CENTRO UNIVERSITÁRIO FEI
LEANDRO POLONI DANTAS

**DESENVOLVIMENTO DE UM INOVADOR MICROKERNEL POR HARDWARE
PARA MICROCONTROLADORES FOCANDO NAS APLICAÇÕES DE SISTEMAS
BASEADOS EM TAREFAS**

São Bernardo do Campo

2018

LEANDRO POLONI DANTAS

**DESENVOLVIMENTO DE UM INOVADOR MICROKERNEL POR HARDWARE
PARA MICROCONTROLADORES FOCANDO NAS APLICAÇÕES DE SISTEMAS
BASEADOS EM TAREFAS**

Tese apresentada ao Centro Universitário FEI,
como parte dos requisitos necessários para
obtenção do título de Doutor em Engenharia
Elétrica. Orientado pelo Prof. Dr. Salvador
Pinillos Gimenez.

São Bernardo do Campo

2018

Dantas, Leandro Poloni.

Desenvolvimento de um inovador microkernel por hardware para microcontroladores focando nas aplicações de sistemas baseados em tarefas / Leandro Poloni Dantas. São Bernardo do Campo, 2018.

194 p. : il.

Tese - Centro Universitário FEI.

Orientador: Prof. Dr. Salvador Pinillos Gimenez.

1. arquitetura. 2. microcontrolador. 3. microkernel por hardware. 4. sistemas baseados em tarefas. 5. RTOS. I. Gimenez, Salvador Pinillos, orient. II. Título.

Aluno: Leandro Poloni Dantas

Matrícula: 514203-9

Título do Trabalho: Desenvolvimento de um inovador microkernel por hardware para microcontroladores focando nas aplicações de sistemas baseados em tarefas.

Área de Concentração: Nanoeletrônica e Circuitos Integrados

Orientador: Prof. Dr. Salvador Pinillos Gimenez

Data da realização da defesa: 24/05/2018

ORIGINAL ASSINADA

Avaliação da Banca Examinadora

São Bernardo do Campo, 24 / 05 / 2018.

MEMBROS DA BANCA EXAMINADORA

Prof. Dr. Salvador Pinillos Gimenez	Ass.: _____
Prof. Dr. Renato Camargo Giacomini	Ass.: _____
Prof. Dr. Orlando Del Bianco Filho	Ass.: _____
Prof. Dr. Roberto D'Amore	Ass.: _____
Prof. Dr. Cesar Giacomini Penteado	Ass.: _____

A Banca Examinadora acima-assinada atribuiu ao aluno o seguinte:

APROVADO ☒

REPROVADO ☐

VERSÃO FINAL DA TESE

ENDOSSO DO ORIENTADOR APÓS A INCLUSÃO DAS
RECOMENDAÇÕES DA BANCA EXAMINADORA

Aprovação do Coordenador do Programa de Pós-graduação

Prof. Dr. Carlos Eduardo Thomaz

Dedico este trabalho aos meus pais Antônio Augusto e Yara, e à minha avó Olga.

AGRADECIMENTOS

Agradeço sobretudo a Deus pela vida concedida, pelo socorro nos momentos de aflição, e pela oportunidade de aperfeiçoamento moral e intelectual.

Agradeço aos meus pais por terem, desde que eu era criança, me incentivado ao estudo e pelo esforço que fizeram para que isso fosse possível.

Agradeço à minha esposa Priscila pelo apoio contínuo.

Agradeço ao meu orientador e amigo professor Dr. Salvador Pinillos Gimenez, o qual tenho profunda admiração e gratidão por tudo que me ensinou, pela dedicação, e pela paciência durante esses quatro anos de trabalho.

Agradeço ao Centro Universitário FEI e à Capes pela bolsa de estudos, e por acreditarem no meu trabalho, oferecendo-me condições ideais para o desenvolvimento deste projeto de pesquisa.

Agradeço ao SENAI-SP, empresa na qual sou funcionário, em especial aos professores Augusto Lins de Albuquerque Neto e Marcos Antônio Felizola, diretor e coordenador técnico respectivamente, pelo apoio oferecido.

Agradeço ao professor Dr. Rodolfo Azevedo da UNICAMP-SP pelas contribuições valiosas oferecidas durante sua participação na banca de qualificação deste trabalho de doutorado e em reunião subsequente.

Agradeço também a senhora Cintia Pires Miranda, *program manager* da Macnica DHW, Brasil, pela doação do kit de desenvolvimento de FPGA ao grupo de pesquisa do Prof. Dr. Salvador Pinillos Gimenez no Centro Universitário FEI, que foi utilizado para validação experimental deste meu projeto de pesquisa de doutorado.

“O que eu ouço, eu esqueço. O que eu vejo, eu lembro. O que eu faço, eu entendo.”

Confúcio, 551-479 AC

RESUMO

Nesse projeto de pesquisa de doutorado foi desenvolvida uma inovadora abordagem de hardware, com objetivo de aumentar o desempenho dos sistemas baseados em tarefas, que se utilizam de microcontroladores com um único núcleo. Para isso um *microkernel* foi desenvolvido em hardware e precisamente interconectado à CPU e à memória de dados de um microcontrolador Plasma. Seu papel é operar paralelamente à CPU, realizando por hardware o escalonamento e o despacho de tarefas. Essa abordagem tende a minimizar o tempo gasto para trocas das tarefas que devem ser executadas pela CPU. Além disso, a arquitetura desenvolvida não impacta na forma como as instruções são executadas e adiciona apenas um único banco de registradores ao hardware da CPU. Sendo assim, ela é adequada tanto para a aplicação em dispositivos de hardware programável, como também para a implementação de circuitos integrados dedicados (*Application-Specific Integrated Circuit*, ASIC).

Os resultados experimentais obtidos, quando o dispositivo proposto foi sintetizado em um *Field Program Gate Array* (FPGA), mostram que é possível realizar trocas de tarefas em até 2 ciclos de *clock*, que representa uma redução em até 99% do tempo médio gasto em relação aos sistemas convencionais, no qual o *microkernel* é desenvolvido por software. Além disso, o desempenho do microcontrolador modificado se mantém estável mesmo quando o tempo entre trocas de tarefas é menor que 100 μ s, considerando uma frequência de *clock* de 25 MHz, ao passo que, na abordagem por software, o desempenho em processamento é degradado em até 79% em relação ao máximo observado quando a troca de tarefas ocorre em períodos superiores a 10 ms.

Do ponto de vista energético, o *microkernel* por hardware proposto foi analisado a partir do leiaute gerado automaticamente utilizando-se as ferramentas da CADENCE para fabricação de um ASIC, e os resultados mostram que ele consome apenas 7% da potência elétrica dinâmica do microcontrolador.

Além disso, os resultados obtidos por esse projeto de pesquisa fortalecem a ideia que essa abordagem inovadora possibilita aplicações que demandam tempos curtos entre trocas de tarefas, tais como aqueles que demandam intenso processamento de entradas e saídas (comunicação serial de dados, controle de motores e codificação/decodificação de áudio) e aplicações de alta tecnologia (*clusters* de máquinas virtuais em plataformas de nuvem).

Palavras-chave: Arquitetura. Microcontrolador. *Microkernel* por Hardware. RTOS. Sistemas Baseados em Tarefas. Sistemas Microprocessados de Tempo Real.

ABSTRACT

In this doctoral research project was developed an innovative hardware approach, aiming to increase the performance of the task-based systems, which use microcontrollers with a single core. For this, a microkernel was developed in hardware and precisely interconnected to the CPU and the data memory of a Plasma microcontroller. Its role is to operate parallel to the CPU, performing hardware scheduling and dispatching tasks. This approach tends to minimize the time spent to switch the tasks that must be executed by the CPU. In addition, the architecture developed does not impact how the instructions are executed and adds only a single register bank to the CPU hardware. Therefore, it is ideal for use in programmable hardware devices, as well as in the implementation of application-specific integrated circuits (ASICs).

The experimental results obtained, when the proposed device was synthesized in a Field Program Gate Array (FPGA), show that it is possible to carry out task switches in up to 2 clock cycles, representing a reduction of up to 99% in the average time spent compared to conventional systems, where the microkernel is developed by software. In addition, the performance of the modified microcontroller remains stable even when the time between task switches is shorter than 100 μ s, considering a clock frequency of 25 MHz, whereas, in the software approach, the performance in processing is degraded in up to 79% in relation to the maximum observed when the task switches occur in periods longer than 10 ms.

From the energy point of view, the proposed hardware microkernel was analyzed from the layout automatically generated using Cadence tools for the manufacture of an ASIC, and the results show that it consumes only 7% of the dynamic electrical power of the microcontroller.

In addition, the results obtained by this research project reinforce the idea that this innovative approach allows applications that require short times between task switches, such as those related to I/O-intensive processing (the serial data communication, motor control, and audio coding/decoding) and high-end applications (clusters of virtual machines on cloud platforms).

Keywords: Architecture. Microcontroller. Microkernel by Hardware. RTOS. Task-Based Systems. Microprocessed Real-time Systems.

LISTA DE ILUSTRAÇÕES

Figura 1 – Execução de instruções sem <i>pipeline</i> (a) e com <i>pipeline</i> (b).....	46
Figura 2 – Memória com três portas.	48
Figura 3 – Arquitetura MIPS simplificada.	49
Figura 4 – Diagrama de blocos da CPU do microcontrolador Plasma.	55
Figura 5 – Técnicas de projeto de sistemas embarcados do tipo <i>low-end</i> . Cada hexágono representa uma atividade ou tarefa. <i>Polled Loop</i> (a), <i>Cyclic Code Structure</i> (b) e <i>State-Driven Code</i> (c).	59
Figura 6 – Tipos de sistemas baseados em interrupções. Cada hexágono representa uma atividade ou ISR. <i>Interrupt-Only Systems</i> (a) e <i>Background/Foreground Systems</i> (b).	60
Figura 7 – Divisão de um sistema operacional em camadas funcionais.	61
Figura 8 – Fila de tarefas aguardando para serem executadas.	63
Figura 9 – Fluxo de estados de uma tarefa num sistema de tempo real.	65
Figura 10 – Fluxo de execução de duas tarefas gerenciadas por um <i>microkernel</i>	66
Figura 11 – Esquema de um <i>mailbox</i>	66
Figura 12 – Classificação dos circuitos integrados digitais utilizados nos sistemas digitais. ..	71
Figura 13 – Metodologia de projetos de circuitos integrados digitais com lógica programável.	72
Figura 14 – Kit de desenvolvimento DE1-SoC da Terasic Technologies.	73
Figura 15 – Fluxo de dados para realização da síntese do <i>gate-level netlist</i> no ambiente do Encounter RTL Compiler.	75
Figura 16 – Exemplo de blocos hierárquicos (entidades) do projeto do circuito do microcontrolador IHM-Plasma e área reservada para o roteamento do leiaute do ASIC.	77
Figura 17 – Etapas do fluxo do projeto do ASIC.	78
Figura 18 – Fluxo de execução de tarefas com <i>microkernel</i> por software (a) e utilizando o IHM (b).	80
Figura 19 – Diagrama de blocos simplificado da arquitetura do microcontrolador IHM-Plasma.	83
Figura 20 – Alteração realizada no conjunto de entradas do Contador de Programa da CPU Plasma para que fosse possível a integração da CPU com o IHM.	85
Figura 21 – Implementação de um adicional IRB à CPU Plasma (em xadrez) para que seja possível a integração da CPU com o IHM.	86

Figura 22 – Integração de uma porta de entrada e saída de dados e uma UART a uma CPU por meio de I/O mapeada em memória de acesso aleatório (<i>Random Access Memory</i> , RAM). Em (a) temos a visão física do hardware e em (b) a visão lógica do mesmo.	87
Figura 23 – Divisão da memória interna do microcontrolador IHM-Plasma para uso do <i>microkernel</i> por hardware.	89
Figura 24 – Modelo de distribuição dos TCBs na memória RAM.	90
Figura 25 – Hierarquia de entidades declaradas em linguagem VHDL que compõe o hardware do IHM.	93
Figura 26 – Bloco Controlador de <i>Reset</i> (obtido da ferramenta RTL Viewer do Quartus II)..	94
Figura 27 – Diagrama simplificado de entradas e saída do bloco escalonador do IHM.....	95
Figura 28 – Diagrama de entradas e saída do bloco escalonador do IHM (obtido da ferramenta RTL Viewer do Quartus II).	96
Figura 29 – Exemplo de distribuição de estados das tarefas de 0 a 7 no registrador <i>task_state7_0</i>	97
Figura 30 – Exemplo com três vetores de tarefas com mesma prioridade utilizados pelo bloco escalonador para definição da próxima tarefa a ser executada.....	98
Figura 31 – Exemplo da árvore binária utilizada para seleção de próxima tarefa a ser executada.	99
Figura 32 – Exemplo do fluxo de dados de <i>backup</i> da última tarefa interrompida (a) e de restauração da próxima tarefa a ser executada (b).	100
Figura 33 – Diagrama de entradas e saídas do bloco gerenciador de contexto do IHM (obtido da ferramenta RTL Viewer do Quartus II).	101
Figura 34 - Exemplo do fluxo de dados entre os blocos de temporização e controle (à esquerda) e o gerenciador de contexto (à direita) para realização de uma troca de contexto.	103
Figura 35 – Três primeiros blocos contadores decrescentes para temporização do período de bloqueio das tarefas periódicas 1, 2 e 3 (obtido da ferramenta RTL Viewer do Quartus II).	105
Figura 36 – Diagrama simplificado de entradas e saídas do contador de <i>time slices</i> (a) e o diagrama de tempo com suas <i>flags</i> de sinalização (b).	107
Figura 37 – FSM Motor (obtido da ferramenta State Machine Viewer do Quartus II).	108
Figura 38 – Identificação das regiões de possível pedido de troca assíncrona de tarefas em um <i>time slice</i>	111
Figura 39 – FSM PC_Backup (obtido da ferramenta State Machine Viewer do Quartus II).	112
Figura 40 – MSFM Task_State (obtido da ferramenta State Machine Viewer do Quartus II).	114

Figura 41 – Exemplo do fluxo de dados realizado pela FSM Task_State para a atualização dos estados das tarefas gerenciadas pelo IHM.....	115
Figura 42 – Exemplo do fluxo de execução de tarefas com detalhamento da operação do IHM (as setas indicam os períodos em que a CPU é interrompida).	117
Figura 43 – Exemplo do fluxo de dados entre a CPU Plasma modificada (à esquerda) e o <i>microkernel</i> por hardware proposto (à direita) com o objetivo de permitir o escalonamento de tarefas por hardware.....	118
Figura 44 – Diagrama de entradas e saídas do IHM (obtido da ferramenta RTL Viewer do Quartus II).	119
Figura 45 – Fluxograma da sub-rotina interrupt_service_routine do arquivo boot.asm, onde à esquerda é descrita a versão original (a) e à direita é descrita a versão modificada para aceitar escalonamento de tarefas por software (b).	135
Figura 46 – Leiaute do microcontrolador IHM-Plasma.....	151
Figura 47 – Esquema elétrico do IHM-Plasma (obtido da ferramenta Schematic Viewer do EDI).....	152
Figura 48 – Fotografia da tela de um osciloscópio com a representação quantitativa dos tempos gastos para execução das etapas necessárias para uma troca de tarefas síncronas e assíncronas realizada por um <i>microkernel</i> por software.....	158
Figura 49 – Fotografia da tela de um osciloscópio com a representação quantitativa dos tempos gastos para execução de uma troca síncrona de tarefas e uma troca assíncrona de tarefas com base na monitoração do sinal switch_out_RT e realizadas pelo <i>microkernel</i> por hardware proposto (IHM).	159
Figura 50 – Frequência de execução de uma tarefa de controle (taskControle) em função do <i>time slice</i> , considerando-se diferentes números de tarefas ativas, quando o escalonamento é realizado por algoritmo em hardware (IHM) e em software (abordagem tradicional).	164
Figura 51 – Diagrama funcional da aplicação do IHM-Plasma como controlador de um console <i>Arcade</i>	167
Figura 52 – Tela principal da IDE MARS utilizada para criação de programas para o microcontrolador IHM-Plasma.....	181
Figura 53 – Procedimento para conversão de arquivo binário em arquivos no padrão Intel HEX utilizados como programa do microcontrolador IHM-Plasma.	183
Figura 54 – Instanciação das <i>Memory Megafunctions</i> no arquivo ram.vhd com destaque para os arquivos hexadecimais (correspondentes ao programa a ser gravado) associada a cada uma delas.	184

Figura 55 – Linhas de código com <i>scripts</i> inseridos no arquivo makefile para compilação dos projetos em C baseados no arquivo plasma_rt.	185
Figura 56 – Procedimento para compilação do projeto C baseado no arquivo plasma_rt.c e conversão do arquivo binário gerado em arquivos no padrão Intel HEX utilizados como programa do microcontrolador IHM-Plasma.	186
Figura 57 – Instanciação das <i>Memory Megafunctions</i> no arquivo ram.vhd com destaque para os arquivos hexadecimais (correspondentes ao programa a ser gravado) associada a cada uma delas.....	187
Quadro 1 – Formato das instruções do tipo R da arquitetura MIPS.	52
Quadro 2 – Formato das instruções do tipo I da arquitetura MIPS.	52
Quadro 3 – Formato das instruções do tipo J da arquitetura MIPS.	53
Quadro 4 – Trecho do código VHDL responsável pelo processo de aceitação de um pedido de interrupção e de uma troca de tarefa pela CPU.	123
Quadro 5 – Trecho do código VHDL responsável pelo pela hierarquia na decisão de desvios por interrupção e pela troca de tarefa.....	123
Quadro 6 – Trecho do código <i>Assembly</i> responsável pela configuração das tarefas e do IHM.	129
Quadro 7 – Trecho do código <i>Assembly</i> responsável pela declaração das tarefas.	130
Quadro 8 – Sub-rotina entry do arquivo boot.asm, responsável pelo desvio da CPU para função main.....	133
Quadro 9 – Sub-rotina interrupt_service_routine do arquivo boot.asm, responsável pelo início e pelo fim do tratamento de uma interrupção pela CPU.	134
Quadro 10 – Sub-rotina OS_AsmInterruptEnable do arquivo boot.asm.	136
Quadro 11 – Função OS_InterruptServiceRoutine do arquivo no_os_rt.c.	137
Quadro 12 – Lista de registradores e memórias presentes no arquivo plasma_rt.h.....	142
Quadro 13 – Estrutura da memória FIFO declarada e utilizada como canal de comunicação entre tarefas.....	143
Quadro 14 – Vetor de <i>status</i> de interrupções do microcontrolador IHM-Plasma (arquivo plasma_RT.vhd).....	145
Quadro 15 – Função trataInt responsável pelo tratamento de interrupções da CPU (arquivo plasma_rt.c),	146
Quadro 16 – Funções-tarefas utilizadas para criação de tarefas (arquivo plasma_rt.c).....	147

Quadro 17 – Função principal onde as tarefas são criadas e o escalonamento é configurado (arquivo <i>plasma_rt.c</i>).	148
Quadro 18 – Árvore hierárquica dos arquivos RTL dos blocos componentes do projeto e declarados no <i>script</i> utilizado pela ferramenta Encounter RTL Compiler.....	150
Quadro 19 – Funções-tarefas para teste de desempenho de processamento (arquivo <i>plasma_rt.c</i>).	163

LISTA DE TABELAS

Tabela 1 – Comparativo entre a proposta deste projeto de pesquisa e os trabalhos relacionados a este, que foram descritos no item 1.2.	38
Tabela 2 – Tabela comparativa entre as características das arquiteturas RISC e CISC de processadores.	45
Tabela 3 – Tabela comparativa entre o número de estágios de <i>pipeline</i> e ciclos de <i>clock</i> por instrução de processadores com 8, 16, 32 e 64 bits.	47
Tabela 4 – Conjuntos de registradores internos da arquitetura dos processadores MIPS.	51
Tabela 5 – Características principais do IHM-Plasma.	83
Tabela 6 – Mapa de memória do microcontrolador Plasma original e do microcontrolador IHM-Plasma.	91
Tabela 7 – Mapeamento de memória RAM reservada para os TCBs utilizados pelo IHM.	92
Tabela 8 – Código de possíveis estados de uma tarefa.	97
Tabela 9 – Mapa de transição de estados da FSM Motor.	109
Tabela 10 – Saídas ativadas em função do estado ativo da FSM Motor.	109
Tabela 11 – Mapa de transição de estados da FSM PC_Backup.	113
Tabela 12 – Saídas ativas em função do estado ativo da FSM PC_Backup.	113
Tabela 13 – Mapa de transição de estados da FSM Task_State.	114
Tabela 14 – Saídas ativas em função do estado ativo da FSM Task_State.	115
Tabela 15 – Hierarquia das entidades dos blocos que descrevem o microcontrolador IHM-Plasma (para entradas na tabela com dois números listados: os números entre parênteses indicam o número de recursos do tipo determinado usado apenas pela entidade específica; os números listados fora dos parênteses indicam o total de recursos do tipo dado usado pela entidade específica e todas as suas subentidades na hierarquia).	125
Tabela 16 – Consumo de recursos de hardware do FPGA em função da versão do microcontrolador sintetizada (o incremento percentual do IHM-Plasma em relação ao Plasma original é mostrado entre parênteses).	126
Tabela 17 – Frequência máxima de operação dos microcontroladores Plasma e IHM-Plasma.	126
Tabela 18 – Mapeamento de memória RAM para a definição da faixa de memória reservada para a pilha (<i>stack</i>) de cada tarefa.	131
Tabela 19 – Hierarquia das entidades do leiaute gerado para o IHM-Plasma com a representação do consumo de <i>gates</i> (<i>standard cells</i>) e área do dispositivo.	153

Tabela 20 – Percentual de recursos de hardware utilizados para implementação do IHM-Plasma em FPGA e ASIC.....	154
Tabela 21 – Análise do consumo de potência estática e dinâmica do ASIC correspondente ao IHM-Plasma.....	155
Tabela 22 – Períodos medidos para execução das etapas necessárias para uma troca de tarefas realizada por um <i>microkernel</i> por software.....	158
Tabela 23 – Medidas feitas para análise do tempo necessário para troca de tarefas, em que os ganhos de desempenho e a redução nos períodos do <i>microkernel</i> por hardware (IHM) são calculados em relação ao <i>microkernel</i> por software.	160
Tabela 24 – <i>Time slice</i> efetivo ($Ef\%$) em função do período do <i>time slice</i>	161
Tabela 25 – Variação no <i>time slice</i> efetivo ($\Delta Ef\%$) em função do número de ciclos de <i>clock</i> por <i>time slice</i> e do tipo de <i>microkernel</i>	165
Tabela 26 – Máxima troca de tarefas considerando a mesma degradação do $Ef\%$	166

LISTA DE ABREVIATURAS E SIGLAS

ABEL	<i>Advanced Boolean Equation Language</i>
AHDL	<i>Altera Hardware Description Language</i>
ALU	Unidade Lógica e Aritmética (<i>Arithmetic Logic Unit</i>)
AMD	<i>Advanced Micro Devices</i>
ARM	<i>Advanced RISK Machine</i>
ASCII	Código Padrão Americano para o Intercâmbio de Informação (<i>American Standard Code for Information Interchange</i>)
ASIC	Circuito Integrado para Aplicação Específica (<i>Application-Specific Integrated Circuit</i>)
Bps	Bits Por Segundo
CAD	Conversor Analógico Digital
Capes	Coordenação de Aperfeiçoamento de Pessoal de Nível Superior
CI	Circuito Integrado
CISC	Computador com um Conjunto Complexo de Instruções (<i>Complex Instruction Set Computer</i>)
CMOS	Metal-Óxido-Semicondutor Complementar (<i>Complementary Metal-Oxide-Semiconductor</i>)
CPI	Ciclo de <i>Clock</i> por Instrução
CPLD	Dispositivo Lógico Programável Complexo (<i>Complex Programmable Logic Device</i>)
CPU	Unidade Central de Processamento (<i>Central Processing Unit</i>)
DDR	Taxa Dupla de Transferência (<i>Double-Data-Rate</i>)
DRC	Verificação das Regras de Projeto (<i>Design Rule Check</i>)
EDA	Automação de Projeto Eletrônico (<i>Electronic Design Automation</i>)
EDI	Encounter Digital Implementation
EEPROM	Memória de Somente Leitura Programável e Apagável Eletricamente (<i>Electrically-Erasable Programmable Read-Only Memory</i>)
ESC	Escalonador
FEI	Fundação Educacional Inaciana Pe. Sabóia de Medeiros
FPGA	Matriz de Portas Lógicas Programáveis em Campo (<i>Field Programmable Gate Array</i>)

FPLA	Matriz Lógica Programável em Campo (<i>Field Programmable Logic Array</i>)
FSM	Máquina de Estados Finitos (<i>Finite-State Machine</i>)
GAL	Lógica de Matriz Genérica (<i>Generic Array Logic</i>)
GAM	Gerenciador de Acesso à Memória
GC	Gerenciador de Contexto
GCC	GNU <i>Compiler Collection</i>
GSD	<i>Graphic Data System</i>
HCPLD	PLD de Alta Capacidade (<i>High-Capacity PLD</i>)
HDL	Linguagem de Descrição de Hardware (<i>Hardware Description Language</i>)
HPS	<i>Hard Processor System</i>
IEEE	<i>Institute of Electrical and Electronics Engineers</i>
IHM	<i>Microkernel</i> por Hardware Intertravado (<i>Interlocked Hardware Microkernel</i>)
ICP	Instrução por Ciclo de <i>Clock</i>
I/O	Entrada/Saída (<i>Input/Output</i>)
IoT	<i>Internet das Coisas (Internet of Things)</i>
IRB	Banco de Registradores Interno (<i>Internal Register Bank</i>)
IS	Serviço de Interrupção (<i>Interrupt Service</i>)
ISA	Arquitetura do Conjunto de Instruções (<i>Instruction Set Architecture</i>)
ISR	Rotina de Serviço de Interrupção (<i>Interrupt Service Routine</i>)
LC	Célula Lógica (<i>Logic Cell</i>)
LCD	Display de Cristal Líquido (<i>Liquid Crystal Display</i>)
LED	Diodo Emissor de Luz (<i>Light Emitting Diode</i>)
LUT	<i>Look-up Table</i>
MIPS	<i>Microprocessor without Interlocked Pipeline Stages</i>
MMMC	<i>Multi-Mode Multi-Corner Analysis</i>
Nop	Nenhuma Operação (<i>No Operation</i>)
Opcode	Código Operacional (<i>Operation Code</i>)
PAL	Lógica de Matriz Programável (<i>Programmable Array Logic</i>)
PDK	Kit de Desenvolvimento de Plugar (<i>Plug-in Development Kit</i>)
PC	Contador de Programa (<i>Program Counter</i>)
PIC	Controlador de Interface Periférica (<i>Peripheral Interface Controller</i>)
PLA	Matriz Lógica Programável (<i>Programmable Logic Array</i>)
PLD	Dispositivo Lógico Programável (<i>Programmable Logic Device</i>)

PROM	Memória de Somente Leitura Programável (<i>Programmable Read-Only Memory</i>)
RAM	Memória de Acesso Aleatório (<i>Random Access Memory</i>)
RC	Encounter RTL Compiler
ROM	Memória de Somente Leitura (<i>Read-Only Memory</i>)
RTL	Nível de Transferência de Registradores (<i>Register Transfer Level</i>)
RTOS	Sistema Operacional de Tempo Real (<i>Real-Time Operating System</i>)
RTS	Sistema de Tempo Real (<i>Real-Time System</i>)
RISC	<i>Reduced Instruction Set Computer</i> (Computador com um Conjunto Reduzido de Instruções)
SDC	Restrições de Projeto Synopsys (<i>Synopsys Design Constraints</i>)
SDRAM	RAM Dinâmica Sincronizada (<i>Synchronous Dynamic RAM</i>)
SFR	Registrador com Função Especial (<i>Special Function Register</i>)
SoC	Sistema em um <i>chip</i> (<i>System-on-Chip</i>)
SP	Ponteiro de Pilha (<i>Stack Pointer</i>)
SPI	<i>Serial Peripheral Interface</i>
SPLD	PLD Simples (<i>Simple PLD</i>)
SRAM	RAM estática (<i>Static RAM</i>)
T&C	Temporização e Controle
TCB	Bloco de Controle de Tarefa (<i>Task Control Block</i>)
TCP/IP	Protocolo de Controle de Transmissão/Protocolo de Internet (<i>Transmission Control Protocol/Internet Protocol</i>)
TFT	Transistor de Película Fina (<i>Thin-Film Transistor</i>)
UART	Transmissor Receptor Assíncrono Universal (<i>Universal Asynchronous Receiver Transmitter</i>)
ULA	Unidade Lógica e Aritmética
USB	Barramento Universal Serial (<i>Universal Serial Bus</i>)
VHDL	<i>Very High Speed Integrated Circuit Hardware Description Language</i>

LISTA DE SÍMBOLOS

B	Byte
$\Delta Ef\%$	Variação no <i>time slice</i> efetivo (%)
e_{cs}	Tempo para troca de tarefas (s)
$Ef\%$	<i>Time slice</i> efetivo (s)
e_i	Tempo de execução (s)
e_{sch}	Tempo para decidir a próxima tarefa (s)
Hz	Hertz
i	Índice
k	Quilo, 10^3
m	Metro
M	Mega, 10^6
ms	Milissegundo, 10^{-3} segundos
ns	Nano segundo, 10^{-9} segundos
p_{cs}	Período entre repetições de troca de tarefas (s)
p_i	Período de repetição (s)
p_{sch}	Período entre repetições para decidir a próxima tarefa
s	Segundo
t_i	Tempo de Execução (s)
U	Fator de ocupação da CPU (%)
μ	Micro, 10^{-6}

SUMÁRIO

1	INTRODUÇÃO	33
1.1	CENÁRIO ATUAL.....	33
1.2	TRABALHOS RELACIONADOS.....	34
1.3	MOTIVAÇÃO E OBJETIVOS	35
2	CONCEITOS FUNDAMENTAIS	41
2.1	ARQUITETURAS DOS PROCESSADORES E COMPUTADORES.....	41
2.1.1	Classificação das arquiteturas de computadores	43
2.1.2	<i>Pipeline</i>	45
2.1.3	Memórias multiportas.....	47
2.2	ARQUITETURA MIPS	48
2.2.1	Conjunto de instruções do MIPS	49
2.2.2	Blocos básicos da microarquitetura MIPS	53
2.2.3	Microcontrolador Plasma	54
2.3	SISTEMAS EMBARCADOS	56
2.3.1	Definição de sistema embarcado	57
2.3.2	Tipos de sistemas embarcados	57
2.3.3	Técnicas de projeto de sistemas embarcados	58
2.4	SISTEMA OPERACIONAL DE TEMPO REAL	61
2.4.1	<i>Microkernel</i>	62
2.4.2	Outras funcionalidades que compõem o <i>Kernel</i>	66
2.4.3	Benefícios e limitações dos RTOSs.....	67
2.5	HARDWARE DIGITAL PROGRAMÁVEL.....	69
2.5.1	Dispositivos lógicos programáveis	69
2.5.2	Linguagens de descrição de hardware	71
2.5.3	Fluxo de projeto de CIs digitais implementados em PLDs	72
2.6	CIRCUITO INTEGRADO DEDICADO	74

2.6.1	Encounter RTL Compiler	74
2.6.2	Encounter Digital Implementation	76
3	O PROJETO DE PESQUISA	79
3.1	PROPOSTA DE UM <i>MICROKERNEL</i> POR HARDWARE PARA SISTEMAS BASEADOS EM TAREFAS	79
3.2	IMPLEMETAÇÃO DA ARQUITETURA DO IHM-PLASMA.....	82
3.2.1	Alterações na CPU do microcontrolador Plasma.....	84
3.2.2	Alterações realizadas na memória do microcontrolador Plasma	86
3.2.3	Implementação dos blocos internos ao IHM	92
3.2.3.1	<i>Controlador de reset</i>	93
3.2.3.2	<i>Escalonador</i>	94
3.2.3.3	<i>Gerenciador de contexto</i>	100
3.2.3.4	<i>Temporização e controle</i>	103
3.2.4	Interface entre o IHM e o microcontrolador Plasma modificado.....	116
3.2.4.1	<i>Mecanismo de troca de tarefas implementado por hardware</i>	121
3.2.5	Dados extraídos do projeto do IHM-Plasma em FPGA.....	123
3.3	PROGRAMAÇÃO DO MICROCONTROLADOR IHM-PLASMA EM LINGUAGEM <i>ASSEMBLY</i>	127
3.4	ARCABOUÇO (<i>FRAMEWORK</i>) PARA PROGRAMAÇÃO DO MICROCONTROLADOR IHM-PLASMA EM LINGUAGEM C	132
3.4.1	Arquivo de partida do programa e desvio para tratamento de interrupções (boot.asm).....	132
3.4.2	Arquivo de apoio ao programa principal (no_os_rt.c).....	137
3.4.3	Arquivo com a biblioteca de funções do IHM e do <i>microkernel</i> por software (hw_microk.c).....	137
3.4.3.1	<i>Funções de acesso ao IHM</i>	138
3.4.3.2	<i>Funções de acesso ao microkernel implementado por software</i>	139
3.4.4	Arquivos principais para criação dos projetos (plasma_rt.h e plasma_rt.c)	141

3.4.4.1	<i>Memória FIFO para comunicação entre tarefas.....</i>	142
3.4.5	Exemplo de criação de um projeto baseado no escalonamento de tarefas por hardware (IHM) e por software usando linguagem C.....	144
3.5	IMPLEMENTAÇÃO DO LEIAUTE DE UM ASIC DO IHM-PLASMA PARA FUTURA FABRICAÇÃO UTILIZANDO A TECNOLOGIA SIGE BICMOS DE 130 NM DA GLOBALFOUNDRIES	148
3.5.1	Tecnologia de fabricação utilizada no ASIC do IHM-Plasma.....	150
3.5.2	Leiaute do ASIC do IHM-Plasma e suas dimensões.....	151
3.5.3	Características geométricas, físicas e elétricas obtidas do leiaute do ASIC do IHM-Plasma	152
4	RESULTADOS EXPERIMENTAIS OBTIDOS DO IHM-PLASMA IMPLEMENTADO NO KIT DE DESENVOLVIMENTO DE1-SOC DA Terasic..	157
4.1	CARACTERIZAÇÃO ELÉTRICA EXPERIMENTAL DO TEMPO GASTO PARA A REALIZAÇÃO DAS TROCAS DE TAREFAS	157
4.2	CARACTERIZAÇÃO EXPERIMENTAL DO <i>TIME SLICE</i> EFETIVO EM FUNÇÃO DO TEMPO GASTO PARA REALIZAÇÃO DAS TROCAS DE TAREFAS (4.1)	160
4.3	CARACTERIZAÇÃO EXPERIMENTAL DE DESEMPENHO EM FUNÇÃO DO <i>TIME SLICE</i> EM CICLOS DO CLOCK DO PROCESSADOR	162
4.4	CARACTERIZAÇÃO EXPERIMENTAL DE DESEMPENHO DO IHM-PLASMA OPERANDO COMO O GERENCIADOR DO JOGO <i>SPACE INVADERS</i>	167
5	CONSIDERAÇÕES FINAIS, CONCLUSÕES, CONTRIBUIÇÕES E TRABALHOS FUTUROS.....	169
	REFERÊNCIAS.....	173
	GLOSSÁRIO	179
	APÊNDICE A – PROCEDIMENTO PARA GERAÇÃO DOS ARQUIVOS HEXADECIMAIS NO PADRÃO INTEL HEX COM OS PROGRAMAS A SEREM EXECUTADOS PELO MICROCONTROLADOR IHM-PLASMA	181
	APÊNDICE B – CÓDIGO FONTE DO PROGRAMA GRAVADO NA MEMÓRIA ROM DO ASIC PROJETADO.....	189

APÊNDICE C – PUBLICAÇÕES	193
---------------------------------------	------------

1 INTRODUÇÃO

Neste capítulo é apresentado uma breve contextualização do cenário atual com relação ao uso de hardwares programáveis e à demanda da melhoria do desempenho dos sistemas embarcados. Em seguida, são apresentados os trabalhos relacionados a esse projeto de pesquisa. E por fim, é apresentada a motivação deste trabalho e seus principais objetivos.

1.1 CENÁRIO ATUAL

A demanda contínua do mercado consumidor de produtos que agregam algum tipo de eletrônica embarcada é responsável por impulsionar a indústria na utilização de dispositivos com algum grau de inteligência, ou seja, aqueles que são capazes de executar diferentes tarefas e realizar a tomada de decisões para atender a um determinado objetivo (8;20;23;35). Dessa forma, o uso de microcontroladores e dispositivos lógicos programáveis se intensificou nas últimas duas décadas (8;11;17;18;20;23;35;47). Além do desenvolvimento e o aperfeiçoamento dos equipamentos, que antes eram meramente mecânicos ou elétricos (fogões, refrigeradores, etc.), agora integram partes eletrônicas, e estão surgindo com grande velocidade. Observa-se dentro desse contexto que a grande maioria desses novos projetos desenvolvidos pelas indústrias (automotiva, áudio e vídeo, etc.) não utilizam circuitos integrados (CIs) dedicados (28;33).

Consequentemente, os sistemas embarcados (do inglês *embedded systems*: dispositivos inteligentes que se utilizam de processador para aplicações específicas) vem sendo utilizados massivamente (54). Cabe aqui ressaltar que a palavra “embarcado” remete ao “oculto do lado de dentro”, aquele que não pode ser visto, ou seja, o usuário não tem acesso (29;30;54;34).

Um sistema embarcado normalmente define necessidades específicas e para realizá-las existem três abordagens que podem ser seguidas (54):

- Desenvolvimento de um *firmware* para um processador e seu específico hardware;
- Configuração de um hardware¹¹ programável (Dispositivo Lógico Programável, *Programmable Logic Device*, PLD) por meio de uma linguagem de descrição de hardware;

¹ As palavras e expressões estrangeiras que não possuem equivalente em português ou que foram consagradas pelo uso corrente não são grafadas sem itálico ou aspas. Esse documento segue o manual de comunicação do Senado Federal.

- Projeto de um circuito integrado de aplicação específica (*Application-Specific Integrated Circuit*, ASIC), que é viável apenas quando os volumes de produção são significativos, pois o custo do projeto é significativo também (26;33).

Existem várias formas de aumentar o desempenho em termos do processamento dos sistemas embarcados. Uma delas é a implementação de hardware dedicado que trabalha paralelamente com a CPU (processamento paralelo), visando a redução do processamento demandado pelo software (39). Dentro desse cenário, muitos estudos foram apresentados a partir da década de 1990, considerando os sistemas embarcados baseados em tarefas, objetivando-se a redução do tempo necessário para decidir qual será próxima tarefa a ser executada e realizar a mudança de contexto da CPU (chamado de tempo de troca de tarefa) (9;14;15;27;32;38;39;41;50;55). Essa abordagem tem importância especial para os sistemas de tempo real, que se concentram em finalizar a execução de uma tarefa antes que um determinado tempo específico (*deadline*) ocorra, no qual são classificados como sistemas embarcados determinísticos.

1.2 TRABALHOS RELACIONADOS

Existem vários trabalhos que objetivaram a melhoria do desempenho dos sistemas baseados em tarefas. Um deles, cuja autoria é de Nakano et al. (38) publicado em 1995, descreveu sobre a implementação de um circuito integrado com um RTOS por hardware, chamado de STRON. Esse circuito integrado se comunicava com um *microkernel* por software, que fazia a interface entre o hardware e a aplicação. O STRON versão 1 foi implementado em tecnologia CMOS com nó tecnológico de 0,8 μm . Por sua vez, Gaitan et al. (15) propuseram o desenvolvimento de um escalonador por hardware e um conjunto independente de registradores para o *pipeline*, chamado de *Multipipeline Register Architecture* (MPRA). Nesta abordagem, todos os registradores do *pipeline*, bem como os conjuntos de registradores internos da CPU são replicados de acordo com o número de tarefas a serem executadas. O escalonador por hardware é responsável por decidir qual será a próxima tarefa a ser executada e por comutar o conjunto de registradores habilitado na arquitetura da CPU. Para que isso fosse possível, foi implementado um conjunto de instruções, que é decodificado e executado pelo escalonador durante a execução do programa. Um outro trabalho dentro desse cenário foi o dos autores Rafla e Gauba (41) que propuseram um gerenciador de contexto por hardware para aplicações com sistemas operacionais de tempo real (*Real-Time Operating System*, RTOS). Nesta abordagem parte do conjunto interno de registradores (*internal register bank*, IRB) da CPU é replicado de

acordo com o número de tarefas. Analogamente a (41), apenas um dos bancos de registradores é selecionado por vez, que corresponde aquela tarefa que está em curso. Neste caso, também foi necessária a criação de novas instruções para permitir com que o RTOS disparasse uma troca de contexto por hardware. Além dessas publicações, Maruyama et al. (32) também sugeriram e implementaram um RTOS completo por hardware, juntamente com outras funções, de forma a garantir um melhor desempenho no processamento de pacotes TCP/IP (Protocolo de Controle de Transmissão/Protocolo de Internet, *Transmission Control Protocol/Internet Protocol*). Além disso, eles introduziram o conceito de fila virtual para o agendamento das tarefas como uma alternativa para simplificar o gerenciamento das tarefas a serem executadas. O hardware proposto recebeu o nome de ARTESSO e foi implementado em ASICs com as tecnologias de CIs Metal-Óxido-Semicondutor Complementar (*Complementary Metal-Oxide-Semiconductor*, CMOS) de 90 nm e 120 nm, respectivamente. Além dos trabalhos citados anteriormente, o dos autores Ong et. al. (39) propuseram uma outra opção de RTOS por hardware implementado em matriz de portas lógicas programáveis em campo (*Field Programmable Gate Array*, FPGA), que se comunicava com um processador NIOS-II, via os seus barramentos. Apesar dos resultados obtidos com o uso desse inovador hardware serem sempre positivos, observaram a pouca aceitação desse tipo de solução em produtos comerciais, em decorrência da necessidade de mão de obra especializada com profundo conhecimento de sistemas operacionais. Por último, podemos citar o trabalho dos autores Vetromille et. al. (55), que analisaram duas opções de implementação do escalonador de tarefas de um RTOS, sendo que um foi implementado por hardware e o outro foi implementado com o uso de um coprocessador. Além das comparações feitas entre essas duas implementações, elas também foram comparadas com um RTOS totalmente implementado por software. Suas conclusões mostraram que ao retirar a função do escalonamento de tarefas do processamento principal da CPU realizado por software sempre reverte na melhoria do desempenho do sistema como um todo. Neste trabalho, o escalonador por hardware foi conectado com a CPU de um processador MicroBlaze através dos barramentos que dão acesso aos seus periféricos. Os testes dessas diferentes implementações foram feitos exclusivamente com o uso de FPGA.

1.3 MOTIVAÇÃO E OBJETIVOS

As propostas descritas na seção 1.2 são todas favoráveis para os sistemas de múltiplas tarefas (multitarefa) e que utilizam um único processador (núcleo). O que limita o desempenho

desses sistemas é o tempo gasto pelo processador para realizar o procedimento de parada da uma tarefa em execução e preparação de outra tarefa para a execução (troca de tarefas em execução) (4;29;30;44). Neste momento, a CPU deve ser preparada com os parâmetros da tarefa a ser executada (4;29;30;44;54). Isso quer dizer que, uma série de registradores internos precisam ser carregados com informações tais como o endereço da próxima instrução a ser executada (*program counter*), o endereço do topo da pilha (*stack pointer*), o valor do registrador de estado da CPU (*status*) e os valores dos registradores utilizados nas operações com a ULA (4;29;30;44;54). Porém, antes disso, os valores desses mesmos registradores (que armazenam dados da tarefa que está sendo executada) precisam ser salvos geralmente na pilha (*stack*: área reservada de memória volátil para esse fim) (4;29;30;44;54). Assim, a tarefa que sai de execução pode ser executada futuramente mediante a restauração dessas informações que estão guardadas nos registradores internos e na pilha. Todo esse processo de armazenamento (*backup*) das informações relativas a tarefa que está sendo executada e a restauração dos valores dos registradores internos e da pilha da outra tarefa a ser executada é chamado de troca de contexto (*context switching*) (4;29;30;44;54).

Originalmente os processadores usados para operar com o RTOS não possuem um hardware dedicado adicional a sua estrutura para realizar a troca da execução de tarefas por hardware, ou seja, isso é feito por software, que por sua vez consome valiosos ciclos de máquina para realizar a troca de contexto em sistemas de tempo real. Outros pontos complexos a serem gerenciados, é a definição da próxima tarefa a ser executada, ou seja, aquela que possui uma maior prioridade que as outras, além da definição do instante em que ela deve ser executada (4;29;30;44;54).

Dentro do cenário apresentado acima, a principal motivação deste projeto de pesquisa é propor uma nova abordagem por hardware para melhorar o desempenho da CPU em sistemas eletrônicos inteligentes baseados em tarefas. Mais especificamente é a implementação por hardware das principais funções de um sistema operacional de tempo real baseado em tarefas. Dessa forma, os principais objetivos gerais desta pesquisa são:

Dessa forma, os principais objetivos gerais desta pesquisa são:

- I. Criar um hardware dedicado e interligado ao processador que seja capaz de minimizar o tempo necessário para a realização das trocas de execução de tarefas focando nos sistemas operacionais de múltiplas tarefas;
- II. Propor uma solução que possa ser aplicada tanto em dispositivos *soft-cores* (processadores implementados usando PLDs, cujos núcleos são ajustados por

software para atender à demanda de cada projeto) como em dispositivos comerciais (*hard-cores*: processadores dedicados fabricados em silício);

- III. Não adicionar e não alterar o conjunto de instruções do processador;
- IV. Reduzir o tempo para a realização das trocas de tarefas pela CPU para valores além daqueles que são utilizados atualmente e sem afetar o desempenho da CPU;
- V. Minimização da quantidade de registradores internos e tamanho da pilha.

Além dos objetivos gerais citados acima, estão também descritos os principais objetivos específicos deste projeto de pesquisa:

- I. Gerar um protótipo funcional em FPGA da nova arquitetura de hardware proposta;
- II. Criar ferramentas de software que facilitem a programação do processador desenvolvido;
- III. Verificar o desempenho da CPU do sistema proposto por hardware em relação a solução convencional por software;
- IV. Implementação de um leiaute para a solução proposta visando a implementação futura de um ASIC.

Para que esses objetivos gerais e específicos sejam atingidos, este projeto de pesquisa visa implementar por hardware todas as ações relacionadas ao escalonamento de tarefas (definição da próxima tarefa a ser executada num sistema de múltiplas tarefas) e a troca do contexto das informações relativas a tarefa que está sendo executada pelo contexto das informações da próxima tarefa a ser realizada, que geralmente são executadas por software. Essas duas funções dizem respeito a uma parte dos sistemas operacionais de múltiplas tarefas, que no caso é chamada de *microkernel* (29;30). Portanto, um *microkernel* inteiramente por hardware é desenvolvido para um processador de 32 bits e é dividido em três blocos principais: (i) **escalonador** para o gerenciamento das tarefas; (ii) **gerenciador de contexto** para realização da troca de contexto das tarefas, e; (iii) o bloco de **temporização e controle**, que gerencia as atividades dos blocos integrantes ao hardware desenvolvido (CPU, registradores internos, memórias, etc.).

A Tabela 1 apresenta um comparativo entre as principais características da solução proposta em relação a outras propostas apresentadas na seção 1.2 (15;32;38;39;41;55).

Tabela 1 – Comparativo entre a proposta deste projeto de pesquisa e os trabalhos relacionados a este, que foram descritos no item 1.2.

Característica	<i>Microkernel</i> proposto	Gaitan et al. (15)	Rafla e Gauba (41)	Maruyama et al. (32)	Nakano et al. (38)	Ong et. al. (39)	Vetromille et. al. (55)
Hardware	<i>microkernel</i>	escalonador	troca de contexto	RTOS	RTOS	RTOS	escalonador (também como coprocessador)
Nome	IHM	nMRPA	-	ARTESSO	STRON	SEOS	HaRTS (Co-SoRTS)
Processador	Plasma modificado	Baseado no MIPS	Plasma modificado	ARM946	genérico	NIOS-II	MiccroBlaze
Algoritmo de escalonamento	preemptivo baseado em prioridades	preemptivo baseado em prioridades	cooperativo	preemptivo baseado em prioridades	baseado em prioridades, utiliza <i>radix sort</i>	preemptivo baseado em prioridades	baseado em prioridades
Número máximo de tarefas	32	32	-	32	32	64	-
Outros recursos em hardware	tarefas com <i>sleep</i> controlado por hardware	<i>mutex</i>	-	cópia de memória, <i>checksum</i> TCP e rearranjo de cabeçalho	<i>flags</i> de eventos, semáforos e temporizadores	semáforos e mailbox	-
Estratégia principal	replica uma única vez o IRB	replica o pipeline e IRB pelo número de tarefas	replica o IRB pelo número de tarefas	memória especial para salvar os IRB (32 vezes maior que um IRB)	criação de um microkernel para servir de interface entre o RTOS e a CPU	RTOS apartado da CPU que se comunica através de pedidos de interrupção	Escalonador apartado da CPU que se comunica via barramento de periféricos
Comunicação com a CPU	via registradores e atua diretamente no núcleo da CPU	via conjunto de instruções especiais, capturadas no pipeline	via conjunto de instruções especiais executadas pela CPU	através de conjunto de registradores compartilhados com a CPU e sinais de aviso	através dos barramentos de dados e endereços e por pedidos de interrupções	através dos barramentos de interconexão de periféricos e por pedidos de interrupções	através do barramento de periféricos
Troca de tarefas (<i>clocks</i>)	até 2	até 1	4 mais o tempo de escalonamento	-	9 mais o tempo para troca de contexto	-	-
Potência dinâmica	15 mW (25 MHz)	-	-	65 mW (50 MHz)	-	-	-
<i>Gates</i> extras (ASIC)	6.463	-	-	170.000	100.000	-	-
Compatibilidade	FPGA/ASIC	FPGA	FPGA	ASIC	FPGA/ASIC	FPGA	FPGA

Boa parte das propostas apresentadas na Tabela 1, além de outras publicadas, criam estruturas de processadores com foco em aplicações baseadas no uso de dispositivos lógicos programáveis (PLD), em geral FPGAs (14;15;27;32;39;41;50;55). Nestes casos, grandes blocos de registradores e memórias são ajustados de acordo com o modelo de tarefa e crescem proporcionalmente com o número de tarefas executadas, sem muito compromisso com o uso de recursos por parte dos desenvolvedores. Outra característica, observada em parte das propostas

(14;27;38;39;55), é a conexão do escalonador ou do *microkernel* implementados por hardware, diretamente aos barramentos do processador, fazendo com que ele se comporte como um outro periférico convencional. Nesses casos, fica impossível minimizar os impactos no tempo necessário para troca de contexto da CPU, uma vez que o processo de troca de contexto impede que a CPU continue utilizando os barramentos para execução de instruções relativas às tarefas. Observa-se ainda, em algumas propostas (15;41), alterações no conjunto de instruções original do processador, a fim de permitir o funcionamento do hardware implementado. Casos como esses, impactam tanto em alterações em um dos blocos mais sensíveis da arquitetura do processador, o decodificador de instruções, como nos softwares utilizados para programação do processador, seja em linguagem de baixo, médio ou alto nível. Diferentemente disso, a solução aqui proposta não altera o conjunto de instruções do processador utilizado como base para o projeto, utiliza apenas registradores com funções especiais para configuração de seu funcionamento e implementa apenas o hardware necessário para que o *microkernel* proposto funcione paralelamente à CPU, realizando, de forma otimizada o escalonamento e o despacho de tarefas. Portanto, visamos com isso, sua incorporação tanto em dispositivos *soft-cores* como em comerciais (*hard-cores*).

Para que haja uma perfeita coexistência entre a CPU e os seus blocos auxiliares (escalonador, gerenciador de contexto e temporização e controle), são necessárias alterações na sua arquitetura, sendo que as principais estão relacionadas à criação de uma réplica do banco de registradores internos e a alteração na forma de acesso de parte da memória RAM, que está reservada para o armazenamento de dados. Uma preocupação durante todo o projeto foi manter a completa compatibilidade da nova arquitetura com a anterior, pois isso permite que todos os códigos escritos para o processador original também sejam executados nessa nova arquitetura proposta por este projeto de pesquisa.

A sequência deste trabalho é dividida como segue. O capítulo 2 traz toda a base teórica necessária para o entendimento da arquitetura interna dos processadores, com destaque para arquitetura MIPS e o microcontrolador Plasma (utilizado como base para realização deste projeto). Além disso, mostra as técnicas utilizadas para desenvolvimento de programas para sistemas embarcados, descreve as características de sistemas operacionais de tempo real, seus benefícios e limitações. Por fim, apresenta a metodologia utilizada para o projeto de sistemas digitais baseados em hardware programável e a metodologia utilizada para o projeto do leiaute de um circuito integrado digital dedicado. No capítulo 3, é descrita detalhadamente a proposta

deste projeto, seu desenvolvimento, a metodologia e o arcabouço criado para programação do microcontrolador IHM-Plasma. No capítulo 4, são apresentadas as características técnicas, tanto do dispositivo implementado em FPGA, como a do ASIC. Além disso, são realizados estudos comparativos experimentais e teóricos que comprovam a eficácia do *microkernel* por hardware incorporado a uma nova arquitetura de microcontroladores. O quinto e último capítulo traz as considerações finais do projeto, apresenta um resumo dos principais resultados obtidos e faz considerações sobre trabalhos futuros. No Apêndice A é apresentado o procedimento utilizado para geração dos arquivos hexadecimais utilizados como programas para o microcontrolador proposto. No Apêndice B foi disponibilizada uma cópia do código fonte gravado na memória ROM do ASIC projetado. E no Apêndice C são citados os trabalhos publicados e em processo de aceitação, frutos deste projeto de pesquisa.

2 CONCEITOS FUNDAMENTAIS

Este capítulo é dividido em seis seções principais que abrangem os conceitos básicos para o desenvolvimento deste projeto de pesquisa. A primeira seção traz conceitos sobre arquiteturas de computadores e detalhes construtivos dos processadores. A segunda seção trata da arquitetura de processadores MIPS e do microcontrolador Plasma, que foi utilizado como a plataforma de hardware para o desenvolvimento e validação deste projeto de pesquisa. A terceira seção disserta sobre os sistemas embarcados (definição, técnicas de projeto, etc.). A quarta seção define os princípios funcionais do *microkernel* de um RTOS. A quinta seção descreve sobre o desenvolvimento de sistemas digitais e embarcados utilizando-se dispositivos lógicos programáveis. E, a última parte deste capítulo apresenta o fluxo de projeto de um ASIC, que foi utilizado para geração automática do leiaute do microcontrolador IHM-Plasma.

2.1 ARQUITETURAS DOS PROCESSADORES E COMPUTADORES

O primeiro passo para se entender a arquitetura de um processador ou computador é aprender sua “linguagem”. As palavras na linguagem do processador/computador são chamadas de instruções. O conjunto de instruções funciona como a interface entre o seu hardware e os programas a serem executados. Normalmente, o projeto de um processador ou um computador é iniciado pela definição de seu conjunto de instruções (21).

O arquiteto de processadores e computadores deve levar em conta pelo menos sete aspectos antes de definir um conjunto de instrução (22), são eles:

- Classe da arquitetura do conjunto de instruções (ISA): define como os operadores das instruções são manipulados. As duas formas mais comuns são *register-memory*, no qual os operadores podem estar na memória (exemplo 80x86), e *load-store*, no qual a memória é acessada apenas através das instruções *load* e *store* [exemplos *Advanced RISC Machine* (ARM) e MIPS] (22);
- Endereçamento da memória: é a forma pela qual o microprocessador endereça/acessa uma ou mais posições de memória para realizar uma operação de (leitura/escrita). Geralmente, o endereçamento dos operandos de uma instrução de um processador, que estão alocados na memória, é feito byte a byte, mesmo em arquiteturas cujos barramentos são maiores que 8 bits. Nas arquiteturas ARM e MIPS, o endereçamento dessas posições de memória deve ser feito de forma

alinhada, dessa forma, considerando um barramento de endereços de 32 bits, o endereço deve ser sempre múltiplo de 4 bytes e o início do endereçamento de memória deve ser mapeada a partir do endereço 0 (22);

- Modo de endereçamento: define a forma de endereçar um objeto na memória. Esse objeto pode ser um registrador, uma constante, ou uma posição de memória (22);
- Tipos, tamanhos e forma de acesso dos operadores das instruções de um processador: define o tipo (caractere ASCII, valor inteiro, ponto flutuante, etc.), o tamanho dos operadores em bytes, e a forma pela qual esses operadores são acessados na memória (leitura/escrita) (22);
- Operações/comandos: são definidas pelas operações aritméticas, lógicas, de transferência de dados, etc. (22);
- Operações/comandos do controle de fluxo de processamento: são definidas pelas operações de desvios condicionais, incondicionais, chamadas de sub-rotinas, e retornos das sub-rotinas, etc. (22);
- Codificação do ISA: basicamente o arquiteto tem duas opções quanto ao tamanho das instruções (medido em bits). Ele pode definir se todas as instruções possuirão o mesmo tamanho ou não. No caso das arquiteturas ARM e MIPS originariamente apresentavam apenas instruções de 32 bits, porém atualmente existem versões dessas arquiteturas com instruções de 64 bits e também de 16 bits. Como exemplos, na arquitetura ARM o ISA de 16 bits é chamado de Thumb-2, já na arquitetura MIPS recebe o nome de MIPS16 (22).

O projetista do programa (software) de um sistema microprocessado deve conhecer detalhadamente o funcionamento de cada uma das instruções do processador e onde os operandos dessas instruções são alocados (memórias, registradores, etc.) (21). Além disso, o projetista de software deve levar em conta mais dois aspectos importantes (21;22), são eles:

- Os relacionados ao hardware projetado (especificações do computador: a frequência de *clock*, os tipos de memória, periféricos, etc.) (22);
- Os relacionados a forma pela qual estão organizadas as memória e registradores, unidade lógica e aritmética, e de todos os blocos construtivos do processador (22).

Cabe aqui ressaltar que os computadores que possuem o mesmo conjunto de instruções podem apresentar desempenhos diferentes, pois depende da forma pela qual suas arquiteturas foram projetadas, considerando que eles utilizam a mesma tecnologia de fabricação de CI

CMOS. Para ilustrar, sabe-se que os processadores da Intel e da AMD, que apresentam o mesmo conjunto de instruções do x86, possuem desempenhos diferentes (22).

2.1.1 Classificação das arquiteturas de computadores

De forma sucinta, classifica-se a arquitetura de um computador em relação ao seu conjunto de instruções e seu hardware. Dois tipos básicos para cada uma delas são amplamente aceitos no meio acadêmico e na indústria de semicondutores e computadores (21;29;30;48).

Do ponto de vista da arquitetura de hardware de um computador, a classificação é feita em função da forma pela qual o processador (CPU) acessa as memórias de programa e de dados. A chamada arquitetura de “Von Neumann” é caracterizada por possuir um único barramento de endereços, dados, e sinais de controle para acessar as memórias de programas e de dados, enquanto que a intitulada de “Harvard” se caracteriza por possuir distintos barramentos de endereços, dados, e sinais de controle para acessar as memórias de programas e de dados (40). Os processadores atuais predominantemente possuem arquiteturas do tipo Harvard ou são baseados nela, pois trabalham com diferentes barramentos de endereços, dados, e sinais de controle para realizar a comunicação entre a CPU, as memórias de programa e de dados e os seus respectivos periféricos.

Do ponto de vista da arquitetura do conjunto de instruções (ISA) de um processador, tem-se duas classificações: a intitulada de “Computador de Conjunto de Instruções Complexo” (*Complex Instruction Set Computer*, CISC) e a de nome “Computador de Conjunto de Instruções Reduzido” (*Reduced Instruction Set Computer*, RISC). É sabido que um processador RISC é mais eficiente em termos de desempenho que a de um CISC, porém mesmo após pouco mais de três décadas da criação da primeira máquina comercial RISC (MIPS – 1985) e o primeiro desktop RISC (SPARC da Sun – 1987), não ocorreu uma total migração do uso comercial das CPUs para o tipo RISC. Isso se justifica, pelos bilhões de dólares investidos no desenvolvimento de softwares compatíveis ao padrão da Intel x86, que é baseado na ISA CISC. A partir da versão 486, as CPUs da Intel adotaram uma versão híbrida de ISA, com um núcleo RISC para a execução de instruções mais simples e mais frequentes e um núcleo CISC para a execução de instruções complexas que levam mais de um ciclo de *clock* para serem executadas (21;29;30;48;49).

Os processadores CISC foram os primeiros a ganhar o mercado e são caracterizados por possuírem um extenso conjunto de instruções, onde a simetria (instruções com mesmos

comprimentos e tempos de execução iguais) entre essas instruções não é obrigatória. As instruções mais complexas como multiplicações e divisões geralmente são realizadas em tempos maiores que aquelas que são mais simples (movimentação de dados, etc.). Esse é o caso da arquitetura de microprocessadores x86 e microcontroladores 8051, que apresentam instruções com variação de tamanho de 1 a 17 bytes e 1 a 3 bytes, respectivamente (18;40;47).

Os processadores RISC foram desenvolvidos com o objetivo de quebrar o paradigma de que instruções complexas resultariam em maior desempenho. Um conjunto enxuto de instruções, todas com mesmo comprimento e tempos de execução semelhantes caracteriza essa arquitetura de conjunto de instrução (30). Dessa forma, as instruções complexas como multiplicações de um processador CISC devem ser implementadas por software num RISC, através de um conjunto de operações de adição sucessivas. Outra alternativa é a construção de unidades aritméticas ou lógicas por hardware especializadas, de forma que os resultados sejam alcançados no mesmo tempo que aqueles gerados pelas instruções simples. As arquiteturas dos microprocessadores ARM, MIPS e dos microcontroladores PIC são exemplos desse tipo de abordagem (11;21).

Num primeiro momento pode parecer que a arquitetura CISC ofereça vantagens sobre a RISC em termos de desempenho. É verdade que programas para realizar uma determinada função implementados com processadores CISC apresentam menos instruções que aqueles implementados com CPU RISC, porém, a arquitetura de instruções RISC combinada com a arquitetura de hardware Harvard (discutida a seguir) favorece a utilização de *pipelines* (técnica de hardware que permite que a CPU realize a busca de uma ou mais instruções na memória de programa além da próxima a ser executada) de forma mais efetiva que a CISC. Assim, mais de uma instrução pode ser rodada ao mesmo tempo, permitindo que os processadores RISC possam concluir a execução de uma instrução utilizando uma menor quantidade de ciclos de *clock*. Além disso, de forma geral, os processadores CISC consomem múltiplos ciclos de *clock* para executar uma instrução, deixando a CPU ociosa por vários ciclos de *clock* até que ocorra a finalização da execução da instrução atual, para que a próxima instrução possa ser buscada na memória e executada (21;30).

As arquiteturas clássicas de hardware e do conjunto de instruções de um processador descritas anteriormente apenas servem de base para o entendimento do seu funcionamento. Muitos outros modelos de arquiteturas de hardware e do conjunto de instruções de CPUs foram desenvolvidos e comercializados (40). Para ilustrar, o ARM, o MIPS e o x86 são modelos de diferentes arquiteturas comerciais amplamente utilizados nas mais variadas aplicações da engenharia (21;29;30;48).

A Tabela 2 traz um resumo comparativo entre as arquiteturas do conjunto de instruções RISC e CISC.

Tabela 2 – Tabela comparativa entre as características das arquiteturas RISC e CISC de processadores.

#	RISC (<i>Reduced Instruction Set Computer</i>)	CISC (<i>Complex Instruction Set Computer</i>)
1	Instruções simples de apenas 1 ciclo	Instruções complexas com vários ciclos
2	Referências a memória só com instruções <i>load</i> e <i>store</i>	Qualquer instrução pode referenciar a memória
3	Uso intensivo de <i>pipeline</i>	Pouco uso de <i>pipeline</i>
4	Instruções executadas por <i>hardware</i>	Instruções interpretadas por microprograma
5	Instruções de formato fixo	Instruções com formato variável
6	Poucas instruções com poucos modos de endereçamento	Muitas instruções com muitos modos de endereçamento
7	A complexidade está no compilador	A complexidade está no microprograma por <i>hardware</i>
8	Vários conjuntos de registradores	Um único conjunto de registradores

Fonte: Autor “adaptado de” Laplante (30), 2004, página 37.

2.1.2 Pipeline

A técnica de projetos mais usada, desde meados da década de 1980, para obter ganhos de velocidade na execução de instruções por um processador é chamada de *pipeline*. Ela consiste na sobreposição temporal da execução das instruções. Para que isso seja possível, a execução é dividida em estágios, assim como uma linha de produção (21;22;29;30;40).

Essa técnica permite a redução da quantidade média de pulsos de *clock* para execução de um conjunto de instruções. A Figura 1 apresenta a comparação temporal da execução de algumas instruções *sem* e *com* a técnica de *pipeline*, neste caso, a execução foi dividida em três estágios, Busca (*Fetch*), Decodifica (*Decode*) e Executa (*Execute*).

Figura 1 – Execução de instruções sem *pipeline* (a) e com *pipeline* (b).

t (ns)	Busca	Decodifica	Executa
4	Instrução 1	-	-
8	-	Instrução 1	-
12	-	-	Instrução 1
16	Instrução 2	-	-
20	-	Instrução 2	-
24	-	-	Instrução 2

(a)

t (ns)	Busca	Decodifica	Executa
4	Instrução 1	-	-
8	Instrução 2	Instrução 1	-
12	Instrução 3	Instrução 2	Instrução 1
16	Instrução 4	Instrução 3	Instrução 2
20	Instrução 5	Instrução 4	Instrução 3
24	Instrução 6	Instrução 5	Instrução 4

(b)

Fonte: Autor.

No exemplo da Figura 1 foi considerado que todos estágios de execução consomem o mesmo tempo de processamento (4 ns por estágio). Quando essa simetria não é verdadeira, a aplicação de *pipeline* deve assumir o tempo do estágio mais demorado como o tempo de cada estágio. Desta forma, o tempo de processamento teórico máximo usando o *pipeline* pode ser expresso pela Equação (1) (22;40).

$$\text{tempo entre instruções com pipeline} = \frac{\text{tempo por instrução em uma máquina sem pipeline}}{\text{número de estágios do pipeline}} \quad (1)$$

Se aplicada ao exemplo apresentado na Figura 1, para transformar uma máquina sem *pipeline* (a) em uma máquina com *pipeline* (b), o tempo entre instruções seria de 12 ns dividido por 3, resultando em 4 ns, ou seja, um terço do tempo utilizado por uma máquina sem *pipeline*. Assim, o ganho máximo teórico de desempenho de uma máquina com *pipeline* comparada a uma sem esse recurso, que executa o mesmo conjunto de instruções, é igual ao número de estágios do *pipeline*, também chamado de profundidade do *pipeline* (40).

Os microprocessadores modernos em geral executam uma ou mais instruções por ciclo de *clock* (IPC), em outras palavras, necessitam menos de um ciclo de *clock* por instrução (CPI),

e tendem a utilizar a arquitetura do conjunto de instruções RISC que, devido à simetria das instruções (mesmo tamanho ocupado na memória), favorece o uso do *pipeline* (22).

No caso dos microcontroladores de pequeno porte, geralmente de 8 e 16 bits, o número de ciclos de *clock* por instrução executada é maior do que 1, mesmo quando usam a técnica de *pipeline*. A Tabela 3 traz alguns exemplos de processadores de 8, 32 e 64 bits, que apresentam diferentes números de estágios de *pipeline* e diferentes CPI (3;11;18;22;47;52).

Tabela 3 – Tabela comparativa entre o número de estágios de *pipeline* e ciclos de *clock* por instrução de processadores com 8, 16, 32 e 64 bits.

Processador	Arquitetura	Barramento (bits)	Pipeline (estágios)	Ciclos de <i>clocks</i> por instrução (mínimo)
Intel Core I3,5,7	RISC e CISC	64	14	<1
MIPS	RISC	32	5	1
ARM Cortex-M4	<i>Advanced</i> RISC	32	3	1
PIC16	RISC	8	2	4
8051	CISC	8	não	12

Fonte: Autor.

2.1.3 Memórias multiportas

Os processadores RISC normalmente dispõem de conjuntos de registradores internos à CPU. Esses registradores são utilizados para armazenamentos dos dados temporários provenientes da memória de dados ou constantes operadas pela ULA (Unidade Lógica e Aritmética) (21).

Em instruções que utilizam dois ou mais operadores, é importante que diferentes dados sejam acessados simultaneamente. Para que isso seja possível, a arquitetura do conjunto de registradores convencional é modificada para que os conteúdos de diferentes endereços possam ser lidos e/ou gravados ao mesmo tempo. Esse tipo de memória é conhecido como mutiportas (21). Cada porta é responsável por um acesso de leitura e/ou escrita de um endereço da memória. A Figura 2 exemplifica uma memória genérica de três portas, com duas portas para leitura e uma para gravação de dados (21).

Figura 2 – Memória com três portas.

Fonte: Autor.

Observa-se na Figura 2 que, com exceção da entrada de *clock* (*Clock*) e o sinal de habilitação de escrita (*WE3*), as entradas e as saídas de sinais são barramentos compatíveis com a capacidade de armazenamento da memória. Os barramentos *A1* e *A2* são responsáveis pelos endereçamentos dos dados a serem lidos nos barramentos de saída de dados (*RD1* e *RD2*). Já o barramento *A3* é responsável pelo endereçamento dos dados a serem escritos na memória, dados esses aplicados ao barramento *WD3*.

2.2 ARQUITETURA MIPS

A arquitetura MIPS começou a ser desenvolvida em 1981 na Universidade de Stanford pelo professor de arquitetura de computadores John L. Hennessy (22;40). A sigla MIPS é a abreviação de Microprocessador sem Intertravamento dos Estágios de *Pipeline*, do inglês *Microprocessor without Interlocked Pipeline Stages*. Justamente em seu nome está o maior diferencial das primeiras versões da arquitetura. A arquitetura MIPS é baseada em uma máquina RISC com 5 estágios de *pipeline* e garante que todas as instruções sejam executadas em apenas um ciclo de *clock* (21;22;40). Para que isso seja possível, nenhum dos estágios do *pipeline* pode demandar mais do que um pulso de *clock* para concluir sua tarefa. Quando isso não é possível, o estágio que demanda mais tempo é o responsável por enviar um sinal de travamento do *pipeline* para que o controlador da CPU possa bloquear tanto a entrada de novas instruções para serem processadas como também, os demais estágios do *pipeline*. Isso é realizado até que a tarefa seja concluída. Essa atividade é chamada de intertravamento dos estágios do *pipeline* (21;22;40).

A Figura 3 ilustra o esquema simplificado da arquitetura interna de um núcleo MIPS (21;22;40).

Figura 3 – Arquitetura MIPS simplificada.

Fonte: Autor “adaptado de” [https://commons.wikimedia.org/wiki/File:MIPS_Architecture_\(Pipelined\).svg](https://commons.wikimedia.org/wiki/File:MIPS_Architecture_(Pipelined).svg), 2018.

A escolha do modelo de arquitetura MIPS, como base para o desenvolvimento do *microkernel* por esse projeto de pesquisa se justifica pela vasta literatura disponível sobre ela. Outro fator importante é a maturidade do núcleo, já consagrado com um histórico de centenas de aplicações e mais de três décadas de utilização e aperfeiçoamentos. Cabe aqui ressaltar que o foco deste trabalho é o desenvolvimento de um *microkernel* por hardware e não de um núcleo propriamente dito de um microcontrolador. Porém, espera-se que o núcleo apresente desempenho que justifique sua utilização em sistemas de tempo real.

2.2.1 Conjunto de instruções do MIPS

Os processadores RISC, tal como o MIPS, têm conjuntos de instruções com quantidade de bits fixa, neste caso 32 (21;22;40). Isso quer dizer que, todas as instruções compreendidas pelo processador são codificadas em um número fixo de bits (21;22;40).

Um dos pilares no desenvolvimento de uma arquitetura, defendido por Patterson e Hennessy (22;40), é que pequeno é sinônimo de rápido. Isso se aplica ao acesso à memória. Operadores armazenados num conjunto reduzido de registradores dentro da CPU são acessados muito mais rapidamente do que operadores armazenados em memória de dados de grande capacidade. Com base nisso, a arquitetura MIPS possui um pequeno conjunto de 32 registradores internos à CPU, os quais são divididos em subconjuntos, com propósitos bem definidos, vide Tabela 4, e são utilizados como operadores da ULA (5;21;40,).

Além dos 32 registradores apresentados na Tabela 4, dois outros registradores, que não fazem parte do conjunto padrão da arquitetura merecem menção, são definidos como **hi** e **lo** (21;40). Esses registradores precisam de instruções especiais para serem acessados e são utilizados para o armazenamento do resultado de operações de multiplicação e divisão (21;40). Na multiplicação, o **hi** armazena os 32 bits mais significativos e o **lo** armazena os 32 bits menos significativos. Na divisão, o **lo** armazena o quociente e o **hi** armazena o resto da operação de divisão entre dois valores.

Qualquer valor que esteja na memória de dados e que deve ser processado, precisa ser movido para um dos registradores antes do processamento (21;40). Após processamento desse valor, ele pode ser movido novamente para memória de dados. Essa é característica básica de uma arquitetura *load-store* (carrega e armazena) (21;40).

Tabela 4 – Conjuntos de registradores internos da arquitetura dos processadores MIPS.

Nome	Número	Uso
\$zero	0	fonte para valor constante 0
\$at	1	temporário usado pelo <i>assembler</i> , reservado para implementar “macro instruções”
\$v0–\$v1	2 a 3	valores para os retornos de funções e avaliação de expressão
\$a0–\$a3	4 a 7	argumentos de funções
\$t0–\$t7	8 a 15	temporários, podem ser usados livremente
\$s0–\$s7	16 a 23	temporários salvos, podem ser usados livremente, mas as funções precisam salvar e restaurar o valor daqueles que forem utilizados
\$t8–\$t9	24 a 25	temporários, podem ser usados livremente
\$k0–\$k1	26 a 27	reservados para o <i>kernel</i> de sistemas operacionais
\$gp	28	ponteiro global, usado para acessar variáveis globais
\$sp	29	ponteiro de pilha (<i>stack</i>)
\$fp	30	ponteiro do quadro, usado em funções para estabelecer uma referência constante para deslocamento até as variáveis locais e parâmetros da função
\$ra	31	endereço de retorno, usado para armazenar o endereço de retorno toda vez que a CPU executa a chamada de uma função

Fonte: Autor.

O conjunto de instruções dos processadores MIPS é dividido em três subconjuntos com formatos construtivos diferentes (21;22;40):

- *R-type* ou tipo R: instruções que operam sobre três registradores;
- *I-type* ou tipo I: instruções que operam sobre dois registradores e um valor constante de 16 bits, chamado de *immediate* ou imediato;
- *J-type* ou tipo J: instruções de salto (*jump*) levando-se em conta um desvio de até 26 bits.

Os processadores que dispõe de uma unidade para cálculos com ponto flutuante possuem um quarto subconjunto de instruções chamado de *F-type* ou tipo F que são usadas nos cálculos com valores reais (21;22;40).

As instruções *R-type* também identificadas como *register-type* operam sobre dois registradores de fonte de dados e um registrador para destino do resultado (21;22;40). Os 32 bits que formam cada instrução são divididos em 6 campos: **op**, **rs**, **rt**, **rd**, **shamt** e **funct** (vide Quadro 1). Os campos **op** e **funct** definem a operação a ser realizada. **op** é abreviatura de *opcode* que representa o código da operação, ele vale zero em todas as instruções *R-type*. **funct** é abreviatura de *function* ou função em português e armazena o código da operação. Os campos **rs** e **rt** armazenam os números dos registradores utilizados como fonte de dados e **rd** armazena o número do registrador utilizado como destino do resultado. O campo **shamt** é utilizado apenas em operações de deslocamento (*shift*) e define a quantidade de deslocamentos. Nas demais operações o seu valor é zero.

Quadro 1 – Formato das instruções do tipo R da arquitetura MIPS.

31	27	26	22	21	16	15	11	10	6	5	0
op	rs	rt	rd	shamt	funct						
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits						

Fonte: Autor “adaptado de” Harris (21), 2012, página 305.

As instruções *I-type*, também identificadas como *immediate-type*, operam com dois registradores de dados e um valor constante (*immediate*). Os 32 bits que formam cada instrução são divididos em 4 campos: **op**, **rs**, **rt** e **imm** (vide Quadro 2). O campo **op** define a operação a ser realizada levando-se em conta os três operandos. Os campos **rs** e **imm** são sempre utilizados como fonte de dados e **rt** é utilizado como fonte de dados para algumas instruções e destino para outras. **imm** também é utilizado como uma constante em operações matemáticas e lógicas, e é utilizado como valor de desvio para acesso à memória em operações de carga e armazenamento (21;22;40).

Quadro 2 – Formato das instruções do tipo I da arquitetura MIPS.

31	26	25	21	20	16	15	0
op	rs	rt	imm				
6 bits	5 bits	5 bits	16 bits				

Fonte: Autor “adaptado de” Harris (21), 2012, página 307.

As instruções *J-type* também identificadas como *jump-type* são utilizadas apenas para desvio na memória de programa, ou seja, definir o endereço da próxima instrução a ser executada. Os 32 bits que formam cada instrução são divididos em 2 campos: **op** e **addr** (vide

Quadro 3). O campo **op** define a operação a ser realizada. O campo **addr** armazena o endereço de desvio (21;22;40)

Quadro 3 – Formato das instruções do tipo J da arquitetura MIPS.

31	26	25	0
op		addr	
6 bits		26 bits	

Fonte: Autor “adaptado de” Harris (21), 2012, página 308.

2.2.2 Blocos básicos da microarquitetura MIPS

Um mesmo conjunto de instruções pode ser interpretado e executado usando diferentes hardwares, chamados de microarquiteturas (21).

O modelo de microarquitetura usado para a interpretação e execução do conjunto de instruções dos processadores MIPS, que foi descrito na seção 2.2.1 deste capítulo, foi ilustrado na Figura 3 (21;40).

A interpretação e execução de cada uma das instruções é dividida em um *pipeline* de 5 estágios. Para isso, a microarquitetura desse processador foi dividida em 5 partes ou estágios, que foram separados por registradores que são acionados pelo *clock* do sistema (21;40). Essas partes são identificadas como segue (21;22;40):

1. IF (*Instruction Fetch* ou Busca Instrução): busca a próxima instrução a ser executada na memória de programa (ROM). Aqui ficam a memória ROM e o contador de programa PC (*program counter*), que é incrementado de 4 em 4 bytes (32 bits, tamanho de uma instrução) ou tem seu valor definido após a execução de uma instrução de desvio;
2. ID (*Instruction Decode* ou Decodifica Instrução): lê dois registradores e define o valor de *offset* para acesso a memória de dados, quando uma operação de carga (*load* – **lw**) é executada. Nesse tipo de instrução um registrador é responsável por armazenar o endereço base, o qual é somado a um valor inteiro sinalizado de 16 bits como *offset*;
3. EX (*Execute or Address Calculation* ou Executa ou Calcula Endereço): utiliza a ULA para o cálculo do endereço da instrução de carga ou realiza operação entre dois registradores;

4. MEM (*Memory Access* ou Acesso à Memória): faz acesso à memória de dados tanto para leitura como para escrita, além disso, define o próximo endereço usado para no ciclo de busca armazenado no contador de programa (incremento de 4 bytes ou desvio);
5. WB (*Write-Back* ou Escreve de Volta): define se o valor direcionado para um registrador parte da saída da ULA ou da memória de dados.

Cada uma das partes da microarquitetura é separada de sua anterior e posterior através de registradores controlados pelo sinal de *clock* do sistema (21;40). Esses registradores controlam o fluxo da instrução em execução ao longo do circuito e são identificados pelas siglas dos estágios que o precedem e o sucedem, como por exemplo IF/ID (Busca Instrução/Decodifica Instrução) (21;40).

Na Figura 3 não foi representado o circuito de controle dos sinais da CPU, esse circuito é responsável pela seleção dos multiplexadores, operações da ULA e análise de exceções na execução das instruções (21;40).

2.2.3 Microcontrolador Plasma

Este projeto utilizou a arquitetura da CPU do microcontrolador Plasma (45) como referência para o desenvolvimento do novo microcontrolador com um *microkernel* por hardware. O Plasma é um microcontrolador RISC de 32 bits, sintetizável em FPGA. Sua versão atual tem como periféricos disponíveis: interfaces UART e Ethernet, controladores para memória SRAM (*static* RAM, memória baseada em *flip-flops* que é parte da estrutura programável dos FPGAs) ou DDR SDRAM (DDR significa *double-data-rate*, ou seja, taxa dupla de transferência, isso quer dizer que esse tipo de memória é capaz de transferir dois dados a cada pulso de *clock*; já SDRAM significa *synchronous dynamic* RAM, memória de acesso dinâmico sincronizada com o barramento do sistema). A CPU Plasma executa todas as instruções do modo de usuário MIPS I, exceto as operações de carga e armazenamento não alinhadas. Sua implementação foi toda feita em linguagem de descrição de hardware VHDL. Possui *pipeline* de dois ou três estágios e é executada com *clock* de 25 MHz em FPGAs Xilinx e Intel (antiga Altera).

A Figura 4 representa o diagrama de blocos simplificado desse microcontrolador. Esse diagrama foi elaborado por seu idealizador, Steve Rhoads, e destaca exclusivamente os blocos (arquivos VHDL) pertencentes à sua CPU.

Figura 4 – Diagrama de blocos da CPU do microcontrolador Plasma.

Fonte: <https://opencores.org/project.plasma.overview>, 01/07/2018.

A seguir são descritas as funções de cada um dos blocos ilustrados na Figura 4:

- PC_next: registrador que armazena o endereço da próxima instrução a ser executada;
- Reg_bank: pequena memória multiportas com capacidade de armazenar 32 *words* com 32 bits cada. É utilizada como banco interno de registradores (IRB) do Plasma (vide Tabela 4 na seção 2.2.1);
- Mem_ctrl: controla o acesso à memória, identifica se há pedido de acesso a memória de dados ou programa e gera os sinais necessários para que os outros blocos procedam com a operação;
- Control: controla o funcionamento da CPU, é o responsável pela decodificação dos *opcodes* e geração dos sinais de controle para toda CPU;
- Bus_mux: funcionada como um roteador de sinais, ele multiplexa os sinais de diferentes fontes para seus destinos;

- Mult: unidade de multiplicação de divisão de 32 bits, utilizada para execução de instruções desse tipo;
- ALU: unidade lógica e aritmética, utilizada para execução de instruções aritméticas de soma e subtração, e lógicas do tipo ‘E’, ‘OU’, ‘OU-Exclusivo’ e ‘NOU’;
- Shifter: unidade de deslocamento de bits, utilizada para execução de instruções do deslocamento de bits à esquerda e à direita.

Podemos destacar algumas razões que justificam a escolha do Plasma como microcontrolador base para o desenvolvimento deste projeto, são elas:

- A qualidade da documentação disponibilizada pelo autor;
- A forma de divisão dos arquivos que compõe o projeto. Cada arquivo representa um dos blocos da CPU. Essa abordagem facilita muito sua análise de funcionamento;
- A maturidade desta CPU, sua primeira versão foi publicada no ano 2001 e vem sofrendo atualizações desde então;
- A certificação de qualidade do site OpenCores, considerado uma referência para desenvolvedores de sistemas baseados em hardware programável;
- A integração com o compilador C GCC. Isso possibilita o desenvolvimento de programas em linguagem C, fundamental para testes do núcleo e do *microkernel*;
- A acessibilidade ao seu desenvolvedor, que sempre se mostrou pronto para esclarecer dúvidas quando a arquitetura do núcleo e as ferramentas de software para compilação e conversão de códigos.

Todas essas razões apresentadas não nos isentaram de validar cada um dos blocos do núcleo separadamente. Toda validação de código foi feita através do software simulador de hardware ModelSim, bem como através do uso do kit de desenvolvimento DE1-SoC fabricado pela Terasic (51), que utiliza tecnologia FPGA da família Cyclone V, atualmente fabricado pela Intel (24), ele pode ser visto na Figura 14 na seção 2.5.3.

O aplicativo utilizado como ambiente de desenvolvimento foi o Quartus II *Web Edition* versão 15.0, que era fornecido pela empresa Altera na data de início deste projeto (25).

2.3 SISTEMAS EMBARCADOS

Nesta seção são apresentadas algumas das estruturas de projeto de sistemas embarcados mais comuns ao engenheiro de aplicações. Começamos pela definição de sistema embarcado e pela classificação desses sistemas de acordo com suas capacidades de atendimento de prazos.

2.3.1 Definição de sistema embarcado

Um sistema embarcado é um dispositivo ou equipamento que possui um processador para execução de uma função específica (29;54).

Esse tipo de sistema interage com o meio ou o usuário, que normalmente não faz qualquer tipo de atualização de hardware ou software e também não é capaz de alterar o que o sistema faz. O software ou *firmware*, que o processador executa, é geralmente gravado em memória não-volátil. (53).

2.3.2 Tipos de sistemas embarcados

Sistemas embarcados podem ser classificados de acordo com a criticidade ou a segurança no cumprimento dos prazos (*deadlines*) para execução de suas tarefas ou rotinas disparadas por eventos (29;30).

Um sistema de tempo não-real (*non-RTS*) é o sistema que opera em ambientes onde não há prazos envolvidos (29;30). É o sistema onde a reação programada para um estímulo com certeza vai acontecer em algum momento futuro (29;30). Para ilustrar podemos citar um forno elétrico programável, onde o ajuste de uma determinada temperatura não está atrelado a um prazo específico para ser atingida, mas isso ocorrerá em algum momento.

Um sistema é classificado como de *tempo real* quando o mesmo deve responder a eventos críticos e que apresentam restrições temporais associados a eles (30;54). Isso quer dizer que, *tempo real* não tem vínculo com rapidez de processamento, e sim está vinculado ao atendimento dos prazos previamente definidos pelo programador, onde o não cumprimento desses prazos pode impactar em falhas e restrições de funcionamento do sistema (30;54). Eles são divididos em três categorias (6;29;30;53), são elas:

1. Sistemas de tempo real não-críticos ou moderados (*soft RTS*): são aqueles onde o não cumprimento de um prazo pode ter efeitos indesejáveis, mas não são catastróficos, apenas gera uma degradação de seu desempenho (6;29;30;53). Nesse tipo de sistema, a reação programada para um estímulo é quase sempre concluída no prazo de um tempo finito conhecido (6;29;30;53). Como por exemplo, o sistema de detecção aos toques em uma tela *touch*. O atraso na resposta ao estímulo, implica apenas no desconforto do usuário;

2. Sistemas de tempo real críticos (*hard* RTS): são aqueles onde o não cumprimento de um prazo pode ter efeitos catastróficos (6;29;30;53). Esses sistemas requerem uma definição muito mais rigorosa. Nesse tipo de sistema, a reação programada para um estímulo é garantida para ser concluída dentro de um tempo finito conhecido (6;29;30;53). Como exemplo, o sistema de airbag (bolsa de ar) dos veículos atualmente produzidos. Em caso de acidente, seu acionamento deve ocorrer em milésimos de segundo. Um atraso no atendimento a esse evento, pode implicar na perda de pelo menos uma vida;
3. Alguns autores consideram ainda um tipo de sistema intermediário entre *hard* e *soft* RTS, chamado de *firm* RTS ou sistema de tempo real rígido: nesse tipo de sistema algumas perdas de prazos são toleradas, mas ultrapassar um certo limite pode causar uma falha catastrófica no sistema (6;29;30)

2.3.3 Técnicas de projeto de sistemas embarcados

Do ponto de vista do programador de sistemas embarcados, o código pode ser modelado usando diferentes técnicas. Em sistemas embarcados considerados *low-end* (pequenos e de baixa complexidade) é possível criar códigos onde o uso de interrupções de programa é descartado (29;30). Dentro deste contexto, destacam-se três técnicas:

1. *Polled Loop* (varredura por laço) é a técnica mais simples de organizar um código. Para aplicá-la, basta criar um laço infinito e dentro dele, deve-se colocar todas as atividades ou tarefas a serem executadas. A Figura 5 (a) exemplifica a aplicação desse tipo de técnica;
2. *Cyclic Code Structure* (estrutura de código cíclico) é a técnica aplicada quando é necessário que algumas tarefas sejam executadas um número de vezes maior do que outras. Isso é feito através da repetição da chamada de uma função dentro do laço criado. A Figura 5 (b) exemplifica a aplicação desse tipo de técnica;
3. *State-Driven Code* (código com estados dirigidos), se baseia na construção de máquinas de estados finitos (FSM), no qual cada tarefa é dividida em subpartes (estados), que são executadas intercaladas com outras subpartes de outras tarefas. A Figura 5 (c) exemplifica a aplicação desse tipo de técnica.

Figura 5 – Técnicas de projeto de sistemas embarcados do tipo *low-end*. Cada hexágono representa uma atividade ou tarefa. *Polled Loop* (a), *Cyclic Code Structure* (b) e *State-Driven Code* (c).

Fonte: Autor.

As duas primeiras técnicas descritas anteriormente (*Polled Loop* e *Cyclic Code Structure*) são também conhecidas como *super loop*, uma vez que elas são baseadas em uma repetição contínua das mesmas atividades ou tarefas, sempre na mesma ordem que foram declaradas no programa (13). Essa técnica oferece como vantagens, a facilidade e a rapidez no desenvolvimento de sua codificação, além de não consumir recursos adicionais de software para o gerenciamento da execução do código, como ocorre nos sistemas operacionais convencionais. Por outro lado, está limitada a sistemas pequenos e com poucas restrições temporais. Pode-se notar que a execução de todo o código é encadeada, se uma tarefa demora demais para ser concluída, todas as demais são atrasadas. Nesse tipo de sistema, todas as tarefas possuem a mesma prioridade e é muito complicado garantir que os tempos estabelecidos para as tarefas sejam cumpridos após qualquer alteração no código fonte (29;30).

A terceira técnica descrita (*State-Driven Code*), tem como desvantagem, que nem todos os sistemas podem ser facilmente divididos em estados e à medida que se tornam mais complexos, as tabelas utilizadas para sua descrição se tornam mais extensas e a possibilidade de cometer um erro durante a transcrição para código de programa aumenta (29;30).

A maioria das soluções em sistemas embarcados utiliza-se de interrupções de programa, disparadas por eventos, para melhorar o controle do fluxo de dados em execução. Essas soluções formam os sistemas dirigidos por interrupção (*Interrupt-driven Systems*) (29;30;53). Eles podem ser divididos em dois tipos:

1. *Interrupt-Only Systems*, aqueles escritos inteiramente e exclusivamente baseados em interrupções. Neste caso, a execução do programa é toda concentrada em na execução das rotinas de tratamento de interrupções (*interrupt service routines*,

ISRs). Assim que o programa inicia e a configuração da máquina é feita, as fontes de interrupções são configuradas e habilitadas. Na sequência, o programa normalmente roda em um laço vazio infinito, esperando que as interrupções aconteçam, consequentemente, a execução das correspondentes ISRs. A Figura 6 (a) exemplifica a aplicação desse tipo de técnica;

2. *Background/Foreground Systems*, aqueles que misturam conceitos de *polled loop* e interrupções. Neste tipo de sistemas, as tarefas com menor prioridade são executadas continuamente no laço principal (*background*, segundo plano) e as interrupções são tratadas pelas ISRs (*foreground*, primeiro plano). A Figura 6 (b) exemplifica a aplicação desse tipo de técnica.

Figura 6 – Tipos de sistemas baseados em interrupções. Casa hexágono representa uma atividade ou ISR. *Interrupt-Only Systems* (a) e *Background/Foreground Systems* (b).

Fonte: Autor.

A Figura 6, nos itens (a) e (b), ilustra um evento externo ao processador, um botão pressionado, e um evento interno ao processador, o estouro de contagem de um temporizador/contador. Em ambos os casos, é necessário o salvamento dos valores de todas as variáveis ou registradores compartilhados, antes de executar as ISRs. E após executadas, é necessário restaurar os valores das mesmas variáveis ou registradores. Isso permite que as informações que estão sendo processadas antes que uma interrupção ocorra, não se percam. Dessa forma, sendo possível a retomada do processamento anterior, assim que o tratamento da interrupção se encerre (29;30).

A desvantagem, dos tipos de sistemas apresentados, é a dificuldade em gerenciar restrições temporais, uma vez que uma interrupção pode atrasar o atendimento de outra. Além disso, a quantidade de vezes que o laço principal é executado está limitada pela frequência em que as interrupções ocorrem (29;30).

Os sistemas operacionais de tempo real foram criados justamente para superar esses problemas e os demais associados aos sistemas baseados nas técnicas acima descritas (*Polled Loop*, *Cyclic Code Structure*, *State-Driven Code*, *Interrupt-driver Systems* e *Background/Foreground Systems*) (29;30).

2.4 SISTEMA OPERACIONAL DE TEMPO REAL

Um RTOS pode ser entendido como uma parte de um sistema operacional tradicional (29;44;53). A Figura 7 representa um sistema operacional dividido em quatro camadas funcionais que, à medida que são empilhadas, recebem uma classificação própria (29).

Figura 7 – Divisão de um sistema operacional em camadas funcionais.

Fonte: Autor “adaptado de” Laplante (29), 2011, página 81.

A combinação das quatro camadas sobre o hardware forma o sistema operacional acessível ao usuário e apto a rodar os aplicativos, geralmente criados para execução em computador (29).

Do ponto de vista de um sistema embarcado, um sistema operacional que atenda a restrições temporais e ofereça suporte à multitarefa em aplicativos escritos para processadores com acesso direto ao hardware, precisa, no mínimo, de capacidade de executar: (i) o escalonamento de tarefas; (ii) a troca de contexto; (iii) a comunicação entre tarefas e; (iv) a sincronização entre tarefas. Esse conjunto de capacidades é representado pelas duas primeiras camadas de software apresentadas na Figura 7. Dessa forma, os RTOSs também costumam ser chamados de *kernels* de tempo real (30,4).

O papel de um RTOS em um RTS é justamente trabalhar para garantir que os tempos de resposta aos estímulos do ambiente sejam cumpridos. Isso caracteriza um sistema determinístico (29;30). A reação dos sistemas de tempo real aos eventos vindos do ambiente externo ocorre em tempos de resposta compatíveis com as exigências do ambiente, e mensuráveis na mesma escala de tempo (29;30).

O programador acostumado a ter o controle do *modus operandi* de sua aplicação, passa a não mais saber quando suas rotinas, aqui chamadas de tarefas (*tasks* ou *threads*), serão executadas (29;30). Ele apenas precisa definir a prioridade e, em alguns casos, o tempo de resposta desejado para cada tarefa (29;30). De forma geral, as tarefas são escritas como códigos para ações específicas ou para acesso a periféricos.

Um RTOS se comporta como uma grande camada de abstração (ocultação dos detalhes construtivos) no projeto de sistemas embarcados (30;53;54). O programador deixa de se preocupar no “como” será executado e se concentra apenas no “o que” será executado (30;53;54). Essa mudança de paradigma pode parecer assustadora para alguns, porém é fundamental no momento em que as aplicações se tornam deveras complexas e carregadas de tarefas concorrentes (29;30;53;54). Tentar prever o funcionamento da aplicação em todas as condições possíveis, se torna tarefa tortuosa e às vezes inviável (29;30;53;54). Pequenas alterações no código fonte podem arruinar os tempos de resposta desejados ou causar comportamentos catastróficos (29;30).

Para se tirar proveito das vantagens dos RTOSs é preciso pagar um preço. Eles consomem quantidade significativa de memória (RAM e ROM), quando aplicado a microcontroladores de 32 bits de baixo custo, que possuem algumas dezenas de quilo bytes de memória RAM e ROM. O FreeRTOS por exemplo, consome entre 5kB e 10kB de memória ROM para ser embarcado em um microcontrolador de ARM7 (43). Do ponto de vista de memória RAM, uma solução convencional possui apenas uma pilha (*stack*) para armazenamento de endereços e dados temporários, enquanto que num sistema com RTOS é criada uma pilha separada para cada uma das tarefas implementadas no sistema (4;5;6;29;30;44;53;54).

2.4.1 *Microkernel*

O *microkernel* é a primeira camada de software criada sobre o hardware (Figura 7) e tem como suas atribuições cuidar do escalonamento de tarefas e troca do contexto da CPU, também chamado de despacho de tarefas. Dessa forma, o *microkernel* pode ser dividido em

dois módulos de software: (i) escalonador (*scheduler*), que define a ordem de execução das tarefas e a frequência com que elas são executadas; (ii) despachador (*dispatcher*) ou gerenciador de contexto, que realiza a troca das tarefas em execução (4;6;29;30;44;54).

O escalonador pode ser do tipo **preemptivo** (antecipativo) ou **cooperativo** (4;29;30;53). No modo preemptivo, a tarefa em execução pode ser interrompida antes de sua finalização. Isso ocorre com base em interrupções temporizadas chamadas de *tick interrupts* ou apenas *tick*, onde é avaliado se existe outra tarefa num estado definido como “pronta para ser executada” (*ready*) com prioridade igual ou superior à sua. (4;29;30;53). Já no modo cooperativo, somente a tarefa em execução (*running*) é capaz de decidir quando será interrompida para que outra passe a ser executada. Ou seja, mesmo que uma tarefa de maior prioridade esteja pronta para ser rodada, ela deve esperar até o término voluntário da tarefa em execução. O modo cooperativo, por conta de não permitir interrupções, não é adequado para sistemas de tempo real (4;29;30;53).

Quando nos referimos a um processador de núcleo único, foco deste trabalho. Para que todas as tarefas sejam executadas segundo os tempos de resposta desejados (princípio do determinismo), o módulo escalonador se baseia em modelos de algoritmos, que consideram diferentes parâmetros como referências e usam diferentes estratégias para seleção das tarefas (4;6;29;30;44;54). Seguem alguns dos modelos de algoritmos amplamente implementados nos sistemas baseados em tarefas (6;29;30):

- **Round-Robin:** as tarefas do sistema são executadas sequencialmente, normalmente são modeladas como laços infinitos. A troca das tarefas é baseada em frações de tempo, chamadas de *time slices* (tempo mínimo disponível para que uma tarefa possa ser executada). Uma tarefa é executada até ser completada ou seu *time slice* encerrar. Se a tarefa não foi encerrada, ela é colocada no final da fila *round-robin* para posterior execução (preempção). A Figura 8 ilustra esse processo (6). Cada troca de tarefa em execução é precedida pela troca de contexto da CPU (6;29;30).

Figura 8 – Fila de tarefas aguardando para serem executadas.

- **Cyclic Code** (código cíclico): ele é altamente determinístico, se baseia num intercalador de tarefas periódicas, que mapeia a ordem de execução das mesmas numa tabela, antes de iniciar sua execução. A decisão da próxima tarefa a ser executada é feita periodicamente, esse tempo é chamado de *frame*. O tempo mínimo para execução das tarefas, de acordo com seus períodos e prazos para execução (*deadlines*), é chamado de hiperperíodo. O algoritmo de código cíclico não aceita preempção, ou seja, as tarefas não são interrompidas antes de completadas. Dessa forma, o tempo ajustado para o *frame* deve ser maior que o período de execução da tarefa mais demorada (6;29;30).
- **Rate-Monotonic** (taxa monotônica): este algoritmo trabalha com tarefas com prioridades fixas definidas de acordo com o período de cada tarefa. As tarefas com tempo de execução menor, têm prioridade maior do que aquelas com maior tempo de execução (6;29;30).
- **Earliest-Deadline-First** (mais próximo do prazo primeiro): este algoritmo trabalha com tarefas periódicas com prioridades dinâmicas. Isso quer dizer que as prioridades das tarefas são definidas e alteradas durante a execução do programa. Assim, o escalonador define a próxima tarefa a ser executada de acordo com seus prazos para execução (*deadlines*) pré-definidos. Aquela que estiver mais próximo do seu prazo, recebe prioridade maior, e se não estiver em execução, pode preceder a tarefa em execução (6;29;30).

Um processador de núcleo único executa uma tarefa de cada vez, porém outras aguardam sua vez para serem executadas. Cada uma das tarefas pode assumir um dos quatro estados listados a seguir (6;29;30;54):

- Rodando/Executando (*Running*): tarefa em execução pelo processador;
- Pronta (*Ready*): pronta para ser executada, mas que não está em execução, porque uma tarefa de prioridade igual ou superior já está em execução;
- Bloqueada (*Blocked*): está à espera de um evento temporal ou externo para estar pronta para ser rodada. Tarefas bloqueadas não estão disponíveis para o escalonador;
- Suspensa (*Suspended*): tarefa não está disponível para o escalonador. As tarefas só entram ou saem deste estado através de comandos específicos.

A Figura 9 representa o fluxo de estados de uma tarefa. Ao longo do tempo, as tarefas podem passar por todos os quatro estados possíveis mais de uma vez. Com exceção do estado

Running, todas as tarefas podem simultaneamente se encontrarem num mesmo estado. Isso ocorre no momento em que elas são criadas (6;29;30;54).

Figura 9 – Fluxo de estados de uma tarefa num sistema de tempo real.

Fonte: Autor.

Sempre que uma tarefa precisa ser interrompida, é de responsabilidade do gerenciador de contexto preparar a CPU para que inicie ou retome a execução da próxima tarefa (4;6;29;30;44;54). Esse processo é chamado de troca de contexto ou em inglês *context switching*. Ele é uma operação necessária e obrigatória para manutenção dos valores corretos nos registradores internos da CPU, endereço do *stack pointer* (SP) e *program counter* (PC) (4;6;29;30;44;54). A troca de contexto pode ser realizada automaticamente por hardware (circuito interno do processador) quando ocorrem interrupções de programa, e é realizada por software quando utilizado um RTOS para realização da troca da tarefa em execução nos sistemas multitarefas. Neste último caso, o RTOS reserva dinamicamente, durante a criação das tarefas, um bloco de memória RAM para o controle de cada uma das tarefas. Esse bloco é chamado de *Task Control Block* (TCB). Cada TCB é formado por uma estrutura de dados com o valor do PC, dos registradores internos da CPU, o nome da tarefa, o estado da mesma e sua prioridade de execução (4;6;29;30;44;54).

A troca de contexto é executada em duas etapas pelo despachador: (i) *backup* do contexto, realizado imediatamente após a execução da última tarefa, onde o contexto da tarefa interrompida é salvo no bloco TCB correspondente; (ii) restauração do contexto, realizado

imediatamente após a execução do algoritmo de escalonamento, onde o contexto, da próxima tarefa a ser executada, é copiado do TCB correspondente para a CPU.

A Figura 10 ilustra o fluxo de processamento realizado por uma CPU gerenciada por um *microkernel* onde a execução de duas tarefas é alternada ao longo do tempo.

Figura 10 – Fluxo de execução de duas tarefas gerenciadas por um *microkernel*.

Fonte: Autor.

2.4.2 Outras funcionalidades que compõem o *Kernel*

A segunda camada de software, que associada a primeira (*microkernel*), compõem um *kernel*. Ela é responsável para comunicação entre tarefas e sincronização de tarefas (Figura 7). Destacamos aqui as três principais técnicas utilizadas para RTOSs (4;6;29;30):

1. *Mailbox* (caixa de correio): é uma técnica para comunicação assíncrona entre tarefas. Ela utiliza uma memória compartilhada do tipo FIFO (*first-in, first-out*, em português primeiro a entrar, primeiro a sair) para criar uma caixa de correio acessível pelas tarefas. O número máximo de mensagens que podem ser mantidas em uma caixa de correio representa sua capacidade. Duas operações básicas estão disponíveis, a de enviar mensagens e a de receber. A tarefa que envia mensagens é chamada de produtora, e que recebe é chamada de consumidora. A Figura 11 ilustra o funcionamento de um *mailbox* com capacidade de seis mensagens.

Figura 11 – Esquema de um *mailbox*.

Fonte: Autor "adaptado de" Buttazzo (6), 2011, página 386.

2. Semáforo binário: é simplesmente uma *flag* compartilhada. Existem duas operações que podem ser executadas em um semáforo: (i) *Give* é a ação que *setar* uma *flag*; (ii) *Take* é a ação que verificar a *flag*. Após essa ação, se a *flag* estiver *setada*, ela será automaticamente *resetada* e a tarefa que fez a verificação executara alguma ação. Assim como na memória FIFO do *mailbox*, a *flag* é declarada como uma variável global compartilhada. Muitas vezes, semáforos binários são usados por tarefas menos prioritárias para evitar que elas sejam interrompidas durante sua execução por tarefas mais prioritárias.
3. *Mutex* (*mutual exclusion* ou em português, exclusão mútua): é um tipo especial de semáforo binário que é usado para controlar o acesso a um recurso que é compartilhado entre duas ou mais tarefas. O *mutex* pode ser conceitualmente considerado como um *token* associado ao recurso que está sendo compartilhado. Para que uma tarefa acesse legitimamente o recurso, ela deve primeiro pegar o *token* (ação chamada de *take*). Quando a tarefa portadora do *token* terminar de usar o recurso, ela deve devolver o *token* (ação chamada de *give*). Somente quando o *token* for retornado, outra tarefa poderá obter o *token* com êxito e acessar com segurança o mesmo recurso compartilhado. Um exemplo simples de seu uso, é quando duas tarefas tentam usar uma mesma porta de comunicação serial para enviar mensagens. O *mutex* impede que as mensagens sejam intercaladas devido ao escalonamento de tarefas. Assim, somente quando a primeira mensagem for completada, a segunda poderá ser enviada.

2.4.3 Benefícios e limitações dos RTOSs

Como apresentado anteriormente, o papel do RTOS assumir a gestão da dinâmica de execução do programa, controlando as entradas e as saídas das tarefas, garantindo sempre uma troca de contexto confiável, e criando meios de comunicação entre as tarefas. Podemos listar os seguintes benefícios em sua utilização(4;6;29;30;44;53;54):

- Possibilidade de priorizar tarefas para garantir que as restrições de tempo da aplicação sejam atendidas;
- Permitir que o sistema seja orientado a eventos, evitando por exemplo rotinas de *polling*, consequentemente melhorando a eficiência do sistema;

- Como o sistema é implementado por tarefas, incentiva uma melhor modularidade da solução a ser desenvolvida;
- Permite ao programador que se concentre no desenvolvimento de aplicativos, em vez da gestão dos recursos (*housekeeping*) utilizados por ele.

Apesar dessas vantagens, existe um preço a ser pago pela alta frequência na alternância entre tarefas (29;30). Quando analisamos a arquitetura interna dos processadores, formada por memórias, registradores, unidade lógica e aritmética (ULA), unidade de controle, entre outros blocos, é fácil concordar que a troca de contexto das tarefas na CPU e o tempo demandado para escolha na próxima tarefas são diferentes de zero (29;30). Da mesma forma, é consenso que, à medida com que o número de trocas de tarefas por unidade de tempo aumenta, esses tempos necessários crescem proporcionalmente (29;30). Esses tempos representam os principais gargalos limitadores da frequência de operação do escalonador, que restringe o desempenho do sistema eletrônico como um todo (29;30).

Para melhor entendimento, considere o fator de ocupação da CPU, chamado de U . Esse fator indica o percentual da capacidade de processamento utilizada pela CPU durante a execução de um programa (29;30). Para medi-lo, num sistema com uma ou mais tarefas, é preciso conhecer o tempo de execução de cada uma delas (e_i) e o período que cada uma deve ser repetida (p_i). No caso de tarefas aperiódicas, deve-se considerar o pior caso, a condição com maior número de execuções por segundo (29;30). O fator de utilização U é calculado através da somatória da razão e_i por p_i de cada uma das tarefas, como apresentado na Equação (2) (29;30).

$$U = \sum_{i=1}^n \frac{e_i}{p_i} \cdot 100\% \quad (2)$$

Liu e Layland (1973) (31) provaram que o limite de U para atender n (número inteiro) tarefas periódicas com o mesmo período é igual a $n \cdot (2^{1/n} - 1)$. Desta forma, quando n tende ao infinito, U é limitado a aproximadamente 70%.

Quando acrescentamos à Equação (1), o tempo necessário para troca de contexto da tarefa (e_{cs}) e o tempo necessário para decidir qual é a próxima tarefa (e_{sch}), considerando que seus períodos são os mesmos das tarefas executadas, fica fácil perceber o impacto que a ação do RTOS tem no fator de utilização do programa executado. A Equação (3) determina o fator de utilização levando-se em conta a ação do RTOS.

$$U = \left(\sum_{i=1}^n \frac{e_i + e_{cs} + e_{sch}}{p_i} \right) \cdot 100\% \quad (3)$$

O fator de utilização U da CPU aumenta proporcionalmente com o aumento na frequência de troca de tarefas, preservando-se o número de tarefas e seus tempos de execução. Quanto maiores forem os tempos e_{cs} e e_{sch} demandados pelo RTOS, mais rapidamente o limite teórico de 70% será alcançado (31).

2.5 HARDWARE DIGITAL PROGRAMÁVEL

Este projeto foi totalmente desenvolvido, testado e validado utilizando tecnologia de hardware digital programável. Por conta disso, nesta seção são apresentados os principais tipos de dispositivos digitais baseados em hardware configurável, classificados como dispositivos lógicos programáveis e amplamente reconhecidos pela sigla PLD (*Programmable Logic Device*) (11). Além disso, ainda nesta seção são descritas as linguagens de descrição de hardware e o fluxo de projetos de circuitos integrados (CIs) digitais baseados em PLDs.

2.5.1 Dispositivos lógicos programáveis

Até o início da década de 1970, haviam apenas três alternativas para desenvolvimento de circuitos digitais (11): (i) uso de CIs convencionais; (ii) utilização de memórias PROM (*programmable read-only memory*), que ao invés de armazenarem dados, eram usadas para que executassem expressões booleanas, e; (iii) contratar um fabricante para que produzisse circuitos integrados com características específicas que atendessem ao projeto em desenvolvimento, os chamados de ASICs. Esses CIs eram muito caros e necessitavam de grande escala de produção para se tornarem comercialmente viáveis (11). Hoje em dia, o desenvolvimento das máscaras litográficas usadas durante a produção de um CI pode custar mais de uma dezena de milhões de dólares (11;33).

Desde então, muitas tecnologias foram criadas para permitir a desenvolvimento de circuitos utilizando PLDs. Até que no ano de 1984, a empresa Altera lançou a primeira família de circuitos integrados CPLD (*Complex Programmable Logic Device*, em português dispositivo lógico programável complexo), que tinha capacidade equivalente a aproximadamente 30.000 portas lógicas. Paralelamente a esse desenvolvimento, no ano de 1985 a empresa Xilinx

anunciou um novo tipo de arquitetura que não derivava de nenhum dos modelos anteriores e já solucionava um dos problemas cruciais do CPLD, o crescimento exponencial da matriz de interconexões, à medida que o número de células do circuito crescia (11). Esse novo tipo de arquitetura para PLDs recebeu o nome de FPGA (11).

Um FPGA é uma estrutura de células lógicas e interconexões totalmente controladas pelo projetista (11). Uma célula lógica, aqui chamada de bloco lógico, não possui matrizes de portas AND ou OR, na verdade dispõe de portas lógicas e blocos lógicos independentes que podem ser interconectados de acordo com a necessidade (11).

Os CPLDs e os FPGAs formam um grupo de dispositivos conhecidos como HCPLDs (*High-Capacity PLDs*), ou seja, PLDs de alta capacidade (11). Esse grupo é caracterizado por dispositivos programáveis com milhares de portas lógicas, podendo chegar a milhões de portas (11).

Esses dispositivos oferecem uma série de vantagens ao projetista (11), são elas:

- São adequados para projetos simples e complexos;
- Diminuem o espaço dos circuitos na placa de circuito impresso;
- Oferecem menos consumo de potência elétrica em relação ao uso de múltiplos CIs, uma vez que trabalham com tensões internas mais baixas que os padrões utilizados em circuitos dedicados e podem ser parcialmente energizados;
- Trabalham em velocidades muitas vezes superiores às velocidades dos processadores ou microcontroladores industriais, uma vez que não executam apenas ações síncronas e permitem intenso paralelismo na lógica implementada;
- São ideais para protótipos e produtos, uma vez que permitem modificações ao longo do projeto e também após a conclusão;
- Reduzem o tempo de desenvolvimento entre o projeto e o produto final, permitindo maior competitividade das empresas que usam essa tecnologia.

Além disso, o projeto pode ser realizado de diferentes maneiras, utilizando álgebra booleana, mapa de Karnaugh, linguagens de descrição de hardware (HDLs), entre outras (11).

Podemos classificar os sistemas digitais como mostrado na Figura 12, onde quatro grandes grupos podem ser visualizados: lógica padrão, PLDs, ASICs e processadores (11).

Figura 12 – Classificação dos circuitos integrados digitais utilizados nos sistemas digitais.

Fonte: Autor.

2.5.2 Linguagens de descrição de hardware

Em meados da década de 1980, as primeiras linguagens de descrição de hardware (HDLs) foram criadas. Seu objetivo era auxiliar os projetistas de sistemas digitais a descreverem os circuitos desenvolvidos. Isso foi de suma importância, uma vez que os dispositivos se tornavam cada vez mais complexos, bem como as soluções criadas (11).

As duas principais linguagens utilizadas atualmente são (11):

- **VHDL** (*Very High Speed Integrated Circuit **H**ardware **D**escription **L**anguage*): foi criada pelo Departamento de Defesa dos Estados Unidos no início da década de 1980 com o objetivo de ser uma maneira compacta para descrever os projetos de circuitos integrados de alta velocidade. Logo foram desenvolvidos programas de computador que podiam ler arquivos escritos em VHDL e simular a lógica desses CIs. O sucesso dessa linguagem foi eminente por ser padronizada pelo IEEE (*Institute of Electrical and Electronics Engineers*). Sua última revisão foi finalizada em 2008 e é identificada como padrão IEEE 1076-2008. A linguagem VHDL permite o desenvolvimento, a verificação, a síntese e o teste do hardware projetado;

- **Verilog**: foi lançada no ano de 1983 e inicialmente era uma linguagem proprietária, porém depois de muita pressão, no ano de 1991, ela foi tornada pública e padronizada. Sua última revisão foi feita no ano de 2002 e lançada em 2003. Ela foi identificada como padrão IEEE 1364-2001 Revision C. A linguagem Verilog tem abrangência semelhante à do VHDL e a padronização pelo IEEE também a tornou muito popular.

Essas linguagens são reconhecidas por todos os softwares EDA (*Electronic Design Automation*) disponibilizados pelos fabricantes de dispositivos lógicos programáveis e também é reconhecida pelos programas utilizados no projeto de ASICs, como o Encounter da Cadence (7).

2.5.3 Fluxo de projeto de CIs digitais implementados em PLDs

A criação de projetos de CIs digitais utilizando PLDs (CPLDs e FPGAs) é fundamentalmente dependente de um ambiente de software EDA. Este projeto utilizou como ferramenta de desenvolvimento o software Quartus II *Web Edition* versão 15.0, que era fornecido pela empresa Altera na data de início do projeto (25).

A Figura 13 apresenta o fluxograma da metodologia utilizada no desenvolvimento de projetos de CIs digitais com lógica programável (11) e que foi aplicada a esse projeto.

Figura 13 – Metodologia de projetos de circuitos integrados digitais com lógica programável.

Fonte: Autor.

Apesar da programação do dispositivo ser a última etapa da metodologia apresentada na Figura 13. Durante o desenvolvimento deste projeto, verificou-se que, à medida que os blocos de hardware eram integrados e a complexidade do circuito aumentava, a ferramenta EDA, utilizada para simulação de circuitos utilizada (ModelSim), não era mais capaz de lidar com o circuito. Sendo necessário uso de outras técnicas para verificação do funcionamento do circuito implementado. Nestes casos, o uso de dispositivos do kit de desenvolvimento utilizado, bem como o uso de osciloscópio digital foi essencial.

A Figura 14 ilustra o kit de desenvolvimento DE1-SoC fabricado pela Terasic (51), que utiliza tecnologia FPGA da família Cyclone V, atualmente fabricado pela Intel (anteriormente Altera) (24). Este foi o kit utilizado durante o desenvolvimento deste projeto.

Figura 14 – Kit de desenvolvimento DE1-SoC da Terasic Technologies.

Fonte: Terasic Technologies (51).

O kit, apresentado na Figura 14, utiliza uma família de FPGAs ideal para desenvolvimento de sistemas completos (*System-on-Chip*, SoC), além de possuir toda a parte de lógica programável, dispõe de um processador *dual-core* ARM Cortex-A9 que opera a 800 MHz. Por conta disso, parte de seus periféricos são dedicados exclusivamente ao processador (*Hard Processor System*, HPS) e a outra parte pode ser conectada ao circuito programável (FPGA). Esta última parte, disponível ao projeto desenvolvido, conta com os seguintes periféricos:

- 64 MB de memória SDRAM;
- *push-buttons*;
- 10 chaves do tipo *slide*;
- 10 *leds* vermelhos;
- 6 displays de 7 segmentos;
- fontes de *clock* de 50 MHz;
- 1 codificador de áudio de 24 bits com qualidade de CD;
- 1 decodificador digital-analógico VGA de 8 bits;
- 1 decodificador de TV (NTSC/PAL/SECAM);
- 1 conector PS/2 para mouse e teclado;
- 1 receptor/emissor infravermelho;
- 2 expansões de 40 pinos com proteção a diodo;
- 1 conversor analógico-digital com interface SPI de 4 pinos.

Parte desses periféricos foram utilizados durante o desenvolvimento, depuração e testes de aplicações para o *microkernel* por hardware, proposto neste projeto.

2.6 CIRCUITO INTEGRADO DEDICADO

Um projeto previamente validado em FPGA pode ser convertido em um leiaute de um ASIC. Para tal, nesta pesquisa foram utilizadas duas ferramentas de software da Cadence (7) chamadas de Encounter RTL Compiler (RC) e Encounter Digital Implementation (EDI).

Diferentemente dos FPGAs, que modelam o hardware com base em estruturas lógicas pré-desenvolvidas e extremamente configuráveis chamadas de *look-up tables* (LUTs) entre outras, o projeto de um circuito integrado dedicado (ASIC) utiliza como base blocos elementares como portas lógicas para descrever todos os circuitos. Essa proposta tende a ser vantajosa quando grandes volumes de produção estão envolvidos, uma vez que o custo de produção é baixo, porém tem custo de projeto elevado quando comparado ao uso de FPGAs (26).

2.6.1 Encounter RTL Compiler

O RC é uma ferramenta de síntese lógica. Seu papel é transformar a descrição abstrata do comportamento do CI desejado, feita em HDL e chamada de *Register Transfer Level*

Synthesis Model (RTL *Sysntesys Model*), em um arquivo implementado em termos de interligações entre portas lógicas (*Gate-Level Netlist*) correspondentes às disponíveis na tecnologia desejada para fabricação do ASIC. Para isso, o RC junta aos arquivos em HDL do projeto às informações da tecnologia de fabricação necessárias para montagem do leiaute. Somente assim, após realizada a síntese, todo o projeto do CI pode ser descrito com base na interligação de portas lógicas.

Apesar do RC possuir uma interface gráfica (GUI), ela é utilizada apenas para verificação e extração de relatórios. Toda a elaboração da síntese do projeto é executada através de comandos ou *scripts* escritos usando *Tool Command Language* (TCL).

A Figura 15 apresenta, de forma simplificada, as entradas e saídas de uma síntese de projeto realizada através do RC.

Figura 15 – Fluxo de dados para realização da síntese do *gate-level netlist* no ambiente do Encounter RTL Compiler.

Fonte: Autor.

Para iniciar o fluxo do RC os seguintes arquivos são necessários:

- Bibliotecas para síntese em formato Liberty (extensão LIB) contendo:
 - Informações de temporização;
 - Informações de dissipação interna de potência das células;
 - Informações de dissipação de potência por fuga.
- Bibliotecas para síntese com extensão LEF (*Library Exchange Format*) contendo:

- Informações sobre as camadas de metal presentes, suas dimensões e as dimensões das vias utilizadas para interligação entre camadas;
- Informação abstrata sobre a formato de cada célula, pinagem e sua distribuição nas camadas de metal;
- Arquivos RTL: são os arquivos escritos em HDL (Verilog ou VHDL) que podem conter código estrutural para combinar módulos de nível inferior, especificações comportamentais de projeto ou implementações em RTL.

As bibliotecas utilizadas normalmente são fornecidas no formato de um pacote de arquivos da tecnologia, chamado de *Plug-in Development Kit* (PDK). Sua distribuição é feita pela *foundry* ou seu parceiro, como no caso desta pesquisa, a MOSIS.

Após a síntese do projeto, ao menos dois tipos de arquivos são gerados:

- Arquivo *Gate-Level Netlist*: é o arquivo Verilog onde todos os arquivos RTL foram convertidos e sintetizados usando apenas as portas lógicas descritas nas bibliotecas da tecnologia utilizada.
- Arquivo de restrições SDC: a extensão de arquivo SDC significa *Synopsys Design Constraints* (restrições de projeto Synopsys), nele são gravadas as restrições atuais de projeto em termos de área, tempos de atrasos e potência dissipada.

2.6.2 Encounter Digital Implementation

O EDI é uma ferramenta de síntese física (*Place & Route*, em português “posicionamento e roteamento”). Seu papel é transformar o *gate-level netlist* em um leiaute funcional com base nas informações da tecnologia escolhida e nas restrições do projeto (frequência, área do *die*, etc.). Ao contrário do RC, o EDI possui uma interface gráfica que pode ser utilizada para realização de todos os passos necessários para geração do leiaute. Outra opção é a utilização de comandos ou *scripts* TCL.

Para iniciar o fluxo do EDI os seguintes arquivos são necessários:

- Arquivo *gate-level netlist*: é o arquivo Verilog gerado pelo RC.
- Bibliotecas para síntese com extensão LEF (*Library Exchange Format*): as mesmas utilizadas para a síntese feita pelo RC.
- Arquivo MMMC (*Multi-Mode Multi-Corner Analysis*): arquivo que contém tabelas de capacitâncias da tecnologia e referência o arquivo SDC gerado pelo RC com as restrições de área, tempos de atrasos e potência dissipada do projeto.

Assim que os arquivos são carregados no EDI é possível visualizar o projeto na forma de blocos hierárquicos limitados de acordo com o número mínimo de elementos lógicos que compõe cada um deles. A Figura 16 ilustra um exemplo com uma série de blocos principais do projeto do microcontrolador desenvolvido nesta pesquisa e a área reservada para disposição de todo o circuito.

Figura 16 – Exemplo de blocos hierárquicos (entidades) do projeto do circuito do microcontrolador IHM-Plasma e área reservada para o roteamento do leiaute do ASIC.

Fonte: Autor.

As etapas seguintes para elaboração do leiaute podem ser resumidas a seguir e são representadas na Figura 17:

- *Floorplan*: definição do tamanho do *die* e posicionamento dos seus blocos hierárquicos [Figura 17(a)];
- *Power*: definição dos pinos de alimentação (VDD e VSS), criação do anel e linhas de alimentação (*power ring* e *stripes*) no *die*, e roteamento dos sinais de alimentação [Figura 17(b)];
- *Placement*: colocação das células padrão (*standard cells*) de cada bloco e pré-roteá-las [Figura 20(c)];
- *Route*: executa a conexão entre todas as células [Figura 17(d)];
- *Verification*: checagem das regras do projeto (design rule check, DRC), definidas nos arquivos do projeto [Figura 20(e)];

Figura 17 – Etapas do fluxo do projeto do ASIC.

Fonte: Autor.

Após concluídas todas essas etapas, o fluxo do projeto pode ser finalizado com a geração de um arquivo GDS II (*Graphic Data System*), que é o formato padrão de arquivos com a descrição de circuitos integrados. Esse arquivo é usado pelas *foundries* como referência para fabricação dos CIs.

3 O PROJETO DE PESQUISA

Este capítulo é iniciado com a descrição do *microkernel* proposto por este projeto de pesquisa. As seções subsequentes descrevem seu desenvolvimento, detalhes sobre sua configuração e os programas que controlam o seu funcionamento. Por fim, objetivando a futura fabricação de um ASIC, é apresentado o leiaute gerado de um CI CMOS dedicado (ASIC), contendo um microcontrolador Plasma modificado integrado ao *microkernel* proposto.

3.1 PROPOSTA DE UM *MICROKERNEL* POR HARDWARE PARA SISTEMAS BASEADOS EM TAREFAS

Como introduzido na seção 1.3, este projeto de pesquisa propõe o desenvolvimento de um inovador *microkernel* para sistemas embarcados baseados em tarefas, que diferentemente da abordagem convencional, onde o *microkernel* é escrito por um software (parte de um RTOS), aqui ele é inteiramente desenvolvido por hardware, se comportando como um periférico especializado no escalonamento e despacho de tarefas. Enquanto periféricos convencionais, como as unidades de entrada/saída (I/Os) e portas de comunicação serial usam conjuntos de registradores compartilhados e mapeado em memória para que seja possível a comunicação com a CPU, o *microkernel* por hardware proposto é profundamente interconectado a diferentes partes da CPU e memórias, que por sua vez é capaz de monitorar sinais, bem como controlar partes de seu hardware sem comprometer a execução das instruções do programa a ser executado. Desta forma, a técnica desenvolvida permite uma perfeita interação entre ambos dispositivos, com o objetivo de criar um novo ecossistema (arcabouço) de hardware (*microkernel*/CPU/memórias), que é capaz de minimizar os gargalos existentes nos sistemas baseados em tarefas. Por conta da forma inovadora com que o *microkernel* proposto se interconecta a CPU e memórias, ele foi chamado de *Interlocked Hardware Microkernel* (IHM).

O IHM é formado por um bloco escalonador, um bloco gerenciador para troca do contexto das tarefas e um bloco de temporização e controle, minuciosamente interconectados com a CPU e a memória. Dessa forma, é possível criar programas baseados em tarefas e executá-las automaticamente de acordo com o *time slice* especificado, suas prioridades e o estado atual de cada uma. Para o escalonamento de tarefas, o escalonador desenvolvido utiliza um algoritmo híbrido baseado no **algoritmo Round-Robin**, que define os *time slices* para as tarefas a serem executadas, e no uso de **preempção**, que é baseada na prioridade atribuída à

cada tarefa. Outra opção disponível é a própria tarefa em execução se colocar em estado de bloqueio ou suspensão. Para isso, a tarefa precisa realizar alterações em alguns registradores.

Existem duas razões principais para que o hardware proposto seja capaz de otimizar e minimizar o tempo necessário para trocas das tarefas, e consequentemente, gerar ganho de desempenho no sistema, são elas:

1. As decisões e as análises feitas pela CPU, com base no código do escalonador executado, passaram a ser feitas por hardware auxiliar operando em paralelo à execução das tarefas;
2. A troca de contexto (*context switching*), que normalmente ocorre após um *tick interrupt*, passou a ser iniciada pelo gerenciador de contexto antes do final do *time slice* e concluída após a próxima tarefa se encontrar em execução. Dessa forma, a troca de contexto ocorre em uma quantidade de ciclos de *clock* muito menor do que a realizada, caso esse procedimento fosse feito via software pela CPU.

A Figura 18 exemplifica a operação de um sistema com duas tarefas com mesma prioridade, que são escalonadas por um *microkernel* por software (a) e pelo IHM (b). O eixo vertical representa a ação do *microkernel* e a execução das tarefas pela CPU, o eixo horizontal representa o tempo dado em *time slices*.

Figura 18 – Fluxo de execução de tarefas com *microkernel* por software (a) e utilizando o IHM (b).

Fonte: Autor.

Considerando a solução por software [Figura 18 (a)], a execução das tarefas é interrompida para que a CPU realize o salvamento do contexto da tarefa interrompida, execute o algoritmo de escalonamento, e então carregue o contexto da próxima tarefa a ser executada. Na nova abordagem proposta por hardware [Figura 18 (b)], o IHM está permanentemente operando em paralelo à CPU, cuidando de todas as tarefas descritas anteriormente na abordagem por software, enquanto que a CPU se encarrega exclusivamente da execução das

tarefas. Assim que o *time slice* chega ao fim, o IHM apenas dispara a troca da tarefa que deve ser executada pela CPU (*task switching*).

Deve-se notar que a expressão *context switching* foi substituída por *task switching*, essa substituição foi proposital e é justificada pela ação antecipada do gerenciador de contexto, como descrito na segunda razão para se ter um ganho de desempenho nos sistemas baseados em tarefas. Do ponto de vista da CPU, é como se ela saltasse da execução de uma tarefa diretamente para execução de outra.

Para que o IHM pudesse ser testado e validado, foi necessário utilizar um microcontrolador como base para sua integração. Como descrito na seção 2.2.3, o microcontrolador escolhido foi o Plasma. Além das razões apresentadas, ele possui uma arquitetura do tipo *load-store*, ideal para a integração do IHM.

Embora o Plasma seja um dispositivo confiável, isso não nos isentaram de validar cada um dos blocos de seu núcleo separadamente, bem como de inserir comentários em 100% do código fonte. Esta foi uma tarefa essencial para permitir a perfeita interação entre o IHM com a nova CPU Plasma customizada. Toda validação de código foi feita através do software simulador de hardware ModelSim, bem como através do uso do kit de desenvolvimento DE1-SoC fabricado pela Terasic (51), que utiliza tecnologia FPGA da família Cyclone V (24) e pode ser visto na Figura 14 na seção 2.5.3.

O aplicativo utilizado como ambiente de desenvolvimento foi o Quartus II *Web Edition* versão 15.0, que era fornecido pela empresa Altera na data de início deste projeto (25).

Cabe aqui ressaltar que todas as modificações implementadas no núcleo, bem como todos os blocos criados para o IHM, ocorreram somente após o estudo completo e minucioso da arquitetura do microcontrolador Plasma, onde foi necessário o entendimento de todas as interações entre os blocos propostos por seu idealizador, bem como, o estudo de seu conjunto de instruções (*Assembly*) e a escolha da melhor ferramenta para conversão dos códigos criados em arquivos binários [MARS (36)]. Esse processo consumiu em torno de um ano de trabalho. Somente após esse período, as modificações e implementações começaram a ser feitas e validadas.

3.2 IMPLIMENTAÇÃO DA ARQUITETURA DO IHM-PLASMA

Esta seção apresenta, de forma detalhada, as características do novo microcontrolador baseado na arquitetura da CPU do microcontrolador Plasma e na implementação do *microkernel* por hardware, ou seja, o IHM (Figura 19).

Como descrito na seção 3.1, o IHM foi dividido em três blocos principais: (i) Escalonador (ESC); (ii) Gerenciador de Contexto (GC) e; (iii) Temporização e Controle (T&C). Estes blocos têm como função principal, cuidar do escalonamento de tarefas e das trocas de contexto das tarefas executadas pela CPU. Para que sua interação com a CPU ocorra de forma a permitir ganhos significativos de desempenho, são necessárias modificações estruturais na arquitetura da CPU e na forma como é feito o acesso a memória de dados/programa. Dentro deste contexto, as principais modificações feitas na CPU foram: (i) criação de um conjunto de registradores internos (*internal register bank*, IRB) extra; (ii) nova entrada para o PC com o endereço da próxima tarefa controlada pelo IHM. Já nas memórias, as modificações foram: (i) criação de novos registradores com funções especiais (*special function registers*, SFRs) para configuração e monitoração do IHM; (ii) divisão da SRAM em três partes, sendo uma com acesso reversível entre a CPU e o IHM para criação dos TCBs.

A Figura 19 apresenta, de forma simplificada, esta nova arquitetura de microcontrolador, que considera todas as modificações descritas anteriormente e integra o IHM. Desta forma, esta nova arquitetura de microcontrolador foi chamada de IHM-Plasma. Esse nome é inspirado no microcontrolador base do projeto (Plasma) e nas funcionalidades do *microkernel* integradas à CPU de forma não convencional, que favorece uma perfeita coexistência entre ambos e permite ganhos reais de desempenho dos sistemas baseados em tarefas.

Figura 19 – Diagrama de blocos simplificado da arquitetura do microcontrolador IHM-Plasma.

Fonte: Autor.

Na Figura 19 são apresentados, com fundo xadrez, todas as principais implementações e os blocos da CPU e da memória que foram alterados. Já a Tabela 5 apresenta as especificações técnicas do microcontrolador IHM-Plasma. Mais detalhes são abordados nas próximas seções deste capítulo, onde o IHM, a CPU e a memória são analisados separadamente.

Tabela 5 – Características principais do IHM-Plasma.

Característica	Valor
Quantidade máxima de tarefas	32
Memória de programa	8 kB ou 2 k <i>words</i>
Memória de dados	8 kB ou 2 k <i>words</i>
Memória de dados/TCB (reversível)	8 kB ou 2 k <i>words</i>
TCB	32 com 64 <i>words</i> /tarefa
Frequência de operação	25 MHz
Pinos de entrada	32
Pinos de saída	32
UART	1 operando a 57.600 bps

Fonte: Autor.

O projeto completo do microcontrolador IHM-Plasma totalmente descrito em linguagem VHDL e ajustado para síntese e execução em FPGA do kit de desenvolvimento DE1-SoC da Terasic Technologies (51) está disponível em formato digital e pode ser acessado através do link na referência (12).

3.2.1 Alterações na CPU do microcontrolador Plasma

O IHM foi integrado à CPU Plasma que usa a ISA MIPS I. Porém, os blocos a serem modificados nesta CPU são comuns a todas as arquiteturas de processadores que utilizam exclusivamente instruções do tipo *load* e *store* para fazer o acesso à memória de dados. Nestes casos, a unidade lógica e aritmética (ULA) não é capaz de operar com dados diretamente na memória, e portanto todas as operações devem ser feitas usando o banco de registradores internos da CPU.

Neste projeto, duas alterações na CPU Plasma foram determinantes para o sucesso da integração dela com o IHM. Ambas foram feitas nos circuitos responsáveis pelo contexto do programa ou tarefas em execução: o Contador de Programa (PC); e o Banco Interno de Registradores (IRB).

I – Alteração na estrutura do PC

O PC é um contador que é responsável por armazenar o endereço da próxima instrução a ser executada pela CPU. Sempre que a CPU finaliza um ciclo de busca, o conteúdo do registrador PC é incrementado de 4 unidades, pois cada instrução tem um tamanho de 4 bytes. Nos casos de execução de instruções de desvios incondicionais/condicionais pela CPU, o endereço da próxima instrução é definido pela própria instrução ou pelo resultado de uma operação realizada pela ULA, e é armazenado no conteúdo do registrador PC.

Para que o IHM possa realizar a troca da tarefa em execução, saltos (*jumpers*) no programa também são necessários. Esses saltos intercalam-se partes dos códigos das tarefas que estão sendo executadas ao longo do tempo. Para permitir a troca da tarefa em execução, um novo bloco composto por sinais de endereçamento, sinais de controle e um multiplexador foi adicionado na entrada do registrador do PC. Dessa forma, o IHM tornou-se responsável pela alteração do valor do conteúdo do registrador PC para que ele possa ter o endereço da próxima tarefa a ser executada. O circuito de saída do PC permanece inalterado e interligado às entradas de endereçamento da memória de programa. A Figura 20 ilustra essa alteração.

Figura 20 – Alteração realizada no conjunto de entradas do Contador de Programa da CPU Plasma para que fosse possível a integração da CPU com o IHM.

Fonte: Autor.

II – Implementação de novo IRB

O IRB da CPU Plasma forma uma pequena memória do tipo multiportas com capacidade para acesso simultâneo ao conteúdo de dois registradores e gravação de um terceiro registrador. Dessa forma, essa memória possui duas entradas de endereçamento para leitura (A1, A2) e uma para escrita (A3). Esses sinais, bem como, o sinal de controle de escrita (WE3) e os sinais de dados (RD1, RD2 e WD3) são originários de diferentes partes da CPU e podem ser vistos na Figura 21.

Este projeto de pesquisa propõe o acréscimo de mais um IRB à CPU, idêntico ao original. Ambos serão usados simultaneamente, enquanto a CPU executa uma tarefa através de um deles (*running set*), o IHM (bloco GC) acessa o outro (*waiting set*) para fazer o *backup* do contexto das informações relativas a tarefa que saiu de execução e copiar as informações relativas do contexto da próxima tarefa a ser executada. A cada *tick interrupt*, os conjuntos de registradores (IRBs) são alternados entre a CPU e o *microkernel*. O bloco T&C do IHM é o responsável por decidir quando alternar os IRBs. As alterações propostas são mostradas com fundo xadrez na Figura 21.

Figura 21 – Implementação de um adicional IRB à CPU Plasma (em xadrez) para que seja possível a integração da CPU com o IHM.

Fonte: Autor.

3.2.2 Alterações realizadas na memória do microcontrolador Plasma

Uma das técnicas mais simples utilizada para alterar a arquitetura de um microcontrolador para se ter o acesso a novos periféricos é chamada de I/O mapeada em memória (*memory-mapped I/O*) (21). Dessa maneira, parte dos endereçamentos da memória é mapeada para acessar as unidades de Entrada e Saída (I/Os) através dos respectivos Registradores de Funções Especiais (*Special Function Registers*, SFRs) (9) dos periféricos que se deseja conectar à CPU. Assim, a CPU pode acessar os periféricos simplesmente com a utilização das instruções de memória. Nessa abordagem, a memória mapeada funciona como um *middleware* ou interface entre a CPU e seus periféricos (21). A Figura 22 ilustra um exemplo em que a CPU tem acesso aos pinos de entrada e saída de dados (*port* ou porta em português) e a uma interface serial de comunicação Transmissor Receptor Assíncrono Universal (*Universal Asynchronous Receiver/Transmitter*, UART). À esquerda observamos a configuração física do hardware (a) e à direita a configuração lógica (b) do ponto de vista do programa escrito para se ter acesso aos periféricos.

Figura 22 – Integração de uma porta de entrada e saída de dados e uma UART a uma CPU por meio de I/O mapeada em memória de acesso aleatório (*Random Access Memory*, RAM). Em (a) temos a visão física do hardware e em (b) a visão lógica do mesmo.

Fonte: Autor “adaptado de” Harris (21), 2012, página 507.

Observando-se a Figura 22 (b) e em outros casos também, o acréscimo de periféricos não afeta a forma como o programa é executado, ele apenas agrega novas funcionalidades ao processador. Essa técnica foi usada para implementação dos SFRs criados para configuração e controle do IHM. Além disso, a memória RAM interna, que era um bloco contínuo para dados e programa, foi dividida em três regiões (programa, dados e dados/TCB).

A seguir, são apresentados os detalhes referentes a estas duas modificações.

I – Novos SFRs para configuração e monitoração do IHM

O projeto do IHM demandou a criação de um conjunto com nove novos registradores de funções especiais. A seguir são listadas as funções de cada um deles:

- `reg_sch` (0x20000080): responsável pela habilitação do escalonamento de tarefas. Apenas o bit 0 é utilizado para ligar (1) ou desligar (0) o escalonador;
- `tick` (0x20000090): seu valor representa o período de duração de um *time slice* medido em períodos de *clock*. Apenas os 30 bits menos significativos são considerados. Levando-se em conta que a frequência do *clock* ajustada para o microcontrolador é de 25 MHz, o período máximo para o disparo de uma troca de tarefas é de aproximadamente 42,95 segundos, muito mais do que a ordem de

grandeza normalmente utilizada, que costuma ser em torno de algumas centenas de microssegundos a alguns milissegundos;

- `task_number` (0x200000A0): este é o único registrador de saída do *microkernel* por hardware, seu valor é atualizado com o número da tarefa em execução pela CPU (de 0 a 31). Apenas o bit correspondente ao número da tarefa assume valor 1. Pode ser usado para depuração de programas ou controle de instância de uma mesma tarefa;
- `task_live` (0x200000B0): cada bit deste registrador representa o estado de uma das 32 possíveis tarefas. Valor 1 significa que a tarefa foi criada e pode ser escalonada e 0 significa que a tarefa não foi criada ou está suspensa;
- `task_pri7_0` (0x200000C0): cada *nibble* (conjunto de 4 bits) é responsável por armazenar o valor da prioridade atribuída a uma das tarefas. Apesar de serem 4 bits para cada tarefa, o *microkernel* por hardware trabalha com apenas 8 níveis de prioridade (0 a 7), quanto maior for o valor, maior será a prioridade da tarefa;
- `task_pri15_8` (0x200000D0): semelhante ao anterior, porém destinado às tarefas de 8 a 15.
- `task_pri23_16` (0x200000E0): semelhante ao anterior, porém destinado às tarefas de 16 a 23.
- `task_pri31_24` (0x200000F0): semelhante ao anterior, porém destinado às tarefas de 24 a 31.
- `task_sleep` (0x20000100): força o bloqueio assíncrono da tarefa em execução, ou seja, não aguarda o fim do *time slice* para troca da tarefa. A tarefa pode ser bloqueada até o final do *time slice* atual (valor 0) ou por até 65.535 *time slices*. Os bits de 0 a 15 armazenam o período desejado de bloqueio e o bit 16 deve ser pulsado positivamente para disparar o pedido de bloqueio. Os demais bits não são utilizados.

II – Divisão da memória RAM

A divisão da faixa de endereços para memória RAM interna foi adotada para facilitar a análise e separação dos dados. Três células de memória idênticas foram instanciadas no projeto, sendo que as duas primeiras foram conectadas diretamente ao barramento de dados e programa, se comportando como uma só. Já a terceira célula de memória tem sua conexão comutável entre a CPU e o IHM (bloco GC) através do bit 0 do registrador responsável pela habilitação do escalonamento de tarefas (`reg_sch`). Enquanto esse bit vale 0, essa faixa de memória se

comporta como uma memória de dados de propósito geral e pode ser acessada pela CPU. Ao alterar o bit para nível lógico 1, o bloco GC passa a ter acesso exclusivo sobre ela, que é dividida em 32 TCBs, uma para cada possível tarefa. A Figura 23 ilustra a divisão da memória e a multiplexação de parte dela para armazenamento dos TCBs.

Figura 23 – Divisão da memória interna do microcontrolador IHM-Plasma para uso do *microkernel* por hardware.

Fonte: Autor.

Na Figura 24 é apresentado a divisão interna do bloco de memória reservado para os TCBs. Cada TCB possui 64 *words* de 4 bytes cada, onde 32 *words* são utilizadas para armazenamento do contexto da tarefa correspondente e os outros 32 restantes estão disponíveis para aplicações futuras. Neste projeto, o contexto de uma tarefa é composto pelo valor do PC e o estado de 31 dos 32 registradores internos da CPU (apenas o registrador \$0 é descartado por seu valor ser fixo em zero). Os 8 kB reservados para os TCBs são suficientes para armazenar até 32 tarefas, limite estabelecido para este projeto e semelhante aos apresentados nas referências (15;32;38).

Figura 24 – Modelo de distribuição dos TCBs na memória RAM.

Fonte: Autor.

Na primeira posição (4 bytes) de cada TCB é salvo o valor do PC e nas subsequentes são salvos os registradores de 1 a 31. A Tabela 6 apresenta o mapa de memória original e o novo mapeamento para uso do IHM. Já a Tabela 7 apresenta o mapeamento detalhado dos 32 TCBs na memória RAM.

Tabela 6 – Mapa de memória do microcontrolador Plasma original e do microcontrolador IHM-Plasma.

Endereço	Descrição	Plasma	IHM-Plasma
0x00000000 - 0x0000ffff	RAM Interna (8 kB) programa e uso geral	✓	✓
0x00010000 - 0x0001ffff	RAM Interna (8 kB) uso geral ou <i>stack</i> das tarefas		✓
0x00020000 - 0x0002ffff	RAM Interna (8 kB) uso geral ou TCBs		✓
0x10000000 - 0x100fffff	RAM Externa (1 MB)	✓	✓
Registadores	Todos de 32 bits (4 bytes endereçáveis)		
0x20000000	Escrita na UART	✓	✓
0x20000000	Leitura da UART	✓	✓
0x20000010	Máscara de interrupções (IRQ_Mask)	✓	✓
0x20000020	Estado das interrupções	✓	✓
0x20000030	GPIO0 Set (ativa os pinos do <i>port</i> de saída)	✓	✓
0x20000040	GPIO0 Clear (<i>reseta</i> os pinos do <i>port</i> de saída)	✓	✓
0x20000050	GPIOA In (<i>port</i> de entrada)	✓	✓
0x20000060	Counter (contador)	✓	✓
0x20000070	Ethernet	✓	✓
0x20000080	Habilita o escalonamento (reg_sch)		✓
0x20000090	Tempo do <i>time slice</i> em pulsos de <i>clock</i> (tick)		✓
0x200000A0	Tarefa em execução (task_number)		✓
0x200000B0	Vetor de tarefas (ativa/não ativa, máx. 32) (task_live)		✓
0x200000C0	Vetor de prioridade das tarefas 7-0 (4 bits por tarefa) (task_pri7_0)		✓
0x200000D0	Vetor de prioridade das tarefas 15-8 (task_pri15_8)		✓
0x200000E0	Vetor de prioridade das tarefas 23-16 (task_pri23_8)		✓
0x200000F0	Vetor de prioridade das tarefas 31-24 (task_pri31_24)		✓
0x20000100	Tempo de <i>sleep</i> ou <i>yield</i> da tarefa atual (task_sleep)		✓
Bits de interrupção			
7	GPIO31	✓	✓
6	GPIO31'	✓	✓
5	EthernetSendDone	✓	✓
4	EthernetReceive	✓	✓
3	Counter(18)	✓	✓
2	Counter(18)'	✓	✓
1	UartWriteBusy'	✓	✓
0	UartDataAvailable	✓	✓

Fonte: Autor.

Tabela 7 – Mapeamento de memória RAM reservada para os TCBs utilizados pelo IHM.

TCB/Tarefa	Endereço Contexto		Endereço Livre	
	Inicial	Final	Inicial	Final
0	0x00020000	0x0002007F	0x00020080	0x000200FF
1	0x00020100	0x0002017F	0x00020180	0x000201FF
2	0x00020200	0x0002027F	0x00020280	0x000202FF
3	0x00020300	0x0002037F	0x00020380	0x000203FF
4	0x00020400	0x0002047F	0x00020480	0x000204FF
5	0x00020500	0x0002057F	0x00020580	0x000205FF
6	0x00020600	0x0002067F	0x00020680	0x000206FF
7	0x00020700	0x0002077F	0x00020780	0x000207FF
8	0x00020800	0x0002087F	0x00020880	0x000208FF
9	0x00020900	0x0002097F	0x00020980	0x000209FF
10	0x00020A00	0x00020A7F	0x00020A80	0x00020AFF
11	0x00020B00	0x00020B7F	0x00020B80	0x00020BFF
12	0x00020C00	0x00020C7F	0x00020C80	0x00020CFF
13	0x00020D00	0x00020D7F	0x00020D80	0x00020DFF
14	0x00020E00	0x00020E7F	0x00020E80	0x00020EFF
15	0x00020F00	0x00020F7F	0x00020F80	0x00020FFF
16	0x00021000	0x0002107F	0x00021080	0x000210FF
17	0x00021100	0x0002117F	0x00021180	0x000211FF
18	0x00021200	0x0002127F	0x00021280	0x000212FF
19	0x00021300	0x0002137F	0x00021380	0x000213FF
20	0x00021400	0x0002147F	0x00021480	0x000214FF
21	0x00021500	0x0002157F	0x00021580	0x000215FF
22	0x00021600	0x0002167F	0x00021680	0x000216FF
23	0x00021700	0x0002177F	0x00021780	0x000217FF
24	0x00021800	0x0002187F	0x00021880	0x000218FF
25	0x00021900	0x0002197F	0x00021980	0x000219FF
26	0x00021A00	0x00021A7F	0x00021A80	0x00021AFF
27	0x00021B00	0x00021B7F	0x00021B80	0x00021BFF
28	0x00021C00	0x00021C7F	0x00021C80	0x00021CFF
29	0x00021D00	0x00021D7F	0x00021D80	0x00021DFF
30	0x00021E00	0x00021E7F	0x00021E80	0x00021EFF
31	0x00021F00	0x00021F7F	0x00021F80	0x00021FFF

Fonte: Autor.

3.2.3 Implementação dos blocos internos ao IHM

Para facilitar o desenvolvimento e entendimento do IHM, ele foi dividido em três blocos principais, como descrito no início da seção 3.2: Escalonador (ESC), Gerenciador de Contexto (GC) e Temporização e Controle (T&C). Além desses três, um quarto bloco, chamado Controlador de *Reset*, foi utilizado. Por se tratar de um bloco mais simples e de propósito geral, ele será o primeiro a ser apresentado.

O hardware do IHM, do ponto de vista construtivo, é formado por uma coleção de arquivos que foram escritos em linguagem de descrição de hardware, chamados de entidades, que descrevem partes específicas de seu funcionamento. A Figura 25 apresenta todo conjunto de entidades que o compõe.

Figura 25 – Hierarquia de entidades declaradas em linguagem VHDL que compõe o hardware do IHM.

Fonte: Autor.

Perceba na Figura 25 que não há uma entidade específica para descrever o bloco de Temporização e Controle (T&C). Na verdade, esse bloco não é representado por um arquivo separado que juntamente com outros compõem o IHM, ele foi desenvolvido no arquivo principal (arquivo *microkernel.vhd*) e é complementado por oito instâncias do bloco Contador Decrescente (arquivo *counter_down.vhd*). A razão para isso é seu papel de integração entre todos os blocos do IHM e a CPU Plasma modificada.

Os subitens dessa sessão tratam de cada um dos arquivos componentes do *microkernel*, porém o tratamento será feito de acordo com a divisão dos blocos funcionais previamente apresentada.

3.2.3.1 Controlador de reset

O Controlador de *Reset* (descrito no arquivo fonte *reset_controller.vhd*) é instanciado no *microkernel* com a identificação “u1_reset_controller”. Ele é composto por apenas dois sinais de entrada (*clk* e *reset_in*) e um sinal de saída (*reset_out*), conforme apresentado na Figura 26.

Seu funcionamento é semelhante ao de um circuito multivibrador monoestável. Sempre que a entrada de *reset* é acionada, a saída se mantém ativada até 16 ciclos de *clock* após a

remoção do sinal de entrada. Essa é uma medida cautelar para evitar falhas durante a energização do processador e efeitos mecânicos conhecidos como trepidação de contato (11) ao pressionar a tecla de *reset* externa. Esse sinal *reset_out* funciona como *reset* de todo o *microkernel* por hardware.

Figura 26 – Bloco Controlador de *Reset* (obtido da ferramenta RTL Viewer do Quartus II).

Fonte: Autor.

3.2.3.2 Escalonador

O bloco escalonador (ESC) é responsável pela análise de todas as tarefas criadas no programa em execução e pela definição de qual delas é a mais apta a ser executada ou ter sua execução retomada após o próximo *tick interrupt* ou término de um *time slice*.

Para isso, combinamos o princípio de funcionamento do algoritmo Round-Robin, que define que as tarefas são executadas até o seu fim ou até que o tempo reservado para a sua execução se esgote (*time slice*), com preempção baseada na prioridade das tarefas e no tempo que elas estão aguardando para serem executadas. Como proposto, o *microkernel* por hardware funciona em paralelo à execução do programa (tarefa). Logo o bloco ESC também funciona em paralelo à CPU realizando ciclos de verificação sucessivos das prioridades e estados das tarefas.

O ESC é subordinado ao bloco T&C, responsável por decidir quando ocorrerá a troca de tarefa e como isso será feito. Assim, sua operação se restringe a, mediante as informações que possui (lista de tarefas ativas, estado das tarefas, prioridades e sinalização de fim de *time slice*), procurar qual tarefa é a candidata mais apta a ser a próxima executada. Como todo o sistema é síncrono, sua saída (número da próxima tarefa) é atualizada a cada ciclo de *clock*.

A Figura 27 representa o diagrama simplificado do bloco escalonador com suas entradas e saída.

Figura 27 – Diagrama simplificado de entradas e saída do bloco escalonador do IHM.

Fonte: Autor.

Com base nas informações de *tick* (final de *time slice*), do registrador de tarefas ativas (*task_live*, aquelas que fazem parte do programa em execução) e na última atualização do estado das tarefas (*running*, *ready*, *blocked* ou *suspended*, conforme a seção 2.4.1), o ESC controla o tempo (medido em *ticks*) que uma tarefa em estado *ready* está aguardando para ser executada. Essa informação é primordial para o desempate entre duas ou mais tarefas que disputam por uma chance para serem executadas e possuem a mesma prioridade.

De posse de todas essas informações, o escalonador atribui uma nota para cada tarefa. Essa nota é composta binariamente pela associação de bits que representam a prioridade da tarefa, tempo de espera (*wait*) e o seu estado. Somente as tarefas com estado *running* ou *ready* tem suas notas avaliadas. Aquela que possuir a maior nota é a escolhida como próxima tarefa. Esse processo de avaliação se repete continuamente, cabe ao bloco T&C escolher o momento que utilizará a informação gerada pelo ESC.

O código do escalonador foi escrito no arquivo fonte *scheduler.vhd* e foi adicionado ao projeto através da instância *u3_scheduler*. A Figura 28 representa em detalhes todos os sinais correspondentes à entrada do bloco (à esquerda) e sua saída (à direita).

Figura 28 – Diagrama de entradas e saída do bloco escalonador do IHM (obtido da ferramenta RTL Viewer do Quartus II).

Fonte: Autor.

Além dos sinais *clk* (*clock* da CPU), *reset* e dos sinais provenientes dos registradores *task_live* e *task_priX_X*, descritos na seção 3.2.2. Para definição do sinal de saída *task_next*, correspondente ao número da próxima tarefa, o ESC se baseia em registradores internos do bloco T&C e em um sinal especial deste mesmo bloco para aplicação de seu algoritmo de escalonamento. Vamos a descrição dos sinais:

- *task_state7_0*: registrador que armazena o estado atual das tarefas de 7 a 0. São dois bits para cada tarefa;
- *task_state15_8*: semelhante ao anterior, porém destinado às tarefas de 15 a 8;
- *task_state23_16*: semelhante ao anterior, porém destinado às tarefas de 23 a 16;
- *task_state31_24*: semelhante ao anterior, porém destinado às tarefas de 31 a 24;
- *wait_flag*: esse sinal é disparado cerca de 9 ciclos de *clock* antes do início da restauração de uma tarefa e é utilizado pelo escalonador para controlar a quantidade de *time slices* que uma tarefa em estado *ready* está aguardando para ser executada.

A Tabela 8 representa o código binário utilizado para cada um dos possíveis estados de uma tarefa. Esse fluxo está ilustrado na Figura 9 na seção 2.4.1.

Tabela 8 – Código de possíveis estados de uma tarefa.

Estado	Código	
Suspensa (<i>suspended</i>)	0	0
Bloqueada (<i>blocked</i>)	0	1
Pronta (<i>ready</i>)	1	0
Rodando (<i>running</i>)	1	1

Fonte: Autor.

A Figura 29 ilustra um exemplo de distribuição dos estados das 8 primeiras tarefas no registrador `task_state7_0`.

Figura 29 – Exemplo de distribuição de estados das tarefas de 0 a 7 no registrador `task_state7_0`.

Tarefa	7		6		5		4		3		2		1		0	
Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Estado	0	0	0	0	0	1	0	1	0	1	1	0	1	0	1	1

Fonte: Autor.

Segundo o padrão adotado para identificação dos estados, no exemplo da Figura 29, a tarefa 0 está rodando enquanto as tarefas 1 e 2 estão prontas, as tarefas 3, 4 e 5 estão bloqueadas e as tarefas 6 e 7 estão suspensas. Isso quer dizer que, apenas as 3 primeiras tarefas estão aptas para o escalonamento.

O *modus operandi* do ESC é bastante prático e eficiente. Para melhor entendimento vamos separá-lo em três partes: (i) controle do tempo de espera; (ii) montagem dos vetores de tarefas e; (iii) árvore binária de seleção.

I – Controle do tempo de espera

O bloco ESC possui um conjunto com 32 contadores crescentes de 12 bits cada um, chamados de `task_waits`. Esses contadores armazenam a quantidade de *time slices* que uma tarefa em estado *ready* está esperando para ser executada. A cada pulso do sinal `wait_flag`, todas as tarefas nesta condição têm seu contador incrementado em uma unidade. Qualquer outro estado de uma tarefa, força o *reset* do seu respectivo contador.

II – Montagem dos vetores de tarefas

Para cada uma das tarefas é montado um vetor com 21 bits, composto pelo número da tarefa (5 bits), prioridade da tarefa (3 bits), tempo de espera (12 bits do contador) e o

complemento do bit menos significativo do estado da tarefa (1 bit). A Figura 30 ilustra um exemplo de montagem dos vetores de algumas tarefas.

Figura 30 – Exemplo com três vetores de tarefas com mesma prioridade utilizados pelo bloco escalonador para definição da próxima tarefa a ser executada.

Campo	Tarefa					Prio.			Tempo de espera													E.
Bit	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
Vetor	0	1	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	1	0	1	
Valor	8					3																1
Magnitude						24.581																
																						Pronta

Campo	Tarefa					Prio.			Tempo de espera													E.
Bit	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
Vetor	0	0	1	1	1	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	1	
Valor	7					3																1
Magnitude						24.577																
																						Pronta

Campo	Tarefa					Prio.			Tempo de espera													E.
Bit	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
Vetor	0	0	0	1	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	
Valor	2					3																0
Magnitude						24.576																
																						Rodando

Fonte: Autor.

III – Árvore binária de seleção

A árvore binária para seleção de tarefa foi inspirada no artigo de GARCIA et al. (1999) (16). Ela compara os vetores de tarefas dois a dois, de seus galhos mais afastados até chegar ao tronco principal da árvore. O vetor com maior magnitude segue para próxima comparação. Por fim, aquele que chegar até a base da árvore portará o número identificador da próxima tarefa a ser executada. Em caso de empate nas magnitudes, o vetor mais à direita será o vencedor.

Considerando os exemplos da Figura 30, se as três tarefas (8, 7 e 2) estivessem disputando para serem a executada no próximo *time slice*, a tarefa 8 seria a escolhida por apresentar a maior magnitude. Perceba que, como todas possuem a mesma prioridade, o tempo de espera maior da tarefa 8 foi determinante para sua escolha.

Agora, se considerarmos que apenas as tarefas 7 e 2 estivessem na disputa. Como as duas possuem a mesma prioridade e ambas estão com o tempo de espera zerado, seria o valor do complemento do bit menos significativo do estado da tarefa que definiria o desempate. A tarefa em execução (neste caso a número 2) sempre está com esse bit zerado, ao passo que, a tarefa em estado pronto (neste caso a número 7) tem esse bit igual a um, garantindo que a

3.2.3.3 Gerenciador de contexto

O bloco GC tem função muito mais simples que a do bloco ESC, seu papel é realizar o *backup* do contexto da tarefa que saiu de execução e realizar a restauração do contexto da tarefa que deve entrar em execução no próximo *time slice*. Sendo subordinado ao bloco T&C, seu funcionamento (*backup* ou *restore*) é disparado por sinais provenientes dele.

Além desses sinais, o GC também está diretamente conectado a um banco interno de registradores da CPU (aquele que não estiver sendo usado para executar a tarefa atual) e a região de memória RAM reservada para alocação das TCBs.

As operações de *backup* e *restore* nunca ocorrem simultaneamente. Porém, existem casos que uma troca de tarefa pode ser antecipada por solicitações de bloqueio feitas pela própria tarefa em execução. O bloqueio pode durar até o final do *time slice* (*yield*) ou por mais de um *time slice* (*sleep*). Nesses casos, quando o *backup* ainda não está concluído, a solicitação de restauração é postergada pelo bloco T&C. Cabe ao GC apenas atender as solicitações e sinalizar sua conclusão.

A Figura 32 representa os dois fluxos possíveis de operação do GC (*backup* e *restore*).

Figura 32 – Exemplo do fluxo de dados de *backup* da última tarefa interrompida (a) e de restauração da próxima tarefa a ser executada (b).

O código do GC foi escrito no arquivo fonte `context_manager.vhd` e foi adicionado ao projeto através da instância `u2_context_manager`. A Figura 33 representa em detalhes todos os sinais correspondentes à entrada do bloco (à esquerda) e suas saídas (à direita), nele a maior parte dos sinais estão relacionados ao acesso à memória RAM reservada para armazenamento dos TCBs e ao acesso ao banco de registradores conectado ao *microkernel* por hardware. Esses sinais serão descritos na seção 3.2.4 que trata da interface entre o *microkernel* por hardware o microcontrolador Plasma.

Figura 33 – Diagrama de entradas e saídas do bloco gerenciador de contexto do IHM (obtido da ferramenta RTL Viewer do Quartus II).

Fonte: Autor.

Sinais trocados entre o GC e o bloco de controle T&C do *microkernel* por hardware:

- `backup_init`: entrada de pedido de *backup* de contexto;
- `backup_ready`: saída de sinalização de conclusão de *backup* de contexto;
- `task_antiga`: informa ao bloco qual é o número da tarefa interrompida;
- `restore_init`: entrada de pedido de *restore* de contexto;
- `restore_ready`: saída de sinalização de conclusão de *restore* de contexto;
- `task_futura`: informa ao bloco qual é o número da próxima tarefa a ser executada.

Todos esses sinais estão relacionados com o contexto das tarefas, que é formado pelo valor de 31 dos 32 registradores internos da CPU Plasma e o valor do PC da próxima tarefa a ser executada (descrito na seção 3.2.2).

O GC é formado praticamente por duas máquinas de estados finitos (FSM). A primeira, chamada de *state_backup*, é responsável pelo *backup* da tarefa que foi interrompida, e a segunda, chamada de *state_restore*, é responsável pela restauração da tarefa que entrará em execução.

A FSM *state_restore* possui 68 estados, e gasta um ciclo de *clock* por estado, totalizando 67 ciclos para avançar do estado S0 (quando o sinal *restore_init* é detectado) até alcançar o estado S67 (quando o sinal *restore_ready* é ativado). São gastos 2 ciclos de *clock* para a restauração de cada registrador ou PC (um para leitura da TCB e outro para gravação do registrador), totalizando 64 ciclos, os outros 3 são para sincronismo.

A FSM *state_backup* possui 37 estados, e também gasta um ciclo de *clock* por estado, totalizando 36 ciclos para avançar do estado S0 (quando o sinal *backup_init* é detectado) até alcançar o estado S36 (quando o sinal *backup_ready* é ativado). Neste caso, é gasto apenas 1 ciclo de *clock* para o *backup* de cada registrador ou PC (lê o registrador na borda de descida do *clock* e grava no TCB na borda de subida), totalizando 32 ciclos, os outros 4 são para sincronismo.

Tanto o processo de *backup*, como o de restauração são executados apenas enquanto os sinais de pedido *backup_init* ou *restore_init* estiverem presentes. Em caso de inibição desses sinais, as máquinas de estados voltam imediatamente para o estado inicial (S0 – aguarda pedido). Além disso, conceitualmente um *backup* não pode ocorrer simultaneamente a uma restauração de contexto. Quem faz o controle para que isso não ocorra é o bloco T&C.

A Figura 34 ilustra o fluxo simplificado de troca de sinais entre o GC e o T&C. Na coluna à esquerda estão as ações do bloco temporização e controle e à direita estão as ações do bloco gerenciador de contexto. O número ‘3’ estilizado, sob o fluxograma, destaca o principal fluxo de sinais entre as duas partes, que é iniciado sempre através de solicitações do bloco T&C, seja para *restore* ou *backup*. O gerenciador de contexto apenas executa ordens. A relação entre os dois blocos pode ser definida como mestre-escravo.

Figura 34 - Exemplo do fluxo de dados entre os blocos de temporização e controle (à esquerda) e o gerenciador de contexto (à direita) para realização de uma troca de contexto.

Fonte: Autor.

3.2.3.4 Temporização e controle

O bloco T&C faz a interligação entre os demais blocos do IHM com a CPU e a memória RAM reservada para as TCBs. Seu papel é controlar o funcionamento do *microkernel* por hardware, gerar a base de tempo do *time slice* e definir o estado atual de cada uma das tarefas ativas.

Este bloco é habilitado através do bit 0 do registrador `reg_sch`. Quando habilitado, o controle do fluxo de execução do programa passa da CPU para o IHM e a faixa de memória RAM reversível, reservada para os TCBs, fica acessível apenas por ele.

O bloco T&C é quem define por quanto tempo cada tarefa será executada. Para isso, usa como base o período de *time slice* salvo dos 30 bits menos significativos do registrador *tick*.

Fator importante, a ser destacado, é que as interrupções de hardware continuam a ser atendidas pela CPU e têm prioridade superior as do *microkernel* por hardware, isso quer dizer que, nenhuma troca de tarefas ocorre enquanto uma interrupção estiver em tratamento.

Este bloco é baseado em três máquinas de estados finitos (FSM):

- 1- Motor: esta é a máquina de estados mais importante para o funcionamento do *microkernel* por hardware, ela define seu *modus operandi*. Tem como entradas sinais relativos à habilitação do escalonamento de tarefas, sinalização de fim de *time slice* (*tick*), troca de contexto síncrona e assíncrona, confirmações de conclusão de *backup* e restauração de tarefa. Já suas saídas geram o pedido de troca de tarefa para CPU, de restauração de contexto, *backup* de contexto e troca de bancos de registradores internos entre a CPU e o *microkernel* por hardware.
- 2- PC_Backup: cuida para que o valor do PC seja salvo corretamente pelo bloco Gerenciador de Contexto mesmo quando trocas de tarefas assíncronas ocorrem.
- 3- Task_State: controla o estado atual de cada tarefa, faz isso com base na configuração do registrador responsável por definir quais tarefas estão ativas no programa em execução (*task_live* – um bit por tarefa), e nos sinais internos que representam a tarefa atual, futura e interrupção temporal de uma ou mais tarefa (*sleep* ou *yield*).

Além das estruturas descritas, o bloco T&C ainda conta com 8 contadores decrescentes, que funcionam atrelados às tarefas de 1 a 8, cujo papel é sinalizar se a tarefa já pode ser desbloqueada após um pedido de bloqueio temporário (*sleep* ou *yield*). Comandos de bloqueio temporário se sobrepõe ao *time slice*, eles implicam em imediata interrupção da CPU e início imediato da restauração do contexto da próxima tarefa a ser executada. Somente após isso, ocorre a troca das tarefas (*task switching*). Esse fluxo de ações faz parte do controle efetuado pela FSM Motor.

O código do bloco de temporização e controle, como explicado no início da seção 3.2.3, foi desenvolvido no arquivo principal do *microkernel* (arquivo *microkernel.vhd*) e é complementado por oito instancias do bloco contador decrescente (arquivo *counter_down.vhd*) usado para bloqueio temporário de tarefas.

As instâncias criadas receberam as seguintes identificações:

- u4_counter_down_1;
- u5_counter_down_2;
- u6_counter_down_3;

- u7_counter_down_4;
- u8_counter_down_5;
- u9_counter_down_6;
- u10_counter_down_7;
- u11_counter_down_8.

Cada um deles funciona atrelado à uma tarefa, presente no sufixo de seu nome (de 1 a 8) e podem partir suas contagens com valores de até 16 bits. A Figura 35 exemplifica os três primeiros contadores.

Figura 35 – Três primeiros blocos contadores decrescentes para temporização do período de bloqueio das tarefas periódicas 1, 2 e 3 (obtido da ferramenta RTL Viewer do Quartus II).

Fonte: Autor.

Segue a descrição de cada um dos sinais de entrada e saída:

- clk: sinal responsável pelo decremento do contador, é conectado ao contador que controla o período do *time slice*, a cada *tick* é gerado um pulso;
- load: pulso de carga do valor de partida do contador, somente é ativado se a tarefa relacionada ao contador estiver ativa. Conectado indiretamente ao bit 16 do registrador *task_sleep*;
- reset: *reseta* o contador. Conectado ao registrador *task_live*. Enquanto a tarefa associada ao contador está desabilitada, o contador se mantém *resetado*;

- *sleep*: valor de carga do contador. Conectado indiretamente aos 16 bits menos significativos do registrador *task_sleep*;
- *cnt_out*: valor atual do contador. Não é utilizado na lógica;
- *done*: sinalização de fim da contagem. É utilizado pelo bloco de temporização e controle para decidir se a tarefa em execução precisa ser bloqueada.

A tarefa 0 e as tarefas na faixa de 9 a 31 não dispõem do recurso de *sleep* ou *yield*. Em caso de geração de pedido de *sleep* (pulso no bit 16 do registrador *task_sleep*) durante a execução de algumas dessas tarefas, o pedido é simplesmente ignorado e a tarefa se mantém em execução.

Além dos contadores exclusivos para tarefas de 1 a 8, o bloco T&C possui um contador crescente com rearme automático para geração do período do *time slice* (configurado no registrador identificado como *tick*). Seu sinal de *clock* é o mesmo da CPU e o funcionamento está condicionado à ausência de pedido de *reset* pela CPU e a habilitação do escalonamento de tarefas.

Três sinais de sincronismo são disparados por esse contador em instantes diferentes, são eles:

- *wait_flag*: avisa ao ESC que as tarefas em estado pronto (*ready*) completaram mais um *time slice* em espera. É disparado 78 pulsos de *clock* antes do fim do *time slice*;
- *restore_flag*: avisa ao T&C que é hora de solicitar ao GC que inicie a restauração do contexto da próxima tarefa. É disparado 69 pulsos de *clock* antes do fim do *time slice*;
- *tick_flag*: avisa ao T&C que o *time slice* terminou e que é hora de fazer a troca de tarefas. É disparado assim que o contador alcança seu valor máximo.

Cada um desses sinais, também chamados de *flags*, fica ativo por um ciclo de *clock*, tempo suficiente para sua detecção pelos circuitos adjacentes. A Figura 36 ilustra o contador (a) e seus sinais de sincronismo em um diagrama de tempo (b).

Figura 36 – Diagrama simplificado de entradas e saídas do contador de *time slices* (a) e o diagrama de tempo com suas *flags* de sinalização (b).

Fonte: Autor.

O diagrama de tempo apresentado na Figura 36 (b), nos permite extrair o período mínimo teórico para o *time slice*. São necessários exatamente 78 ciclos de *clock* para que o *microkernel* por hardware gere os sinais de sincronismo e efetue a troca de tarefas.

As três máquinas de estados finitos, previamente descritas, são detalhadas a seguir.

I – FSM Motor

Essa é a máquina de estados mais importante para operação do IHM, ela gera os pedidos de interrupção da CPU, troca de bancos, restauração e *backup* de contexto.

Sinais de entrada:

- *sleep_yield*: sinalização de pedido assíncrono para troca imediata de tarefa. Gerado quando o contador de *sleep* ou *yield* da tarefa em curso não está zerado, ou seja, tem seu sinal de saída 'done' igual a zero;
- *tick_enable*: *flag* que sinaliza que a próxima tarefa a ser executada (indicada pelo ESC) é diferente da atual. A ausência desse sinal inibi todo procedimento para troca de tarefa, quando a tarefa em curso deve permanecer em curso, tornando o escalonamento mais eficiente;
- *restore_flag*: *flag* de aviso de que é hora de solicitar ao GC que inicie a restauração do contexto da próxima tarefa. gerado pelo temporizador descrito anteriormente;
- *restore_ready*: sinalização do GC que o processo de restauração está concluído;

- `tick_flag`: *flag* de aviso de término do *time slice* e de que é hora de fazer a troca de tarefas. Gerado pelo temporizador descrito anteriormente;
- `switch_out_RT`: sinalização da CPU de que ela está em estado *stopped* (nenhuma instrução em execução);
- `backup_ready`: sinalização do GC que o processo de *backup* está concluído.

Sinais de saída:

- `restore_init`: pedido de restauração de contexto enviado para o GC;
- `task_switch_RT`: pedido de paralização da execução de instruções enviado para CPU;
- `bank_switch`: dispara a ação de inverter os bancos de registradores;
- `backup_init`: pedido de *backup* de contexto enviado para o GC;
- `atualiza_task_ativa`: pedido para que a tarefa ativa seja atualizada com o valor da próxima tarefa e que seu valor seja salvo como tarefa antiga. Usado para controle de troca de tarefas.

A FSM Motor possui 10 estados possíveis (de `s0` a `s9`), os quais quando percorridos são capazes de controlar tanto, trocas síncronas de tarefas (baseadas em *time slice*), quanto trocas assíncronas (baseados em pedidos de *sleep* ou *yield*). A Figura 37 ilustra essa máquina de estados, onde os arcos destacados com asterisco (*) indicam o fluxo de estados para trocas síncronas de tarefas.

Figura 37 – FSM Motor (obtido da ferramenta State Machine Viewer do Quartus II).

Fonte: Autor.

A Tabela 9 e a Tabela 10 apresentam respectivamente as condições necessárias para que ocorram as transições de estados e quais sinais são ativados em cada um dos estados.

Tabela 9 – Mapa de transição de estados da FSM Motor.

Estado		Condição das entradas para que ocorra a transição
Atual	Futuro	
s0	s1	(restore_flag) . (tick_enable)
s0	s6	(sleep_yield)
s1	s2	(restore_ready)
s1	s8	(sleep_yield)
s2	s3	(sleep_yield) + (tick_flag)
s3	s4	(switch_out_RT)
s4	s5	(!switch_out_RT)
s5	s0	(backup_ready)
s5	s9	(sleep_yield)
s6	s7	(tick_enable)
s7	s3	(restore_ready)
s8	s3	(restore_ready)
s9	s6	(backup_ready)

Fonte: Autor.

O ponto de exclamação ‘!’ presente na transição s4 para s5 da Tabela 9 representa a operação lógica NOT. Os estados com outros dois ou mais adjacentes foram representados em ordem de prioridade das adjacências, da maior para a menor.

Tabela 10 – Saídas ativadas em função do estado ativo da FSM Motor.

Estado	Sinais ativos nas saídas
s0	-
s1	restore_init
s2	-
s3	task_switch_RT e atualiza_task_ativa
s4	bank_switch
s5	backup_init
s6	task_switch_RT
s7	task_switch_RT e restore_init
s8	task_switch_RT e restore_init
s9	task_switch_RT e backup_init

Fonte: Autor.

Como é possível notar na Figura 37, se o *microkernel* por hardware proposto trabalhasse apenas com tarefas síncronas, a FSM Motor teria apenas 6 estados e somente forçaria a interrupção da CPU quando todo o contexto estivesse pronto para a inversão dos bancos de registradores (IRBs), consequentemente troca da tarefa em execução. Nesta condição, uma troca de tarefas do ponto de vista da CPU, ocorre em no mínimo 2 ciclos de *clock*. Para que isso seja possível, no instante em que a pausa da CPU é solicitada, a próxima instrução a ser executada não pode ser uma instrução de desvio e a instrução atual não pode ser do tipo multiplicação ou divisão (detalhado na seção 3.2.4.1). Qualquer outra condição implicará em atraso no processo de troca de tarefas.

Os pedidos assíncronos para troca de tarefas (*sleep* ou *yield*) podem ocorrer, a qualquer momento durante um *time slice*, porém são divididos em quatro casos, de acordo com o estado da FSM:

- 1- Estado s0: fração do *time slice* que antecede o início da restauração de contexto. Neste caso a CPU é interrompida e o processo de troca de tarefa é antecipado. A restauração do contexto ocorre simultaneamente ao pedido de pausa da CPU, que só voltará a processar quando a troca de tarefas estiver concluída. Neste caso a interrupção da CPU dura 69 ciclos de *clock* (67 para restauração de contexto e 2 para troca dos bancos de registradores).
- 2- Estado s1: o processo de restauração de contexto já foi iniciado (vide Figura 42 (B) na seção 3.2.4). O procedimento adotado é semelhante ao do caso anterior, porém o tempo máximo de interrupção da CPU pode variar de 69 ciclos de *clock* a apenas 3. Isso dependerá de quanto contexto já foi restaurado quando o pedido de troca foi recebido.
- 3- Estado s2: a restauração de contexto acabou de ser concluída. Este caso tem comportamento semelhante ao da troca síncrona e pode durar no mínimo 2 ciclos de *clock*.
- 4- Estado s5: durante o *backup* de contexto da tarefa que foi interrompida. Esse é o caso mais crítico para disparo de uma nova troca de tarefas. Nele a restauração do contexto não pode ser iniciada enquanto o *backup* não for concluído. A CPU ficará pausada até que a troca de tarefas seja concluída. Esse processo pode levar de 106 ciclos de *clock*, se ocorrer no instante em que o *backup* foi iniciado, até 71 ciclos, se ocorrer há um ciclo de *clock* do seu fim.

A Figura 38 ilustra ao longo de um *time slice*, as regiões de possível ocorrência de um pedido de troca de tarefa assíncrona de acordo com os casos analisados.

Figura 38 – Identificação das regiões de possível pedido de troca assíncrona de tarefas em um *time slice*.

Fonte: Autor.

Esse estudo de casos permite afirmar que o microcontrolador Plasma modificado integrado ao *microkernel* por hardware proposto (IHM-Plasma) pode ter sua CPU interrompida por períodos de 2 a 106 ciclos de *clock* a cada troca de tarefas. Considerando que a frequência de operação do processador foi ajustada para 25 MHz, esses tempos equivalem a uma faixa de 80 ns a 4,24 μ s.

Essas constatações, permitem extrair um outro tempo mínimo para um *time slice*. Considerando o pior tempo do quarto caso apresentado (106 ciclos de *clock*) e supondo que assim que a CPU volta a processar instruções, a *flag* de início de restauração de contexto (*restore_flag*) seja ativada. São necessários mais 69 ciclos de *clock* até a ativação de *flag* de troca de tarefa (*tick_flag*) que marca o fim do *time slice*. A soma dos dois valores é igual 175 ciclos de *clock*, esse é o *time slice* mínimo teórico aplicável ao IHM-Plasma. Considerando a frequência do *clock* de 25 MHz, esse tempo é de exatamente 7 μ s, que implica em uma frequência máxima de troca de tarefas de 142.857 Hz. Quando comparado com as recomendações feitas pelo desenvolvedor do sistema operacional de tempo real FreeRTOS (2;42), onde o *time slice* mínimo deve ser na ordem de 1 ms (frequência máxima de 1 kHz). Concluimos que a solução proposta consegue operar até três ordens de grandeza abaixo dessa recomendação.

II – FSM PC_Backup

A máquina de estados PC_Backup cuida exclusivamente de salvar o valor correto do *Program Counter*, para que seja feita sua cópia no TCB durante o processo de *backup* de contexto.

Sinais de entrada:

- backup_init: pedido de *backup* de contexto enviado pela FSM Motor ao GC;
- switch_out_RT: sinalização da CPU que ela está em estado *stopped* (nenhuma instrução em execução).

Sinais de entrada:

- backup de 1º nível: dispara pedido de cópia temporária do valor da próxima instrução a ser executada, no registrador pc_backup_RT, com base no sinal enviado pela CPU (descrito na seção 3.2.4);
- backup de 2º nível: dispara pedido de cópia do sinal temporário, salvo no registrador pc_backup_RT, no registrador de saída pc_backup, que é uma entrada do bloco GC.

A FSM PC_Backup possui 5 estados possíveis (de s0 a s4), os quais quando percorridos permitem que o valor correto do PC seja disponibilizado ao bloco GC, que fará o *backup* na TCB correspondente a tarefa interrompida durante uma troca de tarefas. A Figura 39 ilustra essa máquina de estados, onde os arcos destacados com asterisco (*) indicam o fluxo de estados para trocas síncronas de tarefas.

Figura 39 – FSM PC_Backup (obtido da ferramenta State Machine Viewer do Quartus II).

Fonte: Autor.

A Tabela 11 e a Tabela 12 apresentam respectivamente as condições necessárias para que ocorra a transição de estados e quais sinais são ativados em cada um dos estados.

Tabela 11 – Mapa de transição de estados da FSM PC_Backup.

Estado		Condição das entradas para que ocorra a transição
Atual	Futuro	
s0	s1	(switch_out_RT)
s1	s2	(!switch_out_RT) . (backup_init)
s2	s0	(!backup_init)
s2	s3	(backup_init) . (switch_out_RT)
s3	s4	(!backup_init)
s4	s2	(!switch_out_RT) . (backup_init)

Fonte: Autor.

Tabela 12 – Saídas ativas em função do estado ativo da FSM PC_Backup.

Estado	Sinais ativos nas saídas
s0	-
s1	backup de 1º nível
s2	backup de 2º nível
s3	backup de 1º nível
s4	-

Fonte: Autor.

A estratégia, de usar dois níveis para o *backup* do PC, foi adotada para compatibilizar essa operação com as trocas assíncronas de tarefas, em que o processo de *backup* de uma tarefa ainda não foi concluído e a CPU já precisou ser interrompida para uma nova troca de tarefas, como descrito no quarto caso estudado na FSM Motor.

Com isso, o valor de PC da primeira tarefa interrompida é preservado na entrada do GC até o fim do primeiro processo de *backup*, enquanto o PC da segunda tarefa está salvo em um registrador temporário. Esse comportamento é semelhante ao de uma memória FIFO (*first-in, first-out*) de dois níveis, onde a carga é controlada pelo aviso de CPU *stopped* e a descarga é controlada pelo aviso de início de um novo *backup* de tarefa.

III – FSM Task_State

A máquina de estados Task_State cuida da atualização dos registradores que armazenam o estado atual das 32 possíveis tarefas.

Sinais de entrada:

- bank_switch: pedido da FSM Motor para que os bancos de registradores sejam invertidos;
- escalonador: registrador que sinaliza se o escalonamento está ativo (reg_sch).

Sinais de entrada:

- tarefas habilitadas em estado *ready*: dispara pedido para que todas as tarefas habilitadas sejam colocadas em estado *ready*;
- ativa = *running*, antiga = *ready*: dispara pedido para atualização do estado da tarefa que entrou e execução e da que foi interrompida;
- atualiza estados: dispara pedido para atualização do estado de todas as tarefas.

A FSM Task_State possui 4 estados possíveis (de s0 a s3), os quais quando percorridos permitem que os estados atuais de todas as tarefas estejam sempre atualizados de forma a garantir a perfeita análise pelo bloco ESC. A Figura 40 ilustra essa máquina de estados, onde os arcos destacados com asterisco (*) indicam o fluxo de estados após a primeira troca de bancos de registradores e enquanto o escalonamento está ativo.

Figura 40 – MSFM Task_State (obtido da ferramenta State Machine Viewer do Quartus II).

Fonte: Autor.

A Tabela 13 e a Tabela 14 representam respectivamente as condições necessárias para que ocorram as transições de estados e quais sinais são ativados em cada um dos estados.

Tabela 13 – Mapa de transição de estados da FSM Task_State.

Estado		Condição das entradas para que ocorra a transição
Atual	Futuro	
s0	s1	(escalonador) . (!bank_switch)
s1	s2	(escalonador) . (bank_switch)
s1	s0	(!escalonador)
s2	s3	(escalonador) . (!bank_switch)
s2	s0	(!escalonador)
s3	s2	(escalonador).(bank_switch)
s3	s0	(!escalonador)

Fonte: Autor.

Tabela 14 – Saídas ativas em função do estado ativo da FSM Task_State.

Estado	Sinais ativos nas saídas
s0	-
s1	tarefas habilitadas em estado ready
s2	ativa = running, antiga = ready
s3	atualiza estados

Fonte: Autor.

Assim que o escalonamento é habilitado (s1), todas as tarefas habilitadas (registrador *task_live*) são colocadas em estado *ready*. Quando ocorre uma troca de bancos (s2) o estado da tarefa interrompida passa para *ready* e da tarefa que passou para execução recebe estado *running*. O restante do tempo (s4), as tarefas têm seus estados constantemente atualizados com base no registrador *task_live* e nas saídas ‘done’ dos contadores decrescentes que controlam o tempo de bloqueio das tarefas de 1 a 8 por pedido de *sleep* ou *yield*. Esse fluxo de atualização é ilustrado na Figura 41.

Figura 41 – Exemplo do fluxo de dados realizado pela FSM Task_State para a atualização dos estados das tarefas gerenciadas pelo IHM.

Fonte: Autor.

O fluxo apresentado na Figura 41 se repete a cada ciclo de *clock* para todas as tarefas, enquanto que o estado s3 se mantiver ativo na FSM Task_State.

3.2.4 Interface entre o IHM e o microcontrolador Plasma modificado

Esta seção é iniciada com a descrição de cada uma das quatro principais ações realizadas pelo IHM, de forma a permitir a rápida troca de tarefas:

- A. Análise e seleção da próxima tarefa. O ESC analisa qual a melhor candidata a ser a próxima tarefa em execução com base em sua prioridade, estado, e tempo de espera. Esse processo é contínuo e cabe ao T&C determinar quando a tarefa selecionada será utilizada como a próxima a ser executada;
- B. Restauração de contexto. Ela é executada pelo GC assim que o módulo de T&C sinaliza que foi alcançado o tempo limite para seu início (pouco antes do fim do *time slice*). O GC copia o contexto da próxima tarefa do TCB correspondente para o IRB disponível e para uma entrada do PC;
- C. Troca de tarefas (*task switching*). Ela é representada pela troca dos IRBs entre a CPU e *microkernel* por hardware;
- D. Salvamento do contexto. Ela é executada pelo GC assim que o módulo de T&C sinaliza que foi concluída a troca dos IRBs. O GC copia o contexto da última tarefa executada, salvo em um registrador auxiliar no *microkernel* por hardware (valor do PC) e no IRB disponível, para o TCB correspondente;

Esse processo é repetido continuamente para cada troca de tarefas executada enquanto o escalonamento estiver habilitado.

A Figura 42 representa o funcionamento do fluxo proposto em um diagrama de tempo, considerando-se um sistema fictício com duas tarefas, onde é possível visualizar as duas primeiras trocas de tarefas. No eixo x, é identificado cada tick ou fim de *time slice*, as letras A, B, C e D identificam cada uma das ações do IHM descritas anteriormente.

No exemplo apresentado na Figura 42, ocorrem duas trocas entre as tarefas 0 e 1. Enquanto as tarefas são executadas pela CPU, o ESC faz a avaliação da mais apta a ser executada. No início do primeiro *time slice* (antes do primeiro tick), o ESC começa a analisar as tarefas com base em seus estados atuais e prioridades [Figura 42 (A)]. Pouco antes do fim do *time slice* o GC prepara o novo contexto [Figura 42 (B)] no IRB não utilizado pela CPU (*waiting set*) e disponibilizou o valor do PC para atualização. Assim que o tick 1 ocorre, as tarefas são trocadas [Figura 42 (C)] e em seguida, três ações correm em paralelo, a CPU inicia

a execução da tarefa 1, o GC faz o salvamento do contexto da tarefa 0 [Figura 42 (D)], e a saída do ESC volta a ser uma opção válida como próxima tarefa. Comportamento semelhante ao descrito, ocorre para troca entre as tarefas 1 e 0.

Figura 42 – Exemplo do fluxo de execução de tarefas com detalhamento da operação do IHM (as setas indicam os períodos em que a CPU é interrompida).

Fonte: Autor.

Considerando o fluxo de ações apresentado e que este desenvolvimento foi baseado em um processador com arquitetura RISC, onde o uso *pipeline* é muito intenso (vide seção 2.1.2), observou-se a necessidade de esvaziar o *pipeline* antes de iniciar uma nova tarefa, essa condição é avaliada à medida em que as tarefas são realizadas. Na literatura, o esvaziamento do *pipeline* é descrito como uma inserção de falsas instruções chamadas de **bolhas** na entrada, até que a última instrução válida complete sua execução (21;22;40).

Após essa explicação geral, continuaremos essa seção nos aprofundando como é feita a interface entre o IHM, a CPU Plasma modificada e a memória RAM.

A Figura 43 representa o fluxograma de operação entre a CPU e o IHM. Na coluna à esquerda estão as ações da CPU e à direita estão as ações do IHM. O ‘S’, sob o fluxograma, destaca o principal fluxo de sinais entre as duas partes, inicia com pedido do IHM para que a CPU interrompa o processamento de instruções e termina com a retomada do processamento pela CPU. Esse movimento em ‘S’ se repete a cada fim de *time slice*, enquanto o escalonamento estiver habilitado, e permite a troca rápida de tarefas pelo IHM. No fluxograma também fica claro que o *backup* do contexto da tarefa interrompida (registradores e PC) é executado logo após a inversão do banco de registradores e remoção do pedido de troca pelo IHM.

Em um ciclo síncrono (baseado em *time slice*), a restauração do contexto da próxima tarefa a ser executada se inicia um pouco antes do final do *time slice* ainda com a CPU em execução, isso somente é possível graças ao segundo banco de registradores internos proposto (vide seção 3.2.1). Esse tempo é ajustado, para que seja exatamente o mesmo necessário para

conclusão da restauração de uma tarefa. Dessa forma, dois ganhos são obtidos: (i) o tempo de interrupção da execução de programa pela CPU é minimizado, podendo chegar a apenas 2 ciclos de *clock*; (ii) a validação da tarefa indicada pelo bloco escalonador é feita muito próximo do final do *time slice*, aumentando a segurança de que a tarefa escolhida é mais adequada segundo os critérios do algoritmo utilizado.

Figura 43 – Exemplo do fluxo de dados entre a CPU Plasma modificada (à esquerda) e o *microkernel* por hardware proposto (à direita) com o objetivo de permitir o escalonamento de tarefas por hardware.

Fonte: Autor.

A

Figura 44 apresenta o bloco *microkernel* (IHM) e a identificação de seus sinais de entrada à esquerda do bloco e seus sinais de saída à direita.

Figura 44 – Diagrama de entradas e saídas do IHM (obtido da ferramenta RTL Viewer do Quartus II).

Fonte: Autor.

Alguns dos sinais apresentados na

Figura 44 não fazem parte da lógica atualmente utilizada no IHM ou são utilizados apenas para depuração. São eles:

- counter;
- intr_enable_RT;
- pause_RT;
- reg_target_out_RT;
- mk_debug;
- rt_index_RT.

Descrição dos demais sinais separados por tipo de atuação ou de periférico que é acessado por eles.

Sincronismo:

- clk: *clock* do processador;
- reset_in: *reset* do processador.

Registadores:

- escalonador: bit 0 do registrador reg_sch, usado para habilitação do escalonamento de tarefas;
- task_live: registrador vetor de tarefas, cada bit identifica a existência de uma tarefa dentre as 32 possíveis;
- task_pri31_24, task_pri23_16, task_pri15_8 e task_pri7_0: registradores responsáveis por armazenar o valor da prioridade atribuída a cada uma das tarefas;
- task_sleep: registrador de tempo de *sleep* ou *yield* da tarefa atual;
- tick: registrador de tempo do *time slice* medido em ciclos de *clock*;
- task_number: registrador do número da tarefa em execução, apenas o bit correspondente ao número da tarefa assume valor 1.

Memória SRAM alocada para armazenamento dos TCBs:

- mk_data_r: entrada de dados provenientes da memória;
- mk_address: saída de endereços para memória;
- mk_byte_we: saída de controle para memória, valor 0x0 significa operação de leitura e valor 0xF operação de escrita nos 4 bytes endereçados;
- mk_data_w: saída de dados para memória.

Contador de programa (PC):

- pc_backup_RT: entrada com o endereço da próxima instrução que seria executada da tarefa que foi interrompida durante o escalonamento;
- pc_RT: saída com o endereço da próxima instrução a ser executada da tarefa que entrará em execução;

Banco de registradores interno:

- reg_source_out_RT: entrada de dados lidos do banco;
- rs_index_RT: endereço do registrador a ser lido do banco;
- reg_dest_new_RT: saída de dados a serem escritos no banco;
- rd_index_RT: endereço do registrador a ser escrito do banco;

Controle da troca de tarefa:

- task_switch_RT: sinal de saída com pedido de paralização da CPU para troca de tarefa;

- `switch_out_RT`: retorno da CPU com sinalização de seu estado (*stopped* ou *running*);
- `sel_bank_RT`: sinal de seleção usado para inversão dos bancos de registradores internos, muda de estado apenas durante a troca de tarefas.

3.2.4.1 Mecanismo de troca de tarefas implementado por hardware

Os sistemas baseados em tarefas escalonadas por um sistema operacional ou *kernel* trabalham com um conjunto de funções especiais chamadas de funções-tarefas. Toda função-tarefa é composta por ao menos dois trechos de código: 1 – *setup* da tarefa; 2 – laço infinito com ações que se repetem continuamente. Um dos maiores desafios para o sistema de gestão de execução das tarefas, neste caso o *microkernel*, é interrompe-las, quando necessário, sem comprometer a execução do código e ser capaz de retomá-las exatamente do ponto onde foram interrompidas. Ações como estas podem causar efeitos colaterais como a perda de resultados parciais ou equívocos em decisões de desvio de código (*branches*).

Isso implica que um pedido de troca de tarefa não pode ser interpretado pela CPU como apenas uma pausa na execução de instruções. Felizmente, esse mecanismo já está presente na CPU e é chamado de serviço de interrupção (*interrupt service*, IS). Sempre que ocorre um pedido de interrupção por *hardware*, este mecanismo é disparado de forma a permitir a pausa segura da CPU antes do desvio para rotina de tratamento de interrupção (*interrupt service routine*, ISR). No caso da CPU Plasma o contador de programa (PC) é desviado para o endereço 0x3C por ação do Coprocessador 0.

Toda instrução executada é analisada pelo módulo Control da CPU (arquivo fonte `control.vhd`) que sinaliza, a partir de seu *opcode* (código operacional), qual deve ser a origem da próxima instrução, as opções são: `FROM_INC4` (instrução seguinte na memória), `FROM_BRANCH` (desvio condicionado ao resultado da operação), `FROM_OPCODE25_0` (desvio incondicional, os primeiros 26 bits da instrução definem o desvio) e `FROM_MICROKERNEL` (desvio para troca de tarefa, implementado neste projeto).

Essa sinalização é interpretada pelo módulo `PC_next` (arquivo fonte `pc_next_RT.vhd`) para que ele faça a seleção de qual dos endereços presentes em suas entradas será encaminhada para o processamento (*pipeline*).

Sempre que um pedido de interrupção é gerado pelo hardware, a lógica principal da CPU (arquivo `mlite_cpu_RT.vhd`) considera dois fatores para aceitar o pedido:

1 – Nenhuma instrução complexa deve estar em processamento. Caso isso ocorra, deve-se aguardar até o fim do processamento para aceitar a interrupção.

A CPU Plasma integrada ao *microkernel* por hardware foi configurada para operar com *pipeline* de 2 níveis (busca e execução). Dessa forma, a maior parte das instruções é executada em apenas 1 ciclo de *clock*, porém as instruções de multiplicação e divisão precisam de 32 ciclos de *clock* para serem completadas. Quando essas instruções estão em curso pela ULA (arquivo fonte mult.vhd) um sinal de *pause* é ativado para que nenhuma outra instrução entre em execução. Isso também inibi o atendimento momentâneo de interrupções.

2 – A instrução em execução não pode ser de desvio, ou seja, dentre as opções desta CPU, ela deve ser do tipo FROM_INC4. Instruções de desvio criam inconsistência no valor de PC que deve ser memorizado, isso ocorre no ISA MIPS I, porque sempre que uma instrução de desvio é executada, a próxima instrução na memória também é executada antes que o desvio seja efetivado. Se memorizássemos o endereço de desvio, a instrução seguinte deixaria de ser executada ao retornarmos, criando inconsistência no código.

Satisfeitas as duas condições, a instrução seguinte é anulada com a inserção de bolhas (instruções nulas) no *pipeline*. Isso perdura até que o PC seja carregado com o endereço da próxima instrução válida e o sinal de pedido de interrupção seja removido.

A integração CPU/IHM utilizou este mesmo recurso para garantir que as trocas de tarefas não criassem nenhuma inconsistência na execução do código das tarefas dos programas. Para isso, acrescentamos uma condição à lógica principal da CPU, com hierarquia inferior ao pedido de interrupção por hardware.

Dessa forma, sempre que o sinal *task_switch_RT* é ativado, além de verificar as duas condições citadas anteriormente, a CPU não deve ter nenhum pedido de interrupção pendente.

O Quadro 4 e o Quadro 5 a seguir, representam partes dos trechos de código VHDL responsáveis por aceitar os pedidos de interrupção e de troca de tarefa. Neles fica clara a hierarquia utilizada para carga do próximo valor do PC.

Complementarmente, assim que o sinal *switch_out* no Quadro 5 (equivalente a *switch_out_RT* na

Figura 44) assume valor 1, o IHM (bloco T&C) faz o *backup* do valor de PC da próxima instrução, inverte os bancos de registradores internos e remove o pedido de troca de tarefa. A volta do sinal *switch_out* para 0 representa que a nova tarefa já está em execução e funciona como disparo para o *backup* da tarefa interrompida.

Quadro 4 – Trecho do código VHDL responsável pelo processo de aceitação de um pedido de interrupção e de uma troca de tarefa pela CPU.

```

1  --Arquivo mlite_cpu_RT,vhd
2  intr_proc: process(clk, reset_in, ...)
3  begin
4      if reset_in = '1' then
5          intr_signal <= '0';
6      elsif rising_edge(clk) then
7          if pause_any = '0' then
8              if intr_in = '1' and intr_enable = '1' and
9                  pc_source = FROM_INC4 then
10                 intr_signal <= '1';
11             else
12                 intr_signal <= '0';
13             end if;
14             if task_switch_RT = '1' and escalonador_RT = '1' and
15                 pc_source = FROM_INC4 then
16                 task_switch_signal <= '1';
17             else
18                 task_switch_signal <= '0';
19             end if;
20         end if;
21     end if;
22 end process;

```

Fonte: Autor.

Quadro 5 – Trecho do código VHDL responsável pela hierarquia na decisão de desvios por interrupção e pela troca de tarefa.

```

1  --Arquivo control.vhd
2  if intr_signal = '1' or is_syscall = '1' then
3      ...
4      pc_source := FROM_LBRANCH;
5      mem_source := MEM_FETCH;
6      exception_out <= '1';
7      switch_out <= '0';
8  else
9      exception_out <= '0';
10
11     --Tratamento de pedido para troca de tarefa
12     if task_switch_signal = '1' then
13         ...
14         pc_source := FROM_MICROKERNEL;
15         mem_source := MEM_FETCH;
16         switch_out <= '1';
17     else
18         switch_out <= '0';
19     end if;
20 end if;

```

Fonte: Autor.

3.2.5 Dados extraídos do projeto do IHM-Plasma em FPGA

A primeira análise feita foi a de consumo de recursos de hardware do FPGA, Intel família Cyclone V (24), quando sintetizamos o microcontrolador Plasma original e o microcontrolador IHM-Plasma resultado deste trabalho de pesquisa.

A Tabela 15 representa toda hierarquia de entidades criadas na declaração dos arquivos VHDL que compõem o microcontrolador IHM-Plasma. Além disso, traz a quantidade de células lógicas (*logic cells*, LCs) e bits de memória consumidos que compõem cada uma das entidades e seus subentidades. Esses dados foram coletados em relatórios gerados pela IDE Quartus II utilizada para síntese lógica do dispositivo.

Na Tabela 15 é possível perceber que as duas últimas colunas (LCs de registradores e bits de memória) se referem a dispositivos utilizados para memorização de dados. A diferença é que, os bits de memória são declarados através de funções especiais criadas pelo fabricante, chamadas de *Memory Megafunctions*. Elas utilizam blocos exclusivos para memória, presentes no FPGA utilizado. Se as memórias fossem declaradas da forma convencional, através de células lógicas, a quantidade de células necessárias ultrapassaria a quantidade presente no FPGA. Mesmo em caso de redução das memórias, outros dois problemas ocorreriam: aumento na complexidade de roteamento das células (podendo até impossibilitar a síntese) e redução da velocidade máxima de operação do dispositivo (1). Como é possível perceber, as *Megafunctions* foram usadas tanto na declaração das memórias de dados/programa, como nos bancos internos de registradores.

Seguindo a análise da Tabela 15, é possível observar que o bloco escalonador (entidade *scheduler*) é o que apresenta o maior consumo individual de LCs combinacionais. Isso é reflexo da arquitetura empregada em seu desenvolvimento. Uma árvore binária de com 32 entradas de 21 bits cada compara as tarefas paralelamente para decidir qual a mais indicada para execução (vide Figura 31). Neste caso, fica clara a troca de recursos por eficiência. Para conseguir atualizar a próxima tarefa a ser executada a cada ciclo de clock, a árvore-binária totaliza 672 bits paralelos em sua entrada. No caso dos FPGAs da família Cyclone V, cada LC combinacional tem no máximo 6 entradas, isso resultaria em, ao menos, 112 LCs apenas no primeiro nível da árvore. Isso ocorre porque o escalonador, além de considerar a prioridade das tarefas (3 bits) e os seus estados (1 bit para *ready* ou *running*), ainda trabalha com mais 12 bits para definição do tempo de espera de cada uma delas e 5 bits para identificar seus números.

Podemos observar também, na Tabela 15, o impacto de cada tarefa que dispõe do recurso de sleep (tarefas periódicas, vide seção 3.2.3.4) tem no total de recursos demandados pelo microkernel por hardware. Cada uma delas demanda de um contador decrescente exclusivo (entidade *conter_down*) que consome 76 LCs combinacionais e 17 LCs de registradores.

Tabela 15 – Hierarquia das entidades dos blocos que descrevem o microcontrolador IHM-Plasma (para entradas na tabela com dois números listados: os números entre parênteses indicam o número de recursos do tipo determinado usado apenas pela entidade específica; os números listados fora dos parênteses indicam o total de recursos do tipo dado usado pela entidade específica e todas as suas subentidades na hierarquia).

Nível	Hierarquia de Entidades	Nome da Instância	LCs combinacionais	LCs de registradores	Bits de memória
1	plasma_RT	plasma1	5.554 (415)	1.845 (391)	200.704 (0)
2	mlite_cpu_RT	u1_cpu_RT	1.793 (7)	369 (2)	4.096 (0)
3	pc_next_RT	u1_pc_next_RT	98 (98)	45 (45)	0 (0)
3	mem_ctrl	u2_mem_ctrl	123 (123)	81 (81)	0 (0)
3	control	u3_control	93 (93)	0 (0)	0 (0)
3	reg_bank_duplo_RT	u4_reg_bank_duplo_RT	415 (284)	80 (1)	4096 (0)
4	reg_bank	u1_reg_bank	64 (64)	39 (39)	2.048 (2.048)
4	reg_bank	u2_reg_bank	67 (67)	40 (40)	2.049 (2.048)
3	bus_mux	u5_bus_mux	223 (223)	0 (0)	0 (0)
3	alu_mult_shifter	678_alu_mult_shifter	834 (202)	161 (0)	0 (0)
4	alu	u1_alu	93 (93)	0 (0)	0 (0)
4	mult	u2_mult	471 (471)	161 (161)	0 (0)
4	shifter	u3_shifter	68 (68)	0 (0)	0 (0)
2	ram	u2_ram	4 (4)	0 (0)	65.536 (65.536)
2	ram	u3_ram_dt	4 (4)	0 (0)	65.536 (65.536)
2	ram	u4_ram_rt	15 (15)	0 (0)	65.536 (65.536)
2	uart	u5_uart	85 (85)	76 (76)	0 (0)
2	microkernel	u7_microkernel	3.238 (495)	1.009 (240)	0 (0)
3	reset_controller	u1_reset_controller	6 (6)	5 (5)	0 (0)
3	context_manager	u2_context_manager	129 (129)	244 (244)	0 (0)
3	scheduler	u3_scheduler	2.003 (2003)	384 (384)	0 (0)
3	counter_down	u4_counter_down_1	76 (76)	17 (17)	0 (0)
3	counter_down	u5_counter_down_2	76 (76)	17 (17)	0 (0)
3	counter_down	u6_counter_down_3	75 (75)	17 (17)	0 (0)
3	counter_down	u7_counter_down_4	76 (76)	17 (17)	0 (0)
3	counter_down	u8_counter_down_5	75 (75)	17 (17)	0 (0)
3	counter_down	u9_counter_down_6	75 (75)	17 (17)	0 (0)
3	counter_down	u10_counter_down_7	76 (76)	17 (17)	0 (0)
3	counter_down	u11_counter_down_8	76 (76)	17 (17)	0 (0)

Fonte: Autor.

A Tabela 16, traz informações relativas à quantidade de LCs e bits de memória consumidos por cada uma das versões do microcontrolador sintetizadas no FPGA. Esses dados

foram coletados em relatórios gerados pela IDE Quartus II utilizada para síntese lógica dos dispositivos.

Tabela 16 – Consumo de recursos de hardware do FPGA em função da versão do microcontrolador sintetizada (o incremento percentual do IHM-Plasma em relação ao Plasma original é mostrado entre parênteses).

Recurso	Plasma	IHM-Plasma (incremento)	
		com <i>sleep</i>	sem <i>sleep</i>
LCs combinacionais	1.882	5.554 (195%)	3.289 (75%)
LCs de registradores	504	1.845 (266%)	1.665 (230%)
Bits de memória	198.656	200.704 (1%)	200.704 (1%)

Fonte: Autor.

O microcontrolador IHM-Plasma foi sintetizado de duas formas diferentes, uma com a função *sleep* responsável por bloquear tarefas periódicas por até 65535 *time slices* e outra sem a função *sleep*. Na Tabela 16, é possível observar que em ambos os casos a quantidade de LCs apresentam incrementos significativos (até 266%), ao passo que a quantidade de bits de memória se mantém praticamente constante. Dessa forma, o impacto do incremento no número de LCs no FPGA é minimizado, uma vez que, os bits de memória estão presentes em quantidade ao menos 27 vezes maior que os LCs e representam a maior fração de recursos consumidos do FPGA.

Outro parâmetro analisado foi a frequência máxima de operação dos microcontroladores sintetizados. Segundo seu projetista (Steve Rhoads) (45), a frequência nominal de operação do microcontrolador Plasma é de 25 MHz. A Tabela 17 apresenta as frequências máximas obtidas.

Tabela 17 – Frequência máxima de operação dos microcontroladores Plasma e IHM-Plasma.

Microcontrolador	Frequência máxima de operação (MHz)
Plasma	43,53
IHM-Plasma	33,26

Fonte: Autor.

De acordo com a Tabela 17, em ambos os casos a frequência obtida é superior ao especificado. O valor inferior do IHM-Plasma em relação ao Plasma original é consequência do aumento dos circuitos lógicos e do uso intenso de registradores para sincronismo dos sinais. Mesmo assim apresenta frequência máxima de operação 32% acima da especificação.

3.3 PROGRAMAÇÃO DO MICROCONTROLADOR IHM-PLASMA EM LINGUAGEM *ASSEMBLY*

Como descrito anteriormente, a integração da CPU Plasma modificada com o IHM não se utilizou de alterações ou incrementos no conjunto de instruções MIPS I.

A proposta, apresentada neste projeto de pesquisa, permite a configuração e habilitação do IHM apenas através do uso de novos registradores de funções especiais (descritos na seção 3.2.2).

As alterações propostas no núcleo Plasma permitem, através dos blocos criados em *hardware*, que um microcontrolador de núcleo único opere como um microcontrolador multitarefa. Portanto, ele pode ser considerado uma alternativa para sistemas de tempo real, porém são necessários alguns cuidados e alterações na forma de estruturação do código do programa.

A solução proposta pode ser utilizada para execução de qualquer programa compatível com a ISA MIPS I. Por outro lado, vantagens são observadas mediante a alterações na estrutura do código fonte e a habilitação do escalonamento de tarefas por *hardware*. Para isso, o código deve ser estruturado de forma fracionada, dividido em tarefas com propósitos bem claros semelhante à técnica de projeto *Interrupt-Only Systems*, conceituada na seção 2.3.3.

Diferentemente da técnica citada, onde as interrupções da CPU disparam a execução de tarefas finitas específicas do programa, para uso do escalonador proposto, o programador pode pensar em suas tarefas como funções ou sub-rotinas de um programa, as quais não retornam nenhum valor e que normalmente após a inicialização de suas variáveis, entram em um laço infinito, que contém o código que descreve sua operação.

Cada uma dessas tarefas deve possuir um valor de prioridade para execução e uma faixa de endereços na memória RAM, onde o seu *stack* é armazenado.

O programador ao desenvolver seu código, deve utilizar a sequência de passos apresentada a seguir, como linha mestra, para o uso do escalonador e para tirar maior proveito de um sistema multitarefa, são eles:

1. Criar a estrutura do programa, passando pela inicialização de variáveis, configuração de periféricos e procedimentos básicos até atingir o ponto onde inicia-se o laço principal do programa;
2. Criar as tarefas como se fossem funções com laços infinitos;

3. De volta ao código principal do programa, é preciso salvar no TCB de cada uma das tarefas, o endereço da tarefa (PC) e o endereço final da região reservada para o *stack* da tarefa (SP);
4. Inicializar os registradores responsáveis pela configuração do IHM. Esses registradores armazenarão informações como duração de cada *time slice* em número de *clocks* da CPU, número das tarefas habilitadas e prioridade de cada uma delas;
5. Habilitar o escalonamento de tarefas através do registrador correspondente;
6. Executar a tarefa 0, chamada de tarefa *idle*. Essa tarefa nunca pode ser desabilitada, bloqueada ou suspensa. Isso porque a CPU precisa ter sempre algo para ser executado, mesmo quando nenhuma atividade prática existir. É por isso que, ela é chamada de *idle* (ociosa) e deve ocupar as possíveis sobras de tempo de processamento em cada *time slice*. Geralmente tem nível de prioridade igual ou inferior ao da tarefa com menor prioridade;
7. Criar um laço infinito vazio no programa. Essa é uma medida apenas preventiva para caso a tarefa 0 não exista ou seja interrompida por erro de programação.

Assim que o escalonador for habilitado, o controle sobre a execução do código passa da CPU para o IHM, que, através do bloco escalonador, avalia a tarefa mais apta para ser executada em cada *time slice* como descrito na seção 3.2.3.2.

O Quadro 6 e o Quadro 7 exemplificam o procedimento descrito anteriormente através de um programa com 3 tarefas escrito em linguagem *Assembly* compatível com o ISA MIPS I.

O programa apresentado usou como ferramenta para depuração o kit de desenvolvimento DE1-SoC da Terasic Technologies (51), onde os pinos de entrada do processador (GPIOA) estão conectadas às chaves digitais e aos pinos de saída (GPIO0), que estão conectados aos *leds*.

No Quadro 6, entre as linhas 4 e 15 é feita pré-configuração dos TCBs das três tarefas do programa com os endereços iniciais de seus PCs e SPs. Entre as linhas 18 e 25 é apresentado toda configuração do IHM, a identificação binária das tarefas ativas (linhas 18 e 19), o ajuste de prioridade dessas tarefas (linhas 20 e 21), o ajuste o *time slice* desejado (linhas 22 e 23) e por fim o escalonamento de tarefas é habilitado (linhas 24 e 25). Perceba que a tarefa 0 tem a menor prioridade dentre as três, essa medida faz dela sempre a última opção do escalonador e somente será executada quando as tarefas 1 e 2 estiverem bloqueadas. Porém, ela é executada ao menos uma vez, assim que o escalonamento é habilitado (linha 26).

Quadro 6 – Trecho do código *Assembly* responsável pela configuração das tarefas e do IHM.

#	Label	Mnemônico	Operadores	Comentário
0		.text		# Os itens subsequentes são instruções
1		.globl main		# Declara que a label main é global
2				#PREPARAÇÃO DAS TAREFAS
3	main:	lui	\$gp,0x0002	# Atualiza global pointer com os 16 msbs (0x0002xxxx)
4		la	\$t1,task0	# Carrega em t1 o endereço da tarefa 0
5		sw	\$t1,0x0000(\$gp)	# Armazena no seu TCB (1o endereço)
6		li	\$t1,0x0001FFFC	# Carrega em t1 o endereço do stack da tarefa 0
7		sw	\$t1,0x0074(\$gp)	# Armazena no seu TCB (29o endereço)
8		la	\$t1,task1	# Carrega em t1 o endereço da tarefa 1
9		sw	\$t1,0x0100(\$gp)	# Armazena no seu TCB (1o endereço)
10		li	\$t1,0x0001FEFC	# Carrega em t1 o endereço do stack da tarefa 1
11		sw	\$t1,0x0174(\$gp)	# Armazena no seu TCB (29o endereço)
12		la	\$t1,task2	# Carrega em t1 o endereço da tarefa 2
13		sw	\$t1,0x0200(\$gp)	# Armazena no seu TCB (1o endereço)
14		li	\$t1,0x0001FD FC	# Carrega em t1 o endereço do stack da tarefa 2
15		sw	\$t1,0x0274(\$gp)	# Armazena no seu TCB (29o endereço)
16				#PREPARAÇÃO DO MICROKERNEL
17		lui	\$gp,0x2000	# Atualiza global pointer com os 16 msbs (0x2000xxxx)
18		li	\$t1,0x07	# Habilita as tarefas 0-2 (bits 0-2)
19		sw	\$t1,0xB0(\$gp)	
20		li	\$t1,0x00000331	# Tarefas 1 e 2 prio. = 3, tarefa 0 prio. = 1
21		sw	\$t1,0xC0(\$gp)	
22		li	\$t1,1000	# Time slice igual a 1000 ciclos de clock
23		sw	\$t1,0x90(\$gp)	
24		li	\$t1,1	# Habilita o escalonador
25		sw	\$t1,0x80(\$gp)	
26		j	task0	# Salta para tarefa 0 (idle)
27		nop		
28	loop:	j	loop	# Loop vazio infinito
29		nop		

Fonte: Autor.

O Quadro 7 destaca as três tarefas criadas, note que em todas elas a penúltima instrução é uma instrução de desvio incondicional *j* (*jump*) para o começo de seu ciclo de repetição.

Uma característica importante da arquitetura de instruções MIPS I é que, sempre a instrução, seguinte a uma instrução de desvio, é executada. Isso ocorre antes que o PC seja ajustado para o novo ponto de execução do programa. Nos exemplos apresentados, foi colocada uma instrução *nop* (*no operation*, em português nenhuma operação) após as instruções de desvio presentes no final de cada uma das tarefas. Essa medida evita que a primeira instrução, da tarefa seguinte (quando existir uma), seja executada criando inconsistência no programa.

Outra opção, geralmente utilizada pelo compilador GCC, é inverter a ordem entre a instrução de desvio incondicional e a instrução imediatamente anterior a ela. Evitando assim a necessidade de uso de operações nulas (*nop*). Como exemplo, na tarefa 1 do Quadro 7, os conteúdos das linhas 54 e 55 seriam invertidos, eliminando a necessidade da instrução *nop* da

linha 56. Essa troca apenas torna o código mais eficiente, uma vez que uma instrução nula seria poupada.

Quadro 7 – Trecho do código *Assembly* responsável pela declaração das tarefas.

#	Label	Mnemônico	Operadores	Comentário
30				#TAREFAS CRIADAS NA FORMA DE SUB-ROTINAS
31	task0:	li	\$t2,0xFFFF	
32		li	\$t1,0x5555	
33	task0_2:	sw	\$t2,0x0040(\$gp)	# Apaga todos os leds
34		nop		
35		sw	\$t1,0x0030(\$gp)	# Acende os leds alternados
36		nop		
37		j	task0_2	# Espera escalonador começar a funcionar
38		nop		
39	task1:	lui	\$gp,0x2000	#Ponteiro para os registradores
40		li	\$t0,0xFFFF	
41		li	\$t1,0x0001	
42		li	\$t2,0x00FF	# Oposto da task 1
43		li	\$t3,0x10002	# Tempo de sleep = 2 time slices, bit 16 = 1
44		li	\$t4,0x00002	# Tempo de sleep = 2 time slices, bit 16 = 0
45	task1_2:	sw	\$t0,0x0040(\$gp)	# Apaga todos os leds
46		nop		
47		sw	\$t1,0x0030(\$gp)	# Acende o led 0
48		nop		
49		lw	\$t5,0x50(\$gp)	# \$t5 <- GPIOA
50		andi	\$t6,\$t5, 0x0001	# \$t6 <- 0x0000000? (máscara para SW0 - EN sleep)
51		beqz	\$t6,task1_2	# Se a chave = 0, desvia. Caso contrário sleep
52		nop		
53		sw	\$t3,0x0100(\$gp)	# Sleep da tarefa atual (pulso no bit 16)
54		sw	\$t4,0x0100(\$gp)	# Sleep da tarefa atual (pulso no bit 16)
55		j	task1_2	
56		nop		
57	task2:	lui	\$gp,0x2000	# Ponteiro para os registradores
58		li	\$t0,0xFFFF	
59		li	\$t1,0x0002	# Oposto da task 0
60		li	\$t2,0x00FF	
61		li	\$t3,0x10004	# Tempo de sleep = 4 time slices, bit 16 = 1
62		li	\$t4,0x00004	# Tempo de sleep = 4 time slices, bit 16 = 0
63	task2_2:	sw	\$t0,0x0040(\$gp)	#Apaga todos os leds
64		nop		
65		sw	\$t1,0x0030(\$gp)	#Acende o led 1
66		nop		
67		lw	\$t5,0x50(\$gp)	# \$t5 <- GPIOA
68		andi	\$t6,\$t5, 0x0002	# \$t6 <- 0x0000000? (máscara para SW1 - EN sleep)
69		beqz	\$t6,task2_2	# Se a chave = 0, desvia. Caso contrário sleep
70		nop		
71		sw	\$t3,0x0100(\$gp)	# Sleep da tarefa atual (pulso no bit 16)
72		sw	\$t4,0x0100(\$gp)	# Sleep da tarefa atual (pulso no bit 16)
73		j	task2_2	
74		nop		

Fonte: Autor.

É importante enfatizar que após a habilitação do escalonamento de tarefas a região da memória reservada para o armazenamento dos TCBs (0x00020000 - 0x0002FFF, vide Tabela 6 na seção 3.2.2) não pode mais ser escrita ou lida pelo programa em execução, ficando exclusivamente conectada ao *microkernel* por hardware. Já a região reservada para dados (0x00010000 - 0x0001FFF) é fracionada em blocos de 64 *words* (256 bytes) para o armazenamento do *stack* de cada uma das tarefas. A Tabela 18 apresenta a distribuição desses blocos de acordo com o número da tarefa.

Tabela 18 – Mapeamento de memória RAM para a definição da faixa de memória reservada para a pilha (*stack*) de cada tarefa.

Stack/Tarefa	Endereço Contexto		Stack/Tarefa	Endereço Contexto	
	Inicial	Final		Inicial	Final
0	0x00010000	0x0001007F	16	0x00011000	0x0001107F
1	0x00010100	0x0001017F	17	0x00011100	0x0001117F
2	0x00010200	0x0001027F	18	0x00011200	0x0001127F
3	0x00010300	0x0001037F	19	0x00011300	0x0001137F
4	0x00010400	0x0001047F	20	0x00011400	0x0001147F
5	0x00010500	0x0001057F	21	0x00011500	0x0001157F
6	0x00010600	0x0001067F	22	0x00011600	0x0001167F
7	0x00010700	0x0001077F	23	0x00011700	0x0001177F
8	0x00010800	0x0001087F	24	0x00011800	0x0001187F
9	0x00010900	0x0001097F	25	0x00011900	0x0001197F
10	0x00010A00	0x00010A7F	26	0x00011A00	0x00011A7F
11	0x00010B00	0x00010B7F	27	0x00011B00	0x00011B7F
12	0x00010C00	0x00010C7F	28	0x00011C00	0x00011C7F
13	0x00010D00	0x00010D7F	29	0x00011D00	0x00011D7F
14	0x00010E00	0x00010E7F	30	0x00011E00	0x00011E7F
15	0x00010F00	0x00010F7F	31	0x00011F00	0x00011F7F

Fonte: Autor.

Diferentemente dos blocos de memória reservados para os TCBs, os blocos reservados para os *stacks* podem ser alterados e redefinidos pelo usuário através do programa. A opção de reservar 64 *words* para cada tarefa está atrelada simplesmente a uma padronização definida de acordo com a complexidade das tarefas utilizadas nos testes realizados.

No Apêndice A é apresentado o ambiente de software utilizado para edição, montagem e geração do arquivo binário a partir de um programa escrito em linguagem *Assembly*, bem como o procedimento de geração dos arquivos hexadecimais utilizados durante a síntese do FPGA, para geração da memória de programa do microcontrolador IHM-Plasma.

Uma série de programas compatíveis com o microcontrolador IHM-Plasma e escritos em linguagem *Assembly* estão disponíveis em formato digital e podem ser acessados através do link na referência (12).

3.4 ARCABOUÇO (FRAMEWORK) PARA PROGRAMAÇÃO DO MICROCONTROLADOR IHM-PLASMA EM LINGUAGEM C

O microcontrolador Plasma é compatível com o compilador GCC e nos permite assim escrever programas em linguagem C de forma muito mais clara e rápida, em relação aos programas escritos em linguagem *Assembly*. Desta forma, nesta seção, são descritos todos os arquivos, recursos disponíveis e desenvolvidos, que formam o arcabouço, para o desenvolvimento de aplicações/sistemas usando linguagem C para o microcontrolador IHM-Plasma.

Neste estudo, as bibliotecas, previamente fornecidas (*boot.asm* e *no_os.c*) pelo criador do Plasma, foram editadas ou substituídas para permitir o uso dos recursos oferecidos pelo IHM. Além disso, uma nova biblioteca (*hd_microk.h* e *hd_microk.c*) foi criada com o objetivo de tornar a programação dos recursos de tempo real (multitarefas) o mais parecido possível com a programação feita quando se utiliza um RTOS comercial. Nela, foram implementadas funções para criação de tarefas, *sleep* de tarefas e inicialização do escalonamento de tarefas. Complementarmente, funções equivalentes para inicialização de um escalonador por software e *sleep* foram implementadas, assim como, funções especiais para realização do escalonamento de tarefas por software com base em interrupções temporizadas da própria CPU.

Essa estratégia permite com que comparativos entre as duas abordagens (hardware e software) sejam facilmente implementados com poucas alterações nos códigos fontes.

O arquivo principal dos projetos desenvolvidos foi chamado de *plasma_rt.c*. Nele são criadas as funções-tarefas e a função principal do programa (responsável pela parametrização do IHM e sua inicialização).

Nas subseções seguintes são descritos os arquivos que compõe esse arcabouço.

3.4.1 Arquivo de partida do programa e desvio para tratamento de interrupções (*boot.asm*)

Este é o arquivo de *boot* da CPU Plasma e é o único arquivo que, necessariamente, deve ser escrito em linguagem *Assembly*.

Cabe aqui esclarecer que, a nomenclatura usada para trechos de códigos com propósitos específicos escritos em linguagem *Assembly* e C foi diferenciada, seguindo o padrão normalmente empregado nos livros e pelos programadores. Chamamos de sub-rotinas os trechos escritos em *Assembly* e de funções os trechos escritos em C.

Dessa forma, a seguir são descritas as principais sub-rotinas utilizadas:

- `entry`: é o primeiro trecho de código presente no arquivo. Após a compilação, é gravado no endereço 0 da memória, graças a uma diretiva especial do compilador (`.set noreorder`). Nele consta o desvio para função ‘main’ do programa escrito em linguagem C, como pode ser visto na linha 42 do Quadro 8, que apresenta o código completo da sub-rotina.

Quadro 8 – Sub-rotina `entry` do arquivo `boot.asm`, responsável pelo desvio da CPU para função `main`.

```

25      .ent    entry
26  entry:
27      .set    noreorder
28
29      #These four instructions should be the first instructions.
30      #convert.exe previously initialized $gp, .sbss_start, .bss_end, $sp
31      la      $gp, _gp           #initialize global pointer
32      la      $5, __bss_start    #$5 = .sbss_start
33      la      $4, _end           #$2 = .bss_end
34      la      $sp, InitStack+488 #initialize stack pointer
35
36  $BSS_CLEAR:
37      sw      $0, 0($5)
38      slt     $3, $5, $4
39      bnez    $3, $BSS_CLEAR
40      addiu   $5, $5, 4
41
42      jal     main
43      nop
44  $L1:
45      j       $L1
46
47      .end    entry

```

Fonte: Autor.

- `interrupt_service_routine`: assim como descreve seu nome, é o ponto de entrada para o tratamento de interrupções, seu ponto de gravação na memória é fixado no endereço 0x0000003F. Essa sub-rotina foi alterada a fim de permitir a criação do escalonador por software.

Originalmente, consistia de quatro ações: salvar no *stack* o contexto na CPU; gerar um vetor com *status* das interrupções; chamar a rotina de tratamento das

interrupções (OS_InterruptServiceRoutine, arquivo no_os_rt.c); e por último restaurar o contexto da CPU.

Após modificada, a sub-rotina passou a fazer o salvamento e restauração completa de todos os registradores da CPU, passou a informar para função de tratamento o valor do SP da tarefa atual e a receber como retorno o SP da próxima tarefa. Dessa forma, ao ser gerada uma interrupção temporal para escalonamento de tarefas, a CPU não retorna mais para antiga tarefa em execução e sim para a próxima a ser executada. Seu código pode ser visto parcialmente no Quadro 9.

Quadro 9 – Sub-rotina interrupt_service_routine do arquivo boot.asm, responsável pelo início e pelo fim do tratamento de uma interrupção pela CPU.

```

51      #address 0x3c
52      .global interrupt_service_routine
53      .ent interrupt_service_routine
54      interrupt_service_routine:
55      .set noreorder
56      .set noat
57
62      addi $29, $29, -136 #adjust sp (considerando s0-s7)
63      sw   $1, 16($29)    #at
64      sw   $2, 20($29)    #v0
97
98      lui   $6, 0x2000
99      lw    $4, 0x20($6)   #IRQ_STATUS
100     lw    $6, 0x10($6)   #IRQ_MASK
101     and   $4, $4, $6     #Envia em $a0 produto de IRQ_STATUS and IRQ_MASK
102     jal   OS_InterruptServiceRoutine #Chama função
103     addi  $5, $29, 0     #Envia em $a1 o valor do $sp
104     #Retornar em $v0 o novo valor de $sp
105     addi  $29, $2, 0
106     #Restore all temporary registers
107     lw    $1, 16($29)    #at
108     lw    $2, 20($29)    #v0
139
140     # addi $29, $29, 104 #adjust sp
141     addi  $29, $29, 136 #adjust sp (considerando s0-s7)
142
143     isr_return:
144     ori   $27, $0, 0x1   #re-enable interrupts
145     jr    $26
146     mtc0  $27, $12       #STATUS=1; enable interrupts
147
148     .end interrupt_service_routine
149     .set at

```

Fonte: Autor.

A Figura 45 ilustra as duas versões da sub-rotina de tratamento de interrupções: original (a), que é aquela que não considera o IHM implementado, e modificada (b), que foi alterada no sentido de poder ser utilizada para implementação de um *microkernel* por software, utilizado para comparação de desempenho com o IHM. Pode-se considerar que, a alteração do SP durante o tratamento da interrupção é a medida chave para criação de um escalonador por software.

Somente assim, é possível saltar de uma tarefa para outra, sem que haja um ponto certo para que isso ocorra durante a execução do programa.

Nos processadores MIPS, os registradores \$s0-\$s7 são chamados de *saved register*, pois sempre que uma função é chamada, se ela fizer uso de algum deles, ela deve salvar o valor do registrador no *stack* antes do uso e restaurá-lo ao término. Por essa razão, originariamente esse conjunto de registradores não era salvo ou restaurado quando a sub-rotina em questão era executada. Uma vez que, passamos a saltar de uma tarefa para outra, esses registradores foram incluídos na sub-rotina. Apesar de não serem explicitamente mencionados na Figura 45 (b), o termo contexto completo se refere a essa alteração.

Figura 45 – Fluxograma da sub-rotina `interrupt_service_routine` do arquivo `boot.asm`, onde à esquerda é descrita a versão original (a) e à direita é descrita a versão modificada para aceitar escalonamento de tarefas por software (b).

- **OS_AsmInterruptEnable**: essa sub-rotina funciona como chave geral para a habilitação de interrupções e, para isso, acessa o registrador \$12 do Coprocessador 0. No bit 0 desse registrador é escrito 1 para habilitar interrupções e escrito 0 para desabilitar. Para que uma interrupção seja habilitada ainda é preciso ativar o bit correspondente no registrador de função especial **IRQ_Mask** (endereço 0x20000010 da memória RAM).

O Quadro 10 apresenta o código dessa sub-rotina. Na linha 158 o valor do registrador \$12 é salvo no registrador \$2 (\$v0) da CPU e posteriormente, na linha 160, o valor do registrador \$12 é atualizado como o valor do registrador \$4 (\$a0) da CPU.

Quadro 10 – Sub-rotina **OS_AsmInterruptEnable** do arquivo **boot.asm**.

```

154     .global OS_AsmInterruptEnable
155     .ent OS_AsmInterruptEnable
156 OS_AsmInterruptEnable:
157     .set noreorder
158     mfc0 $2, $12
159     jr    $31
160     mtc0 $4, $12           #STATUS=1; enable interrupts
161     #nop
162     .set reorder
163     .end OS_AsmInterruptEnable

```

Fonte: Autor.

Aqui vale um esclarecimento de como os compiladores C trabalham com os registradores internos da CPU, quando uma chamada e retorno de função é feita. Considere a linha de código C:

```
res = OS_AsmInterruptEnable(var);
```

Os registradores \$a0-\$a3 são usados para gravação de argumentos de funções, já os registradores \$v0-\$v1 são usados para o retorno de uma função. Logo podemos considerar que, após a compilação do código acima, correrá a seguinte atribuição de valores:

```
$a0 = var
```

```
res = $v0
```

De acordo com o Quadro 10, isso implica que a sub-rotina primeiramente grava o valor pré-atualização de \$12 em \$v0 e em seguida atribui o novo valor (\$a0) à \$12. Assim, quando executada a linha de código C exemplificada, a variável *res* receberá o *status* pré-atualização do registrador \$12 do Coprocessador 0.

3.4.2 Arquivo de apoio ao programa principal (no_os_rt.c)

Este arquivo substituiu o arquivo original no_os.c. Nele foi implementada uma nova função, chamada OS_InterruptServiceRoutine, para tratamento de interrupções, que foi ajustada para as necessidades dos testes com escalonamento de tarefas via software.

O Quadro 11 apresenta a estrutura da função OS_InterruptServiceRoutine. Chamada pela sub-rotina interrupt_service_routine (arquivo boot.asm), essa função recebe como argumentos um vetor com o status das fontes de interrupção e o valor do *stack pointer* (SP) da tarefa interrompida. Como medida de concentrar todo o código relacionado às operações com tarefas no arquivo plasma_rt.c, a função em questão aciona uma terceira função chamada trataInt que foi declarada no arquivo citado. Os mesmos parâmetros são passados como argumentos e seu retorno será o valor do SP da próxima tarefa, também usado como retorno da função OS_InterruptServiceRoutine.

Quadro 11 – Função OS_InterruptServiceRoutine do arquivo no_os_rt.c.

```

61 unsigned int OS_InterruptServiceRoutine(unsigned int status, unsigned int stackp)
62 {
63     //Função para tratamento de interrupções
64     //retorna o valor do no stack pointer (registrador $sp ou $29)
65     stackp = trataInt(status, stackp); // (plasma_rt.c)
66
67     return stackp;
68 }

```

Fonte: Autor.

As demais funções deste arquivo são utilizadas como apoio para conversão de valores, acesso a porta serial e I/Os.

3.4.3 Arquivo com a biblioteca de funções do IHM e do *microkernel* por software (hw_microk.c)

Esse arquivo pode ser considerado uma biblioteca de funções do IHM e do *microkernel* implementado em software.

Sendo assim, podemos dividi-la em dois grupos de funções, as funções de acesso ao hardware (IHM) e as de acesso ao software.

3.4.3.1 Funções de acesso ao IHM

A seguir são apresentadas todas as funções de acesso ao hardware.

- `int sleep_yield_task(unsigned int period)`

Descrição:

Função para pedido de *sleep* ou *yield* da tarefa em execução.

Entrada:

`period`: define o número de *time slices* que a tarefa ficará bloqueada, o valor 0 força o bloqueio apenas até o final o *time slice* atual.

Saída:

Retorna sempre o valor 0.

- `int create_task(int (*task_function_pointer)(), int number, int priority)`

Descrição:

Função para criação de tarefas.

Entradas:

`(*task_function_pointer)()`: ponteiro para uma função-tarefa. Cada tarefa pode ser considerada uma instância de uma função-tarefa. Isso permite que varias tarefas possam ser criadas tendo como base uma mesma função-tarefa;

`number`: número da tarefa criada (0-31). Cada tarefa tem um número exclusivo que vai definir onde será alocado seu TCB e *stack*;

`priority`: prioridade da tarefa. Deve ser um valor de 0 a 7. Quanto maior o número mais relevante a tarefa.

Saída:

Retorna sempre o valor 0.

- `int create_task_a(int (*task_function_pointer)(), int number, int priority, int argumento)`

Descrição:

Outra opção de função para criação de tarefas.

Entradas:

`(*task_function_pointer)()`: ponteiro para uma função-tarefa. Cada tarefa pode ser considerada uma instância de uma função-tarefa. Isso permite que várias tarefas possam ser criadas tendo como base uma mesma função-tarefa;

`number`: número da tarefa criada (0-31). Cada tarefa tem um número exclusivo que vai definir onde será alocado seu TCB e seu *stack*;

`priority`: prioridade da tarefa. Deve ser um valor de 0 a 7. Quanto maior o número mais relevante a tarefa;

`argumento`: representa um valor inteiro que pode ser passado como argumento da função-tarefa. Utilizado para passagem de parâmetro que pode ajustar o funcionamento de uma tarefa. Exemplo: frequência com que a tarefa deve ser executada.

Saída:

Retorna sempre o valor 0.

Essas funções são as únicas necessárias ao programador para que ele possa utilizar o *microkernel* por hardware, as demais configurações do mesmo são feitas por escrita direta em registradores. Assim, como descrito na seção 3.3, a memória reservada para o *stack* de cada tarefa é de 64 *words* e respeita a distribuição apresentada na Tabela 18.

3.4.3.2 Funções de acesso ao microkernel implementado por software

A seguir são apresentadas todas as funções de acesso ao software.

- `void sw_scheduler_init(void)`

Descrição:

Inicializa escalonador definindo os SP de cada tarefa e formando uma lista com a sequência de execução das tarefas de acordo com as tarefas habilitadas. Não considera prioridade para isso.

Entrada:

Nula.

Saída:

Nula.

- void sw_scheduler_init_v2(void)

Descrição:

Inicializa escalonador definindo os SP de cada tarefa e organiza a ordem de execução das tarefas habilitadas. Para isso, supõe que as tarefas com maior numeração têm prioridade maior que as com menor numeração.

Entrada:

Nula.

Saída:

Nula.

- unsigned int sw_scheduler(unsigned int stackp)

Descrição:

Função de escalonamento preemptivo. Sempre que executada interrompe a tarefa atual e passa a executar a próxima tarefa da lista. Trabalha em conjunto com a função sw_scheduler_init, onde as tarefas não possuem prioridades.

Entrada:

stackp: valor do SP da tarefa interrompida. No *stack* já está salvo o PC.

Saída:

Valor do SP da próxima tarefa a ser executada.

- unsigned int sw_scheduler_v2(unsigned int stackp)

Descrição:

Função de escalonamento preemptivo. Sempre que executada interrompe a tarefa atual e passa a executar a próxima tarefa da lista. Trabalha em conjunto com a função sw_scheduler_init_v2, onde as tarefas possuem prioridades.

Entrada:

stackp: valor do SP da tarefa interrompida. No *stack* já está salvo o PC.

Saída:

Valor do SP da próxima tarefa a ser executada.

- unsigned int sw_sleep_yield_task(unsigned int period)

Descrição:

Função para pedido de *sleep* ou *yield* da tarefa em execução. É capaz de forçar uma interrupção de hardware para que a troca de tarefas ocorra imediatamente.

Entrada:

period: define o número de *time slices* que a tarefa ficará bloqueada, o valor 0 força o bloqueio apenas até o final o *time slice* atual.

Saída:

Retorna sempre o valor 0.

As cinco funções descritas, foram criadas exclusivamente para permitir o comparativo de desempenho entre a solução proposta por hardware e as soluções convencionais baseadas em software executado.

A chamada das funções `sw_scheduler` e `sw_scheduler_v2` ocorre no corpo da função de tratamento de interrupções e respeitam a *flag* de interrupção do temporizador interno da CPU Plasma.

3.4.4 Arquivos principais para criação dos projetos (plasma_rt.h e plasma_rt.c)

Este é o par de arquivos principal utilizado para criação de projetos usados para testes e certificações das funcionalidades do microcontrolador Plasma modificado (IHM-Plasma). No arquivo de cabeçalho `plasma_rt.h` são definidas constantes que representam os endereços dos registradores (vide Quadro 12), os endereços dos bits de interrupções e os protótipos das funções auxiliares criadas. Já no arquivo fonte `plasma_rt.c` é feita a declaração da função ‘main’ do projeto, de todas as funções-tarefas, das funções auxiliares, das variáveis globais e de uma pequena memória FIFO que funciona como canal de comunicação entre tarefas.

Quadro 12 – Lista de registradores e memórias presentes no arquivo plasma_rt.h.

```

18  /***** Hardware addresses *****/
19  #define RAM_INTERNAL_BASE 0x00000000 //8KB
20  #define RAM_EXTERNAL_BASE 0x10000000 //1MB or 64MB
21  #define RAM_EXTERNAL_SIZE 0x00100000
22  #define ETHERNET_RECEIVE 0x13ff0000
23  #define ETHERNET_TRANSMIT 0x13fe0000
24  #define MISC_BASE 0x20000000
25  #define UART_WRITE 0x20000000
26  #define UART_READ 0x20000000
27  #define IRQ_MASK 0x20000010
28  #define IRQ_STATUS 0x20000020
29  #define GPIO0_SET 0x20000030
30  #define GPIO0_CLEAR 0x20000040
31  #define GPIOA_IN 0x20000050
32  #define COUNTER_REG 0x20000060
33  #define ETHERNET_REG 0x20000070
34
35  #define SCHEDULER_EN 0x20000080 //Habilita o escalonador
36  #define TIME_SLICE 0x20000090 //Tempo do tick ou time slice em pulsos de clock
37  #define TASK_NUMBER 0x200000A0 //Tarefa ativa (era Quantidade de tarefas 23/09/17)
38  #define TASK_VECTOR 0x200000B0 //Vetor de tarefas (existe/não existe, máx. 32)
39  #define PRI1_VECTOR 0x200000C0 //Vetor de prioridade das tarefas 7~0 (4 bits por tarefa)
40  #define PRI2_VECTOR 0x200000D0 //Vetor de prioridade das tarefas 15~8
41  #define PRI3_VECTOR 0x200000E0 //Vetor de prioridade das tarefas 23~16
42  #define PRI4_VECTOR 0x200000F0 //Vetor de prioridade das tarefas 31~24
43  #define SLEEP_TIME 0x20000100 //Tempo de sleep ou yield da tarefa atual
44
45  #define FLASH_BASE 0x30000000

```

Fonte: Autor.

A seguir é feita a descrição da estrutura da memória FIFO e seu conjunto de funções de acesso, bem como é apresentado um exemplo de criação de projeto utilizando os recursos de escalonamento de tarefas por hardware e software.

3.4.4.1 Memória FIFO para comunicação entre tarefas

Memórias do tipo FIFO normalmente são criadas a partir de memória RAM convencional, porém dispensam o uso de endereçamento. Apenas operações de leitura e escrita são realizadas sem a menção do endereço. *Flags* sinalizam se a mesma se encontra cheia ou vazia.

Para testes da solução proposta, foi criada uma memória FIFO com capacidade de armazenamento de dez valores formatados como variáveis do tipo *char* (1 byte). O Quadro 13 apresenta a estrutura (*struct*) criada para representação desta memória. Perceba que existe um conjunto de 4 variáveis do tipo *unsigned int* (inteiro sem sinal) utilizadas para compatibilizar o funcionamento da memória FIFO com uma memória RAM convencional.

Quadro 13 – Estrutura da memória FIFO declarada e utilizada como canal de comunicação entre tarefas.

```

112     #define fifo_size 10
113
114     struct mem
115     {
116         char dados[fifo_size];
117         unsigned int start;      //Posição da próxima escrita
118         unsigned int end;        //Posição da próxima leitura
119         unsigned int writes;     //Quantidade de escritas
120         unsigned int reads;      //Quantidade de leituras
121     };
122
123     struct mem fifo;

```

Fonte: Autor.

A seguir são apresentadas todas as funções de apoio ao uso desta memória.

- void fifo_reset()

Descrição:

Reseta a memória.

Entrada:

Nula.

Saída:

Nula.

- unsigned int fifo_in(char v)

Descrição:

Escreve um caractere na memória.

Entrada:

v: caractere a ser escrito na memória.

Saída:

1: sucesso na escrita.

0: falha na escrita por conta de a memória estar cheia.

- char fifo_out()

Descrição:

Lê um caractere da memória.

Entrada:

Nula.

Saída:

Próximo caractere ou 0 caso a memória esteja vazia.

- unsigned int fifo_empty()

Descrição:

Verifica se a memória está vazia.

Entrada:

Nula.

Saída:

1: memória vazia.

0: memória não vazia.

- unsigned int fifo_full()

Descrição:

Verifica se a memória está cheia.

Entrada:

Nula.

Saída:

1: memória cheia.

0: memória não cheia.

3.4.5 Exemplo de criação de um projeto baseado no escalonamento de tarefas por hardware (IHM) e por software usando linguagem C

Esta seção apresenta meios de verificar o funcionamento do microcontrolador IHM-Plasma realizando escalonamento de tarefas por hardware (IHM) ou software (interrupções e funções especiais).

O escalonamento por software usa como base interrupções temporizadas geradas pelo contador interno da CPU, que tem como fonte de *clock*, o próprio *clock* da CPU.

Dois tipos de interrupções podem ser geradas, a primeira quando um bit específico do contador (padrão bit 18) vale 1 e a outra quando este bit vale 0.

interrupção (mudança do estado do bit testado), qualquer uma das duas funções de escalonamento (`sw_scheduler` e `sw_scheduler_v2`) pode ser executada. Seu retorno é o SP da próxima tarefa a ser executada.

Quadro 15 – Função `trataInt` responsável pelo tratamento de interrupções da CPU (arquivo `plasma_rt.c`),

```

351 //XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
352 //FUNÇÃO DE TRATAMENTO DE INTERRUPÇÃO
353 //XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
354 unsigned int trataInt(unsigned int status, unsigned int stackp)
355 {
356     unsigned int temp;
357     //Se é interrupção de COUNTER
358     if(status & IRQ_COUNTER18)
359     {
360         //Inverte o tipo de interrupção
361         DisableCounter(); EnableNotCounter();
362         //stackp = sw_scheduler(stackp); // (hd_microk.c)
363         stackp = sw_scheduler_v2(stackp); // (hd_microk.c)
364     }
365     //Se é interrupção de NOT_COUNTER
366     else if(status & IRQ_COUNTER18_NOT)
367     {
368         //Inverte o tipo de interrupção
369         DisableNotCounter(); EnableCounter();
370         //stackp = sw_scheduler(stackp); // (hd_microk.c)
371         stackp = sw_scheduler_v2(stackp); // (hd_microk.c)
372     }
373     return stackp;
374 }

```

Fonte: Autor.

O Quadro 16 apresenta duas funções-tarefas, uma utilizada para a tarefa *idle* (`task0`) e a outra utilizada para as tarefas 1 e 2 (`taskLed`). A primeira sempre que executada envia o caractere ponto (.) pela porta serial. Já a segunda pisca o *led* conectado ao bit da porta de saída correspondente a posição indicada pelo argumento *val*.

Quadro 16 – Funções-tarefas utilizadas para criação de tarefas (arquivo plasma_rt.c).

```

376 //XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
377 //TESTE HD/SW SCHEDULER FUNÇÕES-TAREFAS
378 //XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
379 int task0(char val)
380 {
381     puts("Task0");
382     //Idle
383     while(1)
384     {
385         putchar('.'); //Escreve . pela serial
386     }
387 }
388
389 int taskLed(char val)
390 {
391     int i;
392     puts("Task Led");
393
394     while(1)
395     {
396         //Pisca led escolhido
397         for(i = 1000000;i>0;i--);
398         set_output_pin(val); //Acende
399         for(i = 1000000;i>0;i--);
400         clear_output_pin(val); //Apaga
401     }
402 }

```

Fonte: Autor.

O Quadro 17 representa a função principal do programa, onde três tarefas são criadas. A primeira, a tarefa *idle*, utiliza como referência a função-tarefa *task0* e tem prioridade 0. As outras duas utilizam como referência a função-tarefa *taskLed* e tem prioridade 2.

Perceba que, apesar das duas últimas utilizarem a mesma função-tarefa, elas acionam *leds* diferentes, graças ao argumento passado para função-tarefa (3 e 6).

No exemplo, o bit 7 da porta de entrada (GPIOA) possui uma chave que sinaliza qual o tipo de escalonamento será utilizado. No escalonamento por hardware, basta habilitar o escalonamento para que o IHM passe a controlar a troca de tarefas. Já no caso de escalonamento por software, além de escolher o tipo de escalonamento desejado, é preciso configurar a fonte de interrupção por contador. Em ambos os casos, o programa é desviado para tarefa 0 (*idle*) e passa aguardar a primeira ordem de troca de tarefas.

Quadro 17 – Função principal onde as tarefas são criadas e o escalonamento é configurado (arquivo plasma_rt.c).

```

404 //XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
405 //FUNÇÃO PRINCIPAL
406 //XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
407 int main()
408 {
409     //Criação de tarefas
410     //      função-tarefa    task    prio    arg
411     create_task_a(task0,      0,      0,      0);
412     create_task_a(taskLed,    1,      2,      3);
413     create_task_a(taskLed,    2,      2,      6);
414     //Ajuste do time slice (exclusivo para escalonamento por hardware)
415     TimeSlice(262144); //2^18 clocks = 10,48576ms
416     //Se a chave 7=1 roda o scheduler por hardware, caso contrário por software
417     if(read_input_pin(7))
418     {
419         puts("---by Hardware---");
420         EnableScheduler();
421         //Força o desvio para task0, tarefa idle.
422         task0();
423     }
424     else
425     {
426         puts("---by Software---");
427         //Inicializa o escalonador por software
428         //sw_scheduler_init();
429         sw_scheduler_init_v2();
430         //Setup de interrupção
431         EnableCounter();
432         //Habilita interrupções (coprocessador reg 12 bit 0)
433         OS_AsmInterruptEnable(0x01);
434         //Força o desvio para task0, tarefa idle.
435         task0();
436     }
437     for(;;);
438 }

```

Fonte: Autor.

No Apêndice A é descrito o ambiente de software utilizado para edição dos programas escritos em linguagem C e utilizados para validação do microcontrolador IHM-Plasma. Neste apêndice também é descrito o procedimento utilizado para compilação dos projetos e geração dos arquivos hexadecimais utilizados durante a síntese do FPGA, para geração da memória de programa do microcontrolador.

3.5 IMPLEMENTAÇÃO DO LEAIUTE DE UM ASIC DO IHM-PLASMA PARA FUTURA FABRICAÇÃO UTILIZANDO A TECNOLOGIA SIGE BICMOS DE 130 NM DA GLOBALFOUNDRIES

Esta seção traz os resultados da migração do projeto do microcontrolador IHM-Plasma, que foi criado e validado utilizando a tecnologia FPGA de hardware programável, para o leiaute de um circuito integrado de aplicação específica (ASIC). Para tal, foram utilizadas duas

ferramentas de software da Cadence (7) chamada de Encounter RTL Compiler (RC) e Encounter Digital Implementation (EDI), descritos na seção 2.6.

Apesar da fabricação deste componente não fazer parte do escopo deste projeto de doutorado, o desenvolvimento de seu leiaute permitiu a extração de informações valiosas quanto ao número de elementos lógicos, área e potência consumida.

Para migração do projeto original (testado em FPGA) para leiaute ASIC (*hard-core*), foram necessários ajustes na capacidade de memória e pinagem.

Dispositivos digitais, como microprocessadores e microcontroladores, fazem uso de grande quantidade de memória, seja ela RAM, ROM ou EEPROM. Nestes casos, tanto nos FPGAs como nos ASICs, existem formas especiais para declará-las, a fim de minimizar o uso de recursos e área do dispositivo.

No caso dos FPGAs da Intel (24), existem as chamadas Memory Megafunctions que são estruturas especiais nos CIs que permitem a criação de grandes memórias em diferentes configurações e de forma otimizada. Já no caso dos ASICs, que elaboram seus circuitos com base em portas lógicas elementares, as memórias consomem muito mais espaço na pastilha que os demais circuitos. Como exemplo, os 24 kB de memória RAM demandados pelo microcontrolador Plasma customizado ocupariam em torno de 16 vezes mais área que todo o resto do projeto. Comercialmente, empresas especializadas criam blocos de memórias otimizadas chamados de Propriedade Intelectuais (IPs) de Memórias. Eles possuem dimensões otimizadas, bem como capacidade e tempos de acesso adequados às necessidades de cada projeto (56).

Neste projeto, optamos por restringir o tamanho da memória de dados e da memória reservada para os TCBs em exatamente 1 kB para cada uma, suficiente para executar até 4 tarefas. Já a memória de programa foi substituída por uma memória ROM com um programa previamente armazenado² de 515 instruções *Assembly* ou 2060 bytes.

Quanto à pinagem do projeto, o circuito integrado ASIC foi limitado a 40 pinos para facilitar o encapsulamento, quando ele vier a ser fabricado. Os pinos são os seguintes:

- Alimentação (2 pinos): VDD e VSS;
- *Clock* (2 pinos): CLOCK_25 (entrada), CLOCK_25_OUT (saída barrada)
- Resete (2 pinos): RESET (entrada), RESER_OUT (saída barrada)
- UART (2 pinos): UART_RX (entrada) e UART_TX (saída);

² O código fonte em linguagem C, do programa gravado na memória ROM, está disponível no Apêndice B.

- *Port* de entrada (16 pinos): PORT_IN (pinos de 0 a 15 do projeto original);
- *Port* de saída (16 pinos): PORT_OUT (pinos de 0 a 15 do projeto original).

Os arquivos RTL utilizados para geração do leiaute foram os mesmos empregados no projeto testado no FPGA e escritos em linguagem VHDL. As duas únicas exceções são quanto aos arquivos IHM_plasma_ASIC.vhd (usado como *top level entity*) e rom.vhd (substituiu a RAM usada para memória de programa). O Quadro 18 ilustra a parte do *script* utilizado no RC, onde a árvore hierárquica do projeto está descrita.

Quadro 18 – Árvore hierárquica dos arquivos RTL dos blocos componentes do projeto e declarados no *script* utilizado pela ferramenta Encounter RTL Compiler.

```
# Baseline RTL
set RTL_LIST {
mlite_pack_mod.vhd
IHM_plasma_ASIC.vhd
    plasma_RT.vhd
        mlite_cpu_RT.vhd
            pc_next_RT.vhd
            mem_ctrl.vhd
            control.vhd
            reg_bank_duplo_RT.vhd
                reg_bank.vhd
            bus_mux.vhd
            reset_controller.vhd
            alu_mult_shifter.vhd
                alu.vhd
                mult.vhd
                shifter.vhd
        ram.vhd
        rom.vhd
        uart.vhd
        microkernel.vhd
            context_manager.vhd
            scheduler.vhd
            counter_down.vhd
}
```

Fonte: Autor.

3.5.1 Tecnologia de fabricação utilizada no ASIC do IHM-Plasma

Este projeto de pesquisa utilizou a tecnologia SiGe BiCMOS 8HP de 130 nm da GlobalFoundries (19) para a implementação do leiaute do ASIC desenvolvido (microcontrolador IHM-Plasma). A escolha desta tecnologia ocorreu por conta de ela ser uma das tecnologias atualmente disponibilizadas no programa educacional da MOSIS (MEP) (37). Sendo a MOSIS uma empresa que se especializou em prototipagem de circuitos integrados e fornece anualmente algumas rodadas gratuitas para produção de protótipos às instituições

registradas no MEP, usando o conceito de *wafers* com múltiplos projetos [*Multi-Project Wafers* (MPWs)].

Essa tecnologia de fabricação foi desenvolvida com o intuito de permitir o aumento da velocidade dos circuitos de radiofrequência, a redução do consumo de potência e a redução da área requerida pelo CI. Essas características são fundamentais para circuitos digitais complexos como o apresentado neste projeto. Ela é baseada em silício e germânio com nó tecnológico de 130 nm. Os circuitos desenvolvidos a partir dela podem trabalhar com tensão de alimentação de 1,2 V ou 2,5 V. E no caso da versão disponibilizada pelo MEP, é possível utilizar até 7 camadas de metal para realização do roteamento do circuito.

Demais detalhes sobre essa tecnologia e suas estruturas estão protegidos por acordo de confidencialidade assinado entre a MOSIS e o Centro Universitário FEI (37).

3.5.2 Leiaute do ASIC do IHM-Plasma e suas dimensões

A Figura 46 ilustra o leiaute final do circuito do microcontrolador IHM-Plasma, desenvolvido de acordo com a tecnologia SiGe BiCMOS 8HP de nó tecnológico de 130 nm. Destaque é dado para suas dimensões, inferiores a 2 mm cada, e a área, inferior a 2 mm².

Figura 46 – Leiaute do microcontrolador IHM-Plasma.

Fonte: Autor.

A Figura 47 ilustra o esquema elétrico equivalente ao leiaute gerado, com destaque para os pinos de entrada e saída do dispositivo.

Figura 47 – Esquema elétrico do IHM-Plasma (obtido da ferramenta Schematic Viewer do EDI).

Fonte: Autor.

3.5.3 Características geométricas, físicas e elétricas obtidas do leiaute do ASIC do IHM-Plasma

Após a geração do leiaute do IHM-Plasma, na Tabela 19 é possível observar toda hierarquia de entidades criadas durante a declaração dos arquivos VHDL que o compõe, complementada com a quantidade de *gates* (*standard cells* da tecnologia) e a área ocupada por cada instância divididas em *gates* e trilhas. Esses dados foram coletados em relatórios gerados pela IDE Encounter RTL Compiler utilizada para geração do arquivo *gate-level netlist*.

Tabela 19 – Hierarquia das entidades do leiaute gerado para o IHM-Plasma com a representação do consumo de *gates* (*standard cells*) e área do dispositivo.

Nível	Hierarquia de Entidades	Nome da Instância	Gates	Área dos Gates (μm^2)	Área das Trilhas (μm^2)	Área Total (μm^2)
0	IHM_plasma_ASIC		39.183	1.043.987	1.412.669	2.456.656
1	plasma_RT	plasma1	39.181	1.043.975	1.411.681	2.455.656
2	mlite_cpu_RT	u1_cpu_RT	6.879	167.318	257.828	425.146
3	pc_next_RT	u1_pc_next_RT	197	3.498	3.723	7.221
3	mem_ctrl	u2_mem_ctrl	282	6.342	5.491	11.833
3	control	u3_control	193	1.851	4.207	6.058
3	reg_bank_duplo_RT	u4_reg_bank_duplo_RT	4.221	122.678	172.827	295.505
4	reg_bank	u1_reg_bank	2.012	59.856	74.862	134.718
4	reg_bank	u2_reg_bank	2.012	59.856	74.862	134.718
3	bus_mux	u5_bus_mux	230	3.235	4.323	7.558
3	alu_mult_shifter	u678_alu_mult_shifter	1.665	27.128	47.382	74.510
4	alu	u1_alu	351	5.186	7.737	12.923
4	mult	u2_mult	889	16.980	27.435	44.416
4	shifter	u3_shifter	393	4.531	10.599	15.131
3	reset_controller	u10_reset_controller	14	288	332	620
2	rom	u2_rom	1.485	18.785	49.168	67.954
2	ram	u3_ram_dt	11.635	369.049	446.302	815.351
2	ram	u4_ram_rt	11.635	369.049	446.302	815.351
2	uart	u5_uart	256	4.923	6.851	11.773
2	microkernel	u7_microkernel	6.463	92.592	160.213	252.805
3	reset_controller	u1_reset_controller	14	288	332	620
3	context_manager	u2_context_manager	426	8.077	9.954	18.031
3	scheduler	u3_scheduler	3.935	49.713	100.518	150.230
3	counter_down	u4_counter_down_1	91	1.693	1.912	3.606
3	counter_down	u5_counter_down_2	91	1.693	1.912	3.606
3	counter_down	u6_counter_down_3	91	1.693	1.912	3.606
3	counter_down	u7_counter_down_4	91	1.693	1.912	3.606
3	counter_down	u8_counter_down_5	91	1.693	1.912	3.606
3	counter_down	u9_counter_down_6	91	1.693	1.912	3.606
3	counter_down	u10_counter_down_7	91	1.693	1.912	3.606
3	counter_down	u11_counter_down_8	91	1.693	1.912	3.606
3	(gerado pelo EDI)	csa_tree_eq_458_12_groupi	100	1.379	1.914	3.293
3	(gerado pelo EDI)	csa_tree_eq_465_12_groupi	98	1.350	1.872	3.222
3	(gerado pelo EDI)	inc_add_446_15_40	84	760	1.143	1.904

Fonte: Autor.

Na Tabela 19, é possível perceber o impacto que as memórias têm sobre a área total do circuito integrado. Se somarmos as áreas das duas memórias RAM (2 kB, instâncias u3_ram_dt

e u4_ram_rt) com a área da memória ROM (2060 B, instância u2_rom) e com as áreas dos dois bancos de registradores internos da CPU (256 B, instâncias u1_reg_bank e u2_reg_bank) chegaremos a uma área de 1.968.092 μm^2 , que representa 80% da área total do CI (*gates* + trilhas). Essa foi a razão determinante para que o projeto convertido de FPGA para ASIC fosse ajustado de forma a utilizar apenas a quantidade necessária de memória para rodar um programa de demonstração com 4 tarefas.

É importante ressaltar, que as áreas informadas na Tabela 19 são as áreas mínimas teóricas. Durante o processo de disposição de cada uma das entidades (*placement*) foi definido que a área reservada para cada uma delas seria calculada com base em um fator de ocupação máximo de 60%. Isso quer dizer que, a área real de cada um dos circuitos roteados é 66,7% maior que o valor mínimo calculado ($\text{área mínima}/60\% = \text{área total}/100\%$).

Dando continuidade a esse estudo, foi feita a comparação entre as porcentagens relativas de recursos utilizados no projeto IHM-Plasma considerando sua síntese feita em FPGA (quantidade de LCs e bits de memória) e o leiaute criado para implementação de um ASIC (quantidade de *gates*). A Tabela 20 representa os resultados obtidos desconsiderando os recursos necessários para implementação das memórias, uma vez que as capacidades e tipos implementados foram diferentes.

Tabela 20 – Percentual de recursos de hardware utilizados para implementação do IHM-Plasma em FPGA e ASIC.

Bloco	FPGA (LCs combinacionais)	FPGA (LCs de registradores e bits de memória)	ASIC (<i>gates</i>)
CPU	34,70%	38,60%	35,80%
IHM	53,50%	29,80%	47,50%
IRB adicional	10,40%	29,60%	14,80%
UART	1,40%	2,00%	1,90%

Fonte: Autor.

É possível observar, na Tabela 20, que em ambos os casos o IHM consome aproximadamente 50% dos recursos do microcontrolador, desconsiderando as memórias.

A última análise feita nesta seção, analisou o consumo de potência do microcontrolador IHM-Plasma convertido para um ASIC. Os dados apresentados na Tabela 21 foram coletados em relatórios gerados pela IDE Encounter RTL Compiler utilizada para geração do arquivo

gate-level netlist e são baseados nas informações da tecnologia utilizada na elaboração do leiaute.

Tabela 21 – Análise do consumo de potência estática e dinâmica do ASIC correspondente ao IHM-Plasma.

Nível	Hierarquia de Entidades	Nome da Instância	Estática	Dinâmica			
			Fuga (mW)	Interna (mW)	Trilhas (mW)	Chaveamento (mW)	Relação (%)
0	IHM_plasma_ASIC		0,06	7,58	7,32	14,91	100%
1	plasma_RT	plasma1	0,06	7,58	5,50	13,09	88%
2	mlite_cpu_RT	u1_cpu_RT	0,02	1,27	1,11	2,38	16%
2	rom	u2_rom	0,00	0,04	0,10	0,14	1%
2	ram	u3_ram_dt	0,02	2,91	1,88	4,80	32%
2	ram	u4_ram_rt	0,02	2,92	1,82	4,74	32%
2	microkernel	u7_microkernel	0,00	0,44	0,59	1,03	7%

Fonte: Autor.

Observa-se na Tabela 21, a representação dos principais blocos construtivos do microcontrolador IHM-Plasma. Onde a potência estática é igual a potência de fuga de energia dos componentes, com valor total de 0,06 mW. E a potência dinâmica total, chamada de potência de chaveamento, é igual a soma das potências dissipadas internamente pelos componentes do circuito e as potências dissipadas nas trilhas por onde a corrente circula, com valor total de 14,91 mW. A potência dinâmica depende da frequência de chaveamento (*clock*) do circuito, que neste caso foi ajustada para 25 MHz.

Assim como no estudo de área do circuito, as memórias são responsáveis pela maior parte da potência dissipada pelo microcontrolador, sendo que as duas memórias RAM de 1kB cada (instâncias u3_ram_dt e u4_ram_rt) sozinhas consomem 64% de toda potência dissipada. Outro ponto importante a ser observado, é que o *microkernel* proposto (IHM) consome apenas 7% (1,03 mW) de toda a potência.

4 RESULTADOS EXPERIMENTAIS OBTIDOS DO IHM-PLASMA IMPLEMENTADO NO KIT DE DESENVOLVIMENTO DE1-SOC DA Terasic

Neste capítulo são apresentados os resultados experimentais obtidos a partir do estudo do microcontrolador IHM-Plasma implementado em FPGA [kit de desenvolvimento DE1-SoC da Terasic (51) (vide Figura 14)], que apresenta como principal característica a presença de um *microkernel* por hardware, ideal para sistemas de tempo real e baseados em tarefas. Nas próximas seções são apresentadas análises dinâmicas/funcionais comparativas entre o IHM-Plasma e soluções convencionais totalmente baseadas em software para o escalonamento e despacho de tarefas.

4.1 CARACTERIZAÇÃO ELÉTRICA EXPERIMENTAL DO TEMPO GASTO PARA A REALIZAÇÃO DAS TROCAS DE TAREFAS

Neste estudo avaliamos o tempo gasto para realização das trocas de tarefas considerando um *microkernel* escrito por software e com a utilização do *microkernel* por hardware (IHM).

Como visto na seção 3.3, quando o escalonamento é realizado por software, o uso de interrupções de hardware é primordial. Podemos dividir o tempo gasto para o escalonamento em três etapas: (i) *backup* do contexto da tarefa em curso; (ii) execução do algoritmo de escalonamento para definição da próxima tarefa e; (iii) restauração do contexto da próxima tarefa.

Para executar essas verificações, a sub-rotina `interrupt_service_routine` do arquivo `boot.asm` (Quadro 9) foi alterada de forma a ser possível identificar o tempo gasto em cada uma das etapas, através do uso de um osciloscópio digital (modelo MIN DS0201). Para isso, um pino de saída do processador foi ativado e desativado de forma a que a forma de onda gerada representasse o período de cada uma das etapas (tempo entre duas bordas da onda). A Figura 48 ilustra a medida feita.

Figura 48 – Fotografia da tela de um osciloscópio com a representação quantitativa dos tempos gastos para execução das etapas necessárias para uma troca de tarefas síncronas e assíncronas realizada por um *microkernel* por software.

Fonte: Autor.

Para que as medidas dos períodos envolvidos fossem realizadas com precisão suficiente, o *clock* do processador foi diminuído do padrão de 25 MHz para apenas 1 kHz. Isso implicou em um ciclo de *clock* com período de exatamente 1 ms. De forma a comprovar os períodos medidos, medidas sucessivas foram realizadas. A Tabela 22 representa os resultados em ciclos de *clock* e convertidos em microssegundos considerando *clock* de 25 MHz. A representação das etapas de uma troca de tarefa em função do número de ciclos de *clock* é uma figura de mérito genérica, uma vez que independe da frequência do *clock* utilizada pelo microprocessador.

Tabela 22 – Períodos medidos para execução das etapas necessárias para uma troca de tarefas realizada por um *microkernel* por software.

Etapa	Período (ciclos de <i>clock</i>)	Período (μs) para <i>clock</i> de 25 MHz
<i>Backup</i> (a)	71	2,84
<i>Restore</i> (b)	64	2,56
Troca de contexto (a+b)	135	5,4
Algoritmo	101	4,04
Total	236	9,44

Fonte: Autor.

Procedimento semelhante ao apresentado para o software foi realizado utilizando-se o escalonamento por hardware. Porém, neste caso, apenas o tempo entre a paralização da CPU para troca dos bancos de registradores e a retomada do processamento foi medido. Para isso,

um programa com duas tarefas de mesma prioridade, porém com comportamentos distintos foi utilizado. Nele, uma das tarefas sofre apenas trocas síncronas (preempção, troca no final do *time slice*) e a outra tarefa sofre apenas trocas assíncronas através de pedido de *sleep* por 1 *time slice*. Os tempos de paralização da CPU foram medidas através do sinal `switch_out_RT`, que como descrito na seção 3.2.4.1, sinaliza que a CPU está em estado *stopped*. A Figura 49 representa uma amostragem das medidas feitas.

A CPU inicia a carga de uma nova instrução (ciclo de *fetch*) somente após a desativação do sinal `switch_out_RT`, para que no ciclo de *clock* seguinte, a próxima instrução seja executada. Essa afirmação é feita, uma vez que o microcontrolador IHM-Plasma está trabalhando com *pipeline* de dois níveis (*fetch* e *execute*).

No caso da tarefa síncrona, a paralização é de exatamente 1 ciclo de *clock*, porém ainda é necessário mais um ciclo para que a próxima instrução entre em execução, totalizando assim dois ciclos de *clock* de interrupção da CPU. Considerando a frequência de *clock* de 25 MHz, a paralização total foi de 80 ns.

Para tarefa assíncrona, a paralização medida foi de 72 ciclos de *clock*, totalizando 73 ciclos entre a última execução de instrução e a próxima. Da mesma forma, para frequência de *clock* de 25 MHz, a paralização total foi de 2,92 μ s.

Figura 49 – Fotografia da tela de um osciloscópio com a representação quantitativa dos tempos gastos para execução de uma troca síncrona de tarefas e uma troca assíncrona de tarefas com base na monitoração do sinal `switch_out_RT` e realizadas pelo *microkernel* por hardware proposto (IHM).

Fonte: Autor.

A Tabela 23 representa um resumo das medidas feitas e o comparativo de ganho de desempenho da solução por hardware frente a solução por software. Vale ressaltar que na

solução por software não há diferença entre os tempos para trocas síncronas e assíncronas de tarefas.

Tabela 23 – Medidas feitas para análise do tempo necessário para troca de tarefas, em que os ganhos de desempenho e a redução nos períodos do *microkernel* por hardware (IHM) são calculados em relação ao *microkernel* por software.

Troca de tarefa	Período (ciclos de <i>clock</i>)	Período (μ s) <i>clock</i> de 25MHz	Ganho de desempenho (vezes)	Redução no período (%)
Software	236	9,44	-	-
Hardware Síncrona	2	0,08	118,0	99,15
Hardware Assíncrona	73	2,92	3,23	69,07

Fonte: Autor.

As medidas coletadas permitem observar um ganho de desempenho do microcontrolador Plasma com *microkernel* por hardware (IHM-Plasma), tanto nas trocas assíncronas de tarefas (3 vezes mais rápidas) como nas síncronas (118 vezes mais rápidas). Quando analisado do ponto de vista de redução no período para tal, observa-se uma redução entre 69% e 99% nos tempos.

É importante destacar que o ganho obtido levou em consideração um algoritmo extremamente simplificado para o escalonamento por software e totalmente insensível a variações nas prioridades das tarefas após o início do escalonamento. Podendo ele ser classificado como estático. Ao passo que, o algoritmo implementado em hardware, além de possuir essa sensibilidade, é capaz de avaliar dentre as tarefas com mesma prioridade, aquela que se encontra há mais tempo em espera por execução, sendo classificado como dinâmico.

As observações feitas levam a hipótese de que, em sistemas baseados em tarefas, quanto maior for a frequência de trocas de tarefas, mais a solução proposta por hardware tende a se destacar. As próximas duas seções buscam verificar essa hipótese.

4.2 CARACTERIZAÇÃO EXPERIMENTAL DO *TIME SLICE* EFETIVO EM FUNÇÃO DO TEMPO GASTO PARA REALIZAÇÃO DAS TROCAS DE TAREFAS (4.1)

Esta seção visa comprovar experimentalmente a hipótese apresentada na seção anterior, de que em sistemas baseados em tarefas, quanto maior for a frequência de trocas de tarefas, mais a solução proposta por hardware tende a se destacar. Para isso, uma nova figura de mérito,

chamada de *time slice* efetivo ($Ef\%$), é proposta. Ela expressa a fração percentual de um *time slice* que a CPU tem para executar uma tarefa, conforme representado pela Equação (4).

$$Ef\% = \frac{\text{Time Slice} - \text{Tempo para troca de tarefa}}{\text{Time Slice}} \cdot 100\% \quad (4)$$

Os tempos medidos para realização de trocas síncronas e assíncronas de tarefas associados a valores teóricos de *time slice* permitiram determinar o $Ef\%$ sob diferentes condições. A Tabela 24 representa os resultados deste estudo considerando dois cenários:

1. Apenas trocas síncronas de tarefas (preempção);
2. Uma troca síncrona e uma assíncrona por *time slice*.

A Tabela 24 representa *time slices* entre 2^{18} a 2^8 ciclos de *clock*. Considerando a frequência padrão de operação do microcontrolador Plasma (25 MHz), observamos períodos de aproximadamente 10 ms a 10 μ s.

Tabela 24 – *Time slice* efetivo ($Ef\%$) em função do período do *time slice*.

<i>Time Slice</i> (ciclos de <i>clock</i>)	<i>Time Slice</i> (ms) com um <i>clock</i> = 25 MHz	Cenário 1			Cenário 2		
		Software	Hardware	Ganho	Software	Hardware	Ganho
2^{18}	10,486	99,9	100,0	0,1	99,8	100,0	0,2
2^{17}	5,243	99,8	100,0	0,2	99,6	99,9	0,3
2^{16}	2,621	99,6	100,0	0,4	99,3	99,9	0,6
2^{15}	1,311	99,3	100,0	0,7	98,6	99,8	1,2
2^{14}	0,655	98,6	100,0	1,4	97,1	99,5	2,5
2^{13}	0,328	97,1	100,0	2,9	94,2	99,1	5,1
2^{12}	0,164	94,2	100,0	6,1	88,5	98,2	11,0
2^{11}	0,082	88,5	99,9	12,9	77,0	96,3	25,2
2^{10}	0,041	77,0	99,8	29,7	53,9	92,7	71,9
2^9	0,020	53,9	99,6	84,8	7,8	85,4	992,5
2^8	0,010	7,8	99,2	1170,0	-84,4	70,7	$\rightarrow \infty$

Fonte: Autor.

Em ambos os cenários analisados, o escalonamento por software apresenta grande deterioração do $Ef\%$ à medida que o *time slice* é reduzido. O pior caso observado, ocorreu no cenário 2 com $Ef\% = -84,4\%$. O valor negativo significa que o período de *time slice* não é suficiente para realização das trocas de tarefas demandadas. Por outro lado, o escalonamento

realizado por hardware, se mantém estável no cenário 1, com deterioração inferior a 1% ($E_f\% = 99,2\%$ no pior caso). Nesta mesma condição, a solução por software já se torna inviável, com $E_f\% = 7,8\%$ apenas.

No cenário 2, quando trocas síncronas e assíncronas de tarefas ocorrem a cada *time slice*, o *microkernel* por hardware proposto apresenta excelente resposta até 2^{10} ciclos de *clock/time slice* ($E_f\% = 92,7\%$). Porém, mesmo no caso mais crítico, a degradação calculada não atingiu 30% ($E_f\% = 70,7\%$). Essa degradação se deve principalmente pelas trocas assíncronas de tarefas, quando o *microkernel* por hardware não é capaz de prever o instante da solicitação de troca feita pela CPU. Nestes casos, a CPU é parada até que o contexto da próxima tarefa seja carregado no IRB disponível ao bloco gerenciador de contexto do IHM e a inversão dos IRBs seja efetuada.

Os ganhos calculados mostram que o *microkernel* por hardware proposto tem desempenho sempre superior ao implementado em software. Resultados expressivos são observados quando o *time slice* se torna inferior a 2^{12} ciclos de *clock*. Esses resultados apontam que a solução proposta, pode oferecer vantagens para sistemas críticos e com *deadlines* curtos. Neles, assim que uma tarefa prioritária é ativada (passa para estado *ready*), o tempo disponível para que sua execução seja completada é relativamente curto, e quanto mais rápido o escalonador fizer sua detecção e a troca de tarefas ocorrer, maiores as chances de sucesso na operação.

4.3 CARACTERIZAÇÃO EXPERIMENTAL DE DESEMPENHO EM FUNÇÃO DO *TIME SLICE* EM CICLOS DO CLOCK DO PROCESSADOR

Neste estudo avaliamos comparativamente o impacto do uso de escalonamento por hardware e por software, considerando diferentes quantidades de tarefas, com a mesma prioridade e diferentes períodos de *time slice*. Como figura de mérito, foi medido o número de execuções completas por segundo de uma tarefa de controle, que realizava a função de um simples pisca-pisca de *leds*. Esta tarefa foi baseada em um laço de programa limitado por uma variável, que após incrementada de 0 a 100.000, permitia a inversão do nível lógico de dois pinos de saída do FPGA. Um deles conectado eletricamente a um *led* e o outro conectado a entrada de um osciloscópio digital (modelo MIN DS0201).

O Quadro 19 ilustra os dois modelos de funções-tarefas escritas usando linguagem C. Como é identificado através de seus nomes, uma serve para o controle e a outra para apoio.

Foram declaradas diferentes quantidades de instâncias da função-tarefa de apoio (0, 1, 4, 14 e 31) e apenas uma instância da função-tarefa de controle, totalizando diferentes números de tarefas ativas (1, 2, 5, 15 e 32). A Figura 50 ilustra o gráfico da frequência de execução completa da tarefa de controle (baseada na função `taskControle` mostrada no Quadro 19) em função do *time slice*, parametrizada em função no número de tarefas ativas e da forma utilizada para o escalonamento. No escalonamento por software, utilizou-se um algoritmo extremamente simples baseado em uma lista circular indexada para definir a próxima tarefa, desconsiderando-se os níveis de prioridade (função `sw_scheduler` descrita na seção 3.4.3.2).

Para que esta figura de mérito fosse independente da frequência de *clock* da CPU (neste caso, igual a 25 MHz), o eixo do *time slice* (abscissa) foi representado em função do número de ciclos de *clock* ao invés do período. Sua variação foi de 2^8 a 2^{18} ciclos de *clock* (10,24 μ s a 10,48576 ms).

Quadro 19 – Funções-tarefas para teste de desempenho de processamento (arquivo `plasma_rt.c`).

```

808 //XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
809 //TESTE HD/SW SCHEDULER COM n TAREFAS E n TIME SLICES (TESTE DE DESEMPENHO)
810 //XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
811 int taskApoio(int v)
812 {
813     puts("Task "); puts(itoa2(v));
814     putchar('\r'); putchar('\n');
815     while(1)
816     {
817         set_output_pin(1);          //LED 1
818         clear_output_pin(1);        //LED 1
819     }
820 }
821 int taskControle(int v)
822 {
823     int i;
824     puts("Task "); puts(itoa2(v)); puts(" controle");
825     putchar('\r'); putchar('\n');
826     while(1)
827     {
828         //Pisca led
829         for(i = 100000;i>0;i--);
830         set_output_pin(0);          //LED 0
831         set_output_pin(15);         //GPIO_0 pino 20
832         for(i = 100000;i>0;i--);
833         clear_output_pin(0);        //LED 0
834         clear_output_pin(15);       //GPIO_0 pino 20
835     }
836 }

```

Fonte: Autor.

Figura 50 – Frequência de execução de uma tarefa de controle (taskControle) em função do *time slice*, considerando-se diferentes números de tarefas ativas, quando o escalonamento é realizado por algoritmo em hardware (IHM) e em software (abordagem tradicional).

Fonte: Autor.

Conforme mostrado na a Figura 50, à medida que a quantidade de tarefas ativas aumenta, a frequência de processamento da tarefa monitorada diminuiu proporcionalmente (frequência $\cong 62,5 / \text{número de tarefas}$). Nesta experiência, todas as tarefas sofreram preempção para dar lugar à próxima tarefa.

Considerando o escalonamento sendo feito por hardware e o mesmo número de tarefas, podemos observar que a frequência de execução da tarefa monitorada não é afetada mesmo quando o *time slice* é muito curto. Isso acontece porque o *microkernel* por hardware interrompe a CPU somente quando é necessário executar a troca dos bancos internos de registradores (IRBs) [vide Figura 42 (A)]. No entanto, quando a execução do escalonamento é realizada por software, o oposto é observado a partir de 2^{12} (4.096) ciclos de *clock* (Região I da Figura 50). Isso pode ser justificado devido ao tempo gasto pelo software para executar a mudança de contexto e decidir qual a próxima tarefa a ser executada, conforme mostrado na Tabela 22. Além disso, mesmo quando apenas uma tarefa é executada, sua frequência de execução é drasticamente afetada, uma vez que a CPU continua sendo interrompida pelo escalonador para verificar se essa tarefa é de fato a próxima tarefa a ser executada. O que não ocorre no escalonamento por hardware, uma vez que antes de interromper a CPU, o bloco de temporização e controle verifica se a próxima tarefa informada pelo escalonador é diferente da atual.

Vale ressaltar que, à medida que a complexidade do *microkernel* por software aumenta, maior será o tempo gasto para troca de tarefas, fazendo com que o período crítico de *time slice* observado (2^{12} ciclos de *clock*) seja deslocado para direita no gráfico da Figura 42. Consequentemente, tornando a solução por software mais sensível a variações no *time slice*.

Considerando-se que a redução do *time slice* implica no atendimento de uma maior quantidade de tarefas em sistemas de tempo real, podemos melhorar o desempenho de processamento em termos de latências, prazos, previsão e execução das tarefas. A Tabela 25 é baseada nos dados usados na representação da Figura 50 para o escalonamento de 32 tarefas. Ela representa a variação na frequência de execução completa da tarefa de controle em função da redução do *time slice*. Esta análise usa como referência a frequência medida para o período mais longo de *time slice* (2^{18} ciclos de *clock*), quando o $Ef\%$ tende a 100%. Dessa forma, os resultados são apresentados em porcentagem de variação do *time slice* efetivo ($\Delta Ef\%$) e levam em conta a faixa de valores mais crítica para os escalonadores, de 2^8 (256) a 2^{14} (16.384) ciclos de *clock/time slice*.

Tabela 25 – Variação no *time slice* efetivo ($\Delta Ef\%$) em função do número de ciclos de *clock* por *time slice* e do tipo de *microkernel*.

Time Slice (ciclos de clock)	2^{14}	2^{13}	2^{12}	2^{11}	2^{10}	2^9	2^8
$\Delta Ef\%$ por Software	0,0	-1,6	-3,1	-7,8	-17,2	-38,0	-79,4
$\Delta Ef\%$ por Hardware	-1,5	-1,5	-1,5	0,0	0,0	-3,0	-3,0

Fonte: Autor.

Com base na Tabela 25, podemos verificar que, o uso da estrutura de *microkernel* por hardware proposta é notavelmente capaz de manter seu desempenho de processamento, mesmo quando o *time slice* é reduzido para apenas 2^8 ciclos de *clock*. Neste caso, observa-se uma redução máxima de apenas 3% no seu desempenho frente a redução de 79,4% apresentada pelo sistema puramente baseado em software, ou seja, um ganho de desempenho de processamento de 76,4%. Além disso, considerando um *clock* de 25 MHz, o processador Plasma associado ao *microkernel* por hardware proposto pode executar até 97.000 ($\cong 4.000/\text{MHz}$) trocas de tarefas por segundo praticamente sem perder o seu desempenho máximo de processamento (apenas 3%: obtido no ponto A na Figura 50).

Outra análise, decorrente da Tabela 25, é feita considerando os valores de *time slice* em que ambas as soluções (*microkernel* por hardware e software) apresentam comportamento

equivalente e com pouca degradação do $Ef\%$. Esses pontos foram destacados em tom cinza na Tabela 25. A partir deles, foi gerada a Tabela 26, que apresenta as máximas frequências de trocas de tarefas, bem como analisa o ganho em frequência do *microkernel* por hardware proposto (IHM) frente a abordagem tradicional por software.

Tabela 26 – Máxima troca de tarefas considerando a mesma degradação do $Ef\%$.

<i>Microkernel</i>	<i>Time Slice</i> ($\Delta Ef\% = -3\%$)	Trocas de Tarefas (freq. = 25 MHz)	Trocas de Tarefas/MHz	Ganho
Software	2^{12}	4.000	160	-
IHM	2^8	97.000	4.000	24 vezes

Fonte: Autor.

Através da Tabela 26, é possível perceber que, a solução proposta oferece um ganho de 24 vezes na quantidade máxima de trocas de tarefas por segundo frente a abordagem convencional (software). Esse ganho tão expressivo, pode ser revertido em maior liberdade e flexibilidade aos programadores para desenvolverem seus sistemas, onde o número máximo de trocas de tarefas e período do *time slice*, passam a assumir limites diferentes dos trabalhados atualmente.

Do ponto de vista de economia de energia, a Equação (5) mostra que existe uma relação direta entre a potência dissipada de um circuito e sua frequência de operação (46).

$$P_D = f \times C \times V_{DD}^2 \quad (5)$$

Onde P_D é a potência dissipada, f é a frequência de chaveamento (operação), C é capacitância do dispositivo e V_{DD} é sua tensão de alimentação.

Dessa forma, considerando os pontos A e B da Figura 50, para se obter um desempenho equivalente entre as duas soluções, é possível reduzir a frequência de *clock* do processador customizado em aproximadamente 5 vezes ($A/B = 1,92 \text{ Hz}/0,4 \text{ Hz}$), o que resultaria automaticamente em uma redução de também 5 vezes na potência dissipada pelo microcontrolador. A mesma relação foi observada para todas as outras quantidades de tarefas ativas estudadas. Isso nos permite afirmar que, o IHM-Plasma é mais eficiente do ponto de vista energético em relação ao microcontrolador Plasma executando um *microkernel* por software, considerando-se um mesmo desempenho de processamento.

4.4 CARACTERIZAÇÃO EXPERIMENTAL DE DESEMPENHO DO IHM-PLASMA OPERANDO COMO O GERENCIADOR DO JOGO *SPACE INVADERS*

Esta nova proposta (IHM) possibilita o desenvolvimento de uma série de aplicações de sistemas baseados em tarefas, que são inviáveis nos sistemas puramente baseados em software. Devido principalmente às restrições quanto a frequência máxima para troca de tarefas ($< 100 \mu s$). Como exemplo destacam-se os sistemas que demandam por tarefas com intenso processamento de *I/Os*, onde o *deadline* é muito curto e não atendimento pode representar falha no sistema como um todo. Seguem alguns exemplos práticos:

- Virtualização de periféricos (UART, PWM, ...);
- Codificação/decodificação de áudio e sinais infravermelhos;
- Controle de múltiplos servo motores;
- Controle de display gráfico.

Nesta seção, apresentamos um exemplo de aplicação no qual kit de desenvolvimento DE1-SoC da Terasic (51) passou a se comportar como um console *Arcade* controlado pelo IHM-Plasma que executa o clássico jogo *Space Invaders*.

A Figura 51 ilustra seu diagrama funcional, onde é possível notar que o controlador (IHM-Plasma) se comunica com um *joystick* através de protocolo infravermelho padrão NEC (aplicativo especialmente desenvolvido para celular) e com um computador via UART a 57600 bps. O computador funciona como um controlador de vídeo que apenas reproduz o *sprites* do jogo (imagens que compõe o jogo) na tela, de acordo com os comandos enviados pelo console. Todas as regras do jogo são processadas no console, ou seja, no aplicativo baseado em tarefas criado para o IHM-Plasma.

Figura 51 – Diagrama funcional da aplicação do IHM-Plasma como controlador de um console *Arcade*.

Fonte: Autor.

Nesta aplicação foram criadas três tarefas:

- *taskIR*: usada para recepção dos comandos enviados ao console pelo *joystick* (celular). Como a recepção é um processo que depende do usuário pressionar alguma tecla no *joystick*, essa tarefa precisa ser executada a cada *time slice* para verificar se existe uma nova *string* infravermelha para ser recebida. Por conta disso, esta é a tarefa com maior prioridade (nível 4);
- *taskFrame*: usada para tratar os comandos recebidos, definir a posição dos *sprites* na tela, verificar as regras do jogo, como impactos entre os tiros e os alienígenas, e enviar ao computador o novo *frame* a ser apresentado na tela. Essa tarefa é executada 30 vezes por segundo (aproximadamente uma vez a cada 33 ms) e tem prioridade intermediária (nível 2);
- *task0*: essa é a tarefa *idle* do sistema e é usada para calcular o nível relativo de utilização da CPU. Sua prioridade é mínima, ou seja, ela é executada apenas durante o tempo livre de um *time slice* (nível 0).

O link para o vídeo com a demonstração da operação deste console *Arcade* pode se encontra na referência (10).

Assim como nos outros experimentos deste capítulo, o desempenho do IHM-Plasma utilizando seu *microkernel* por hardware (IHM) foi comparado com o desempenho do *microkernel* executado por software. Esta aplicação foi testada com *time slice* de 81,92 μ s, onde foi possível observar que a sobra de tempo de processamento da CPU quando executando o *microkernel* por hardware é 38% maior que a sobre observada quando a CPU executou o *microkernel* por software. Este resultado, confirma de forma prática o melhor desempenho do IHM-Plasma em sistemas com *time slice* curtos.

Todos os programas criados para testes do microcontrolador IHM-Plasma estão disponíveis em formato digital e podem ser acessados através do link na referência (12).

5 CONSIDERAÇÕES FINAIS, CONCLUSÕES, CONTRIBUIÇÕES E TRABALHOS FUTUROS

O texto desta tese foi iniciado com uma breve descrição do cenário atual dos sistemas embarcados, em que foi destacada a importância do uso de hardware programável para o atendimento de demandas específicas e busca por maior desempenho dos sistemas baseados em tarefas. Passamos por um estudo de propostas publicadas na literatura que se enquadram neste contexto. Porém, foi observado que a maior parte das soluções empregadas comercialmente, utilizam RTOSs totalmente baseados em software, que são executados pelas CPUs dos processadores. Esse tipo de solução se baseia na execução de múltiplas tarefas alternadas ao longo do tempo, que têm sua ordem de execução definida por uma unidade chamada de escalonador (*scheduler* em inglês). Foi nesse ponto que encontramos a motivação desse projeto de pesquisa. Sabendo-se que o gargalo desses sistemas se localiza justamente no momento em que é necessário realizar uma troca de tarefas, quando a CPU deixa de executar tarefas para executar o algoritmo do escalonador e realizar a troca do contexto de seus registradores internos. O tempo gasto para executar essas ações se torna mais crítico à medida que os períodos mínimos entre trocas de tarefas (*time slice*) são reduzidos, refletindo diretamente em perda de desempenho da CPU.

O trabalho aqui apresentado teve como objetivo, preencher essas lacunas através do desenvolvimento de um *microkernel* por hardware eficientemente integrado a uma CPU. O microcontrolador, resultado deste desenvolvimento, foi chamado de IHM-Plasma. Onde, a sigla IHM representa a expressão em inglês *Interlocked Hardware Microkernel* e Plasma é o nome do microcontrolador utilizado como referência para o desenvolvimento desta pesquisa.

O *microkernel* por hardware proposto, também chamado de IHM, tem como característica principal reduzir o tempo necessário para troca de tarefas, através da execução de um escalonador, um gerenciador de contexto das tarefas e um bloco de temporização e controle totalmente criados em hardware que operam paralelamente à execução das tarefas pela CPU. A arquitetura do microcontrolador proposto permite que uma troca de tarefas possa ocorrer em até 2 ciclos de *clock*, o que é pelo menos 118 vezes mais rápido daquele observado através de uma solução equivalente por software. Para que isso fosse possível, o conjunto de registradores internos à CPU foi replicado uma única vez e uma parte da memória de dados foi compartilhada com o *microkernel* por hardware para o armazenamento do contexto das tarefas.

O *microkernel* por hardware proposto consome aproximadamente a mesma quantidade de recursos que a CPU. Isso foi observado, tanto durante sua síntese para testes em FPGA, como também durante o projeto de seu leiaute para geração de um ASIC. Por outro lado, a análise da potência dinâmica do dispositivo proposto considerando-se a solução por ASIC, mostrou que o consumo da potência do IHM é 56% menor que a da CPU. Essa característica pode qualificá-lo como uma alternativa viável para se ter ganho no desempenho do sistema como um todo, frente a alternativa de utilização de múltiplos núcleos programáveis.

Outra característica importante do IHM-Plasma é que, mesmo em sistemas que demandam *time slices* muito curtos (atualmente menores que 100 μ s), o tempo efetivo de processamento das tarefas pela CPU é muito pouco afetado. Experimentalmente verificamos uma redução do *time slice* efetivo de apenas 3% com o uso do IHM, frente a uma redução de 79% quando um *microkernel* por software é executado com o IHM desabilitado. Estes testes foram executados utilizando-se o kit de desenvolvimento DE1-SoC da Terasic, considerando-se um *time slice* de 2^8 ciclos de *clock* e uma frequência de *clock* da CPU de 25 MHz.

Quando as duas alternativas de *microkernel* foram comparadas, operando com a mesma degradação do *time slice* efetivo (3%), observou-se que o IHM é capaz de executar 24 vezes mais trocas de tarefas por segundo em relação a abordagem por software. Isso amplia enormemente as possibilidades de desenvolvimento para os projetistas de sistemas baseados em tarefas, que passam a poder explorar muito mais o recurso de trocas rápidas e intensas de tarefas sem se preocupar com a perda de desempenho da CPU.

Acreditamos que essa abordagem inovadora, possa abrir novas perspectivas de aplicações em sistemas baseados em tarefas que precisam de *time slices* curtos, tais como aqueles relacionados à comunicação serial de dados em alta velocidade, ao controle de múltiplos servo motores, a codificação/decodificação de áudio, ao processamento de regras de jogos, etc. Nesses casos, uma outra possibilidade é a utilização do IHM-Plasma com uma frequência de *clock* menor, sem alterar o desempenho de um sistema baseado em tarefas equivalente, que utiliza um RTOS ou um *microkernel* por software. Consequentemente, isso acarreta numa significativa redução no consumo de energia elétrica (excelente para as aplicações que envolvem equipamentos eletrônicos móveis).

Cabe ainda aqui ressaltar que, a proposta apresentada reduz drasticamente a quantidade de código necessário para implementação de um RTOS para o IHM-Plasma, uma vez que toda camada de software correspondente ao *microkernel* foi migrada para hardware. Além disso, simplifica a forma de programação de um sistema de tempo real, onde apenas é necessário

declarar as tarefas a serem executadas e configurar um conjunto de 8 registradores para ativar o escalonamento de tarefas (neste caso até 32 tarefas).

A seguir são listadas as contribuições científicas geradas por este trabalho:

- I. Retirar parte das atribuições de um sistema multitarefas do software e delegá-la ao hardware de modo a permitir melhora no desempenho da CPU.
- II. Esta solução não demanda por alteração no conjunto de instruções da CPU e faz uso apenas de registradores com funções especiais para sua configuração de seu funcionamento.
- III. Implementa apenas o hardware necessário para que o *microkernel* proposto funcione paralelamente à CPU, realizando de forma otimizada o escalonamento e o despacho de tarefas. Sendo adequado tanto para em dispositivos *soft-cores* como comerciais (*hard-cores*).
- IV. O *microkernel* proposto foi minuciosamente interconectado à CPU. O que permite a criação de programas baseados em tarefas com diferentes prioridades e executá-las automaticamente de acordo com o *time slice* especificado, suas prioridades e o estado atual de cada uma.
- V. O *microkernel* proposto opera permanentemente em paralelo à CPU, cuidando de todo processo de escalonamento e troca de tarefa. Ele trabalha de forma antecipativa preparando o contexto da próxima tarefa antes que a CPU seja interrompida, essa é uma das razões para se ter um ganho de desempenho no sistema baseado em tarefas. Do ponto de vista da CPU, é como se ela saltasse da execução de uma tarefa diretamente para execução de outra.
- VI. Considerando a frequência do *clock* de 25 MHz, o *microkernel* proposto é capaz de realizar mais de 142 mil trocas de tarefas por segundo, ou seja, uma a cada 7 μ s. O que é três ordens de grandeza abaixo do convencional.
- VII. As medidas coletadas apontam que o IHM-Plasma, quando escalonando tarefas por hardware, apresenta uma redução no tempo de troca de tarefas de 69% para troca assíncronas e 99% para trocas síncronas quando comparado com um *microkernel* equivalente executado por software.
- VIII. Os ganhos calculados mostram que o *microkernel* por hardware proposto tem desempenho sempre superior ao implementado em software. Resultados expressivos são observados quando o *time slice* se torna inferior a 2^{12} ciclos de

clock. Esses resultados apontam que a solução proposta, é vantajosa em sistemas críticos e com *deadlines* curtos.

- IX. O IHM-Plasma é notavelmente capaz de manter seu desempenho de processamento, mesmo quando o *time slice* é reduzido para apenas 2⁸ ciclos de *clock*. Neste caso, observa-se uma redução máxima de apenas 3% no seu desempenho frente a redução de 79,4% apresentada pelo sistema puramente baseado em software.

Dados os resultados e contribuições deste trabalho. Espera-se que a Indústria de CIs possa absorver essa tecnologia aqui desenvolvida e consequentemente, viabilizar a criação de inovadores processadores ainda mais eficientes e mais integrados, com foco nas aplicações de sistemas baseados em tarefas e sem que seja necessário alterar o conjunto original de instruções.

Por fim, o autor destaca a importância da continuidade deste projeto de pesquisa. Para isso, elenca suas propostas de trabalhos futuros, são elas:

- I. Qualificação de alunos de graduação e pós-graduação para que possam utilizar o IHM-Plasma e propor novos desenvolvimentos e melhorias;
- II. Continuar a desenvolver aplicações com *time slices* curtos utilizando o IHM-Plasma, a fim de comprovar sua viabilidade frente as soluções comerciais totalmente baseadas em software;
- III. Aplicar em outros núcleos do tipo soft-cores a mesma metodologia utilizada para integração do IHM no microcontrolador Plasma e comparar os resultados;
- IV. Dar prosseguimento no processo de fabricação do ASIC desenvolvido. Para isso, é necessário financiamento por parte de órgãos de fomento à pesquisa e concluir o leiaute do CI, acrescentando seus *pads* e anéis de proteção;
- V. Após a fabricação do ASIC, realizar sua caracterização elétrica experimental e o desenvolvimento de diferentes aplicações de interesse comercial.

REFERÊNCIAS

1. ALTERA CORPORATION (San Jose). **Introduction to Megafunction IP Cores**. 2013. Disponível em: <https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/ug/archives/ug_intro_to_megafunctions_131.pdf>. Acesso em: 30 maio 2018.

2. AMAZON WEB SERVICES. **The FreeRTOS™ Reference Manual: API Functions and Configuration Options**. 2017. Disponível em: <https://www.freertos.org/Documentation/RTOS_book.html>. Acesso em: 30 maio 2018.

3. ARM LTD.. **Cortex-M Series**. Disponível em: <<http://www.arm.com/products/processors/cortex-m/index.php>>. Acesso em: 3 maio 2016.

4. BARRY, Richard. **Using the FreeRTOS Real Time Kernel: Standard Edition**. [s.l.]: Richard Barry, 2010. (FreeRTOS Tutorial Books).

5. BRITTON, Robert L.. **MIPS Assembly Language Programming**. [s.l.]: Pearson, 2003. 168 p.

6. BUTTAZZO, Giorgio C.. **Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications**. 3. ed. [s.l.]: Springer, 2011. (Real-Time Systems Series).

7. CADENCE DESIGN SYSTEMS. **EDA Tools and IP for System Design Enablement**. 2018. Disponível em: <<https://www.cadence.com/>>. Acesso em: 30 maio 2018.

8. CHERYL BERGLUND COUPÉ (Estados Unidos). Eecatalog.com (Ed.). **Microcontrollers Make Waves in Automotive and Internet of Things**. 2014. Disponível em: <<http://eecatalog.com/8bit/2014/01/31/microcontrollers-make-waves-in-automotive-and-internet-of-things/>>. Acesso em: 30 maio 2018.

9. CHOE, Tae-young et al. An improved multi-priority preemptive scheduler for transputer-based real-time systems. In: EUROMICRO WORKSHOP ON REAL-TIME SYSTEMS, 8., 1996, L'aquila. **Proceedings...** [s.l.]: Ieee, 1996. p. 181 - 186.

10. DANTAS, Leandro Poloni. **Apresentação do IHM-Plasma rodando Space Invaders**. 2018. Disponível em: <<https://youtu.be/VQfT0fHCugc>>. Acesso em: 06 jul. 2018.

11. DANTAS, Leandro Poloni; ARROIO, Ricardo. **Eletrônica Digital: técnicas digitais e dispositivos lógicos programáveis**. São Paulo: SENAI-SP Editora, 2014. 560 p. (Informações Tecnológicas. Área Eletroeletrônica).

12. DANTAS, Leandro Poloni. **GitHub LePoloni**. 2018. Disponível em: <<https://github.com/LePoloni>>. Acesso em: 19 jul. 2018.

13. EMBEDDED Systems/Super Loop Architecture. 2018. Disponível em: <https://en.wikibooks.org/wiki/Embedded_Systems/Super_Loop_Architecture>. Acesso em: 30 maio 2018.

14. FARBER, Georg et al. Improving processor utilization with a task classification model based application specific hard real-time architecture. In: INTERNATIONAL WORKSHOP ON REAL-TIME COMPUTING SYSTEMS AND APPLICATIONS, 4., 1997, Taipei.Proceedings... . [s.l.]: Ieee, 1997. p. 276 - 283.

15. GAITAN, Vasile Gheorghita; GAITAN, Nicoleta Cristina; UNGUREAN, Ioan. CPU Architecture Based on a Hardware Scheduler and Independent Pipeline Registers. **Ieee Transactions On Very Large Scale Integration (vlsi) Systems**, [s.l.], v. 23, n. 9, p.1661-1674, set. 2015. Institute of Electrical & Electronics Engineers (IEEE). <http://dx.doi.org/10.1109/tvlsi.2014.2346542>.

16. GARCIA, A.; SAEZ, S.; VILA, J.. A binary-tree architecture for scheduling real-time systems with hard and soft tasks. In: XII SYMPOSIUM ON INTEGRATED CIRCUITS AND SYSTEMS DESIGN, 12., 1999, Natal. **Proceedings...** . Natal: Ieee, 1999. p. 78 - 81.

17. GIMENEZ, Salvador Pinillos; DANTAS, Leandro Poloni. **Microcontrolador PIC18: conceitos, operação, fluxogramas e programação**. São Paulo: Érica, 2015. (Eixos).

18. GIMENEZ, Salvador Pinillos. **Microcontroladores 8051: teoria e prática**. São Paulo: Érica, 2010.

19. GLOBALFOUNDRIES. **SiGe HP Technologies**. 2018. Disponível em: <<https://www.globalfoundries.com/technologies/rf-soi-and-sige-technologies/sige-hp-technologies>>. Acesso em: 30 maio 2018.

20. GRAND VIEW RESEARCH (Estados Unidos). **Microcontroller Market Size Worth USD 26.98 Billion By 2020**. 2015. Disponível em: <<https://www.grandviewresearch.com/press-release/global-microcontroller-market>>. Acesso em: 30 maio 2018.

21. HARRIS, David Money; HARRIS, Sarah L.. **Digital Design and Computer Architecture**. 2. ed. [s.l.]: Morgan Kaufmann, 2012.

22. HENNESSY, John L.; PATTERSON, David A.. **Computer Architecture: A Quantitative Approach**. 5. ed. [s.l.]: Morgan Kaufmann, 2011. (The Morgan Kaufmann Series in Computer Architecture and Design).

23. IHS (Estados Unidos). **Microcontroller Market Growth Tied to Rise in Internet of Things Applications, IHS Says**. 2015. Disponível em: <<http://press.ihs.com/press-release/technology/microcontroller-market-growth-tied-rise-internet-things-applications-ihs-sa>>. Acesso em: 30 maio 2018.

24. INTEL CORPORATION. **Intel® FPGAs.** [20--]. Disponível em: <<https://www.altera.com/products/fpga/overview.html>>. Acesso em: 30 maio 2018.

25. INTEL CORPORATION. **QUARTUS PRIME SOFTWARE.** Disponível em: <<https://www.altera.com/products/design-software/fpga-design/quartus-prime/overview.html>>. Acesso em: 30 maio 2018.

26. KAESLIN, Hubert. **Digital Integrated Circuit Design: From VLSI Architectures to CMOS Fabrication.** Nova Iorque: Cambridge University Press, 2008.

27. KOHŮTKA, Lukáš; VOJTKO, Martin; KRAJCOVIC, Tibor. Hardware Accelerated Scheduling in Real-Time Systems. In: EASTERN EUROPEAN REGIONAL CONFERENCE ON THE ENGINEERING OF COMPUTER BASED SYSTEMS, 4., 2015, Brno.**Conference.** [S. l.]: Ieee, 2015. p. 142 - 143.

28. KUON, Ian; ROSE, Jonathan. Measuring the Gap Between FPGAs and ASICs. **Ieee Transactions On Computer-aided Design Of Integrated Circuits And Systems**, [s. l.], v. 26, n. 2, p.203-215, 22 jan. 2007. Mensal.

29. LAPLANTE, Phillip A.; OVASKA, Seppo J.. **Real-time Systems Design and Analysis: tools for the practitioner.** 4. ed. [s.l.]: Wiley-IEEE Press, 2011.

30. LAPLANTE, Phillip A.. **Real-Time Systems Design and Analysis: an engineer's handbook.** 3. ed. [s. l.]: I. E. E. E.Press, 2004.

31. LIU, C. L.; LAYLAND, James W.. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. **Journal Of The Acm**, [s.l.], v. 20, n. 1, p.46-61, 1 jan. 1973. Association for Computing Machinery (ACM). <http://dx.doi.org/10.1145/321738.321743>.

32. MARUYAMA, Naotaka; ISHIHARA, Tohru; YASUURA, Hiroto. An RTOS in hardware for energy efficient software-based TCP/IP processing. **2010 Ieee 8th Symposium On Application Specific Processors (sasp)**, [s.l.], p.58-63, jun. 2010. IEEE. <http://dx.doi.org/10.1109/sasp.2010.5521147>.

33. MAXFIELD, Max. **ASIC, ASSP, SoC, FPGA – What's the Difference?** 2014. Disponível em: <https://www.eetimes.com/author.asp?section_id=216&doc_id=1322856>. Acesso em: 06 jul. 2018.

34. MAYA, Paulo Alvaro; LEONARDI, Fabrizio. **Controle Essencial.** 2. ed. São Paulo: Pearson Education do Brasil, 2014.

35. MCGRATH, Dylan. **Microcontroller Market Plagued by Price Erosion.** 2015. Disponível em: <http://www.eetimes.com/document.asp?doc_id=1327470>. Acesso em: 30 maio 2018.

36. MISSOURI STATE UNIVERSITY (Estados Unidos). **MARS (MIPS Assembler and Runtime Simulator):** An IDE for MIPS Assembly Language Programming. 2014. Disponível em: <<http://courses.missouristate.edu/KenVollmar/mars/>>. Acesso em: 01 jun. 2018.
37. MOSIS. **Processes Available for MEP Accounts.** 2018. Disponível em: <<https://www.mosis.com/pages/products/mep/mep-available-processes>>. Acesso em: 30 maio 2018.
38. NAKANO, Takumi et al. Hardware Implementation of a Real-time Operating System. In: TRON PROJECT INTERNATIONAL SYMPOSIUM, 12., 1995, Tokyo. **Proceedings...** [s.l.]: Ieee, 1995. p. 34 - 42.
39. ONG, Soon Ee et al. SEOS: Hardware Implementation of Real-Time Operating System for Adaptability. In: INTERNATIONAL SYMPOSIUM ON COMPUTING AND NETWORKING, 1., 2013, Matsuyama. **Symposium.** [s.l.]: Ieee, 2013. p. 612 - 616.
40. PATTERSON, David A.; HENNESSY, John L.. **Organização de Projeto de Computadores:** A Interface Hardware/Software. 2. ed. Rio de Janeiro: Ltc, 2000.
41. RAFLA, Nader I.; GAUBA, Deepak. Hardware implementation of context switching for hard real-time operating systems. In: INTERNATIONAL MIDWEST SYMPOSIUM ON CIRCUITS AND SYSTEMS (MWSCAS), 54., 2011, Seoul. **Symposium.** [s.l.]: Ieee, 2011. p. 1 - 4.
42. REAL TIME ENGINEERS LTD. (Ed.). **Customisation.** Disponível em: <<https://www.freertos.org/a00110.html>>. Acesso em: 30 maio 2018.
43. REAL TIME ENGINEERS LTD.. **FreeRTOS FAQ - Memory Usage, Boot Times & Context Switch Times.** Disponível em: <<http://www.freertos.org/FAQMem.html>>. Acesso em: 3 maio 2016.
44. REAL TIME ENGINEERS LTD.. **RTOS Fundamentals.** Disponível em: <<http://www.freertos.org/implementation/a00002.html>>. Acesso em: 30 maio 2018.
45. RHOADS, Steve. **Plasma - most MIPS I(TM) opcodes :: Overview. 2001.** Disponível em: <<https://opencores.org/project,plasma,overview>>. Acesso em: 30 maio 2018.
46. SEDRA, Adel S.; SMITH, Kenneth C.. **Microeletrônica.** 5. ed. São Paulo: Pearson Prentice Hall, 2007. 864 p.
47. SILVA JUNIOR, Vidal Pereira da. **Aplicações Práticas do Microcontrolador 8051.** 11. ed. São Paulo: Érica, 2003.
48. STALLINGS, William. **Arquitetura e Organização de Computadores:** Projeto para o Desempenho. 5. ed. São Paulo: Prentice Hall, 2002.

49. TANENBAUM, Andrew S.. **Organização Estruturada de Computadores**. 5. ed. São Paulo: Pearson Prentice Hall, 2007. 464 p.

50. TANG, Yi; BERGMANN, Neil W.. A Hardware Scheduler Based on Task Queues for FPGA-Based Embedded Real-Time Systems. **Ieee Transactions On Computers**, [s.l.], v. 64, n. 5, p.1254-1267, 1 maio 2015. Institute of Electrical & Electronics Engineers (IEEE). <http://dx.doi.org/10.1109/tc.2014.2315637>.

51. TERIC INC.. **DE1-SoC Board**. Disponível em: <<http://www.terasic.com.tw/cgi-bin/page/archive.pl?Language=English&CategoryNo=167&No=836>>. Acesso em: 30 maio 2018.

52. TORRES, Gabriel. **Por Dentro da Arquitetura Intel Core**. 2006. Disponível em: <<http://www.clubedohardware.com.br/artigos/por-dentro-da-microarquitetura-intel-core/1203>>. Acesso em: 3 maio 2016.

53. VALVANO, Jonathan W.. **Embedded Microcomputer Systems: Real Time Interfacing**. 3. ed. [s.l.]: CL Engineering, 2011.

54. VALVANO, Jonathan W.. **Embedded Systems: Real-Time Operating System for ARM® Cortex™-M Microcontrollers**, Volume 3. 3. ed. [s.l.]: [Http://users.ece.utexas.edu/~valvano/](http://users.ece.utexas.edu/~valvano/), 2014.

55. VETROMILLE, Melissa et al. RTOS Scheduler Implementation in Hardware and Software for Real Time Applications. In: IEEE INTERNATIONAL WORKSHOP ON RAPID SYSTEM PROTOTYPING, 17., 2006, Chania. **Proceedings...** . Los Alamitos: Ieee Computer Society, 2006. p. 163 - 168.

56. ZARRINFAR, Farzad. **Optimizing Embedded Memory for the Latest ASIC and SOC Designs**. 2013. Disponível em: <<http://chipdesignmag.com/display.php?articleId=5260>>. Acesso em: 30 maio 2018.

GLOSSÁRIO

A

B

C

D

deadline – prazo para execução de uma tarefa

E

F

FIFO – *First-In, First-Out*, que em português significa primeiro a entrar, primeiro a sair, refere-se a estruturas de dados do tipo fila.

flush – em arquiteturas de *pipeline*, é o ato de esvaziar o *pipeline* quando ocorre um desvio no código

função-tarefa – função modelo escrita na forma de laço infinito utilizada pelos RTOSs para criação de tarefas

G

GDS II – formato de arquivo de banco de dados que é o padrão da indústria para definição de circuitos integrados e seus leiautes. É um formato de arquivo binário que representa formas geométricas planas, rótulos de texto e outras informações sobre o leiaute de forma hierárquica

GNU – versão do sistema operativo Unix cujo objetivo é oferecer um sistema operacional completo e totalmente composto por software livre

H

I

instanciação de tarefa – processo de criação de uma tarefa na memória do processador a partir de uma função modelo chamada de função-tarefa

J

K

kernel – a menor parte do sistema operacional que fornece escalonamento, despacho, comunicação entre as tarefas

L

latência – tempo entre a geração de um estímulo e o início de seu tratamento

M

megafunctions – biblioteca livre de blocos de propriedade intelectual (IP) parametrizáveis, fornecidos pela Intel (anteriormente Altera), que são otimizados para a síntese lógica eficiente em seus dispositivos de lógica programável

microkernel – *kernel* que fornece apenas escalonamento e despacho de tarefas

N

O

P

Q

R

S

super loop – estrutura de programa composta por um laço infinito, com todas as tarefas do sistema contidas nesse laço

stack – estrutura de dados do tipo *first-in, first-out* (FIFO).

T

tick interrupt – interrupção para sinalização de fim de um período de *time slice*

time slice – período fixo que uma tarefa tem para ser executada em sistemas baseados no algoritmo de escalonamento Round-Robin

U

V

W

X

Y

Z

APÊNDICE A – PROCEDIMENTO PARA GERAÇÃO DOS ARQUIVOS HEXADECIMAIS NO PADRÃO INTEL HEX COM OS PROGRAMAS A SEREM EXECUTADOS PELO MICROCONTROLADOR IHM-PLASMA

Neste apêndice é descrito o procedimento adotado para geração dos arquivos hexadecimais no padrão Intel HEX, a partir dos programas escritos em linguagem *Assembly* e C, para serem executados no microcontrolador IHM-Plasma.

Os programas escritos exclusivamente em linguagem *Assembly* foram criados no ambiente de desenvolvimento MARS (36) versão 4.5. A Figura 52 ilustra sua tela principal.

Figura 52 – Tela principal da IDE MARS utilizada para criação de programas para o microcontrolador IHM-Plasma.

Fonte: Autor.

O procedimento utilizado para geração dos arquivos hexadecimais no formato Intel HEX, a partir de programas escritos em linguagem *Assembly* é apresentado a seguir.

Os códigos *assembly* devem ser gravados a partir da posição zero (0x00000000) da memória de programas do microcontrolador IHM-Plasma. Para isso, é preciso configurar o ponto de início do programa para este endereço da seguinte forma:

1. Acessar o menu *Settings* → *Memory Configuration...*;
2. Na caixa *Configuration*, selecionar a opção *Compact, Text at Address 0*.

Na IDE MARS a referência *Text* se refere a memória de programa.

Para realizar a validação, montagem do programa (conversão das instruções *assembly* em códigos binários chamados de *opcodes*) e salvamento em arquivo binário correspondente ao programa, proceder da seguinte forma:

1. Acessar o menu *Run* e clicar na opção *Assemble*;
2. Acessar o menu *File* e clicar na opção *Dump Memory...*;
3. Na caixa de opções *Dump Format*, selecionar a opção *Binary*;
4. Clicar no botão *Dump to File...*;
5. Por último, salvar o arquivo com o nome “test.bin”.

A próxima etapa deste procedimento é a conversão do arquivo binário “test.bin” em quatro arquivos no formato Intel HEX, onde cada um dos arquivos é correspondente a um dos quatro bytes que formam cada uma das instruções *assembly* no padrão MIPS I. Esse procedimento é necessário, uma vez que a entidade escrita em VHDL correspondente a memória do microcontrolador IHM-Plasma (arquivo ram.vhd) utilizou quatro *Memory Megafunctions* da Intel para criação do bloco de memória no FPGA. Cada uma das *Memory Megafunctions* é responsável por um dos bytes de cada uma das instruções do programa.

Para realização da conversão do arquivo, utilizar o conjunto de ferramentas de software disponibilizadas pelo idealizador do microcontrolador Plasma, Steve Rhoads, e disponível para download no site OpenCores (45). Neste conjunto de ferramentas também está disponível o compilador C GCC com *script* pré-configurados compatíveis com o ISA MIPS I (arquivo “makefile”).

Neste caso, utilizar o aplicativo “gmake.exe” para conversão do arquivo “test.bin” em quatro arquivos no formato Intel HEX nomeados como “code0.hex”, “code1.hex”, “code2.hex” e “code3.hex”. Procedimento:

1. Copiar o arquivo “test.bin” na mesma pasta onde se encontra o programa “gmake.exe”;
2. Abrir o *prompt* de comandos do Windows;
3. Digitar: “cd <caminho do diretório onde se encontra o programa>”;
4. Executar o programa digitando: “gmake tohex”.

A Figura 53 ilustra o procedimento realizado no *prompt* de comandos do Windows.

Figura 53 – Procedimento para conversão de arquivo binário em arquivos no padrão Intel HEX utilizados como programa do microcontrolador IHM-Plasma.

```
C:\Users\Leandro>cd c:\plasma\trunk\tools  
c:\plasma\trunk\tools>gmake tohex  
bintohehex.exe test.bin  
copy code*.hex ..\vhd1  
code0.hex  
code1.hex  
code2.hex  
code3.hex  
4 arquivo(s) copiado(s).
```

Fonte: Autor.

Por fim, para que os arquivos gerados sejam usados na síntese para FPGA, é necessário copiá-los na mesma pasta onde se encontram os arquivos VHDL do projeto do microcontrolado IHM-Plasma.

No arquivo “ram.vhd” é feita a associação de cada um dos arquivos gerados (“code0.hex”, “code1.hex”, “code2.hex” e “code3.hex”) a uma das instâncias da *Memory Megafunction* chamada *lpm_ram_dq* (*lpm_ram_io_component0*, *lpm_ram_io_component1*, *lpm_ram_io_component2* e *lpm_ram_io_component3*).

A Figura 54 ilustra de forma sintética o processo de instanciação das *Memory Megafunctions* com destaque para o arquivo hexadecimal associado a cada uma delas.

Figura 54 – Instanciação das *Memory Megafunctions* no arquivo ram.vhd com destaque para os arquivos hexadecimais (correspondentes ao programa a ser gravado) associada a cada uma delas.

```

altera_ram:
if memory_type = "ALTERA_LPM" generate
    signal byte_we : std_logic_vector(3 downto 0);
begin
    byte_we <= write_byte_enable when enable = '1' else "0000";
    lpm_ram_io_component0 : lpm_ram_dq
        GENERIC MAP (
            ...
            --lpm_file => "code0.hex",      --C
            lpm_file => "code3.hex",      --Assembly
            ...
        )
    lpm_ram_io_component1 : lpm_ram_dq
        GENERIC MAP (
            ...
            --lpm_file => "code1.hex",      --C
            lpm_file => "code2.hex",      --Assembly
            ...
        )
    lpm_ram_io_component2 : lpm_ram_dq
        GENERIC MAP (
            ...
            --lpm_file => "code2.hex",      --C
            lpm_file => "code1.hex",      --Assembly
            ...
        )
    lpm_ram_io_component3 : lpm_ram_dq
        GENERIC MAP (
            ...
            --lpm_file => "code3.hex",      --C
            lpm_file => "code0.hex",      --Assembly
            ...
        )
end generate; --altera_ram

```

Fonte: Autor.

Complementarmente, o procedimento utilizado para geração dos arquivos hexadecimais no formato Intel HEX, a partir de programas escritos em linguagem C é apresentado a seguir.

No caso dos programas escritos em linguagem C, de acordo com o arcabouço apresentado na seção 3.4, eles foram editados no aplicativo Notepad++, porém poderiam ter sido editados em qualquer editor de textos.

Uma vez configurado o compilador C GCC de acordo com as instruções disponibilizadas pelo idealizador do microcontrolador Plasma, Steve Rhoads, e disponível para download no site OpenCores (45). É preciso alterar o arquivo “makefile” com os *scripts* adequados para compilação de um projeto que utilize como arquivo principal o arquivo “plasma_rt.c”. A Figura 55 apresenta as linhas de código com os *scripts* inseridos no arquivo original. A inserção foi feita a partir da linha 270 do arquivo.

Figura 55 – Linhas de código com *scripts* inseridos no arquivo makefile para compilação dos projetos em C baseados no arquivo plasma_rt.

```

270 # Alterações Leandro 5/8/17 para teste do IHM-Plasma
271 plasma_rt:
272     $(AS_MIPS) -o boot.o boot.asm
273     $(GCC_MIPS) plasma_rt.c
274     $(GCC_MIPS) no_os_rt.c
275     $(GCC_MIPS) hw_microk.c
276     $(LD_MIPS) -Ttext 0 -e entry -Map test.map -s -N -o test.axf \
277     boot.o plasma_rt.o no_os_rt.o hw_microk.o
278     -$(DUMP_MIPS) --disassemble test.axf > test.lst
279     $(CONVERT_BIN)
280     $(CP) code.txt $(VHDL_DIR)

```

Fonte: Autor.

Procedimento para compilação C do projeto:

1. Copiar todos os arquivos do projeto na mesma pasta onde se encontra o arquivo “makefile”, bem como o compilador e o programa “gmake.exe”;
2. Abrir o *prompt* de comandos do Windows;
3. Digitar: “cd <caminho do diretório onde se encontra o programa>”;
4. Executar o compilador digitando: “make plasma_rt”.

Um dos arquivos gerados pelo compilador é o arquivo hexadecimal “test.bin” que traz o programa compilado no formato binário. Assim como no procedimento apresentado para programas escritos em Assembly, o aplicativo “gmake.exe” deve ser utilizado para conversão do arquivo “test.bin” em quatro arquivos no formato Intel HEX nomeados como “code0.hex”, “code1.hex”, “code2.hex” e “code3.hex”. Para isso, basta digitar “gmake tohex” no *prompt* de comandos do Windows.

A Figura 56 ilustra o procedimento de compilação e conversão realizado no *prompt* de comandos do Windows.

Figura 56 – Procedimento para compilação do projeto C baseado no arquivo `plasma_rt.c` e conversão do arquivo binário gerado em arquivos no padrão Intel HEX utilizados como programa do microcontrolador IHM-Plasma.

```

c:\>cd c:\plasma\trunk\tools

c:\plasma\trunk\tools>make plasma_rt
..\gccmips_elf\as -o boot.o boot.asm
..\gccmips_elf\gcc -O2 -Wall -c -s -fno-pic -mips1 -mno-abicalls plasma_rt.c
..\gccmips_elf\gcc -O2 -Wall -c -s -fno-pic -mips1 -mno-abicalls no_os_rt.c
..\gccmips_elf\gcc -O2 -Wall -c -s -fno-pic -mips1 -mno-abicalls hw_microk.c
hw_microk.c: In function 'sw_scheduler_init':
hw_microk.c:230: warning: implicit declaration of function 'puts'
hw_microk.c:230: warning: implicit declaration of function 'putchar'
hw_microk.c:274: warning: implicit declaration of function 'itoa10'
hw_microk.c: In function 'sw_scheduler_init_v2':
hw_microk.c:336: warning: implicit declaration of function 'itoa2'
..\gccmips_elf\ld -Ttext 0 -e entry -Map test.map -s -N -o test.axf \
boot.o plasma_rt.o no_os_rt.o hw_microk.o
..\gccmips_elf\objdump --disassemble test.axf > test.lst
convert_bin.exe
test.axf -> code.txt & test.bin
Entry=0x0 length=6544=0x1990
copy code.txt ..\vhdl
1 arquivo(s) copiado(s).

c:\plasma\trunk\tools>gmake tohex
bintohehex.exe test.bin
copy code*.hex ..\vhdl
code0.hex
code1.hex
code2.hex
code3.hex
4 arquivo(s) copiado(s).

c:\plasma\trunk\tools>_
  
```

Fonte: Autor.

O restante deste procedimento é similar ao aplicado para programas escritos em linguagem *Assembly*. A única diferença está na ordem com que os arquivos gerados são associados as instâncias das *Memory Megafunctions*. A Figura 54 ilustra de forma sintética o processo de instanciação das *Memory Megafunctions* com destaque para o arquivo hexadecimal associado a cada uma delas.

Figura 57 – Instanciação das *Memory Megafunctions* no arquivo ram.vhd com destaque para os arquivos hexadecimais (correspondentes ao programa a ser gravado) associada a cada uma delas.

```

altera_ram:
if memory_type = "ALTERA_LPM" generate
    signal byte_we : std_logic_vector(3 downto 0);
begin
    byte_we <= write_byte_enable when enable = '1' else "0000";
    lpm_ram_io_component0 : lpm_ram_dq
        GENERIC MAP (
            ...
            lpm_file => "code0.hex",      --C
            --lpm_file => "code3.hex",    --Assembly
            ...
        )
    lpm_ram_io_component1 : lpm_ram_dq
        GENERIC MAP (
            ...
            lpm_file => "code1.hex",      --C
            --lpm_file => "code2.hex",    --Assembly
            ...
        )
    lpm_ram_io_component2 : lpm_ram_dq
        GENERIC MAP (
            ...
            lpm_file => "code2.hex",      --C
            --lpm_file => "code1.hex",    --Assembly
            ...
        )
    lpm_ram_io_component3 : lpm_ram_dq
        GENERIC MAP (
            ...
            lpm_file => "code3.hex",      --C
            --lpm_file => "code0.hex",    --Assembly
            ...
        )
end generate; --altera_ram

```

Fonte: Autor.

APÊNDICE B – CÓDIGO FONTE DO PROGRAMA GRAVADO NA MEMÓRIA ROM DO ASIC PROJETADO

Neste apêndice consta o código fonte escrito em linguagem C do programa gravado na memória ROM do ASIC projetado para o microcontrolador IHM-Plasma.

O programa escrito declara 4 tarefas com as seguintes funções:

- Task 0: pisca-pisca com temporização através de laço no programa;
- Task 1: pisca-pisca com temporização através do recurso de *sleep* de tarefas oferecido pelo *microkernel* por hardware;
- Task 2: copia o valor recebido na UART nos 8 bits menos significativos do *port* de saída do microcontrolador;
- Task 3: monitora o port de entrada e a cada alteração de estado, envia o valor lido pela UART.

Os bits 8 e 9 do *port* de entrada são usados para seleção do *time slice* do escalonador.

```

#include "plasma_rt_ASIC.h"

//Macros utilizadas
#define MemoryRead(A) (*(volatile unsigned int*)(A))
#define MemoryWrite(A,V) *(volatile unsigned int*)(A)=(V)

#define EnableScheduler() *(volatile unsigned int*)(SCHEDULER_EN)=(1)
#define DisableScheduler() *(volatile unsigned int*)(SCHEDULER_EN)=(0)

#define TimeSlice(V) *(volatile unsigned int*)(TIME_SLICE)=(V)

#define EnableCounter() *(volatile unsigned int*)(IRQ_MASK)=(MemoryRead(IRQ_MASK) | IRQ_COUNTER18)
#define DisableCounter() *(volatile unsigned int*)(IRQ_MASK)=(MemoryRead(IRQ_MASK)&~IRQ_COUNTER18)

#define EnableNotCounter() *(volatile unsigned int*)(IRQ_MASK)=(MemoryRead(IRQ_MASK) | IRQ_COUNTER18_NOT)
#define DisableNotCounter() *(volatile unsigned int*)(IRQ_MASK)=(MemoryRead(IRQ_MASK)&~IRQ_COUNTER18_NOT)

#define EnableTask(t) *(volatile unsigned int*)(TASK_VECTOR)=(MemoryRead(TASK_VECTOR) | (1<<t))
#define DisableTask(t) *(volatile unsigned int*)(TASK_VECTOR)=(MemoryRead(TASK_VECTOR)&~(1<<t))

#define EnableUartRX() *(volatile unsigned int*)(IRQ_MASK)=(MemoryRead(IRQ_MASK) | IRQ_UART_READ_AVAILABLE)
#define DisableUartRX() *(volatile unsigned int*)(IRQ_MASK)=(MemoryRead(IRQ_MASK)&~IRQ_UART_READ_AVAILABLE)

//Protótipos de funções utilizadas de outros arquivos
void OS_AsmInterruptEnable(int);
int putchar(int value);
int puts(const char *string);
int kbhit(void);
int clear_output_pin(unsigned int pin);
int set_output_pin(unsigned int pin);
unsigned int read_input_pin(unsigned int pin);
unsigned int read_output_pin(unsigned int pin);
int sleep_yield_task(unsigned int period);
int create_task_a(int (*task_function_pointer)(), int number, int priority,
int argumento);
void scheduler_init(void);

//XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
//FUNÇÃO DE TRATAMENTO DE INTERRUPÇÃO TIMER
//XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
unsigned int trataInt(unsigned int status, unsigned int stackp)
{
    return stackp;
}

//XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
//TESTE HD SCHEDULER COM 4 TAREFAS PARA ASIC
//XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
//Input para UART
int task3(int v)
{
    int i, ant;
    ant = 0;

    putchar('T'); putchar((char)v);
    while(1)
    {
        i = MemoryRead(GPIOA_IN) & 0x000000FF;
        if(i != ant)
        {
            ant = i;
            putchar((char)i);
        }
    }
}

```



```

        }
        sleep_yield_task(10);
    }
}
//UART para Output
int task2(int v)
{
    int i;

    putchar('T'); putchar((char)v);

    while(1)
    {
        if(kbhit())
        {
            i = MemoryRead(UART_READ);
            MemoryWrite(GPIO0_CLEAR, 0x000000FF);
            MemoryWrite(GPIO0_SET, 0x000000FF & i);
        }
        sleep_yield_task(10);
    }
}
//Pisca-pisca por sleep
int task1(int v)
{
    putchar('T'); putchar((char)v);

    while(1)
    {
        //Pisca led
        set_output_pin(9); //GPIO_0 JP1 pino 14
        sleep_yield_task(1000);
        clear_output_pin(9); //GPIO_0 JP1 pino 14
        sleep_yield_task(1000);
    }
}
//Idle - pisca-pisca por delay
int task0(int v)
{
    int i;

    putchar('T'); putchar((char)v);

    while(1)
    {
        //Pisca led
        for(i = 100000; i>0; i--);
        set_output_pin(8); //GPIO_0 JP1 pino 13
        for(i = 100000; i>0; i--);
        clear_output_pin(8); //GPIO_0 JP1 pino 13
    }
}

int main()
{
    int j,k;

    //Desabilita interrupções (coprocessador reg 12 bit 0)
    OS_AsmInterruptEnable(0x00);
    //Reset das tarefas ativas
    MemoryWrite(TASK_VECTOR, 0);

    k = (MemoryRead(GPIOA_IN)&0x30)>>8;

    //
    create_task_a(task0, 0, função, task, prio, arg
quadrada create_task_a(task1, 1, 1, '0'); //Idle - Gerador de onda
          create_task_a(task1, 1, 2, '1');
          create_task_a(task2, 2, 3, '2');

```

```

create_task_a(task3, 3,      3,    '3');

//Time slice
switch(k)
{
    case 0:      j = 25000; break;      //1ms @ 25MHz
    case 1:      j = 12500; break;      //500us @ 25MHz
    case 2:      j = 2500; break;      //100us @ 25MHz
    case 3:      j = 500; break; break; //20us @ 25MHz
    default:     j = 25000; break;      //1ms @ 25MHz
}
Timeslice(j);

puts("IHM-Plasma by LPD (2018)"); putchar('\r'); putchar('\n');
puts("TS="); putchar(k+'0'); putchar('\r'); putchar('\n');

Enablescheduler();

//Força o desvio para task0, tarefa idle.
//Isso é necessário porque quando ocorre a primeira interrupção do
microkernel,
//o PC atual é salvo como PC da task0.
task0(0);
//Looping infinito (evita erros caso a tarefa 0 falhe)
for(;;);
}

```

APÊNDICE C – PUBLICAÇÕES

1. SEMINATEC 2016 – XI Workshop sobre Semicondutores; Micro & Nano Tecnologia

Resumo publicados nos anais do evento e apresentado na sessão de pôsteres no dia 14 de abril de 2016 em Campinas-SP. Aqui é apresentada a proposta desse projeto e os resultados esperados.

Título do trabalho: *Innovative CPU Architecture Focused on Real-Time Systems*

Sítio eletrônico: <http://www.unicamp.br/unicamp/noticias/2016/04/14/12o-seminatec-mostra-tendencias-em-micro-e-nanotecnologia-aos-alunos>

2. SEMINATEC 2018 – XIII Workshop sobre Semicondutores; Micro & Nano Tecnologia

Resumo publicados nos anais do evento e apresentado na sessão de pôsteres no dia 19 de abril de 2018 em São Bernardo do Campo-SP (ISBN: 978-85-98123-12-7).

Título do trabalho: *Design of a Customized Microcontroller Focusing on Task-Based Systems*

Sítio eletrônico: <http://fei.edu.br/seminatec/#proceedings>

3. SIM 2018 - 33º SIMPÓSIO SUL DE MICROELETRÔNICA

Artigo publicados nos anais do simpósio e apresentado de forma oral no dia 4 de maio de 2018 em Curitiba-PR.

Título do trabalho: *Using Interlocked Hardware Microkernel to Boost the Processing Performance of Task-Based Systems*

Sítio eletrônico: <http://emicro.ufpr.br/sim2018.htm>

4. IEEE Embedded Systems Letters

Artigo submetido em novembro de 2017 para esse importante periódico internacional, que reporta os principais resultados que foram obtidos por esse projeto de pesquisa. Destacamos que esse artigo já passou por duas avaliações realizadas por três revisores e o veredito por parte do Editor foi que respondêssemos as considerações (algumas maiores e outras menores) sugeridas por eles. O artigo foi reformulado levando-se em conta os comentários e sugestões

dos revisores. A última ressubmissão foi realizada em 30 de junho de 2018 e neste momento ele se encontra em reanálise.

Título do trabalho: *A Novel Processor Architecture with a Hardware Microkernel to Improve the Performance of Task-Based Systems*

Sítio Eletrônico: <http://ieee-ceda.org/publication/ieee-embedded-systems-letters-esl>