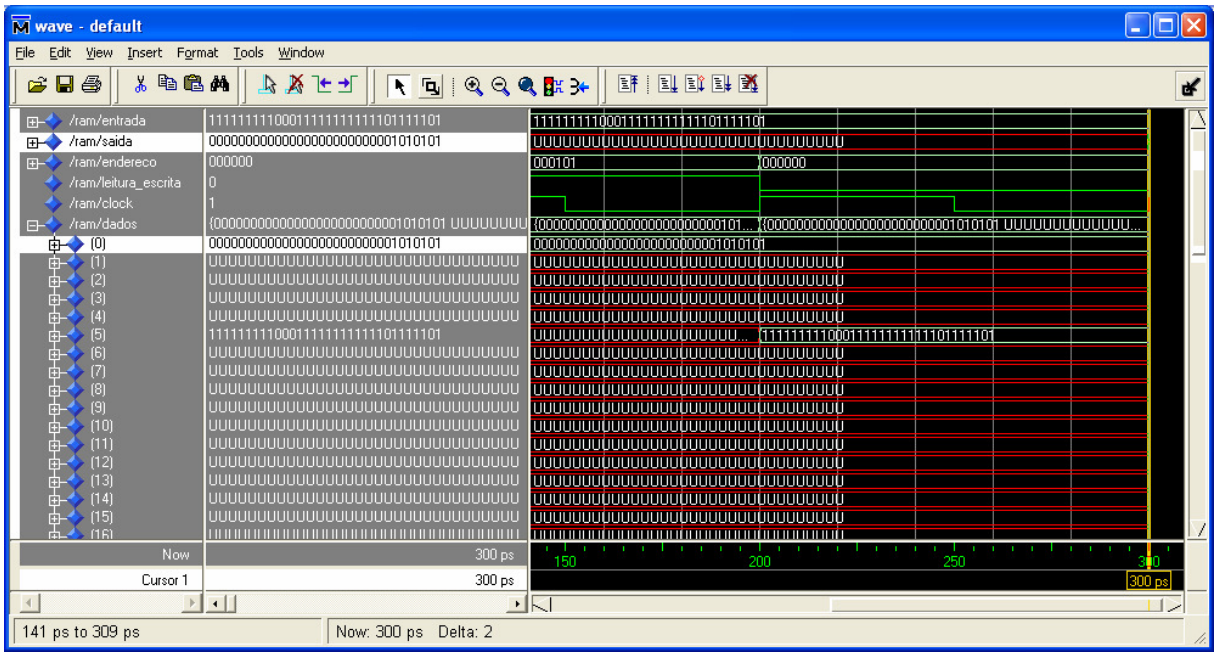


Figura 66 – Leitura Memória RAM

Memória ROM

Leitura

Na figura 67 observa-se a leitura da posição ‘010010’, quando ocorre uma borda de subida do sinal de *clock*, a porta de saída retorna o valor gravado nesta posição, neste caso o valor lido é idêntico ao endereço, porque para efeitos de testes os valores gravados na ROM são idênticos as suas respectivas posições.

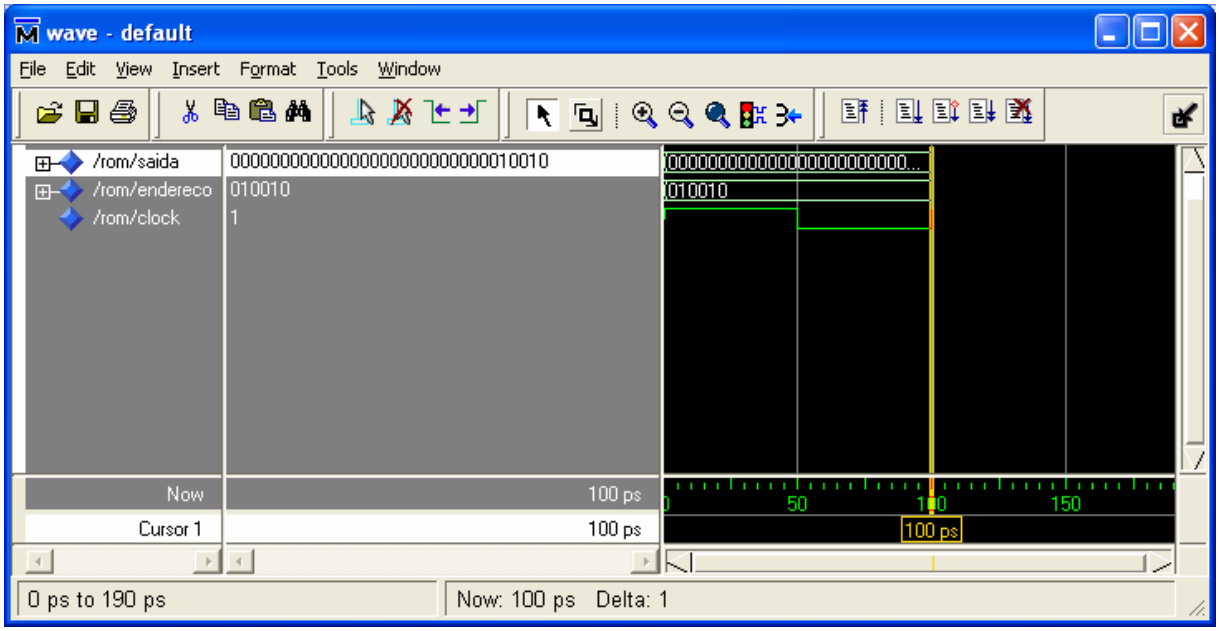


Figura 67 – Leitura Memória ROM

Memória

O módulo de memória faz a integração entre a ROM e a RAM, e relata erro ocoridos durante uma escrita ou leitura. A seguir são descritos dois erros gerados no intuito de verificar seu funcionamento.

Erro ao tentar acessar endereço inexistente

Qualquer tentativa de leitura ou escrita de qualquer endereço que não esteja dentro dos limites da capacidade da memória, gera um sinal de erro conforme mostra a figura 68 que é indicada pelo nível alto no sinal de erro, que ocorre posteriormente ao acionamento da instrução pelo *clock*. Neste exemplo o erro ocorreu devido à tentativa de leitura da posição ‘100001100000’, que corresponde ao endereço 2144, sendo o seu limite 128.

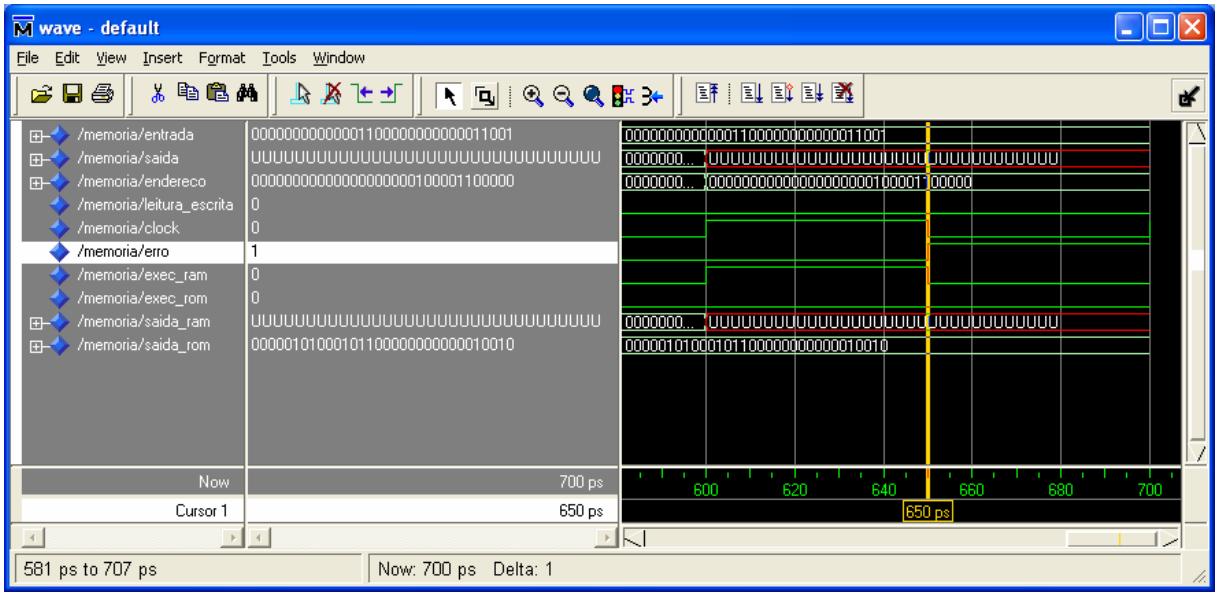


Figura 68 – Erro Memória RAM

Erro ao tentar escrever na ROM

A figura 69 mostra a geração de um erro, durante a presença do sinal de *clock*, ocasionado pela tentativa de escrita no primeiro endereço da memória ROM. Pode-se notar que leitura_escrita está com o valor ‘1’ indicando a intenção de escrita na memória ROM.

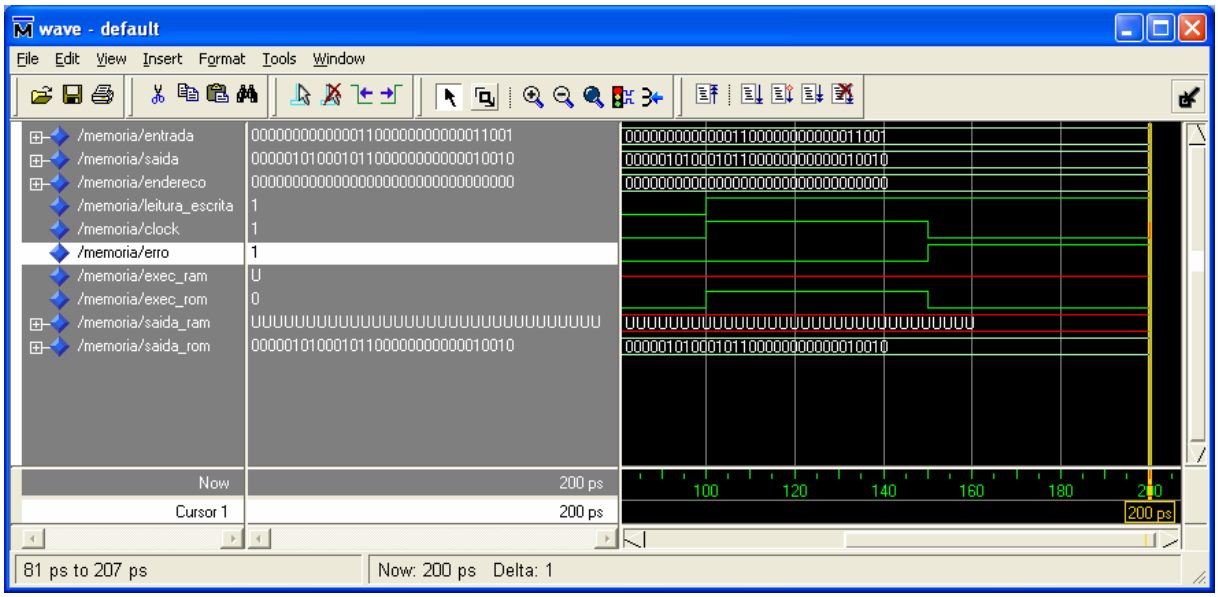


Figura 69 – Erro Memória ROM

Multiplexadores e Demultiplexadores

O *demultiplex* empregado neste trabalho é um circuito que possui uma entrada e 32 saídas, onde o sinal da entrada é enviado a uma única saída previamente especificada na porta

de endereço. Como este módulo não depende de sinal de *clock*, o valor da entrada selecionada é enviado imediatamente a saída referenciada pela porta de endereço.

Os *multiplex* empregados são: um modelo com 2 entradas, 1 saída e 1 bit de extensão, e dois modelos com 32 entradas e 1 saída, sendo um modelo com 1 bit e outro com 32 bits de extensão. O funcionamento de um *multiplex* pode ser resumido da seguinte forma: o sinal proveniente de uma das várias entradas é enviado a saída, de acordo com o controle de seleção de entrada, neste caso indicado como endereço. Assim como o *demultiplex*, este módulo não depende de sinal de *clock*, o valor da entrada selecionada é enviado imediatamente a saída.

Demultiplex 1x32

Nas figuras 70 e 71 pode-se observar o funcionamento do *demultiplex* através do envio do sinal '1' para cada uma das 32 saídas, individualmente, por meio da inserção incremental dos valores na porta endereço e as demais saídas receberam o valor contido na porta padrão, neste caso '0'.

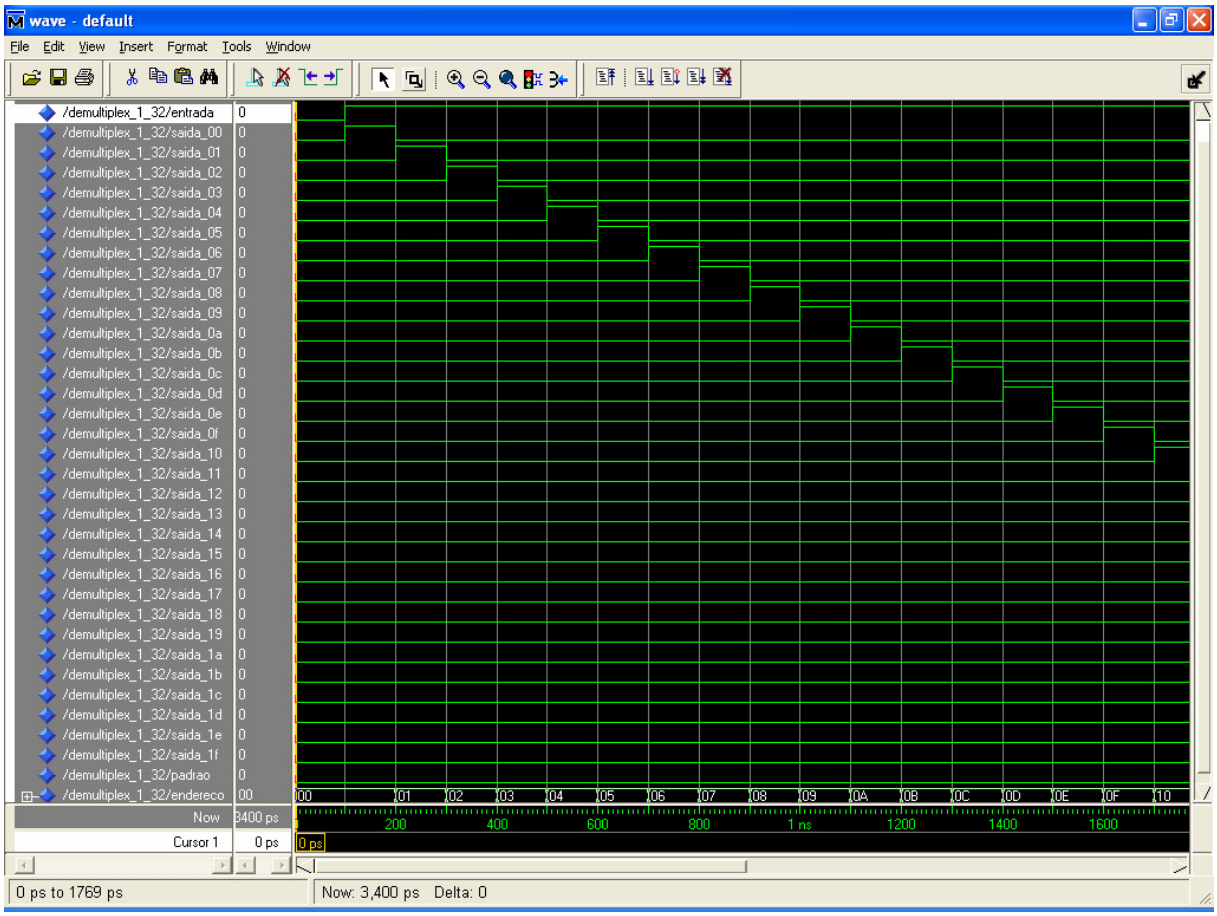


Figura 70 – Teste do *Demultiplex* 1x32

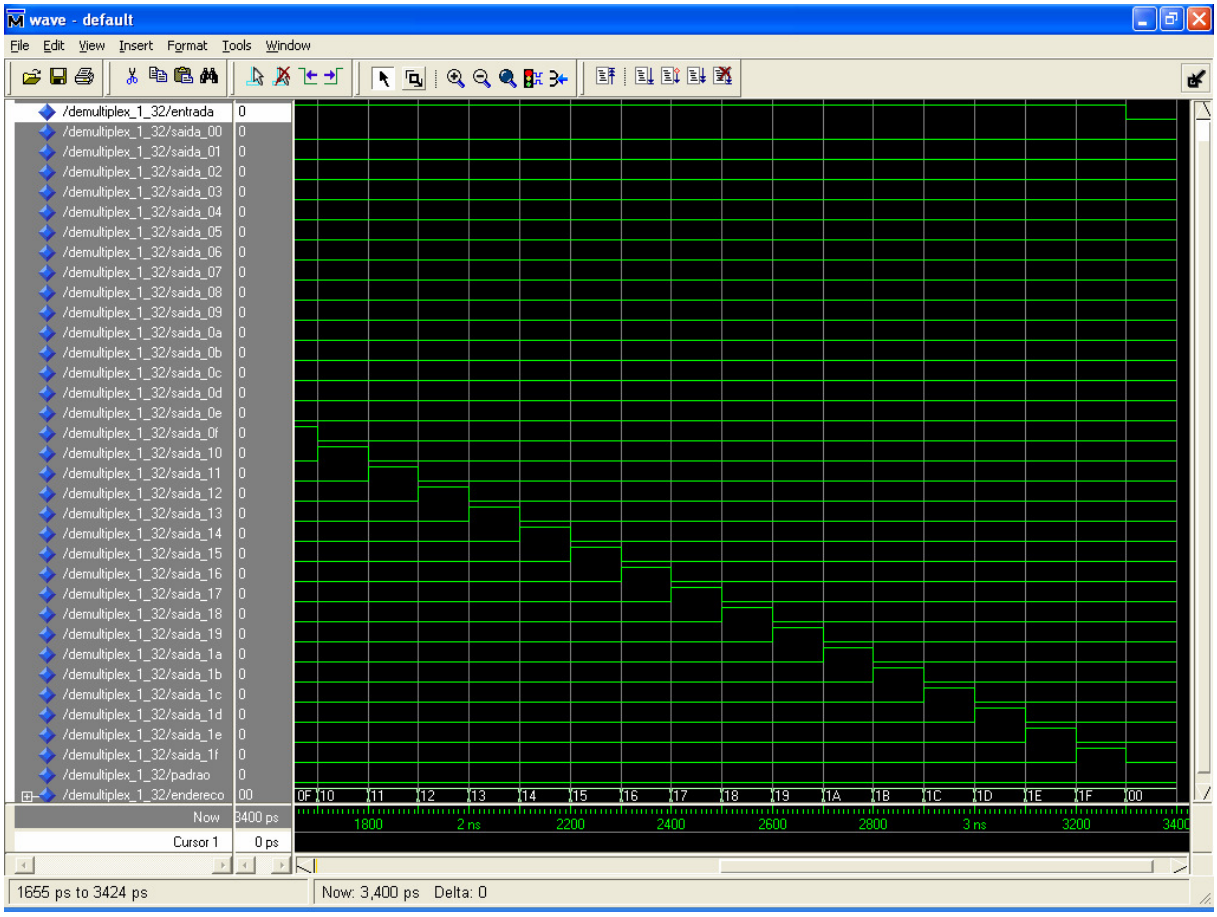


Figura 71 – Teste do *Demultiplex* 1x32

Multiplex de 1 Bit

Multiplex 32x1

Pode-se observar na figura 72 o funcionamento do *multiplex* através da seleção individual dos sinais ‘0’ ou ‘1’ que foram inseridos alternadamente em cada uma das 32 entradas, a saída recebe o valor de uma única entrada previamente selecionada através da inserção incremental dos valores na porta endereço.

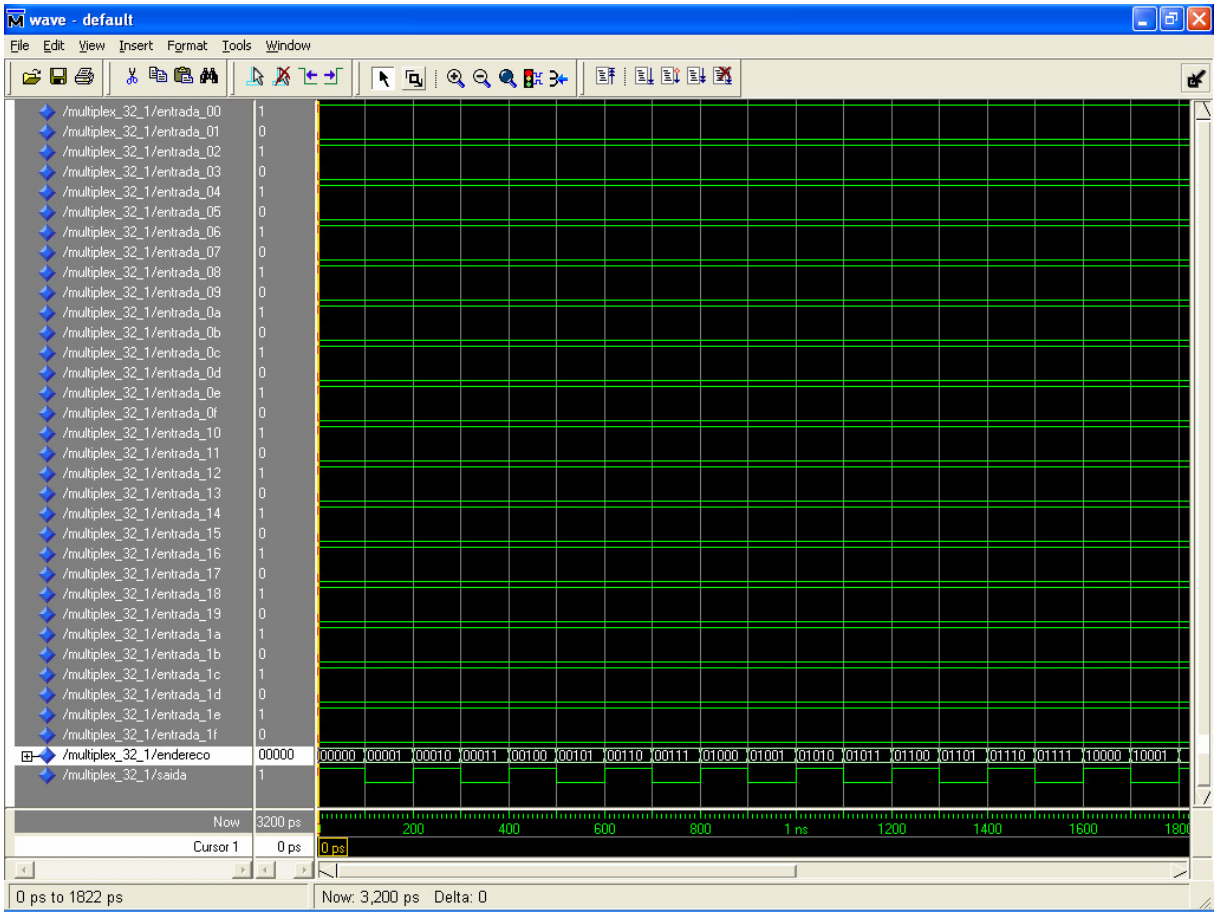


Figura 72 – Teste do *Multiplex* 32x1

Multiplex 32 bits

Os *multiplex* de 32 bits utilizados no circuito são os *multiplex* 2x1 e 32x1, o *multiplex* 2x1 é utilizado em conjunto com RCCEI, ULA e Acumulador.

Multiplex 2x1

Na figura 73 pode-se observar o funcionamento do *multiplex* 2x1, onde a porta controle ao receber o valor ‘0’ direciona o valor contido na entrada_0 a saída, caso a porta controle receba o valor ‘1’ a saída receberá o valor contido na porta_1.

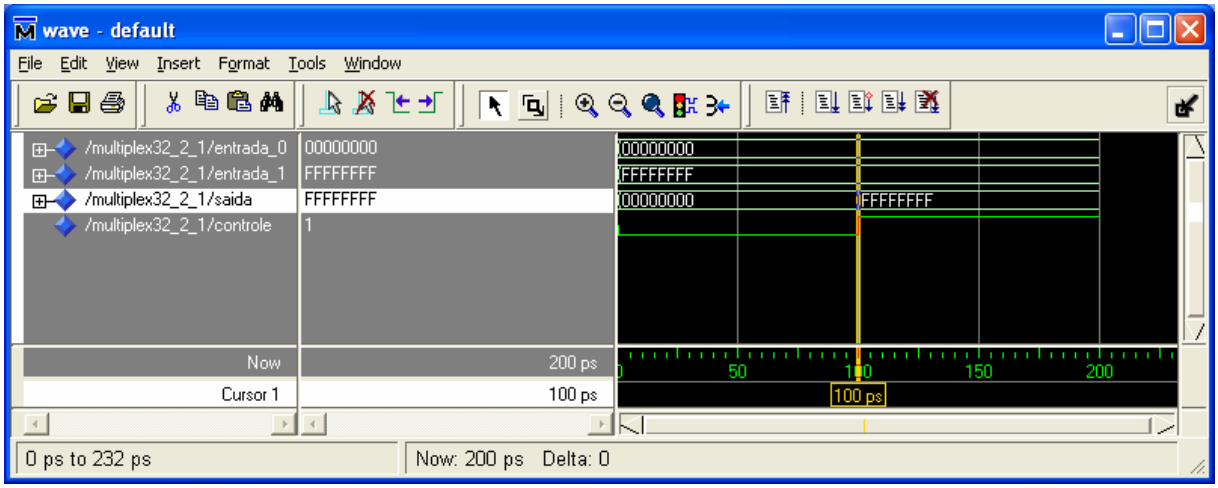


Figura 73 – Teste do *Multiplex* 2x1

Multiplex 32x1

Pode-se observar na figura 74 o funcionamento do *multiplex* 32x1, sendo o seu funcionamento similar ao *multiplex* 32x1 de 1 bit, a única diferença é que este *multiplex* manipula palavras com 32 bits de extensão.

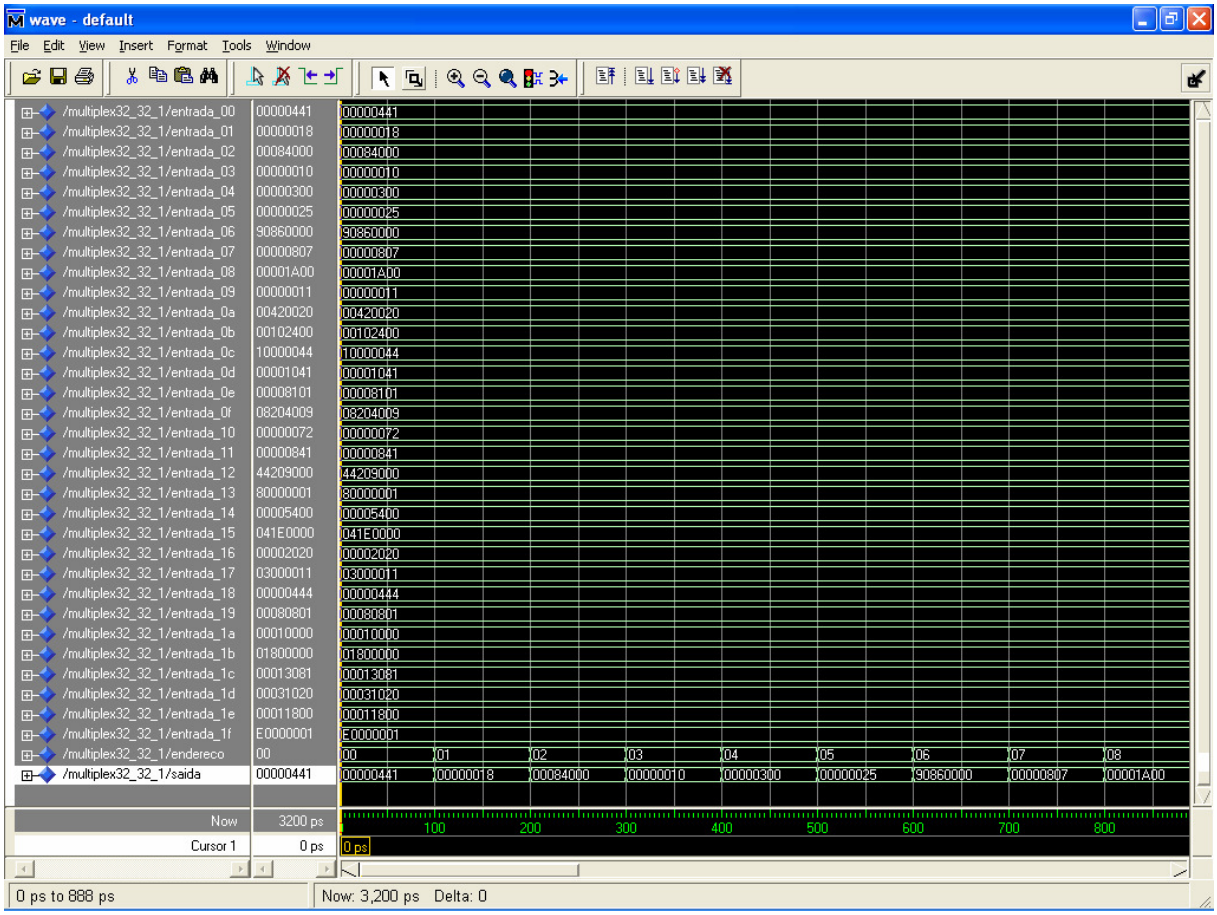


Figura 74 – Teste do *Multiplex* 32x1

Registradores e Contadores

Flip Flop

Pode-se observar na figura 75 o funcionamento do *flip flop*, ele armazena um bit, e recebe o valor que será armazenado durante a borda de subida de *clock*, sua saída mantém o valor contido nele.

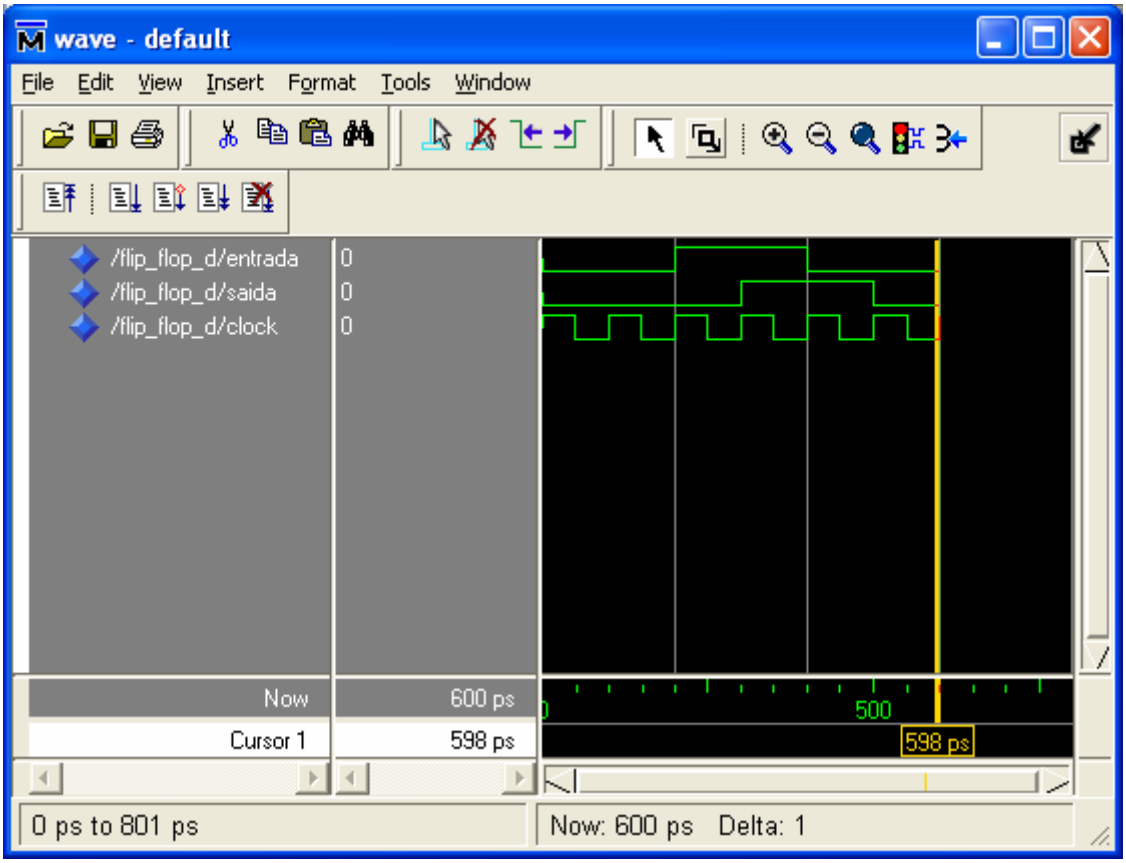


Figura 75 – Flip Flop D

Pilha

A pilha implementada é composta por um conjunto de registradores, possui 8 níveis e 32 bits de extensão, sua função é armazenar os dados em uma lista linear onde o primeiro elemento a entrar será o último a sair.

Pode-se observar na figura 76 a função de escrita na pilha, na figura 77 a indicação de pilha cheia e na figura 78 a indicação de pilha vazia.

Escrita na pilha

A primeira etapa na utilização da pilha é iniciar seus valores, isto é realizado deixando o sinal iniciar em nível alto, com os valores iniciados será indicado que a pilha está vazia, note na figura 76 que o pino pilha_vazia permanece em nível alto depois de iniciar a

pilha até realizar a leitura. Para que o valor da entrada seja empilhado no topo da pilha o sinal de escrita deve permanece em nível alto durante a borda de subida do *clock*.

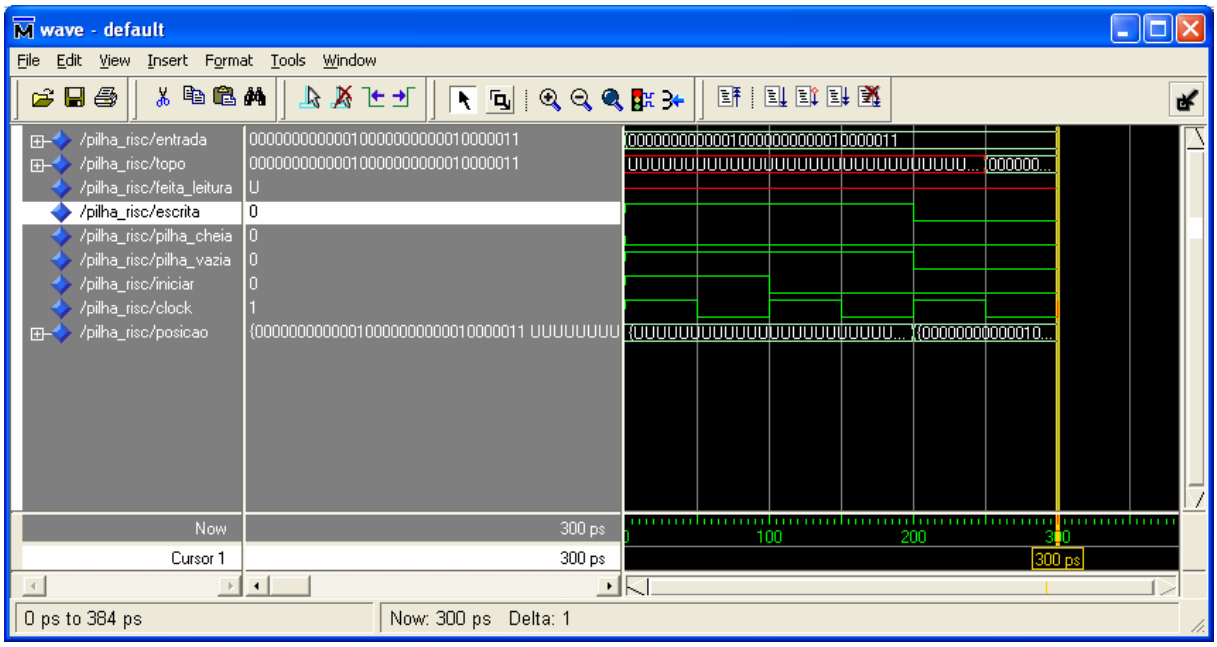


Figura 76 – Escrita na Pilha

Pilha Cheia

Com as oito posições da pilha ocupadas, não é possível empilhar mais nada nela, e o módulo indica isto deixando o sinal pilha_cheia com o valor ‘1’ até que seja efetuado uma operação de desempilhar.

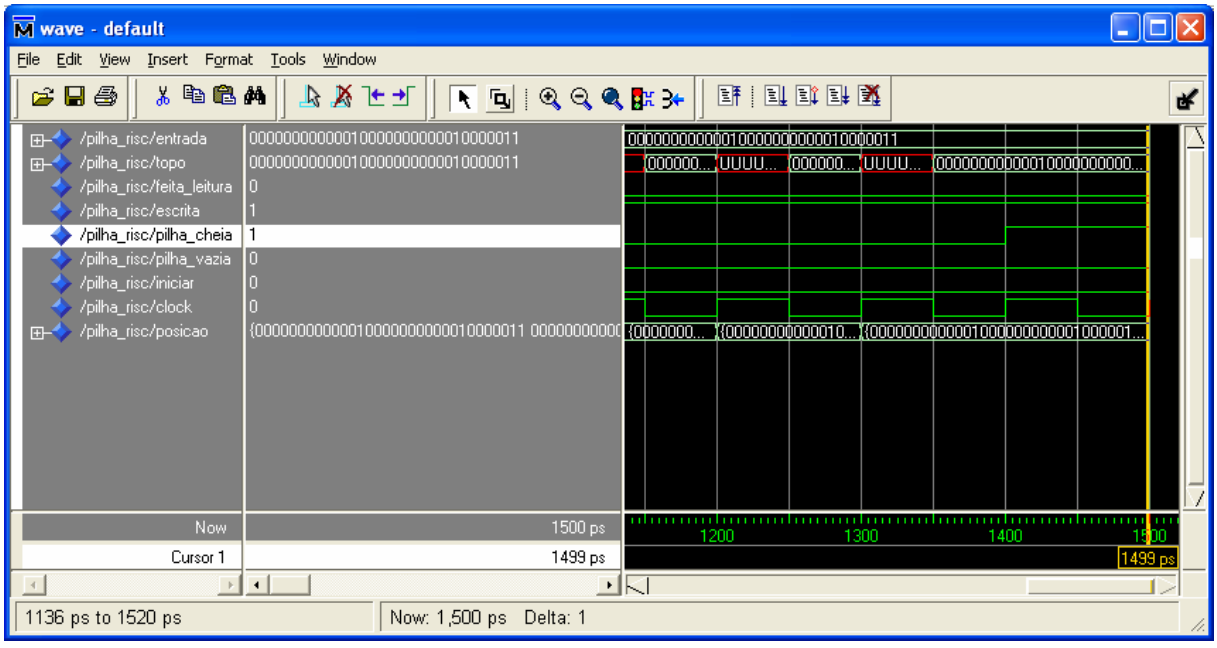


Figura 77 – Pilha Cheia

Pilha Vazia

Através da figura 78 é possível observar que para desempilhar da pilha é necessário deixar o sinal feita_leitura valendo ‘1’ durante uma borda de subida de *clock*, nesta figura também é demonstrado que após realizar a leitura do primeiro elemento da pilha o sinal pilha_vazia fica em nível alto indicando que já foi desempilhado o último valor contido na pilha.

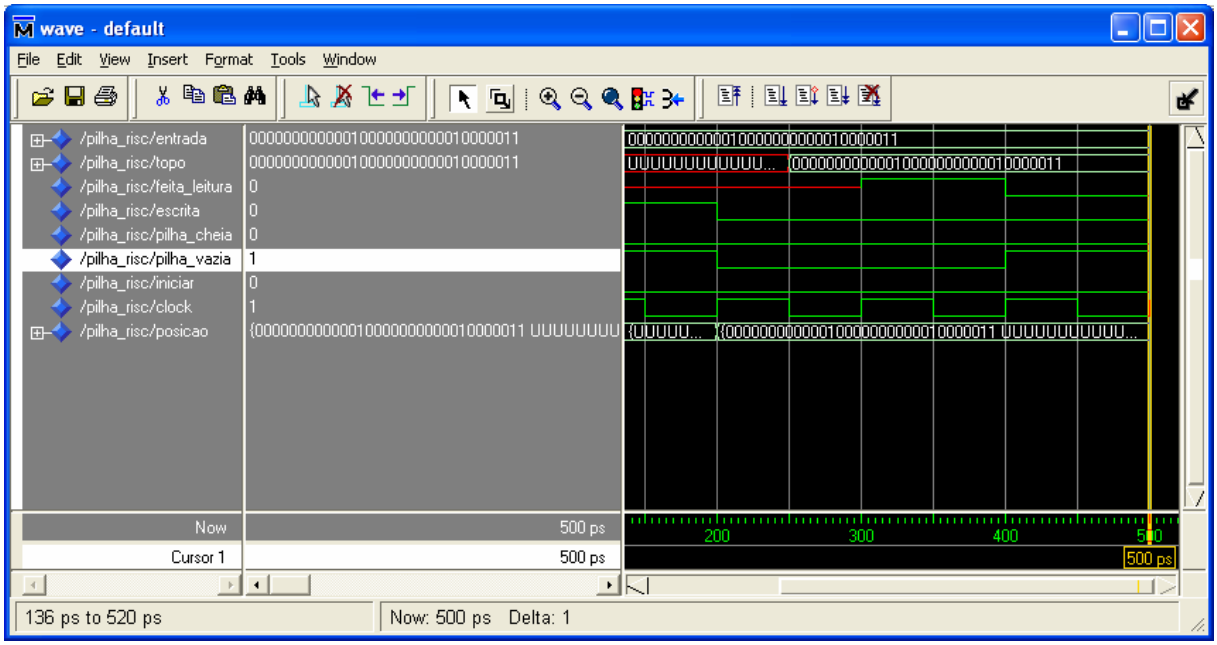


Figura 78 – Pilha Vazia

RCCEI - Registrador Código de Condição e de Erro e Interrupção

O RCCEI é um registrador de 32 posições com 1 bit de extensão, e sua função é registrar informações a respeito de determinadas instruções que foram executadas, registra também informações a respeito das interrupções que podem ocorrer, seja interna ou externa.

Escrita Modo Normal

Pode-se observar na figura 79 o funcionamento do RCCEI no Modo Escrita Normal, o valor contido na entrada é armazenado no registrador após a borda de subida do *clock*.

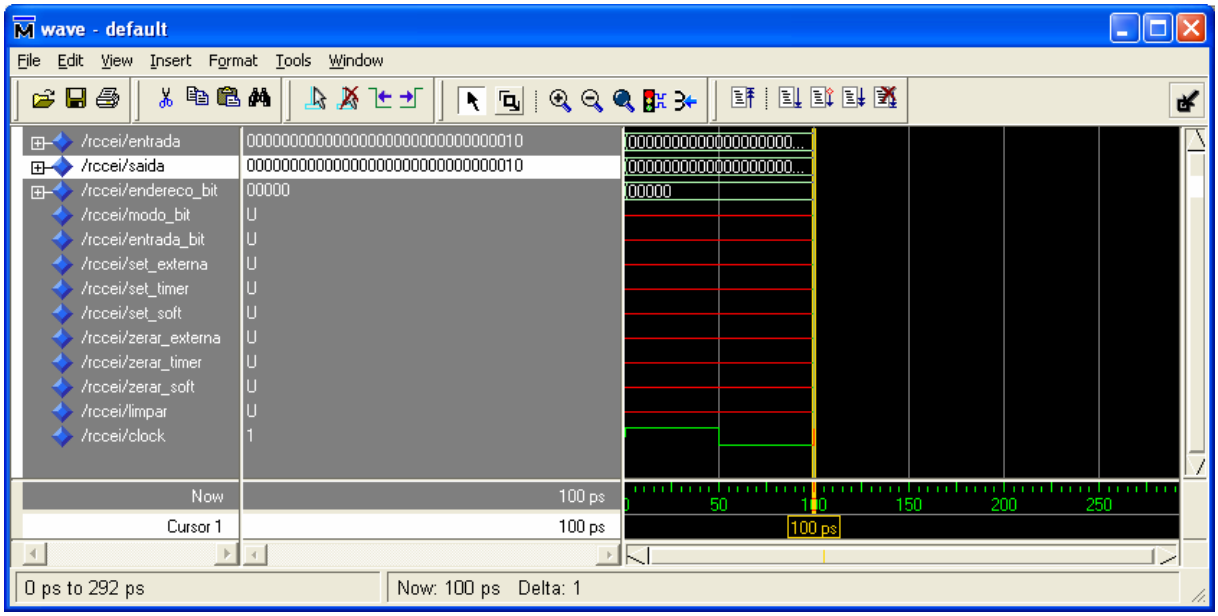


Figura 79 – RCCEI Escrita Modo Normal

Modo Bit

Pode-se observar na figura 80 o funcionamento do RCCEI no modo Bit. Primeiro indica-se qual bit será alterado pelo sinal endereco_bit, e deixa o sinal modo_bit em nível alto, e com a borda de subida de *clock*, o valor contido na entrada_bit é transferido para o bit indicado no endereco_bit.

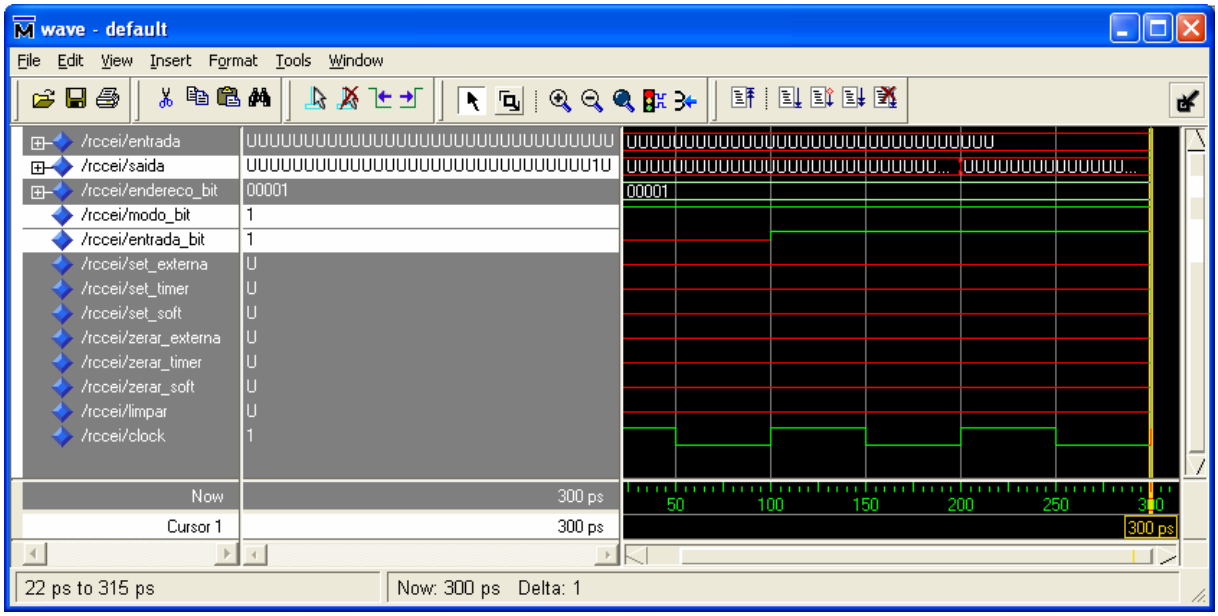


Figura 80 – RCCEI Modo Bit

Pinos de Set

Pode-se observar na figura 81 o funcionamento dos pinos de Set do RCCEI. Com uma das entradas set nível alto a saída correspondente a interrupção recebe valor ‘1’.

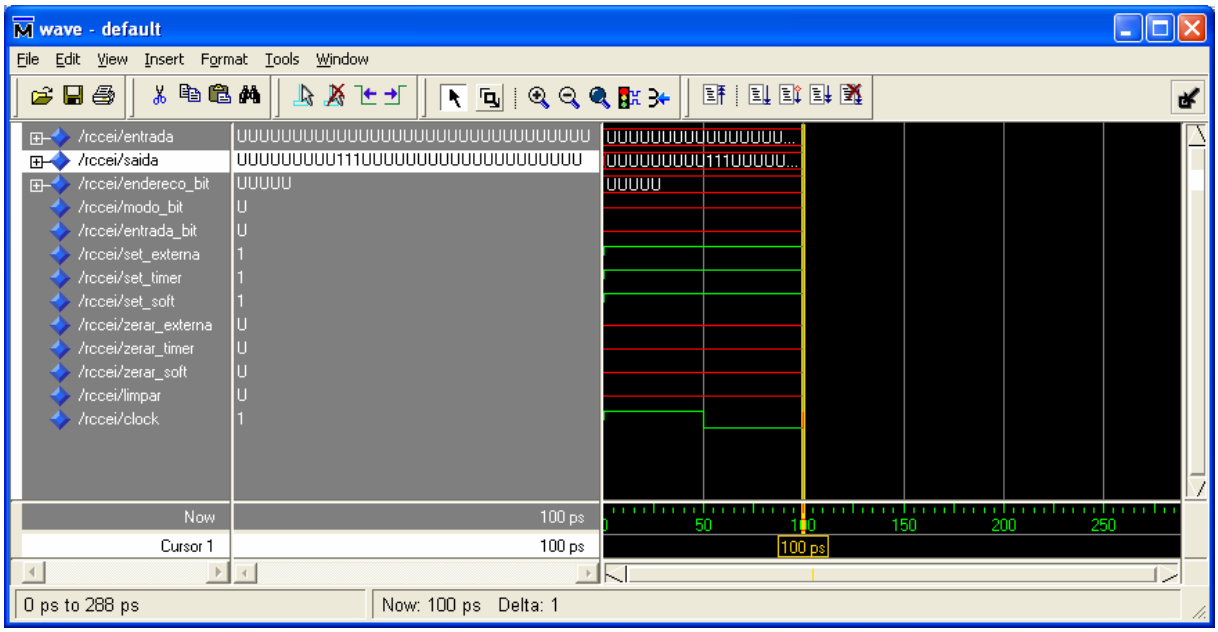


Figura 81 – RCCEI Modo Set

Pinos de Zerar

Pode-se observar na figura 82 o funcionamento dos pinos de zerar do RCCEI. Com uma das entradas zerar em nível alto a saída correspondente a interrupção recebe valor ‘0’.

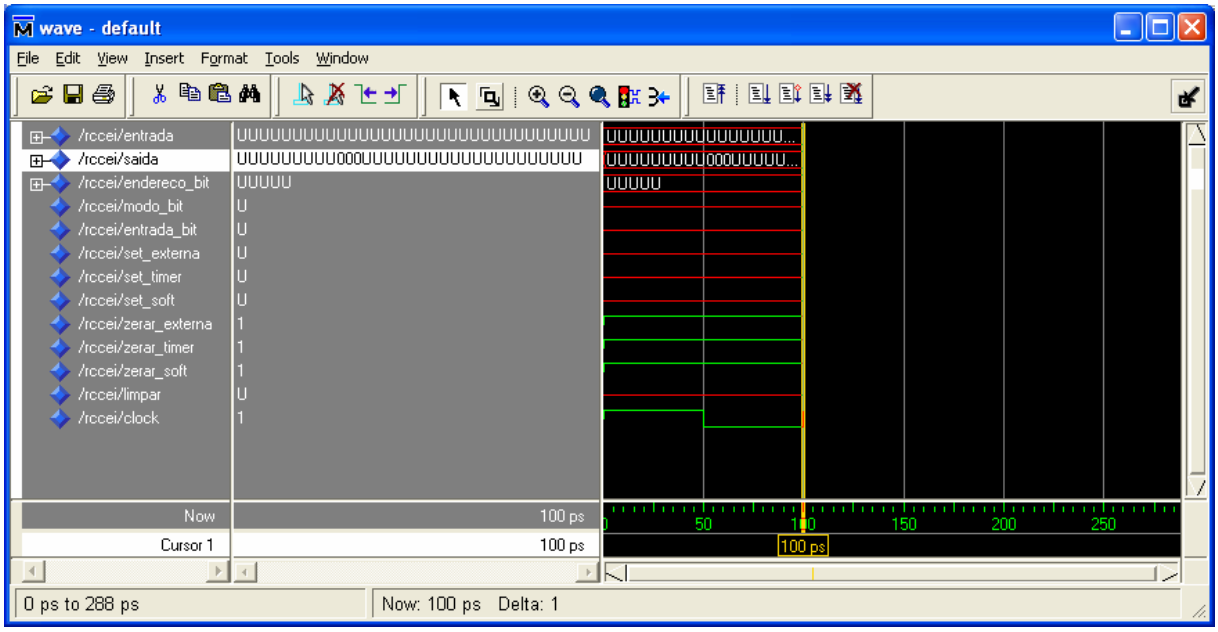


Figura 82 – RCCEI Modo Zerar

Limpar

Pode-se observar na figura 83 o funcionamento do RCCEI com o pino Limpar em nível alto, todos os bits da saida recebem o valor ‘0’.

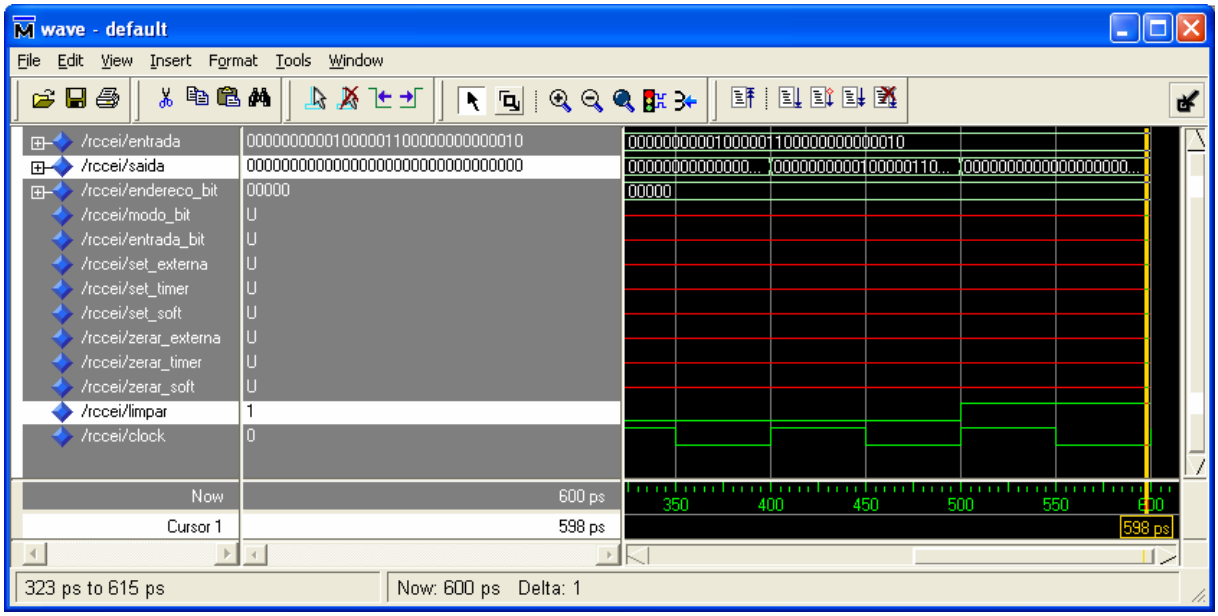


Figura 83 – RCCEI Modo Limpar

Registrador 32

Os demais registradores possuem comportamento semelhante ao RCCEI, nenhum outro possui os pinos de set e zerar, e alguns não possuem o modo bit, nem o pino de limpar, com funcionamento semelhante ao *clear*.

Pode-se observar na figura 84 o Registrador32 no Modo Bit. Observe que o bit endereçado pelo pino endereco_bit recebe o valor da entrada_bit, após a borda de subida de *clock*.

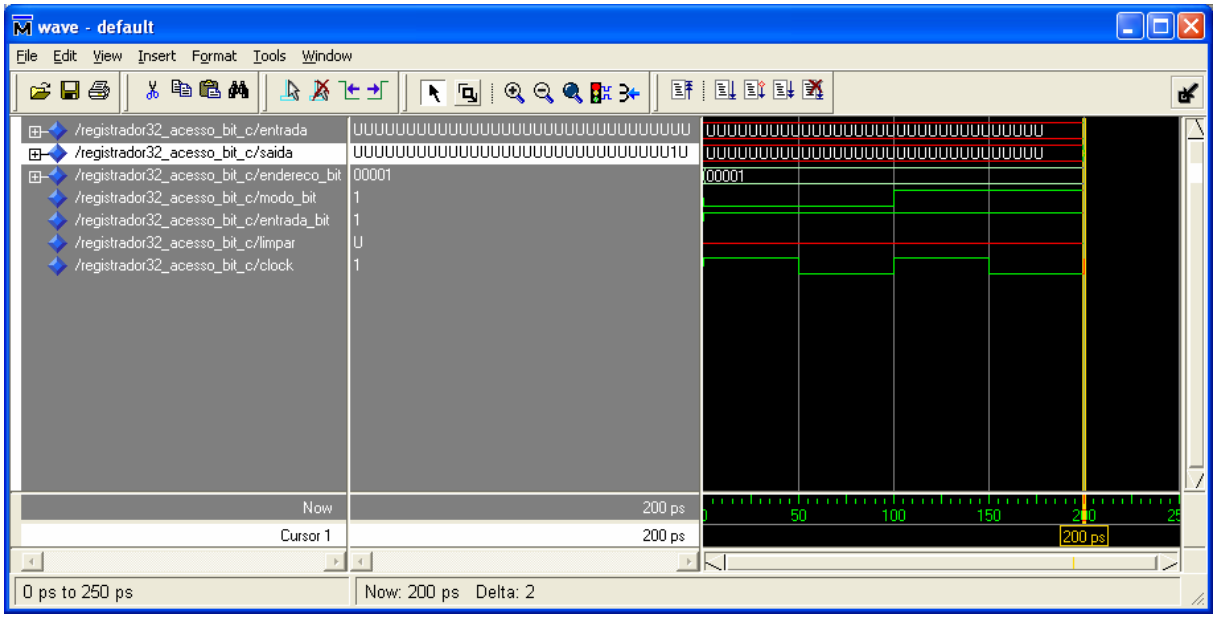


Figura 84 – Registrador32 Modo Acesso Bit com Modo Bit

Na figura 85 pode ser observado o efeito do pino limpar no Registrador32, a saída recebe zero em seus bits com limpar em nível alto.

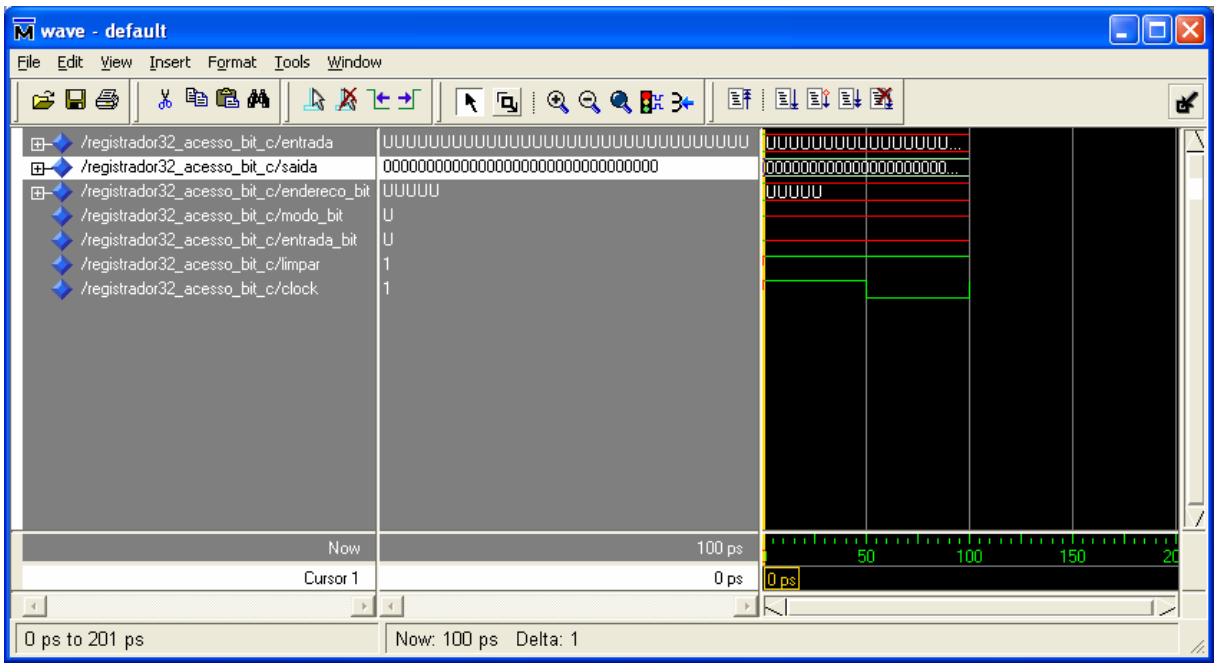


Figura 85 – Registrador32 Modo Limpar

Pode-se observar na figura 86 que o Registrador32, recebe o valor da entrada após a borda de subida de *clock*, caso modo_bit e limpar não estejam valendo ‘1’.

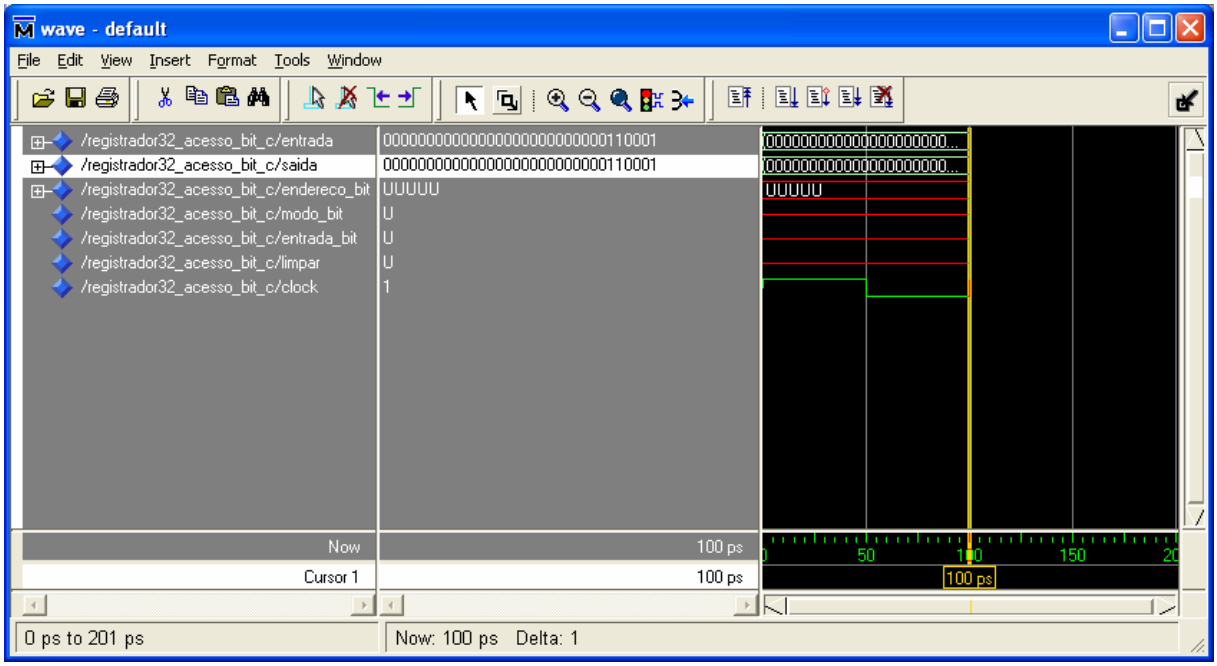


Figura 86 – Registrador32 Modo Normal

Registrador 32 - Contador

Pode-se observar na figura 87 o Registrador32 Contador, ele é um contador crescente, portanto tem seu valor incrementado a cada ciclo de *clock*, desde que os pinos limpar e leitura, não estejam valendo ‘1’. Observe na figura 87 que o primeiro passo realizado foi setar o valor zero no contador, isto foi realizado através do pino Limpar que funciona como um *clear*.

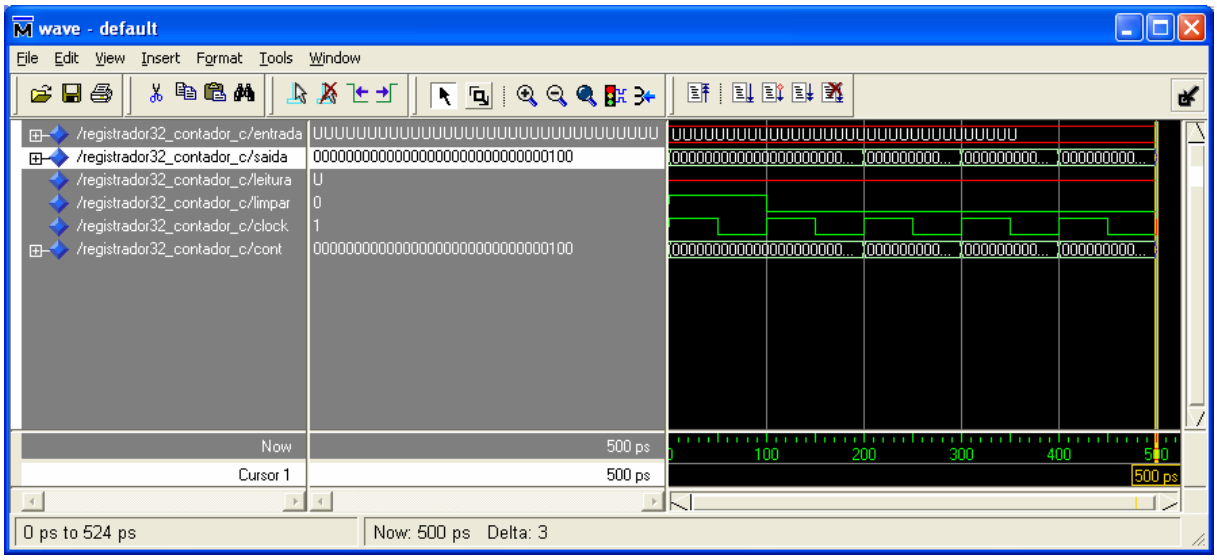


Figura 87 – Registrador32 Contador

Registrador 32 Contador - Leitura

Pode-se observar na figura 88 o Registrador32_contador, com o pino Leitura em nível alto, opera como um registrador, armazenando o valor da entrada após a borda de subida do *clock*.

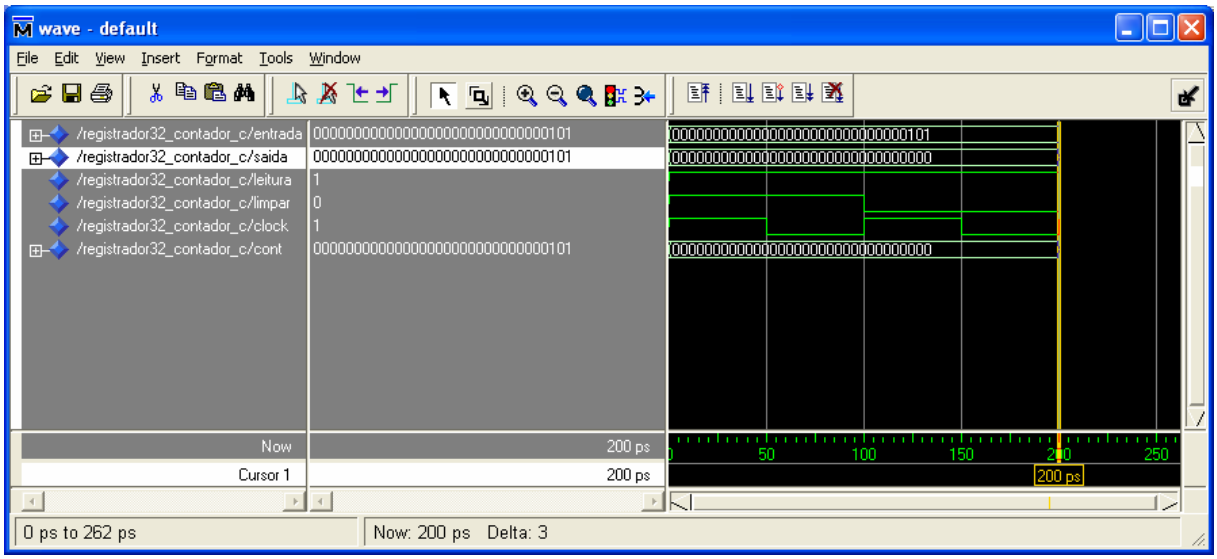


Figura 88 – Registrador32 Leitura

Timer

O *timer* é um contador decrescente, que volta ao valor inicial quando seu valor chegar a zero, e indica isto levando o pino interrupção a nível alto por um ciclo de *clock*, ele possui um pino limpar que atribui ‘1’ em todos seus bits, semelhante a um *preset*.

Leitura

Na figura 89 é demonstrado uma operação de leitura, e posteriormente, efetuam-se ciclos de *clock* até ele voltar ao valor inicial. Para realiza a leitura disponibilizou o valor “4” em sua entrada, foi atribuído ‘1’ ao pino leitura e efetuado um sinal de *clock*, depois atribuído ‘0’ a leitura e continuou com o sinal de *clock*, a cada borda de subida seu valor foi decrementado até chegar a zero, quando ele volta a valer “4” e seta o bit de interrupção.

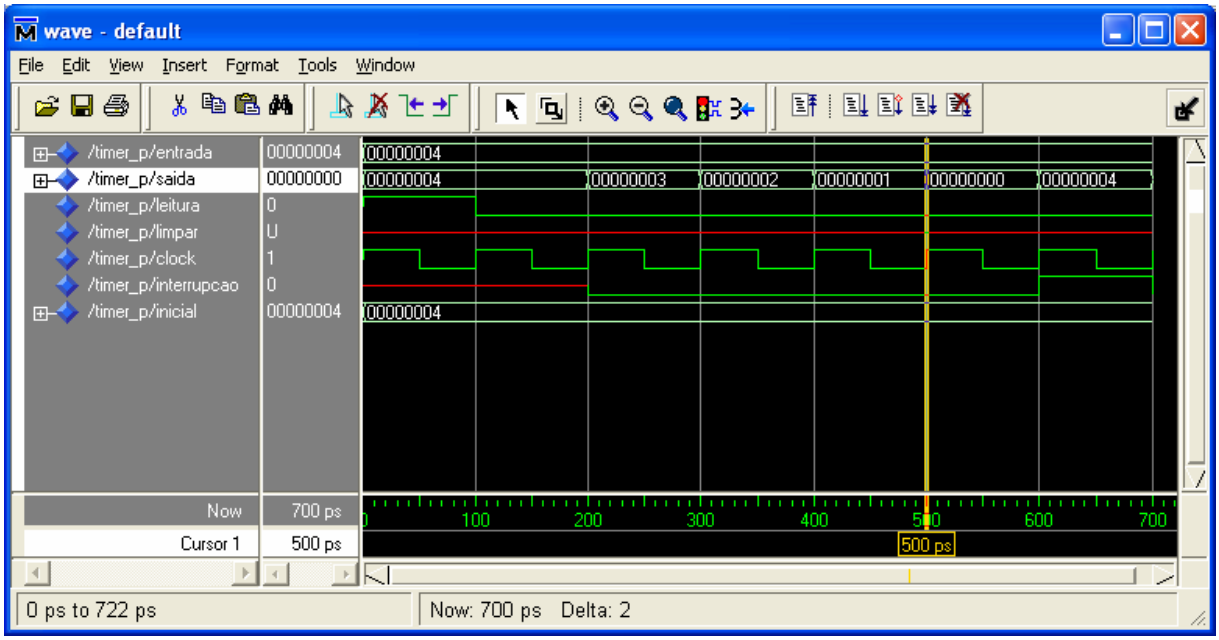


Figura 89 – Timer Leitura

Limpar

Pode-se observar na figura 90 o Timer que com o pino Limpar em nível alto é atribuído o valor ‘1’ em todos seus bits.

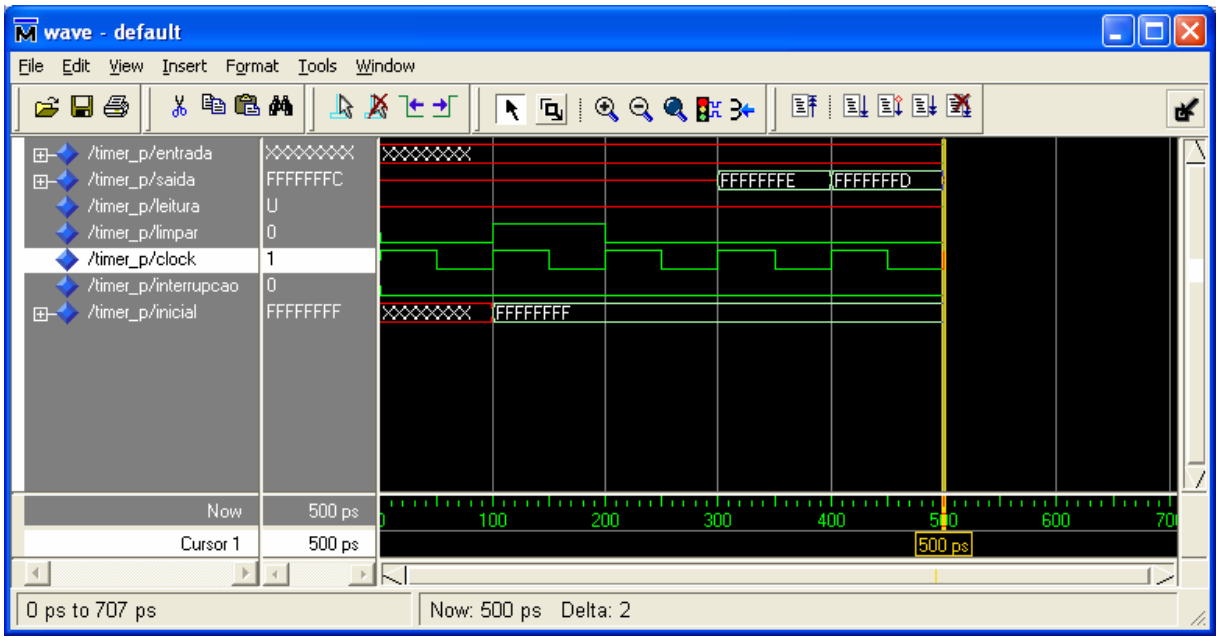


Figura 90 – Timer Limpar

ULA

A ULA é uma entidade que integra os componentes de deslocamento, lógica, operador relacional, adição, subtração, divisão e multiplicação. A seguir são demonstrados alguns dos testes realizados nos seus componentes.

Somador_subtrator

Pode-se observar na figura 91 alguns testes de adição. Para realizar a adição o pino de operação deve valer ‘0’, os valores dos operandos de 32 bits serão somados, caso o pino de vem_1 estiver em nível alto é somado 1 no resultado, como pode ser observado na figura 92, o vai_1 recebe ‘1’ se o resultado tiver mais de 32 bits.

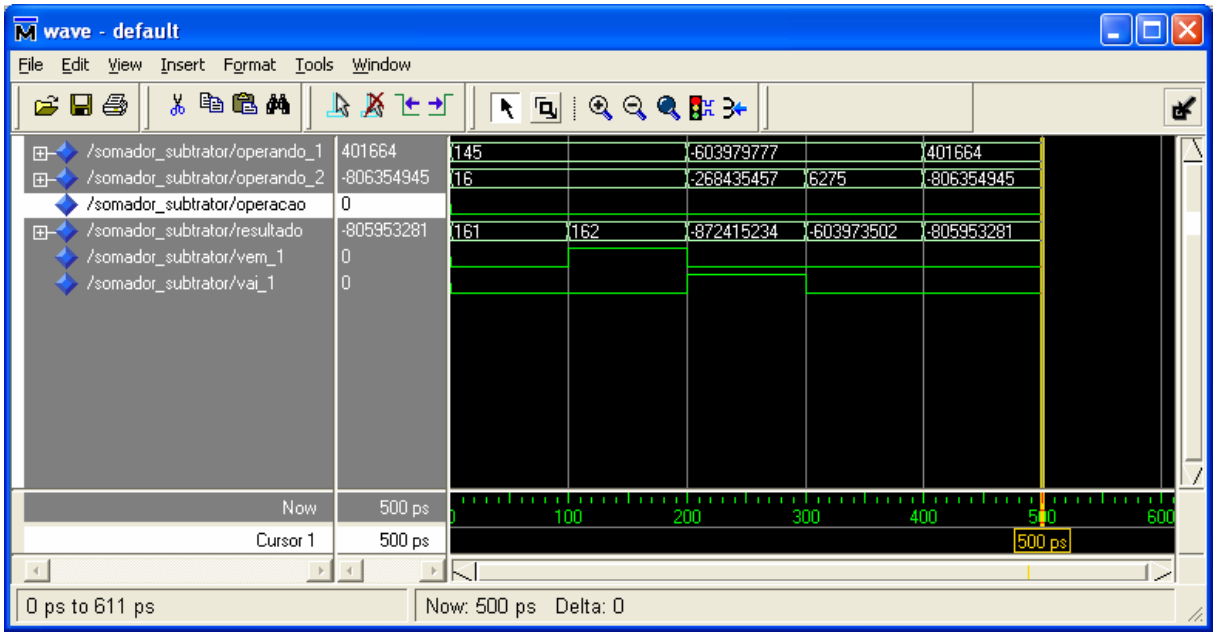


Figura 91 – Teste do módulo Somador

A subtração é realizada pela mesma entidade, quando o pino operação estiver valendo ‘1’, neste caso o vem_1 em nível alto subtrai 1 do resultado.

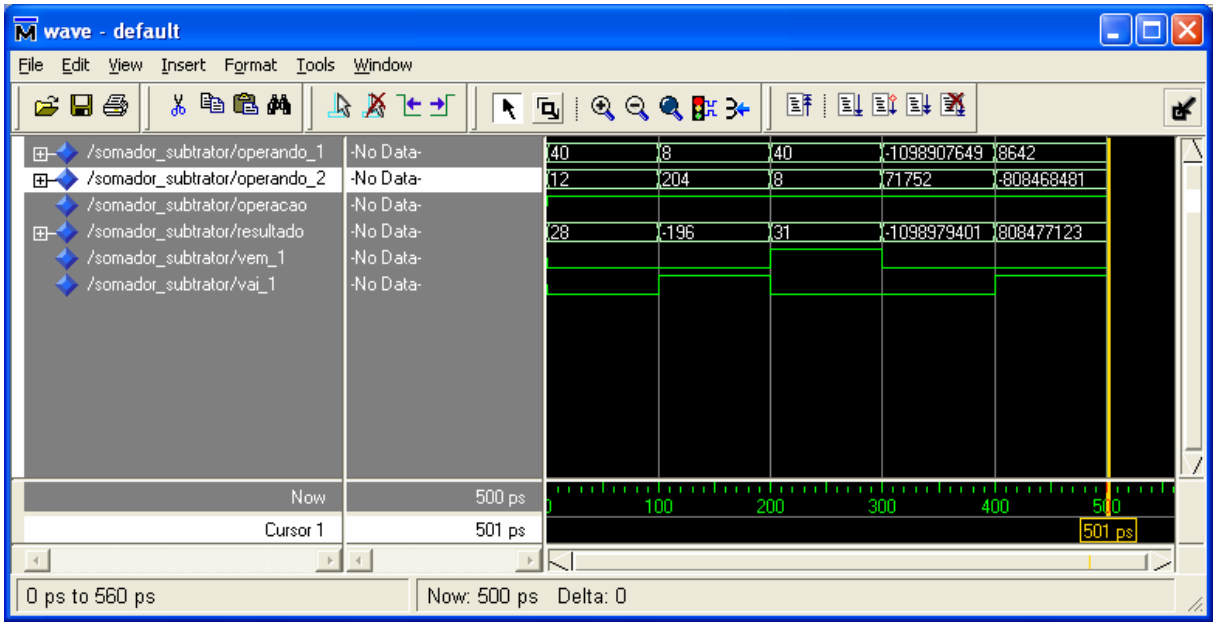


Figura 92 – Teste do módulo Subtrator

Multiplicador

O multiplicador realiza a operação de multiplicação entre dois operandos de 32 bits em 32 ciclos de *clock*, para tal deve-se iniciar a operação, disponibilizando os valores e deixando o pino iniciar em '1', feito este passo, volta o pino iniciar para '0', e após 32 ciclos de *clock*, a porta resultado recebe o resultado da operação, como pode ser observado pela figura 93.

Caso o resultado ultrapasse 32 bits o transbordo receberá '1'.

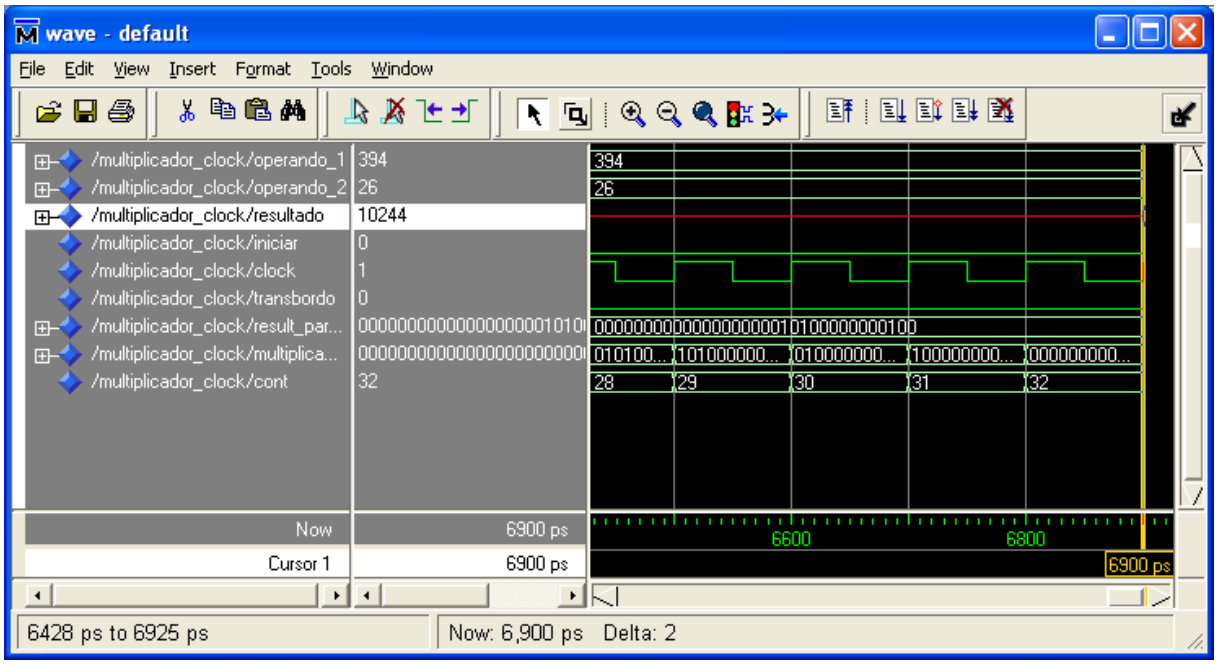


Figura 93 – Teste do módulo Multiplicador

A figura 94 apresenta um segundo teste de multiplicação com um número negativo.

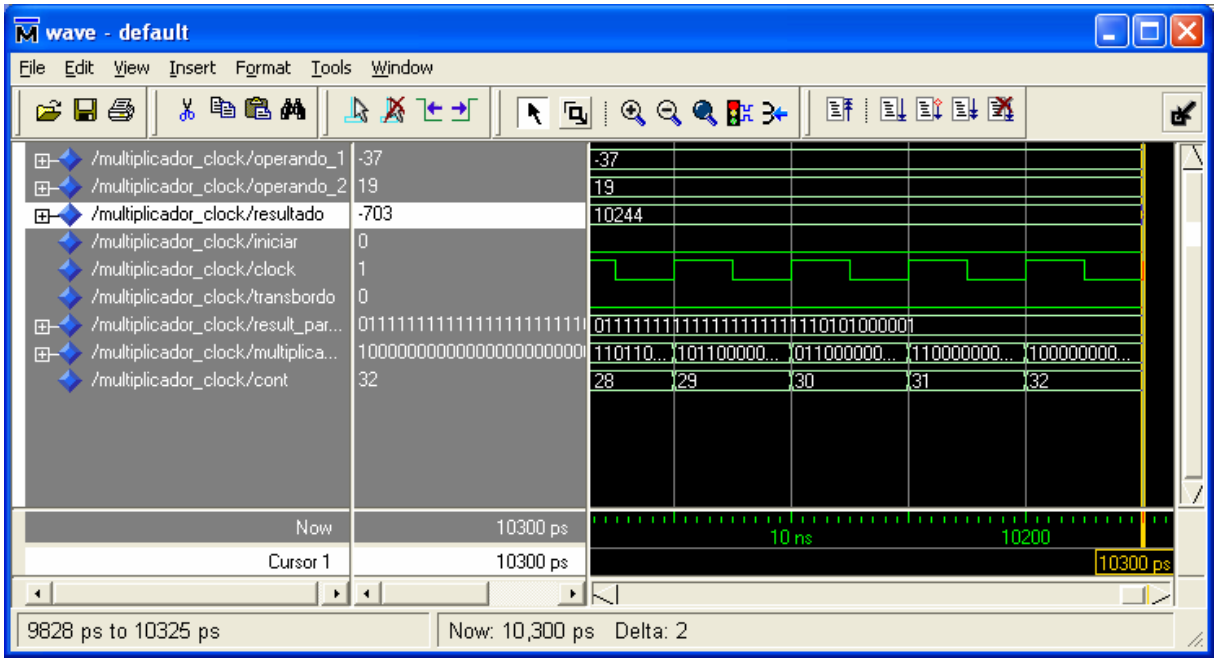


Figura 94 – Multiplicação por número negativo

Divisor

O divisor funciona de forma semelhante ao multiplicador, inicialmente deve-se informar os valores iniciais, e setar o pino iniciar, depois zerar ele, e passando 32 ciclos de clock o resultado será disponibilizado, e o resto também. Se o resultado do resto for diferente de zero o pino div_queb recebe ‘1’, e se ocorrer uma divisão por zero o pino div_zero recebe ‘1’, conforme pode ser observado na figura 95.

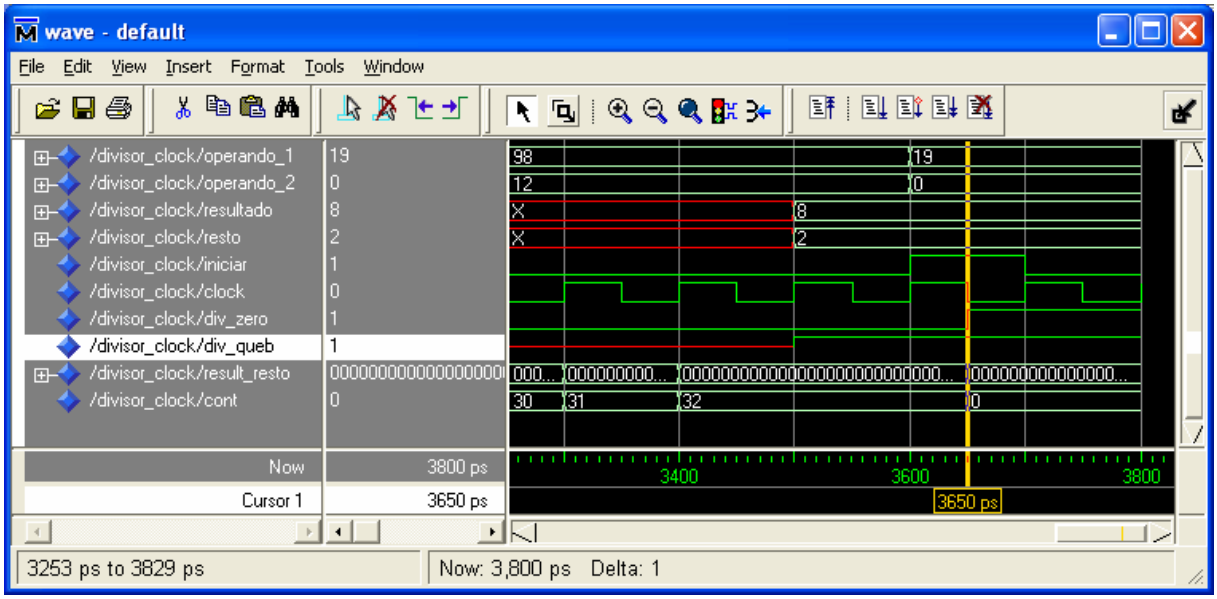


Figura 95 – Teste do módulo Divisor

Deslocamento

O módulo de deslocamento realiza uma operação de deslocamento ou rotação de acordo com valor da porta operação. Pode-se observar na figura 96 a realização da operação de rotação esquerda com a porta operação valendo “000”.

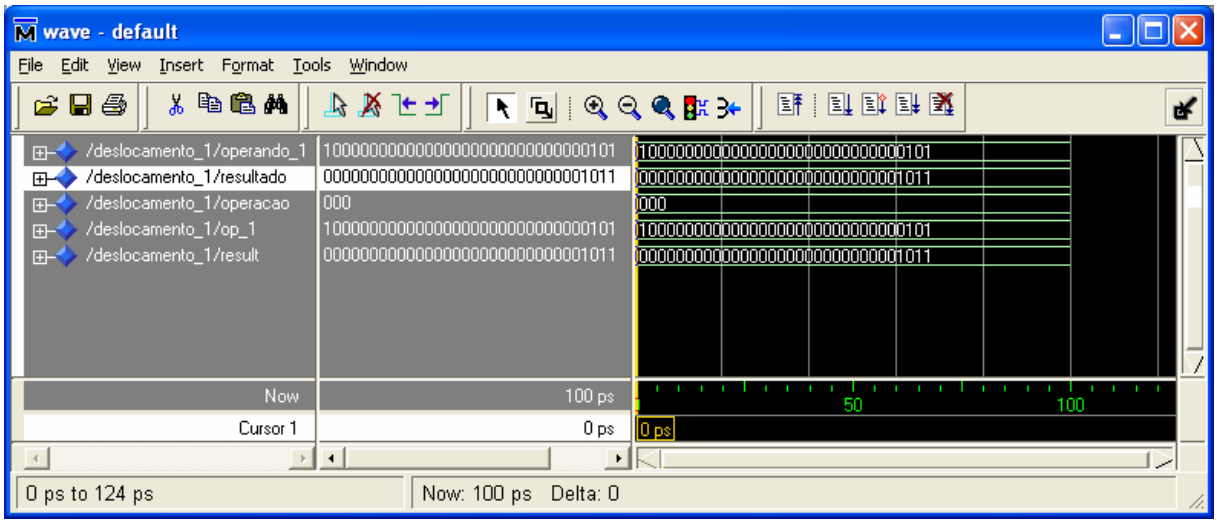


Figura 96 – Rotação a Esquerda

A figura 97 demonstra a operação de rotação direita sem deslocar o valor contido no bit 31 da entrada.

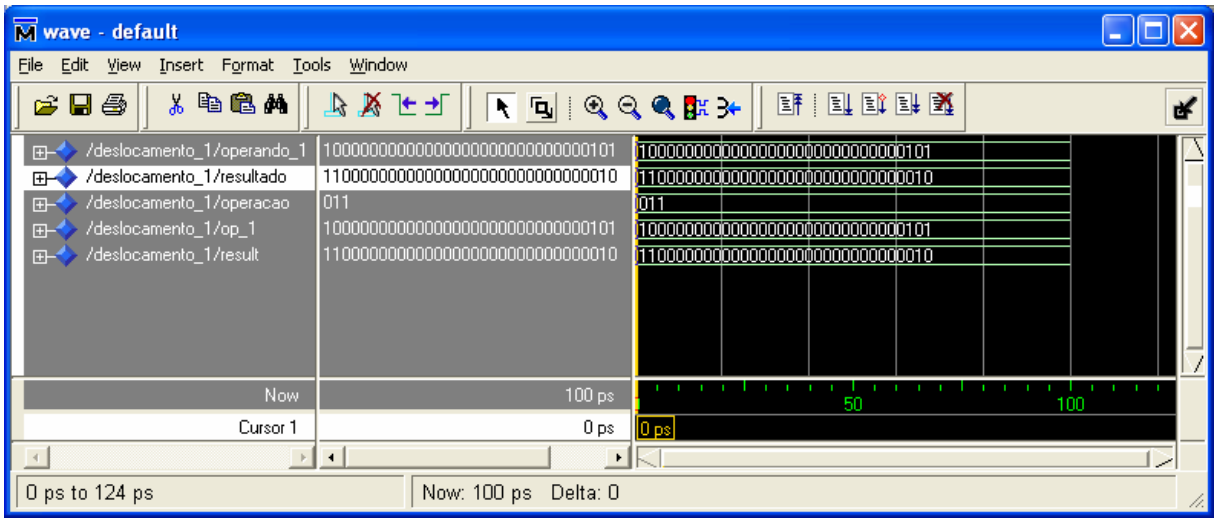


Figura 97 - Rotação direita sem sinal

E a figura 98 descreve uma operação de deslocamento à esquerda, realizado com o valor da porta operação igual a “100”.

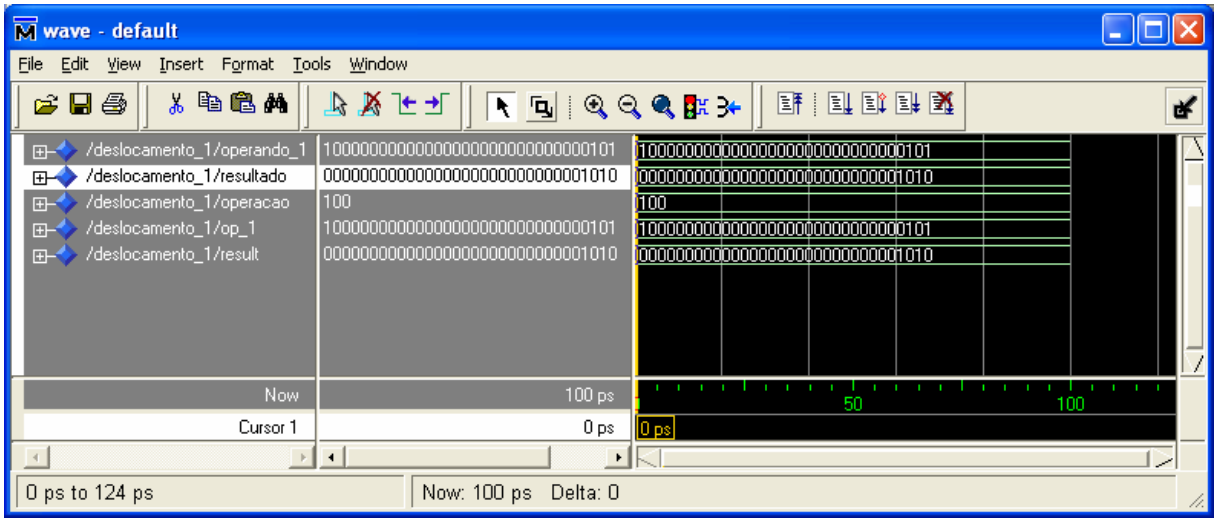


Figura 98 - Deslocamento à Esquerda

Lógica

O módulo de lógica realiza operações lógicas de: AND, OR, XOR, e NOT; Como exemplo a figura 99, demonstra a operação de negação.

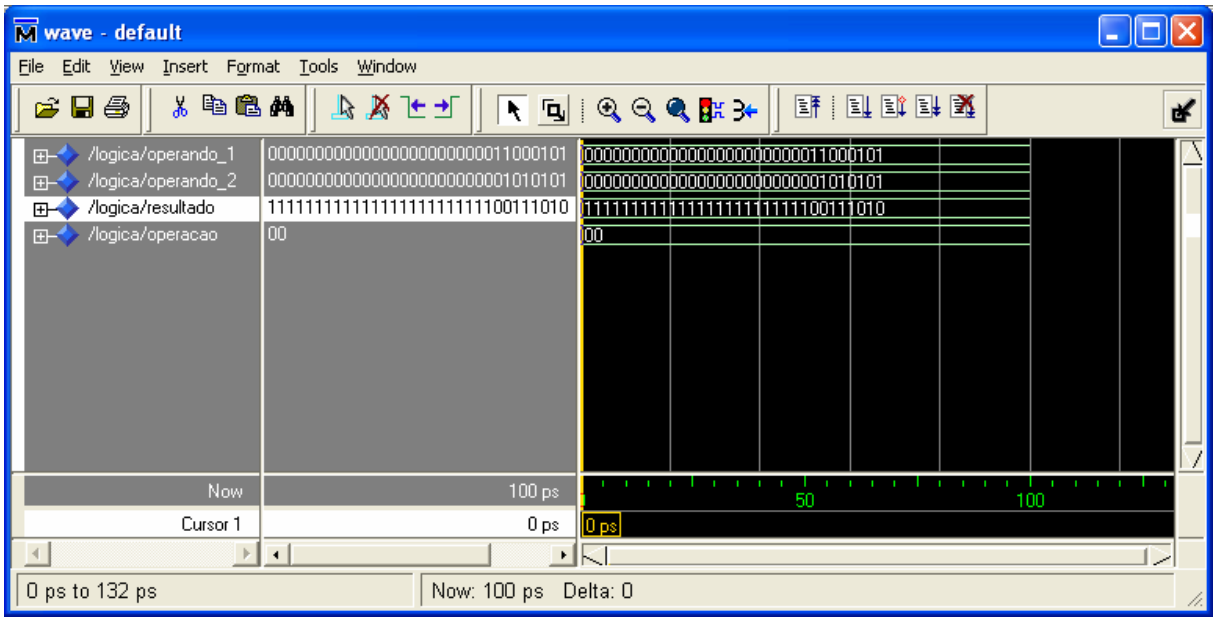


Figura 99 – Operação de negação

A figura 100 demonstra a operação E (AND), após mudar o valor do pino de operação, o resultado da operação recebe o valor da operação realizada bit a bit.

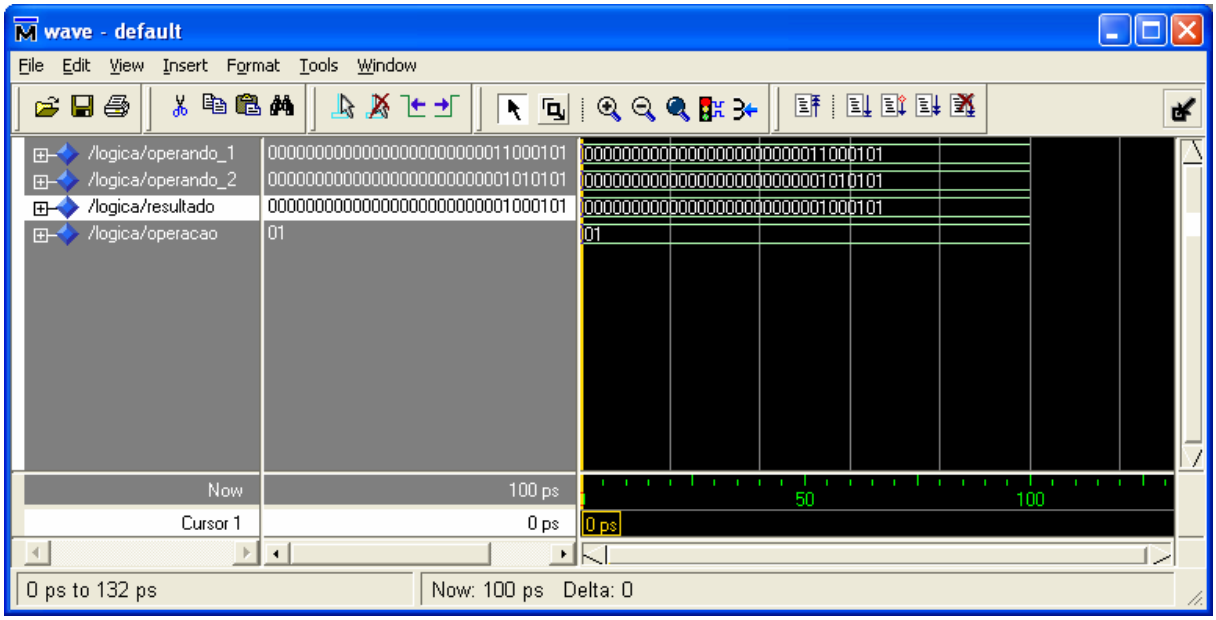


Figura 100 - Operação E

Operador Relacional

O operador relacional realiza as comparações entre os valores das entradas, Observe que caso o operando_1 seja maior que o operando_2 o resultado será “01” como pode ser observado na figura 101, se for o inverso o resultado será “10”.

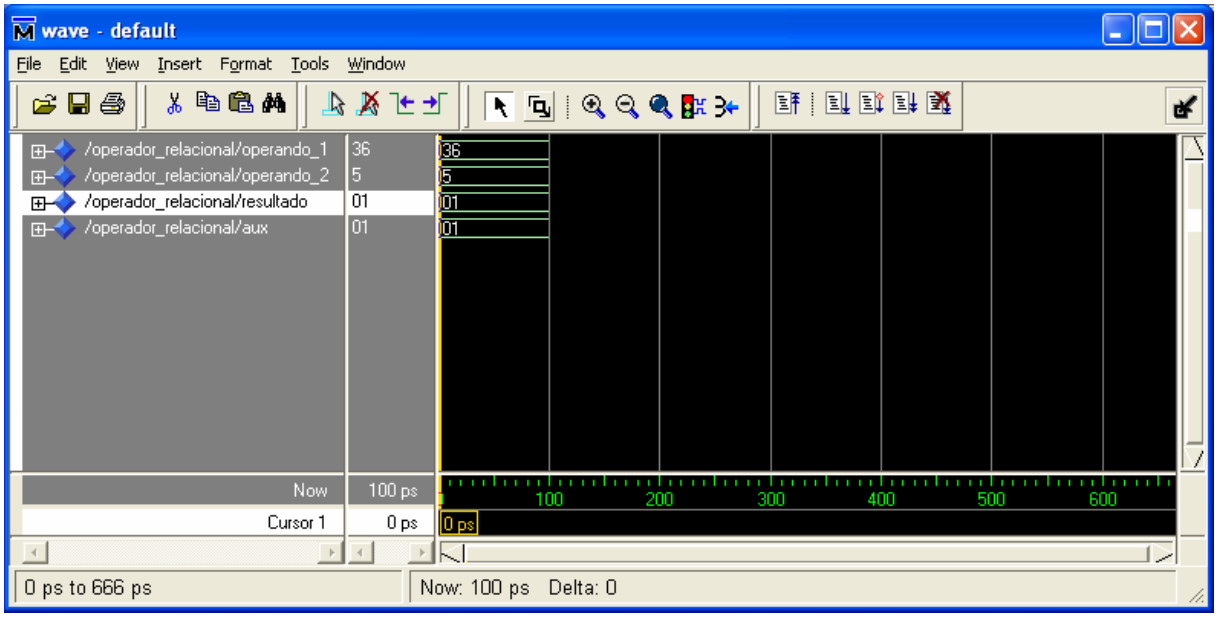


Figura 101 – Operando_1 maior

Na figura 102 é demonstrado um teste em que as entradas são iguais, e como resultado se obtém o valor “00”, indicando que nenhum é maior.

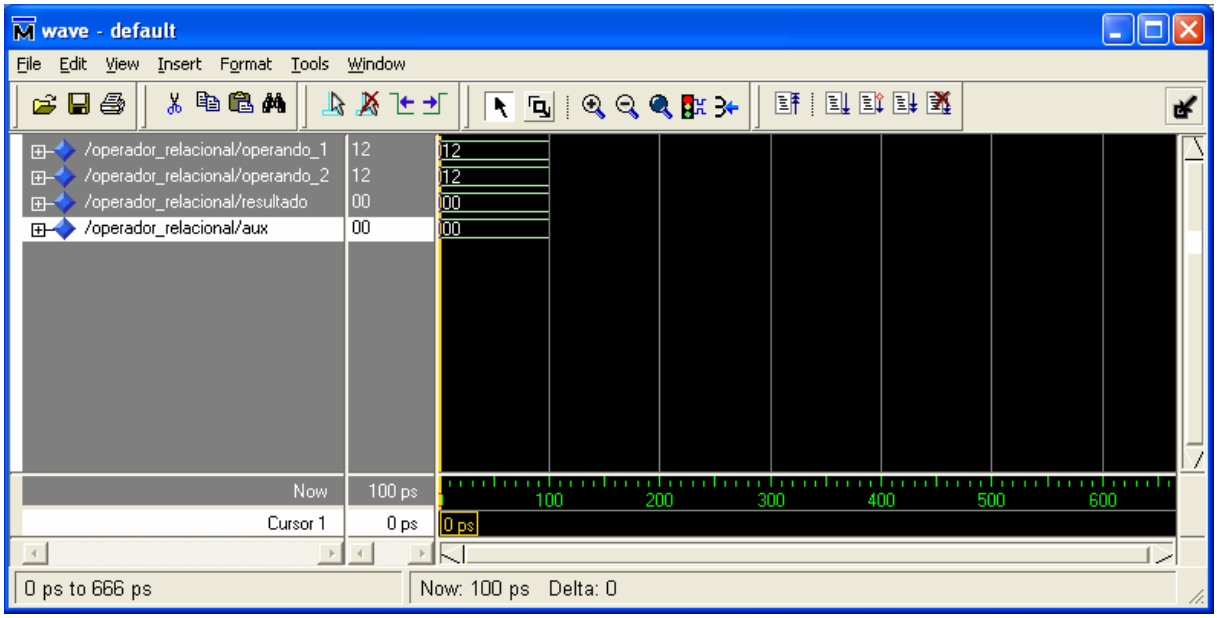


Figura 102 – Operação Igual

APÊNDICE E – Montador

Na figura 103 está descrito o montador implementado, observe que ele possui o botão abrir arquivo, este botão é utilizado para abrir um arquivo já existente, o botão salvar é utilizado par salvar alguma alteração feita no programa ou criar um novo arquivo.

O montador realiza a montagem do arquivo bit quando for aberto ou salvo um arquivo, após realizar a montagem pode-se criar um arquivo no formato para a entidade ROM através do botão criar arquivo binário, ou visualizar as instruções em bits no próprio montador com o botão visualizar arquivo binário, que retorna ao arquivo fonte do programa se ele for pressionado de novamente.

Observe que as linhas iniciadas com # são comentários.

Nas instruções que necessitam de um valor, como o "saltar_mais", deve-se informar um valor em hexadecimal iniciado com o caractere #. Por exemplo, para saltar 16 posições da memória a partir do endereço atual, usa-se o comando: "**saltar_mais #10;**".

Quando for necessário inserir uma constante em uma posição da memória ROM, deve ser informado um número em hexadecimal iniciado com o caractere & e finalizado com um ponto e vírgula. Por exemplo, se for escrito: "**&FF;**", o montador irá converter para um número binário de 32 bits.

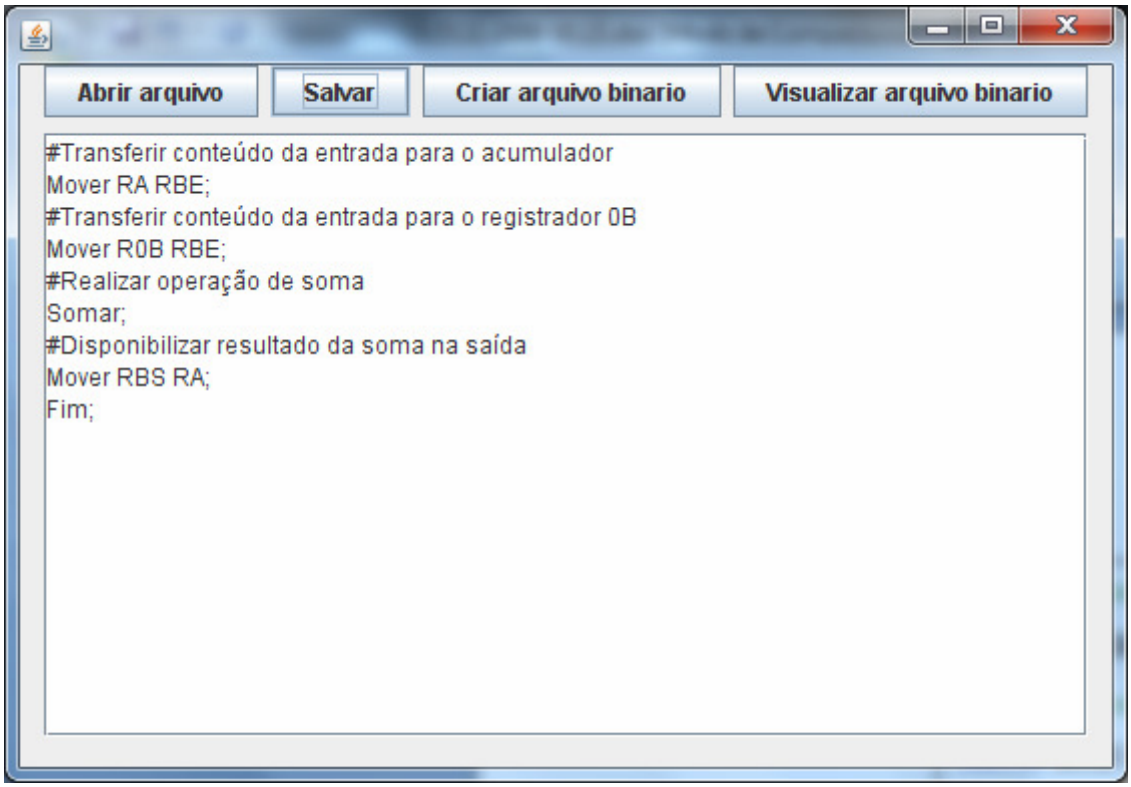


Figura 103 – Montador implementado em JAVA