

**INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA DE
SANTA CATARINA - CÂMPUS FLORIANÓPOLIS
DEPARTAMENTO ACADÊMICO DE ELETRÔNICA
CURSO SUPERIOR DE ENGENHARIA ELETRÔNICA**

LUCAS ODAIR SILVA

**CONTROLADOR DE INTERRUPÇÕES PARA UM
SOFTCORE RISC-V**

Florianópolis, 2021

**INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA DE
SANTA CATARINA - CÂMPUS FLORIANÓPOLIS
DEPARTAMENTO ACADÊMICO DE ELETRÔNICA
CURSO SUPERIOR DE ENGENHARIA ELETRÔNICA**

LUCAS ODAIR SILVA

CONTROLADOR DE INTERRUPÇÕES PARA UM SOFTCORE RISC-V

Trabalho de conclusão de curso submetido
ao Instituto Federal de Educação, Ciência
e Tecnologia de Santa Catarina como parte
dos requisitos para obtenção do título de
engenheiro eletrônico

Orientador:
Prof. Dr. Eng Renan Augusto Starke

Florianópolis, 2021

Ficha de identificação da obra elaborada pelo autor.

Silva, Lucas Odair
CONTROLADOR DE INTERRUPÇÕES PARA UM SOFTCORE RISC-V
/ Lucas Odair Silva ; orientação de Renan Augusto Starke.
- Florianópolis, SC, 2021.
82 p.

Trabalho de Conclusão de Curso (TCC) - Instituto Federal
de Santa Catarina, Câmpus Florianópolis. Bacharelado
em Engenharia Eletrônica. Departamento Acadêmico
de Eletrônica.
Inclui Referências.

1. RISC-V. 2. Soft cores. 3. Interrupção. 4. Linguagens
de descrição de hardware. I. Starke, Renan Augusto.
II. Instituto Federal de Santa Catarina. Departamento
Acadêmico de Eletrônica. III. Título.

CONTROLADOR DE INTERRUPÇÕES PARA UM SOFTCORE RISC-V

LUCAS ODAIR SILVA

Este Trabalho foi julgado adequado para obtenção do Título de Engenheiro Eletrônico em Abril de 2021 e aprovado na sua forma final pela banca examinadora do Curso de Engenharia Eletrônica do Instituto Federal de Educação Ciência, e Tecnologia de Santa Catarina.

Florianópolis, 16 de Abril, 2021.

Banca Examinadora:

RENAN AUGUSTO

STARKE:00609149989

Assinado de forma digital por RENAN
AUGUSTO STARKE:00609149989
Dados: 2021.04.20 10:54:56 -03'00'

Renan Augusto Starke, Dr. Eng

Andreu Carminati

Andreu Carminati, Dr. Eng

Charles Borges de Lima

Charles Borges de Lima, Dr. Eng

RESUMO

Este trabalho apresenta uma implementação do mecanismo de interrupção para um *soft core* RISC-V, identificando e elaborando os elementos de *hardware* e *software* cruciais para o gerenciamento e controle do estado do núcleo. Quanto aos componentes inerentes ao *hardware*, foram implementados: os registradores de controle e status, bem como suas instruções de escrita e leitura; os mecanismos de troca de contexto; o gerenciamento de interrupções nos periféricos. Considerou-se os conceitos e as simulações do funcionamento de cada circuito projetado. Em relação ao *software*, são apresentadas as estruturas utilizadas para gerenciamento e controle das interrupções, tal como o código comum para interrupções e o vetor de ponteiros para os manipuladores de interrupções.

Palavras-chave: RISC-V. Soft cores. Interrupção. Linguagens de descrição de hardware.

ABSTRACT

This document presents an implementation of the interrupt mechanism for a RISC-V soft core, such as identification and implementation of hardware and software elements for the management and control of the core state. The following hardware components have been implemented: status and control registers, as well as their writing and reading instructions; context switching mechanisms; and peripheral interruptions management. Important concepts and simulations of the operation of each projected circuit were also considered. About the software layer, the structures used to manage and control interruptions are introduced, such as the common code for interruptions and the vector pointers for interrupt handlers.

Keywords: RISC-V. Soft cores. Interrupts. Hardware description language.

LISTA DE ILUSTRAÇÕES

Figura 1 – Transferência de controle via interrupções	15
Figura 2 – Diagrama de implementação das principais unidades funcionais do RISC-V e suas interconexões	17
Figura 3 – Diagrama das principais unidades funcionais do RISC-V, incluindo multiplexadores e linhas de controle necessárias	19
Figura 4 – Diagrama das principais unidades funcionais do RISC-V, incluindo o controlador de interrupções	20
Figura 5 – <i>Machine-mode status register</i>	21
Figura 6 – <i>Machine interrupt-pending register</i>	22
Figura 7 – <i>Machine interrupt-enable register</i>	22
Figura 8 – <i>Machine cause register</i>	23
Figura 9 – Machine trap-vector base-address register	23
Figura 10 – Machine exception program counter register	23
Figura 11 – Machine trap vector table base-address register	23
Figura 12 – Machine trap vector table 2 base-address register	24
Figura 13 – Control and Status Registers Instructions	24
Figura 14 – Trap-return instructions	25
Figura 15 – Componentes de software para gerenciar interrupções.	27
Figura 16 – Diagrama do fluxo de programa com interrupção.	28
Figura 17 – Diagrama da interface dos registradores CSRs	31
Figura 18 – Representação do controlador dos registradores CSRs.	32
Figura 19 – Diagrama de estados das instruções CSRs	34
Figura 20 – Diagrama de execução da instrução CSRRW	35
Figura 21 – Diagrama de execução da instrução CSRRS	36
Figura 22 – Diagrama de execução da instrução CSRRC	36
Figura 23 – Diagrama de execução da instrução CSRRWI	37
Figura 24 – Diagrama de execução da instrução MRET	37
Figura 25 – Representação da unidade PC.	38
Figura 26 – Diagrama de execução de uma interrupção no componente PC. . .	39
Figura 27 – Diagrama de execução do gerenciamento do registrador PC em uma interrupção.	39
Figura 28 – Endereços base dos periféricos.	40
Figura 29 – Enable General Input Interrupt register	41
Figura 30 – General Input Interrupt Edge register	41
Figura 31 – Diagrama de execução do periférico <i>GPIO</i>	41
Figura 32 – Disposição dos endereços dos registradores do periférico <i>GPIO</i> . . .	42
Figura 33 – Enable Timer Interrupt register	42

Figura 34 – Diagrama de execução do periférico <i>Timer</i>	43
Figura 35 – Disposição dos endereços dos registradores do periférico <i>Timer</i> . . .	43
Figura 36 – Funcionalidade dos registradores <i>mtvec</i> e <i>mcause</i>	44
Figura 37 – Diagrama das atribuições do <i>software</i> relacionadas a interrupções. .	46
Figura 38 – Código em linguagem C das principais funções do programa de teste.	47
Figura 39 – Diagrama de inicialização dos <i>CSRs mstatus</i> , <i>mtvec</i> e <i>mtvt</i>	48
Figura 40 – Diagrama de inicialização do <i>CSR mvt2</i> e do programa principal. .	48
Figura 41 – Diagrama de inicialização do <i>Timer</i>	49
Figura 42 – Diagrama de inicialização do <i>GPIO</i>	49
Figura 43 – Diagrama da habilitação das interrupções globais, de <i>Timer</i> e externas	50
Figura 44 – Diagrama da troca de contexto com interrupção de <i>GPIO</i>	51
Figura 45 – Diagrama do cálculo do endereço do manipulador da interrupção de <i>GPIO1</i>	52
Figura 46 – Diagrama de retorno do código comum para interrupções.	52
Figura 47 – Diagrama da troca de contexto com interrupção de <i>Timer</i>	53
Figura 48 – Diagrama do cálculo do endereço do manipulador da interrupção de <i>TC0A</i>	54
Figura 49 – Diagrama de retorno do código comum para interrupção de <i>Timer</i> .	55

LISTA DE TABELAS

Tabela 1 – Especificações das instruções de acesso aos registradores <i>CSRs</i> .	34
Tabela 2 – Especificações da instrução MRET	37
Tabela 3 – Resumo da síntese do <i>hardware</i>	56

LISTA DE ABREVIATURAS E SIGLAS

CLIC	<i>Core-Local Interrupt Controller</i> - Controlador de interrupção local de núcleo
CSR	<i>Control and Status Register</i> - Registradores de status de controle
FPGA	<i>Field-Programmable Gate Arrays</i> - Matrizes de portas de campo programáveis
GPIO	<i>General Purpose Input/Output</i> - Entrada / Saída de Uso Geral
ISA	<i>Instruction Set Architecture</i> - Conjunto de Instruções
MEIP	<i>Machine External Interrupt-Pending</i> - Interrupção Externa da Máquina Pendente
MIE	<i>Machine Interrupt-Enable</i> - Habilitação de Interrupção de Máquina
MPIE	<i>Machine Pending Interrupt-Enable</i> - Habilitação de Interrupção de Máquina Pendente
MTIP	<i>Machine Timer Interrupt-Pending</i> - Interrupção de Temporizador de Máquina Pendente
PLD	<i>Programmable Logic Device</i> - Dispositivo lógico programável
RISC	<i>Reduced Instruction Set Computer</i> - Computador com um conjunto reduzido de instruções
VHDL	<i>Very High Speed Integrated Circuits Hardware Description Language</i> - Linguagem de descrição de hardware de circuitos integrados de altíssima velocidade

SUMÁRIO

1	INTRODUÇÃO	12
1.1	Justificativa	13
1.2	Descrição do problema	13
1.3	Objetivo geral	13
1.4	Objetivos específicos	13
2	FUNDAMENTAÇÃO TEÓRICA	14
2.1	Interrupções	14
2.2	RISC-V	16
2.2.1	Arquitetura RISC-V	16
2.2.2	Registradores de controle e status	21
2.2.3	Instruções especiais das interrupções	24
2.3	Dispositivos lógicos programáveis	25
2.4	<i>Elementos de software em interrupções</i>	26
3	METODOLOGIA	29
4	PROJETO E RESULTADOS	31
4.1	Inserção e controle dos registradores CSRs	31
4.2	Implementação das Instruções CSR	34
4.3	Implementação da Instrução MRET	37
4.4	Mecanismo de troca de contexto	38
4.5	Suporte a interrupções nos periféricos	40
4.5.1	Configurações das interrupções no GPIO	40
4.5.2	Configurações das interrupções de Timer	42
4.6	<i>Elementos de software</i>	44
4.7	Integração	47
4.7.1	Interrupção de GPIO	50
4.7.2	Interrupção de Timer	52
4.8	Resultados da síntese	56
5	CONSIDERAÇÕES FINAIS	57
	REFERÊNCIAS	58

APÊNDICES	59
APÊNDICE A – CÓDIGO VHDL DO CONTROLADOR	60
APÊNDICE B – CÓDIGO <i>ASSEMBLY</i> GERADO	66

1 INTRODUÇÃO

Nos últimos 30 anos, a indústria da tecnologia adotou a inovação em um ritmo nunca antes visto. Nesse período houve uma série de novos computadores, cujo o surgimento parecia revolucionar a indústria da computação, revolução essa, que era interrompida apenas pelo fato de outra pessoa ter construído um computador ainda melhor (PATTERSON, 2017).

Uma série de arquiteturas de conjuntos de instruções (ISAs - Instruction Set Architecture) foram introduzidas no mercado nas décadas passadas, e grande parte vem sendo descontinuada ao decorrer do tempo, como a AMD Am29000, a Digital Alpha, a Digital VAX, o Hewlett Packard PA-RISC, Intel i860, Intel i960, Motorola 88000, e a Zilog Z8000 (PATTERSON; WATERMAN, 2017).

ISAs RISC (*Reduced Instruction Set Computer*) como ARM e RISC-V optam pela modularidade do conjunto de instruções, mantendo-o menos complexo e garantindo desempenho, tornando possível que essas ISAs continuem modernas mesmo ao passar de décadas. Em questão de custo, o RISC-V não adota a cobrança de taxas para uso, sendo uma ISA aberta, gerida por uma fundação sem fins lucrativos. Por se tratar de uma ISA aberta, o RISC-V conta com seus manuais de implementação disponíveis para uso livre, esse fator, aliado a ser uma ISA moderna a torna uma ótima escolha para implementação em ambiente acadêmico.

Os processadores utilizam de vários mecanismos para obter melhor desempenho. Um mecanismo que é muito utilizado e praticamente indispensável, é o tratamento de interrupções, uma vez que provê aumento da eficiência do sistema, principalmente quando é feito o acesso à periféricos mais lentos. As interrupções também são utilizadas para prover temporizadores, permitindo eventos temporizados e/ou escalonamento de tarefas quando aplicado em sistemas operacionais. Diante disso, este projeto foi delimitado ao estudo e implementação de um tratador de interrupções para um núcleo RISC-V.

O projeto que implementa a base do núcleo RISC-V utilizado para a implementação do tratador de interrupções foi criado por Starke (2021), com o propósito de ser uma plataforma para estudo no contexto da unidade curricular de Dispositivos Lógicos-Programáveis, do Curso Superior de Engenharia Eletrônica do Instituto Federal de Educação, Ciência e Tecnologia de Santa Catarina. O projeto possui a *hardware* do microprocessador multiciclo e periféricos, descrito em linguagem VHDL e conta com a infraestrutura para a utilização de compiladores de código em linguagem C.

Neste Primeiro Capítulo está situado a organização do trabalho, seguida da justificativa, descrição do problema e objetivos. O Segundo Capítulo apresenta a fundamentação teórica, contemplando os conceitos relacionados com interrupções e

o tratamento delas na ISA RISC-V. O Terceiro Capítulo descreve a metodologia. Por sua vez, o Quarto Capítulo detalha o projeto do tratador de interrupções, seguido da apresentação dos resultados. Por fim, as considerações finais estão presentes no Capítulo 5.

1.1 Justificativa

A eficiência é sempre um parâmetro importante em qualquer sistema, seja em termos de eficiência energética ou temporal. O gerenciamento de eventos por interrupções atua em ambas as frentes, reduzindo o tempo em que o processador fica ocioso e diminuindo a latência na comunicação com os periféricos. As interrupções também permitem que periféricos requisitem o tempo de processamento, viabilizando a atuação do processador de forma assíncrona, reduzindo a latência no processamento dos eventos externos.

1.2 Descrição do problema

Diante dos benefícios trazidos pelo tratamento de eventos por interrupções, como implementar o gerenciamento de interrupções em um núcleo RISC-V?

1.3 Objetivo geral

Desenvolver um controlador de interrupções para um processador RISC-V de acordo com os documentos de especificação da ISA, visando aumentar a eficiência no gerenciamento de periféricos.

1.4 Objetivos específicos

Com o objetivo geral apresentado, destaca-se os seguintes objetivos específicos:

- a) revisar os parâmetros utilizados para gerenciar as interrupções;
- b) projetar um controlador de interrupções;
- c) Implementar um controlador de interrupções;
- d) testar o funcionamento do controlador;
- e) analisar os resultados alcançados.

2 FUNDAMENTAÇÃO TEÓRICA

Este capítulo tem como objetivo apresentar os conceitos em que as interrupções em microprocessadores se baseiam, tanto da perspectiva do *software* quanto do *hardware*. Aborda-se também a topologia RISC-V, explorando os principais tópicos relativos ao tratamento de interrupções para esse sistema.

2.1 Interrupções

Em um microprocessador simples, a unidade de controle busca, decodifica e executa instruções sequencialmente da memória de acordo com o programa armazenado. Portanto, um núcleo sem pipeline e sem replicação das estruturas internas pode executar apenas uma instrução por vez de forma sequencial. É comum que microprocessadores atuem com base em diversos periféricos, demandando que o núcleo execute diversas partes de código para cada um. Para interagir com os periféricos existem duas diferentes abordagens, por *polling* ou interrupção (JIMÉNEZ; PALOMERA; COUVERTIER, 2014).

Em uma abordagem por *polling*, o núcleo verifica ou consulta continuamente o status de cada periférico para determinar se é necessário executar algum serviço. Isso exige que o núcleo permaneça parte do tempo vinculado a cada um dos periféricos apenas para determinar se precisa tratar alguma informação, ocasionando baixa eficiência.

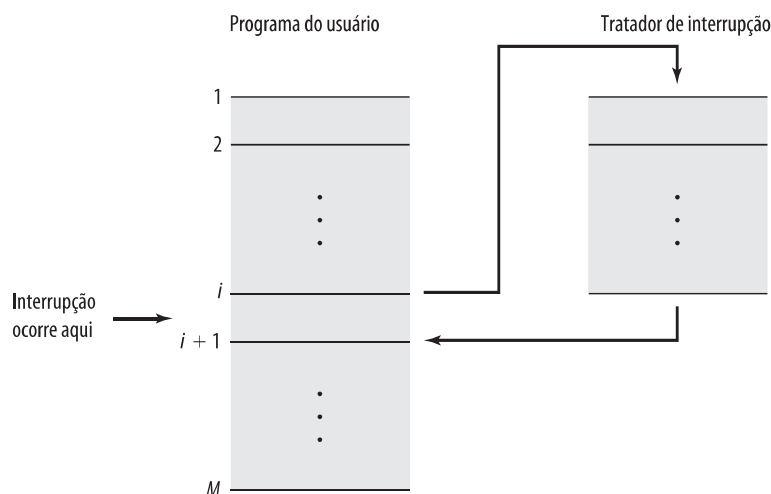
Quando as interrupções são utilizadas, é possível que os periféricos notifiquem o núcleo para que ele troque de contexto quando ocorrer um evento, com o objetivo de melhorar a eficiência do processamento. Segundo Stallings (2010), a maioria dos dispositivos externos é muito mais lenta que o processador, fazendo com que o mesmo fique ocioso entre cada operação de escrita ou leitura dos periféricos. As interrupções resolvem esse problema, uma vez que o processador não precisa interrogar cada periférico, pois ele será notificado quando requisitado pelo dispositivo. A utilização de interrupções permite o uso de modos de baixo consumo ou execução de outras tarefas, ocasionando aumento no número de instruções que realizam trabalho útil, elevando a eficiência do processador.

Além de proporcionar o aumento da eficiência, as interrupções também são utilizadas para possibilitar que sistemas operacionais possam escalonar suas diversas tarefas, por meio das interrupções temporizadas. Do mesmo modo, as interrupções podem ser utilizadas para tratar possíveis falhas em *software*, como um *overflow*, divisão por zero ou tentativa de acesso ilegal a memória, por exemplo (PATTERSON; WATERMAN, 2017).

Quando ocorre a interrupção, é necessário que o tratador de interrupções

tome uma série de ações para que, como visto de forma simplificada na Figura 1, quando a interrupção for finalizada, o contexto do processador retorne ao estado anterior à interrupção. Não deve haver prejuízos ao programa interrompido, além do tempo que ficou em espera.

Figura 1 – Transferência de controle via interrupções



Fonte: Stallings (2010, p. 62).

As interrupções são implementadas de diversas formas para cada arquitetura de processadores, mas Jiménez, Palomera e Couvertier (2014) listam uma série de etapas comuns. Assumindo que o processador esteja executando uma instrução arbitrária os passos são:

1. Concluir a instrução que está sendo executada. Nunca divide-se as instruções, atendendo os pedidos de interrupção sempre entre as instruções.
2. Salvar o valor atual do PC (*Program Counter*). Geralmente são salvos também os registradores do núcleo.
3. Limpar o registrador de habilitação de interrupção global, desabilitando as interrupções.
4. Alterar o valor do PC para o endereço do código responsável por tratar a interrupção.
5. Executar a rotina que gerencia a interrupção correspondente. A rotina termina quando a instrução especial de retorno é executada.
6. Restaurar o *Program Counter* e qualquer outro registrador que foi salvo na Etapa 2. Restaurar o PC causa a retomada do programa interrompido na próxima instrução de onde foi previamente interrompido.

2.2 RISC-V

O RISC-V foi introduzido em 2011, com o intuito de ser uma ISA universal, e para isso, segundo Patterson e Waterman (2017), o RISC-V foi desenvolvido pensando em satisfazer os seguintes requisitos:

- Ser eficiente para todos os tipos de microarquiteturas.
- Possuir flexibilidade, atendendo desde o menor microcontrolador embarcado até um computador de alto desempenho.
- Ser compatível com as linguagens de programação mais populares do mercado.
- Ter estabilidade: Sua ISA base deve permanecer sem ser alterada ou descontinuada.
- Acomodar diferentes tecnologias para sua implementação, tais como *FPGAs* (*Field-Programmable Gate Arrays*), *ASICs* (*Application-Specific Integrated Circuits*), *chips* customizados e futuras tecnologias.

Outro ponto importante para a popularidade e perspectiva de crescimento do RISC-V, é que diferentemente da maior parte das outras ISAs, ele é fornecido sob licenças de código aberto, não exigindo taxas para seu uso comercial, criando assim um interesse maior pelo seu uso e gerando uma comunidade ativa em torno de si com a *RISC-V Foundation* (PATTERSON; WATERMAN, 2017).

Assim como outras ISAs, o RISC-V fornece proteção entre diferentes contextos fazendo isso na forma de instruções privilegiadas. Especifica-se os níveis de privilégio de usuário, supervisor, hiper-visor e máquina (PATTERSON; WATERMAN, 2017, p. 109). O nível de máquina possui o maior grau de privilégio e é o único modo obrigatório da ISA, com o código rodando sobre esse nível sendo inerentemente confiável visto que é o mais baixo nível de acesso ao *hardware*. Já os modos de supervisor e usuário são direcionados para o sistema operacional e para as aplicações de propósito geral respectivamente, enquanto o modo hiper-visor provê suporte para virtualização de sistema operacional (WATERMAN et al., 2016a, p. 3).

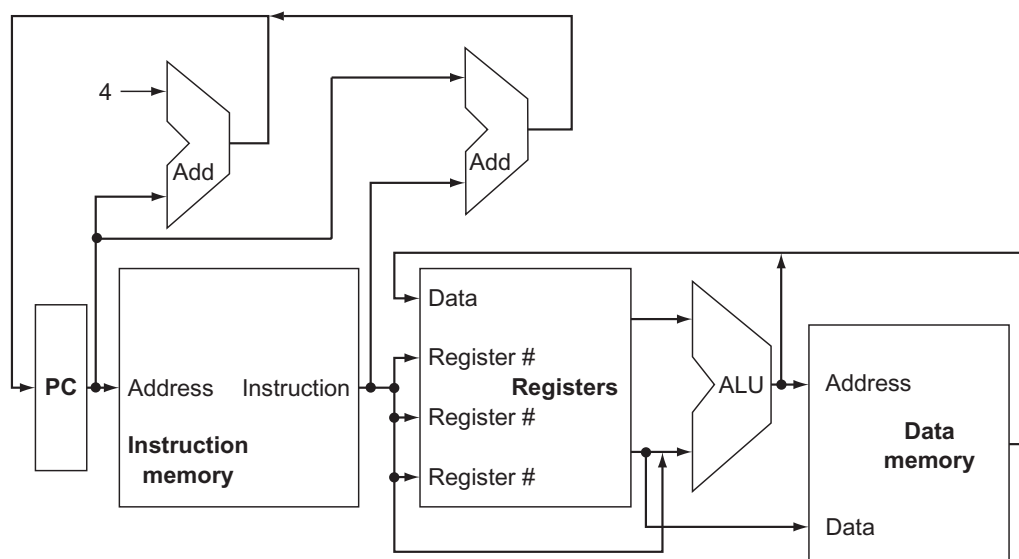
2.2.1 Arquitetura RISC-V

Quanto a implementação da arquitetura, é apresentado na Figura 2 de forma simplificada os principais componentes que compõe um núcleo RISC-V para uma implementação a nível de máquina. Sendo eles:

- PC - (*Program Counter*): É o registrador que indica qual a posição de memória da instrução atual do processador.

- **Add - (Adders):** Somadores responsáveis por calcular o próximo valor do PC.
- **Instruction memory:** É a região de memória onde é armazenado o programa a ser executado.
- **Registers:** São os registradores responsáveis pelo armazenamento interno ao núcleo.
- **ALU - (Arithmetic Logic Unit):** A unidade lógica aritmética é encarregada de realizar os cálculos e operações lógicas, como soma, subtração, multiplicação, deslocamento, entre outros.
- **Data Memory:** É a região de memória volátil, onde é salvo o contexto atual de execução, como variáveis, vetores, e outros tipos de dados.

Figura 2 – Diagrama de implementação das principais unidades funcionais do RISC-V e suas interconexões



Fonte: Patterson (2017, p. 238).

Com os componentes presentes na Figura 2, é possível realizar três classes de instruções, as instruções lógicas e aritméticas, as instruções de acesso a memória e as instruções de desvio. Para todos os conjuntos de instruções, os primeiros dois passos realizados pelo núcleo são os mesmos. Primeiramente a instrução a ser processada é carregada da *Instruction memory*, no endereço apontado pelo registrador PC, a esse passo é dado no nome de *Fetch*. O segundo passo é o *Read*, onde são lidos os registradores apontados pela instrução atual (PATTERSON, 2017).

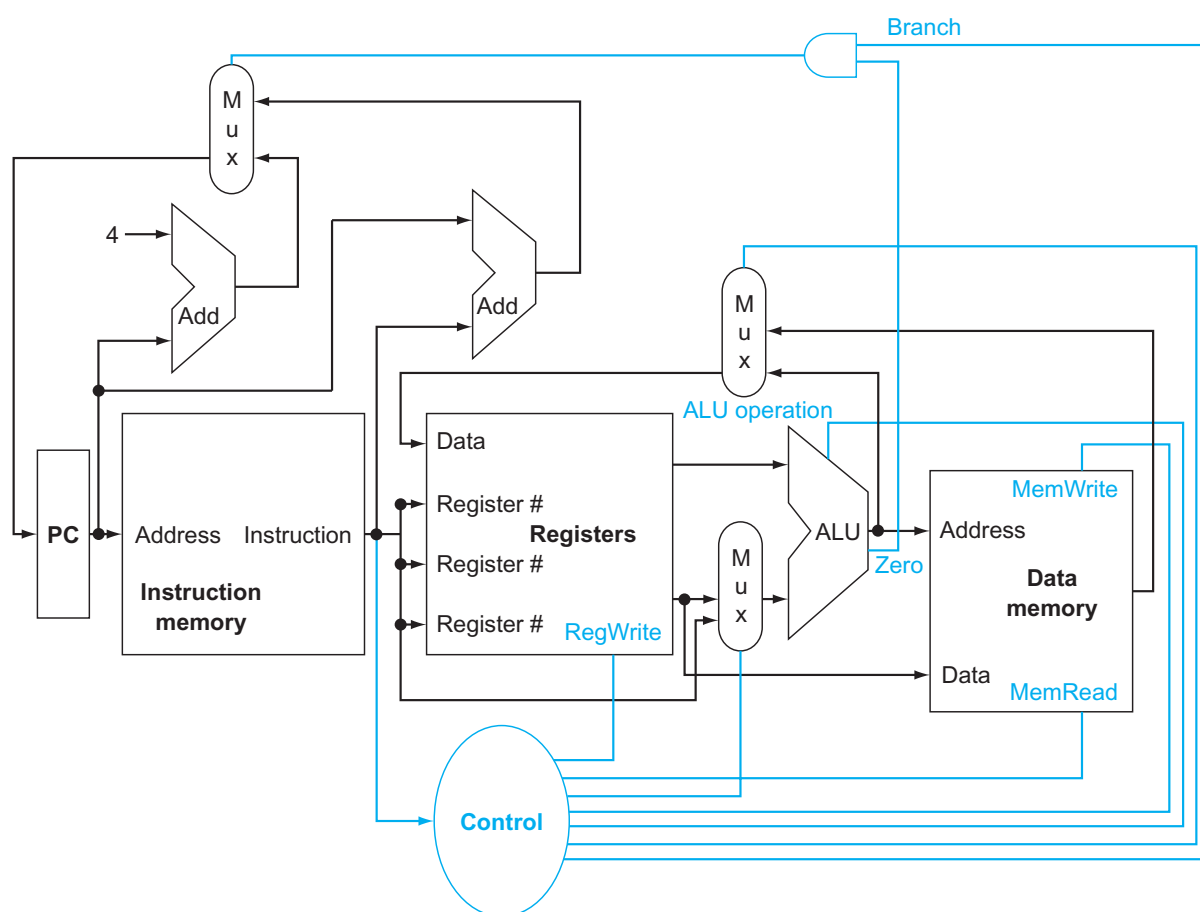
Após essas duas etapas, as próximas ações para execução da instrução dependem da classe de instrução, com cada uma manipulando a ALU para diferentes

propósitos. As instruções lógicas e aritméticas utilizam-se da ALU para realizar suas operações. As instruções de acesso a memória utilizam a ALU para o cálculo dos endereços de memória a serem lidos ou escritos. Já as instruções de desvio podem também utilizar a ALU para as comparações que determinarão qual será a próxima instrução a ser executada (PATTERSON, 2017).

Depois de usar a ALU, as ações necessárias para concluir as instruções das várias classes são diferentes. Uma instrução de acesso a memória precisará acessá-la para ler ou gravar dados. Uma instrução lógico-aritmética deve escrever os dados da ALU em um registrador. Por último, para uma instrução de desvio condicional, pode ser necessário alterar o próximo endereço do PC com base na comparação, caso contrário, o PC deve ser incrementado em quatro para obter o endereço da instrução subsequente (PATTERSON, 2017).

A Figura 2 possui somente as principais conexões entre os componentes funcionais e abstrai os multiplexadores e o sistema de controle, necessários para uma implementação real. Esses componentes e suas interconexões estão dispostas na Figura 3, onde é possível verificar a adição de três multiplexadores e da unidade de controle. O componente *Control* tem como parâmetro de entrada a instrução atual, e com base nela traça os passos a serem seguidos para o correto processamento da instrução, realizando o controle de escrita e leitura nos componentes de memória e do gerenciamento do fluxo de dados por meio dos multiplexadores.

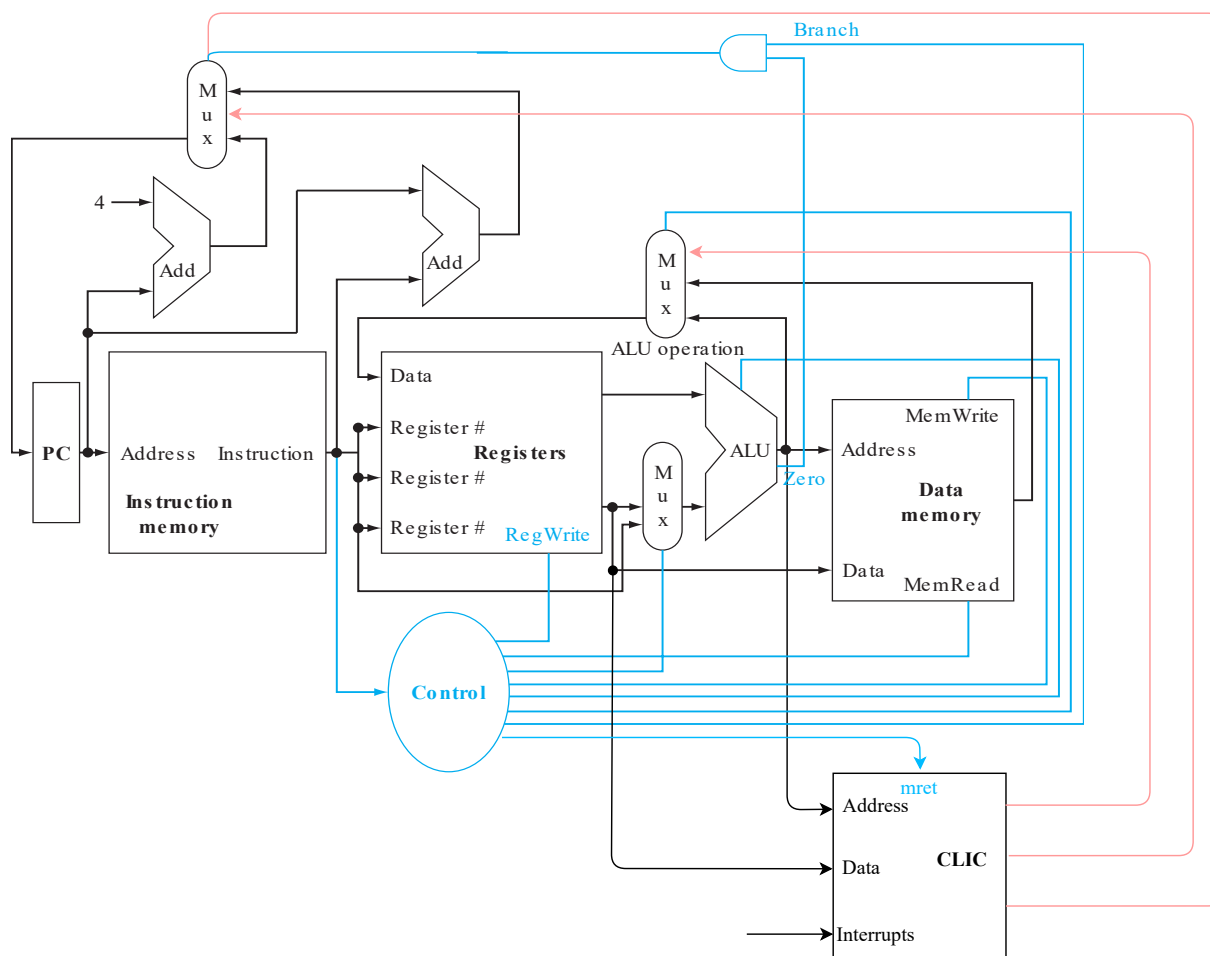
Figura 3 – Diagrama das principais unidades funcionais do RISC-V, incluindo multiplexadores e linhas de controle necessárias



Fonte: Patterson (2017, p. 239).

As unidades funcionais apresentadas na Figura 3, conseguem executar as instruções básicas de controle de fluxo, acesso a memória e cálculo, mas não fornecem suporte as interrupções do núcleo. Para que seja possível tratar as interrupções, é necessário adicionar o componente CLIC (*Core-Local Interrupt Controller*), com suas respectivas conexões com os demais elementos do núcleo, como na Figura 4.

Figura 4 – Diagrama das principais unidades funcionais do RISC-V, incluindo o controlador de interrupções



Fonte: Adaptado de Patterson (2017, p. 239).

A unidade CLIC é responsável por controlar os registradores de controle e status referentes as interrupções, que serão abordados na Subseção 2.2.2, e tem como principal função alterar o fluxo do programa para que seja executada uma região específica de código, que por sua vez realiza as operações necessárias para tratar corretamente a interrupção, como abordado na Seção 2.1.

O RISC-V divide as interrupções em três diferentes tipos. As interrupções de *software*, de *Timer* e externas. Dentre esses três tipos de interrupções, as que possuem maior prioridade são as interrupções externas, seguidas das de *Timer* e por fim das de *software*. Cada interrupção pertencente a uma dessas classes possui também diferentes níveis de prioridade (WATERMAN et al., 2016a, p. 29).

Enquanto outras ISAs utilizam os termos interrupção e exceção como sinônimos, o RISC-V usa o termo exceção para qualquer condição anormal de operação

associado a uma instrução, enquanto o termo interrupção refere-se a um evento externo ao núcleo. Ambas requerem que ocorra a transferência de controle do núcleo (WATERMAN et al., 2016b).

2.2.2 Registradores de controle e status

O RISC-V define uma série de registradores de acesso privilegiado que são utilizados para controle e status do microprocessador, também é disponibilizado parte dos endereços para uso de propósito específico em cada implementação. No contexto de interrupções, existem oito principais registradores, sendo eles:

- **mstatus** - *Machine status register*

O registrador *mstatus* rastreia e controla o estado operacional de cada núcleo RISC-V, a composição completa do registrador pode ser vista na Figura 5. Onde *MPIE* (*Machine Pending Interrupt Enable*) armazena o valor de habilitação de interrupção enquanto há uma interrupção pendente. Os *bits* *HPIE*, *SPIE* e *UPIE* são análogos ao *MPIE*, porém são utilizados respectivamente pelos níveis de privilégio de hiper-visor, supervisor e usuário. O *bit* *MIE* (*Machine Interrupt Enable*) mantém o valor do bit de habilitação de interrupção global, sendo *HIE*, *SIE* e *UIE* análogos a ele (WATERMAN et al., 2016a).

O *bits* *MPP* (*Machine Previous Privilege*) e suas variações *HPP* e *SPP* mantém o nível de privilégio anterior à ocorrência de uma interrupção, de forma que é possível restaurar o nível de privilégio caso este seja alterado pelo tratador de interrupção (WATERMAN et al., 2016a).

Os demais registradores são responsáveis por tarefas relacionadas a memória, *MXR* (*Make eXecutable Readable*) e *PUM* (*Protect User Memory*) gerenciam o privilégio de memória; *SD*, *FS* e *XS* fazem o controle de contexto nas diferentes extensões do RISC-V; e o campo *VM* faz a gerenciamento de virtualização de memória. Os campos marcados com *WPRI* são reservados para uso futuro (WATERMAN et al., 2016a).

Figura 5 – Machine-mode status register

XLEN-1	XLEN-2	29	28	24	23	20	19	18	17	16	15			
SD	WPRI	VM[4:0]	(WARL)	WPRI	MXR	PUM	MPRV	XS[1:0]						
1	XLEN-30	5	4	1	1	1	2							
14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
FS[1:0]	MPP[1:0]	HPP[1:0]	SPP	MPIE	HPIE	SPIE	UPIE	MIE	HIE	SIE	UIE			
2	2	2	1	1	1	1	1	1	1	1	1	1	1	1

Fonte: Waterman et al. (2016a, p. 20).

O símbolo **XLEN** é referente ao tamanho dos registradores do processador, podendo ser de 32 ou 64 bits atualmente.

- **mip** - *Machine interrupt pending register*

O registrador *mip* lista as interrupções atualmente pendentes para cada modo de privilégio e para cada classe de interrupção, conforme a Figura 6. Os *bits* MEIP, MTIP e MSIP correspondem respectivamente as interrupções externas, de *timer* e de *software* pendentes. Da mesma maneira, os demais *bits* possuem a mesma atribuição, representando essas mesmas classes para os modos de hiper-visor, supervisor e usuário.

Figura 6 – Machine interrupt-pending register

XLEN-1	12	11	10	9	8	7	6	5	4	3	2	1	0
WIRI	MEIP	HEIP	SEIP	UEIP	MTIP	HTIP	STIP	UTIP	MSIP	HSIP	SSIP	USIP	
XLEN-12	1	1	1	1	1	1	1	1	1	1	1	1	1

Fonte: Waterman et al. (2016a, p. 28).

- **mie** - *Machine interrupt-enable register*

Da mesma forma que o registrador *mip*, o registrador *mie* lista as interrupções atualmente habilitadas para cada modo de privilégio e para cada classe de interrupção, conforme a Figura 7. Sendo os *bits* MEIE, MTIE e MSIE correspondem respectivamente as habilitações das interrupções externas, de *timer* e de *software*. Com os demais *bits* representando essas mesmas classes para os modos de hiper-visor, supervisor e usuário.

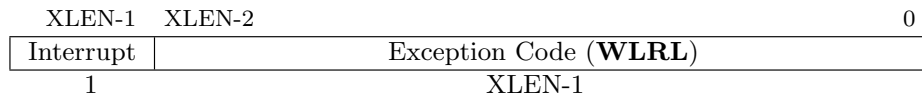
Figura 7 – Machine interrupt-enable register

XLEN-1	12	11	10	9	8	7	6	5	4	3	2	1	0
WPRI	MEIE	HEIE	SEIE	UEIE	MTIE	HTIE	STIE	UTIE	MSIE	HSIE	SSIE	USIE	
XLEN-12	1	1	1	1	1	1	1	1	1	1	1	1	1

Fonte: Waterman et al. (2016a, p. 28).

- **mcause** - *Machine cause register*

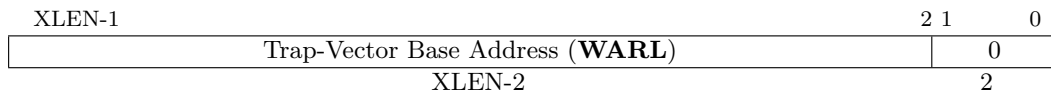
O registrador *mcause* é responsável por indicar qual a causa da exceção ou interrupção que ocorreu.

Figura 8 – Machine cause register

Fonte: Waterman et al. (2016a, p. 34).

- **mtvec** - *Machine trap-vector base-address register*

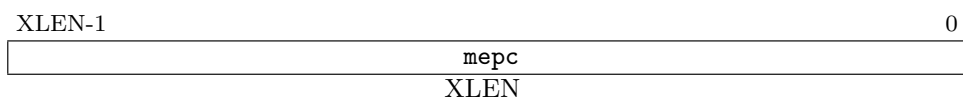
O registrador *mtvec* armazena o endereço base do vetor de tratadores de interrupções, sempre alinhado em 4 *bytes*, fazendo com que os dois *bits* menos significativos sejam sempre zero, em conformidade com a Figura 9

Figura 9 – Machine trap-vector base-address register

Fonte: Waterman et al. (2016a, p. 26).

- **mepc** - *Machine exception program counter register*

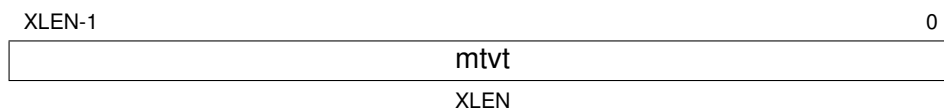
O registrador *mepc* captura o valor do próximo contador de programa no momento que ocorre uma interrupção, esse valor é utilizado para retornar ao final do tratamento da interrupção (WATERMAN et al., 2016a, p. 33).

Figura 10 – Machine exception program counter register

Fonte: Waterman et al. (2016a, p. 34).

- **mtvt** - *Machine trap vector table base-address register*

O registrador *mtvt* é um CSR personalizado, usado para armazenar o endereço do código comum para exceções.

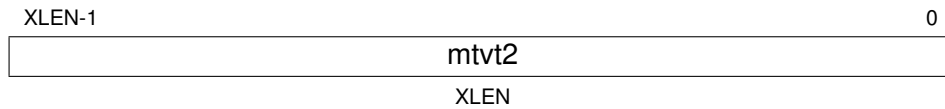
Figura 11 – Machine trap vector table base-address register

Fonte: Elaboração própria.

- **mtvt2** - *Machine trap vector table 2 base-address register*

O registrador *mtvt2* é um CSR personalizado, usado para armazenar o endereço do código comum para interrupções.

Figura 12 – Machine trap vector table 2 base-address register



Fonte: Elaboração própria.

2.2.3 Instruções especiais das interrupções

Tratando-se do acesso aos registradores de status de controle e do tratamento de interrupções, o RISC-V possui instruções específicas para estes propósitos, sendo elas:

- **CSR** - *Control and status register instructions*

Como os registradores da Subseção 2.2.2 são de acesso privilegiado, o RISC-V implementa seis instruções específicas para lidar com eles, em nível de privilégio de máquina. As instruções e os campos que as compõe estão descritas na Figura 13.

Figura 13 – Control and Status Registers Instructions

31	20 19	15 14	12 11	7 6	0
csr	rs1	funct3	rd	opcode	
12	5	3	5	7	
source/dest	source	CSRRW	dest	SYSTEM	
source/dest	source	CSRRS	dest	SYSTEM	
source/dest	source	CSRRC	dest	SYSTEM	
source/dest	uimm[4:0]	CSRRWI	dest	SYSTEM	
source/dest	uimm[4:0]	CSRRSI	dest	SYSTEM	
source/dest	uimm[4:0]	CSRRCI	dest	SYSTEM	

Fonte: Waterman et al. (2016b, p. 22).

A instrução CSRRW lê o valor do CSR, estendendo-o com zeros caso necessário e o escrevendo em rd, enquanto o valor do registrador rs1 se torna o novo valor do CSR.

A instrução CSRRS lê o valor do CSR, estendendo com zeros caso necessário e o escrevendo em rd, enquanto o valor do registrador rs1 é tratado como uma máscara de bits que especifica as posições dos bits a serem definidas no CSR.

A instrução CSRRRC lê o valor do CSR, o estendendo com zeros caso necessário e o escrevendo em rd, enquanto o valor do registrador rs1 é tratado como uma máscara de bits que especifica as posições dos bits a serem limpos no CSR.

As variantes CSRRWI, CSRRSI e CSRRCI são similares a CSRRW, CSRRS e CSRRC respectivamente, exceto por o valor de rs1 ser utilizado como imediato para cada operação ao invés do valor do registrador (WATERMAN et al., 2016a, p.22).

- **MRET** - *Trap-return instructions*

A instrução de retorno de interrupção *MRET* altera o contador de programa para o valor contido no CSR *mepc*, retornando ao ponto do código que havia sido interrompido.

Figura 14 – Trap-return instructions

31	20 19	15 14	12 11	7 6	0
funct12	rs1	funct3	rd	opcode	
12	5	3	5	7	
MRET/HRET/SRET/URET	0	PRIV	0	SYSTEM	

Fonte: Waterman et al. (2016b, p. 35).

Existem também outras instruções de retorno de interrupções, uma para cada modo de privilégio como observado na Figura 14, mas somente a implementação da instrução *MRET* é obrigatória.

As instruções de escrita e leitura dos CSRs são essenciais para a execução das etapas 3 e 4 citadas na Seção 2.1. Essas instruções atuam diretamente no registrador *mstatus*, que desabilita as interrupções globais, conforme a etapa 3 da Seção 2.1, elas controlam também os registradores *mtvec*, *mcause* e *mtvt2*, responsáveis pelo fluxo de execução das interrupções.

Já a instrução *MRET* é fundamental para restaurar o *PC*, finalizando o tratamento da interrupção conforme a etapa 6 da Seção 2.1.

2.3 Dispositivos lógicos programáveis

Os dispositivos lógicos programáveis (*PLDs - Programmable Logic Devices*) foram introduzidos em meados da década de 1970 com o intuito de construir circuitos lógicos combinacionais programáveis. No entanto, ao contrário dos microprocessadores que podem executar diferentes programas (mas com *hardware* fixo), a programabilidade dos PLDs é a nível de *hardware*. Em outras palavras, um PLD é um *chip* de uso

geral cujo *hardware* pode ser configurado para atender a especificações particulares (PEDRONI, 2008).

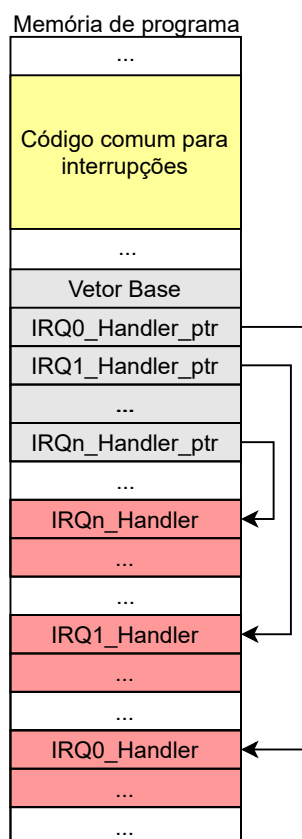
Existem diversas arquiteturas de PLDs, divididos basicamente em SPLDs (Simple PLDs), CPLDs (Complex PLDs) e FPGAs (Field Programmable Gate Arrays). Dentre estes, segundo Pedroni (2008) os FPGAs destinam-se principalmente a projetos complexos, de grande porte e de alto desempenho, como transceptores de *gigabit*, comutadores de alta frequência, HDTV, transmissores sem fio e outras aplicações de telecomunicações.

Por possibilitarem a implementação de *hardware* de forma prática e ágil, os FPGAs tornaram-se parte do processo de desenvolvimento de circuitos integrados digitais. Uma vez que aliados aos *softwares* de síntese e depuração, fornecem ferramentas que facilitam o desenvolvimento, encurtando assim o tempo entre a concepção e a produção de dispositivos de *hardware*.

2.4 Elementos de software em interrupções

Nas seções anteriores foram observados os aspectos do *hardware* necessário para a implementação de um núcleo RISC-V com suporte a interrupções. Contudo, para que seja possível executar aplicações que se utilizem desse mecanismo, é necessária a adição de componentes de *software*, responsáveis por realizar parte das etapas vistas na Seção 2.1. Esses elementos de *software* são apresentados na Figura 15, sendo eles:

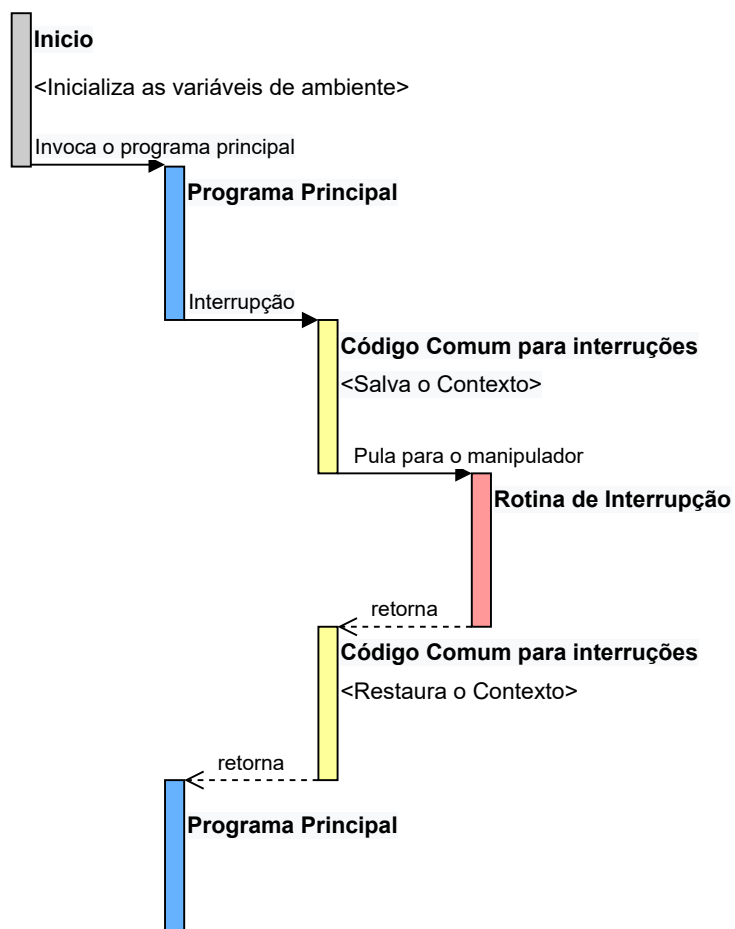
- Código comum para interrupções: É o código que gerencia a interrupção, sendo o primeiro a ser processado após a ocorrência de um evento, tendo as atribuições de salvar o contexto dos registradores do núcleo, invocar o manipulador específico de cada interrupção e por fim, restaurar os registradores do núcleo.
- Vetor Base: É o vetor que armazena os ponteiros para os endereços dos manipuladores das interrupções.
- Manipuladores das interrupções: É o código que trata o evento por trás da interrupção, geralmente atuando nas informações oriundas do periférico pelo qual é responsável.

Figura 15 – Componentes de software para gerenciar interrupções.

Fonte: Elaboração própria.

Além de introduzir os componentes presentes na Figura 15, também foram alterados outros componentes já presentes. De forma que é apresentado na Figura 16 o diagrama do fluxo de programa com a ocorrência de uma interrupção, nela é possível visualizar os principais elementos de *software*, que são:

- O código inicial, responsável por inicializar as variáveis de ambiente e os registradores *CSRs* necessários;
- O programa principal, que é encarregado das funções básicas do sistema, tais como iniciar os periféricos e interrupções;
- O código comum para interrupções e a rotina de interrupção, ambos já mencionados anteriormente.

Figura 16 – Diagrama do fluxo de programa com interrupção.

Fonte: Elaboração própria.

3 METODOLOGIA

Este trabalho tem o intuito de apresentar os conceitos e técnicas relacionadas a implementação do mecanismo de interrupções em um microprocessador RISC-V. Dessa forma, Gil (2002) classifica esse tipo de pesquisa como exploratória, pois tem como objetivo propiciar maior intimidade com o problema, buscando torná-lo mais explícito ou a constituir hipóteses. Pode-se afirmar que estes tipos de pesquisas têm como objetivo principal o aperfeiçoamento de ideias ou a descoberta de intuições.

Para o desenvolvimento do *hardware*, foram empregados as aplicações *Sigasi Studio XL* (Sigasi, 2021), para edição e gerenciamento do projeto, *ModelSim* (Siemens, 2021), para simulação e depuração do sistema, e o *Quartus Prime* (Intel, 2021) para compilação e síntese física do *hardware*. Para a implementação dos elementos de *software* do microprocessador, foi utilizado o compilador RISC-V *GCC* (GCC, 2021), que gera o código binário utilizado pelo microprocessador.

Também foi empregada a linguagem VHDL (*Very High Speed Integrated Circuits Hardware Description Language*), que se trata de uma linguagem de descrição de *hardware* definida pelo padrão IEEE 1164 (PEDRONI, 2008).

Este trabalho trata principalmente da elaboração dos mecanismos de controle e gerenciamento de interrupções. Por esse motivo, foi utilizado como base o projeto desenvolvido pelo Prof. Dr. Renan Augusto Starke (STARKE, 2021) que implementa apenas o núcleo RISC-V na versão multi-ciclo. Este projeto é utilizado no contexto da unidade curricular de Dispositivos Lógicos-Programáveis (PLD) do Curso Superior de Engenharia Eletrônica pertencente ao Departamento Acadêmico de Eletrônica (DA-ELN) do Instituto Federal de Santa Catarina (IFSC). Esse projeto tem como objetivo desenvolver um microcontrolador de forma colaborativa entre os alunos do final do curso, bem como trazer temas de projeto de conclusão de curso multidisciplinares relevantes.

A partir do projeto base, foram analisados os componentes e mecanismos necessários para a implementação do controle de interrupções, identificando os elementos que precisariam ser adicionados ou modificados. Na sequência, foram implementadas alterações necessárias, apresentando a simulação e princípio de funcionamento de cada um individualmente e da operação do sistema completo.

Para a implementação do projeto do microcontrolador RISC-V, foi utilizada a FPGA Intel MAX 10, presente na placa de desenvolvimento DE10-LITE da fabricante TerasiC (TerasiC Inc., 2021). Contando com 50.000 elementos lógicos e unidades de memória incorporadas no encapsulamento, o MAX 10 fornece as condições necessárias para a implementação das unidades básicas vistas na seção 2.2.

A implementação completa foi submetida ao repositório de Starke (2021),

e será utilizada pelos alunos da unidade curricular de PLD na implementação de interrupções em novos periféricos.

4 PROJETO E RESULTADOS

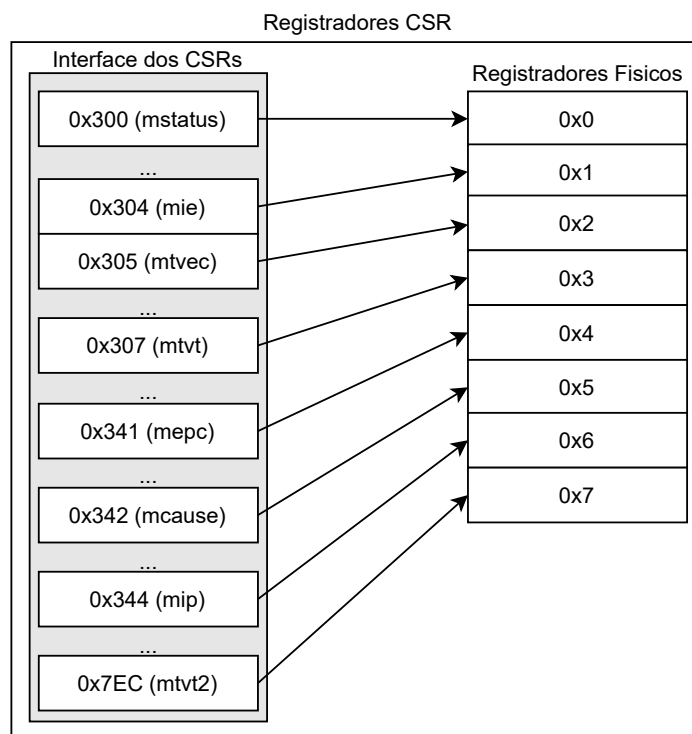
Neste capítulo será abordado o projeto do controlador de interrupções para o núcleo RISC-V desenvolvido por Starke (2021). Apresenta-se também a implementação dos componentes de *hardware*, tais como o controle dos registradores CSRs, as instruções de acesso aos CSRs, de retorno de interrupção e a sua integração com os elementos de *software*.

4.1 Inserção e controle dos registradores CSRs

Os registradores de controle e status são armazenados diretamente no núcleo do processador na forma de um banco de registradores. Possuindo 12 *bits* de endereçamento, esse banco pode conter até 4096 registradores, cada um deles destinado a funções específicas.

Como o projeto do tratador de interrupções requer somente uma fração dos registradores CSRs e os endereços dos registradores utilizados não são sequenciais, foi implementada uma interface para manter a coerência dos endereços com as especificações da ISA. Dessa forma, foi implementado um banco com 8 registradores, com a interface encarregando-se de redirecionar os dados aos endereços desse banco, como apresentado na Figura 17.

Figura 17 – Diagrama da interface dos registradores CSRs



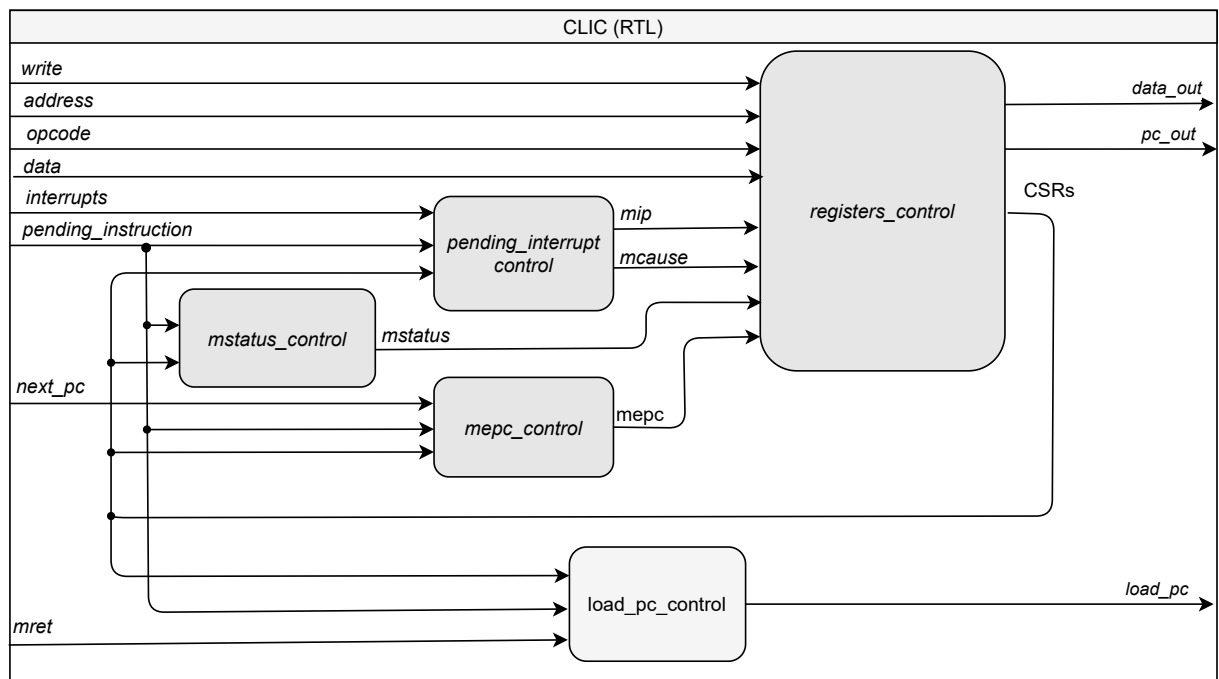
Fonte: Elaboração própria.

Dessa maneira, os endereços dos registradores são os mesmos das especificações, sem que seja necessário destinar recursos de memória do FPGA para registradores sem uso. Além disso, não se prejudica a adição de novas funcionalidades no futuro, em vista que é possível introduzir novos CSRs caso necessário.

Os registradores CSRs são acessados por *software* e *hardware*, de forma que há registradores em que o *software* não pode realizar operações de escrita. Dos registradores implementados, *mepc*, *mcause* e *mie* podem apenas serem lidos pelo *software*, isso ocorre porque o valor desses registradores é definido pelo *hardware* no momento em que ocorre uma interrupção. Para proteger esses registradores das operações de escrita foi gerado uma máscara, impedindo a modificação do valor dos mesmos.

No Apêndice A está contido o código em linguagem *VHDL* do componente que implementa o registro e controle dos CSRs utilizados e também gerencia as instruções de acesso aos CSRs e de retorno de interrupção. Esse componente é representado de forma simplificada na Figura 18.

Figura 18 – Representação do controlador dos registradores CSRs.



Fonte: Elaboração própria.

Como os registradores *mepc*, *mcause*, *mie* e partes de *mstatus* são controlados por *hardware*, foram designados blocos específicos para gerenciá-los. Sendo os seguintes blocos responsáveis por esses quatro registradores:

- *Pending interrupt control*:

No escopo desse projeto foram implementadas as interrupções externas e de *timer*, provendo o mecanismo de interrupções para os periféricos e temporizadores. O bloco *pending interrupt control* recebe o vetor *interrupts*, que contém todas as interrupções do sistema, e com isso controla o valor dos registradores *mip* e *mcause*.

O registrador *mip* mantém os campos *MEIP* e *MTIP* definidos para nível lógico alto quando há respectivamente uma ou mais interrupções externas ou de temporizador pendentes. Já o registrador *mcause* é definido com o identificador da interrupção a ser executada, sendo esse valor utilizado para calcular o endereço do tratador da interrupção em questão.

- *Mstatus control*:

Quando o núcleo é interrompido, o bloco *mstatus control*, fica encarregado de armazenar no *bit MPIE* o valor de *MIE*, limpando *MIE* em seguida. O mesmo ocorre para os outros níveis de privilégio, porém, como o núcleo implementado possui somente o nível de máquina, só serão tratados os *bits* pertinentes a esse nível. Quando o núcleo termina de executar a interrupção, a operação inversa é realizada, alterando o valor do *bit MIE* para o valor presente em *MPIE* e limpando *MIE*.

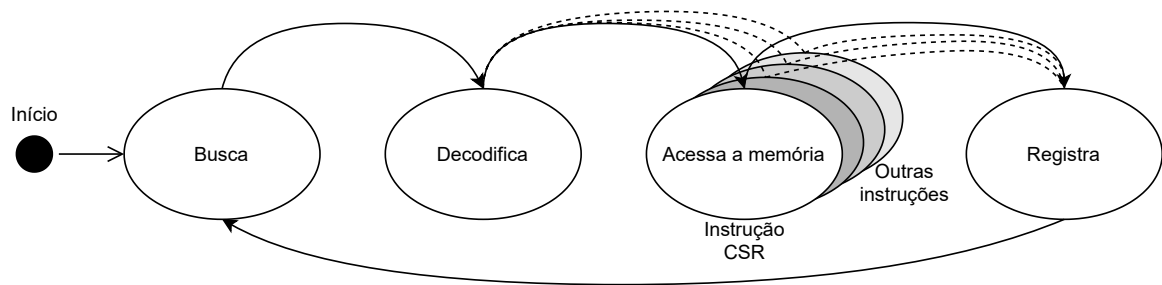
- *Mepc control*:

O bloco de controle do registrador *mepc* comanda a captura do próximo valor do *PC*. Armazena-se o endereço da próxima instrução a ser executada no momento em que o núcleo troca de contexto para a interrupção. Para realizar esta operação, foi adicionado ao componente que gerencia o *PC* uma saída adicional contendo o valor do endereço da próxima instrução a ser executada. Caso ocorra uma interrupção durante a execução de uma instrução de desvio, a instrução de retorno de interrupção pode restituir o programa no endereço correto.

4.2 Implementação das Instruções CSR

A maioria das instruções, incluindo as instruções de acesso aos registradores *CSRs*, seguem o diagrama de estados apresentado na Figura 19. Com os dois primeiros estados sendo responsáveis, respectivamente pela leitura e decodificação de uma instrução da memória.

Figura 19 – Diagrama de estados das instruções *CSRs*



Fonte: Elaboração própria.

Como mencionado na subseção 2.2.3, as instruções de *CSR* são agrupadas como operações de sistema, com a distinção entre cada instrução dessa classe realizada com base no campo "*funct3*". Na Tabela 1 estão presentes os campos *opcode* e *funct3* que caracterizam cada uma das seis instruções de acesso aos *CSRs*.

Tabela 1 – Especificações das instruções de acesso aos registradores *CSRs*

31	20	19	15	14	12	11	7	6	0	
<i>source/dest</i>	<i>source</i>		<i>funct3</i>			<i>dest</i>		<i>opcode</i>		
csr	rs1		001			rd		1110011		CSRRW
csr	rs1		010			rd		1110011		CSRRS
csr	rs1		011			rd		1110011		CSRRC
csr	zimm		101			rd		1110011		CSRRWI
csr	zimm		110			rd		1110011		CSRRSI
csr	zimm		111			rd		1110011		CSRRCI

Fonte: Adaptado de Waterman et al. (2016b).

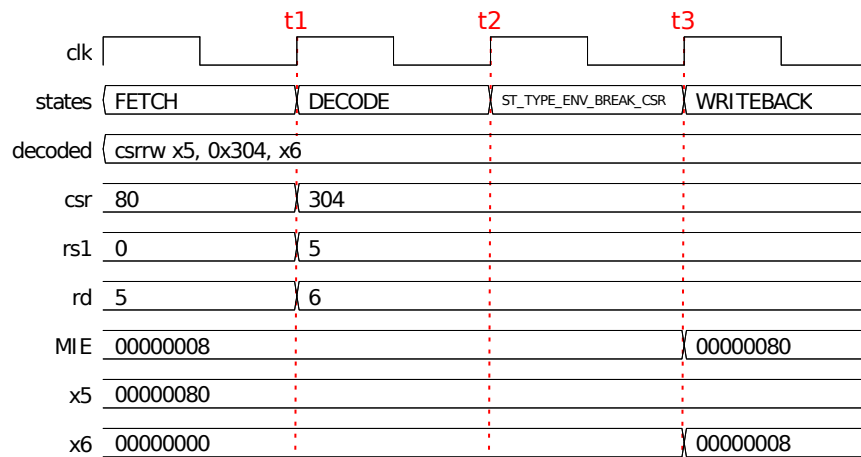
Caso a unidade de controle identifique uma das instruções acima no processo de decodificação, o próximo passo é realizar o acesso aos registradores, com as particularidades de cada operação. Para exemplificar o funcionamento da implementação, a seguir são apresentados as simulações do gerenciamento dessas instruções pelo sistema.

Para demonstrar o funcionamento da instrução CSRRW, o exemplo da Figura 20 realiza a operação de escrita do valor do registrador x5 no *CSR MIE*, e

armazena o valor de *MIE* no registrador x6. Conforme o diagrama apresentado na Figura 19, as etapas para a execução da instrução são as seguintes:

1. Ler a instrução da memória, visto no campo "*decoded*" da Figura 20.
2. Decodificar os campos definidos pelo formato da instrução, carregando os valores dos campos "*csr*", "*rs1*" e "*rd*".
3. Executar o acesso ao registrador MIE, realizando a leitura do valor atual.
4. Registrar o valor lido de MIE no registrador x6 e atualizar o valor de MIE para o obtido de x5.

Figura 20 – Diagrama de execução da instrução CSRRW

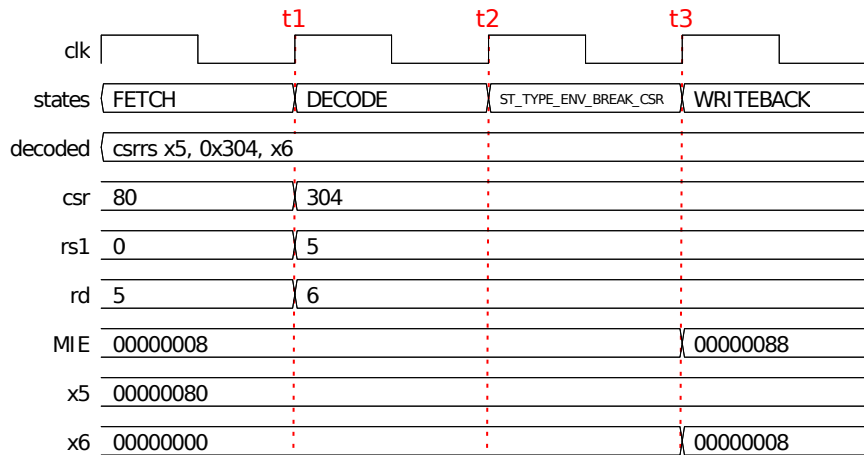


Fonte: Elaboração própria.

Geralmente é necessário apenas escrever um novo valor em um registrador *CSR*, sem que o valor anterior precise ser salvo em um registrador do núcleo. Essa operação é realizada por meio da utilização do registrador x0 na posição *rd*, visto que o valor deste registrador não pode ser alterado e é sempre zero.

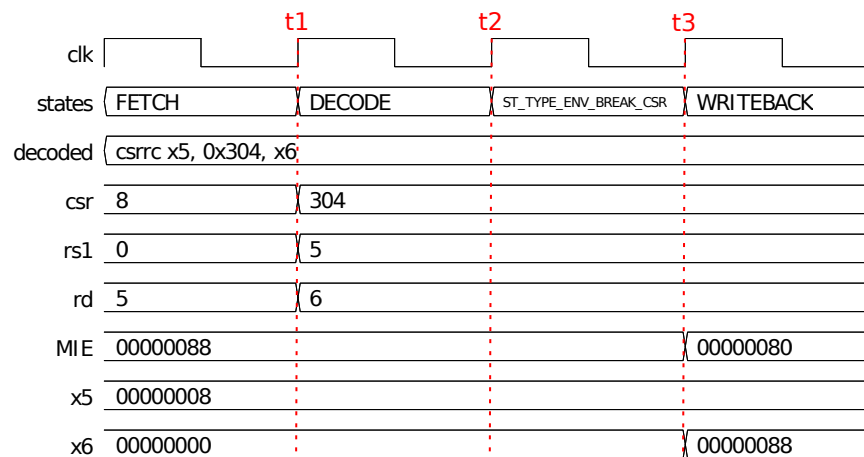
Como pode ser visto na Figura 20, o valor anterior do registrador MIE não é utilizado para definir seu novo valor, de forma que a operação realizada pela instrução CSRRW altera totalmente o registrador em questão. Em contrapartida, as instruções CSRRS e CSRRC a seguir utilizam o valor anterior do CSR para calcular o novo valor.

A Figura 21 contém os mesmos registradores do caso anterior, porém a instrução executada é a CSRRS, que utiliza o valor contido em *rs1* como uma máscara para definir os bits do registrador CSR pra nível alto. Essa abordagem, ao contrário da anterior, não afeta os *bits* que já estão ativos, como é possível verificar na etapa 4 da Figura 21.

Figura 21 – Diagrama de execução da instrução CSRRS

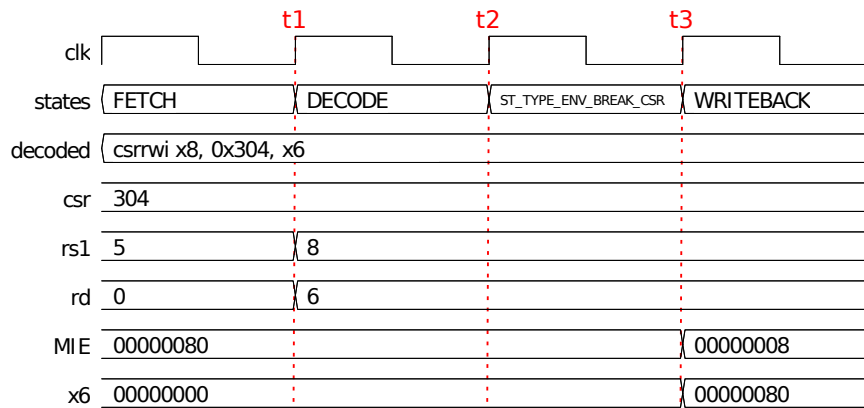
Fonte: Elaboração própria.

Igualmente ao caso anterior, a Figura 21 contém os mesmos registradores do primeiro exemplo, porém desta vez a instrução executada é a CSRRC, que manipula o valor contido em rs1 como uma máscara para limpar os *bits* do registrador CSR. Assim como na instrução vista anteriormente, a instrução CSRRC não afeta os *bits* que não pertencem a máscara contida em rs1.

Figura 22 – Diagrama de execução da instrução CSRRC

Fonte: Elaboração própria.

As instruções CSRRWI, CSRRSI e CSRRCI têm funcionamento semelhante as instruções CSRRW, CSRRS e CSRRC, respectivamente. Com a única diferença que, ao invés de utilizar o valor do campo rs1 como endereço de registrador, este valor é empregado como imediato. Dessa forma, evita-se uma instrução adicional para carregar um valor em um registrador, todavia, há a limitação do campo rs1 possui somente 5 *bits*, restringindo o uso dessas instruções para imediatos maiores que 5 bits.

Figura 23 – Diagrama de execução da instrução CSRRWI

Fonte: Elaboração própria.

4.3 Implementação da Instrução MRET

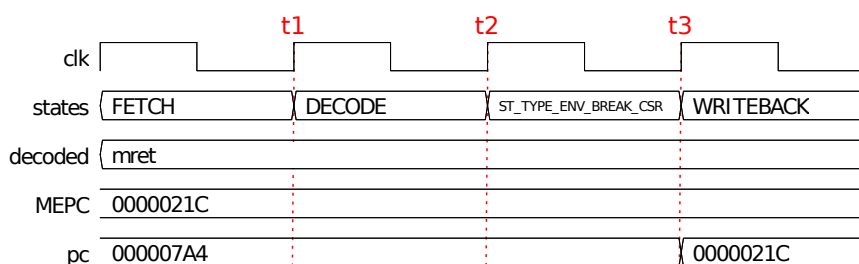
Assim como as instruções da Seção 4.2, a instrução MRET também é executada conforme as etapas presentes na Figura 19. É também considerada uma instrução de sistema e os campos restantes que compõem a instrução MRET são apresentados na Tabela 2.

Tabela 2 – Especificações da instrução MRET

31	20	19	15	14	12	11	7	6	0	
<i>funct12</i>				<i>rs1</i>		<i>funct3</i>		<i>rd</i>		<i>opcode</i>
001100000010				00000		000		00000		1110011
										MRET

Fonte: Adaptado de Waterman et al. (2016b).

Como exposto na Seção 2.2.3, a instrução MRET tem a função de alterar o *program counter* para o valor previamente escrito no registrador de controle e status MEPC. Na Figura 24 é demonstrado um caso de utilização da instrução MRET, onde o valor do campo “pc” é substituído pelo valor do CSR MEPC ao fim da execução da instrução.

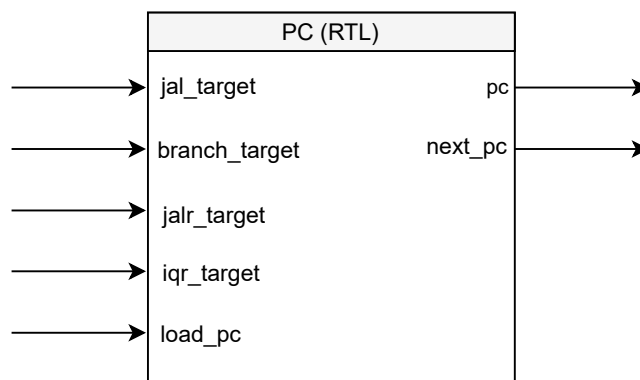
Figura 24 – Diagrama de execução da instrução MRET

Fonte: Elaboração própria.

4.4 Mecanismo de troca de contexto

Como visto na Seção 2.1, para realizar a troca de contexto do núcleo, é necessário alterar o *program counter*. Dessa forma, para que essa operação seja possível, foram adicionadas duas novas funcionalidades na unidade PC. Adicionou-se as entradas “*irq target*” e “*load pc*” encarregadas de alterar o PC, e da saída “*next pc*” responsável por disponibilizar o valor do próximo PC calculado. O componente PC, com as interfaces de entrada e saída, é ilustrado na Figura 25.

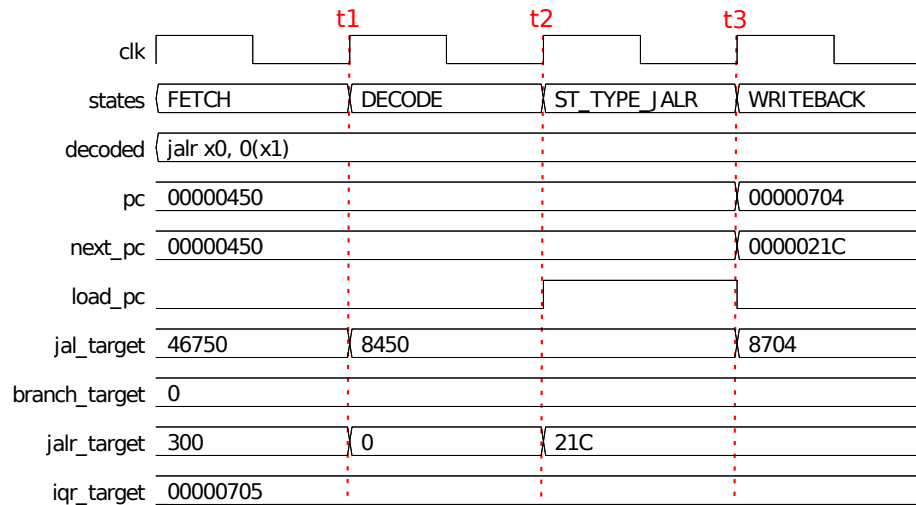
Figura 25 – Representação da unidade PC.



Fonte: Elaboração própria.

Para demonstrar o funcionamento dos recursos adicionados, na Figura 26 apresenta-se um caso onde o núcleo é direcionando o para o tratamento de uma interrupção. No momento em que o sinal “*load pc*” é ativado, o valor do campo “*irq target*” é carregado para o campo “*pc*”.

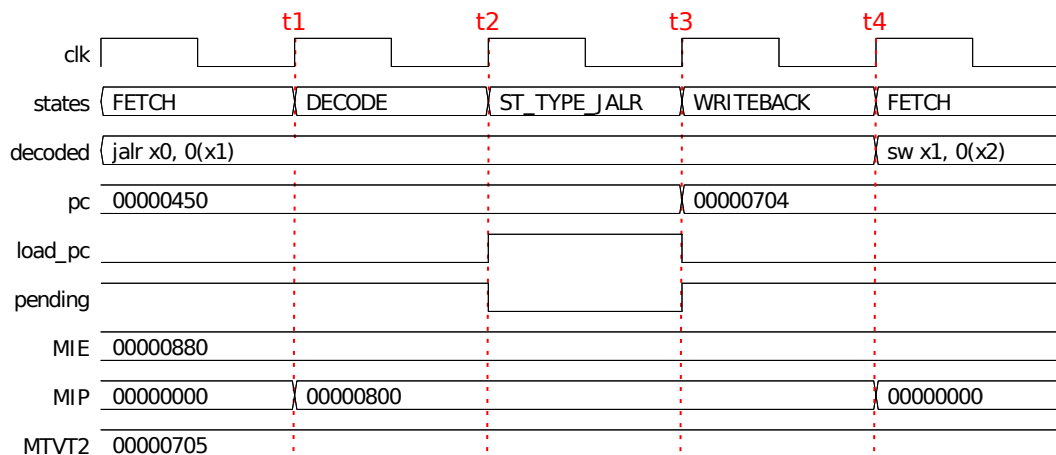
Também é possível visualizar na Figura 26, o campo “*next pc*”, que mantém o valor do PC, que foi calculado pela instrução em execução. Como no exemplo, a instrução é “*jalr*” (*jump and link register*) o valor do “*next pc*” é o calculado por essa instrução, presente no campo “*jalr target*”. Essa informação é essencial para que o programa retorne ao fluxo normal após a ocorrência de uma interrupção.

Figura 26 – Diagrama de execução de uma interrupção no componente PC.

Fonte: Elaboração própria.

Com o mecanismo de troca de contexto implementado na unidade *PC*, é necessário controlá-lo. Essa tarefa é realizada pelo bloco “*load pc control*”, presente no componente *CSR*, conforme a Figura 18. As entradas e saídas do bloco estão representadas na Figura 27, onde é demonstrado um exemplo da ocorrência de uma interrupção.

A unidade identifica a interrupção pelo registrador *MIP*, e caso a classe a qual pertence a interrupção esteja ativa, conforme o registrador *MIE*, e não exista uma instrução pendente (campo “*pending*”), o sinal “*load pc*” é ativado. E como visto anteriormente, essa operação resulta na troca do *PC*, pelo valor do registrador *CSR* adequado, que nesse caso é o *MTVT2*.

Figura 27 – Diagrama de execução do gerenciamento do registrador PC em uma interrupção.

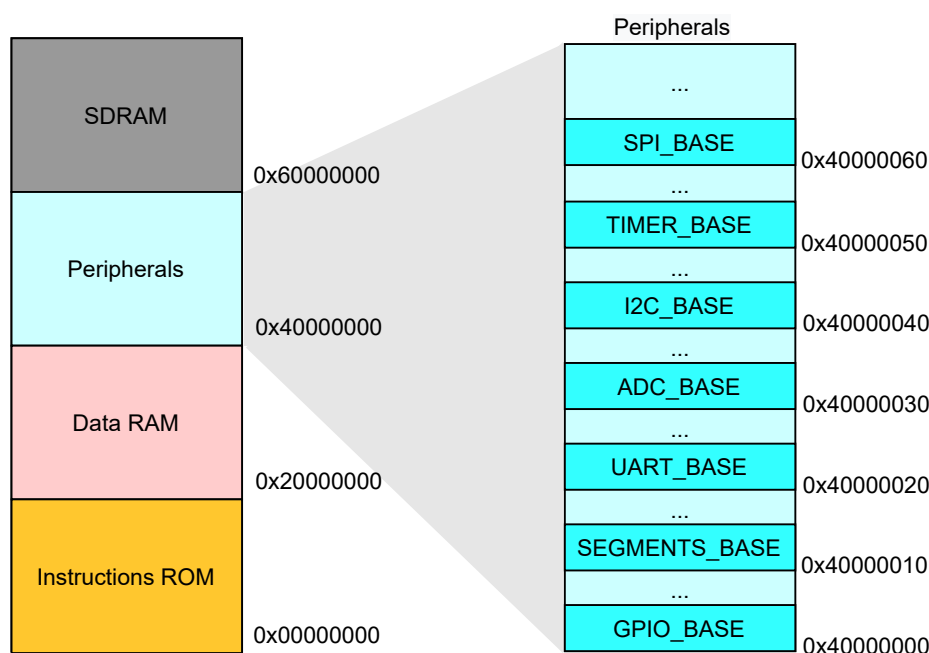
Fonte: Elaboração própria.

4.5 Suporte a interrupções nos periféricos

Implementados os componentes do núcleo relativos as interrupções, é necessário modificar também os periféricos para que estes consigam informar a ocorrência de um evento de interrupção. Dessa forma, essa seção aborda as alterações introduzidas aos periféricos GPIO (*General Purpose Input/Output*) e ao *Timer*.

Antes da implementação das funcionalidades dos periféricos mencionados, foram definidas as regiões de endereço respectivas a cada periférico. Essa alteração visa organizar as futuras implementações de funcionalidades e periféricos, de forma a garantir a compatibilidade entre os módulos no tocante aos endereços. As definições dos endereços base de cada periférico estão apresentados à direita da Figura 28. Também pode ser observado que Starke (2021) segmenta os endereços nas quatro regiões vistas à esquerda da figura, de maneira que os periféricos são acessados pelo barramento principal de dados do núcleo.

Figura 28 – Endereços base dos periféricos.



Fonte: Adaptado de Starke (2021).

4.5.1 Configurações das interrupções no GPIO

Os mecanismos presentes nos CSRs para controle de habilitação de interrupções atuam de forma global ou por classe, não sendo assim possível habilitar somente uma interrupção em específico. Para essa tarefa, foi adicionado o registrador EGII (*Enable General Input Interrupt*) ao periférico *GPIO*, de forma a controlar precisamente qual entrada está habilitada a interromper o processador.

A Figura 29 retrata os campos do registrador EGII, onde foram definidas habilitações individuais para os cinco primeiros pinos de entrada, e agrupadas a habilitação das entradas de índice cinco a nove, e de dez a quinze.

Figura 29 – Enable General Input Interrupt register

31	7	6	5	4	3	2	1	0
WPRI	EGII10_15	EGIIQ5_9	EGIIQ4	EGIIQ3	EGIIQ2	EGIIQ1	EGIIQ0	

Fonte: Elaboração própria.

Foi adicionada também a funcionalidade de seleção de borda para cada interrupção, com essa função sendo configurada pelo registrador GIIIE (*General Input Interrupt Edge*). Caso o campo de GIIIE, referente a interrupção, encontre-se em nível lógico alto, a interrupção ocorre na borda de descida, caso contrário, a interrupção ocorre na borda de subida, sendo o último o comportamento padrão.

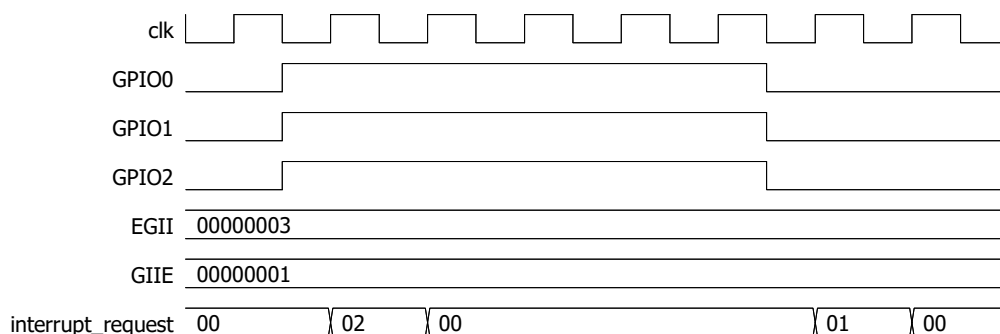
Figura 30 – General Input Interrupt Edge register

31	7	6	5	4	3	2	1	0
WPRI	GIIE10_15	GIIEQ5_9	GIIEQ4	GIIEQ3	GIIEQ2	GIIEQ1	GIIEQ0	

Fonte: Elaboração própria.

Para demonstrar a atuação das funcionalidades adicionadas, é apresentada na Figura 31 três entradas de *GPIO* com a mesma forma de onda, assim sendo possível verificar diferentes casos de uso. Após a borda de subida das entradas de *GPIO*, é gerada uma requisição de interrupção para o GPIO1, pois este é o único que se encontra habilitado em *EGII* e configurado para borda de subida, conforme *GIIE*. Também é gerada uma requisição de interrupção para GPIO0 quando ocorre a borda de descida, visto que está habilitado em *EGII* e com borda de decida selecionada em *GIIE*. Para a entrada GPIO2 não é gerada em nenhum momento uma requisição, pois este se encontra desabilitado em *EGII*.

Figura 31 – Diagrama de execução do periférico *GPIO*.



Fonte: Elaboração própria.

Com a adição dos registradores *EGII* e *GIIE*, os endereços pertencentes ao *GPIO* ficaram dispostos conforme a Figura 32. Os registradores *INPUT* e *OUTPUT* já faziam parte do periférico, sendo responsáveis pelo acionamento e leitura dos pinos, respectivamente.

Figura 32 – Disposição dos endereços dos registradores do periférico *GPIO*.

...	0x4000000F
GIIE	0x40000003
EGII	0x40000002
OUTPUT	0x40000001
INPUT	0x40000000 (<i>GPIO_BASE</i>)

Fonte: Elaboração própria.

4.5.2 Configurações das interrupções de Timer

Assim como para o *GPIO*, foi necessário adicionar ao *Timer* suporte a interrupções, inserindo o registrador *ETI* (*Enable Timer Interrupt*) afim de controlar precisamente quais as interrupções habilitadas, e também foi inserido o mecanismo de detecção de borda, gerando uma requisição sempre na borda de subida do sinal.

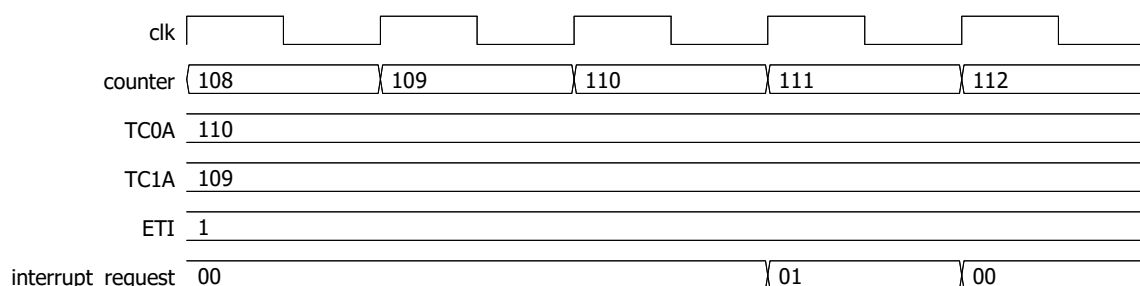
O *Timer* implementado no projeto possui seis comparadores, de forma que foram utilizados os valores de saída desses comparadores para gerar as interrupções. Assim, o registrador *ETI* possui campos respectivos a cada um desses comparadores, conforme a Figura 33.

Figura 33 – Enable Timer Interrupt register

31	6	5	4	3	2	1	0
WPRI	ETIC2B	ETIC2A	ETIC1B	ETIC1A	ETIC0B	ETIC0A	

Fonte: Elaboração própria.

Para demonstrar a utilização do registrador *ETI*, na Figura 34 é apresentado um exemplo. Quando *counter* atinge o valor de *TC0A*, ocorre uma requisição de interrupção. O mesmo não ocorre para *TC1A*, visto que somente *TC0A* está habilitado no registrador *ETI*.

Figura 34 – Diagrama de execução do periférico *Timer*.

Fonte: Elaboração própria.

Com o acréscimo do registradores *ETI*, o *Timer* passou a ter os registradores dispostos conforme a Figura 32. Exceto pelo registrador *ETI*, os registradores presentes na figura já faziam parte do periférico utilizado como base, sendo eles: *RESET*, que tem a função de limpar o contador; *MODE*, que seleciona entre os modos de operação de disparo único, limpar no topo, cíclico ou limpar na comparação; *PRESCALER*, que é valor pelo qual o *clock* do sistema será dividido; *TOP COUNTER*, que se trata do valor máximo de contagem; e *TC0A* a *TC2B*, são os valores dos comparadores (STARKE, 2021).

Figura 35 – Disposição dos endereços dos registradores do periférico *Timer*.

...	0x4000005F
ETIC	0x4000005A
TC2B	0x40000059
TC2A	0x40000058
TC1B	0x40000057
TC1A	0x40000056
TC0B	0x40000055
TC0A	0x40000054
TOP_COUNTER	0x40000053
PRESCALER	0x40000052
MODE	0x40000051
RESET	0x40000050 (TIMER_BASE)

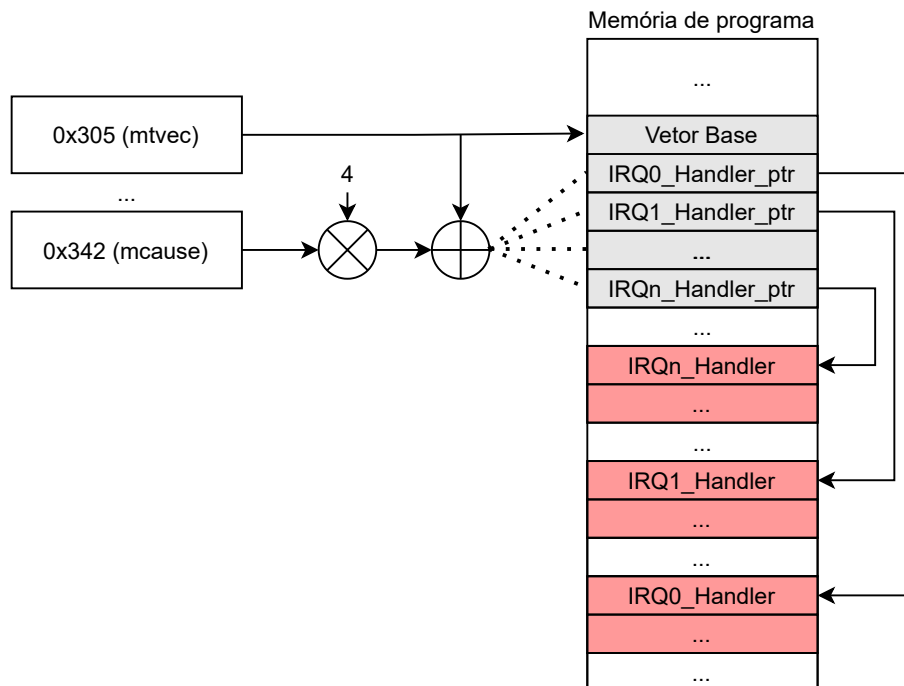
Fonte: Elaboração própria.

4.6 Elementos de *software*

Nas seções anteriores, foram apresentados os mecanismos implementados em *hardware*, porém, para que o sistema ofereça suporte a interrupções é necessário elaborar os componentes de *software* apropriados.

O primeiro componente introduzido foi o vetor base das interrupções, que se trata de um espaço de memória destinado a guardar um ponteiro para o endereço dos tratadores das interrupções. Além de expor essa característica, na Figura 36 é explicitada a relação do vetor base com os registradores *mtvec* e *mcause*. É possível visualizar que o valor do CSR *mcause* é primeiro multiplicado por quatro para se obter o deslocamento do vetor base, que ao ser somado ao valor de *mtvec* retorna o endereço da função responsável por tratar a interrupção.

Figura 36 – Funcionalidade dos registradores *mtvec* e *mcause*.

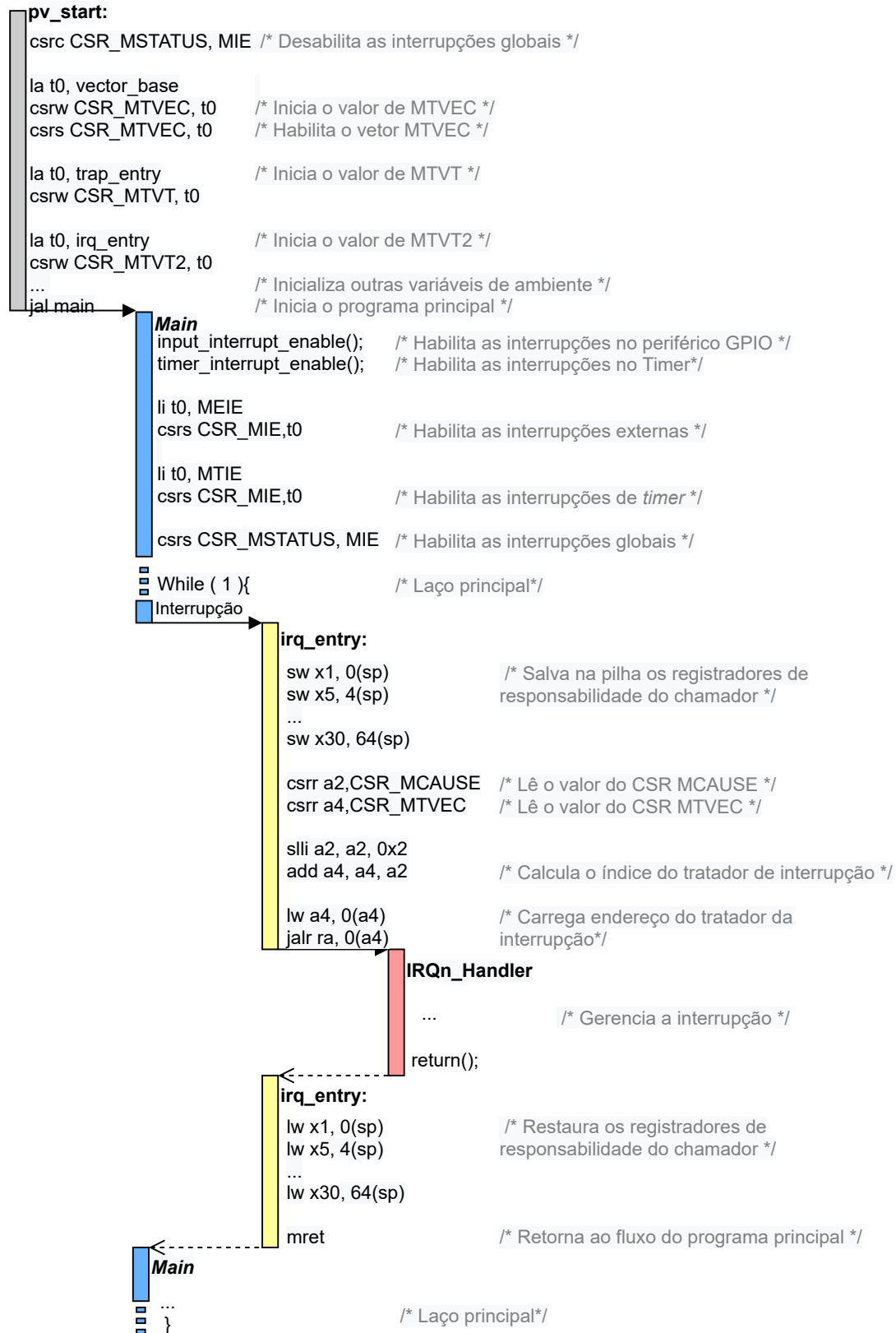


Fonte: Elaboração própria.

Para detalhar o fluxo do programa, já previamente abordado na Seção 2.4, a Figura 37 traz as principais funções do *software*, sendo elas:

- *pv_start*: É o vetor de *reset*, sendo assim o primeiro código a ser executado, também é responsável por inicializar as variáveis de ambiente, como o ponteiro das variáveis globais e o da pilha. Além disso, inicializa no contexto de interrupções os CSRs *mtvec*, *mtvt* e *mtvt2*, além de desabilitar as interrupções globais no *mstatus*, enquanto o restante das variáveis de ambiente não são inicializadas.

- `main`: É o programa principal do usuário, onde são executadas as tarefas sequenciais. Com relação as interrupções, esse código é encarregado de habilitar as interrupções externas e de *timer* no CSR *mie*, e ativar as interrupções globais no CSR *mstatus*. Estando então habilitadas as interrupções, o núcleo trocará de contexto quando ocorrer um evento no periférico.
- `irq_entry`: É o primeiro código executado quando ocorre uma interrupção, também é responsável por gerenciar o estado do processador e invocar o tratador específico de cada interrupção. Essas tarefas são realizadas em três etapas: a primeira é o armazenamento dos registradores do núcleo na memória; a segunda etapa é calcular o endereço do tratador da interrupção pendente e invocá-lo; por fim, a última etapa é restaurar os registradores do núcleo para os valores armazenados na memória.
- `IRQn_Handler`: É o programa do usuário que trata os dados provenientes da interrupção realizando as operações pertinentes a cada periférico.

Figura 37 – Diagrama das atribuições do **software** relacionadas a interrupções.

Fonte: Elaboração própria.

4.7 Integração

Para demonstrar o funcionamento dos componentes apresentados nas seções anteriores, essa seção traz os principais pontos referentes a execução do código contido no Apêndice B, que é o resultado da compilação do programa que executa as funções contidas no código abaixo.

Figura 38 – Código em linguagem C das principais funções do programa de teste.

```
1  void EXTI0_IRQHandler(void){ // Tratador das interrupções de GPIO0
2      OUTBUS = 0x52;
3  }
4
5  void EXTI1_IRQHandler(void){ // Tratador das interrupções de GPIO1
6      OUTBUS = 0xf4;
7  }
8
9  void TIMER0_0A_IRQHandler(void){ // Tratador das interrupções do Comparador 0A
10     OUTBUS = OUTBUS + 1;
11 }
12
13 int main(){
14     input_interrupt_enable(GPIO0,FALLING_EDGE); // Iniciando Interrupções de GPIO0
15     input_interrupt_enable(GPIO1,RISING_EDGE); // Iniciando Interrupções de GPIO1
16     init_timer0 (); // Iniciando o Timer, com interrupções para o comparador 0A
17
18     extern_interrupt_enable(true); // Habilitando interrupções Externas
19     timer_interrupt_enable(true); // Habilitando interrupções de Timer
20     global_interrupt_enable(true); // Habilitando as interrupções Globais
21
22     while (1){ // Laço principal
23         HEX0 = ~timer_get_output0A()<<7;
24     }
25
26     return 0;
27 }
```

Fonte: Elaboração própria.

Na Figura 39 é apresentada a inicialização dos *CSRs* dentro do vetor de reset `pv_start`. Sendo as interrupções globais desabilitadas em `mstatus`, enquanto que `mtvec` recebe o endereço do vetor `vector_base` e sua habilitação. Já o *CSR* `mtvt` é inicializado com o endereço do código comum para exceções `trap_entry`.

Figura 39 – Diagrama de inicialização dos CSRs *mstatus*, *mtvec* e *mtvt*

pc	00000000	0000000C	00000010	0000001C	00000020
decoded	csrrci x8, 0x300, x0	csrrw x5, 0x305, x0	csrrsi x1, 0x305, x0	csrrw x5, 0x307, x0	auipc x5, 0x0
MSTATUS	00000000				
MTVEC	00000000		000007A8	000007A9	
MTVT	00000000				00000640
MTVT2	00000000				

Fonte: Elaboração própria.

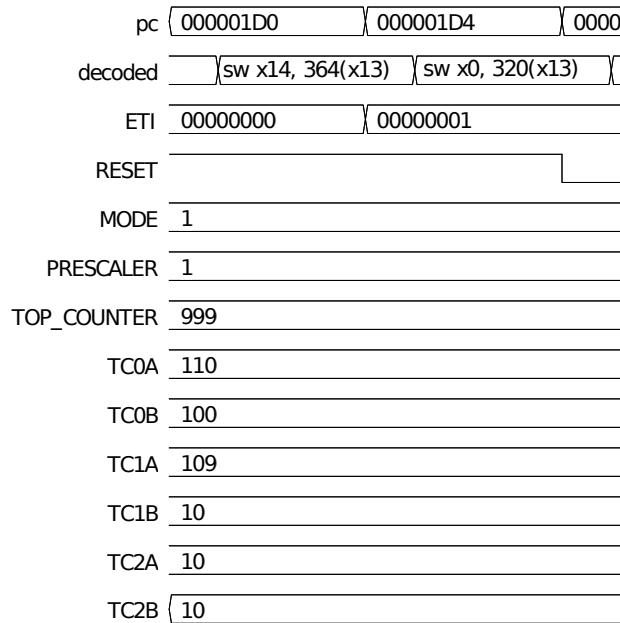
Na sequência, como apresentado na Figura 40 é iniciado o *CSR mvt2*, recebendo o endereço do código comum para interrupções *irq_entry*. Após iniciar as demais variáveis de ambiente, é invocado o programa principal *main*.

Figura 40 – Diagrama de inicialização do *CSR mvt2* e do programa principal.

pc	00000028	00000050	000001DC	
decoded	csrrw x5, 0x7EC, x0	jal x1, 396	addi x2, x2, -16	
MSTATUS	00000000			
MTVEC	000007A9			
MTVT	00000640			
MTVT2	00000000	00000704		

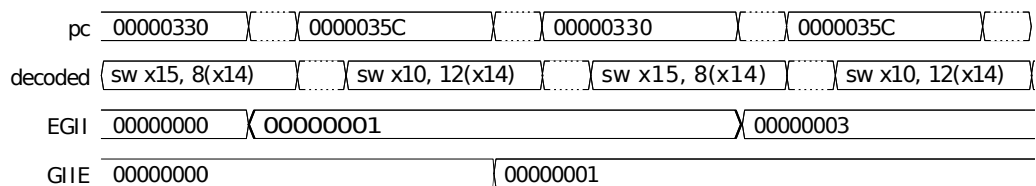
Fonte: Elaboração própria.

Ainda a respeito da inicialização do sistema no exemplo em questão, dentro do programa principal são iniciados o *Timer* e o *GPIO*. Na Figura 41 são apresentadas as configurações utilizadas para o *Timer*. Nela é possível verificar que as interrupções oriundas do comparador *TC0A* foram habilitadas no registrador *ETI*.

Figura 41 – Diagrama de inicialização do *Timer*

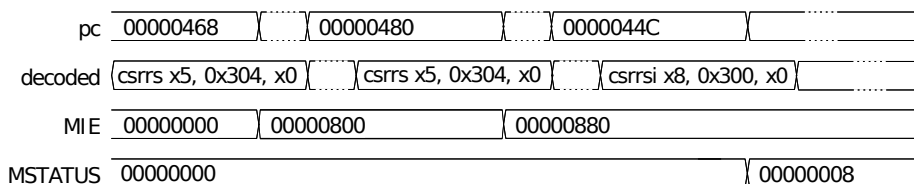
Fonte: Elaboração própria.

Já na Figura 42 são apresentadas as configurações utilizadas para o *GPIO*, onde é possível observar a habilitação das interrupções para o *GPIO0* e *GPIO1* no registrador *EGII*. Também se pode verificar a configuração para borda de subida para *GPIO1* e borda de descida para *GPIO0*, conforme o registrador *GIIE*.

Figura 42 – Diagrama de inicialização do *GPIO*.

Fonte: Elaboração própria.

Em seguida, a configuração do *GPIO* e *Timer*, são habilitadas as interrupções externas e de *Timer* no *CSR mie*. Por fim são habilitadas as interrupções globais no *CSR mstatus*, conforme apresentado na Figura 43.

Figura 43 – Diagrama da habilitação das interrupções globais, de *Timer* e externas

Fonte: Elaboração própria.

Após todas as etapas de inicialização e habilitação das interrupções, é executado o laço principal do programa, com o processador podendo ser interrompido a qualquer momento pelos sinais habilitados. Dessa forma, as subseções a seguir apresentam as etapas executadas na ocorrência de interrupções de *GPIO* e *Timer*.

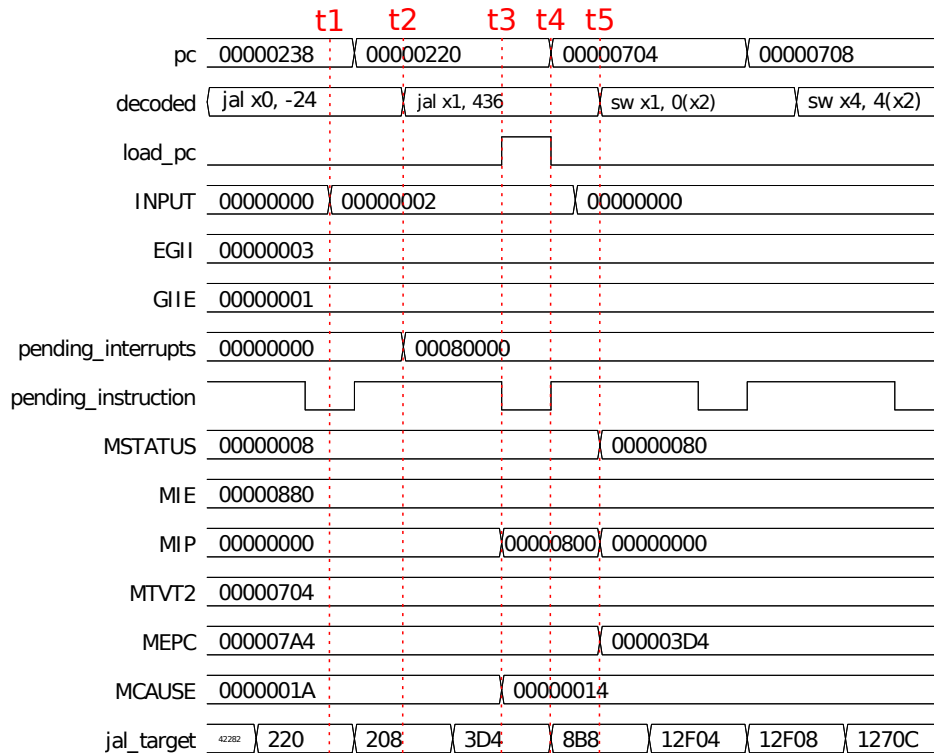
4.7.1 Interrupção de GPIO

Para analisar a troca de contexto no momento em que ocorre uma interrupção relacionada ao *GPIO* a Figura 44 exibe as interações entre os diversos sinais e registradores que gerenciam as interrupções. Para compreender melhor a Figura 44, a mesma foi dividida em 5 diferentes tempos, sendo eles:

- **t1:** É o momento em que ocorre a alteração no *GPIO1*, representada no campo *INPUT*.
- **t2:** Nesse instante são verificados os registradores *EGII* e *GIIE*, que se encontram com as interrupções habilitadas em borda de subida para o *GPIO1*. Com isso o sinal “*pending interrupts*” recebe a requisição da interrupção do *GPIO1*.
- **t3:** Nesse momento os *CSRs mip* e *mcause* são alterados para corresponderem a interrupção em questão. Com o sinal *pending instruction* em nível lógico baixo e o *CSR mip* indicando uma interrupção externa, são verificadas as habilitações das interrupções globais e externas nos *CSRs mstatus* e *mie*, respectivamente. Verificada a habilitação da interrupção, é gerado o sinal *load pc*.
- **t4:** É o instante em que ocorre a troca de contexto, com o núcleo deixando de executar o programa principal e passando a executar o código comum para interrupções. Essa operação é acarretada pelo sinal *load pc* e resulta na troca do *pc* pelo valor armazenado no *CSR mvt2*.
- **t5:** É a última interação relacionada a troca de contexto para o código comum, nesse momento os *CSRs mstatus*, *mip* e *mepc* são atualizados. As interrupções globais ficam desabilitadas em *mstatus*, e a interrupção externa limpa em *mip*,

enquanto o valor da próxima instrução do programa principal é armazenada em *mepc*.

Figura 44 – Diagrama da troca de contexto com interrupção de *GPIO*



Fonte: Elaboração própria.

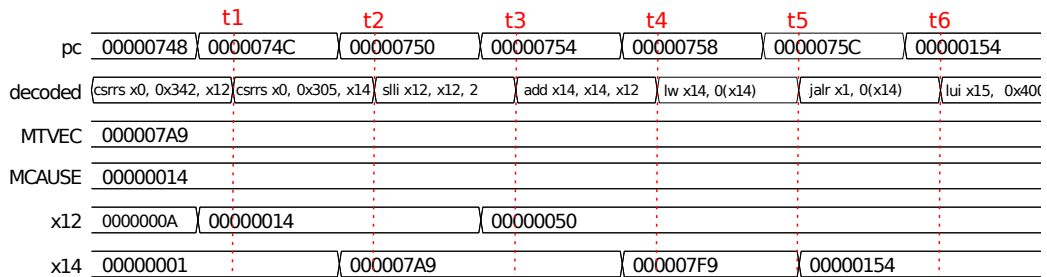
A Figura 44 ilustra também o início do código comum para interrupções, onde os registradores do núcleo são salvos na memória. Após terminado o armazenamento dos registradores, é necessário invocar o manipulador da interrupção de *GPIO1*.

As etapas para calcular o endereço do manipulador da interrupção de *GPIO1*, seguem as operações apresentadas na Figura 36, e são detalhadas na Figura 45, sendo dividida em seis etapas, que são:

- **t1:** O Valor do *CSR mcause* é lido e armazenado no registrador x12.
- **t2:** O Valor do *CSR mtvec* é lido e armazenado no registrador x14.
- **t3:** O Valor do registrador x14 é multiplicado por 4, obtendo o deslocamento do ponteiro do manipulador da interrupção *GPIO1* em relação ao *vector base*.
- **t4:** O Valor dos registradores x12 e x14 são somados, obtendo o endereço do ponteiro para o manipulador da interrupção.

- **t5:** A memória é lida no endereço contido em x14, obtendo o endereço do manipulador da interrupção.
- **t6:** O manipulador da interrupção é chamado executando a rotina que fará a comunicação com o periférico.

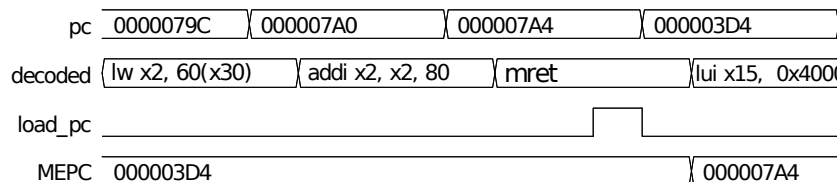
Figura 45 – Diagrama do cálculo do endereço do manipulador da interrupção de *GPIO1*



Fonte: Elaboração própria.

Em seguida, a rotina de interrupção retorna ao código comum para interrupções que restaura os registradores previamente salvos. Por fim, como apresentado na Figura 46, é utilizada a instrução *mret*, que modifica o *pc* para o valor salvo em *mepc* na Figura 45.

Figura 46 – Diagrama de retorno do código comum para interrupções.



Fonte: Elaboração própria.

Executada a instrução *mret* o programa principal volta a ser processado na exata posição em que foi interrompido, dando seguimento a aplicação.

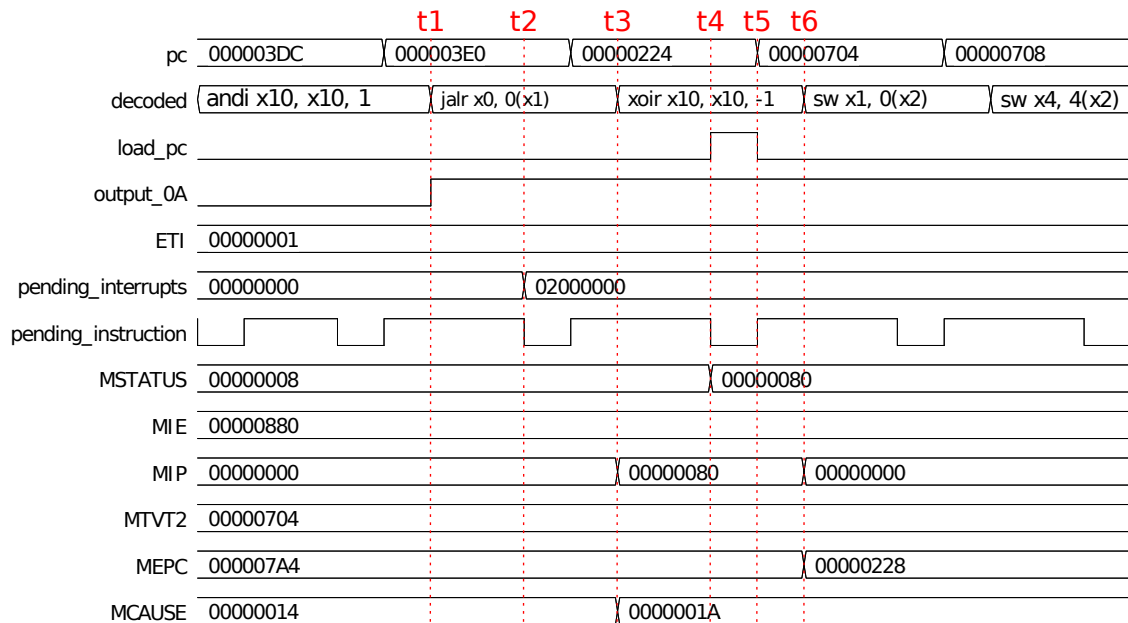
4.7.2 Interrupção de Timer

Da mesma forma que para o *GPIO*, para analisar a troca de contexto quando há um interrupção de, *Timer* na Figura 47 são apresentadas as interações entre os sinais e registradores que controlam as interrupções. Para melhor entendimento da Figura 47, ela foi dividida em 5 diferentes momentos, sendo eles:

- **t1:** É o momento em que ocorre a alteração na saída do comparador TC0A do *Timer*, representada no campo *output 0A*;

- **t2:** Nesse instante é verificado o registrador *ETI*, sendo validada a habilitação da interrupção para o sinal *output 0A*. Com isso o sinal *pending interrupts* recebe a requisição da interrupção do comparador *TC0A*;
- **t3:** Nesse momento os *CSRs mip* e *mcause* são alterados para corresponderem a interrupção do comparador *TC0A*. Dessa forma, com o sinal *pending instruction* em nível lógico baixo e o *CSR mip* indicando uma interrupção externa, são verificadas as habilitações das interrupções globais e de *Timer* nos *CSRs mstatus* e *mie*, respectivamente. Validada a habilitação da interrupção, é gerado o sinal *load pc*;
- **t4:** É o instante a troca de contexto é realizada, com o núcleo passando a executar o código comum para interrupções em detrimento do programa principal. Essa operação é iniciada pelo sinal *load pc*, e acarreta na troca do *pc* pelo valor armazenado no *CSR mvt2*;
- **t5:** É a ultima interação relacionada a troca de contexto para o código comum. Nesse momento os *CSRs mstatus*, *mip* e *mepc* são atualizados. Dessa forma, as interrupções globais são desabilitadas em *mstatus* e a interrupção de *Timer* é limpa em *mip*, enquanto o valor da próxima instrução do programa principal é armazenada em *mepc*;

Figura 47 – Diagrama da troca de contexto com interrupção de *Timer*.



Fonte: Elaboração própria.

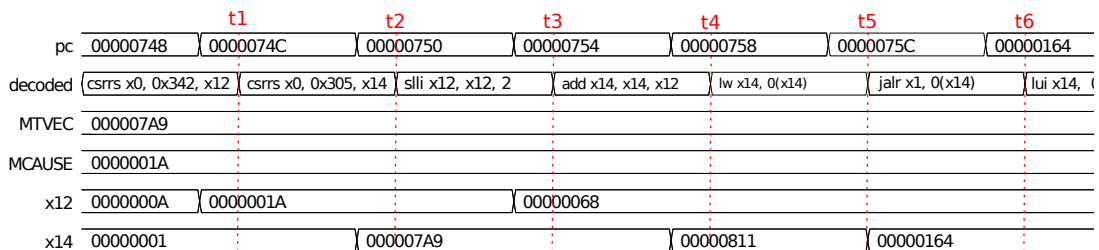
A Figura 47 traz também o início do código comum para interrupções, onde os registradores do núcleo são salvos na memória. Após terminado o armazenamento

dos registradores, é necessário invocar o manipulador da interrupção de *TC0A*, assim como realizado para o *GPIO1* na subseção 4.7.1.

As etapas para calcular o endereço do manipulador da interrupção de *TC0A*, são detalhadas na Figura 45, sendo dividida nas mesmas seis etapas vistas para o *GPIO1*, sendo elas:

- **t1:** O Valor do *CSR mcause* é lido e armazenado no registrador x12.
- **t2:** O Valor do *CSR mtvec* é lido e armazenado no registrador x14.
- **t3:** O Valor do registrador x14 é multiplicado por 4, obtendo o deslocamento do ponteiro do manipulador da interrupção *TC0A* em relação ao *vector base*.
- **t4:** O Valor dos registradores x12 e x14 são somados, obtendo o endereço do ponteiro para o manipulador da interrupção.
- **t5:** A memória é lida no endereço contido em x14, obtendo o endereço do manipulador da interrupção.
- **t6:** O manipulador da interrupção é chamado executando a rotina que tratará a interrupção de *Timer*.

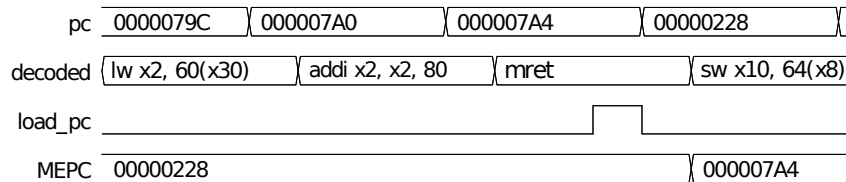
Figura 48 – Diagrama do cálculo do endereço do manipulador da interrupção de *TC0A*.



Fonte: Elaboração própria.

Na sequência, a rotina da interrupção retorna ao código comum que, por sua vez, restaura os registradores anteriormente salvos. Por fim, como apresentado na Figura 49, a instrução *mret* é utilizada para modificar o *pc* para o valor previamente salvo em *mepc* na Figura 48.

Figura 49 – Diagrama de retorno do código comum para interrupção de *Timer*



Fonte: Elaboração própria.

Assim como analisado na Subseção 4.7.1, após a execução da instrução *mret*, o programa principal volta a ser executado normalmente.

O tempo necessário para tratar uma interrupção pode ser calculado da seguinte forma. São 19 instruções para que o tratador de interrupções específico seja chamado e mais 17 instruções para que ocorra o retorno ao fluxo principal. Considerando que cada instrução leva 4 ciclos para ser executada no núcleo multi-ciclo, há no total 144 ciclos adicionais necessários para salvar e retornar do contexto, desconsiderando as instruções do tratador.

4.8 Resultados da síntese

Por fim, após a validação do funcionamento do sistema por meio de simulação, o mesmo foi sintetizado com o *software* Quartus Prime (Intel, 2021). A síntese do projeto com o tratador de interrupções resultou em um total de 8.631 elementos lógicos, perante os 7.724 elementos do projeto base, significando um aumento de 16% para 17% de uso das 49.760 unidades logicas presentes no FPGA (Terasic Inc., 2021) utilizado. O resumo de todos os elementos utilizados se encontram presentes na Tabela 3.

Tabela 3 – Resumo da síntese do *hardware*

Quartus Prime Version	18.1.0 Build 625 09/12/2018 SJ Lite Edition
Revision Name	de10_lite
Top-level Entity Name	de0_lite
Family	MAX 10
Device	10M50DAF484C7G
Timing Models	Final
Total logic elements	8,631 / 49,760 (17 %)
Total registers	1196
Total pins	149 / 360 (41 %)
Total virtual pins	0
Total memory bits	67,584 / 1,677,312 (4 %)
Embedded Multiplier 9-bit elements	16 / 288 (6 %)
Total PLLs	1 / 4 (25 %)
UFM blocks	0 / 1 (0 %)
ADC blocks	0 / 2 (0 %)

Fonte: Gerado a partir de Intel (2021).

Após a síntese, o programa foi gravado no *FPGA*, funcionando como previsto nas simulações e operando com frequência de trabalho de 1MHz. A frequência de 1MHz foi estabelecida por permitir a utilização de osciloscópios e analisadores lógicos de bancada para a depuração dos periféricos. A frequência máxima reportada pela ferramenta de síntese é cerca de 50MHz, sem considerar otimizações.

5 CONSIDERAÇÕES FINAIS

O controle de interrupções é um processo complexo, que exige atuação em diversos componentes do microcontrolador. Assim, baseando-se na especificação da ISA RISC-V, foram estabelecidas as alterações necessárias para a realização do controle de interrupções no projeto base, sendo essas alterações listadas a seguir:

1. Adição dos registradores CSRs;
2. Adição das instruções de acesso aos registradores CSRs;
3. Adição da instrução de retorno de interrupção;
4. Alteração no componente *PC*, possibilitando a troca do valor do contador;
5. Adição do controle de prioridade das interrupções;
6. Alteração nos periféricos, inserindo suporte a interrupções;
7. Adição dos vetores e funções no *software*

Após estabelecer as mudanças necessárias, cada uma foi implementada, verificando e validando seu funcionamento individualmente por meio de simulação. A abordagem de dividir o controle de interrupções em uma série de procedimentos de implementação mais simples, e com validação de cada etapa, resultou em um projeto mais claro e consistente, facilitando a integração do sistema completo.

Com o sistema finalizado, foi possível sintetizar e validar o *hardware* físico com a placa de desenvolvimento (Terasic Inc., 2021), de forma que foram alcançados os objetivos específicos estipulados na Seção 1.4. Com a implementação do tratador de interrupções e sua disponibilização no repositório de Starke (2021), este projeto proporciona aos novos alunos da disciplina de Dispositivos Lógicos-Programáveis, e qualquer outra pessoa interessada, uma plataforma para a implementação de periféricos com interrupções, aprimorando seus conhecimentos acerca da descrição de *hardware* e de microcontroladores.

Também podem ser implementadas novas funcionalidades e topologias para o controle das interrupções. Como sugestão para futuras melhorias, poderia ser implementado o controle de prioridades aninhadas, mecanismos para redução da latência e preempção de interrupções.

REFERÊNCIAS

- GCC. *RISC-V GCC*. 2021. Data de acesso: 31 mar. 2021. Disponível em: <<https://github.com/riscv/riscv-gcc>>. Citado na página 29.
- GIL, A. C. *Como elaborar projetos de pesquisa*. 4. ed. [S.l.]: Editora Atlas S.A., 2002. ISBN 9788522478408, 8522478406. Citado na página 29.
- Intel. *Quartus Prime*. 2021. Data de acesso: 23 mar. 2021. Disponível em: <<https://www.intel.com.br/content/www/br/pt/software/programmable/quartus-prime/overview.html>>. Citado 2 vezes nas páginas 29 e 56.
- JIMÉNEZ, M.; PALOMERA, R.; COUVERTIER, I. *Introduction to Embedded Systems: Using Microcontrollers and the MSP430*. 1. ed. [S.l.]: Springer-Verlag New York, 2014. ISBN 978-1-4614-3142-8, 978-1-4614-3143-5. Citado 2 vezes nas páginas 14 e 15.
- PATTERSON, D.; WATERMAN, A. *The RISC-V Reader*. 1. ed. [S.l.]: Strawberry Canyon, 2017. ISBN 978-0999249109. Citado 3 vezes nas páginas 12, 14 e 16.
- PATTERSON, J. L. H. D. A. *Computer Organization and Design: The Hardware Software Interface [RISC-V Edition]*. 1st. ed. [S.l.]: Morgan Kaufmann, 2017. (The Morgan Kaufmann Series in Computer Architecture and Design). ISBN 0128122757, 9780128122754. Citado 5 vezes nas páginas 12, 17, 18, 19 e 20.
- PEDRONI, V. *Digital Electronics and Design with VHDL*. Elsevier Science, 2008. ISBN 9780080557557. Disponível em: <<https://books.google.com.br/books?id=-ZAccwyQeXMC>>. Citado 2 vezes nas páginas 26 e 29.
- Siemens. *ModelSim*. 2021. Data de acesso: 31 mar. 2021. Disponível em: <<https://eda.sw.siemens.com/en-US/ic/modelsim/>>. Citado na página 29.
- Sigasi. *Sigasi Studio XL*. 2021. Data de acesso: 31 mar. 2021. Disponível em: <<https://www.sigasi.com/>>. Citado na página 29.
- STALLINGS, W. *Arquitetura e organização de computadores*. 8ª. ed. [S.l.]: Pearson Prentice Hall, 2010. ISBN 9788576055648. Citado 2 vezes nas páginas 14 e 15.
- STARKE, R. A. *riscv-multicycle*. [S.l.]: GitHub, 2021. <<https://github.com/xtarke/riscv-multicycle>>. Citado 6 vezes nas páginas 12, 29, 31, 40, 43 e 57.
- Terasic Inc. *DE10-Lite Board*. 2021. Data de acesso: 23 mar. 2021. Disponível em: <<https://www.terasic.com.tw/cgi-bin/page/archive.pl?Language=English&CategoryNo=218&No=1021&PartNo=1>>. Citado 3 vezes nas páginas 29, 56 e 57.
- WATERMAN, A. et al. *The RISC-V Instruction Set Manual Volume II: Privileged Architecture Version 1.9*. [S.l.], 2016. Disponível em: <<http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-129.html>>. Citado 6 vezes nas páginas 16, 20, 21, 22, 23 e 25.
- WATERMAN, A. et al. *The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Version 2.1*. [S.l.], 2016. Disponível em: <<http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-118.html>>. Citado 5 vezes nas páginas 21, 24, 25, 34 e 37.

APÊNDICES

APÊNDICE A – CÓDIGO VHDL DO CONTROLADOR

```

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use work.decoder_types.all;

entity csr is
  port(
    clk      : in std_logic;
    rst      : in std_logic;
    pending_inst : in std_logic;          --! Pending Instruction
    write     : in std_logic;             --! Write CSR
    next_pc   : in std_logic_vector(31 downto 0); --! Next pc generate by current
    Inst
    csr_addr  : in integer;               --! CSR address from imm_i
    csr_new   : in std_logic_vector(31 downto 0); --! New value from rsl_data or imm
    from rsl
    opcodes   : in opcodes_t;             --! Instruction decoding
    information. See decoder_types.vhd
    mret      : in std_logic;             --! Machine trap return
    interrupts : in std_logic_vector(31 downto 0); --! Bit vector of interrupts,
    according with "vector_base:" on start.S

    csr_value : out std_logic_vector(31 downto 0); --! CSR value on current csr_addr
    load_mepc : out std_logic;               --! load machine exeption program
    counter
    mepc_out  : out std_logic_vector(31 downto 0) --! Value of machine exeption
    program counter
  );
end entity csr;

architecture RTL of csr is

  signal mstatus_mask : std_logic_vector(31 downto 0);          -- Used to modify CSR
  MSTATUS between process
  signal pending_interrupts : std_logic_vector(31 downto 0);    -- Register of pending
  interrupts
  signal mip_in : std_logic_vector(31 downto 0);                -- Signal to store on CSR
  MIP
  signal mcause_in : unsigned(31 downto 0);                    -- Signal to store on CSR MCAUSE
  signal load_mepc_reg : std_logic;                             -- Signal to register
  load_mepc

  type machine_reg is array (7 downto 0) of std_logic_vector(31 downto 0);
  signal mreg : machine_reg;                                    -- Vector containing the
  CSRs

  -- Machine Trap Setup registers
  constant MSTATUS : unsigned(15 downto 0) := x"0_300"; -- Machine Status Register
  ( Contains global interrupt-enable )
  constant MIE : unsigned(15 downto 0) := x"1_304"; -- Machine interrupt-enable
  register
  constant MTVEC : unsigned(15 downto 0) := x"2_305"; -- Machine trap-vector base-
  address register
  constant MTVT : unsigned(15 downto 0) := x"3_307"; -- Machine irq-vector base-
  address register

```

```

constant MTVT2      : unsigned(15 downto 0) := x"4_7EC"; -- Machine irq-vector handler
                    entry address

-- Machine Trap Handling
constant MEPC       : unsigned(15 downto 0) := x"5_341"; -- Machine Exception program
                    counter
constant MCAUSE      : unsigned(15 downto 0) := x"6_342"; -- Machine trap cause
constant MIP         : unsigned(15 downto 0) := x"7_344"; -- Machine interrupt-pending
                    register

-- CSR Bits
-- MIP
constant MIP_MSIP    : std_logic_vector(31 downto 0) := x"00000008"; -- Machine
                    Software interrupt-pending
constant MIP_MTIP    : std_logic_vector(31 downto 0) := x"00000080"; -- Machine
                    Timer interrupt-pending
constant MIP_MEIP    : std_logic_vector(31 downto 0) := x"00000800"; -- Machine
                    Extern interrupt-pending
-- MSTATUS
constant MSTATUS_MIE : std_logic_vector(31 downto 0) := X"00000008"; -- Global
                    Machine interrupt enable bit
constant MSTATUS_PIE : std_logic_vector(31 downto 0) := X"00000080"; -- Global
                    Machine interrupt enable bit holder

-- interrupts masks
constant EXTI0_IRQ    : integer := 18;
constant EXTI1_IRQ    : integer := 19;
constant EXTI2_IRQ    : integer := 20;
constant EXTI3_IRQ    : integer := 21;
constant EXTI4_IRQ    : integer := 22;
constant EXTI5_9_IRQ  : integer := 23;
constant EXTI10_15_IRQ : integer := 24;
constant TIMER0_0A_IRQHandler : integer := 25;
constant TIMER0_0B_IRQHandler : integer := 26;
constant TIMER0_1A_IRQHandler : integer := 27;
constant TIMER0_1B_IRQHandler : integer := 28;
constant TIMER0_2A_IRQHandler : integer := 29;
constant TIMER0_2B_IRQHandler : integer := 30;
-- TODO: Add more IRQ Handlers

begin

interrupt_pending_control : process (clk, rst) is
begin
    if rst = '1' then
        pending_interrupts<=x"0000_0000";
        mip_in<=x"0000_0000";
        mcause_in<=x"0000_0000";

    elsif rising_edge(clk) then
        pending_interrupts<=interrupts or pending_interrupts; -- Register interrupts
        if (mret = '1') then -- when mret, the previous pending interrupt is cleared
            pending_interrupts <= pending_interrupts and not std_logic_vector(
to_unsigned(1,32) sll to_integer(unsigned(mcause_in)-1));
        end if;
    end if;
end process;

```

```

        if ((mreg(to_integer(MSTATUS (15 downto 12))) AND MSTATUS_MIE) /= x"00000000")
then -- Check if Global Interrupts are Enabled

        if ((mreg(to_integer(MIE (15 downto 12))) and MIP_MTIP) /= x"0000_0000"
and
        -- Check if Timer Interrupts are Enabled
        (pending_interrupts(TIMER0_2B_IRQHandler downto TIMER0_0A_IRQHandler))
/= "000000" -- Check if Timer IRQs are Pending
        ) then
            mip_in <= MIP_MTIP; -- Set Timer interrupt Pending
            if (pending_interrupts(TIMER0_0A_IRQHandler) /= '0') then
                mcause_in <= to_unsigned(TIMER0_0A_IRQHandler + 1, 32); -- Set
mcause to TIMER0_0A_IRQHandler vector_base address
            elsif (pending_interrupts(TIMER0_0B_IRQHandler) /= '0') then
                mcause_in <= to_unsigned(TIMER0_0B_IRQHandler + 1, 32);
            elsif (pending_interrupts(TIMER0_1A_IRQHandler) /= '0') then
                mcause_in <= to_unsigned(TIMER0_1A_IRQHandler + 1, 32);
            elsif (pending_interrupts(TIMER0_1B_IRQHandler) /= '0') then
                mcause_in <= to_unsigned(TIMER0_1B_IRQHandler + 1, 32);
            elsif (pending_interrupts(TIMER0_2A_IRQHandler) /= '0') then
                mcause_in <= to_unsigned(TIMER0_2A_IRQHandler + 1, 32);
            elsif (pending_interrupts(TIMER0_2B_IRQHandler) /= '0') then
                mcause_in <= to_unsigned(TIMER0_2B_IRQHandler + 1, 32);
            end if;

            elsif ((mreg(to_integer(MIE (15 downto 12))) and MIP_MEIP) /= x"0000_0000
" and
            -- Check if External Interrupts are Enabled
            (pending_interrupts(EXTI0_15_IRQ downto EXTI0_IRQ)) /= "0000000" --
Check if External IRQs are Pending
            ) then
                mip_in <= MIP_MEIP; -- Set External interrupt Pending
                if (pending_interrupts(EXTI0_IRQ) /= '0') then
                    mcause_in <= to_unsigned(EXTI0_IRQ + 1, 32); -- Set mcause to
EXTI0_IRQ vector_base address
                elsif (pending_interrupts(EXTI1_IRQ) /= '0') then
                    mcause_in <= to_unsigned(EXTI1_IRQ + 1, 32);
                elsif (pending_interrupts(EXTI2_IRQ) /= '0') then
                    mcause_in <= to_unsigned(EXTI2_IRQ + 1, 32);
                elsif (pending_interrupts(EXTI3_IRQ) /= '0') then
                    mcause_in <= to_unsigned(EXTI3_IRQ + 1, 32);
                elsif (pending_interrupts(EXTI4_IRQ) /= '0') then
                    mcause_in <= to_unsigned(EXTI4_IRQ + 1, 32);
                elsif (pending_interrupts(EXTI5_9_IRQ) /= '0') then
                    mcause_in <= to_unsigned(EXTI5_9_IRQ + 1, 32);
                elsif (pending_interrupts(EXTI10_15_IRQ) /= '0') then
                    mcause_in <= to_unsigned(EXTI10_15_IRQ + 1, 32);
                end if;
            end if;
        end if;

        if (load_mepc_reg='1' or mret='1' OR (pending_inst='0' and ((mreg(to_integer
(MIP (15 downto 12))) and mreg(to_integer(MIE (15 downto 12)))) /= x"00000000")))
then
            mip_in <= x"0000_0000"; -- Clearing MIP when get in the interrupt
        end if;

    end if;
end process interrupt_pending_control;

```

```

mstatus_control : process (clk, rst) is
begin
    if rst = '1' then
        mstatus_mask <= x"00000000";
    elsif rising_edge(clk) then

        -- Register load_mepc value
        case((mret = '1' OR (pending_inst='0' and ((mreg(To_integer(MIP (15 downto
12)))) and mreg(To_integer(MIE (15 downto 12)))) /= x"00000000")))) is
            when true => load_mepc_reg <= '1';
            when others => load_mepc_reg <= '0';
        end case;

        -- Control mstatus_mask
        if((mreg(To_integer(unsigned(MSTATUS (15 downto 12)))) AND MSTATUS_MIE) /= x"
00000000") then -- test if global interrupts are enabled
            if((mreg(To_integer(unsigned(MIE (15 downto 12)))) AND mreg(To_integer(
unsigned(MIP (15 downto 12)))) /= x"00000000") then -- test if Software|Timer|
Extern interrupts are enabled
                mstatus_mask <= (MSTATUS_MIE or MSTATUS_PIE) xor mstatus_mask; -- Set
MSTATUS_MIE to '0' and MSTATUS_PIE to '1'
            end if;
        elsif (mret='1') then
            mstatus_mask <= (MSTATUS_MIE or MSTATUS_PIE) xor mstatus_mask; -- Set
MSTATUS_MIE to '1' and MSTATUS_PIE to '0'
        else
            mstatus_mask <= x"00000000";
        end if;

    end if;
end process mstatus_control;

registers_control : process (clk, rst) is
    variable mrindex: integer range 0 to 31;
    variable protect_mask: std_logic_vector(31 downto 0); -- Protect read-only bits
on registers
begin
    mrindex:=0;
    protect_mask:=x"00000000";
    if rst = '1' then
        csr_value <= (others=>'0');
        -- Clear all CSR cells
        mreg(0) <= (others => '0');
        mreg(1) <= (others => '0');
        mreg(2) <= (others => '0');
        mreg(3) <= (others => '0');
        mreg(4) <= (others => '0');
        mreg(5) <= (others => '0');
        mreg(6) <= (others => '0');
        mreg(7) <= (others => '0');
    end if;
end process registers_control;

```

```

elsif rising_edge(clk) then
    -- Get index and protect_mask from CSR of csr_addr
    case (to_unsigned(csr_addr,12)) is
        when MSTATUS (11 downto 0) =>
            mrindex:=To_integer(MSTATUS(15 downto 12));
            protect_mask:="01111111100000000001111011000100";
        when MIE      (11 downto 0) =>
            mrindex:=To_integer(MIE      (15 downto 12));
            protect_mask:="11111111111111111010001000100";
        when MTVEC    (11 downto 0) =>
            mrindex:=To_integer(MTVEC    (15 downto 12));
        when MTVT      (11 downto 0) =>
            mrindex:=To_integer(MTVT      (15 downto 12));
        when MTVT2     (11 downto 0) =>
            mrindex:=To_integer(MTVT2     (15 downto 12));
        when MEPC      (11 downto 0) =>
            mrindex:=To_integer(MEPC      (15 downto 12));
            protect_mask:="11111111111111111111111111111111";
        when MCAUSE    (11 downto 0) =>
            mrindex:=To_integer(MCAUSE    (15 downto 12));
            protect_mask:="11111111111111111111111111111111";
        when MIP       (11 downto 0) =>
            mrindex:=To_integer(MIP       (15 downto 12));
            protect_mask:="11111111111111111111111111111111";
        when others =>
            mrindex:=7;
            if(write = '1') then
                report "Not implemented" severity Failure;
            end if;
    end case;

    -- write the CSR value according with instruction
    if (write = '1') then
        case opcodes.funct3(1 downto 0) is
            when "01" => -- CSRRW / CSRRWI
                mreg(mrindex) <= (mreg(mrindex) AND protect_mask) OR (csr_new AND
                (NOT protect_mask));
            when "10" => -- CSRRS / CSRRSI
                mreg(mrindex) <= mreg(mrindex) OR (csr_new AND (NOT protect_mask));
            when "11" => -- CSRRC / CSRRCI
                mreg(mrindex) <= mreg(mrindex) AND (NOT (csr_new AND (NOT
                protect_mask)));
            when others =>
                report "Not implemented" severity Failure;
        end case;
    end if;

    csr_value <= mreg(mrindex); -- Read value from CSR of csr_addr

    mreg(To_integer(MIP      (15 downto 12))) <= mip_in; -- Update MIP
    mreg(To_integer(MCAUSE   (15 downto 12))) <= std_logic_vector(mcause_in); --
    Update MCAUSE

    if (mrindex /= To_integer(MSTATUS(15 downto 12))) then

```

```

        mreg(To_integer(MSTATUS(15 downto 12))) <= mreg(To_integer(MSTATUS (15
downto 12))) xor mstatus_mask; -- Update MSTATUS with mask
    end if;

    if(load_mepc_reg='1') then
        mreg(To_integer(MEPC (15 downto 12)))<=next_pc;    -- Save MEPC when get
in a interrupt
    end if;

    end if;

end process registers_control;

with (mret ='1' OR (pending_inst='0' and ((mreg(To_integer(MIP (15 downto 12))) and
mreg(To_integer(MIE (15 downto 12)))) /=x"00000000"))) select
    load_mepc<= '1' when true, -- Set value of load_mepc
    '0' when others;

with mret select
    mepc_out <= mreg(To_integer(unsigned(MTVT2 (15 downto 12)))) and x"ffffffC"
when '0',
    mreg(To_integer(unsigned(MEPC (15 downto 12)))) when '1',
    x"00000000" when others;

end architecture RTL;

```

csr.vhd

APÊNDICE B – CÓDIGO ASSEMBLY GERADO

```

00000000 <_pvstart>:
 0: 30047073      csrci mstatus,8
 4: 00000297      auipc t0,0x0
 8: 7a428293      addi t0,t0,1956 # 7a8 <vector_base>
c: 30529073      csrw mtvec,t0
10: 3050e073      csrsi mtvec,1
14: 00000297      auipc t0,0x0
18: 62c28293      addi t0,t0,1580 # 640 <trap_entry>
1c: 30729073      csrw mtvt,t0
20: 00000297      auipc t0,0x0
24: 6e428293      addi t0,t0,1764 # 704 <irq_entry>
28: 7ec29073      csrw 0x7ec,t0
2c: 02001137      lui sp,0x2001
30: 80010113      addi sp,sp,-2048 # 2000800 <heap_end.2546+0x384>
34: 020001b7      lui gp,0x2000
38: 00018193      mv gp,gp
3c: ff010113      addi sp,sp,-16
40: 00012023      sw zero,0(sp)
44: 00012223      sw zero,4(sp)
48: 00012423      sw zero,8(sp)
4c: 00012623      sw zero,12(sp)
50: 18c000ef      jal ra,1dc <main>
54: 00100073      ebreak
...

00000100 <_start>:
100: 02000197      auipc gp,0x2000
104: f0018193      addi gp,gp,-256 # 2000000 <__global_pointer$>
108: 43418513      addi a0,gp,1076 # 2000434 <_edata>
10c: 45818613      addi a2,gp,1112 # 2000458 <__lock__atexit_recursive_mutex>
110: 40a60633      sub a2,a2,a0
114: 00000593      li a1,0
118: 378000ef      jal ra,490 <memset>
11c: 00000517      auipc a0,0x0
120: 76450513      addi a0,a0,1892 # 880 <__libc_fini_array>
124: 704000ef      jal ra,828 <atexit>
128: 7b8000ef      jal ra,8e0 <__libc_init_array>
12c: 00012503      lw a0,0(sp)
130: 00410593      addi a1,sp,4
134: 00000613      li a2,0
138: 0a4000ef      jal ra,1dc <main>
13c: 70c0006f      j 848 <exit>

00000140 <_fini>:
140: 00008067      ret

00000144 <EXTI0_IRQHandler>:
144: 040007b7      lui a5,0x4000
148: 05200713      li a4,82
14c: 00e7a223      sw a4,4(a5) # 4000004 <heap_end.2546+0x1ffffb88>
150: 00008067      ret

00000154 <EXTI1_IRQHandler>:
154: 040007b7      lui a5,0x4000
158: 0f400713      li a4,244
15c: 00e7a223      sw a4,4(a5) # 4000004 <heap_end.2546+0x1ffffb88>

```

```

160: 00008067          ret

00000164 <TIMER0_0A_IRQHandler>:
164: 04000737          lui  a4,0x4000
168: 00472783          lw   a5,4(a4) # 4000004 <heap_end.2546+0x1ffffb88>
16c: 00178793          addi a5,a5,1
170: 00f72223          sw   a5,4(a4)
174: 00008067          ret

00000178 <TIMER0_0B_IRQHandler>:
178: 04000737          lui  a4,0x4000
17c: 00472783          lw   a5,4(a4) # 4000004 <heap_end.2546+0x1ffffb88>
180: 00178793          addi a5,a5,1
184: 00f72223          sw   a5,4(a4)
188: 00008067          ret

0000018c <init_timer0>:
18c: 040006b7          lui  a3,0x4000
190: 00100713          li   a4,1
194: 14e6a023          sw   a4,320(a3) # 4000140 <heap_end.2546+0x1ffffcc4>
198: 14e6a223          sw   a4,324(a3)
19c: 14e6a423          sw   a4,328(a3)
1a0: 3e700613          li   a2,999
1a4: 14c6a623          sw   a2,332(a3)
1a8: 06e00613          li   a2,110
1ac: 14c6a823          sw   a2,336(a3)
1b0: 06400613          li   a2,100
1b4: 14c6aa23          sw   a2,340(a3)
1b8: 06d00613          li   a2,109
1bc: 14c6ac23          sw   a2,344(a3)
1c0: 00a00613          li   a2,10
1c4: 14c6ae23          sw   a2,348(a3)
1c8: 16c6a023          sw   a2,352(a3)
1cc: 16c6a223          sw   a2,356(a3)
1d0: 16e6a623          sw   a4,364(a3)
1d4: 1406a023          sw   zero,320(a3)
1d8: 00008067          ret

000001dc <main>:
1dc: ff010113          addi sp,sp,-16
1e0: 00112623          sw   ra,12(sp)
1e4: 00812423          sw   s0,8(sp)
1e8: 00100593          li   a1,1
1ec: 00100513          li   a0,1
1f0: 134000ef          jal  ra,324 <input_interrupt_enable>
1f4: 00000593          li   a1,0
1f8: 00200513          li   a0,2
1fc: 128000ef          jal  ra,324 <input_interrupt_enable>
200: f8dff0ef          jal  ra,18c <init_timer0>
204: 00100513          li   a0,1
208: 254000ef          jal  ra,45c <extern_interrupt_enable>
20c: 00100513          li   a0,1
210: 268000ef          jal  ra,478 <timer_interrupt_enable>
214: 00100513          li   a0,1
218: 230000ef          jal  ra,448 <global_interrupt_enable>
21c: 04000437          lui  s0,0x4000
220: 1b4000ef          jal  ra,3d4 <timer_get_output0A>
224: fff54513          not  a0,a0
228: 04a42023          sw   a0,64(s0) # 4000040 <heap_end.2546+0x1ffffbc4>

```

```

22c: 1b8000ef      jal ra,3e4 <timer_get_output0B>
230: fff54513      not a0,a0
234: 04a42223      sw a0,68(s0)
238: fe9ff06f      j 220 <main+0x44>

0000023c <_access>:
23c: 0040006f      j 240 <unimplemented_syscall>

00000240 <unimplemented_syscall>:
240: 000017b7      lui a5,0x1
244: d0878793      addi a5,a5,-760 # d08 <__FRAME_END__+0x4>
248: 05500713      li a4,85
24c: 100006b7      lui a3,0x10000
250: 00178793      addi a5,a5,1
254: 00e6a023      sw a4,0(a3) # 10000000 <heap_end.2546+0xdffffb84>
258: 0007c703      lbu a4,0(a5)
25c: fe071ae3      bnez a4,250 <unimplemented_syscall+0x10>
260: 00100073      ebreak

00000264 <_read>:
264: 00000513      li a0,0
268: 00008067      ret

0000026c <_write>:
26c: 00c58733      add a4,a1,a2
270: 00e58c63      beq a1,a4,288 <_write+0x1c>
274: 100006b7      lui a3,0x10000
278: 00158593      addi a1,a1,1
27c: fff5c783      lbu a5,-1(a1)
280: 00f6a023      sw a5,0(a3) # 10000000 <heap_end.2546+0xdffffb84>
284: feb71ae3      bne a4,a1,278 <_write+0xc>
288: 00060513      mv a0,a2
28c: 00008067      ret

00000290 <_close>:
290: 00000513      li a0,0
294: 00008067      ret

00000298 <_fstat>:
298: ff010113      addi sp,sp,-16
29c: 00112623      sw ra,12(sp)
2a0: 59c000ef      jal ra,83c <__errno>
2a4: 00200793      li a5,2
2a8: 00f52023      sw a5,0(a0)
2ac: fff00513      li a0,-1
2b0: 00c12083      lw ra,12(sp)
2b4: 01010113      addi sp,sp,16
2b8: 00008067      ret

000002bc <_sbrk>:
2bc: 00050793      mv a5,a0
2c0: 47c1a703      lw a4,1148(gp) # 200047c <heap_end.2546>
2c4: 00071663      bnez a4,2d0 <_sbrk+0x14>
2c8: 45818713      addi a4,gp,1112 # 2000458 <__lock__atexit_recursive_mutex>
2cc: 46e1ae23      sw a4,1148(gp) # 200047c <heap_end.2546>
2d0: 47c1a503      lw a0,1148(gp) # 200047c <heap_end.2546>
2d4: 00a787b3      add a5,a5,a0
2d8: 46f1ae23      sw a5,1148(gp) # 200047c <heap_end.2546>
2dc: 00008067      ret

```

```

000002e0 <_exit>:
 2e0: 00100073          ebreak

000002e4 <init_data_section>:
 2e4: ff010113          addi  sp,sp,-16
 2e8: 00112623          sw   ra,12(sp)
 2ec: 00018613          mv   a2,gp
 2f0: 43418793          addi  a5,gp,1076 # 2000434 <_edata>
 2f4: 40c78633          sub  a2,a5,a2
 2f8: 000015b7          lui  a1,0x1
 2fc: d3058593          addi  a1,a1,-720 # d30 <_data_lma>
 300: 00018513          mv   a0,gp
 304: 674000ef          jal  ra,978 <memcpy>
 308: 00c12083          lw   ra,12(sp)
 30c: 01010113          addi  sp,sp,16
 310: 00008067          ret

00000314 <delay_>:
 314: fff00793          li   a5,-1
 318: fff50513          addi  a0,a0,-1
 31c: fef51ee3          bne  a0,a5,318 <delay_+0x4>
 320: 00008067          ret

00000324 <input_interrupt_enable>:
 324: 04000737          lui  a4,0x4000
 328: 00872783          lw   a5,8(a4) # 4000008 <heap_end.2546+0x1fffb8c>
 32c: 00a7e7b3          or   a5,a5,a0
 330: 00f72423          sw   a5,8(a4)
 334: 00059e63          bnez  a1,350 <input_interrupt_enable+0x2c>
 338: 040007b7          lui  a5,0x4000
 33c: 00c7a703          lw   a4,12(a5) # 400000c <heap_end.2546+0x1fffb90>
 340: fff54513          not  a0,a0
 344: 00e57533          and  a0,a0,a4
 348: 00a7a623          sw   a0,12(a5)
 34c: 00008067          ret
 350: 04000737          lui  a4,0x4000
 354: 00c72783          lw   a5,12(a4) # 400000c <heap_end.2546+0x1fffb90>
 358: 00a7e533          or   a0,a5,a0
 35c: 00a72623          sw   a0,12(a4)
 360: 00008067          ret

00000364 <timer_config>:
 364: 040007b7          lui  a5,0x4000
 368: 14a7a223          sw   a0,324(a5) # 4000144 <heap_end.2546+0x1fffcc8>
 36c: 14b7a423          sw   a1,328(a5)
 370: 14c7a623          sw   a2,332(a5)
 374: 00008067          ret

00000378 <timer_reset>:
 378: 040007b7          lui  a5,0x4000
 37c: 00100713          li   a4,1
 380: 14e7a023          sw   a4,320(a5) # 4000140 <heap_end.2546+0x1fffcc4>
 384: 1407a023          sw   zero,320(a5)
 388: 00008067          ret

0000038c <timer_set_compare0A>:
 38c: 040007b7          lui  a5,0x4000
 390: 14a7a823          sw   a0,336(a5) # 4000150 <heap_end.2546+0x1fffc4>

```

```

394: 00008067          ret

00000398 <timer_set_compare0B>:
398: 040007b7          lui  a5,0x4000
39c: 14a7aa23          sw   a0,340(a5) # 4000154 <heap_end.2546+0x1ffffcd8>
3a0: 00008067          ret

000003a4 <timer_set_compare1A>:
3a4: 040007b7          lui  a5,0x4000
3a8: 14a7ac23          sw   a0,344(a5) # 4000158 <heap_end.2546+0x1ffffcdc>
3ac: 00008067          ret

000003b0 <timer_set_compare1B>:
3b0: 040007b7          lui  a5,0x4000
3b4: 14a7ae23          sw   a0,348(a5) # 400015c <heap_end.2546+0x1ffffce0>
3b8: 00008067          ret

000003bc <timer_set_compare2A>:
3bc: 040007b7          lui  a5,0x4000
3c0: 16a7a023          sw   a0,352(a5) # 4000160 <heap_end.2546+0x1ffffce4>
3c4: 00008067          ret

000003c8 <timer_set_compare2B>:
3c8: 040007b7          lui  a5,0x4000
3cc: 16a7a223          sw   a0,356(a5) # 4000164 <heap_end.2546+0x1ffffce8>
3d0: 00008067          ret

000003d4 <timer_get_output0A>:
3d4: 040007b7          lui  a5,0x4000
3d8: 1687a503          lw   a0,360(a5) # 4000168 <heap_end.2546+0x1ffffcec>
3dc: 00157513          andi a0,a0,1
3e0: 00008067          ret

000003e4 <timer_get_output0B>:
3e4: 040007b7          lui  a5,0x4000
3e8: 1687a503          lw   a0,360(a5) # 4000168 <heap_end.2546+0x1ffffcec>
3ec: 00355513          srli a0,a0,0x3
3f0: 00157513          andi a0,a0,1
3f4: 00008067          ret

000003f8 <timer_get_output1A>:
3f8: 040007b7          lui  a5,0x4000
3fc: 1687a503          lw   a0,360(a5) # 4000168 <heap_end.2546+0x1ffffcec>
400: 00155513          srli a0,a0,0x1
404: 00157513          andi a0,a0,1
408: 00008067          ret

0000040c <timer_get_output1B>:
40c: 040007b7          lui  a5,0x4000
410: 1687a503          lw   a0,360(a5) # 4000168 <heap_end.2546+0x1ffffcec>
414: 00455513          srli a0,a0,0x4
418: 00157513          andi a0,a0,1
41c: 00008067          ret

00000420 <timer_get_output2A>:
420: 040007b7          lui  a5,0x4000
424: 1687a503          lw   a0,360(a5) # 4000168 <heap_end.2546+0x1ffffcec>
428: 00255513          srli a0,a0,0x2
42c: 00157513          andi a0,a0,1

```

```

430: 00008067          ret

00000434 <timer_get_output2B>:
434: 040007b7          lui  a5,0x4000
438: 1687a503          lw   a0,360(a5) # 4000168 <heap_end.2546+0x1fffcec>
43c: 00555513          srli  a0,a0,0x5
440: 00157513          andi  a0,a0,1
444: 00008067          ret

00000448 <global_interrupt_enable>:
448: 00050663          beqz  a0,454 <global_interrupt_enable+0xc>
44c: 30046073          csrsi mstatus,8
450: 00008067          ret
454: 30047073          csrci mstatus,8
458: 00008067          ret

0000045c <extern_interrupt_enable>:
45c: 000012b7          lui  t0,0x1
460: 80028293          addi  t0,t0,-2048 # 800 <vector_base+0x58>
464: 00050663          beqz  a0,470 <extern_interrupt_enable+0x14>
468: 3042a073          csrs  mie,t0
46c: 00008067          ret
470: 3042b073          csrc  mie,t0
474: 00008067          ret

00000478 <timer_interrupt_enable>:
478: 08000293          li   t0,128
47c: 00050663          beqz  a0,488 <timer_interrupt_enable+0x10>
480: 3042a073          csrs  mie,t0
484: 00008067          ret
488: 3042b073          csrc  mie,t0
48c: 00008067          ret

00000490 <memset>:
490: 00f00313          li   t1,15
494: 00050713          mv   a4,a0
498: 02c37e63          bgeu  t1,a2,4d4 <memset+0x44>
49c: 00f77793          andi  a5,a4,15
4a0: 0a079063          bnez  a5,540 <memset+0xb0>
4a4: 08059263          bnez  a1,528 <memset+0x98>
4a8: ff067693          andi  a3,a2,-16
4ac: 00f67613          andi  a2,a2,15
4b0: 00e686b3          add  a3,a3,a4
4b4: 00b72023          sw   a1,0(a4)
4b8: 00b72223          sw   a1,4(a4)
4bc: 00b72423          sw   a1,8(a4)
4c0: 00b72623          sw   a1,12(a4)
4c4: 01070713          addi  a4,a4,16
4c8: fed766e3          bltu  a4,a3,4b4 <memset+0x24>
4cc: 00061463          bnez  a2,4d4 <memset+0x44>
4d0: 00008067          ret
4d4: 40c306b3          sub  a3,t1,a2
4d8: 00269693          slli  a3,a3,0x2
4dc: 00000297          auipc t0,0x0
4e0: 005686b3          add  a3,a3,t0
4e4: 00c68067          jr   12(a3)
4e8: 00b70723          sb   a1,14(a4)
4ec: 00b706a3          sb   a1,13(a4)
4f0: 00b70623          sb   a1,12(a4)

```

```

4f4: 00b705a3      sb  a1,11(a4)
4f8: 00b70523      sb  a1,10(a4)
4fc: 00b704a3      sb  a1,9(a4)
500: 00b70423      sb  a1,8(a4)
504: 00b703a3      sb  a1,7(a4)
508: 00b70323      sb  a1,6(a4)
50c: 00b702a3      sb  a1,5(a4)
510: 00b70223      sb  a1,4(a4)
514: 00b701a3      sb  a1,3(a4)
518: 00b70123      sb  a1,2(a4)
51c: 00b700a3      sb  a1,1(a4)
520: 00b70023      sb  a1,0(a4)
524: 00008067      ret
528: 0ff5f593      andi a1,a1,255
52c: 00859693      slli a3,a1,0x8
530: 00d5e5b3      or  a1,a1,a3
534: 01059693      slli a3,a1,0x10
538: 00d5e5b3      or  a1,a1,a3
53c: f6dff06f      j 4a8 <memset+0x18>
540: 00279693      slli a3,a5,0x2
544: 00000297      auipc t0,0x0
548: 005686b3      add a3,a3,t0
54c: 00008293      mv  t0,ra
550: fa0680e7      jalr -96(a3)
554: 00028093      mv  ra,t0
558: ff078793      addi a5,a5,-16
55c: 40f70733      sub a4,a4,a5
560: 00f60633      add a2,a2,a5
564: f6c378e3      bgeu t1,a2,4d4 <memset+0x44>
568: f3dff06f      j 4a4 <memset+0x14>

```

Disassembly of section .text.deregister_tm_clones:

0000056c <deregister_tm_clones>:

```

56c: 43418513      addi a0,gp,1076 # 2000434 <_edata>
570: 43418793      addi a5,gp,1076 # 2000434 <_edata>
574: 00a78a63      beq a5,a0,588 <deregister_tm_clones+0x1c>
578: 00000317      auipc t1,0x0
57c: a8830313      addi t1,t1,-1400 # 0 <_pvstart>
580: 00030463      beqz t1,588 <deregister_tm_clones+0x1c>
584: 00030067      jr  t1
588: 00008067      ret

```

Disassembly of section .text.register_tm_clones:

0000058c <register_tm_clones>:

```

58c: 43418513      addi a0,gp,1076 # 2000434 <_edata>
590: 43418593      addi a1,gp,1076 # 2000434 <_edata>
594: 40a585b3      sub a1,a1,a0
598: 4025d593      srai a1,a1,0x2
59c: 01f5d793      srli a5,a1,0x1f
5a0: 00b785b3      add a1,a5,a1
5a4: 4015d593      srai a1,a1,0x1
5a8: 00058a63      beqz a1,5bc <register_tm_clones+0x30>
5ac: 00000317      auipc t1,0x0
5b0: a5430313      addi t1,t1,-1452 # 0 <_pvstart>
5b4: 00030463      beqz t1,5bc <register_tm_clones+0x30>
5b8: 00030067      jr  t1
5bc: 00008067      ret

```

Disassembly of section .text.__do_global_dtors_aux:

000005c0 <__do_global_dtors_aux>:

```

5c0: 4791c783      lbu a5,1145(gp) # 2000479 <completed.3272>
5c4: 04079063      bnez a5,604 <__do_global_dtors_aux+0x44>
5c8: ff010113      addi sp,sp,-16
5cc: 00112623      sw ra,12(sp)
5d0: f9dff0ef      jal ra,56c <deregister_tm_clones>
5d4: 00000797      auipc a5,0x0
5d8: a2c78793      addi a5,a5,-1492 # 0 <_pvstart>
5dc: 00078a63      beqz a5,5f0 <__do_global_dtors_aux+0x30>
5e0: 00000517      auipc a0,0x0
5e4: 72450513      addi a0,a0,1828 # d04 <__FRAME_END__>
5e8: 00000097      auipc ra,0x0
5ec: 000000e7      jalr zero # 0 <_pvstart>
5f0: 00c12083      lw ra,12(sp)
5f4: 00100793      li a5,1
5f8: 46f18ca3      sb a5,1145(gp) # 2000479 <completed.3272>
5fc: 01010113      addi sp,sp,16
600: 00008067      ret
604: 00008067      ret

```

Disassembly of section .text.frame_dummy:

00000608 <frame_dummy>:

```

608: 00000797      auipc a5,0x0
60c: 9f878793      addi a5,a5,-1544 # 0 <_pvstart>
610: 02078663      beqz a5,63c <frame_dummy+0x34>
614: ff010113      addi sp,sp,-16
618: 44018593      addi a1,gp,1088 # 2000440 <_bss_start>
61c: 00000517      auipc a0,0x0
620: 6e850513      addi a0,a0,1768 # d04 <__FRAME_END__>
624: 00112623      sw ra,12(sp)
628: 00000097      auipc ra,0x0
62c: 000000e7      jalr zero # 0 <_pvstart>
630: 00c12083      lw ra,12(sp)
634: 01010113      addi sp,sp,16
638: f55ff06f      j 58c <register_tm_clones>
63c: f51ff06f      j 58c <register_tm_clones>

```

Disassembly of section .text.trap:

00000640 <trap_entry>:

```

640: 00112023      sw ra,0(sp)
644: 00412223      sw tp,4(sp)
648: 00512423      sw t0,8(sp)
64c: 00612623      sw t1,12(sp)
650: 00712823      sw t2,16(sp)
654: 00a12a23      sw a0,20(sp)
658: 00b12c23      sw a1,24(sp)
65c: 00c12e23      sw a2,28(sp)
660: 02d12023      sw a3,32(sp)
664: 02e12223      sw a4,36(sp)
668: 02f12423      sw a5,40(sp)
66c: 03012623      sw a6,44(sp)
670: 03112823      sw a7,48(sp)
674: 03c12a23      sw t3,52(sp)
678: 03d12c23      sw t4,56(sp)

```

```

67c: 03e12e23      sw   t5,60(sp)
680: 05f12023      sw   t6,64(sp)
684: 341022f3      csrr  t0,mepc
688: 04512023      sw   t0,64(sp)
68c: 300022f3      csrr  t0,mstatus
690: 04512223      sw   t0,68(sp)
694: 7c4022f3      csrr  t0,0x7c4
698: 04512423      sw   t0,72(sp)
69c: 34202573      csrr  a0,mcause
6a0: 00010593      mv    a1,sp
6a4: 04012283      lw   t0,64(sp)
6a8: 34129073      csrw  mepc,t0
6ac: 04412283      lw   t0,68(sp)
6b0: 30029073      csrw  mstatus,t0
6b4: 04812283      lw   t0,72(sp)
6b8: 7c429073      csrw  0x7c4,t0
6bc: 00012083      lw   ra,0(sp)
6c0: 00412203      lw   tp,4(sp)
6c4: 00812283      lw   t0,8(sp)
6c8: 00c12303      lw   t1,12(sp)
6cc: 01012383      lw   t2,16(sp)
6d0: 01412503      lw   a0,20(sp)
6d4: 01812583      lw   a1,24(sp)
6d8: 01c12603      lw   a2,28(sp)
6dc: 02012683      lw   a3,32(sp)
6e0: 02412703      lw   a4,36(sp)
6e4: 02812783      lw   a5,40(sp)
6e8: 02c12803      lw   a6,44(sp)
6ec: 03012883      lw   a7,48(sp)
6f0: 03412e03      lw   t3,52(sp)
6f4: 03812e83      lw   t4,56(sp)
6f8: 03c12f03      lw   t5,60(sp)
6fc: 05010113      addi sp,sp,80
700: 30200073      mret

```

Disassembly of section .text.irq:

00000704 <irq_entry>:

```

704: 00112023      sw   ra,0(sp)
708: 00412223      sw   tp,4(sp)
70c: 00512423      sw   t0,8(sp)
710: 00612623      sw   t1,12(sp)
714: 00712823      sw   t2,16(sp)
718: 00a12a23      sw   a0,20(sp)
71c: 00b12c23      sw   a1,24(sp)
720: 00c12e23      sw   a2,28(sp)
724: 02d12023      sw   a3,32(sp)
728: 02e12223      sw   a4,36(sp)
72c: 02f12423      sw   a5,40(sp)
730: 03012623      sw   a6,44(sp)
734: 03112823      sw   a7,48(sp)
738: 03c12a23      sw   t3,52(sp)
73c: 03d12c23      sw   t4,56(sp)
740: 03e12e23      sw   t5,60(sp)
744: 05f12023      sw   t6,64(sp)
748: 34202673      csrr  a2,mcause
74c: 30502773      csrr  a4,mtvec
750: 00261613      slli  a2,a2,0x2
754: 00c70733      add  a4,a4,a2

```

```

758: 00072703      lw  a4,0(a4)
75c: 000700e7      jalr a4
760: 00012083      lw  ra,0(sp)
764: 00412203      lw  tp,4(sp)
768: 00812283      lw  t0,8(sp)
76c: 00c12303      lw  t1,12(sp)
770: 01012383      lw  t2,16(sp)
774: 01412503      lw  a0,20(sp)
778: 01812583      lw  a1,24(sp)
77c: 01c12603      lw  a2,28(sp)
780: 02012683      lw  a3,32(sp)
784: 02412703      lw  a4,36(sp)
788: 02812783      lw  a5,40(sp)
78c: 02c12803      lw  a6,44(sp)
790: 03012883      lw  a7,48(sp)
794: 03412e03      lw  t3,52(sp)
798: 03812e83      lw  t4,56(sp)
79c: 03c12f03      lw  t5,60(sp)
7a0: 05010113      addi sp,sp,80
7a4: 30200073      mret

```

Disassembly of section .init:

000007a8 <vector_base>:

```

7a8: 959ff06f      j 100 <_start>
...
7f4: 0144          addi  s1,sp,132
7f6: 0000          unimp
7f8: 0154          addi  a3,sp,132
...
80e: 0000          unimp
810: 0164          addi  s1,sp,140
812: 0000          unimp
814: 0178          addi  a4,sp,140
...

```

Disassembly of section .text.atexit:

00000828 <atexit>:

```

828: 00050593      mv   a1,a0
82c: 00000693      li   a3,0
830: 00000613      li   a2,0
834: 00000513      li   a0,0
838: 25c0006f      j    a94 <__register_exitproc>

```

Disassembly of section .text.__errno:

0000083c <__errno>:

```

83c: 42c18793      addi  a5,gp,1068 # 200042c <_impure_ptr>
840: 0007a503      lw    a0,0(a5)
844: 00008067      ret

```

Disassembly of section .text.exit:

00000848 <exit>:

```

848: ff010113      addi  sp,sp,-16
84c: 00000593      li    a1,0
850: 00812423      sw    s0,8(sp)
854: 00112623      sw    ra,12(sp)

```

```

858: 00050413      mv  s0,a0
85c: 340000ef      jal ra,b9c <__call_exitprocs>
860: 00000797      auipc a5,0x0
864: 4cc78793      addi a5,a5,1228 # d2c <_global_impure_ptr>
868: 0007a503      lw  a0,0(a5)
86c: 03c52783      lw  a5,60(a0)
870: 00078463      beqz a5,878 <exit+0x30>
874: 000780e7      jalr a5
878: 00040513      mv  a0,s0
87c: a65ff0ef      jal ra,2e0 <_exit>

```

Disassembly of section .text.__libc_fini_array:

00000880 <__libc_fini_array>:

```

880: ff010113      addi sp,sp,-16
884: 00812423      sw  s0,8(sp)
888: fffff797      auipc a5,0xfffff
88c: 77878793      addi a5,a5,1912 # 0 <_pvstart>
890: fffff417      auipc s0,0xfffff
894: 77040413      addi s0,s0,1904 # 0 <_pvstart>
898: 40f40433      sub s0,s0,a5
89c: 00112623      sw  ra,12(sp)
8a0: 00912223      sw  s1,4(sp)
8a4: 40245413      srai s0,s0,0x2
8a8: 02040263      beqz s0,8cc <__libc_fini_array+0x4c>
8ac: 00241493      slli s1,s0,0x2
8b0: ffc48493      addi s1,s1,-4
8b4: 00f484b3      add s1,s1,a5
8b8: 0004a783      lw  a5,0(s1)
8bc: fff40413      addi s0,s0,-1
8c0: ffc48493      addi s1,s1,-4
8c4: 000780e7      jalr a5
8c8: fe0418e3      bnez s0,8b8 <__libc_fini_array+0x38>
8cc: 00812403      lw  s0,8(sp)
8d0: 00c12083      lw  ra,12(sp)
8d4: 00412483      lw  s1,4(sp)
8d8: 01010113      addi sp,sp,16
8dc: 865ff06f      j 140 <_fini>

```

Disassembly of section .text.__libc_init_array:

000008e0 <__libc_init_array>:

```

8e0: ff010113      addi sp,sp,-16
8e4: 00812423      sw  s0,8(sp)
8e8: 01212023      sw  s2,0(sp)
8ec: fffff417      auipc s0,0xfffff
8f0: 71440413      addi s0,s0,1812 # 0 <_pvstart>
8f4: fffff917      auipc s2,0xfffff
8f8: 70c90913      addi s2,s2,1804 # 0 <_pvstart>
8fc: 40890933      sub s2,s2,s0
900: 00112623      sw  ra,12(sp)
904: 00912223      sw  s1,4(sp)
908: 40295913      srai s2,s2,0x2
90c: 00090e63      beqz s2,928 <__libc_init_array+0x48>
910: 00000493      li  s1,0
914: 00042783      lw  a5,0(s0)
918: 00148493      addi s1,s1,1
91c: 00440413      addi s0,s0,4
920: 000780e7      jalr a5

```

```

924: fe9918e3      bne s2,s1,914 <__libc_init_array+0x34>
928: fffff417      auipc s0,0xfffff
92c: 6d840413      addi s0,s0,1752 # 0 <_pvstart>
930: fffff917      auipc s2,0xfffff
934: 6d090913      addi s2,s2,1744 # 0 <_pvstart>
938: 40890933      sub s2,s2,s0
93c: 40295913      srai s2,s2,0x2
940: 801ff0ef      jal ra,140 <_fini>
944: 00090e63      beqz s2,960 <__libc_init_array+0x80>
948: 00000493      li s1,0
94c: 00042783      lw a5,0(s0)
950: 00148493      addi s1,s1,1
954: 00440413      addi s0,s0,4
958: 000780e7      jalr a5
95c: fe9918e3      bne s2,s1,94c <__libc_init_array+0x6c>
960: 00c12083      lw ra,12(sp)
964: 00812403      lw s0,8(sp)
968: 00412483      lw s1,4(sp)
96c: 00012903      lw s2,0(sp)
970: 01010113      addi sp,sp,16
974: 00008067      ret

```

Disassembly of section .text.memcpy:

00000978 <memcpy>:

```

978: 00a5c7b3      xor a5,a1,a0
97c: 0037f793      andi a5,a5,3
980: 00c508b3      add a7,a0,a2
984: 06079263      bnez a5,9e8 <memcpy+0x70>
988: 00300793      li a5,3
98c: 04c7fe63      bgeu a5,a2,9e8 <memcpy+0x70>
990: 00357793      andi a5,a0,3
994: 00050713      mv a4,a0
998: 06079863      bnez a5,a08 <memcpy+0x90>
99c: ffc8f613      andi a2,a7,-4
9a0: fe060793      addi a5,a2,-32
9a4: 08f76c63      bltu a4,a5,a3c <memcpy+0xc4>
9a8: 02c77c63      bgeu a4,a2,9e0 <memcpy+0x68>
9ac: 00058693      mv a3,a1
9b0: 00070793      mv a5,a4
9b4: 0006a803      lw a6,0(a3)
9b8: 00478793      addi a5,a5,4
9bc: 00468693      addi a3,a3,4
9c0: ff07ae23      sw a6,-4(a5)
9c4: fec7e8e3      bltu a5,a2,9b4 <memcpy+0x3c>
9c8: fff60793      addi a5,a2,-1
9cc: 40e787b3      sub a5,a5,a4
9d0: ffc7f793      andi a5,a5,-4
9d4: 00478793      addi a5,a5,4
9d8: 00f70733      add a4,a4,a5
9dc: 00f585b3      add a1,a1,a5
9e0: 01176863      bltu a4,a7,9f0 <memcpy+0x78>
9e4: 00008067      ret
9e8: 00050713      mv a4,a0
9ec: ff157ce3      bgeu a0,a7,9e4 <memcpy+0x6c>
9f0: 0005c783      lbu a5,0(a1)
9f4: 00170713      addi a4,a4,1
9f8: 00158593      addi a1,a1,1
9fc: fef70fa3      sb a5,-1(a4)

```

```

a00: ff1768e3      bltu  a4,a7,9f0 <memcpy+0x78>
a04: 00008067      ret
a08: 0005c683      lbu   a3,0(a1)
a0c: 00170713      addi  a4,a4,1
a10: 00377793      andi  a5,a4,3
a14: fed70fa3      sb    a3,-1(a4)
a18: 00158593      addi  a1,a1,1
a1c: f80780e3      beqz  a5,99c <memcpy+0x24>
a20: 0005c683      lbu   a3,0(a1)
a24: 00170713      addi  a4,a4,1
a28: 00377793      andi  a5,a4,3
a2c: fed70fa3      sb    a3,-1(a4)
a30: 00158593      addi  a1,a1,1
a34: fc079ae3      bnez  a5,a08 <memcpy+0x90>
a38: f65ff06f      j     99c <memcpy+0x24>
a3c: 0005a683      lw    a3,0(a1)
a40: 0045a283      lw    t0,4(a1)
a44: 0085af83      lw    t6,8(a1)
a48: 00c5af03      lw    t5,12(a1)
a4c: 0105ae83      lw    t4,16(a1)
a50: 0145ae03      lw    t3,20(a1)
a54: 0185a303      lw    t1,24(a1)
a58: 01c5a803      lw    a6,28(a1)
a5c: 02458593      addi  a1,a1,36
a60: 00d72023      sw    a3,0(a4)
a64: ffc5a683      lw    a3,-4(a1)
a68: 00572223      sw    t0,4(a4)
a6c: 01f72423      sw    t6,8(a4)
a70: 01e72623      sw    t5,12(a4)
a74: 01d72823      sw    t4,16(a4)
a78: 01c72a23      sw    t3,20(a4)
a7c: 00672c23      sw    t1,24(a4)
a80: 01072e23      sw    a6,28(a4)
a84: 02470713      addi  a4,a4,36
a88: fed72e23      sw    a3,-4(a4)
a8c: faf768e3      bltu  a4,a5,a3c <memcpy+0xc4>
a90: f19ff06f      j     9a8 <memcpy+0x30>

```

Disassembly of section .text.__register_exitproc:

00000a94 <__register_exitproc>:

```

a94: fe010113      addi  sp,sp,-32
a98: 00812c23      sw    s0,24(sp)
a9c: 43018413      addi  s0,gp,1072 # 2000430 <__atexit_recursive_mutex>
aa0: 00912a23      sw    s1,20(sp)
aa4: 00050493      mv    s1,a0
aa8: 00042503      lw    a0,0(s0)
aac: 01212823      sw    s2,16(sp)
ab0: 01312623      sw    s3,12(sp)
ab4: 01412423      sw    s4,8(sp)
ab8: 00112e23      sw    ra,28(sp)
abc: 00058913      mv    s2,a1
ac0: 00060a13      mv    s4,a2
ac4: 00068993      mv    s3,a3
ac8: 220000ef      jal  ra,ce8 <__retarget_lock_acquire_recursive>
acc: 00000797      auipc a5,0x0
ad0: 26078793      addi  a5,a5,608 # d2c <_global_impure_ptr>
ad4: 0007a703      lw    a4,0(a5)
ad8: 14872783      lw    a5,328(a4)

```

```

adc: 08078663      beqz  a5,b68 <__register_exitproc+0xd4>
ae0: 0047a703      lw    a4,4(a5)
ae4: 01f00813      li    a6,31
ae8: 00042503      lw    a0,0(s0)
aec: 08e84463      blt   a6,a4,b74 <__register_exitproc+0xe0>
af0: 00271813      slli  a6,a4,0x2
af4: 02049e63      bnez  s1,b30 <__register_exitproc+0x9c>
af8: 00170713      addi  a4,a4,1
afc: 00e7a223      sw    a4,4(a5)
b00: 010787b3      add   a5,a5,a6
b04: 0127a423      sw    s2,8(a5)
b08: 1f8000ef      jal   ra,d00 <__retarget_lock_release_recursive>
b0c: 00000513      li    a0,0
b10: 01c12083      lw    ra,28(sp)
b14: 01812403      lw    s0,24(sp)
b18: 01412483      lw    s1,20(sp)
b1c: 01012903      lw    s2,16(sp)
b20: 00c12983      lw    s3,12(sp)
b24: 00812a03      lw    s4,8(sp)
b28: 02010113      addi  sp,sp,32
b2c: 00008067      ret
b30: 010785b3      add   a1,a5,a6
b34: 0945a423      sw    s4,136(a1)
b38: 1887a683      lw    a3,392(a5)
b3c: 00100613      li    a2,1
b40: 00e61633      sll   a2,a2,a4
b44: 00c6e6b3      or    a3,a3,a2
b48: 18d7a423      sw    a3,392(a5)
b4c: 1135a423      sw    s3,264(a1)
b50: 00200693      li    a3,2
b54: fad492e3      bne   s1,a3,af8 <__register_exitproc+0x64>
b58: 18c7a683      lw    a3,396(a5)
b5c: 00c6e633      or    a2,a3,a2
b60: 18c7a623      sw    a2,396(a5)
b64: f95ff06f      j     af8 <__register_exitproc+0x64>
b68: 14c70793      addi  a5,a4,332
b6c: 14f72423      sw    a5,328(a4)
b70: f71ff06f      j     ae0 <__register_exitproc+0x4c>
b74: 18c000ef      jal   ra,d00 <__retarget_lock_release_recursive>
b78: fff00513      li    a0,-1
b7c: f95ff06f      j     b10 <__register_exitproc+0x7c>

```

Disassembly of section .text.startup.register_fini:

00000b80 <register_fini>:

```

b80: fffff797      auipc a5,0xfffff
b84: 48078793      addi  a5,a5,1152 # 0 <_pvstart>
b88: 00078863      beqz  a5,b98 <register_fini+0x18>
b8c: 00000517      auipc a0,0x0
b90: cf450513      addi  a0,a0,-780 # 880 <__libc_fini_array>
b94: c95ff06f      j     828 <atexit>
b98: 00008067      ret

```

Disassembly of section .text.__call_exitprocs:

00000b9c <__call_exitprocs>:

```

b9c: fd010113      addi  sp,sp,-48
ba0: 01612823      sw    s6,16(sp)
ba4: 43018b13      addi  s6,gp,1072 # 2000430 <__atexit_recursive_mutex>

```

```

ba8: 01512a23      sw  s5,20(sp)
bac: 00050a93      mv  s5,a0
bb0: 000b2503      lw  a0,0(s6)
bb4: 01312e23      sw  s3,28(sp)
bb8: 01412c23      sw  s4,24(sp)
bbc: 01712623      sw  s7,12(sp)
bc0: 01912223      sw  s9,4(sp)
bc4: 02112623      sw  ra,44(sp)
bc8: 02812423      sw  s0,40(sp)
bcc: 02912223      sw  s1,36(sp)
bd0: 03212023      sw  s2,32(sp)
bd4: 01812423      sw  s8,8(sp)
bd8: 00058b93      mv  s7,a1
bdc: 10c000ef      jal ra,ce8 <__retarget_lock_acquire_recursive>
be0: 00000797      auipc a5,0x0
be4: 14c78793      addi a5,a5,332 # d2c <_global_impure_ptr>
be8: 0007ac83      lw  s9,0(a5)
bec: 00100a13      li  s4,1
bf0: fff00993      li  s3,-1
bf4: 148ca903      lw  s2,328(s9)
bf8: 02090863      beqz s2,c28 <__call_exitprocs+0x8c>
bfc: 00492483      lw  s1,4(s2)
c00: fff48413      addi s0,s1,-1
c04: 02044263      bltz s0,c28 <__call_exitprocs+0x8c>
c08: 00249493      slli s1,s1,0x2
c0c: 009904b3      add s1,s2,s1
c10: 040b8863      beqz s7,c60 <__call_exitprocs+0xc4>
c14: 1044a783      lw  a5,260(s1)
c18: 05778463      beq a5,s7,c60 <__call_exitprocs+0xc4>
c1c: fff40413      addi s0,s0,-1
c20: ffc48493      addi s1,s1,-4
c24: ff3416e3      bne s0,s3,c10 <__call_exitprocs+0x74>
c28: 02812403      lw  s0,40(sp)
c2c: 000b2503      lw  a0,0(s6)
c30: 02c12083      lw  ra,44(sp)
c34: 02412483      lw  s1,36(sp)
c38: 02012903      lw  s2,32(sp)
c3c: 01c12983      lw  s3,28(sp)
c40: 01812a03      lw  s4,24(sp)
c44: 01412a83      lw  s5,20(sp)
c48: 01012b03      lw  s6,16(sp)
c4c: 00c12b83      lw  s7,12(sp)
c50: 00812c03      lw  s8,8(sp)
c54: 00412c83      lw  s9,4(sp)
c58: 03010113      addi sp,sp,48
c5c: 0a40006f      j d00 <__retarget_lock_release_recursive>
c60: 00492783      lw  a5,4(s2)
c64: 0044a683      lw  a3,4(s1)
c68: fff78793      addi a5,a5,-1
c6c: 04878a63      beq a5,s0,cc0 <__call_exitprocs+0x124>
c70: 0004a223      sw  zero,4(s1)
c74: fa0684e3      beqz a3,c1c <__call_exitprocs+0x80>
c78: 18892783      lw  a5,392(s2)
c7c: 008a1733      sll a4,s4,s0
c80: 00492c03      lw  s8,4(s2)
c84: 00f777b3      and a5,a4,a5
c88: 00079e63      bnez a5,ca4 <__call_exitprocs+0x108>
c8c: 000680e7      jalr a3
c90: 00492783      lw  a5,4(s2)

```

```

c94: f78790e3      bne a5,s8,bf4 <__call_exitprocs+0x58>
c98: 148ca783      lw  a5,328(s9)
c9c: f92780e3      beq a5,s2,clc <__call_exitprocs+0x80>
ca0: f55ff06f      j  bf4 <__call_exitprocs+0x58>
ca4: 18c92783      lw  a5,396(s2)
ca8: 0844a583      lw  a1,132(s1)
cac: 00f77733      and a4,a4,a5
cb0: 00071c63      bnez a4,cc8 <__call_exitprocs+0x12c>
cb4: 000a8513      mv  a0,s5
cb8: 000680e7      jalr a3
cbc: fd5ff06f      j  c90 <__call_exitprocs+0xf4>
cc0: 00892223      sw  s0,4(s2)
cc4: fb1ff06f      j  c74 <__call_exitprocs+0xd8>
cc8: 00058513      mv  a0,a1
ccc: 000680e7      jalr a3
cd0: fc1ff06f      j  c90 <__call_exitprocs+0xf4>

```

Disassembly of section .text.__retarget_lock_init:

```

00000cd4 <__retarget_lock_init>:
cd4: 00008067      ret

```

Disassembly of section .text.__retarget_lock_init_recursive:

```

00000cd8 <__retarget_lock_init_recursive>:
cd8: 00008067      ret

```

Disassembly of section .text.__retarget_lock_close:

```

00000cdc <__retarget_lock_close>:
cdc: 00008067      ret

```

Disassembly of section .text.__retarget_lock_close_recursive:

```

00000ce0 <__retarget_lock_close_recursive>:
ce0: 00008067      ret

```

Disassembly of section .text.__retarget_lock_acquire:

```

00000ce4 <__retarget_lock_acquire>:
ce4: 00008067      ret

```

Disassembly of section .text.__retarget_lock_acquire_recursive:

```

00000ce8 <__retarget_lock_acquire_recursive>:
ce8: 00008067      ret

```

Disassembly of section .text.__retarget_lock_try_acquire:

```

00000cec <__retarget_lock_try_acquire>:
cec: 00100513      li  a0,1
cf0: 00008067      ret

```

Disassembly of section .text.__retarget_lock_try_acquire_recursive:

```

00000cf4 <__retarget_lock_try_acquire_recursive>:
cf4: 00100513      li  a0,1
cf8: 00008067      ret

```

```
Disassembly of section .text.__retarget_lock_release:

00000cfc <__retarget_lock_release>:
cfc: 00008067      ret

Disassembly of section .text.__retarget_lock_release_recursive:

00000d00 <__retarget_lock_release_recursive>:
d00: 00008067      ret
```

isr_test.lss