UNIVERSIDADE DO VALE DO ITAJAÍ
CENTRO DE CIÊNCIAS TECNOLÓGICAS DA TERRA E DO MAR
CURSO DE CIÊNCIA DA COMPUTAÇÃO

ANÁLISE E PORTABILIDADE DE UM SISTEMA OPERACIONAL DE TEMPO REAL PARA O BIP

por

Hendrig Wernner Maus Santana Gonçalves

Itajaí (SC), junho de 2013

UNIVERSIDADE DO VALE DO ITAJAÍ
CENTRO DE CIÊNCIAS TECNOLÓGICAS DA TERRA E DO MAR
CURSO DE CIÊNCIA DA COMPUTAÇÃO

ANÁLISE E PORTABILIDADE DE UM SISTEMA OPERACIONAL DE TEMPO REAL PARA O BIP

Área de Sistemas Operacionais

por

Hendrig Wernner Maus Santana Gonçalves

Relatório apresentado à Banca Examinadora do Trabalho Técnico-científico de Conclusão do Curso de Ciência da Computação para análise e aprovação.
Orientador: Fabrício Bortoluzzi, M.Sc.
Co-orientador: Cesar Albenes Zeferino, Dr.

Itajaí (SC), junho de 2013

À Bruna e Alessandra, as mulheres da minha vida.
À Valmir, meu pai, o homem que um dia almejo ser.

AGRADECIMENTOS

Sempre que realizamos agradecimentos esquecemos o nome de pessoas importantes então não irei dizer nomes.

Agradeço primeiramente à minha família (tanto os mais próximos quanto os mais distantes) por terem me dado apoio durante esse período da minha vida.

Agradeço aos meus orientadores por terem dedicado seu tempo ao sonho de outra pessoa.

Agradeço aos meus avaliadores por terem expressado as suas opiniões sobre um trabalho com o objetivo de fazê-lo dar certo.

Agradeço aos professores que tive na Universidade. Todos eles foram importantes e me deram o maior presente que eu poderia receber, que é o presente de ter conhecimento e saber aplica-lo.

Agradeço aos meus amigos por todas as opiniões, dicas e incentivos.

Agradeço por chegar a mais esta etapa da minha vida. Que um dia eu possa agradecer por ter chegado mais longe!

“Chaos isn't a pit. Chaos is a ladder. Many who try to climb it fail and never get to try again. The fall breaks them. And some, are given a chance to climb. They refuse, they cling to the realm or the gods or love. Illusions. Only the ladder is real. The climb is all there is.”

Petyr ‘Littlefinger’ Baelish em Game of Thrones

RESUMO

GONÇALVES, Hendrig W. M. S. **Análise e portabilidade de um sistema operacional de tempo real para o BIP.** Itajaí, 2012. 86 f. Trabalho Técnico-científico de Conclusão de Curso (Graduação em Ciência da Computação) – Centro de Ciências Tecnológicas da Terra e do Mar, Universidade do Vale do Itajaí, Itajaí, 2013.

Este trabalho documenta a criação do BIP/OS, um sistema operacional de tempo real básico a ser utilizado como ferramenta de apoio ao ensino de conceitos de sistemas operacionais. O BIP/OS é um sistema operacional que é executado sobre o microcontrolador µBIP, o membro mais completo da família de processadores BIP, utilizados como ferramentas de apoio ao ensino nos primeiros semestres dos cursos de Ciência da Computação e Engenharia de Computação da Universidade do Vale do Itajaí - UNIVALI. Para a criação do BIP/OS são considerados os conceitos-chave de sistemas operacionais de tempo real, além da utilização destes conceitos em sistemas operacionais de tempo real comerciais, mais especificamente para aqueles criados para executar sobre o PIC16, cuja arquitetura é fonte de inspiração para o µBIP. Por fim, é documentada a criação do BIP/OS.

Palavras-chave: Sistemas Operacionais. Sistemas Operacionais de Tempo Real. Arquitetura e Organização de Computadores.

ABSTRACT

This work deals with the creation of BIP/OS, a real time operating system to be used as a basic tool to support teaching of operating systems concepts. The BIP/OS is an operating system that runs on the μBIP microcontroller, the most complete member of the family of processors BIP, used as tools to support teaching in the first semesters of courses in Computer Science and Computer Engineering from the Universidade do Vale do Itajai - UNIVALI. For the creation of BIP/OS are considered the key concepts of real-time operating systems, and the use of these concepts in commercial real-time operating systems, specifically for those designed to run on the PIC16, whose architecture is a source of inspiration for μBIP. Finally, the creation of BIP/OS is documented.

***Keywords**: Operating Systems. Real Time Operating Systems. Computers Architecture and Organization*

LISTA DE FIGURAS

LISTA DE QUADROS

LISTA DE ABREVIATURAS E SIGLAS

ADL	Architecture Description Language
API	Application Programming Interface
BIP	Basic Instruction-set Processor
BIP/OS	BIP Operating System
FIFO	First In, First out
FPGA	Field Programmable Gate Arrays
IC-UNICAMP	Instituto de Computação da Universidade de Campinas
LEDS	Laboratory of Embedded and Distributed Systems
PIC	Programmable Inteligent Computer
RF	Requisito Funcional
RNF	Requisito Não-Funcional
RTOS	Real Time Operating System
TTC	Trabalho Técnico-científico de Conclusão de Curso
UNIVALI	Universidade do Vale do Itajaí
VHDL	VHSIC Hardware Description Language
VHSIC	Very High-Speed Integrated Circuits

SUMÁRIO

1 INTRODUÇÃO

O BIP (Basic Instruction-set Processor) é um processador cuja arquitetura foi projetada para facilitar o aprendizado de conceitos introdutórios de arquitetura e organização de computadores em fases iniciais de cursos de graduação na área de Computação (PEREIRA, 2008).

A família de processadores BIP possui cinco versões, todas desenvolvidas no Laboratório de Sistemas Embarcados e Distribuídos (LEDS – Laboratory of Embedded and Distributed Systems) da Universidade do Vale do Itajaí (UNIVALI). O processador BIP I (MORANDI et al., 2006) foi concebido com oito instruções que possibilitam o controle, armazenamento em memória, carga no acumulador e instruções aritméticas. O BIP II (MORANDI; RAABE; ZEFERINO, 2006), além das instruções do BIP I, recebeu instruções para suporte à laços de repetição e desvios. Ao BIP III (RECH, 2011) foram acrescidos suporte à instruções de lógica e operações binárias. O BIP IV (RECH, 2011) incorpora instruções de entrada e saída e chamadas de procedimentos. O quinto integrante da família BIP é o μBIP (PEREIRA, 2008), desenvolvido com o intuito de ensino de sistemas embarcados, ao qual foram acrescidas funcionalidades típicas de microcontroladores.

Neste trabalho foi avaliada a possibilidade dessa família de processadores vir a acomodar um sistema operacional. Desse modo tornar-se-ia possível a inclusão desses recursos de ensino nas disciplinas de sistemas operacionais nos cursos de Ciência da Computação e Engenharia de Computação. Em uma primeira análise, foram levados em conta a preservação da simplicidade da arquitetura, considerado um item importante dada a finalidade educacional da mesma, a viabilidade da utilização do sistema no ensino e a simplicidade em detrimento do desempenho ou outros critérios de desenvolvimento.

A pretensão foi criar um sistema operacional básico, escrito para o processador BIP que mais se adequasse às necessidades, neste caso, o μBIP, levando em conta as limitações desse processador. Foram necessárias alterações no mesmo para que fosse possível concluir o projeto.

Considerando os itens acima, foi necessário investigar o tipo de sistema operacional que melhor se aproveitaria dos recursos do processador em questão, sem extravasar os limites impostos pela arquitetura simples do mesmo. Segundo Stankovic e Rajkumar (2004) um sistema operacional de tempo real (RTOS – Real Time Operating System) provê suporte básico para escalonamento, gestão de recursos, sincronização, comunicação, temporização precisa e suporte à entrada e saída.

De acordo com Deitel, Deitel e Choffnes (pag. 225, 2005)

> Um sistema operacional de tempo real é diferente de um sistema padrão, porque cada operação deve apresentar resultados corretos e que retornem dentro de um certo período de tempo. Sistemas de tempo real são usados em aplicações críticas quanto a tempo, como sensores de monitoração. Geralmente são sistemas pequenos, embarcados.

Shaw (2001) afirma existirem diversas aplicações que se utilizam de sistemas de tempo real, como sistemas de controle de veículos, sistemas militares de controle de tiros, sistemas de automação predial, entre outros. Portanto, sistemas operacionais de tempo real são suficientemente grandes para contemplar o ensino de conceitos de sistemas operacionais, mas pequenos o bastante para serem suportados por arquiteturas simples como a do BIP.

Segundo Pereira (2008), tanto a arquitetura quanto alguns elementos do conjunto de instruções do BIP são inspiradas no PIC (Programmable Interface Controller). Existem no mercado algumas alternativas de RTOS que podem ser utilizadas em microcontroladores PIC, muitas delas de código-fonte aberto. Esses sistemas permitem que sejam verificadas soluções de implementação que seriam facilmente aplicadas ao BIP, devido à proximidade conceitual das arquiteturas do BIP e do PIC.

Sabendo, portanto, que os processadores BIP são baseados nos conceitos dos microcontroladores PIC, e sabendo também que existem RTOS que funcionam para microcontroladores PIC, propôs-se como trabalho técnico-científico de conclusão de curso a análise e portabilidade de um sistema operacional de tempo real para o BIP, visando aumentar a utilização desta arquitetura para fins de ensino, levando em conta a simplicidade do sistema e o potencial educacional do mesmo, além de considerar a simplicidade da arquitetura para a qual o sistema operacional está sendo portado.

1.1 PROBLEMATIZAÇÃO

1.1.1 Formulação do Problema

Em virtude da simplicidade da arquitetura BIP, além das ferramentas já criadas para a mesma, como a IDE Bipide (RECH, 2011), esta família de processadores tem se mostrado útil nas disciplinas de circuitos digitais, arquitetura e organização de computadores, algoritmos e compiladores, atuando como instrumento auxiliar para fixação do conteúdo das mesmas.

A disciplina de sistemas operacionais não poderia usufruir do BIP por falta justamente de um sistema operacional de referencia que pudesse executar sobre a plataforma. A pergunta

de pesquisa, portanto, foi: Algum dos processadores da família BIP é capaz de suportar um sistema operacional? Em caso negativo, quais elementos seriam necessários para se obter este suporte?

Adicionalmente, as seguintes perguntas auxiliares foram investigadas: Qual a complexidade que este sistema operacional pode ter? Existe algum tipo específico de sistema operacional ideal para a arquitetura BIP?

1.1.2 Solução Proposta

A arquitetura dos processadores BIP possui forte influência da arquitetura dos microcontroladores PIC, da Microchip. Existem diversos sistemas operacionais de tempo real que executam nestes microcontroladores, o que torna válido realizar o porte de um destes RTOS (Real Time Operating Systems) para a arquitetura BIP, mais especificamente para o processador BIP que melhor se adequasse ao sistema.

Assim sendo, este trabalho realizou uma análise dos RTOS já existentes, levando em consideração a qualidade e a facilidade de compreensão das funcionalidades do código-fonte dos mesmos, para, em seguida, realizar a portabilidade do mesmo para a arquitetura BIP.

Após o processo de análise foi realizada a simulação do RTOS portado utilizando as ferramentas já criadas nos trabalhos anteriores a este, dentre elas o simulador da arquitetura do μBIP escrito em ArchC (PEREIRA, 2008) e o modelo em VHDL (VHSIC Hardware Description Language) do μBIP (PEREIRA, 2008), ambos com as devidas alterações necessárias, garantindo a funcionalidade do mesmo.

1.2 OBJETIVOS

1.2.1 Objetivo Geral

O objetivo geral deste trabalho foi desenvolver e documentar o BIP/OS, um sistema operacional mínimo para a arquitetura BIP.

1.2.2 Objetivos Específicos

O objetivo geral foi alcançado por meio do cumprimento de cinco objetivos específicos:

- Revisar a literatura de processadores comerciais PIC e do processador BIP em busca dos sistemas operacionais que venham a servir como referência. O produto gerado ao final desta etapa será o capítulo de revisão bibliográfica contendo uma comparação dos trabalhos relacionados a esta proposta e a escolha pela adaptação de um desses sistemas operacionais;
- Modelar o sistema operacional mais adaptável ao BIP e mapear as mudanças necessárias para sua portabilidade. O produto gerado por esta etapa consiste na modelagem de um sistema operacional para o BIP, possivelmente por meio de engenharia reversa, e um artefato relacionando as mudanças esperadas para realização no TTC II;
- Portar o sistema operacional do PIC para o BIP. O produto gerado nesta etapa foi um apêndice contendo o conjunto de modificações aplicadas à um código-fonte pré-existente representando sua portabilidade para o BIP;
- Simular, testar e validar o sistema operacional gerado para a arquitetura BIP; e
- Documentar e divulgar este trabalho de conclusão de curso através da elaboração de um artigo técnico-científico que será encaminhado para publicação em periódicos ou eventos da área.

1.3 Metodologia

A metodologia deste trabalho de conclusão de curso é dividida em seis partes:

- Conceituação: nesta etapa foram realizados estudos sobre os principais conceitos de sistemas operacionais, focando especificamente em sistemas operacionais de tempo real, além de estudos sobre os processadores BIP e PIC, utilizando-se de livros e manuais para compreender os mesmos;
- Estudo e análise dos RTOS candidatos: nesta etapa foram analisados os RTOS disponíveis para o PIC que possuam código-fonte aberto e proposta de utilização que se adequem às limitações arquiteturais do BIP. Ao final da mesma, foi selecionado o sistema operacional de tempo real que melhor se adequou ao BIP;

- Especificação do RTOS: esta etapa consistiu na especificação da portabilidade do sistema selecionado na etapa anterior para a arquitetura do BIP, aplicando melhorias no decorrer deste processo;

- Implementação: esta etapa consistiu na implementação do RTOS especificado anteriormente;

- Avaliação e validação: nesta etapa foi realizada a avaliação do RTOS gerado, utilizando-se dos simuladores disponíveis para o BIP, além da aplicação de testes que comprovem e validem o funcionamento do RTOS gerado; e

- Documentação: nesta etapa foi realizada a documentação de todo o processo existente neste trabalho, desde a conceituação até a validação do mesmo, incluindo também a criação de um artigo científico para publicação.

1.4 Estrutura do trabalho

Este trabalho está dividido em quatro capítulos. O Capítulo 1, Introdução, apresentou uma visão geral do trabalho. O Capítulo 2, Fundamentação Teórica, apresenta os principais conceitos necessários para a realização do projeto, além das pesquisas realizadas para a escolha do RTOS que melhor se adeque ao mesmo. O Capítulo 3, Desenvolvimento, descreve o processo de portabilidade do RTOS selecionado. O Capítulo 4, Conclusão, apresenta os resultados preliminares obtidos durante a execução do projeto.

2 FUNDAMENTAÇÃO TEÓRICA

Neste capítulo é apresentada a revisão bibliográfica sobre os temas abordados no projeto com o objetivo de situar o leitor no contexto do trabalho.

2.1 Processadores BIP

De acordo com Pereira (2008), a família de processadores BIP foi desenvolvida por pesquisadores do Laboratório de Sistemas Embarcados e Distribuídos (LEDS) do curso de Ciência da Computação da Universidade do Vale do Itajaí – UNIVALI com o objetivo de disponibilizar uma arquitetura simples o suficiente para servir como objeto de ensino das disciplinas voltadas à arquitetura de computadores e eletrônica. Tais processadores foram desenvolvidos gradativamente, de forma que cada nova versão é o resultado de aprimoramentos realizados em versões anteriores.

De acordo com Morandi et.al. (2006), foram estabelecidas três diretrizes para o desenvolvimento do BIP:

1. A arquitetura deveria ser simples, permitindo não apenas o seu entendimento mas também possibilitando a sua implementação nas disciplinas de circuitos digitais;

2. A arquitetura deveria ser extensível; e

3. Todas as ferramentas de apoio a serem desenvolvidas deveriam priorizar a utilização da estrutura de linguagens compreendidas por alunos das séries iniciais.

O Quadro 1 apresenta um comparativo do conjunto de instruções disponíveis em cada tipo de processador BIP. Na mesma é possível notar uma evolução do conjunto de instruções com o passar do tempo, possibilitando a adição de novas funcionalidades a cada novo processador da família, sem abrir mão das instruções implementadas anteriormente.

Instrução	**Descrição**	**BIP I**	**BIP II**	**BIP III**	**BIP IV**	**µBIP**
HLT	Halt	×	×	×	×	×
STO operando	Store	×	×	×	×	×
LD operando	Load	×	×	×	×	×
LDI operando	Load Immediate	×	×	×	×	×
ADD operando	Add	×	×	×	×	×
ADDI operando	Add Immediate	×	×	×	×	×
SUB operando	Subtract	×	×	×	×	×
SUBI operando	Subtract Immediate	×	×	×	×	×
BEQ operando	Branch On Equal		×	×	×	×
BNE operando	Branch on Not Equal		×	×	×	×
BGT operando	Branch Bigger Than		×	×	×	×
BGE operando	Branch Bigger or Equal		×	×	×	×
BLT operando	Branch Less Than		×	×	×	×
BLE operando	Branch Less or Equal		×	×	×	×
JMP operando	Jump		×	×	×	×
NOT	Not			×	×	×
AND operando	And			×	×	×
ANDI operando	And Immediate			×	×	×
OR operando	Or			×	×	×
ORI operando	Or Immediate			×	×	×
XOR operando	XOR			×	×	×
XORI operando	XOR Immediate			×	×	×
SLL operando	Shift Left			×	×	×
SRL operando	Shift Right			×	×	×
STOV operando	Store Vector				×	×
LDV operando	Load Vector				×	×
RETURN	Return				×	×
RETINT	Return Interruption					×
CALL	Call				×	×

Quadro 1: Comparativo de instruções da família BIP

Fonte: adaptado de Pereira(2008)

Mais detalhes sobre as instruções dos processadores BIP são apresentados no Apêndice A.

2.1.1μBIP

O processador μBIP é um dos representantes mais completos da família de processadores BIP, visto que o mesmo possui o maior número de instruções disponíveis. O mesmo apresenta suporte às interrupções, manipulação de vetores e entrada e saída de dados. De acordo com Pereira (2008), o mesmo foi concebido com o objetivo de estender a utilização da arquitetura do BIP para o curso de Mestrado em Computação Aplicada, no qual os microcontroladores são estudados de forma mais aprofundada. Assim sendo, o μBIP possui uma arquitetura e uma organização muito semelhantes aos microcontroladores comerciais, em especial os microcontroladores PIC16, da Microchip, que inspiraram o projeto do μBIP.

O μBIP estende a arquitetura dos processadores anteriores ao mesmo, herdando instruções do BIP I, BIP II, BIP III e BIP IV. Portanto, subentende-se que o estudo da arquitetura do μBIP implica também no estudo da arquitetura dos demais processadores BIP.

2.1.2 Especificação da arquitetura do μBIP

Segundo Pereira (2008) o tamanho da palavra de dados do μBIP é de 16 bits, permitindo assim o suporte a tipos inteiros com sinal de 16 bits, contemplando valores existentes entre -32768 e 32767.

O μBIP possui também um formato de instrução composto por 5 bits para o código da operação e 11 bits para o operando, o que permite endereçar até 2048 (2^{11}) instruções, utilizando para isso o operando de 11 bits para informar o endereço absoluto quando utilizado um desvio, sendo também de 11 bits o espaço de endereçamento da memória de dados (PEREIRA, 2008).

O espaço da memória de dados é dividido em duas partes: 1024 endereços para memória de dados propriamente dita, e 1024 endereços para entrada e saída mapeadas em memória. De acordo com Pereira (2008) o mapeamento da E/S em memória foi escolhido por ser uma abordagem mais simples, já que ela possibilita o uso das mesmas instruções para manipulação de memória implementadas no μBIP e nos processadores anteriores, o que economiza o número de instruções (PEREIRA, 2008).

O μBIP conta também com cinco registradores especiais, comuns a todos os processadores BIP, sendo eles *(i)* PC (Program Counter), que armazena o endereço da

instrução corrente; (*ii*) ACC (Accumulator), que mantém o resultado da Unidade Funcional, composta, no µBIP, por uma ULA e dois *barrel shiffers*, utilizados para realizar os deslocamentos associados às instruções SLL e SRL; (*iii*) STATUS, que contém informações sobre os resultados obtidos nas operações da ULA. Além destes, outros registradores específicos do µBIP são: (*iv*) INDR (Index Register), que registram o índice do vetor nas operações de manipulação de vetores; e (*v*) SP (Stack Pointer), que aponta para o topo da pilha utilizada para suporte à chamadas de procedimentos (PEREIRA, 2008).

O registrador STATUS ainda possui os seguintes *flags* no µBIP: (*i*) Z, que indica se o resultado da operação da ULA é zero ou não; (*ii*) N, que informa se o resultado obtido na operação da ULA é negativo ou não; e (*iii*) C, que indica se ocorreu um *carry-out* ou um *borrow* em instruções aritméticas (PEREIRA, 2008).

O µBIP possui também uma pilha que dá suporte à procedimentos com 8 endereços de profundidade, pilha esta utilizada pelas instruções CALL, RETURN e RETINT. Essa pilha armazena o próximo valor do registrador PC após a chamada de uma função pela instrução CALL ou após uma interrupção. Esse valor é restaurado no registrador PC quando ocorre uma chamada através das instruções de RETURN e RETINT.

Com exceção da instrução HLT e das instruções de desvio, sempre é considerado o registrador ACC como um operando implícito, a ser utilizado junto com o operando explicito definido pela instrução em uso (PEREIRA, 2008).

2.1.3 Simulação do µBIP

O trabalho de Pereira (2008) gerou como resultado não apenas o µBIP, mas também um conjunto de ferramentas úteis para simulação do µBIP. De acordo com o autor, foi criado um modelo da arquitetura do µBIP utilizando-se do ArchC.

De acordo com The ArchC Team (2007), o ArchC é uma linguagem de descrição de arquitetura (Architecture Description Language, ADL) baseada no SystemC e criada pelo Instituto de Computação da Universidade de Campinas (IC-UNICAMP), que possibilita a criação de ferramentas de software, dentre elas, simulador, montador, ligador e depurador, com base na descrição da arquitetura de um processador.

Esta ferramenta torna possível a simulação de grande parte dos recursos do µBIP, incluindo as interrupções do temporizador, não simulando porém as interrupções geradas pela subida de borda da porta 0 da entrada port_0.

Ainda no mesmo trabalho, Pereira (2008) criou um modelo do µBIP em VHDL. Tal modelo pode ser sintetizado em uma placa FPGA (Field Programable Gate Arrays), tornando possível também a verificação completa da funcionalidade de interrupções, necessária neste trabalho.

2.2 Microcontroladores PIC

Segundo Pereira (2002), os microcontroladores PIC são uma família de dispositivos fabricados pela Microchip. Os mesmos utilizam uma arquitetura RISC, frequências de clock de até 40Mhz, até 2048k word de memória de programa e até 3968 bytes de memória RAM. Ainda de acordo com o autor, é possível encontrar diversos periféricos internos nestes microcontroladores, como temporizadores e contadores, memórias EEPROM interna, geradores/comparadores/amostradores PWM, conversores A/D de até 12 bits, interfaces de barramento CAN, I2C, SPI, entre outros.

De acordo com Pereira (2008) grande parte dos conceitos arquiteturais e de organização do µBIP são inspirados na série PIC16, da Microchip. É possível notar semelhanças entre o microcontrolador da Microchip e o µBIP não apenas no conjunto de instruções, mas também no conjunto de registradores de uso especial dos dois componentes e no modo como são implementadas as interrupções.

2.2.1 Traduções de Instruções em Assembly do PIC16 para o µBIP

É possível traduzir as instruções assembly dos processadores PIC16 para instruções do µBIP. As instruções do PIC16 podem ser divididas, de acordo com Pereira (2002), em instruções de manipulação de registradores, instruções aritméticas, instruções de operações lógicas, instruções de desvio e instruções de controle.

O Quadro 2 mostra as instruções do PIC16, descrevendo o funcionamento geral das mesmas e mostrando qual ou quais instruções do µBIP realizam a mesma operação.

O registrador W é o registrador equivalente ao registrador ACC no μBIP, sendo responsável por armazenar o resultado das operações realizadas. Nota-se também que o registrador W é utilizado como um dos operandos nas operações lógicas e aritméticas (PEREIRA, 2002).

O PIC16 conta também com um registrador especial chamado STATUS, onde são armazenados flags que sinalizam as condições de algumas operações. As *flags* existentes no PIC16 são: *(i)* C (Carry/Borrow), utilizado para indicar a ocorrência de carry-out ou borrow em operações matemáticas e lógicas, *(ii)* DC (Digit Carry/Digit Borrow), utilizado para indicar a ocorrência de *carry-outs* ou *borrows* em grupos de 4 bits, e *(iii)* Z (Zero), utilizado para indicar se o resultado de uma função foi zero ou não (PEREIRA, 2002). Destes *flags* apenas o DC não está presente no μBIP.

Instrução no PIC16	**Descrição**	**Instruções no µBIP**
ADDW k	W = k + W	ADDI k
ADDWF f, d	Memória[d] = Memória[f] + W	ADD f // STO d
SUBLW k	W = k – W	SUBI k
SUBWF f, d	Memória[d] = Memória[f] – W	SUB f // STO d
ANDLW k	W = k AND W	ANDI k
ANDWF f, d	Memória[d] = Memória[f] AND W	AND f // STO d
IORLW k	W = k OR W	ORI k
IORWF f, d	Memória[d] = Memória[f] OR W	OR f // STO d
XORLW k	W = k XOR W	XORI k
XORWF f, d	Memória[d] = Memória[f] XOR W	XOR f // STO d
COMF f, d	Memória[d] = NOT Memória[f]	LD f // NOT // STO d
MOVLW k	W = k	LDI k
MOVWF f	Memória[f] = W	STO f
MOVF f, d	SE(d = 0 OU d= W) W = Memória[f]	LD f
	SE(d != 0) Memória[d] = Memória[f]	LD f // STO d
CLRF f	Memória[f] = 0	LDI 0 // STO f
CLRW	W = 0	LDI 0
INCF f, d	Memória[d] = Memória[f] + 1	LD f // ADDI 1 // STO d
DECF f, d	Memória[d] = Memória[f] – 1	LD f // SUB 1 // STO d
GOTO k	PC = k	JMP k
CALL k	ToS = PC / PC = k	CALL k
RETURN	PC = ToS	RETURN
RETLW k	PC = ToS / W = k	RETURN // LDI k
RETFIE	PC = ToS	RETINT
NOP	Não realiza nada	HLT

Quadro 2: Comparativo entre instruções do PIC16 e do µBIP

É possível ver que a maioria das instruções do PIC pode ser transcrita diretamente em até três operações do µBIP. Entretanto, algumas instruções do PIC requerem um método de conversão mais elaborado, devido às suas características específicas com relação ao tratamento dos bits das mesmas.

Instrução no PIC16	**Descrição**	**Instruções no µBIP**	**Exemplo**
BCF f, b	Apaga o bit representado pelo operando b do registrador representado pelo operando f	LD f // ANDI b_x // STO f	BCF 0x0A, 3
			LD 0x0A ANDI 11110111 STO 0x0A
BSF f, b	Seta o bit representado pelo operando b do registrador representado pelo operando f	LD f // ORI b_x // STO 0x0A	BSF 0x0A, 3
			LD 0x0A ORI 00001000 STO 0x0A
SWAPF f, d	Troca os nibbles do registrador f, armazenando o resultado no registrador representado por d	LD f // STO tmp// SLL 4 // STO tmp_2 // LD tmp // SLR 4 // OR tmp // STO d	
BTFSC f, b	Verifica se o bit indicado por b no registrador f está em nível 0. Em caso afirmativo, pula a próxima instrução.	LD f // ANDI b_x // BEQ PC+1	BTFSC 0xFA, 3
			LD 0xFA ANDI 00001000 BEQ PC+1
BTFSS f, b	Verifica se o bit indicado por b no registrador f está em nível 1. Em caso afirmativo, pula a próxima instrução.	LD f // ORI b_x // BEQ PC+1	BTFSS 0xFA, 3
			LD 0xFA ORI 11110111 BEQ PC+1
DECFSZ f, d	Decrementa o conteúdo de f e pula a próxima instrução caso o valor resultante seja igual a zero. Salva o valor resultante em d.	LD f // SUBI 1 // STO d // BEQ PC+1	DECFSZ 0x0D, 0x0F
			LD 0x0D SUBI 1 STO 0x0F BEQ PC+1
INCFSZ f, d	Incrementa o conteúdo de f e pula a próxima instrução caso o valor resultante seja igual a zero. Salva o valor resultante em d.	LD f // ADDI 1 // STO d // BEQ PC+1	INCFSZ 0x0D, 0x0F
			LD 0x0D ADDI 1 STO 0x0F BEQ PC+1
RRF f, d	Rotaciona 1 posição para a direita o valor do registrador f e armazena o resultado em Memória[d]	LD f // STO t1 // SRL 1 // STO t2 // LD t1 // SLL 15 // OR t2 // STO d	
RLF f, d	Rotaciona 1 posição para a esquerda o valor do registrador f e armazena o resultado em Memória[d]	LD f // STO t1 // SLL 1 // STO t2 // LD t1 // SRL 15 // OR t2 // STO d	

Quadro 3: Conversão especial de instruções do PIC para o µBIP

O Quadro 3 mostra um conjunto de instruções nas quais não é trivial a conversão de instruções oriundas do PIC. As instruções BCF e BSF, por exemplo, utilizam um campo para indicar o bit a ser alterado no registrador *f*, indicado por *b* na tabela. A instrução gerada para o µBIP, nesse caso, necessita de um imediato chamado aqui de *b_x*, que contém um valor a ser utilizado para *"setar"* ou limpar um valor no registrador *f* equivalente ao representado por *b* na instrução original. Em ambos os casos é utilizada uma instrução lógica para realizar a operação, como mostrado no exemplo do Quadro 3.

Outro detalhe interessante pode ser visto nas instruções de desvio BTFSS, BTFSC, DECFSZ e INCFSZ. Estas instruções funcionam de forma que, caso determinado bit de algum registrador esteja em estado 0 ou 1, as mesmas pulam uma instrução. Neste caso, o PC é incrementado em mais 1. Entretanto, não há como manipular o registrador PC no µBIP, o que obriga a utilização de uma outra abordagem quando ocorrerem estas instruções.

2.2.2 Interpretação de Instruções C do PIC16 para o µBIP

De uma forma geral, embora seja prático realizar a conversão de instruções Assembly do PIC16 em instruções Assembly do µBIP, existem restrições que tornam essa forma de conversão inviável.

A primeira restrição ocorre no caso de instruções de desvio. As instruções de desvio condicional do PIC16 são abordadas de forma a utilizar-se do registrador que contém o endereço da instrução atual (PC). Não é possível escrever uma instrução diretamente equivalente para o µBIP, o que torna necessária a análise da funcionalidade do código Assembly em questão para que seja possível escrever um comando equivalente.

A segunda restrição dá-se justamente na análise do código Assembly do PIC16. Os códigos-fonte analisados neste trabalho, salvo algumas partes específicas, são todos disponibilizados em linguagem C. Sendo assim, existe a necessidade de compilar tais códigos, gerando um código-fonte em Assembly do PIC16, para então realizar a tradução do mesmo para o Assembly do µBIP. Entretanto, tais códigos gerados pelos compiladores são pouco compreensíveis, visto que o compilador utiliza notações próprias e genéricas para identificar os endereços dos desvios. Todo o comentário feito pelo autor do código-fonte original é perdido, o que dificulta a análise do código.

Até o presente momento não existe um compilador C para o µBIP. Porém é possível interpretar o código-fonte escrito em C e traduzi-lo diretamente para o assembly do BIP manualmente. Tal abordagem possibilita a escrita de um código otimizado para o µBIP.

2.3 Sistemas Operacionais

De acordo com Tanembaum (2010), “os sistemas operacionais realizam basicamente duas funções não relacionadas: fornecer aos programadores de aplicativos […] um conjunto de recursos abstratos claros em vez de recursos confusos de hardware e gerenciar esses recursos de hardware”.

Dentre os elementos que compõe um sistema operacional está o agendador de tarefas, ou escalonador. De acordo com Oliveira, Carissimi e Toscani (2004), “em qualquer sistema operacional que implemente multiprogramação, diversos processos disputam os recursos disponíveis no sistema, a cada momento”. Segundo Deitel, Deitel e Choffnes (2005), “quando um sistema pode escolher os processos que executa, deve ter uma estratégia – denominada política de escalonamento [...] – para decidir quais processos executar em determinado instante”.

Um algoritmo de escalonamento pode ainda ser preemptivo ou não preemptivo. De acordo com Deitel, Deitel e Choffnes (2005), um algoritmo não preemptivo permite que um processo, uma vez possuindo o processador, não seja interrompido pelo mesmo. Em contrapartida, um algoritmo preemptivo faz com que seja possível o processo que possui o processador ser interrompido.

Uma das políticas de escalonamento possíveis é chamada de escalonamento por prioridade. De acordo com Oliveira, Carissimi e Toscani (2004), “quando os processos de um sistema possuem diferentes prioridades, essa prioridade pode ser utilizada para decidir qual processo é executado a seguir. Um algoritmo de escalonamento desse tipo pode ser implementado através de um alista ordenada conforme a prioridade dos processos.”.

Outra política comum é a política do escalonamento por alternância circular, mais conhecido também como escalonamento *round-robin*. Em um escalonamento *round-robin* “processos são despachados na ordem FIFO[1], mas recebem uma quantidade de tempo limitada

[1] FIFO – First In, First Out (Primeiro a entrar, primeiro a sair).

de tempo de processador denominada intervalo de tempo ou quantum.” (DEITEL; DEITEL; CHOFFNES, 2005). Quando o quantum de um processo acaba, o sistema causa uma interrupção e chama o próximo processo da fila, colocando o processo que sofreu interrupção no final da fila.

Neste texto preferiu-se utilizar a expressão agendador de tarefas ao invés de escalonador de tarefas. Ambas, escalonador e agendador, são traduções para o termo em inglês *scheduler*.

2.4 Sistemas Operacionais de Tempo Real

Um sistema operacional de tempo real difere de um sistema operacional convencional pelo fato de adicionar os seguintes princípios, mencionados por Stallings (2012):

1. Determinismo, no sentido de que um RTOS executa suas operações num período fixo ou conhecido de tempo;

2. Responsividade, significando que deve ser possível não apenas prever o período de execução de um processo, mas também prever o tempo necessário para servir a operação após uma interrupção ou solicitação

3. Controle do usuário, ou seja, um RTOS permite ao usuário controlar, por exemplo, o funcionamento dos algoritmos de escalonamento.

4. Confiabilidade, no que diz respeito à tolerância a faltas. Um RTOS deve ser capaz de contornar uma falha sem simplesmente reiniciar o sistema; e

5. Suporte a operações de falhas de software, podendo o sistema preservar as informações e dados do mesmo, buscando corrigir ou minimizar a falha.

Ainda segundo Tanembaum (2010), “esses sistemas são caracterizados por terem o tempo como um parâmetro fundamental”. Essas características norteiam o projeto de um sistema operacional de tempo real, indicando também suas limitações.

Em sistemas operacionais de tempo real é necessária a existência de determinismo nas tarefas devido à necessidade de manter o controle do tempo. Em um sistema operacional comum, geralmente não determinista, nem sempre é possível prever o termino de um processo,

o que é uma característica inerente dos sistemas operacionais de tempo real. Em contrapartida sistemas deterministas permitem a previsão do tempo de execução de cada tarefa existente no sistema operacional (SHAW, 2003).

Com relação à responsividade, um sistema operacional de tempo real deve prover um bom tempo de resposta à interrupções geradas durante a execução do mesmo. O motivo é o mesmo da necessidade de determinismo. Tempos de resposta à interrupções devem ser pequenos para não afetarem os programas a serem executados no sistema operacional de uma forma crítica, fazendo o sistema perder o sentido para o qual foi criado (STALLINGS, 2012).

A capacidade de prover um maior controle do usuário é outra característica importante de um sistema operacional de tempo real. Sistemas operacionais comuns não permitem, por exemplo, que a prioridade de um processo seja definida. Em um sistema operacional de tempo real é necessária uma maior capacidade de controle por parte do usuário no que diz respeito ao acesso à processos do sistema operacional (STALLINGS, 2012).

Confiabilidade é uma característica importante em um sistema operacional de tempo real devido à característica principal do sistema com relação ao tempo. Em casos de falhas a perda de performance pode ter consequências catastróficas para as aplicações que estão em execução, o que exige que o sistema operacional seja capaz de contornar falhas de forma rápida e eficiente, sem perda de performance (STALLINGS, 2012).

O suporte a falhas de software é a capacidade do sistema de manter a integridade dos dados que estão sendo armazenados no mesmo, tentando minimizar os problemas causados pela falha e buscando uma correção para o problema (STALLINGS, 2012).

Com relação ao tempo, Shaw (2003) diz que um sistema de tempo real pode ser considerado um sistema de tempo real estrito ou um sistema de tempo real tolerante, ou ainda, sistemas de tempo real críticos ou sistemas de tempo real não críticos

2.4.1 Sistemas de Tempo Real Críticos e Não Críticos

Sistemas de tempo real críticos, ou sistemas de tempo real estrito, são sistemas onde o existem prazos estritos que devem ser cumpridos, sob a penalidade de mal funcionamento do sistema. Um exemplo de sistema de tempo real crítico, de acordo com Shaw (2003), é o sistema de controle de um elevador, no qual, caso alguma restrição de tempo não seja

cumprida exatamente, o elevador possa parar entre dois andares, e não no andar especificado. Neste caso, a tarefa que está sendo executada deve ser realizada obrigatoriamente no tempo determinado.

Um sistema de tempo real não crítico, ou tolerante, é um sistema no qual existe um prazo menos rigoroso para uma tarefa a ser executada. Deve-se notar que embora o prazo não seja estrito, existe um prazo a ser respeitado. O sistema deve executar a tarefa preferencialmente no prazo especificado. Um bom exemplo de sistema não crítico é um sistema de telefone que falha ocasionalmente ao fazer uma comunicação (SHAW, 2003).

2.5 Sistemas Operacionais de Tempo Real Comerciais

Existem no mercado diversos sistemas operacionais de tempo real disponíveis. Este trabalho irá ater-se a discutir sistemas operacionais de tempo real que rodem sobre o PIC e que sejam soluções de código-fonte aberto. RTOS com essas características possuem, a princípio, maior possibilidade de serem portados para o μBIP.

Os sistemas operacionais a serem estudados a seguir consistem basicamente em um conjunto de funções, como agendadores de tarefas, gestores de processos, algumas implementações de pilhas, listas e outras estruturas de dados.

Outra característica comum aos sistemas estudados é que as funções e programas gerados pelo programador são “enxertados” no código-fonte do sistema operacional, devido à ausência de uma memória externa onde possam ser armazenados tais programas, para serem compilados posteriormente. Isso implica em um único arquivo hexadecimal, contendo não apenas o sistema operacional, mas também os programas criados para operar sobre o mesmo, a ser gravado no microcontrolador, o que facilita a criação de um sistema determinista.

2.5.1 OSA

O OSA é um sistema operacional de tempo real desenvolvido para microcontroladores PIC10, PIC12, PIC16, PIC18, PIC24 e dsPIC, além de controladores Atmel AVR de 8 bits e controladores da STMicroeletronics STM8.

De acordo com OSA(2008), o OSA é um sistema com suporte a múltiplas tarefas cooperativas. O OSA conta com 126 funções (mais 22 funções consideradas antigas, não sendo recomendado o uso das mesmas), divididas da seguinte forma:

- Funções de sistema (19 funções);
- Controle de tarefas (11 funções);
- Semáforos binários (6 funções*);*
- Semáforos de contagem (10 funções);
- Ponteiros para mensagens (8 funções*);*
- Mensagens simples (8 funções);
- Filas de ponteiros de mensagens (9 funções*);*
- Filas de mensagens simples (9 funções);
- Flags (16 funções);
- Funções para temporizadores de tarefas (9 funções*);*
- Funções para temporizadores estáticos (12 funções);
- Filas de temporizadores (9 funções).

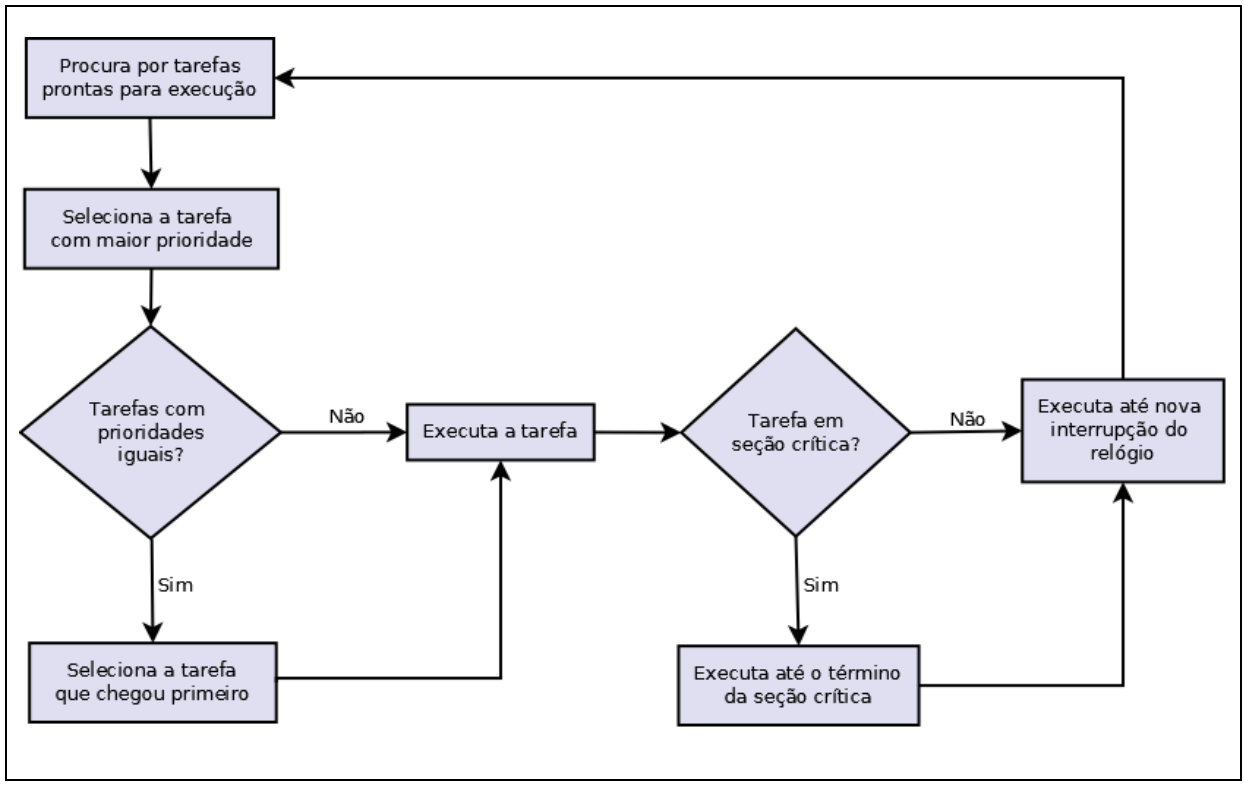

Figura 1: Diagrama de funcionamento do agendador de tarefas do OSA

Segundo OSA (2008) o agendador de tarefas do OSA primeiro verifica quais as tarefas do sistema que estão prontas para serem executadas. Cada tarefa possui uma prioridade, sendo 0 a maior prioridade e 7 a menor prioridade. Após realizar essa verificação, o agendador seleciona a tarefa com maior prioridade. Caso existam duas tarefas com a mesma prioridade, a primeira a estar pronta é a primeira que é executada. Esse processo é realizado para todas as tarefas, com exceção apenas de tarefas que entrem na dita “sessão crítica”. Tais tarefas são priorizadas e monopolizam o uso do processador. A Figura 1 mostra o funcionamento do agendador.

2.5.2 FreeRTOS

O FreeRTOS é um sistema operacional de tempo real genérico que roda, entre outros microcontroladores, também em microcontroladores PIC18 (FREERTOS, 2012).

O FreeRTOS é um sistema operacional que funciona em diversos microcontroladores, entre eles, no PIC18 (FREERTOS, 2012). É o que mais possui documentação disponibilizada.

O FreeRTOS possibilita a escolha de duas políticas de escalonamento. A primeira, dita preemptiva, é a mesma política de escalonamento do OSA, diferindo apenas no que diz

respeito aos níveis de prioridade de cada tarefa. Tarefas podem ter prioridade entre 3 (no caso, a maior prioridade) e 1, sendo que a prioridade 0 indica uma tarefa ociosa (FREERTOS, 2012).

A segunda política de escalonamento possível para o FreeRTOS é uma política dita cooperativa, onde ocorre a troca de contexto apenas se uma tarefa é bloqueada ou se uma função de troca de contexto (*taskYELD()*) é chamada (FREERTOS, 2012).

A função *taskYELD()* faz parte da API (Application Programming Interface) de funções do FreeRTOS, composta por 87 funções, divididas nas seguintes categorias:

- Criação de tarefas (3 funções);
- Controle de tarefas (7 funções);
- Utilidades de tarefas (15 tarefas);
- Controle do *Kernel* do RTOS (9 tarefas);
- Funções específicas para controle de MPU (Memory Protected Units – Unidades Protegidas de Memória) (3 funções);
- Funções para controle de fila (18 funções);
- Funções específicas para semáforos/mutexes (11 funções);
- Controle de Timers de Software (13 funções); e
- Funções de Co-rotinas (8 funções) (FREERTOS, 2012).

Por ser um sistema voltado para microcontroladores mais complexos que o PIC16, o seu tamanho é maior, contendo funções que, se traduzidas para o µBIP, sobrepujariam as limitações do mesmo, sendo estudado neste trabalho apenas para verificar possíveis métodos e funções que pudessem ser utilizadas no BIP/OS.

2.5.3 BRTOS

O BRTOS (Brazilian Real-Time Operating System) é um sistema operacional de tempo real desenvolvido para microcontroladores de pequeno porte. Dentre os microcontroladores compatíveis, está o PIC18.

Segundo BRTOS (2012a) o BRTOS é um sistema operacional com escalonamento preemptivo por prioridades, o que torna obrigatória a associação de uma prioridade a cada tarefa a ser executada pelo sistema. O BRTOS também suporta a instalação de até 32 tarefas. O BRTOS ainda conta com alguns recursos de gerenciamento como semáforos, mutex, caixas de mensagens e filas. De acordo com BRTOSa (2012) “O mutex utiliza o protocolo *priority ceiling* com o intuito de evitar *deadlocks* e inversões de prioridade”.

De acordo com BRTOS (2012b), a API do BRTOS possui 37 funções para controle dos seguintes recursos:

- Serviços do Núcleo do BRTOS (14 tarefas);
- Semáforos (4 tarefas);
- Mutex (4 tarefas);
- Caixa de mensagem (4 tarefas);
- Filas (6 tarefas); e
- Filas dinâmicas (5 tarefas).

O BRTOS pode ser considerado um sistema simples, leve e com baixa ocupação de memória, recursos estes considerados essenciais neste tipo de projeto. Segundo BRTOS (2012a) o sistema possui tamanho total próximo de 100 bytes e ocupa até 2KB da memória de programa quando utilizadas as configurações mínimas.

2.5.4 µGNU/RTOS

De acordo com uGNU/RTOS (2012), o µGNU/RTOS apresenta um Shell que deve ser utilizado através da porta RS232. O Shell permite a execução de um número limitado de comandos, listados no Quadro 4:

Dentre os sistemas operacionais estudados para este projeto, o µGNU/RTOS é o mais simples de todos. No código-fonte disponibilizado no site do projeto temos um pequeno conjunto de 11 arquivos contendo um arquivo principal, uma biblioteca de definições, alguns códigos específicos para comandos de entrada e saída e um pequeno Shell.

Comando	**Descrição**
--help	Apresenta um help mínimo no prompt de comando
-movlw [valor]	Copia o valor expresso em *valor* para o registrador W. *valor* neste contexto é um número decimal
-movwf [endereço]	Copia o valor do registrador W para o endereço *endereço* expresso na função.
-macro	Executa uma macro

Quadro 4: Funcionalidades do µGNU/RTOS

Além de ser o menor dos sistemas operacionais de tempo real analisados, o µGNU/RTOS também é o mais limitado, apresentando o menor número de funcionalidades.

É possível editar os arquivos do código-fonte do µGNU/RTOS para adicionar novas macros e novas funcionalidades, mas as mesmas apenas serão executadas em série, sem realizar múltiplas tarefas.

2.5.5 PICOS18

O PICOS18 é um sistema operacional de tempo real projetado para o PIC18. Segundo PICOS18 (2012) o PICOS18 é um sistema baseado no padrão OSEK/VDX, um padrão aberto de desenvolvimento voltado para a indústria automotiva e robótica.

De acordo com PICOS18 (2012), o PICOS18 possui como diferencial a utilização de instruções de manipulação de pilha e manipulação do registrador PC.

Ainda de acordo com PICOS18 (2012), é possível utilizar nesse sistema operacional dois tipos de agendamento de tarefas. O agendamento de tarefas pode acontecer de forma cooperativa, com chamadas ao agendador no meio do código que está sendo executado, ou pode ocorrer de modo preemptivo, em caso de interrupções.

O modo de funcionamento do agendador de tarefas é semelhante ao utilizado nos demais sistemas operacionais encontrados neste trabalho. Caso não existam tarefas com

diferentes prioridades, o sistema procede com uma política de escalonamento *Round Robin*, escolhendo a próxima tarefa da fila. Caso contrário, a tarefa com maior prioridade tem preferencia para assumir o processador.

De acordo com PRAGMATEC (2006) a API do PICOS18 possui 35 funções, divididas em:

- Gerenciamento de tarefas (7 funções);
- Gerenciamento de eventos (5 tarefas);
- Gerenciamento de alarmes (6 tarefas);
- Serviços do *Kernel* (3 tarefas);
- Gerenciamento de interrupções (6 tarefas);
- Gerenciamento de recursos (3 tarefas); e
- Rotinas de interrupção (5 tarefas).

2.5.6 Comparativo dos Sistemas Operacionais de Tempo Real

Conforme analisados nas seções, os sistemas operacionais de tempo real consistem basicamente de um agendador de tarefas e um gestor de processos que controlam as tarefas que serão executadas no microcontrolador.

Nem todos os sistemas estudados possuem capacidade para rodar no µBIP, como é o caso do FreeRTOS. Isso se deve ao fato de não ser possível armazenar o sistema de forma completa na memória do µBIP.

Alguns dos sistemas estudados neste trabalho são inadequados para portabilidade por se utilizarem de recursos que não estão disponíveis no µBIP. Neste caso, o exemplo é o PICOS18, que utiliza instruções de manipulação do registrador PC, o que não é possível ser feito no µBIP.

O Quadro 5 mostra um comparativo entre os sistemas operacionais de tempo real estudados neste trabalho.

	OSA	**FreeRTOS**	**BRTOS**	**µGNU/RTOS**	**PICOS18**
Escalonamento	Round Robin com prioridade	Round Robin com prioridade/ Cooperativo	Round Robin com prioridade	_	Round Robin com prioridade/ Cooperativo
Funções API	126 funções	87 funções	37 funções	4 funções	35 funções
Presença de gestor de Processos	Sim	Sim	Sim	-	Sim

Quadro 5: Comparativo entre os sistemas operacionais de tempo real estudados

Constatou-se também que os sistemas estudados podem ser tanto sistemas operacionais de tempo real crítico quanto sistemas operacionais de tempo real não-críticos. Tal definição depende da tarefa a ser executada pelo sistema.

Após a análise decidiu-se portar não um sistema operacional inteiro, mas partes interessantes de cada sistema operacional. A escolha, neste caso, foi realizada da seguinte forma:

- O algoritmo de escalonamento do OSA é o mais compreensível dos RTOS estudados, sendo portanto, o algoritmo a ser portado para o BIP/OS;
- As funções de gerenciamento de tarefas da API do PICos18 são as mais simples, o que justifica a escolha das mesmas;
- É válido criar uma rotina de tratamento de interrupções e um gestor de tarefas próprios para o BIP/OS, devido à incompatibilidade destes itens com o µBIP.

3 DESENVOLVIMENTO

O principal objetivo deste trabalho foi criar um sistema operacional de tempo real usável e compreensível. Embora ainda não exista um compilador C com o qual seria possível criar um sistema operacional nesta mesma linguagem, acredita-se que a linguagem assembly do µBIP foi empregada de modo suficientemente didático e compreensível.

Logo, este capítulo descreve o desenvolvimento do BIP/OS do modo como entendeu-se ser o mais claro e didático para fins de utilização na disciplina de sistemas operacionais das graduações em Ciência da Computação e Engenharia da Computação do Campus Itajaí da UNIVALI.

3.1 Análise de Requisitos

Na fase de projeto foram levantados os seguintes requisitos, necessários para a criação do sistema operacional BIP/OS. Estão divididos entre requisitos funcionais e requisitos não funcionais.

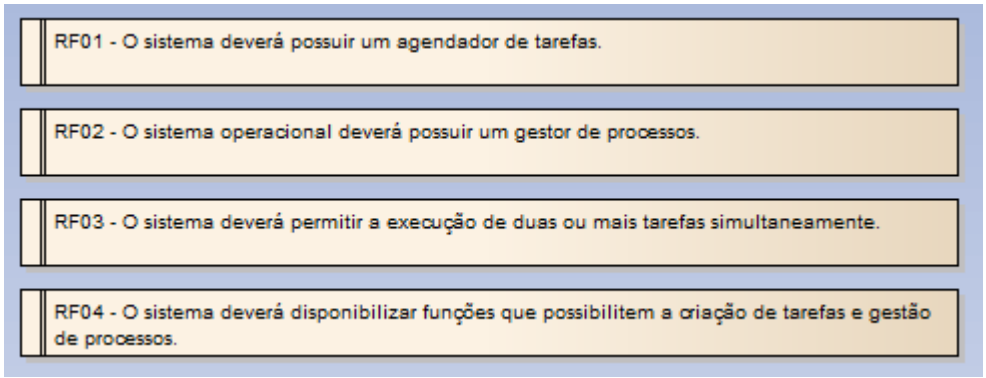

Figura 2: Requisitos funcionais do BIP/OS

Os requisitos funcionais, mostrados na Figura 2, dizem respeito às funcionalidades que estarão disponíveis no BIP/OS.

Primeiramente, o sistema possui um agendador de tarefas (também conhecido como escalonador de tarefas) que troca as tarefas em execução no sistema. Este agendador utiliza como estratégia de escalonamento a mesma utilizada pelo PICOS18, ou seja, um agendador de tarefas que escalona os processos utilizando um algoritmo *Round-Robin*, ou de revezamento circular associado a um critério de prioridade onde um identificador de processo define sua posição na fila. O segundo requisito funcional é a estrutura que descreve as tarefas

em execução no sistema. Neste caso, o gestor de processos desempenha a função específica de verificar quais tarefas estão em execução e quais são as próximas a serem executadas.

O terceiro requisito indica que o sistema deve ser multitarefa. Dois ou mais processos devem poder rodar “simultaneamente” no processador.

O quarto requisito funcional indica que devem ser disponibilizadas macros para a criação e gestão de tarefas. Esse item é importante porque o mesmo cumpre o requisito de que um sistema operacional deve ser uma interface entre o processador e os programas do usuário.

Os requisitos não-funcionais deste projeto, listados na Figura 3, definem basicamente a forma como será feito e testado o sistema operacional BIP/OS.

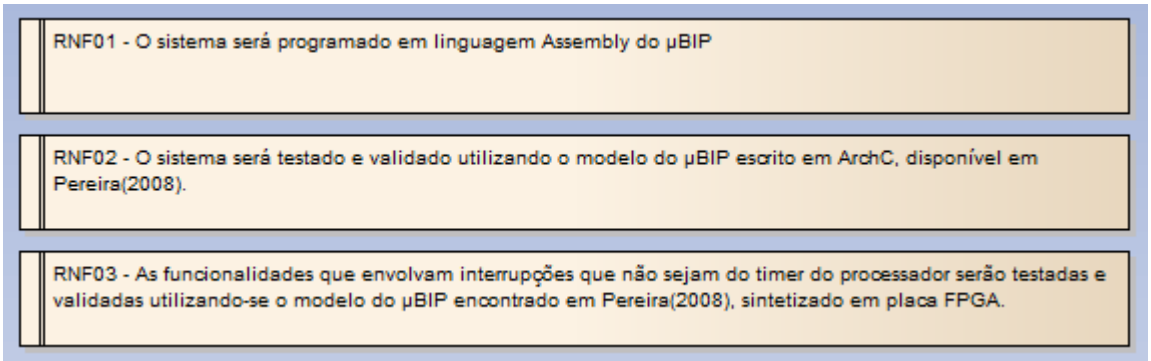

Figura 3: Requisitos não funcionais do BIP/OS

Primeiramente, o sistema foi implementado utilizando-se a linguagem Assembly do µBIP. Conforme demonstrado no início deste capítulo, não existem compiladores para as linguagens de alto nível disponíveis para o µBIP. Embora exista uma IDE criada para a arquitetura BIP que permite a criação de código-fonte em Portugol (RECH, 2011), a mesma possui suporte apenas para os processadores BIP I ao IV, não suportando a criação de um sistema operacional em Portugol para o µBIP. Sendo assim, justificou-se a utilização do Assembly como linguagem para a criação do sistema operacional.

O segundo e o terceiro requisitos não-funcionais dizem respeito à forma como o sistema foi testado e validado.

Pereira (2008) aponta a impossibilidade de utilizar o ArchC para a simulação de interrupções geradas pela subida da borda do pino 0 do registrador port_0, já que o ArchC não possui suporte para a geração de sinais externos ao processador, sendo então necessária outra

forma de simulação que não apenas o modelo escrito em ArchC para uma total validação do sistema operacional. Para tal, fez-se útil o modelo sintetizável do µBIP escrito em VHDL, visto ter sido possível sintetizá-lo em uma placa FPGA para então testar as interrupções e validar o terceiro requisito não-funcional.

Resumidamente, o sistema operacional implementado possui uma rotina de tratamento de interrupções, um agendador de tarefas e uma API básica dispostos na memória do µBIP como mostra a Figura 4.

Essa estrutura foi alocada na memória de programa do µBIP, sendo suficientemente pequena para não ultrapassar o limite de 2K instruções disponíveis em sua memória. Em vista deste fato, escolheu-se utilizar 1K instruções para descrever o sistema operacional e todos os seus recursos, deixando livres 1K instruções para programas feitos pelo usuário.

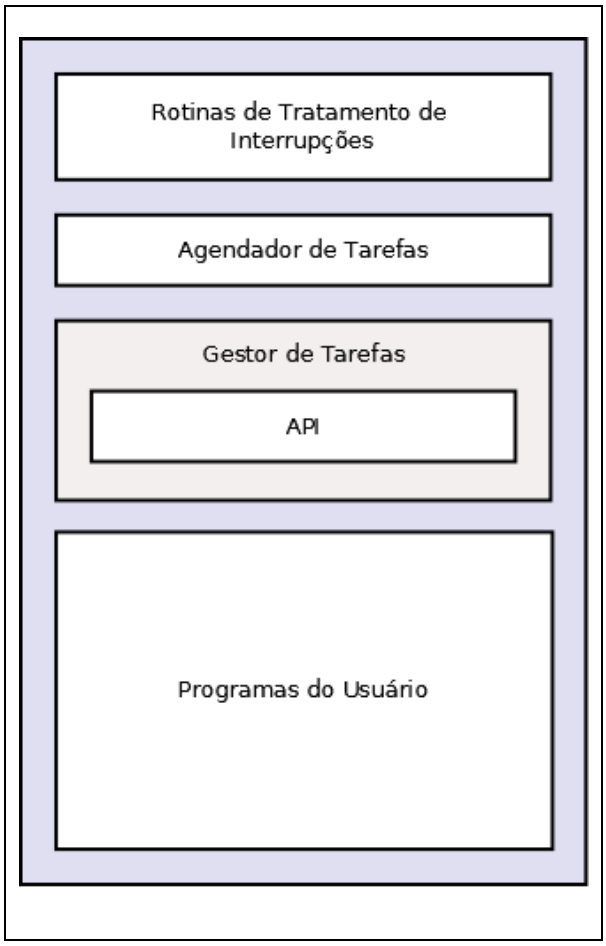

Figura 4: Distribuição de recursos na memória de programa do µBIP

3.1.1 Alterações realizadas no modelo ArchC do µBIP

Após análise dos recursos necessários para a execução de um sistema operacional no µBIP definiu-se que os seguintes itens seriam adicionados ao modelo do µBIP escrito em ArchC:

- Adição das instruções PUSH, POP e JR (*Jump Register*);
- Definição de um endereço contendo os valores do topo da pilha (ToS – Top of Stack) e do ponteiro da pilha (STKPTR)

As alterações foram adicionadas ao modelo descrito em PEREIRA(2008) para o µBIP, criando-se uma nova versão deste modelo.

```
01 void ac_behavior( push )
02 {
03     //Operation
04     ac_word operand1 = ACC.read();
05     STACK.push(operand1);
06     DataMemoryWrite(memRAM,TOS,STACK.top());
07     DataMemoryWrite(memRAM,STKPTR,STACK.stkptr());
08     //Increment PC
09     inc_PC(ac_pc);
10     //Output information
11     sprintf(strBuffer, "PUSH 0x%04X", operand1);
12     myPrint("push", operand, strBuffer);
13 }
14
15 void ac_behavior( pop )
16 {
17     //Operation
18     ac_word operand1 = STACK.pop();
19     DataMemoryWrite(memRAM,TOS,STACK.top());
20     DataMemoryWrite(memRAM,STKPTR,STACK.stkptr());
21     ACC.write(operand1);
22     //Increment PC
23     inc_PC(ac_pc);
24     //Output information
25     sprintf(strBuffer, "POP 0x%04X", operand1);
26     myPrint("pop", operand, strBuffer);
27 };
28
29 void ac_behavior( jr )
30 {
31     ac_word operand1 = DataMemoryRead(memRAM, operand);
32     sprintf(strBuffer, "PC=0x%04X",operand1);
33     myPrint("jr", operand1, strBuffer);
34     //Update PC
35     set_PC(ac_pc, operand1);
36 }
```

Quadro 6: Instruções PUSH, POP e JR, adicionadas no arquivo ubip_isa.cpp

No arquivo ubip_isa.cpp foram adicionadas as linhas de código contidas no Quadro 6. A instrução PUSH empilha o valor contido no registrador ACC no topo da pilha de procedimentos do μBIP. A instrução POP realiza o oposto, desempilhando o topo da pilha de procedimentos e salvando o valor desempilhado no registrador ACC.

A instrução JR desvia o endereço atual para um endereço armazenado em uma posição da memória, posição esta indicada pelo operando da instrução. Se, por exemplo, o endereço 0x234 contiver o valor 0x123 e for escrita a instrução JR 0x234 o registrador PC irá receber o valor contido no endereço 0x234, ou seja, desviará para 0x123.

Também foram definidos dois endereços específicos, contidos no arquivo `ubip_address.H`. Esse endereços foram o endereço do topo da pilha (ToS) no endereço 0x433 e o endereço do ponteiro da pilha (STKPTR) no endereço 0x434.

3.1.2 Desenvolvimento do Kernel do BIP/OS

O sistema operacional proposto é composto de 4 estruturas básicas: 1) Um conjunto de rotinas para tratamento de interrupções; 2) Um agendador de tarefas; 3) Um gestor de processos; e 4) Uma API para o sistema operacional.

Por ordem de necessidade, primeiramente foi criada uma API básica para o sistema. Esta API conterá funções para criação, pausa, retorno e encerramento de tarefas, além de uma função para remoção da tarefa da lista de tarefas ativas.

Tais funções implicam na adição de duas funcionalidades inexistentes no µBIP, necessárias para a troca de contexto e gerenciamento das tarefas. Atualmente não é possível obter o valor corrente do registrador PC no µBIP, o que significa ser impossível manter o registro do último valor do registrador PC em cada tarefa que está sendo executada no processador.

Anteriormente, pensou-se em criar duas instruções novas para o µBIP, LPC e SPC (Load PC e Store PC) para controlar os valores do registrador PC. Após uma análise mais aprofundada constatou-se que a criação de tais instruções não resolveria o problema de uma forma completa.

Para poder trocar o contexto de uma tarefa é necessário salvar o contexto da tarefa que está em execução. Por contexto, entende-se o conjunto de registradores que a tarefa está utilizando, além dos valores que a mesma colocou na pilha de procedimentos. As instruções LPC e SPC apenas manipulariam os valores do registrador PC, deixando de lado as informações sobre os outros registradores.

A solução encontrada para esse problema foi a criação de duas instruções para a manipulação da pilha de procedimentos.

Trocas de contexto são solicitadas por funções chamadas após uma interrupção. As interrupções do µBIP desviam o registrador PC para o endereço 0x01, salvando o valor de retorno no topo da pilha de procedimentos, e a partir deste ponto é chamada uma função para tratar a interrupção. Esse fato por si só já invalida a criação de uma instrução LPC, pois a mesma seria criada para ser chamada no momento após a interrupção, momento este em que o registrador PC sempre possui o valor 0x01.

Como o próximo valor do registrador PC, no momento imediatamente após a interrupção, está no topo da pilha, basta desempilhar o valor para se obter o PC da tarefa que estava em execução, valor este que poderia ser salvo em qualquer outro local da memória de dados que fosse acessível por meio de instruções LD.

Contexto de tarefa

O contexto de uma tarefa no μBIP é composto por sua pilha de procedimentos, pelo registrador STATUS, pelo registrador INDR, pelo registrador ACC e pelo registrador PC. Outros endereços da memória também devem ser salvos como contexto de uma tarefa, como os registradores port0_dir, port0_data, port1_dir e port1_data.

Definido este contexto e garantindo-se que o mesmo seja devidamente salvo após uma interrupção é possível retornar à tarefa que foi interrompida de forma que nenhuma informação necessária para a sua execução seja perdida. A tarefa retorna exatamente para o local e estado no qual estava antes da interrupção.

O contexto da tarefa armazenaria também informações sobre o PC inicial e final da tarefa, além de seu status e da sua prioridade. É possível visualizar a estrutura na Figura 5.

Figura 5: Estrutura de armazenamento de contexto da tarefa

No contexto do BIP/OS essa estrutura é chamada de CONTEXT_TABLE, ou tabela de contexto, e toda tarefa que executa suas operações no BIP/OS possui uma CONTEXT_TABLE. O contexto de cada tarefa ocupa 13 palavras na memória de dados.

3.1.3 API do BIP/OS

O BIP/OS possui em sua API 6 funções específicas para a gestão de tarefas e mais 4 funções auxiliares. As seções a seguir descrevem cada uma dessas tarefas.

Função OS_TSK_CREATE

A função OS_TSK_CREATE é responsável por cadastrar as informações básicas de uma tarefa na CONTEXT_TABLE. É impossível retornar o estado de uma tarefa após a interrupção sem a mesma estar cadastrada na CONTEXT_TABLE.

Também é a função OS_TSK_CREATE que define o identificador único da tarefa. Esse identificador único varia de 0x0 a 0xF, o que significa que o BIP/OS pode executar até 16 tarefas em pseudo-paralelismo.

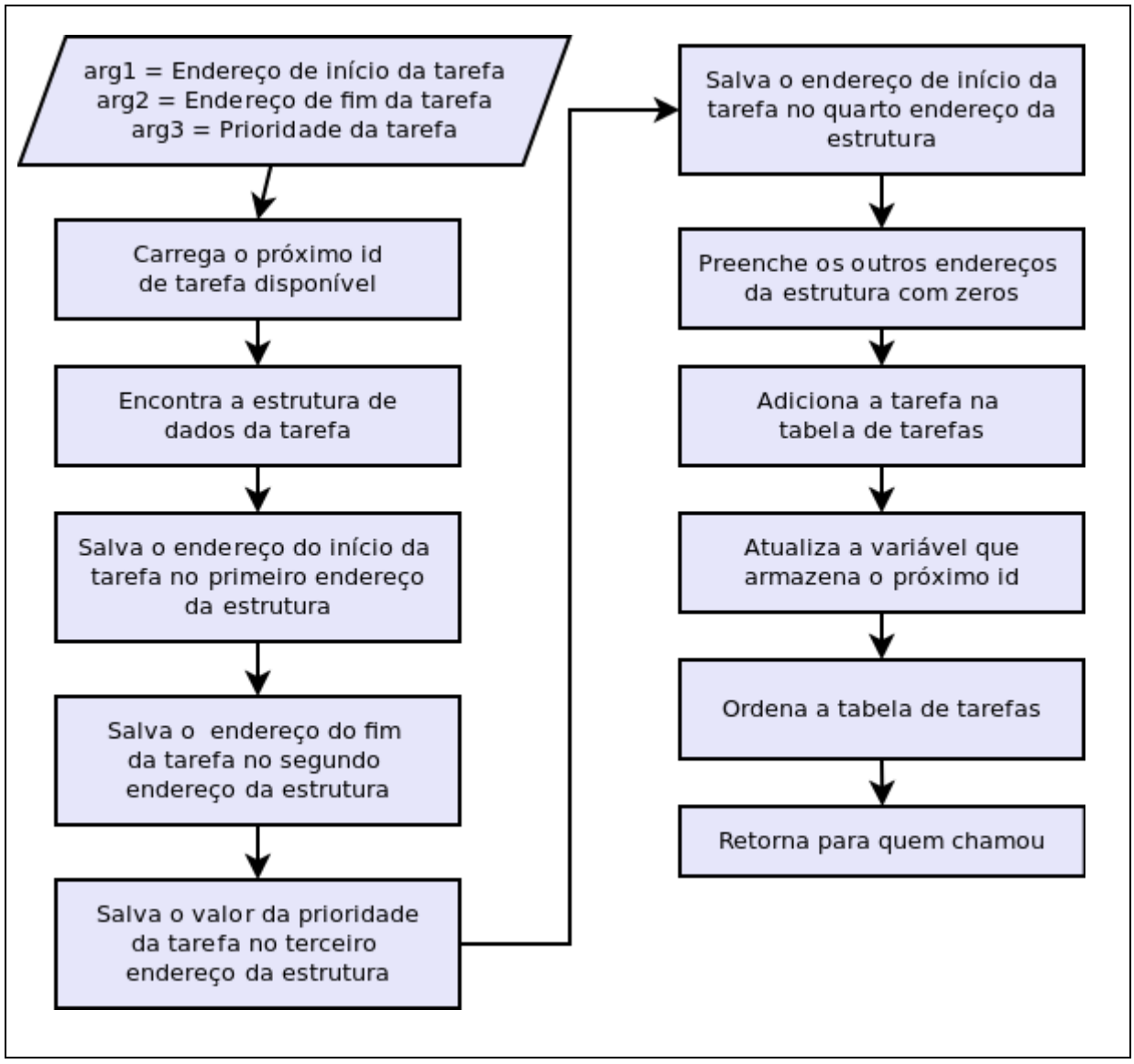

Figura 6: Fluxograma de funcionamento da função OS_TSK_CREATE

A Figura 6 mostra como funciona a função OS_TSK_CREATE. A mesma recebe como argumentos o endereço inicial, o endereço final e a prioridade da tarefa. Em posse desses argumentos, a função carrega o próximo identificador de tarefa disponível. O valor do identificador de tarefas está contido em uma variável global que é incrementada a cada criação de tarefas, iniciando em 0x0 e podendo chegar à 0xF.

Após identificar o próximo identificador disponível, a função encontra a estrutura de dados da respectiva tarefa. O endereço da CONTEXT_TABLE de uma tarefa é composto pelo seu identificador deslocado em 4 posições para a esquerda, acrescido do endereço 0x700, ou seja, (id << 0x4) + 0x700.

O primeiro endereço da CONTEXT_TABLE é reservado para o endereço de início da tarefa, neste caso, armazenado no primeiro argumento enviado para a função.

O segundo endereço da CONTEXT_TABLE, que pode ser acessado no endereço 0x7X1, onde X é o identificador da tarefa, contém o endereço do fim da tarefa.

O terceiro argumento enviado para a função OS_TSK_CREATE, a prioridade da tarefa, é armazenado pela mesma no endereço 0x7X2. A prioridade da tarefa varia de 0 a 3, sendo que o menor valor é a maior prioridade.

Depois de salvar a prioridade da tarefa a função OS_TSK_CREATE armazena no endereço 0x7X3 o primeiro argumento, o endereço inicial da tarefa. Esse endereço será também o primeiro valor do PC dessa tarefa.

Os demais campos da CONTEXT_TABLE são preenchidos com zeros.

Em seguida a tarefa é cadastrada em um vetor, utilizado para armazenar as tarefas que estão em execução. Esse vetor é chamado, no contexto do BIP/OS, de TASK_TABLE, e é consultado pelo agendador de tarefas para verificar quais as próximas tarefas a serem executadas. Apenas tarefas não finalizadas são mantidas na TASK_TABLE.

O endereço base da TASK_TABLE é o endereço 0x5B0. Para armazenar uma informação na TASK_TABLE função OS_TSK_CREATE leva em consideração uma variável global chamada tsk_quantity, utilizando a mesma como índice. Essa variável começa em 0 e armazena a quantidade de tarefas que não estão encerradas no sistema. Quando uma tarefa é criada essa variável é incrementada. Após o encerramento de uma tarefa, essa variável é decrementada.

Depois de cadastrar a tarefa na TASK_TABLE a função OS_TSK_CREATE incrementa o valor do próximo identificador e chama uma função de ordenação, a função BUBBLE_SORT. A função BUBBLE_SORT nada mais é do que um algoritmo bubble sort implementado para ordenar o vetor 0x5B0. Depois de ordenado, a função OS_TSK_CREATE executa um comando RETURN.

Função OS_TSK_PAUSE

A função OS_TSK_PAUSE é responsável por realizar a pausa e o salvamento de contexto de uma tarefa. Ela é chamada pelo agendador antes de iniciar uma nova tarefa.

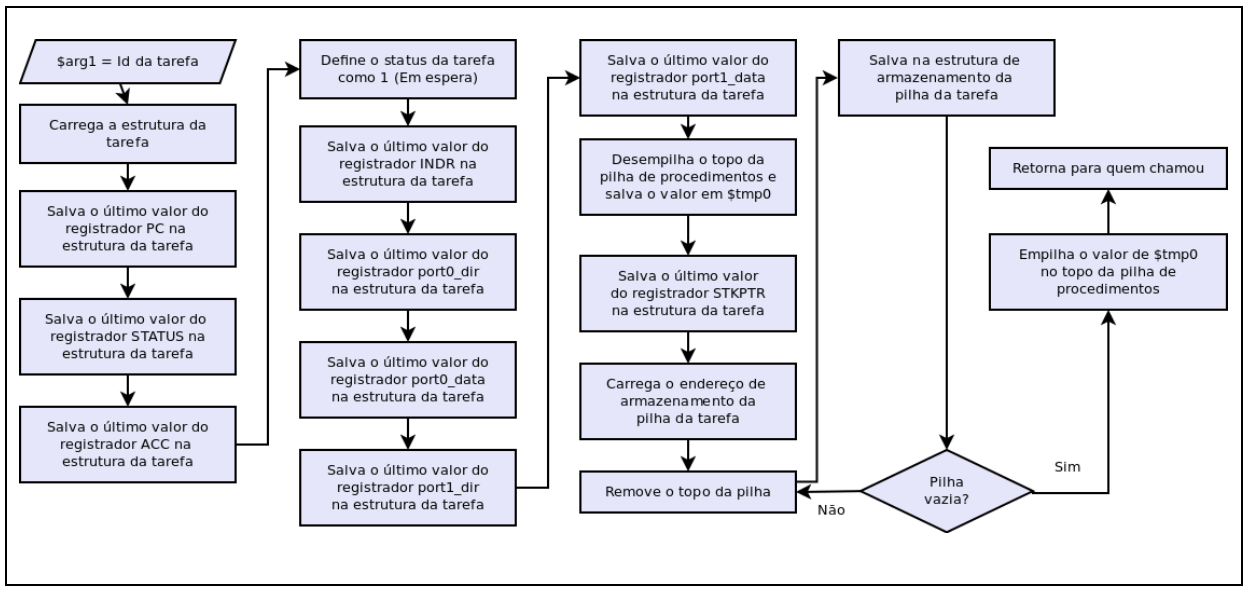

Figura 7: Fluxograma de funcionamento da função OS_TSK_PAUSE

A função OS_TSK_PAUSE recebe como argumento o identificador da tarefa a ser pausada. Com base nesse argumento a tarefa carrega a CONTEXT_TABLE daquela tarefa e salva o último valor do registrador PC no endereço 0x7X3.

Após esse processo, são salvos os registradores STATUS no endereço 0x7X4 e ACC no endereço 0x7X5.

Em seguida o valor 1 é armazenado no endereço 0x7X6, que contém o estado da tarefa. Uma tarefa pode assumir apenas 4 estados: 0 para “criada”, 1 para “em espera”, 2 para “em execução” e 3 para “finalizada”.

Após alterar o estado da tarefa, os registradores INDR e os valores das portas port0_dir, port0_data, port1_dir e port1_data são armazenados respectivamente nos endereços 0x7X7, 0x7X8, 0x7X9, 0x7XA e 0x7XB.

Realizado este processo a função OS_TSK_PAUSE desempilha um valor do topo da pilha. Este valor é o endereço para o qual a função OS_TSK_PAUSE deve retornar após executar. Esse valor é salvo em uma variável temporária para que não se misture com o contexto da pilha de procedimentos da tarefa.

Feito esse processo, o registrador STKPTR, que contém o ponteiro da pilha, é salvo no endereço 0x7XC.

O processo realizado a seguir consiste na remoção dos itens da pilha de procedimentos. Esses itens serão armazenados em uma estrutura mantida pelo BIP/OS chamada STACK_CONTEXT_TABLE. Foi utilizada uma estrutura separada para salvar a pilha de procedimentos porque assim o processo de salvamento deixaria menos espaços inutilizados entre o contexto de cada tarefa. O acesso à STACK_CONTEXT_TABLE é realizado carregando-se o identificador da tarefa, deslocando esse valor 3 bits para a esquerda, e somando ao endereço 0x680. Esse processo é exemplificado na Figura 8.

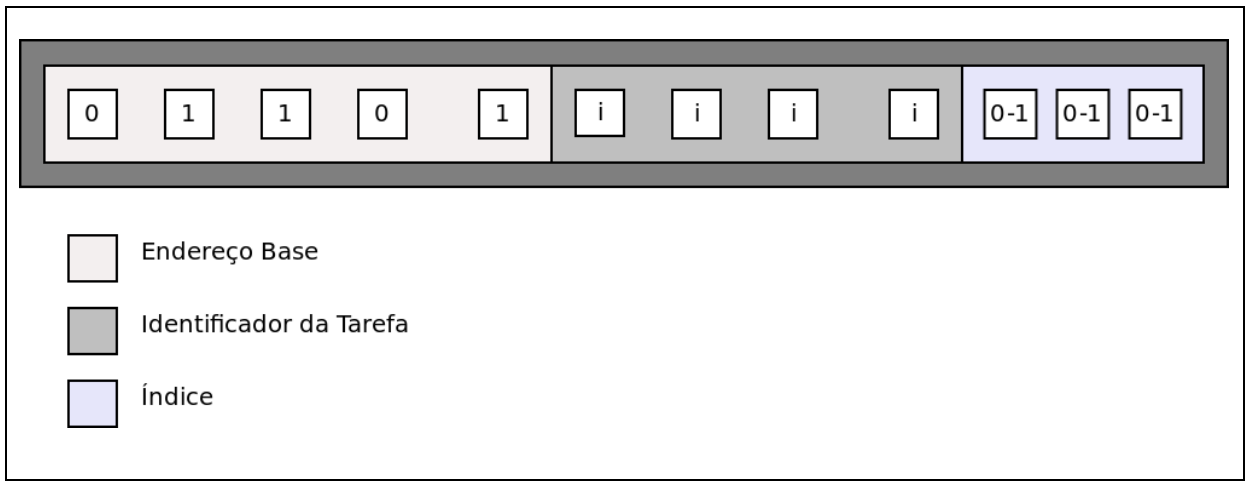

Figura 8: Formação do endereço para inserção de dados na STACK_CONTEXT_TABLE

A pilha de procedimentos então é limpa, tendo seu conteúdo armazenado na STACK_CONTEXT_TABLE. Após esse processo o endereço de retorno da tarefa é colocado de volta na pilha e a função chama uma instrução de RETURN.

Note que os valores dos registradores PC, ACC, STATUS e INDR foram previamente salvos, na rotina de interrupção, em variáveis globais, o que mantém os seus valores iguais aos valores existentes antes da interrupção da tarefa.

Função OS_TASK_RETURN

A função OS_TASK_RETURN é responsável por retornar o estado de uma tarefa anteriormente pausada. Isso implica em retornar os valores de cada registrador para o seu devido local, permitindo que a tarefa pausada retome suas atividades.

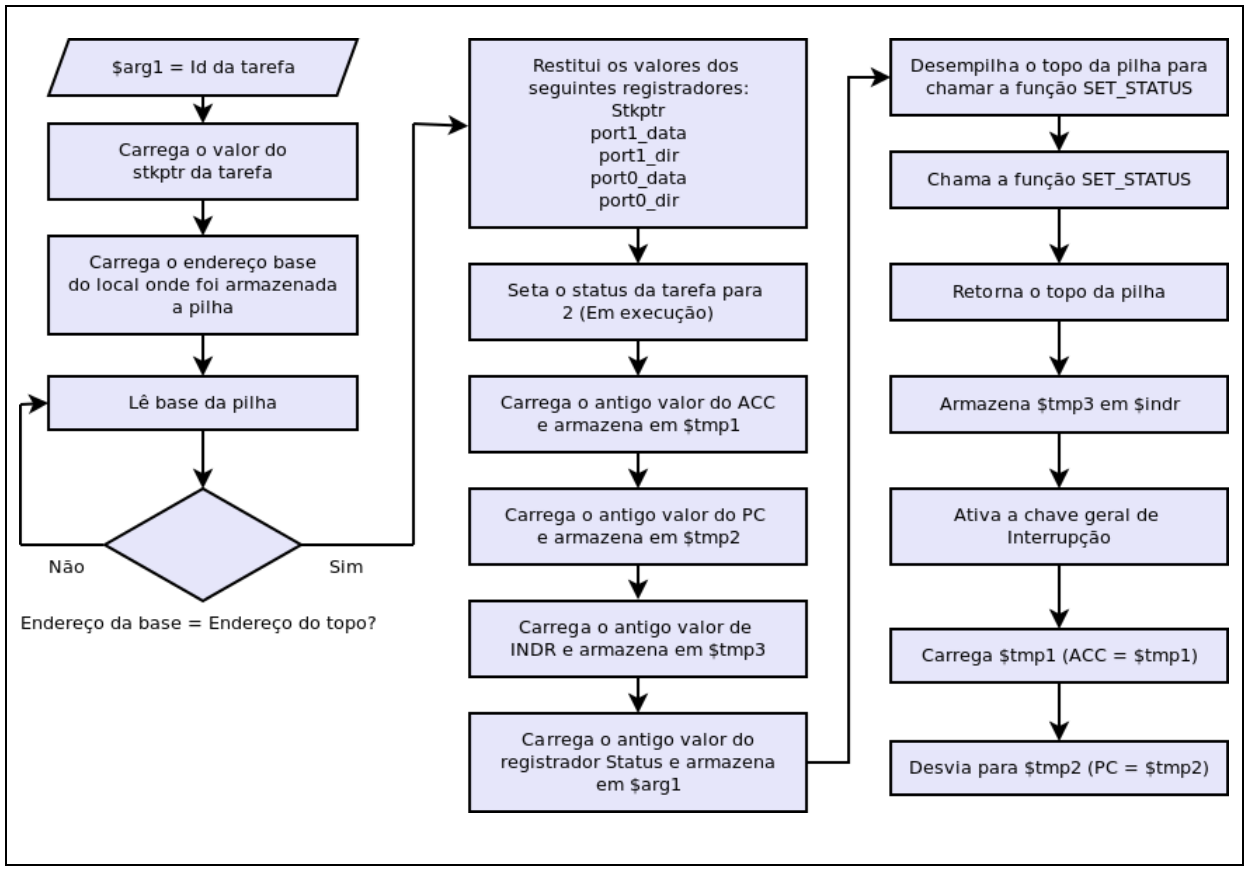

Figura 9: Fluxograma do funcionamento da função OS_TSK_RETURN

Primeiramente a função OS_TSK_RETURN carrega o identificador da tarefa e seleciona o último valor do registrador STKPTR. Esse processo é realizado para iniciar a recuperação dos valores contidos anteriormente na pilha de procedimentos. Após carregar o valor do STKPTR, é calculado o endereço base da pilha e inicia-se um laço de repetição que copia o valor da STACK_CONTEXT_TABLE para a pilha de processos.

Feita essa recuperação da pilha os valores dos registradores port1_data, port1_dir, port0_data e port0_dir são recuperados. Esses registradores especiais, depois de recuperados, não terão seus valores alterados no decorrer da função OS_TSK_RETURN.

Em seguida o estado da tarefa a ser retornada é passado para 2, que significa que a tarefa está “em execução”.

Os antigos valores dos registradores ACC, PC e INDR são carregados para variáveis temporárias.

Em seguida o topo da pilha é armazenado em uma variável temporária e o valor do registrador STATUS é carregado como argumento da função SET_STATUS. A função

SET_STATUS é responsável por alterar o valor do registrador STATUS com base em um registrador que contenha os antigos valores do registrador STATUS antes da interrupção da tarefa que está retornando. Essa função é necessária porque o registrador STATUS é alterado apenas por instruções aritméticas, não podendo ser alterado por uma instrução STO.

Em seguida a chave geral de interrupção é ativada e os valores dos registradores INDR e ACC, anteriormente armazenados em variáveis temporárias são restituídos e a função desvia para o último valor do PC da tarefa. Esse desvio é realizado por uma instrução JR, não definida no escopo do μBIP. O desvio é realizado por essa instrução porque o valor para o qual a função deve desviar é dinâmico, variando a cada execução. Uma instrução JMP desvia para um endereço fixo na memória, e uma instrução JR desvia para o endereço contido em um endereço.

Agendador de Tarefas (SCHEDULER)

O agendador de tarefas, conhecido no contexto do BIP/OS como SCHEDULER, pode ser considerado como um dos principais elementos que caracterizam o BIP/OS como um sistema operacional. Tal função define qual a próxima tarefa a ser executada, além de chamar os processos responsáveis por pausar a tarefa atual.

Para que a função SCHEDULER funcione corretamente, a função OS_TSK_CREATE precisa manter uma tabela de tarefas, a TASK_TABLE, tabela esta ordenada em ordem crescente. Essa tabela possui como conteúdo a concatenação da prioridade da tarefa e seu identificador.

Quanto menor o valor da prioridade, maior a mesma, ou seja, caso uma tarefa possua prioridade 1 e a tarefa seguinte possua prioridade 0, a tarefa mais importante é a que tem o menor valor de prioridade, neste caso, a tarefa que possui prioridade 0.

Em contrapartida, quanto menor o identificador da tarefa, mais antiga é a mesma. Ao se concatenar o valor da prioridade de uma tarefa com o seu identificador e realizar uma ordenação crescente desses valores tem-se uma lista ordenada contendo as tarefas mais importantes primeiro, ordenadas em seguida por ordem de criação.

Garantindo-se a integridade dessa tabela garante-se também que será necessário apenas um algoritmo round-robin para criar um escalonamento por prioridade.

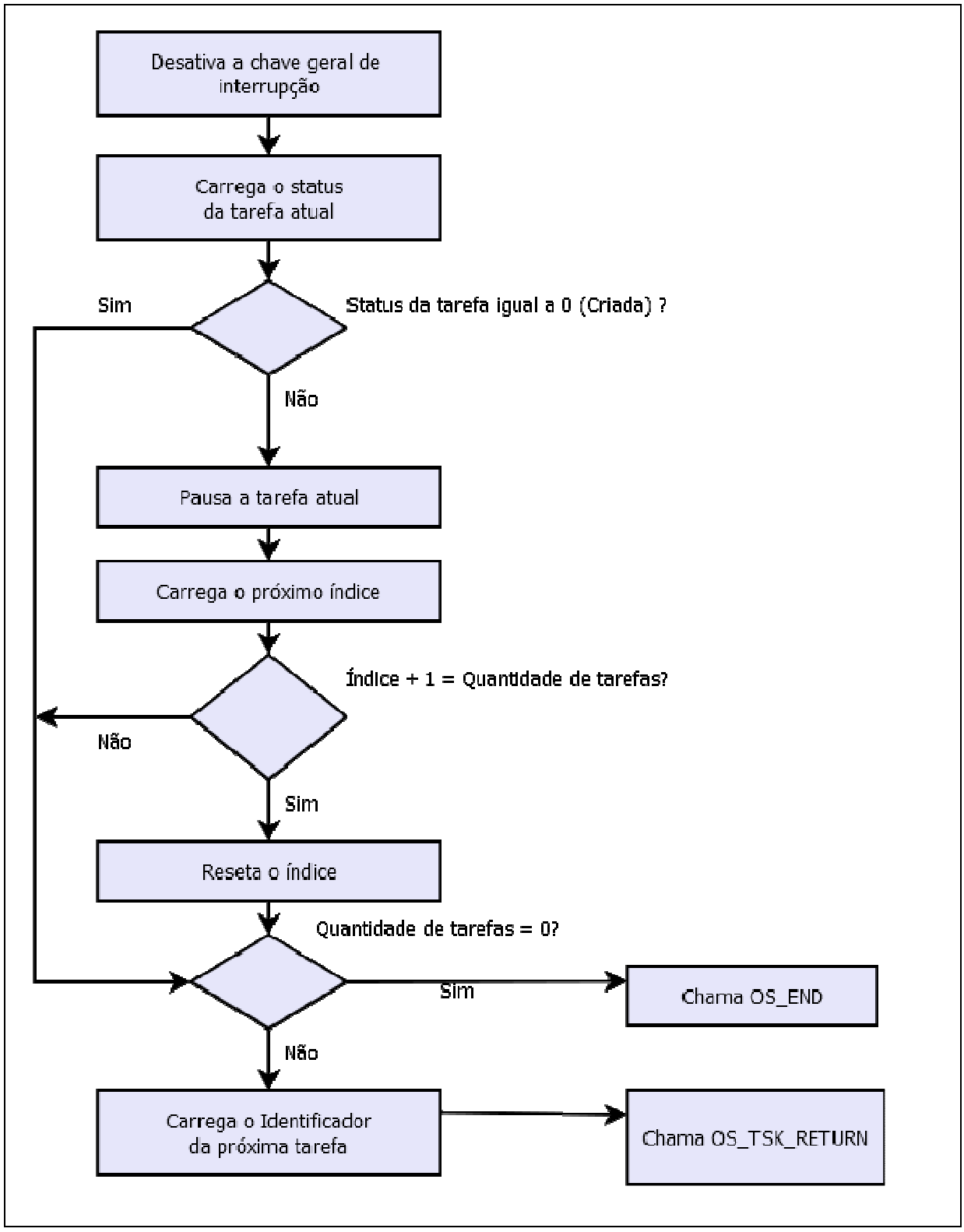

Figura 10: Agendador de tarefas do BIP/OS

De acordo com a Figura 10, a primeira preocupação do algoritmo de escalonamento do BIP/OS é desligar a chave geral de interrupções. Isso impede que o processo de troca de contexto seja interrompido antes de seu término.

Após bloquear a geração de uma interrupção o algoritmo verifica se a tarefa atualmente em execução encontra-se em estado 0, ou seja, recém criada. Se for possível afirmar que sim, significa que é a primeira vez que o sistema operacional está executando a função SCHEDULER.

Caso essa verificação resulte em uma negação, a tarefa atual é pausada e verifica-se qual a próxima tarefa a ser executada. Essa verificação é feita incrementando-se em 1 o índice atual utilizado pela TASK_TABLE. Caso o próximo número seja igual ou maior que a quantidade de tarefas que estão sendo executadas no sistema o índice é resetado.

Após esse processo, realiza-se uma verificação para ver se ainda existem tarefas a serem executadas no sistema. Caso não existam, a função SCHEDULER chama uma função chamada OS_END, que é uma função que possui uma instrução HLT. Caso contrário o identificador da próxima tarefa é carregado e armazenado como argumento para a função OS_TSK_RETURN, chamada em seguida.

Rotina de Interrupção

A rotina de interrupção é o primeiro conjunto de instruções realizadas pelo BIP/OS. A Figura 11 mostra o fluxograma de funcionamento da rotina de tratamento de interrupções.

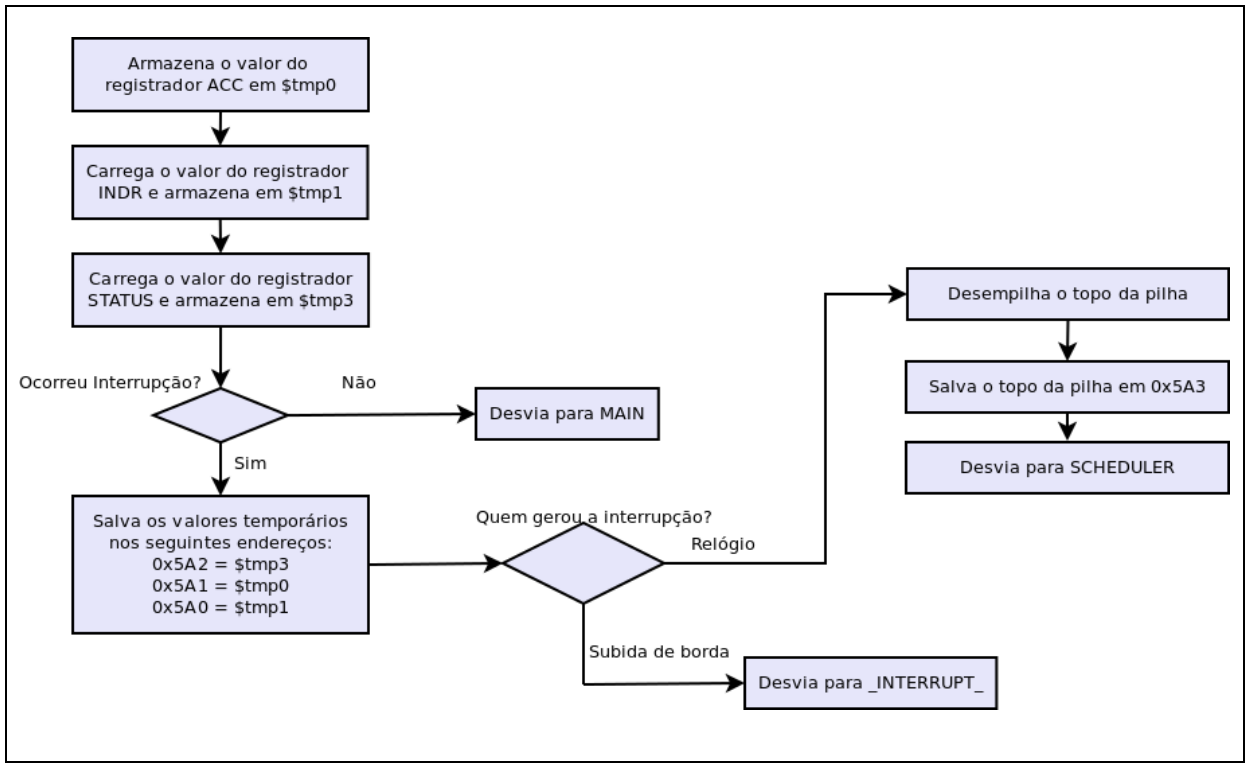

Figura 11: Rotina de interrupção

A primeira instrução a ser executada é o armazenamento do conteúdo do registrador ACC em uma variável temporária. Em seguida, os valores dos registradores INDR e STATUS também são armazenados em variáveis temporárias. Isso ocorre para que seja possível salvar o contexto de uma tarefa que foi interrompida por uma interrupção.

Salvos esses valores, a rotina verifica se ocorreu alguma interrupção. Em caso negativo o sistema desvia para a rotina MAIN, que contém a inicialização do sistema e das tarefas a serem executadas no mesmo. Essa situação ocorre quando o sistema é ligado.

Caso o evento tenha sido gerado por uma interrupção os valores temporários são salvos em endereços específicos da memória para acesso posterior.

Depois disso a rotina de interrupção verifica quem gerou a interrupção. Caso a mesma tenha sido gerada pela subida da borda 0 da porta 0 o sistema simplesmente devia para a rotina chamada _INTERRUPT_, que pode conter um programa que o usuário deseje executar em caso de interrupção. O retorno da rotina _INTERRUPT_ deve ser feito desviando para a rotina INTERRUPT_RETURN.

Caso a interrupção tenha sido gerada pelo temporizador o sistema desempilha o topo da pilha de procedimentos e salva o valor em 0x5A3. Esse valor é o valor do próximo PC que a tarefa interrompida iria executar. Salvo esse valor, a função SCHEDULER é chamada.

Rotina MAIN e funcionamento do BIP/OS

A rotina MAIN contém a inicialização das variáveis que o sistema irá utilizar e a declaração das tarefas a serem executadas pelo sistema.

Para que uma tarefa possa executar no BIP/OS é necessário, na rotina MAIN, chamar a função OS_TSK_CREATE, passando como parâmetro os endereços de início e fim da tarefa e sua prioridade. Após essa declaração a rotina MAIN chama a função SCHEDULER, que irá iniciar e organizar a execução das tarefas.

Cada tarefa deve conter, ao seu fim, uma chamada à função OS_TSK_END. Essa chamada torna o algoritmo de escalonamento um algoritmo dito cooperativo. No BIP/OS uma tarefa sempre deve avisar ao sistema que terminou sua execução. Se esse processo não for feito, pode ocorrer uma “invasão” do espaço da tarefa que está escrita imediatamente após o término da tarefa que finalizou a sua execução.

Ciclo de vida de uma tarefa no BIP/OS

O ciclo de vida de uma tarefa é exemplificado no diagrama mostrado na Figura *12*. Na mesma são mostradas 3 tarefas. As tarefas 1 e 2 possuem a prioridade 1, e a tarefa 3 possui prioridade 0. No BIP/OS, quanto menor o valor da prioridade, mais importante a tarefa. Após a criação das 3 tarefas o sistema operacional escolhe qual a primeira tarefa a executar e passa o processador para a mesma. Até que ocorra um estouro do temporizador ou uma interrupção a tarefa 3 irá executar suas operações. Entretanto, quando um desses dois eventos ocorrer, o sistema operacional irá salvar o contexto da tarefa e chamará a próxima tarefa da lista.

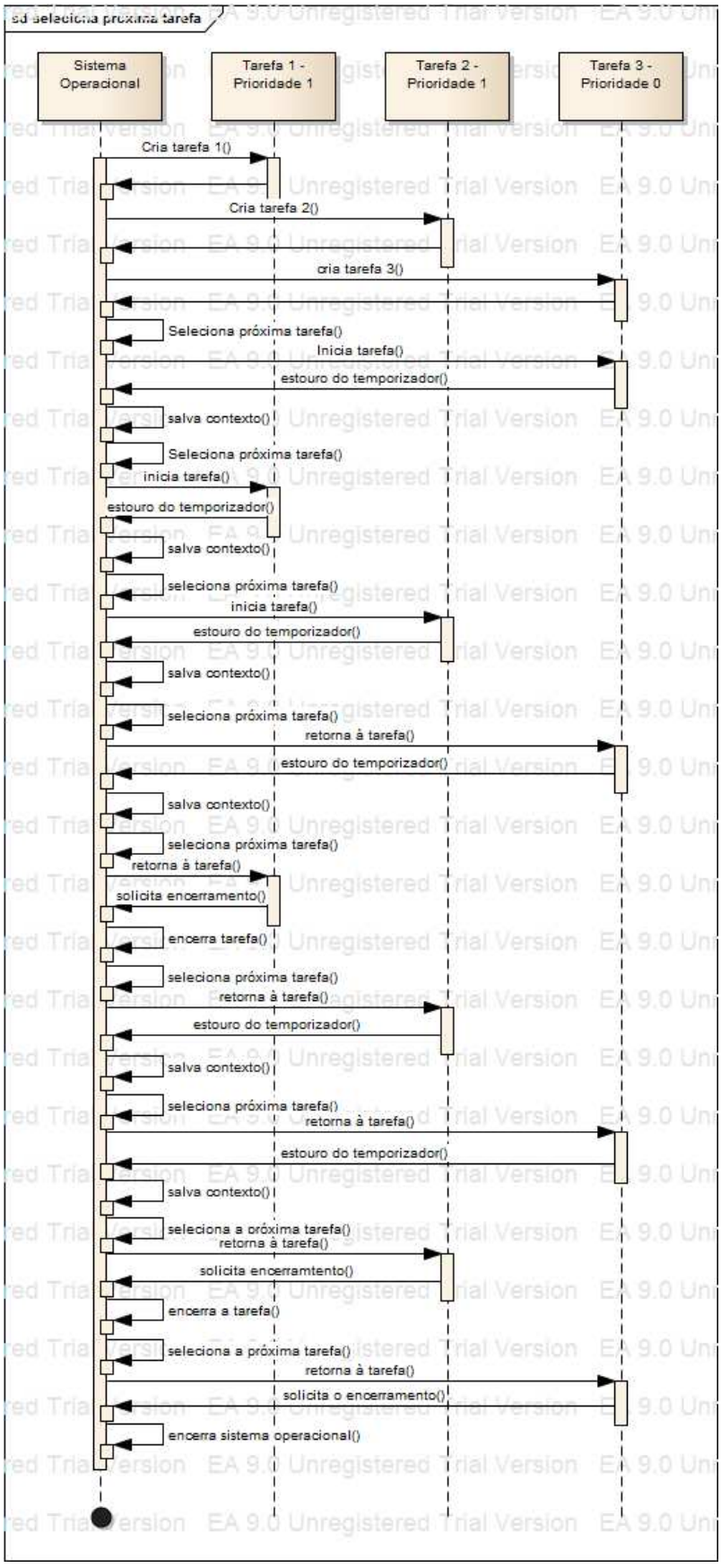

Figura 12: Ciclo de vida de uma tarefa no BIP/OS

No BIP/OS uma tarefa só é finalizada quando solicita encerramento através da função OS_END. Conforme o diagrama da Figura *12*, a primeira tarefa a terminar neste exemplo foi a tarefa 1. Quando terminou seu processamento a mesma solicitou o encerramento dela mesma e o sistema operacional removeu a tarefa da lista de tarefas ativas. O mesmo ocorre até que todas as tarefas estejam encerradas.

Quando todas as tarefas estão concluídas, o sistema operacional encerra suas atividades.

Organização da memória de dados

A memória de dados do µBIP está organizada de acordo com o Quadro *7*. O BIP/OS organiza a memória de dados de forma a usar os endereços superiores a 0x3FF para o sistema operacional, e endereços inferiores ou iguais a 0x3FF para programas do usuário.

Endereço	Descrição	Endereço	Descrição
0x000 a 0x3FF	Espaço de memória do usuário	0x6F0 a 0x6F7	Contexto da pilha da tarefa E
0x400	Port0_dir	0x6F8 a 0x6FF	Contexto da pilha da tarefa F
0x401	Port0_data	0x700 a 0x70C	Área de contexto da tarefa 0
0x402	Port1_dir	0x70D a 0x70F	Não utilizado
0x403	Port1_data	0x710 a 0x71C	Área de contexto da tarefa 1
0x404 a 0x40F	Não utilizado	0x71D a 0x71F	Não utilizado
0x410	Tmr0_config	0x720 a 0x72C	Área de contexto da tarefa 2
0x411	Tmr0_value	0x72D a 0x72F	Não utilizado
0x412	Time slice config	0x730 a 0x73C	Área de contexto da tarefa 3
0x413 a 0x41F	Não utilizado	0x73D a 0x73F	Não utilizado
0x420	Int_config	0x740 a 0x74C	Área de contexto da tarefa 4
0x421	Int_status	0x74D a 0x74F	Não utilizado
0x422 a 0x42F	Não utilizado	0x750 a 0x75C	Área de contexto da tarefa 5
0x430	Mcu_config	0x75D a 0x75F	Não utilizado
0x431	Indr	0x760 a 0x76C	Área de contexto da tarefa 6
0x432	Status	0x76D a 0x76F	Não utilizado
0x433	ToS	0x770 a 0x77C	Área de contexto da tarefa 7
0x434	Stkptr	0x77D a 0x77F	Não utilizado
0x435 a 0x5AF	Não utilizado	0x780 a 0x78C	Área de contexto da tarefa 8
0x5B0 a 0x5BF	Tabela de tarefas	0x78D a 0x78F	Não utilizado
0x5C0 a 0x67F	Não utilizado	0x790 a 0x79C	Área de contexto da tarefa 9
0x680 a 0x687	Contexto da pilha da tarefa 0	0x79D a 0x79F	Não utilizado
0x688 a 0x68F	Contexto da pilha da tarefa 1	0x7A0 a 0x7AC	Área de contexto da tarefa A
0x690 a 0x697	Contexto da pilha da tarefa 2	0x7AD a 0x7AF	Não utilizado
0x698 a 0x69F	Contexto da pilha da tarefa 3	0x7B0 a 0x7BC	Área de contexto da tarefa B
0x6A0 a 0x6A7	Contexto da pilha da tarefa 4	0x7BD a 0x7BF	Não utilizado
0x6A8 a 0x6AF	Contexto da pilha da tarefa 5	0x7C0 a 0x7CC	Área de contexto da tarefa C
0x6B0 a 0x6B7	Contexto da pilha da tarefa 6	0x7CD a 0x7CF	Não utilizado

0x6B8 a 0x6BF	Contexto da pilha da tarefa 7	0x7D0 a 0x7DC	Área de contexto da tarefa D
0x6C0 a 0x6C7	Contexto da pilha da tarefa 8	0x7DD a 0x7DF	Não utilizado
0x6C8 a 0x6CF	Contexto da pilha da tarefa 9	0x7E0 a 0x7EC	Área de contexto da tarefa E
0x6D0 a 0x6D7	Contexto da pilha da tarefa A	0x7ED a 0x7EF	Não utilizado
0x6D8 a 0x6DF	Contexto da pilha da tarefa B	0x7F0 a 0x7FC	Área de contexto da tarefa F
0x6E0 a 0x6E7	Contexto da pilha da tarefa C	0x7FD a 0x7FF	Não utilizado
0x6E8 a 0x6EF	Contexto da pilha da tarefa D		

Quadro 7: Organização da memória de dados do µBIP

Sem contar com os registradores de uso específico o BIP/OS ocupa 380 posições na memória de dados para realizar suas operações.

3.1.4 Plano de testes

Para validar o trabalho realizado foi seguido um roteiro de testes definidos na formulação do TTC 1.

Os primeiros testes a serem realizados foram testes de criação, pausa, encerramento e liberação de tarefas. O Quadro 8 mostra quais testes foram realizados e quais os resultados esperados e obtidos.

Teste	**Resultado Esperado**	**Resultado Obtido**
Criação de tarefa	Criação da estrutura de dados a ser utilizada pelo gestor de tarefas	Bem sucedido
Criação de duas tarefas em seqüência	Criação de uma estrutura de dados individual para cada tarefa criada.	Bem sucedido
Pausa de tarefa	Registrador PC salvo na estrutura de dados mantida pelo gestor de tarefas; Status da tarefa alterado	Bem sucedido
Encerramento de tarefa	Remoção da estrutura de dados da tarefa utilizada pelo gestor de tarefas	Alterado
Liberação de tarefa	Remoção da estrutura de dados da tarefa e remoção da tarefa da lista de tarefas utilizada pelo agendador de tarefas	Alterado
Retorno de Tarefa	O status da tarefa deverá ser alterado	Bem sucedido

Quadro 8: Plano de testes da API do BIP/OS

Após a realização dos testes apenas o encerramento da tarefa e a liberação da tarefa foram alterados. O BIP/OS não remove a tarefa da estrutura de dados representada pela CONTEXT_TABLE e pela STACK_CONTEXT_TABLE porque tal processo é

desnecessário. A justificativa para tal resultado é que o número de tarefas a serem executadas no sistema não será maior do que 0xF tarefas. Implementar um algoritmo que removesse as informações sobre a tarefa seria desperdício de processamento.

Com relação à liberação de uma tarefa os resultados foram satisfatórios, mas também alterados. Com a não necessidade de se remover as informações da tarefa finalizada da CONTEXT_TABLE e da STACK_CONTEXT_TABLE não se pode afirmar que o teste foi bem sucedido, visto que tal remoção estava prevista no teste. Entretanto, a tarefa é sim removida da TASK_TABLE, que é reorganizada quando uma tarefa se encerra.

Após executados estes testes, foi verificado o funcionamento da rotina de tratamento de interrupções. Tal rotina foi implementada de forma a chamar o agendador de tarefas quando ocorressem estouros do temporizador do sistema, ou executar outro processo, quando a interrupção fosse gerada pela subida da borda no pino 0 do registrador port_0. O Quadro 9 mostra quais testes foram realizados e o resultado obtido em cada teste.

Teste	**Resultado Esperado**	**Resultado Obtido**
Estouro do temporizador	Chamada do agendador de tarefas	Bem sucedido
Criação de uma tarefa; Configuração da rotina de interrupção para acessar a tarefa criada; Interrupção gerada pela subida de borda do pino 0 do registrador port_0	Desvio para a tarefa criada. Execução da mesma até o próximo estouro do temporizador.	Bem sucedido

Quadro 9: Plano de testes para a rotina de tratamento de interrupções

O agendador de tarefas utiliza os recursos agrupados no gestor de tarefas para realizar suas operações, além de utilizar-se das interrupções geradas pelo relógio do sistema para ser iniciado. Os testes indicados no Quadro 10 mostram quais as funcionalidades esperadas e quais os resultados obtidos.

Teste	**Resultado Esperado**	**Resultado Obtido**
Criação de duas tarefas com prioridades diferentes. Troca de contexto	A tarefa com maior prioridade deverá executar primeiro, dando lugar, em seguida, para a tarefa de menor prioridade	Bem sucedido
Criação de duas tarefas com prioridades iguais. Troca de contexto	A tarefa que estiver listada primeiro deverá se executada antes da segunda tarefa	Bem sucedido
Criação de três tarefas, duas delas com prioridades diferentes. Troca de contexto	A tarefa com maior prioridade deverá executar primeiro. Em seguida, a tarefa que estiver listada primeiro deve ser executada, sendo a última tarefa listada também a última tarefa executada.	Bem sucedido

Quadro 10: Plano de testes do agendador de tarefas

Estes testes foram realizados utilizando-se do modelo do µBIP II escrito em ArchC. O ArchC foi utilizado nos testes que não envolveram interrupções geradas pela subida da borda 0 da porta 0. Para os testes com interrupções, foi empregado o modelo do µBIP II escrito em VHDL e sintetizado em placa FPGA.

Para os teste realizados o *preescaler* do µBIP foi configurado para realizar uma interrupção a cada 131070 ciclos. Esta quantidade de ciclos define quanto tempo uma tarefa realizou suas atividades no sistema operacional.

4 CONCLUSÕES

Com este trabalho chegou-se à conclusão de que o modelo inicial do μBIP não era suficiente para a implementação de um sistema operacional. Entretanto, foram necessárias poucas alterações para viabilizar o projeto. Essas alterações foram evolutivas e não retiraram as características iniciais do μBIP. A arquitetura do mesmo ainda continua simples e compreensível e seu conjunto de instruções ainda continua fácil de aprender e útil para o ensino. Ocorre apenas que suas possibilidades foram ampliadas.

Com relação ao BIP/OS considera-se que o mesmo é um sistema operacional simples e fácil de compreender, independente de sua implementação em Assembly. As funções da API estão amplamente comentadas e buscou-se escrever o sistema operacional da forma mais legível possível.

O BIP/OS poderá ser utilizado em sala de aula de diversas formas. É possível criar projetos de implementação de tarefas para o mesmo que tornem possível a visualização da troca de contexto destas tarefas. Também é possível escrever novas funcionalidades e drivers para o BIP/OS.

A implementação do BIP/OS trouxe à tona vários conceitos de sistemas operacionais importantes, como a importância do chaveamento de contexto, a necessidade de bloqueios, neste caso representados pelas desativações das interrupções, e um modelo funcional de agendador de tarefas.

É possível visualizar com o BIP/OS como funciona cada um desses conceitos, exemplificando o por quê de um processo ser feito de uma maneira ou de outra. O autor acredita que a utilização do BIP/OS em sala de aula poderá auxiliar o aluno das disciplinas de sistemas operacionais a compreender melhor o que está sendo estudado.

Com relação às melhorias a serem feitas no BIP/OS e no μBIP II existem certos itens que seriam de grande auxílio. O principal deles é a implementação do isolamento da memória utilizada por uma tarefa. Esse isolamento atualmente não é possível porque não existe uma unidade de gerenciamento de memória no μBIP II. Para que a mesma fosse implementada seria necessário reorganizar a memória de dados do μBIP, isolando informações que poderiam ser acessadas apenas por uma tarefa de informações que poderiam ser acessadas pelo sistema

operacional. Possivelmente o aumento do tamanho da palavra de dados do μBIP, hoje com apenas 16 bits, e o aumento da quantidade de opcodes disponíveis, possibilitaria tal alteração.

Outro recurso que seria extremamente útil seria a criação de um compilador C para o assembly do BIP. Essa alteração tornaria possível também a criação de um sistema operacional para o BIP escrito em linguagem C, mais compreensível que o Assembly.

REFERÊNCIAS

BRTOSa. **BRTOS:** Brazilian Real Time Operating System. 2012. Disponível em: <http://code.google.com/p/brtos/>. Acesso em: 22 out. 2012.

BRTOSb. **Manual de referência do BRTOS**. Versão 1.7x. Outubro de 2012. Disponível em: <http://brtos.googlecode.com/files/Manual%20de%20refer%C3%AAncia%20-%20BRTOS%201.7x.pdf>. Acesso em: 05 out. 2012.

DEITEL, H. M.; DEITEL, P. J.; CHOFFNES, D. R. **Sistemas operacionais.** 3.ed. São Paulo: Pearson Prentice Hall, 2005

FREERTOS. BARRY, Richards. **FreeRTOS**: Market Leading RTOS(Real Time Operating System) for embedded systems supporting 33 microcontroller architectures. 2012. Disponível em: <http://www.freertos.org>. Acesso em: 27 out. 2012.

MORANDI, Diana ; PEREIRA, Maicon Carlos ; RAABE, André Luis Alice ; ZEFERINO, Cesar Albenes . **Um processador básico para o ensino de conceitos de arquitetura e organização de computadores.** Hífen, Uruguaiana, v. 30, p. 73-80, 2006.

MORANDI, Diana ; RAABE, André Luis Alice ; ZEFERINO, Cesar Albenes. Processadores para Ensino de Conceitos Básicos de Arquitetura de Computadores. In: WORKSHOP DE EDUCAÇÃO EM ARQUITETURA DE COMPUTADORES, 1., 2006, Ouro Preto. **Proceedings of the 18th International Symposium on Computer Architecture and High Performance Computing - Workshops**. Porto Alegre : SBC, 2006. p. 17-24.

OLIVEIRA, Rômulo Silva; CARISSIMI, Alexandre da Silva; TOSCANI, Simão Sirineo. **Sistemas Operacionais**. 3.ed. Porto Alegre: Instituto de Informática da UFRGS: Editora Sagra Luzzato, 2004.

OSA. **OSA Documentation.** 2008. Disponível em: <http://www.pic24.ru/doku.php/en/osa/ref/intro>. Acesso em: 15 out. 2012.

PEREIRA, Maicon Carlos. **µBip:** microcontrolador básico para o ensino de sistemas embarcados. Itajaí, 2008. 161 f. Trabalho de Conclusão de Curso (Graduação em Ciência da Computação) - Centro de Ciências Tecnológicas da Terra e do Mar, Universidade do Vale do Itajaí, Itajaí, 2008.

PEREIRA, Fabio. **Microcontroladores PIC:** técnicas avançadas. São Paulo: Érica, 2002.

PICOS18. **PICos18. 2012.** Disponível em: <http://www.picos18.com/index_us.htm>. Acesso em: 05 out. 2012.

PRAGMATEC. **PICos18:** Real Time Kernel for PIC18. Maio de 2006. Disponível em: <http://www.picos18.com/Download/PICOS18_API_us.pdf>. Acesso em: 07 out.2012.

RECH, Paulo Roberto Machado. **BIP IV:** especificação e suporte na ide bipide. Itajaí, 2011. 97 f. Trabalho de Conclusão de Curso (Graduação em Ciência da Computação) - Centro de Ciências Tecnológicas da Terra e do Mar, Universidade do Vale do Itajaí, Itajaí, 2011.

SHAW, Alan C. **Sistemas e software de tempo real.** Porto Alegre: Bookman, 2001.

STALLINGS, William. Multiprocessor and Real-Time Scheduling. In: **Operating Systems:** internals and design principles**.** 7.ed. New Jersey: Prentice Hall, 2012. p 430 – 473.

STANKOVIC, John A.; RAJKUMAR, R. **Real Time Operating Systems**. In Real Time Systems. Kluwer Academic Publishers. 2004

TANEMBAUM, Andrew S. **Sistemas operacionais modernos.** 3.ed. São Paulo: Pearson Prentice Hall, 2010.

uGNU/RTOS. **uGNU/RTOS Homepage**. 2012. Disponível em: <http://microgn urtos.sourceforge.net >. Acesso em: 25 out. 2012.

APÊNDICE A. CONJUNTO DE INSTRUÇÕES DO µBIP

Este apêndice apresenta uma descrição mais detalhada do conjunto de instrução do µBIP. As subseções o conjunto de instruções de cada classe de instrução.

CONTROLE

HLT	***Halts***
Sintaxe:	`HLT`
Descrição:	Desabilita a atualização do PC
Opcode:	00000
Status:	Nenhum bit afetado
PC:	Não é atualizado
Operação:	Nenhuma operação é realizada
Exemplo:	`HLT`
Antes da instrução:	`PC = 0x0100`
Após a instrução	`PC = 0x0100`

ARMAZENAMENTO

STO	***Store***
Sintaxe:	`STO operand`
Descrição:	Armazena o conteúdo do registrador ACC em uma posição da memória indicada por *operand*
Opcode:	00001
Status:	Nenhum bit afetado
PC:	`PC = PC + 1`
Operação:	`Memory[operand] = ACC`
Exemplo:	`STO 0x0002`
Antes da instrução:	`ACC = 0x0005` `MEM[2] = 0x0000`
Após a instrução	`ACC = 0x0005` `MEM[2] = 0x0005`

CARGA

LD	***Load***
Sintaxe:	`LD operand`
Descrição:	Carrega um valor armazenado em uma posição de memória indicado por *operand* para o registrador ACC.
Opcode:	00010
Status:	Nenhum bit afetado
PC:	`PC = PC + 1`
Operação:	`ACC = Memory[operand]`
Exemplo:	`LD 0x0002`
Antes da instrução:	`ACC = 0x0000` `MEM[2] = 0x0005`
Após a instrução	`ACC = 0x0005` `MEM[2] = 0x0005`

LDI	***Load Immediate***
Sintaxe:	`LDI operand`
Descrição:	Carrega o valor indicado por *operand* para o registrador ACC
Opcode:	00011
Status:	Nenhum bit afetado
PC:	`PC = PC + 1`
Operação:	`ACC = operand`
Exemplo:	`LDI 0x0005`
Antes da instrução:	`ACC = 0x0000`
Após a instrução	`ACC = 0x0005`

ARITMÉTICA

ADD	***Add***
Sintaxe:	`ADD operand`
Descrição:	O valor do registrador ACC é somado com o valor armazenado na posição de memória indicada por *operand* e o resultado armazenado no registrador ACC
Opcode:	00100
Status:	Z, N e C são afetados por esta operação
PC:	`PC = PC + 1`
Operação:	`ACC = ACC + Memory[operand]`
Exemplo:	`ADD 0x0002`
Antes da instrução:	`ACC = 0x0004` `MEM[2] = 0x0003`
Após a instrução	`ACC = 0x0007` `MEM[2] = 0x0003`

ADDI	***Add Immediate***
Sintaxe:	`ADDI operand`
Descrição:	O valor do registrador ACC é somado com o valor indicado por *operand* e o resultado armazenado no registrador ACC
Opcode:	00101
Status:	Z, N e C são afetados por esta operação
PC:	`PC = PC + 1`
Operação:	`ACC = ACC + operand`
Exemplo:	`ADDI 0x0002`
Antes da instrução:	`ACC = 0x0004`
Após a instrução	`ACC = 0x0006`

SUB	***Subtract***
Sintaxe:	`SUB operand`
Descrição:	O valor do armazenado na posição de memória indicada por *operand* é subtraído do registrador ACC e o resultado armazenado no registrador ACC
Opcode:	00110
Status:	Z, N e C são afetados por esta operação

PC:	`PC = PC + 1`
Operação:	`ACC = ACC – Memory[operand]`
Exemplo:	`SUB 0x0002`
Antes da instrução:	`ACC = 0x0004` `MEM[2] = 0x0003`
Após a instrução	`ACC = 0x0001` `MEM[2] = 0x0003`

SUBI ***Subtract Immediate***

Sintaxe:	`SUBI operand`
Descrição:	O valor representado por *operand* é subtraído do registrador ACC e o resultado armazenado no registrador ACC
Opcode:	00111
Status:	Z, N e C são afetados por esta operação
PC:	`PC = PC + 1`
Operação:	`ACC = ACC – operand`
Exemplo:	`SUB 0x0002`
Antes da instrução:	`ACC = 0x0004`
Após a instrução	`ACC = 0x0002`

LÓGICA BOOLEANA

NOT ***Not***

Sintaxe:	`NOT`
Descrição:	Aplica o operador de negação lógico ao valor armazenado no registrador ACC e armazena o resultado em ACC
Opcode:	01111
Status:	Z e N são afetados por esta operação
PC:	`PC = PC + 1`
Operação:	`ACC = NOT(ACC)`
Exemplo:	`NOT`
Antes da instrução:	`ACC = 0x000F`
Após a instrução	`ACC = 0xFFF0`

AND ***And***

Sintaxe:	`AND operand`
Descrição:	Aplica o operador “E” lógico entre o valor armazenado no registrador ACC e o valor armazenado na posição de memória indicada por *operand* e armazena o resultado em ACC
Opcode:	10000
Status:	Z e N são afetados por esta operação
PC:	`PC = PC + 1`
Operação:	`ACC = ACC AND Memory[operand]`
Exemplo:	`AND 0x0002`
Antes da instrução:	`ACC = 0x00C4` `MEM[2] = 0x0007`
Após a instrução	`ACC = 0x0004` `MEM[2] = 0x0007`

ANDI	***And Immediate***
Sintaxe:	`ANDI operand`
Descrição:	Aplica o operador “E” lógico entre o valor armazenado no registrador ACC e o valor indicado por *operand* e armazena o resultado em ACC
Opcode:	10001
Status:	Z e N são afetados por esta operação
PC:	`PC = PC + 1`
Operação:	`ACC = ACC AND operand`
Exemplo:	`ANDI 0x0003`
Antes da instrução:	`ACC = 0x00C5`
Após a instrução	`ACC = 0x0001`

OR	***Or***
Sintaxe:	`OR operand`
Descrição:	Aplica o operador “OU” lógico entre o valor armazenado no registrador ACC e o valor contido no endereço indicado por *operand* e armazena o resultado em ACC
Opcode:	10010
Status:	Z e N são afetados por esta operação
PC:	`PC = PC + 1`
Operação:	`ACC = ACC OR Memory[operand]`
Exemplo:	`OR 0x0002`
Antes da instrução:	`ACC = 0x0004` `MEM[2] = 0x00C0`
Após a instrução	`ACC = 0x00C4` `MEM[2] = 0x00C0`

ORI	***Or Immediate***
Sintaxe:	`ORI operand`
Descrição:	Aplica o operador “OU” lógico entre o valor armazenado no registrador ACC e o valor indicado por *operand* e armazena o resultado em ACC
Opcode:	10011
Status:	Z e N são afetados por esta operação
PC:	`PC = PC + 1`
Operação:	`ACC = ACC OR operand`
Exemplo:	`OR 0x0002`
Antes da instrução:	`ACC = 0x0004`
Após a instrução	`ACC = 0x0006`

XOR	***Xor***
Sintaxe:	`XOR operand`
Descrição:	Aplica o operador “OU” lógico exclusivo entre o valor armazenado no registrador ACC e o valor contido no endereço indicado por *operand* e armazena o resultado em ACC
Opcode:	10100
Status:	Z e N são afetados por esta operação

PC:	`PC = PC + 1`
Operação:	`ACC = ACC XOR Memory[operand]`
Exemplo:	`XOR 0x0002`
Antes da instrução:	`ACC = 0x0006` `MEM[2] = 0x0004`
Após a instrução	`ACC = 0x0002` `MEM[2] = 0x0004`

XORI — *Xor Immediate*

Sintaxe:	`XOR operand`
Descrição:	Aplica o operador "OU" lógico exclusivo entre o valor armazenado no registrador ACC e o valor indicado por *operand* e armazena o resultado em ACC
Opcode:	10101
Status:	Z e N são afetados por esta operação
PC:	`PC = PC + 1`
Operação:	`ACC = ACC XOR operand`
Exemplo:	`XORI 0x0002`
Antes da instrução:	`ACC = 0x0006`
Após a instrução	`ACC = 0x0004`

DESVIO

BEQ — *Branch Equal*

Sintaxe:	`BEQ operand`
Descrição:	Atualiza o valor do PC com o valor do campo *operand* caso o resultado da ULA seja igual a zero
Opcode:	01000
Status:	Nenhum bit é afetado

PC:	`SE (STATUS.Z = 1) ENTAO`
	`  PC = operand`
	`SENAO`
	`  PC = PC + 1`

Operação:	Nenhuma operação realizada

Exemplo:	`LDI 0x0005`
	`SUB 0x0008`
	`BEQ 0x0002`

Antes da instrução:	`PC = 0x000A MEM[8] = 0x0005` `STATUS.Z = 1 STATUS.N = 0`
Após a instrução	`PC = 0x0002`

BNE — *Branch Non-Equal*

Sintaxe:	`BNE operand`
Descrição:	Atualiza o valor do PC com o valor do campo *operand* caso o resultado da ULA não seja igual a zero
Opcode:	01001
Status:	Nenhum bit é afetado

PC:	`SE (STATUS.Z = 0) ENTAO`
	`  PC = operand`
	`SENAO`
	`  PC = PC + 1`

Operação: Nenhuma operação realizada

Exemplo:
```
LDI 0x0005
SUB 0x0008
BNE 0x0002
```

Antes da instrução:
```
PC = 0x000A MEM[8] = 0x0006
STATUS.Z = 0 STATUS.N = 1
```

Após a instrução
```
PC = 0x0002
```

BGT *Branch Greater Than*

Sintaxe:
```
BGT operand
```

Descrição: Atualiza o valor do PC com o valor do campo *operand* caso o resultado da ULA seja maior que zero

Opcode: 01010

Status: Nenhum bit é afetado

PC:
```
SE (STATUS.Z = 0) E (STATUS.N = 0) ENTAO
   PC = operand
SENAO
   PC = PC + 1
```

Operação: Nenhuma operação realizada

Exemplo:
```
LDI 0x0005
SUB 0x0008
BGT 0x0002
```

Antes da instrução:
```
PC = 0x000A MEM[8] = 0x0004
STATUS.Z = 0 STATUS.N = 0
```

Após a instrução
```
PC = 0x0002
```

BGE *Branch Greater Equal*

Sintaxe:
```
BGE operand
```

Descrição: Atualiza o valor do PC com o valor do campo *operand* caso o resultado da ULA seja igual ou maior que zero

Opcode: 01011

Status: Nenhum bit é afetado

PC:
```
SE (STATUS.N = 0) ENTAO
   PC = operand
SENAO
   PC = PC + 1
```

Operação: Nenhuma operação realizada

Exemplo:
```
LDI 0x0005
SUB 0x0008
BGE 0x0002
```

Antes da instrução:
```
PC = 0x000A MEM[8] = 0x0005
STATUS.Z = 1 STATUS.N = 0
```

Após a instrução
```
PC = 0x0002
```

BLT *Branch Less Than*

Sintaxe:
```
BLT operand
```

Descrição: Atualiza o valor do PC com o valor do campo *operand* caso o resultado da ULA seja menor que zero

Opcode: 01100

Status: Nenhum bit é afetado

PC:
```
SE (STATUS.N = 1) ENTAO
   PC = operand
```

```
SENAO
   PC = PC + 1
```

Operação: Nenhuma operação realizada

Exemplo:
```
LDI 0x0005
SUB 0x0008
BLT 0x0002
```

Antes da instrução: `PC = 0x000A MEM[8] = 0x0006 STATUS.Z = 0 STATUS.N = 1`

Após a instrução `PC = 0x0002`

BLE — *Branch Less Equal*

Sintaxe: `BLE operand`

Descrição: Atualiza o valor do PC com o valor do campo *operand* caso o resultado da ULA seja igual ou menor que zero

Opcode: 01101

Status: Nenhum bit é afetado

PC:
```
SE (STATUS.Z = 1) OU (STATUS.N = 1) ENTAO
   PC = operand
SENAO
   PC = PC + 1
```

Operação: Nenhuma operação realizada

Exemplo:
```
LDI 0x0005
SUB 0x0008
BLE 0x0002
```

Antes da instrução: `PC = 0x000A MEM[8] = 0x0005 STATUS.Z = 1 STATUS.N = 0`

Após a instrução `PC = 0x0002`

JMP — *Jump*

Sintaxe: `JMP operand`

Descrição: Atualiza o valor do PC com o valor do campo *operand*, realizando um desvio incondicional

Opcode: 01110

Status: Nenhum bit é afetado

PC: `PC = operand`

Operação: Nenhuma operação realizada

Exemplo: `JMP 0x0002`

Antes da instrução: `PC = 0x000A`

Após a instrução `PC = 0x0002`

JR — *Jump Register*

Sintaxe: `JR operand`

Descrição: Atualiza o valor do PC com o valor do contido na posição de memória representada pelo campo *operand*, realizando um desvio incondicional

Opcode: 11111

Status: Nenhum bit é afetado

PC: `PC = Memória[operand]`

Operação:	Nenhuma operação realizada
Exemplo:	`JR 0x0002`
Antes da instrução:	`PC = 0x000A MEM[0x0002] = 0x00B5`
Após a instrução	`PC = 0x00B5`

DESLOCAMENTO LÓGICO

SLL — *Shift Left Logical*

Sintaxe:	`SLL operand`
Descrição:	Realiza o deslocamento lógico à esquerda no número de bits indicado por *operand* sobre o registrador ACC, armazenando o resultado da operação no registrador ACC
Opcode:	10110
Status:	Z e N são afetados por esta operação
PC:	`PC = PC + 1`
Operação:	`ACC = ACC << operand`
Exemplo:	`SLL 0x0002`
Antes da instrução:	`ACC = 0x0001`
Após a instrução	`ACC = 0x0004`

SRL — *Shift Right Logical*

Sintaxe:	`SRL operand`
Descrição:	Realiza o deslocamento lógico à direita no número de bits indicado por *operand* sobre o registrador ACC, armazenando o resultado da operação no registrador ACC
Opcode:	10111
Status:	Z e N são afetados por esta operação
PC:	`PC = PC + 1`
Operação:	`ACC = ACC >> operand`
Exemplo:	`SLL 0x0002`
Antes da instrução:	`ACC = 0x0004`
Após a instrução	`ACC = 0x0001`

MANIPULAÇÃO DE VETOR

LDV — *Load Vector*

Sintaxe:	`LDV operand`
Descrição:	Carrega o valor armazenado no endereço dado por (*operand* + INDR) para o registrador ACC
Opcode:	11000
Status:	Nenhum bit é afetado
PC:	`PC = PC + 1`
Operação:	`ACC = Memory[operand + INDR]`

Exemplo:	`LDI 0x0001`
	`STO $indr`
	`LDV $0`

Antes da instrução:	`ACC = 0x0000 MEM[0] = 0x0004` `MEM[1] = 0x0007`
Após a instrução	`ACC = 0x0007 MEM[0] = 0x0004` `MEM[1] = 0x0007`

STOV	***Store Vector***
Sintaxe:	`STOV operand`
Descrição:	Armazena o valor do registrador ACC no endereço indicado por (*operand* + INDR)
Opcode:	11001
Status:	Nenhum bit é afetado
PC:	`PC = PC + 1`
Operação:	`Memory[operand + INDR] = ACC`

Exemplo:	`LDI 0x0001`
	`STO $indr`
	`LDI 0x0003`
	`STOV $0`

Antes da instrução:	`ACC = 0x0000 MEM[0] = 0x0004` `MEM[1] = 0x0007`
Após a instrução	`ACC = 0x0007 MEM[0] = 0x0004` `MEM[1] = 0x0003`

SUPORTE A PROCEDIMENTOS

CALL	***Call***
Sintaxe:	`CALL operand`
Descrição:	Realiza uma chamada de procedimento para o endereço indicado em *operand*. Antes da chamada o endereço de retorno (PC + 1) é armazenado no topo da pilha (ToS)
Opcode:	11100
Status:	Nenhum bit é afetado
PC:	`ToS = PC + 1` `PC = operand`
Operação:	Nenhuma operação realizada
Exemplo:	`CALL 0x00F0`
Antes da instrução:	`PC = 0x00D3 ToS = 0xXXXX`
Após a instrução	`PC = 0x00F0 ToS = 0x00D4`

RETURN	***Return***
Sintaxe:	`RETURN`
Descrição:	Retorna da sub-rotina, recuperando o valor do PC armazenado no tipo da pilha (ToS). O procedimento chamado deve utilizar o ACC para retornar um valor ao procedimento chamador
Opcode:	11010
Status:	Nenhum bit é afetado
PC:	`PC = ToS`
Operação:	Nenhuma operação realizada
Exemplo:	`RETURN`
Antes da instrução:	`PC = 0x00F0 ToS = 0x00D4` `ACC = Valor de retorno`
Após a instrução	`PC = 0x00D4 ToS = 0xXXXX` `ACC = Valor de retorno`

RETINT	***Return Interrupt***
Sintaxe:	`RETINT`
Descrição:	Retorna de uma interrupção para o endereço do programa antes da interrupção e habilita a interrupção geral
Opcode:	11011
Status:	Nenhum bit é afetado
PC:	`PC = ToS`
Operação:	Nenhuma operação realizada
Exemplo:	`RETINT`
Antes da instrução:	`PC = 0x00F0 ToS = 0x00D4`
Após a instrução	`PC = 0x00D4 ToS = 0xXXXX`

MANIPULAÇÃO DE PILHA

PUSH	***Push***
Sintaxe:	`PUSH`
Descrição:	Armazena o conteúdo do registrado ACC no topo da pilha de procedimentos
Opcode:	11101
Status:	Nenhum bit é afetado
PC:	`PC = PC + 1`
Operação:	`ToS = ACC`
Exemplo:	`PUSH`
Antes da instrução:	`ToS = 0xXXXX ACC=0x3B1`
Após a instrução	`ToS = 0x3B1 ACC=0x3B1`

POP	***Pop***
Sintaxe:	`POP`
Descrição:	Remove o conteúdo do topo da pilha de procedimentos (ToS) e armazena o conteúdo no registrador ACC
Opcode:	11110
Status:	Nenhum bit é afetado
PC:	`PC = PC + 1`
Operação:	ACC = ToS ToS = ToS – 1
Exemplo:	`POP`
Antes da instrução:	`ToS = 0x3B1 ToS -1 = 0x000 ACC=0x000`
Após a instrução	`ToS = 0x000 ACC=0x3B1`

APÊNDICE B. CÓDIGO-FONTE DO BIP/OS

```
#
# BIP/OS
# Author: Hendrig Wernner M. S. Gonçalves
#

.data
  # Variáveis
  prx_tsk_id     : .word 0x000      # Proximo id de tarefa
  tsk_quantity   : .word 0x000      # Quantidade de tarefas
  next_tsk       : .word 0x000      # Próxima tarefa
  get_next_tsk   : .word 0x000
  next_tsk_ind   : .word 0x000      # Índice da próxima tarefa
  current_tsk_ind : .word 0x000     # Tarefa atual
  lst_acc_value  : .word 0x000      # Endereço para armazenar o último valor do acumulador
  lst_indr_value  : .word 0x000     # Endereço para armazenar o último valor do Indr
  lst_status_value: .word 0x000     # Endereço para armazenar o último valor do Status
  lst_pc_value   : .word 0x000      # Endereço para armazenar o último valor do PC
.text
#=============================================================================
# Trecho de interrupção
# Este é o ponto para onde são desviadas as interrupções do sistema.
#=============================================================================

  # Carrega os valores dos registradores para salvamento de contexto
  JMP MAIN             # $tmp0 é um endereço específico
  STO $tmp0
  LD $indr              #
  STO $tmp1             # $tmp1 é um endereço específico
  LD $status            #
  STO $tmp2             # $tmp2 é um endereço específico

  LD $int_status         # Carrega o registrador $int_status
  ANDI 0x0003           # Caso o valor seja 0, não ocorreu interrupção, então...
  BEQ MAIN              # Pula para MAIN
  LD lst_acc_value       # Carrega o endereço onde será armazenado o último
  STO $indr             # valor do registrador ACC antes da interrupção
  LD $tmp0              #
  STOV 0x0000           # O registrador $zero é o endereço 0x000

  LD lst_indr_value      # Carrega o endereço onde será armazenado o último
  STO $indr             # valor do registrador Indr antes da interrupção
  LD $tmp1              #
  STOV 0x0000           #

  LD lst_status_value     # Carrega o endereço onde será armazenado o último
  STO $indr             # valor do registrador Status antes da interrupção
  LD $tmp0              #
  STOV 0x0000           #

  POP                  # Se foi iniciada pelo relógio, desempilha o topo da pilha
  STO $tmp0             # E salva o valor do mesmo no endereço especificado
  LD lst_pc_value        # Carrega o endereço onde será armazenado o último
  STO $indr             # valor do registrador PC antes da interrupção
  LD $tmp0              #
  STOV 0x0000           #

  LD $int_status        # Verificar se a interrupção foi gerada por relógio ou externamente
  ANDI 0x002            #
  BNE _INTERRUPT_        # Se foi gerada externamente, vai para o trecho de interrupção

  JMP SCHEDULER          # Vai para o SCHEDULER

INTERRUPT_RETURN:        # Rotina de retorno de interrupção
  LD current_tsk_ind     # Carrega o índice da tarefa atual
  ANDI 0x0007           #
  STO $arg1             # Carrega o Id da tarefa atual
  JMP OS_TSK_RETURN      # Retorna a tarefa

#=============================================================================
# Fim do trecho de interrupção
#=============================================================================

#=============================================================================
#   Utils
# Funções úteis para o sistema
#=============================================================================

#=============================================================================
# SET_STATUS
```

```
80 # Atualiza o estado do registrador STATUS
81 # Argumentos:
82 # $arg0 = Valor do registrador status
83 #================================================================================
84 SET_STATUS:
85   LD $arg0            # Carrega o argumento 0, que contém o Status
86   SUBI 0x6            #
87   BNE CN_NOT_SET       # Os flags C e N estão setados?
88   LDI 0xFFE           # Força o set nos flags C e N
89   SUBI 0xFFF           #
90   RETURN              # Retorna
91 CN_NOT_SET:           #
92   LD $arg0            #
93   SUBI 0x5            #
94   BNE CZ_NOT_SET       # Os flags C e Z estão setados?
95   LDI 0x1             #
96   ADDI 0xFFF           # Força o set nos flags C e Z
97   RETURN              # Retorna
98
99 Z_NOT_SET:            #
100   LD $arg0            #
101   SUBI 0x4            #
102   BNE C_NOT_SET        # O flag C está setado ?
103   LDI 0x2             #
104   ADDI 0xFFF           # Força um set no flag C
105   RETURN              # Retorna
106 C_NOT_SET:           #
107   LD $arg0            #
108   SUBI 0x2            #
109   BNE N_NOT_SET        # O Flag N está setado?
110   LDI 0xFFA           #
111   SUBI 0x2            # Força um set no Flag N
112   RETURN              # Retorna
113 N_NOT_SET:           #
114   LD $arg0            #
115   SUBI 0x1            #
116   BNE Z_NOT_SET        # O Flag Z está setado?
117   LDI 0x1             #
118   ANDI 0x0            # Força um set no flag Z
119   RETURN              # Retorna
120 Z_NOT_SET:           #
121   LDI 0x3             #
122   SUBI 0x2            # Limpa os flags
123   RETURN              # Retorna
124
125 #================================================================================
126 # BUBBLE_SORT
127 # Algoritmo bubble sort para ordenar a lista de tarefas
128 #================================================================================
129 BUBBLE_SORT:
130
131   LD tsk_quantity      # Carrega o tamanho do vetor
132   SUBI 0x0000          #
133   BNE NOT_EMPTY_LIST    # Verifica se existem itens na lista de tarefas
134   RETURN              #
135 NOT_EMPTY_LIST:        #
136   LD tsk_quantity      #
137   SUBI 0x1            #
138   STO $tmp0           # $tmp0 = tsk_quantity - 1 (tmp0 = k)
139   LDI 0x0             #
140   STO $tmp1           # $tmp1 = 0 (tmp1 = i)
141 FOR_CMD:              #
142   LD $tmp0            #
143   SUB $tmp1           #
144   BEQ END_FOR_CMD      # Comando for ($tmp2 = 1; $tmp2<=tsk_quantity; $tmp2++)
145   LDI 0x0000          #
146   STO $tmp2           # $tmp2 = 0 (tmp2 = j)
147 WHILE_BS:            #
148   LD $tmp2            #
149   SUB $tmp0           #
150   BGE END_WHILE_BS      # Desvia se $tmp2 > $tmp0
151   LD $tmp2            #
152   STO $indr           #
153   LDV 0x05B0          # TAB_PROCESSOS[$tmp2]
154   STO $tmp3           # $tmp3 = TAB_PROCESSOS[$tmp2] (v[j])
155   LD $indr            #
156   ADDI 0x001          #
157   STO $indr           #
158   LDV 0x05B0          # TAB_PROCESSOS[$tmp2 + 1]
159   STO $tmp4           # $tmp4 = TAB_PROCESSOS[$tmp2 + 1]
160   LD $tmp3            #
161   SUB $tmp4           #
162   BLE END_IF_BS        # Desvia se $tmp3 <= $tmp4
163   LD $tmp3            #
164   STO $tmp5           # $tmp5 = $tmp3
```

```
165   LD $tmp2             #
166   ADDI 0x001           #
167   STO $indr            #
168   LDV 0x05B0           #
169  STO $tmp3            # $tmp3 = TAB_PROCESSOS[$tmp2 + 1]
170   LD $tmp2             #
171
172  STO $indr            #
173   LD $tmp3             #
174   STOV 0x5B0           # TAB_PROCESSOS[$tmp2] = $tmp3
175   LD $indr             #
176   ADDI 0x01            #
177   STO $indr            #
178   LD $tmp5             #
179   STOV 0x05B0          # TAB_PROCESSOS[$tmp2 + 1] = $tmp5
180 END_IF_BS:            #
181   LDI 0x1             #
182   ADD $tmp2            #
183   STO $tmp2            # $tmp2 ++
184   JMP WHILE_BS         #
185 END_WHILE_BS:         #
186   LD $tmp1            #
187   ADDI 0x01            #
188   STO $tmp1            #
189   JMP FOR_CMD          #
190 END_FOR_CMD:          #
191   RETURN              #
192
193 #================================================================================
194 #   Funções de Escrita
195 #================================================================================
196 OS_WRITE_PORT0:
197   LDI 0xFFE             # Gera um bloqueio para escrita
198   AND $int_config        #
199   STO $int_config        #
200
201   LD $arg0              # Escreve o argumento 0
202   STO $port0_data        # Na porta 0, dedicada para escrita
203
204   LDI 0x001             #
205   OR $int_config         # Remove o bloqueio
206   STO $int_config        #
207
208   RETURN                #
209
210 OS_WRITE_PORT1:
211   LDI 0xFFE             # Gera um bloqueio para escrita
212   AND $int_config        #
213   STO $int_config        #
214
215   LD $arg0              # Escreve o argumento 0
216   STO $port1_data        # Na porta 1, dedicada para escrita
217
218   LDI 0x001             #
219   OR $int_config         # Remove o bloqueio
220   STO $int_config        #
221
222   RETURN                #
223
224 #================================================================================
225 # API
226 #================================================================================
227
228 # O BIP/OS possui as seguintes funções em sua API
229 #
230 # = Criação de tarefa      (OS_TSK_CREATE)
231 # = Inicialização de tarefa (OS_TSK_START)
232 # = Pausa de tarefa        (OS_TSK_PAUSE)
233 # = Retorno de tarefa      (OS_TSK_RETURN)
234 # = Encerramento de tarefa  (OS_TSK_END)
235 # = Remoção de tarefa      (OS_TSK_REMOVE)
236
237 #================================================================================
238 # OS_TSK_CREATE
239 #
240 # Cria a tarefa na estrutura de dados mantida pelo sistema operacional
241 # Argumentos:
242 # $arg1 = Endereço inicial da tarefa
243 # $arg2 = Endereço final da tarefa
244 # $arg3 = Prioridade da tarefa
245 3================================================================================
246 OS_TSK_CREATE:
247
248   LD prx_tsk_id          # Carrega o próximo endereço
249   SLL 0x4               # Desloca o valor logicamente para a esquerda 4 vezes
```

```
250   ADDI 0x700             # Soma o resultado com o endereço 0x700, dando o endereço onde
ficarão os
251   STO $indr              # argumentos da tarefa. Salva esse resultado em $indr
252
253   LD $arg1               # Carrega o primeiro argumento da tarefa, início da tarefa
254   STOV 0x0000             # Armazena o valor em 0x7X0, onde X é o id da tarefa
255
256   LD $indr               # Carrega o valor da variável indr
257   ADDI 1                 # Atualiza tmp0 para conter o próximo endereço de tarefa
258   STO $indr               #
259   LD $arg2               # Carrega o segundo argumento da tarefa, fim da tarefa
260   STOV 0x0000             # Armazena o valor em 0x7X1, onde X é o id da tarefa
261
262   LD $indr               #
263   ADDI 1                 # Atualiza tmp0 para conter o próximo endereço de tarefa
264   STO $indr               #
265   LD $arg3               # Carrega o terceiro argumento (prioridade da tarefa)
266   STOV 0x0000                 # Salva na estrutura a prioridade da tarefa no endereço 0x7X2
267
268   LD $indr               #
269   ADDI 0x0001             # Atualiza $indr para conter o próximo endereço de tarefa
270   STO $indr               #
271   LD $arg1               # Carrega o primeiro argumento (início da tarefa)
272   STOV 0x0000             # Salva no endereço 0x7X3, que contém o pc da tarefa
273
274   LD prx_tsk_id           #
275   SLL 0x0004             #
276   ADDI 0x070C             # Define $tmp0 como 0x7XC, último endereço na tabela
277   STO $tmp0              # de contexto da tarefa
278
279 LOOP_FILL_ZERO:           # Preenche com zero os demais endereços
280   LD $indr               # Carrega o registrador $indr
281   ADDI 0x0001             # Adiciona mais 1
282   STO $indr              # $indr = $indr + 1
283   LDI 0x0000             #
284   STOV 0x0000             # MEM[0x7X$indr] = 0x0
285   LD $indr               #
286   SUB $tmp0              #
287   BNE LOOP_FILL_ZERO       #
288
289   LD tsk_quantity          #
290  STO $indr               # Define o valor como índice do vetor
291   LD $arg3               # Carrega a prioridade da tarefa
292   SLL 0x0004             # Desloca 4 bits para a esquerda
293   ADD prx_tsk_id          # Adiciona com o Id da tarefa
294   STO $tmp0              # $tmp0 = 0x0PI, onde P é a prioridade e I é o Id
295   LD $tmp0               # Carrega o valor da prioridade
296   STOV 0x5B0             # Salva o valor na tabela de processos
297   LD tsk_quantity          # Carrega a quantidade de tarefas
298   ADDI 0x1               # Soma mais 1
299   STO tsk_quantity         #
300
301   LD prx_tsk_id           # Carrega o próximo Id
302   ADDI 0x0001             # Soma mais 1
303   STO prx_tsk_id          # Salva o valor
304
305   CALL BUBBLE_SORT         # Ordena a tabela de processos do menor para o maior
306
307   RETURN                 #
308
309 #================================================================================
310 # OS_TSK_START (UNUSED)
311 #
312 # Inicia a tarefa
313 # Essa função não é chamada por uma instrução CALL, mas sim por uma instrução
314 # JMP. Ou seja, ela não guarda valores na pilha
315 #
316 # Argumentos:
317 # $arg1 = Id da tarefa a ser iniciada
318 #================================================================================
319 OS_TSK_START:
320
321   LD $arg1               #
322   SLL 0x4                #
323   ADDI 0x702              #
324   STO $indr              # Endereço da prioridade da tarefa (0x7X2)
325   LDV 0x0000             # Carrega a prioridade da tarefa
326
327   SLL 0x4                # Desloca 4 bits à esquerda
328   ADD $arg1              # Soma com o Id da tarefa
329   STO $tmp0              #
330
331   LD $indr               # Carrega o valor do registrador $indr
332   STO $tmp1              # Armazena temporariamente
333
```

```
334   LD $tmp1              # Recarrega o valor antigo de $indr
335   STO $indr             #
336
337   ADDI 0x6              # Carrega 0x6
338   STO $indr             # Armazena em $indr o endereço do status da tarefa (0x7X6)
339   LDI 2                 # Carrega o valor 2 (Status 2 = Em execução)
340   STOV 0x0000           # Salva na estrutura ( MEM[0+0x7X6] = 2)
341
342   LD $indr              # Carrega o endereço do status da tarefa (0x7X6)
343   SUBI 0x3              # Subtrai para o endereço do PC da tarefa (0x7X3)
344   STO $indr             # Armazena o valor no registrador $indr
345   LDV 0x0000            # Carrega MEM[0+0x7X3]
346   STO $tmp1
347
348   JR $tmp1              # Desvia para o pc inical da tarefa
349
350 #================================================================================
351 # OS_TSK_PAUSE
352 #
353 # Pausa a tarefa em execução, salvando o contexto da mesma
354 #
355 # Argumentos:
356 # $arg1 = Id da tarefa
357 #================================================================================
358 OS_TSK_PAUSE:
359
360   LD lst_pc_value        # Carrega o último pc em andamento
361   STO $indr             # Carrega o endereço de armazenamento do valor do
362   LDV 0x0000            # pc da tarefa
363   STO $tmp0             # $tmp0 = Último PC da tarefa
364   LD $arg1              #
365   SLL 0x4               #
366   ADDI 0x703            #
367   STO $indr             # Endereço base dos argumentos da tarefa
368   STO $tmp1             #
369   LD $tmp0              #
370   STOV 0x0000           # Salva o PC no endereço 0x7X3, sendo X o Id da tarefa
371
372   LD lst_status_value     # Carrega o último valor do registrador status
373   STO $indr             #
374   LDV 0x0000            #
375   STO $tmp0             # $tmp0 = last status value
376   LD $tmp1              #
377   ADDI 0x0001           #
378   STO $tmp1             #
379   STO $indr             # Carrega o endereço de armazenamento do valor do registrador
status
380   LD $tmp0              #
381   STOV 0x0000           # Armazena o valor do acumulador no endereço 0x7X4
382
383   LD $indr              #
384   ADDI 0x1              #
385   STO $tmp1             #
386   LD lst_acc_value       # Carrega o último valor do acumulador da tarefa
387   STO $indr             # Carrega o endereço de armazenamento do valor do acumulador
388   LDV 0x0000            #
389   STO $tmp0             #
390   LD $tmp1              #
391
392   STO $indr             #
393   LD $tmp0              #
394   STOV 0x0000           # Armazena o valor do acumulador no endereço 0x7X5
395
396   LD $indr              #
397   ADDI 0x1              #
398   STO $indr             # Carrega o endereço de armazenamento do valor do status da tarefa
399   STO $tmp1             #
400   LDI 0x1               # Carrega o status da tarefa (1, em espera)
401   STOV 0x0000           # Armazena o status da tarefa no endereço 0x7X6
402
403   LD lst_indr_value       # Carrega o índice do vetor
404   STO $indr             #
405   LDV 0x0000            #
406   STO $tmp0             #
407   LD $tmp1              #
408   ADDI 0x0001           #
409   STO $tmp1             #
410   STO $indr             # Carrega o endereço de armazenamento do valor do índice do vetor
411   LD $tmp0              #
412   STOV 0x0000           # Armazena o valor do índice do vetor no endereço 0x7X7
413
414   LD $indr              #
415   ADDI 0x1              #
416   STO $indr             # Carrega o endereço de armazenamento do valor da direção
417   LD $port0_dir          # do registrador port0_data
```

```
418   STOV 0x0000            # Armazena o valor em 0x7X8
419
420   LD $indr               #
421   ADDI 0x1               #
422   STO $indr              # Carrega o endereço de armazenamento do valor
423   LD $port0_data          # contido no registrador port0_data
424   STOV 0x0000            # Armazena o valor em 0x7X9
425
426   LD $indr               #
427   ADDI 0x1               #
428   STO $indr              # Carrega o endereço de armazenamento do valor da direção
429   LD $port1_dir           # do registrador port1_data
430   STOV 0x0000            # Armazena o valor em 0x7XA
431
432   LD $indr               #
433   ADDI 0x1               #
434   STO $indr              # Carrega o endereço de armazenamento do valor
435   LD $port1_data          # contido no registrador port1_data
436   STOV 0x0000            # Armazena o valor em 0x7XB
437
438   POP                   # Retira o endereço da chamada a esta função
439   STO $tmp2              #
440
441   LD $indr               #
442   ADDI 0x1               #
443   STO $indr              # Carrega o endereço do registrador $stkptr
444   LD $stkptr             #
445   STOV 0x0000            # Armazena o valor em 0x7XC
446
447   LD $arg1               # Carrega o id da tarefa
448   SLL 0x3               # Desloca o endereço para a esquerda
449   ADDI 0x680             # Adiciona ao endereço base da STACK_CONTEXT_TABLE
450   STO $tmp1              # $tmp1 = base da pilha
451
452   LD $stkptr             # Ajusta o valor do registrador STKPTR
453   SUBI 0x0001            # para que o mesmo utilize o mesmo índice que a
454   STO $tmp0              # tabela de contexto da pilha
455
456   LD $arg1               #
457   SLL 0x0003             #
458   ADDI 0x0680            #
459   ADD $tmp0              # Carrega o endereço para o topo da pilha
460   STO $tmp0              #
461   STO $indr              #
462
463 STACK_CONTEXT_LOOP_POP:         #
464   LD $indr                    #
465
466  SUB $tmp1                   # Desvia se o índice for menor que o valor do
467   BLT END_STACK_CONTEXT_LOOP_POP  # registrador STKPTR da tarefa
468   POP                        # Loop de salvamento da pilha
469   STOV 0x0000                 # Salva o topo da pilha no endereço 0x6Xi
470   LD $indr                   # Carrega o endereço 0x6Xi
471   SUBI 0x1                   # Carrega o próximo endereço (0x6Xi-1)
472   STO $indr                  # Salva o mesmo
473   JMP STACK_CONTEXT_LOOP_POP     # Volta para o loop e continua desempilhando
474 END_STACK_CONTEXT_LOOP_POP:      #
475
476   LD $tmp2                # Carrega o endereço da função que chamou
477   PUSH                   # Retorna para a pilha
478
479   RETURN                  #
480
481 #================================================================================
482 # OS_TSK_RETURN
483 #
484 # Retorna a tarefa em pausa, carregando o contexto da mesma
485 #
486 # Argumentos:
487 # $arg1 = Id da tarefa
488 #================================================================================
489 OS_TSK_RETURN:
490
491   LD $arg1               #
492   SLL 0x4                #
493   ADDI 0x70C              # Carrega o endereço onde foi armazenado o valor do
494   STO $indr              # registrador $stkptr
495   LDV 0x0000              #
496   STO $tmp0              # Salva o mesmo em $tmp0
497
498   LD $arg1               #
499  SLL 0x3                #
500   ADDI 0x680              #
501   STO $tmp1              # Carrega o endereço da base da pilha (0x6X0)
502
```

```
503   LD $tmp0                    #
504   SUBI 0x0000                  # Verifica se existiam itens na pilha
505   BEQ END_STACK_CONTEXT_LOOP_PUSH  #
506
507   LD $tmp0                    # Carrega o valor do Stack Pointer, que
508   SUBI 0x0001                  # começa em 1, e
509   STO $tmp0                   # Ajusta para o índice, que começa em 0
510
511   LD $tmp1                    #
512   ADD $tmp0                   #
513   STO $tmp0                   # Carrega o endereço do topo da pilha
514
515 STACK_CONTEXT_LOOP_PUSH:         #
516   LD $tmp0                    # Carrega o endereço topo
517   SUB $tmp1                   # Carrega o endereço base
518   BLT END_STACK_CONTEXT_LOOP_PUSH  #
519   LD $tmp0                    #
520   STO $indr                   #
521   LDV 0x0000                   # Carrega o valor contido no endereço 0x6Xi
522   PUSH                        # Salva no topo da pilha
523   LD $indr                     # Carrega o valor da variável tmp1 endereço 0x6Xi
524   SUBI 0x1                     # Diminui 1 do valor do endereço
525   STO $tmp0                    #
526   JMP STACK_CONTEXT_LOOP_PUSH     #
527 END_STACK_CONTEXT_LOOP_PUSH:      #
528
529   LD $arg1              #
530   SLL 0x4               #
531   ADDI 0x70B             # Transforma o mesmo em 0x7XB (Endereço do registrador port1_data
do contexto da tarefa)
532   STO $indr             #
533   LDV 0x0000             # Carrega o valor do registrador port1_data do contexto da tarefa
534   STO $port1_data         # Joga o valor no local
535
536   LD $indr              #
537   SUBI 0x1              # Diminui em 1 o endereço (0x7XA = Endereço do registrador
port1_dir)
538   STO $indr             #
539
540   LDV 0x0000             #
541   STO $port1_dir          # Joga o valor no local
542
543   LD $indr              #
544   SUBI 0x1              # Diminui em 1 o endereço (0x7X9 = Endereço do registrador
port0_data)
545   STO $indr             #
546   LDV 0x0000             # Carrega o valor contido no endereço
547   STO $port0_data         # Joga o valor no local
548
550   LD $indr              #
551   SUBI 0x1              # Diminui em 1 o endereço (0x7X8 = Endereço do registrador
port0_dir)
552   STO $indr             #
553   LDV 0x0000             # Carrega o valor contido no endereço
554   STO $port0_dir          # Joga o valor no local
555
556   LD $arg1              # Carrega o argumento da tarefa
557   SLL 0x0004             #
558   ADDI 0x0706            # Carrega o endereço (0x7X6 = Endereço do status da tarefa)
559   STO $indr             #
560   LDI 0x2               #
561   STOV 0x000             # Salva o status da tarefa como sendo 2 (2 - Em execução)
562
563   LD $indr              #
564   SUBI 0x0001            # Diminui em 1 o endereço (0x7X5 = Endereço do registrador acc)
565   STO $indr             #
566   LDV 0x0000             # Carrega o valor contido no endereço
567   STO $tmp1              # Salva numa variável temporária
568
569   LD $indr              #
570   SUBI 0x2              # Diminui em 2 o endereço (0x7X3 = Endereço do registrador pc)
571   STO $indr             #
572   LDV 0x0000             #
573   STO $tmp2              # Salva numa variável temporária
574
575   LD $indr              #
576   ADDI 0x4              # Aumenta o endereço em 4 (0x7X7 = Endereço do registrador indr)
577   STO $indr             #
578   LDV 0x0000             # Carrega o valor contido no endereço
579   STO $tmp3              # Joga o valor no local
580
581   LD $arg1              #
582   SLL 0x4               # Diminui em 1 o endereço (0x7X4 = Endereço do registrador status)
583   ADDI 0x0704            #
584   STO $indr             #
```

```
585   LDV 0x0000             # Carrega o valor contido no endereço
586   STO $arg0              # Salva o valor do registrador STATUS em $arg0
587
588   POP                   # Desempilha para não prejudicar o contexto da tarefa
589   STO $tmp0              # quando chamar a função SET_STATUS
590
591   LDI 0x001              #
592   OR $int_config          # Prepara para a remoção do bloqueio
593   STO $tmp4              #
594
595   CALL SET_STATUS         # Retorna o estado do registrador status
596
597   LD $tmp0               #
598   PUSH                  # Retorna o topo da pilha
599
600   LD $tmp3               # Retorna o valor do índice
601   STO $indr              #
602
603   LD $tmp4               # Carrega o valor antigo da configuração de
604   STO $int_config         # interrupção sem alterar o registrador status
605
606   LD $tmp1               # Carrega o valor do acumulador
607   JR $tmp2               # Pula para a última linha da tarefa
608
609 #================================================================================
610 # OS_TSK_END
611 #
612 # Encerra uma tarefa
613 #
614 # Argumentos:
615 # $arg0 = Id da tarefa
616 #================================================================================
617 OS_TSK_END:
618   LDI 0xFFE              # Gera um bloqueio
619   AND $int_config         #
620   STO $int_config         #
621
622   LD current_tsk_ind      # Carrega o argumento da tarefa (id da tarefa)
623   STO $indr              #
624   LDV 0x05B0             #
625   ANDI 0x0007            #
626   SLL 0x4                # Desloca em 4 posições o id da tarefa
627   ADDI 0x706             # Adiciona 0x706, formando o endereço do status da tarefa (0x7X6)
628   STO $indr
629
630   LDI 0x3                # Carrega o status 3 = Tarefa encerrada
631   STOV 0x0000            # Salva o status na estrutura
632
633   JMP OS_TSK_REMOVE       #
634
635 # ===============================================================================
636 # OS_TSK_REMOVE
637 #
638 # Remove uma tarefa. Em TODAS as tarefas a última linha deve ser um
639 # JMP para este endereço.
340 #
641 # ===============================================================================
642 OS_TSK_REMOVE:
643
644   LD current_tsk_ind      # Carrega o índice da tarefa
645   STO $tmp1              # Armazena em $tmp1
646
647   LD tsk_quantity             # Verifica se o índice é igual a 0
648   SUBI 0x0001                #
649   SUB $tmp1                  # Se o mesmo for, não precisa remover da lista
450   BEQ END_REMOVE_TASK_FROM_LIST  #
651
652
653   LDI 0x0                # Carrega o valor 0
654   STO $tmp2              # Armazena em $tmp2
655 LOOP_REBUILD_TASK_LIST:     #
656   LD $tmp2               # Carrega o valor do índice
657   SUB $tmp1              # Verifica se o valor contido no índice
658   BEQ REMOVE_TASK_FROM_LIST  # é igual ao valo de $tmp1
659   LD $tmp2               # Se não for igual, carrega $tmp2
660   ADDI 0x1               # e faz $tmp2 = $tmp2 + 1
661   STO $tmp2              #
662   JMP LOOP_REBUILD_TASK_LIST # E volta para o loop
663 REMOVE_TASK_FROM_LIST:      # Se for igual,
664   LD $tmp2               # Carrega $tmp2
665   STO $tmp3              # Armazena o mesmo em $tmp3
666   LD $tmp2               # Carrega novamente $tmp2
667   ADDI 0x1               # faz $tmp2 = $tmp2 + 1
668   STO $tmp2              #
669   STO $indr              # $indr = $tmp2
```

```
670   LDV 0x5B0               # Carrega o valor no vetor
671   STO $tmp4               # Armazena em $tmp4
672   LD $tmp3                # Carrega $tmp3 (Antigo $tmp2)
673   STO $indr               #
674   LDV 0x05B0              #
675   STO $tmp5               #
676   LD $tmp4                #
677   STOV 0x5B0              # TSK_LIST[ $tmp3 ] = $tmp4
678   LD $tmp2                #
679   STO $indr               #
680   LD $tmp5                #
681   STOV 0x05B0             #
682   LD tsk_quantity            #
683   SUBI 0x0001               #
684   SUB $tmp2                 # Verifica se a quantidade de tarefas contidas na lista
685  BLE END_REMOVE_TASK_FROM_LIST # Encerra a rotina se for igual
686   JMP REMOVE_TASK_FROM_LIST    # Senão, volta para o loop
687
688 END_REMOVE_TASK_FROM_LIST:     #
689
690   LD tsk_quantity         #
691   SUBI 0x0001            # Diminui o número de tarefas no sistema
692   STO tsk_quantity        #
693
694   CALL BUBBLE_SORT        # Ordena a lista de tarefas
695
696   LDI 0x0001             #
697   STO get_next_tsk        # Indica que a próxima tarefa pode ser chamada
698
699   LD tsk_quantity         #
700   SUBI 0x0001            #
701   SUB current_tsk_ind      # Se o índice atual for igual à quantidade de tarefas
702   BNE RESET_TSK_IND       # reseta o índice
703
704   LDI 0x0000             #
705   STO current_tsk_ind      #
706
707 RESET_TSK_IND:            #
708   LD tsk_quantity         # Verifica se ainda existem tarefas para executar
709   SUBI 0x0000            #
710   BEQ OS_END             # Se não existem mais, pula para o fim
711
712   JMP SCHEDULER           # Senão, pula para o SCHEDULER
713
714
715 #=============================================================================
716 # SCHEDULER
717 #
718 # Este scheduler é um simples round=robin, visto que a lista de
719 # tarefas está sempre organizada.
720 #=============================================================================
721 SCHEDULER:
722   LDI 0xFFE              # Gera um bloqueio
723   AND $int_config         #
724   STO $int_config         #
725
726   LD current_tsk_ind         #
727   STO $indr                 #
728   LDV 0x05B0                # Carrega o índice da tarefa atual
729   ANDI 0x0007               #
730   SLL 0x0004                #
731   ADDI 0x0706               #
732   STO $indr                 # Carrega o status da tarefa
733   LDV 0x0000                #
734   STO $tmp0                 # Armazena em $tmp0
735   LD $tmp0                  #
736   SUBI 0x000                # Status da tarefa igual a "criada"
737   BEQ GOTO_TASK             #
738
739   LD get_next_tsk           #
740   SUBI 0x0001               # Status da tarefa igual a "encerrada"
741   BEQ GOTO_TASK             #
742
743   LD current_tsk_ind         #
744   STO $indr                 #
745   LDV 0x5B0                 # Carrega o valor dentro da tabela de tarefas
746   ANDI 0xF                  # Extrai o Id da tarefa a ser pausada
747   STO $arg1                 #
748   CALL OS_TSK_PAUSE          # Pausa a tarefa
749
750   LD current_tsk_ind         # Carrega o índice no qual está a tarefa corrente
751   ADDI 0x0001               # Adiciona mais 1
752   SUB tsk_quantity          #
753   BEQ RESET_ROUND_ROBIN      # Se o indice da próxima tarefa for igual à quantidade de
tarefas
```

```
754                          # reseta o round=robin
755
756   LD current_tsk_ind         # Carrega o índice da tarefa atual
757   ADDI 0x1                  #
758   STO current_tsk_ind        # Atualiza para o próximo
759
760   JMP GOTO_TASK             # Senão, pula para a tarefa
761
762 RESET_ROUND_ROBIN:           #
763   LDI 0x0000                #
764   STO current_tsk_ind        # current task index = 0
765
766 GOTO_TASK:
767   LD tsk_quantity            # Carrega a quantidade de tarefas ativas
768   SUBI 0x0000               # Se a mesma for igual a zero
769   BEQ OS_END                # Encerra
770
771   LDI 0x0000                # Desliga flag para pegar a próxima tarefa
772   STO get_next_tsk           #
773
774   LD current_tsk_ind         # Carrega o índice da tarefa
775   STO $indr                 #
776   LDV 0x5B0                 #
777   ANDI 0x000F               # Carrega o id da tarefa
778   STO $arg1                 #
779
780   JMP OS_TSK_RETURN          # Retorna para a tarefa
781
782 #================================================================================
783 # OS_END
784 # Encerra as operações do SO
785 #================================================================================
786 OS_END:                     # Encerra o SO
787   HLT                      #
788
789 #================================================================================
790 # Rotina de Interrupção
791 #================================================================================
792 _INTERRUPT_:
793   LD current_tsk_ind       # Carrega o índice atual da tarefa
794   ANDI 0x0007             #
795   STO $arg1               # Carrega o identificador da tarefa
796   CALL OS_TSK_PAUSE        # Pausa a tarefa atual
797
798   #
799   # Interruption instructions here...
800   #
801
802   JMP INTERRUPT_RETURN     # Retorna para o ponto de interrupção
803 #================================================================================
804 # MAIN
805 #================================================================================
806 MAIN:
807   LDI 0x05A3              # Endereço onde será armazenado o valor do último PC
808   STO lst_pc_value         #
809   LDI 0x05A2              # Endereço onde será armazenado o valor do último STATUS
810   STO lst_status_value     #
811   LDI 0x05A1              # Endereço onde será armazenado o valor do último ACC
812   STO lst_acc_value        #
813   LDI 0x05A0              # Endereço onde será armazenado o valor do último INDR
814   STO lst_indr_value       #
815   LDI 0x0001              #
816   STO $tmr0_config         # Configura o prescaller
817   LDI 0x01FF
818   STO 0x0412
819   LDI 0x0000              #
820   STO $int_config          # Desativa interrupções
821
822 #================================================================================
823 # Chamada aos Programas do Usuário
824 #================================================================================
825  LDI INIT_TSK_1           # Endereço do início da tarefa
826   STO $arg1               # registrador de argumento 1
827
828  LDI END_TSK_1            # Endereço do fim da tarefa
829   STO $arg2               # registrador de argumento 2
830   LDI 0x0001              # Prioridade da tarefa 1
831   STO $arg3               #
832   CALL OS_TSK_CREATE       # Chama o criador de tarefas
833
834   LDI INIT_TSK_2           #
835   STO $arg1               #
836   LDI END_TSK_2            #
837   STO $arg2               #
838   LDI 0x0001              #
```

```
839   STO $arg3              #
840   CALL OS_TSK_CREATE  #
841
842   LDI INIT_TSK_3        #
843   STO $arg1            #
844   LDI END_TSK_3        #
845   STO $arg2            #
846   LDI 0x0000            #
847   STO $arg3            #
848   CALL OS_TSK_CREATE  #
849
850   LDI 0x0007            #
851   STO $int_config         #
852   JMP SCHEDULER       #
853
854 #=============================================================================
855 #  Programas do usuário
856 # À partir deste ponto é que começa a mágica...
857 #=============================================================================
858
859 #=============================================================================
860 # Programa 1
861 #=============================================================================
862 F1_0:            # Função que subtrai 2 do valor contido em 0x111
863   LD 0x111       #
864   SUBI 0x0002     #
865   STO 0x0111      #
866   CALL F1_1       # E chama a função F1_1
867   RETURN        #
868
869 F1_1:            # Função que adiciona 1 ao valor de 0x111
870   LD 0x0111      #
871   ADDI 0x0001     #
872   STO 0x0111      #
873   CALL F1_2       # E chama a função F1_2
874   RETURN        #
875
876 F1_2:            # Função que subtrai 2 do valor contido em 0x111
877   LD 0x0111      #
878   SUBI 0x0002     #
879   STO 0x0111      # E retorna
880   RETURN        #
881
882 # ============================================================================
883 INIT_TSK_1:       # Início da tarefa 1
884   LDI 0x01FF      # Carrega 0x1FF
885   STO 0x0111      # Armazena em 0x111
886 L1:              #
887   LD 0x111        # Carrega 0x111
888   SUBI 0x1        # Subtrai 1 do valor contido em 0x111
889   STO 0x111       # Armazena o resultado em 111
890   CALL F1_0       # Chama a função F1_0
891   LD 0x111        # Carrega o valor contido em 0x111
892   SUBI 0x0        # Verifica se o valor é igual a 0x0
893   BNE L1          # Desvia se for diferente
894   JMP OS_TSK_END # Encerra a tarefa
895 END_TSK_1:       #
896
897 #=============================================================================
898 # Programa 2
899 #=============================================================================
900 INIT_TSK_2:       # Início da tarefa 2
901  LDI 0x02FF      # Carrega o valor 0x2FF
902   STO 0x222      # Armazena em 0x222
903 L2:              #
904   LD 0x222        # Carrega o valor contido em 0x222
905   SUBI 0x1        # Subtrai 1 do valor contido em 0x222
906   STO 0x222       # Armazena o resultado em 0x222
907   LD 0x222        # Carrega o valor contido em 0x222
908   SUBI 0x0        # Verifica se o mesmo está igual a 0
909   BNE L2          # Desvia se estiver diferente
910   JMP OS_TSK_END # Encerra a tarefa
911 END_TSK_2:       #
912
913 #=============================================================================
914 # Programa 3
915 #=============================================================================
916 INIT_TSK_3:         # Início da tarefa 3
917   LDI 0x00FF        # Carrega o valor 0xFF
918   STO 0x333         # Armazena o valor em 0x333
919 L3:                #
920   LD 0x333          # Carrega o valor contido em 0x333
921   SUBI 0x1          # Subtrai 1 do valor contido em 0x333
922   STO 0x333        # Armazena o resultado em 0x333
923   LD 0x333          # Carrega o valor contido em 0x333
```

```
924   SUBI 0x0          # Verifica se o mesmo é igual a 0x0
925   BNE L3            # Desvia se for diferente
926   JMP OS_TSK_END   # Encerra a tarefa
927 END_TSK_3:          #
```