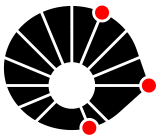
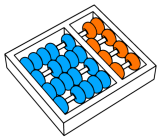




ReonV: Implementação de um processador Leon3 com Instruções RISC-V

Remoção da Janela de Registradores e Implementação de um Branch Bredictor

V. M. Eichemberger

Relatório Técnico - IC-PFG-18-35
Projeto Final de Graduação
2018 - Dezembro

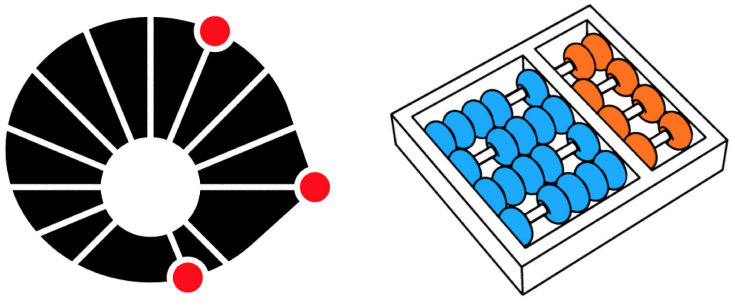
UNIVERSIDADE ESTADUAL DE CAMPINAS
INSTITUTO DE COMPUTAÇÃO

Engenharia de Computação - Curso 34 (AA)
Instituto de Computação - UNICAMP, 2s2018

Projeto Final de Graduação

ReonV: Implementação de um processador *Leon3* com Instruções *RISC-V*

Remoção da Janela de Registradores e Implementação de um Branch Predictor



Relatório Técnico - IC-PFG-18-35

Vítor Marge Eichemberger, RA 149053
Orientador: Prof. Dr. Rodolfo Jardim de Azevedo

Resumo

Este documento consiste no relatório do trabalho executado como Projeto Final de Graduação (PFG) do referido aluno, sob a orientação do *Prof. Dr. Rodolfo Jardim de Azevedo*. A área escolhida para o projeto é *Arquitetura de Computadores* (da grande área *Sistemas de Computação*), e o título "*ReonV: Implementação de um processador Leon 3 com Instruções RISC-V*". O projeto foi executado ao longo do segundo semestre de 2018, compreendendo os meses de Agosto/2018 a Novembro/2018, encerrando-se no início de Dezembro/2018.

O *Instituto de Computação da Unicamp* (*IC-UNICAMP*) [1], assim como muitas outras universidades e institutos de pesquisas, trabalha com a área de *Arquitetura de Computadores* [2], logo também com processadores. Neste meio há a preferência por uma arquitetura chamada *RISC-V* [3], porém esta não possui uma diversidade de suporte tão grande. Assim o IC iniciou um projeto para trocar a arquitetura do *Leon3*, processador com amplo suporte, de *SPARC* (sua original) para *RISC-V*, objetivando aumentar a gama de suporte ao *RISC-V*. Este projeto foi batizado de ***ReonV***.

Este trabalho consistiu portanto na continuidade de um projeto já iniciado anteriormente pelo IC, começada através de uma Iniciação Científica, porém mais focado em algumas tarefas em específico: A remoção da *Janela de Registradores* do *Leon3* e a implementação de um *Branch Predictor* no *ReonV*. Sendo feito também, neste projeto, um contador de ciclos, para auxiliar nas atividades, que fica como uma ótima contribuição extra.

1 Introdução

O *ReonV* tem origem através de um ramo de pesquisa do IC sobre supervisão do Prof. Rodolfo, para desenvolver uma versão do processador *RISC-V* que tenha maior suporte a kits de desenvolvimento de *FPGA* [21], utilizando como base o processador Leon3.

O *RISC-V* é um conjunto de instruções (do inglês *ISA*, *Instruction Set Architecture* [22]) aberto, desenvolvido pela *University of California* para uso livre de qualquer propósito, focado nos princípios *RISC* (*reduced instruction set computing*), por esses motivos acabou sendo muito adotado pela área de pesquisa e hoje é amplamente utilizado, inclusive no *Instituto de Computação* da UNICAMP. Já o *Leon3* é um processador SPARC desenvolvido pela *Sun Microsystems*.

Infelizmente as implementações do *RISC-V* não suportam muitas placas FPGAs, ao contrário do *Leon3* que possui uma diversidade bem maior. Assim, para sanar este problema, foi realizada a alteração do conjunto de instruções do Leon3 para suportar as instruções do RISC-V, permitindo rodar programas *RISC-V* nas placas suportadas pelo Leon3. Este projeto foi batizado de *ReonV* (uma mistura de ambos os nomes) e foi iniciado através de uma Iniciação Científica. Parte do desenvolvimento já foi realizada e agora é necessário completar o ambiente de execução.

O restante será realizado por dois alunos em dois Projeto Finais de Graduação distintos em conjunto com o aluno da iniciação científica. Sendo este em questão, mais focado pro lado do hardware, objetivando adaptar o processador em 2 aspectos importantes, a Janela de Registradores (funcionalidade da arquitetura antiga que não é mais usada) e o Branch Predictor (funcionalidade essencial de qualquer arquitetura que precisa ser implementada).

Janela de Registradores [7] é uma funcionalidade usada por algumas ISA, como a SPARC, para melhorar a performance de chamadas (procedure calls, subrotinas) [8], uma operação bem comum. Basicamente consiste em possuir um banco de registradores com múltiplas janelas (que podem ser interpretadas como sub-bancos internos) chegando a possuir até 520 registradores no caso do SPARC, diferente do banco tradicional que, sem janela, em geral fica na faixa dos 32, ou seja, uma bela economia de circuitos.

Branch Predictor [9] é um circuito digital que tenta adivinhar qual direção um branch [23] (instrução de controle de fluxo) irá tomar antes mesmo de ter a resposta definitiva deste. Seu propósito é melhorar o fluxo no pipeline [10], desempenhando um papel fundamental em atingir uma alta performance efetiva.

2 Objetivos e Atividades

Dado que boa parte do ramo de pesquisa mencionado está dependendo do *ReonV*, o objetivo desse *Projeto Final de Graduação* (PFG) consiste em retomar e acelerar o desenvolvimento dele, em continuidade da Iniciação Científica e em contato com os outros pesquisadores do ramo.

Foram listadas um conjunto de atividades que podem ser separadas em dois conjuntos distintos mas complementares que visam finalizar a implementação do processador em questão. Um conjunto mais voltado à adaptações de hardware e outro voltado à software.

A intenção deste projeto em particular é contribuir de forma significativa no avanço do *ReonV* com foco no hardware, através das seguintes tarefas:

- Remover a funcionalidade de *Janela de Registradores*;
- Implementar um *Branch Predictor* para funcionar com *ISA RISC-V*;

Ainda houve, também, a implementação de um contador de ciclos, algo que não estava previsto inicialmente, mas que acabou sendo necessário para testes, já que é uma estatística muito importante. Além de ter ajudado no projeto, fica como um legado bastante positivo, dado a importância e utilidade dessa estatística.

3 Justificativa

Como já evidenciado, o *IC* necessitava desse projeto, tanto que este já havia sido começado por uma Iniciação Científica, porém ainda se encontrava no início, apenas com o básico, com elementos faltando, e alguns estando incompletos. Assim sendo justificada a iniciativa deste PFG.

Sobre as atividades do projeto e contribuições escolhidas para integrar ao *ReonV*, foi optado por começar com aquelas que eram mais importantes e prioritárias, necessitando ser feitas o quanto antes.

A *Janela de Registradores* é utilizada no *SPARC* porém não no *RISC-V*, assim para tanto precisando ser removida para que o *ReonV* possa operar corretamente com esta *ISA*. Até o presente momento o *ReonV* apresentava um *workaround* para resolver esta questão, através de um mapeamento dos registradores do *Leon3* para seus equivalentes no *RISC-V*, ou seja, a *janela* era ignorada, mas continuava presente, ocupando espaço, justificando a necessidade de sua remoção.

Já com relação ao *Branch Predictor*, o *ReonV* se encontrava sem um ativamente implementado, devido às modificações, estando assim sem um ganho de eficiência importante, e portanto precisando de um.

4 Metodologia

O projeto tomou como base o repositório e conteúdo anteriormente utilizado [6] pela Iniciação Científica que começou este projeto, de autoria do aluno *Lucas Castro*, que se resume ao código, configurações e design do processador junto com scripts para construção, teste e execução daquele. Também foi utilizado as ferramentas necessárias para sua execução (ferramentas para lidar com hardware):

- ***Xilinx Vivado*** [12], software de síntese para compilar, montar e rodar o design do processador;
- ***Xilinx XMD*** [13], software para transferir o projeto sintetizado à placa;
- ***FPGA Digilent Nexys 4 DDR*** [14], placa onde o processador era montado e executado

Além das ferramentas para depuração, testes e execução de código ou programas dentro do processador sintetizado na *FPGA* (ferramentas para lidar com software):

- ***GRMON3*** [15], para depuração e comunicação com a placa;
- ***RISC-V Toolchain*** [16], conjunto de ferramentas para lidar com RISC-V, incluindo o compilador para código *RISC-V*;

Devido a adversidade de algumas ferramentas (especialmente as da *Xilinx*) não rodarem na máquina do autor deste PFG (aluno *Vítor Marge Eichemberger*) - estas não são compatíveis com *Mac* - optou-se por utilizar um servidor remoto, disponibilizado por seu orientador, este servidor ficava no próprio IC-UNICAMP, rodando em *Linux Ubuntu* [17] e com interface *Xfce* [18]. Assim necessitando de mais ferramentas para lidar com o servidor:

- ***Terminal via SSH***, para comunicação direta por interface de comando;
- ***X2Go*** [19], software para desktop remoto;

O desenvolvimento e modificações do processador se deu através de programação em linguagem de descrição de hardware ***VHDL***. Já para programas testes foi usado ***linguagem de programação C*** e ***linguagem de montagem para RISC-V*** [20]. Utilizando editores de texto e o *Vivado*, mas com preferência pelo editor de texto ***Atom***, devido ao grande ganho de produtividade gerado por suas funcionalidades e plugins.

As etapas inicialmente consistiam em Preparação do Ambiente, Estudo, Remoção da Janela de Registradores, Implementação do Branch Predictor e Revisão/Testes. Com as atividades ocorrendo da seguinte forma:

- Reuniões iniciais com o orientador, e planejamento;
- Preparação do Ambiente de Trabalho;
- Aprendizado através do estudo e testes das ferramentas e do processador (o *ReonV*);
- Reuniões com o *Lucas* para entender o estado anterior do projeto;
- Remoção da Janela de Registradores (às vezes encurtado para "*janela*") do ReonV;
- Implementação de um *Branch Predictor* (às vezes abreviado como "*BP*") no ReonV;
- Correções sobre o *Branch Predictor*;
  Sempre, ao final de cada etapa, fazendo os respectivos testes e revisão.

E, com uma atividade não (inicialmente) planejada ocorrendo entre a remoção da *janela* e a implementação do *BP*, a Implementação de um Contador de Ciclos, detalhada mais adiante (nos tópicos de *Desenvolvimento*, mais precisamente em *5.4*).

5 Desenvolvimento

5.1 Considerações Iniciais

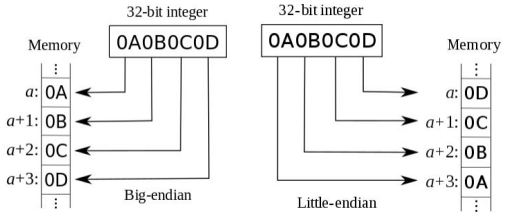
Algum entendimento básico do tema de *Arquitetura de Computadores* pode ser necessário para compreender integralmente este relatório.

5.1.1 *RISC-V* e *Leon3*

Tanto *ReonV* quanto o *Leon3* (que segue arquitetura *SPARC*) apresentam um pipeline de 7 estágios. A arquitetura não especifica isso, pois é uma decisão da implementação. O ReonV acabou usando os mesmos estágios o Leon3 por conveniencia (era mais fácil manter a mesma estrutura). São eles:

1. **Fetch**, a próxima instrução é retirada da memória;
2. **Decode**, decodifica a instrução;
3. **Register Access**, acessa os registradores necessários;
4. **Execute**, executa operações de lógica e aritmética;
5. **Memory**, acessa a memória;
6. **Exception**, trata exceções;
7. **Write Back**, os dados são escritos de volta (salvos);

Mas esta é uma das poucas semelhanças, pois em geral não são parecidos e em muitas situações optam por implementações opostas. É o caso, por exemplo do *endianness* [24], enquanto que *SPARC* (consequentemente o *Leon3*) usa *big-endian*, *RISC-V* usa *little-endian*.



Endianness

Outra diferença importante é quanto ao banco de registradores, *RISC-V* define um banco simples e fixo em 32 registradores de 32 bits, enquanto que o *Leon3* usa um banco variável com *Janela de Registradores*, conforme definida pelo *SPARC*.

5.1.2 *GRMON*

GRMON (*Gaisler Research Monitor*), é uma ferramenta (conhecida como *monitor*) de debug e comunicação com o *Leon3*, fornecida pela *Gaisler Research* [11] (sua desenvolvedora). Foi amplamente usado neste projeto.

É importante ressaltar que ele é voltado ao *Leon3*, e portanto não compatível com o *RISC-V*. Infelizmente não há outra opção no momento, dado que não existe um *monitor* para *ReonV* e que o *GRMON* é código fechado (portanto não pode ser adaptado como o *Leon3*). Motivo pelo qual outro trabalho é executado em cima do ReonV, em paralelo a este, pelo aluno Ricardo Charf, objetivando criar um substituto para o *GRMON*, provavelmente batizado de *RVMON*, *ReonV Monitor*.

Outros pontos importantes de se destacar são que os valores exibidos por ele em sua maioria estão em hexadecimal (há capturas de tela dele neste relatório), e que ao rodar um programa, ele automaticamente zera todos os registradores.

5.2 Preparação do Ambiente de Trabalho

Após as reuniões iniciais, o trabalho foi iniciado pela preparação do ambiente.

Primeiramente foram baixadas e instaladas as ferramentas necessárias para a interação com o servidor, logo em seguidas as demais ferramentas do projeto no servidor (todas essas são detalhadamente explicadas no tópico *4, Metodologia*), com o *ReonV* e suas dependências sendo instalados segundo tutorial no repositório do oficial *ReonV* [6].

Houve algumas dificuldades relacionadas às instalações, devido à incompatibilidades, porém as versões corretas foram encontradas e devidamente instaladas sem mais problemas.

Na sequência foram feitos testes relacionados a execução, um pequeno problema de *USB* ocorreu, mas pode ser corrigido ao plugar e desplugar (manualmente) a *FPGA* do servidor. Após isso o código básico de teste provido pelo Lucas rodou sem problemas, indicando um correto funcionamento de todo o material de trabalho.

Após isso entrou-se em uma fase de estudo, iniciada por uma reunião com o *Lucas*, onde ele explicou toda a arquitetura e organização dos arquivos do projeto. Basicamente os códigos fonte ficam na pasta *lib* (dentro dela, em *gaisler* está os fontes das unidades principais do processador) e configurações específicas de cada placa ficam em pastas relativas (com nome da *FPGA* em questão) dentro de *designs*, sendo a *FPGA* deste projeto (*Digilent Nexys 4 DDR*) na pasta *designs/leon3-digilent-nexys4ddr*. Também foi mostrado como configurar e executar o *ReonV* através do *Vivado*.

No mesmo período pequenos erros foram corrigidos (valores fixos que foram adaptados para *"softcode"*).

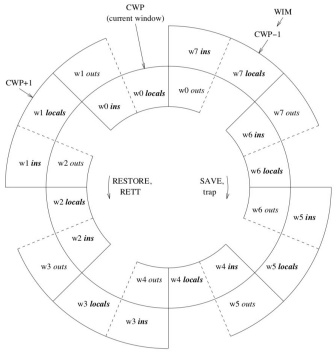
Posteriormente o repositório original foi clonado [11] para poder executar as modificações sem prejudicar o repositório original.

Ao longo do projeto houve alguns erros relacionado a redes (em geral problemas ou lentidão relacionados a máquina ou rede local).

5.3 Remoção da *Janela de Registradores*

5.3.1 Contextualização

O *Leon3*, seguindo a arquitetura *SPARC*, apresenta um mecanismo de *Janela de Registradores* com janelas variando de 2 a 32 (8 é o valor padrão), sendo cada uma com 24 registradores (8 proprios, 8 de interseção com a janela seguinte e 8 com a anterior), além de mais 8 registradores globais, conforme figura abaixo.



Modelo da *Janela de Registradores* segundo manual do *SPARC*.

5.3.2 Remoção da Janela

A primeira tentativa foi uma mudança no código *VHDL*, alterando as constantes relativas ao tamanho da janela de 2 para 1, porém não surtiu efeito. Mais tarde na reunião inicial feita com o *Lucas* (citada no tópico *5.2*, *Preparação do Ambiente de Trabalho*), foi constatado que na verdade o código fonte da pasta *lib* tem a parte de constantes sobrescrita por arquivos de configurações relativos a cada placa, separados em sua relativa pasta dentro de *designs*, portanto a primeira tentativa não tinha efeito. Tentou-se forçar a configuração para 1, porém isso acarreta em erros, o que faz sentido, dado que a estrutura de *janela* ainda estava presente e seu valor mínimo é 2, portanto partiu-se para outra abordagem, por hora a quantidade de *janelas* foi fixada em 2.

Um estudo sobre o código começou a ser feito para buscar entender onde estava e como funciona a estrutura de implementação da *janela*. Em paralelo foram executados os primeiros deste sobre o uso da placa, para ter parâmetros comparativos, que são exibidos mais adiante.

Após o estudo, conseguiu-se determinar onde e como estava implementada a *janela*. Basicamente ela consistia num mecanismo complexo de memória usando Sync RAMs que variavam para cada dispositivo e para cada tamanho de janela, porém havia ainda uma estrutura mais simples e genérica, provavelmente para placas não contempladas pela estrutura mais customizada, esta consistia numa banco com o número de registradores relativos ao total, e as janelas portanto funcionam como máscaras de endereçamento para os respectivos registrador, ou seja, nessa estrutura o banco de registradores era um vetor sequencial de

registradores com janelas "virtuais", as janelas não existiam de fato, eram só a forma como eram endereçado os registradores.

Essa estrutura se distribuía pelos arquivos da seguinte forma:

- **iu3.vhd** - Unidade de Inteiros do processador (unidade "principal");
- **leon3x.vhd** - Top-level, arquivo fonte mais alto nível definindo todo o processador;
- **regfile_3p_l3.vhd** - Top-level do banco de registradores;
- **regfile_3p.vhd** - Fonte que decidia entre a implementação genérica oucustomizada do banco;
- **memory_inferred.vhd** - Fonte que implementava várias estruturas de memória genérica;
- **generic_regfile_3p** - entidade dentro de memory_infered que implementava o banco de registradores genérico;

Seguindo na ordem listada, do **iu3.vhd** ao **generic_regfile_3p** (dentro de **memory_inferred.vhd**) é a sequência de fluxo entre os códigos para acessar do processador o banco de registradores.

O número de janelas é passado ao código **leon3x.vhd,** *top-level* (alto nível que monta a arquitetura), pelos arquivos de configuração. O *top-level* por sua vez, calcula o número de registradores e o número de bits de endereçamento deles e os repassa para o banco de registradores. Por fim este é montado de acordo com a placa, escolhendo entre um genérico ou "customizado" mais específico a cada caso.

Com tudo isso em mente, foi possível fazer as adaptações necessárias, basicamente o código foi alterado para sempre usar o banco de registradores genérico e este adaptado para funcionar conforme o definido pelo RISC-V, ou seja, um banco simples de 32 registradores de 32 bits cada. Sinais auxiliares (como os de endereçamento) foram adaptados também.

5.3.3 Análise e Testes

Referência Inicial - Síntese			
Resources	**Estimation**	**Available**	**Utilization %**
LUT	8854	63400	13,97%
LUTRAM	97	19000	0,51%
FF	5348	126800	4,22%
BRAM	23	135	17,04%
IO	81	210	38,57%
BUFG	14	32	43,75%
PLL	5	6	83,33%
Configurado 2 Janelas (antes 8) - Síntese			
Resources	**Estimation**	**Removed**	**Improvement**
LUT	8902	-48	-0,54%
LUTRAM	97	0	0,00%
FF	5335	13	0,24%
Outros se mantiveram iguais			
Removido o essencial da janela - Síntese			
Resources	**Estimation**	**Removed**	**Improvement**
LUT	8845	9	0,10%
LUTRAM	145	-48	-33,10%
FF	5347	1	0,02%
BRAM	21	2	9,52%
Outros se mantiveram iguais			

Referência Inicial - Implementação			
Resources	**Estimation**	**Available**	**Utilization %**
LUT	8592	63400	13,55%
LUTRAM	97	19000	0,51%
FF	5344	126800	4,21%
BRAM	23	135	17,04%
IO	81	210	38,57%
BUFG	11	32	34,38%
PLL	5	6	83,33%
Configurado 2 Janelas (antes 8) - Implementação			
Resources	**Estimation**	**Removed**	**Improvement**
LUT	8641	-49	-0,57%
LUTRAM	97	0	0,00%
FF	5331	13	0,24%
Outros se mantiveram iguais			
Removido o essencial da janela - Implementação			
Resources	**Estimation**	**Removed**	**Improvement**
LUT	8586	6	0,07%
LUTRAM	145	-48	-33,10%
FF	5343	1	0,02%
BRAM	21	2	9,52%
Outros se mantiveram iguais			

Remoção da estrutura da janela nas memorias		
Resultados Gerais (finais comparados aos disponíveis)		
LUTRAM Removida	-0,25%	
BRAM Remvoida	1,48%	
GANHO GERAL	1,23%	

Tabelas comparando o impacto da *janela* no uso da *FPGA*.

Como pode-se notar pelas tabelas o ganho não foi tão grande, parte disso se deve a alguns sinais auxiliares restantes (ainda relativos ao modelo antigo) e também por ter muita coisa que é fixa, os endereçamentos, por exemplo, estavam fixos em 10 bits (no caso de 32 janelas) porém para o caso de 2 janelas (que usam apenas 6 bits) esse valor poderia ser reduzido automaticamente pelo código, mas isso não era feito.

E ainda que hajam sinais a remover ou reduzir bits fixos, não são tão significativos por em geral serem apenas auxiliares não tão maiores que os atuais, e pelo objetivo mais importante, obter um banco de registradores no padrão *RISC-V*, ter sido atingido.

Por fim foi testado o funcionamento através do *GRMON*, testando todos os registradores (com leitura e escrita neles), também foi pega uma referência para saber como fica exibida os 32 registradores *RISC-V* na estrutura do *GRMON*.

O registrador 0xE (14) em muitos momentos é sobrescrito pelo GRMON, pois é usado por ele como apontador da pilha (stack pointer).

5.4 Contagem de Ciclos

5.4.1 O Problema

Ao iniciar os preparativos para a etapa de implementar o Branch Predictor, foram discutidas formas de como validar se a implementação deste estaria funcional, em outras palavras como testar confirmando o funcionamento. A resposta é um tanto quanto simples: basta verificar a eficiência do processador, que pode ser estimada pelo número de ciclos executados.

Em resumo, o *Branch Predictor* gera um ganho ao adivinhar o fluxo de um programa, reduzindo o gasto de tempo de esperar a resposta do fluxo, e ainda que este erre, o tempo gerado pela correção é o mesmo que simplesmente esperar a resposta, logo há apenas 2 situações, uma em que o *BP* erra e acaba gastando o mesmo tempo de esperar a decisão de fluxo, e outra em que ele acerta e assim "consegue pular" essa etapa de espera pela decisão, gerando nesta situação um ganho de tempo proporcional a esse "pulo".

Dado que um processador consiste num pipeline com N estágios, com cada estágio sendo executado no tempo de um ciclo, esse "pulo" é o número de ciclos economizado, e pode ser calculado pela diferença entre o estágio em que o *BP* faz a adivinhação (um chute) e o estágio onde a resposta (decisão de qual fluxo tomar) chega, sendo o "chute" sempre feito no primeiro estágio. No caso do *ReonV*, que segue *RISC-V* com 7 estágios, o "chute" é feito no *Fetch* (estágio 1), e a resposta deste chega no *Execute* (estágio 4), logo o ganho é **3** (4-1), ou seja, ele gasta 4 ciclos se erra, mas se acerta gasta apenas 1 (economizando 3).

Dessa forma a eficiência de um *BP* pode ser estimada por **E = TA x EC** (**TA** - *Taxa de Acerto*, **EC** - *Economia de Ciclos*), e sendo *taxa de acerto* e *economia de ciclos* nunca menores que zero, logo o ganho sempre é positivo, justificando sua importância na eficiência.

Assim, retomando a questão de testes, para garantir que um *BP* funciona uma ótima estimativa é ver se este gera redução no número de ciclos.

O problema que se segue é o fato do *Leon3* não possuir um contador de ciclo ou uma forma precisa de contar os ciclos. Ele possui um *timer*, porém este não distingue os ciclos, contando também durante os estados de *exceção* e *debug*, que não são interessantes para estimativa da eficiência do *BP*, por exemplo, este pode reduzir o número de ciclos, porém o processador pode entrar no modo *debug* e ou ser interrompido por uma *exceção* e gastar ciclos causando a impressão de que o *BP* possa estar errado, quando não está. E com isso deparou-se com um novo desafio: Como contar os ciclos de forma coerente para estimar o funcionamento do BP?

Bom como já dito, bastaria contar apenas quando o processador está executando o código de usuário e não tratando de eventos externos, mas para isso o *Leon3* não tinha estrutura pronta e portanto um contador de ciclos precisou ser implementado.

5.4.2 Busca por uma Solução

O primeiro passo foi investigar, indo atrás das possibilidades para as duas perguntas em questão:

1 - Onde salvar a contagem?

- **Contador interno próprio**, inviável pois o *GRMON* não reconheceria, pois só conhece a estrutura do *Leon3*;
- **No banco de registradores**, inviável pois ocupa espaço e tempo no banco, ou teria o trabalho de implementar instruções novas para isso;
- **Registrador Y**, viável pois é um registrador do SPARC que o RISC-V não usa e é interno ao pipeline, facilitando a utilização;

Devido a praticidade de usar como contador e por já ser facilmente lido pelo *GRMON*, a opção escolhida foi usar o registrador **Y**

2 - Como contar?

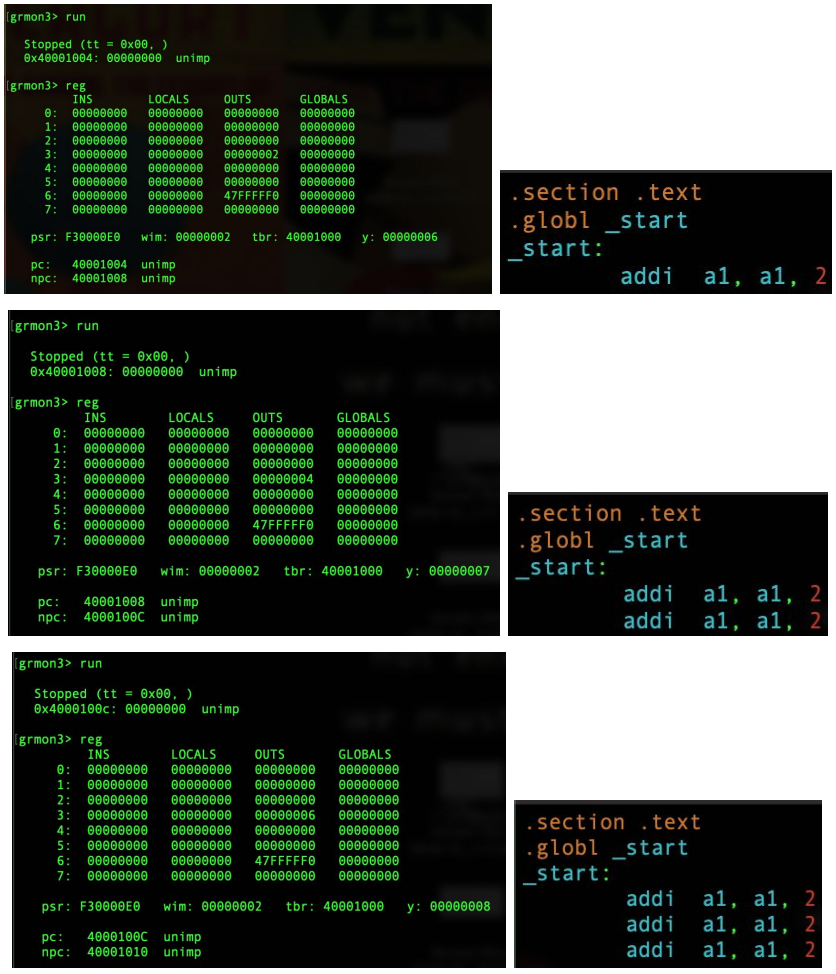
- Contar quantas vezes o processador muda de instrução;
- Contar os ciclos entre a instrução inicial e a final de um programa;
- Contar ciclos usando uma *flag* ou *variável* para considerar só o estado desejado do processador;

Todas foram testadas. As duas primeiras não deram resultados coerentes, porque não abrangiam todas as situações que podem ocorrer no processador ou não paravam a contagem quando deviam, já a terceira obteve sucesso ao encontrar e usar uma variável interna que indica o estado do processador (entre *Rodando*, *Em Exceção* e *Em Debug*)

Portanto, após essas investigações e testes, foi escolhido usar o registrador **Y** contando os ciclos quando o estado interno do processador é "*Rodando*" ("*run*" do inglês).

5.4.1 Análise e Testes

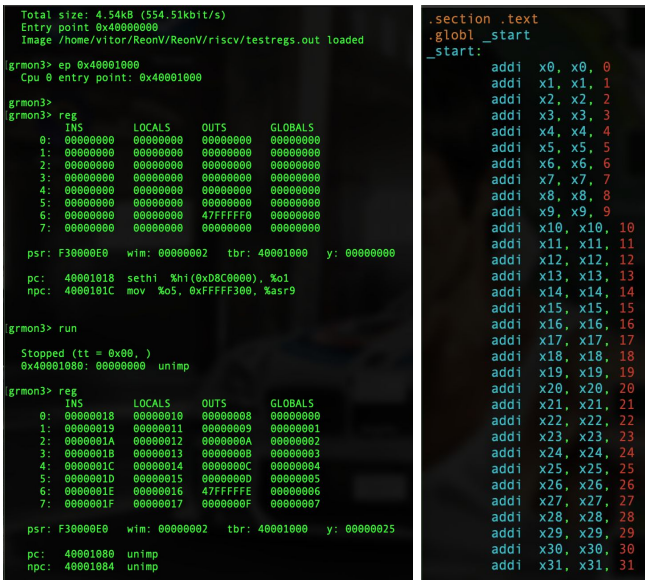
Por fim foram feitos mais testes para confirmar e todos foram bem coerentes, o número de ciclos diferia constantemente em **1** a menos do estimado pela teoria, justificado pela forma implementada, que acaba não contando o último ciclo.



Testes básicos do contador de ciclos.

Como pode ser visto, o número de ciclos estimado sempre fica abaixo de **1** do esperado, **ciclos = I + E - 1** (**I** - *Número de Instruções*, **E** - *Número de Estágios*) ou para nosso caso **I + 7 * 1 = I + 6**. Repare que os resultados são sempre **I+5**. E para testes de códigos maiores (códigos em *C*), mais difíceis de estimar o número de ciclos teórico, o contador deu resultados coerentes, proporcionais ao tamanho e complexidade do código.

Com tudo o acima validando este contador de ciclos.



Teste de escrita em todos os registradores. O registrador 14 (0xE) é sempre sobrescrito pelo GRMON.

Por fim foi executado um teste final que também comprovou o funcionamento do Banco de Registradores de forma correta sem mais a janela, além do número de ciclos -1 (0x25 = 37 = 32 instruções + 7 estágios -2).

5.5 Implementação do *Branch Predictor*

5.5.1 Contextualização

Como já explicado um *Branch Predictor* é uma funcionalidade que objetiva o ganho de eficiência ao tentar adivinhar o fluxo das instruções dentro do processador, sendo responsável por arriscar ou não mudar este fluxo antes da resposta. Existem diversos tipos de *Branch Predictor* e independente do tipo haverá um ganho, mesmo que muito pequeno, como já explicado no tópico sobre o problema da contagem de ciclos (tópico *5.4.2*).

A esmagadora maioria dos modelos acaba se apoiando em uma tabela, com informações sobre as quais o *BP* opera para decidir se salta (muda o fluxo) ou não, variando de modelo para modelo o que é armazenado e como isso influencia na decisão.

A maioria salva algo usado para identificar a instrução (em geral chamado de *tag*) junto com o endereço de salto (para onde o fluxo pode ser desviado, em geral chamado de *alvo* ou *destino*), com a maioria usando também informações relativas a uma máquina de estados, quem de fato faz a decisão de fluxo.

Há também opções bem simples como *Always Taken*, "sempre salta", ou mesmo a ausência de um *predictor*, situação conhecida como *Always Not Taken*, "nunca salta", ambos por sua vez usando bem menos informações, a segunda nem usa informações auxiliares.

O RISC-V por ser um ISA não especifica como deve ser seu *BP*, porém especifica o funcionamento da instrução de *branch*:

Conditional Branches

All branch instructions use the B-type instruction format. The 12-bit B-immediate encodes signed offsets in multiples of 2, and is added to the current `pc` to give the target address. The conditional branch range is ±4 KiB.

31	30 25	24 20	19 15	14 12	11 8	7	6 0
imm[12]	imm[10:5]	rs2	rs1	funct3	imm[4:1]	imm[11]	opcode
1	6	5	5	3	4	1	7
offset[12,10:5]		src2	src1	BEQ/BNE	offset[11,4:1]		BRANCH
offset[12,10:5]		src2	src1	BLT[U]	offset[11,4:1]		BRANCH
offset[12,10:5]		src2	src1	BGE[U]	offset[11,4:1]		BRANCH

Instrução de branch, conforme especificado no manual do RISC-V.

5.5.2 Modelo Anterior do ReonV

Antes de iniciar o processo, o primeiro passo foi descobrir e estudar o modelo que estava implementado no *ReonV*. O que faria mais sentido seria um *Always Not Taken* (implementação básica sem sistema de predição), porém não era o caso.

O que **foi descoberto é que na verdade os *branches* não estavam sendo tratados como deveriam**, havia apenas um mecanismo muito simples para evitar problemas, mas totalmente ineficiente: quando um *branch* era identificado, todo o *pipeline* era parado e segurado até se ter o resultado em mãos e aí decidir o fluxo. E ainda havia 2 agravantes:

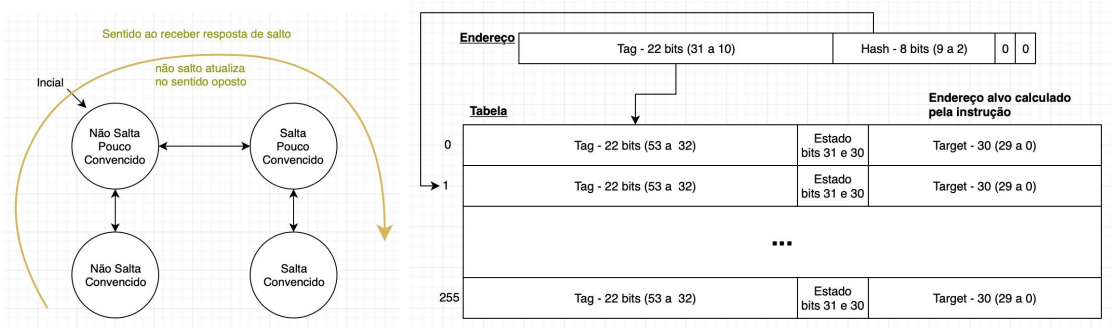
1. O resultado da decisão só saía após o estágio *Execute*, ou seja, em *Memory*, não sendo adiantado através de um mecanismo de *fowarding*.
2. Para segurar o *pipeline* no caso de *branch*, o processador precisa saber que a instrução é um *branch*, e isso só pode ser feito no segundo estágio, em *Decode*, o que portanto causava a entrada de 1 instrução no pipeline (em *Fetch*, primeiro estágio) que hora era descartada, hora não.

Em resumo, **todo *branch* gastava ao menos 3 ciclos** e se saltasse ainda criava uma bolha (na instrução adiantada erradamente em Fetch) gastando 4 ciclos nesse caso.

Um modelo que cumpre sua função de executar corretamente as instruções, mas falha absurdamente quanto a eficiência.

5.5.3 Modelo Inicial

O modelo inicialmente pensado foi um Branch Predictor de **2 bits de estado,** atualizado em cima de uma tabela de referência. Na tabela armazena tag, 2 bits para uma máquina de estados e o *alvo.* Toda vez que uma instrução é do tipo *branch* (o salto condicional) ela é armazenada na tabela com o estado inicial, e se ocorre de novo tem seu estado atualizado seguindo a lógica e estrutura abaixo:



Lógica e estrutura do modelo inicial do *BP*.

Resumindo a estrutura: do endereço eram tirados **8 bits de hash** (**256 entradas**), que era calculado pelos primeiros 8 bits pulando os 2 primeiros que são sempre 0, e na sequência os restantes do endereço formavam os **22 bits de tag** (que identifica aquele endereço para o caso de colisão), com a tabela salvando na linha (cujo número era calculado pelo hash) a tag, o estado do *BP* para aquele endereço, e seu endereço alvo (para onde provavelmente ele saltaria), também ignorando neste último seus 2 bits finais que são sempre 0. Conforme exibido na imagem acima.

Com o modelo definido, partiu-se para sua implementação. Primeiramente foram feitos sinais, tipos, funções e procedimentos para sintetizar a estrutura básica. Na sequência fazendo e testando um sinal de controle para anulação em caso de erro, e logo após, um sistema de *fowarding* de informações importantes, afinal na hora de corrigir o salto é preciso ter informações sobre o "chute" (decisão do fluxo tomada logo no início do primeiro estágio, pelo *BP*), como qual foi o fluxo tomado e mais tarde (após o correto cálculo) qual seria a resposta correta, informações essas que surgem antes do estágio onde é corrigido o salta e que precisam ser repassadas até ele.

Infelizmente, na hora de fazer a integração de fato entre o processador e o *BP*, aquele passou a se comportar de forma errônea. Correções foram feitas porém o problema persistiu.

Sem muitos resultados e não conseguindo identificar o problema, partiu-se para um plano B, com possibilidade de retomada do plano A futuramente: seria implementado uma opção bem mais básica, um *Always Not Taken*, depois outra um pouco menos simples, um *Always Taken*, na finalidade de tentar identificar melhor onde e qual seria o problema, posteriormente voltando novamente ao modelo inicial.

5.5.4 Mudança de Modelo

Após falhas no modelo inicial, optou-se por uma estratégia incremental, tentar modelos mais simples e posteriormente retomar o modelo inicial, mais complexo.

O primeiro e mais simples modelo implementado foi o *Always Not Taken*, conhecido em português como *nunca salta*, que como o nome diz, supõe que nunca deve saltar (mudar/desviar o fluxo), tão simples que nem usa uma tabela auxiliar, portanto nem sendo considerado um *predictor*.

Apesar de parecer não ter ganho de tão simples, ele apresenta sim um ganho, baixo, mas existente sobre o modelo anterior do ReonV, afinal sempre acerta quando não deve saltar, e quando erra não gasta mais ciclos, mas sim a mesma quantidade do cenário anterior (mais detalhes explicado já no tópico *5.4.2*, sobre o problema da contagem de ciclos).

Esse modelo obteve sucesso, não causou comportamento errôneo no processador, e conforme previsto gerou um pequeno ganho de desempenho, **ficando com speedup na casa dos 140%**, acelerando o processador 40% a mais na média, sendo **o menor valor ficando na casa dos 120% e o maior na casa dos 160%**.

Fora isso, **também foi muito útil pois ajudou na correção** de diversos erros de implementação e problemas gerados pelo modelo anterior.

Com o resultado positivo partiu-se para a implementação de um modelo considerado intermediário entre esse e o inicial, o *Always Taken*, ou *sempre salta* no português. Este passa a usar uma tabela, afinal precisa registrar os endereços que podem saltar (com instrução de *branch*), um *predictor* simples, por isso considerado intermediário. Mas infelizmente não foi possível fazê-lo funcionar, assim como o inicial passou a apresentar comportamento errôneo, sem resultados positivos.

O problema foi investigado mais a fundo e foi encontrado o local, mas não a causa nem o que exatamente: algum problema durante a leitura ou escrita da tabela, esse procedimento não está se comportando como deveria, apesar de ter sido revisado várias vezes. Dado a proximidade do prazo, infelizmente, teve-se que optar por manter o plano B, mas 2 hipóteses sobre o erro ainda tiveram tempo de ser formuladas:

- Problema de *endian*, dado que o *endian* do *RISC-V* é o oposto do *Leon3*;
- Algum erro muito sútil de conta ou acesso a índice;

Por fim foram feitas adaptações finais para manter a estrutura do modelo inicialmente desejado, para correção futura, mas sem prejudicar o projeto de qualquer forma.

Foi agregado portanto, **um modelo já pré-implementado e diversas correções** que deixaram a parte de tratamento de *branch* do *ReonV* funcionando exatamente como deveria (num cenário sem um *Branch Predictor*). Um ótimo exemplo é que agora *ReonV* é capaz de anular instruções em caso de erro de salto, algo que antes não era possível.

Devido ao tamanho de código implementado recomendo o acesso ao fonte principal (*lib/gaisler/leon3v3/iu3.vhd*) e o restante do repositório deste projeto [11], pois ocuparia muito espaço tê-lo aqui. A seguir uma pequena amostra da principal estrutura de dados do *BP*.

```
-- 2 bit BP control type
type bp2b_ctrl_type is record
    table          : bp2b_table_type; -- 2 bit BP table
    last_taken     : std_ulogic; -- boolean indicating if last branch was taken
    last_hit       : std_ulogic; -- boolean indicating if last prediction was correct
    last_condition : std_ulogic; -- boolean indicating if last condition was true
    total_misses   : word; -- total BP misses
    total_hits     : word; -- total BP hits
end record;
```

Trecho de código do *BP*, tipo responsável por seu controle

5.5.5 Análise e Testes

Para a parte de anulação em caso de erro foi feito um teste da seguinte forma, o código do processador foi adaptado que uma instrução específica fosse interpretada como um salto errado e acionasse o sinal de anulação/correção específico do *BP*, conforme imagens abaixo comprovam:

```
40000000 Binary data              4.4kB /   4.4kB   [==============>] 100%
Total size: 4.42kB (566.00kbit/s)
Entry point 0x40000000
Image /home/vitor/ReonV/ReonV/riscv/annul.out loaded

grmon3> ep 0x40001000
 Cpu 0 entry point: 0x40001000

grmon3> reg
        INS       LOCALS    OUTS      GLOBALS
   0:  00000000  00000000  00000000  00000000
   1:  00000000  00000000  00000000  00000000
   2:  00000000  00000000  00000000  00000000
   3:  00000000  00000000  00000000  00000000
   4:  00000000  00000000  00000000  00000000
   5:  00000000  00000000  00000000  00000000
   6:  00000000  00000000  47FFFFF0  00000000
   7:  00000000  00000000  00000000  00000000

 psr: F30000E0   wim: 00000002   tbr: 40001000   y: 00000000

 pc:   40001018  sethi  %hi(0x14140400), %o1
 npc:  4000101C  sethi  %hi(0x16140000), %o1


grmon3> run

 Stopped (tt = 0x00, )
 0x4000102c: 00000000  unimp

grmon3> reg
        INS       LOCALS    OUTS      GLOBALS
   0:  00000000  00000000  00000000  00000000
   1:  00000000  00000000  00000000  00000000
   2:  00000000  00000000  00000001  00000000
   3:  00000000  00000000  00000005  00000000
   4:  00000000  00000000  00000000  00000000
   5:  00000000  00000000  00000000  00000000
   6:  00000000  00000000  47FFFFF0  00000000
   7:  00000000  00000000  00000000  00000000

 psr: F30000E0   wim: 00000002   tbr: 40001000   y: 00000010

 pc:   4000102C  unimp
 npc:  40001030  unimp
```

```
.section .text
.globl _start
_start:
  addi x11, x11,  1 # sem problemas
  addi x11, x11,  1 # sem problemas
  addi x11, x11,  1 # sem problemas
  addi x11, x11,  1 # sem problemas
  addi x11, x11,  1 # sem problemas
  addi x12, x12,  0 # Anula o restante ao chegar em memory
  addi x10, x10, 16 # execute (anulado)
  addi x10, x10,  8 # reg access (anulado)
  addi x10, x10,  4 # decode (anulado)
  addi x10, x10,  2 # fetch (anulado)
  addi x10, x10,  1 # sem problemas, não é anulada
```

```
3599    -- for now I am testing instructions annul
3600    if (r.m.ctrl.inst = "00000000000001100000011000010011" or r.m.ctrl.inst = "00010011000001100000011000000000") then
3601        bp2b_annul := '1';
3602    end if;
```

Acima a esquerda resultados do teste, acima a direita o código testado e abaixo o trecho no código do ReonV que interpretava a instrução "*addi x12, x12, 0*" como erro e anulava as demais.

Repare acima que idealmente deveríamos ter, ao final da execução, os valores 5 em x11 e 1 em x10, que podemos confirmar nos resultados. Ou seja, teste bem sucedido, o sinal de controle para anulação funciona.

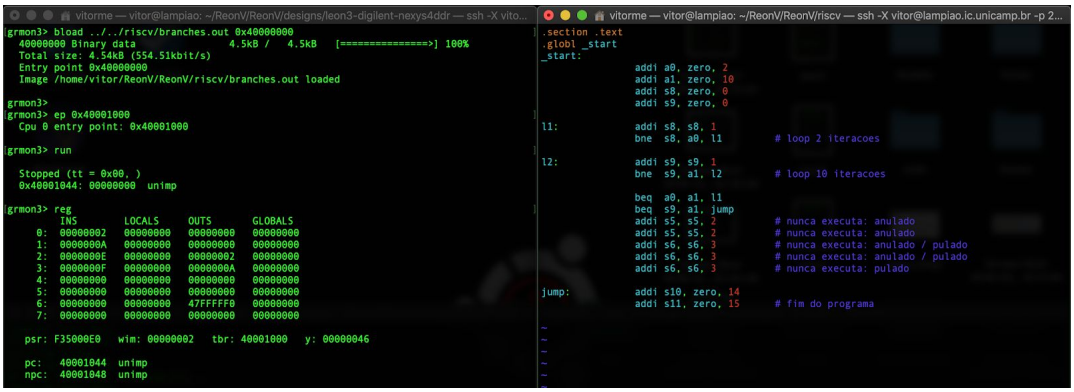
Abaixo a tabela comparativa que comprova o desempenho do *BP* implementado.

Programa	**Descrição**	**Sem BP**	**Always Not Taken**	**Speedup Médio**
loop.s	Loop simples em linguagem de máquina	84	54	155,6%
branches.s	Várias instruções de *branch*	112	70	160,0%
print_nested_loop.c	loop aninhado com chamada do *printf*	75622	58068	130,2%
programa do Lucas	programa de teste feito pelo *Lucas* em *C*	6689679	5258256	127,2%
OBS: Valores em medidos em ciclos pelo contador implementado no registrador Y			**Ganho Médio Geral**	**143,3%**

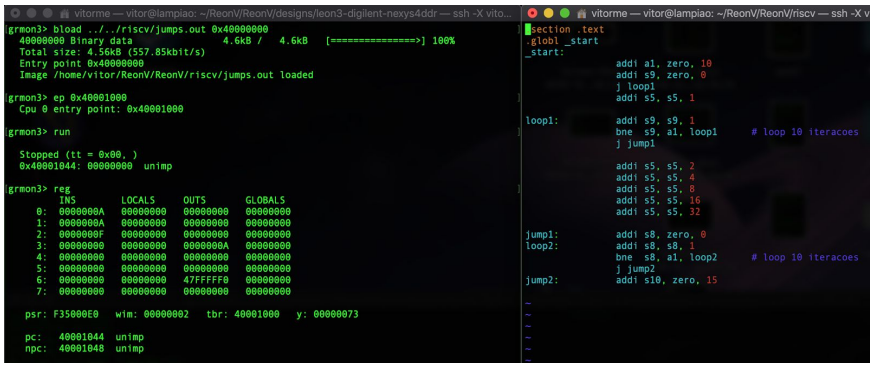
Tabela comparativa da implementação do *Always Not Taken* BP

Autor: Vítor Marge Eichemberger
Orientador: Prof. Dr. Rodolfo Jardim de Azevedo

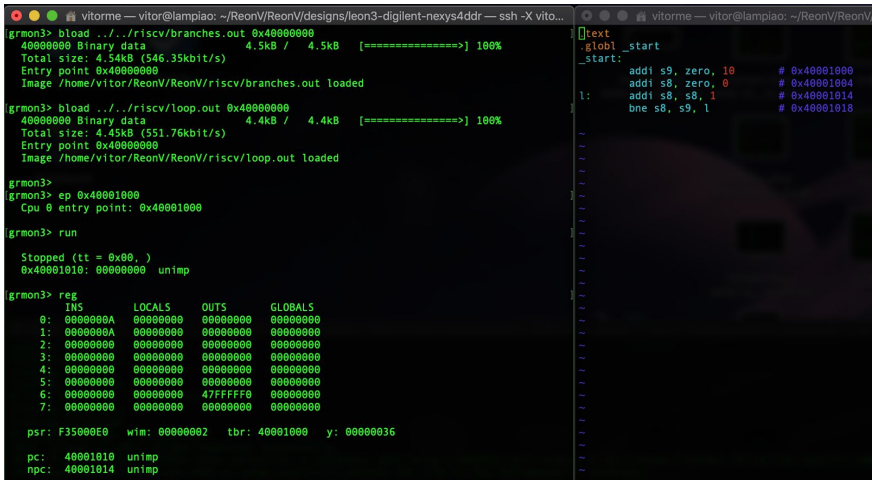
Abaixo seguem testes de execuções com o *BP nunca salta*, comprovando seu correto funcionamento.



branches.s - código para testar instruções de branch.



jumps.s - código para testar saltos não condicionais.



loop.s - código para testar um loop simples.

```
vitorme — vitor@lampiao: ~/ReonV/ReonV/designs/leon3-digilent-nexys4ddr — ss
grmon3> bload ../../riscv/print_nested_for.out 0x40000000
  40000000 Binary data              10.4kB /  10.4kB   [==============>] 100%
  Total size: 10.42kB (554.18kbit/s)
  Entry point 0x40000000
  Image /home/vitor/ReonV/ReonV/riscv/print_nested_for.out loaded

grmon3> ep 0x40001000
  Cpu 0 entry point: 0x40001000

grmon3> run
sum = 4200

  Stopped (tt = 0x00, )
  0x40001b80: 73001000  call  0x0C005B80

grmon3> reg y
        y = 58066 (0x0000e2d2)

grmon3>
```

```c
#include <stdio.h>
#include "mini_printf.h"

int main() {
        int n = 20, sum = 0;

        // sum = n * sum(1..n) = 4200 or 0x1068, if n = 20
        for(int i = 0; i < n; i++) {
                for(int j = 0; j < n; j++) {
                        sum += (j+1);
                }
        }
        printf("sum = %d\n", sum);

        return 0;
}
```

print_nested_for.c - código para testar loops aninhados em C.

```c
int fib();
#include <stdio.h>
#include "mini_printf.h"
#include "posix_c.h"

int main(){
        int n = 20;

        printf("Testing fib calculator, array dealing, write, read and lseek functions\
n\n");

        // Allocates memory for arrays
        int* array = (int*)_sbrk(n * sizeof(int));
        int* array2 = (int*)_sbrk(n * sizeof(int));

        printf("Address array 1: 0x%x\n", array);
        printf("Address array 2: 0x%x\n", array2);

        // Calculates fib(i), 1 <= i <= n
        for(int i = 1; i <= n; i++)
                array[i-1] = fib(i);

        // Writes results on output memory section
        _write(3, (char*) array,n * sizeof(int));

        // Sets pointer to beginning of output memory section
        _lseek(3, -n*sizeof(int), SEEK_CUR);

        // Reads the results into array2
        _read(3, (char*)array2, n * sizeof(int));

        // Checks if they were copied correctly
        int correct = 1;
        printf("correct = %d\n", correct);
        for(int i = 0; i < n; i++)
                if(array[i] != array2[i])
                {
                        //printf("ops!\n");
                        correct = -1;
                        //printf("(%02d)  %04d != %04d\n", i, array[i], array2[i]);
                }

        printf("correct = %d\n", correct);
        if(correct == 1)
                printf("\nCopied array1 into array2!\n");
```

```c
        // Checks if they were copied correctly
        int correct = 1;
        printf("correct = %d\n", correct);
        for(int i = 0; i < n; i++)
                if(array[i] != array2[i])
                {
                        //printf("ops!\n");
                        correct = -1;
                        //printf("(%02d)  %04d != %04d\n", i, array[i], array2[i]);
                }

        printf("correct = %d\n", correct);
        if(correct == 1)
                printf("\nCopied array1 into array2!\n");
        else
                printf("Oh no, error!\n");

        // Uses console
        for(int i = 1; i <= n; i++)
                printf("(%02d)  %04d = %04d\n", i, array[i-1], array2[i-1]);

        return 0;
}

int fib(int i){
        if(i == 1 || i == 2)
                return 1;
        return fib(i-1) + fib(i-2);
}
```

Código de teste do *Lucas*, hora referido como "*lucas_prog*" ou "*programa do Lucas*".

```
vitorme — vitor@lampiao: ~/ReonV/ReonV/designs/leon3-digilent-nexys4ddr — ssh -X vito...
grmon3> bload ../../riscv/lucas_prog.out 0x40000000
  40000000 Binary data              11.7kB /  11.7kB   [==============>] 100%
  Total size: 11.74kB (562.53kbit/s)
  Entry point 0x40000000
  Image /home/vitor/ReonV/ReonV/riscv/lucas_prog.out loaded

grmon3>
grmon3> ep 0x40001000
  Cpu 0 entry point: 0x40001000

grmon3> run
Testing fib calculator, array dealing, write, read and lseek functions

Address array 1: 0x43000400
Address array 2: 0x43000450
Fib(01) = 0001
Fib(02) = 0001
Fib(03) = 0002
Fib(04) = 0003
Fib(05) = 0005
Fib(06) = 0008
Fib(07) = 0013
Fib(08) = 0021
Fib(09) = 0034
Fib(10) = 0055
Fib(11) = 0089
Fib(12) = 0144
Fib(13) = 0233
Fib(14) = 0377
Fib(15) = 0610
Fib(16) = 0987
Fib(17) = 1597
Fib(18) = 2584
Fib(19) = 4181
Fib(20) = 6765

Copied array1 into array2!
```

```
vitorme — vitor@lampiao: ~/ReonV/ReonV/designs/leon3-digilent-nexys4ddr — ssh -X vito...
Address array 1: 0x43000400
Address array 2: 0x43000450
Fib(01) = 0001
Fib(02) = 0001
Fib(03) = 0002
Fib(04) = 0003
Fib(05) = 0005
Fib(06) = 0008
Fib(07) = 0013
Fib(08) = 0021
Fib(09) = 0034
Fib(10) = 0055
Fib(11) = 0089
Fib(12) = 0144
Fib(13) = 0233
Fib(14) = 0377
Fib(15) = 0610
Fib(16) = 0987
Fib(17) = 1597
Fib(18) = 2584
Fib(19) = 4181
Fib(20) = 6765

Copied array1 into array2!


Testing printf

Testing big int:  1234567890 = 1234567890
Big int in hex:   0x499602d2 = 0x499602d2
Testing negative: -987654321 = -987654321
Negative in hex:  0xC521974F = 0xc521974f
Testing unsigned: 4279383126 = 4279383126
Unsigned in hex:  0xff123456 = 0xff123456
This is a string = This is a string
C comes after B which comes after A!


Finished main!


  Stopped (tt = 0x00, )
  0x40001b80: 73001000  call  0x0C005B80

grmon3> reg y
        y = 5329920 (0x00515400)
```

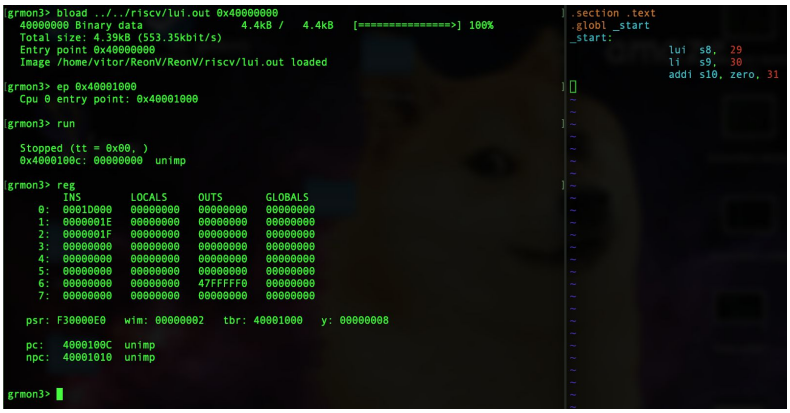
Execução do programa de teste do Lucas.

Autor: Vítor Marge Eichemberger
Orientador: Prof. Dr. Rodolfo Jardim de Azevedo

5.6 Considerações Finais

5.6.1 Adversidades Encontradas

Durante o andamento do projeto um ou outro problema na implementação do ReonV foram encontrados, provavelmente devido a se tratar de um projeto muito jovem. Dentre esses, um que se destacou foi na instrução lui, esta devia carregar um valor num dado registrador, porém está carregando um valor deslocado erroneamente.



Repare que dentre as 3 instruções para carregar valores, a *lui* carrega deslocado.

Outras barreiras encontradas que acabaram dificultando e impactando sobre o desenvolvimento foram:

- Documentação não muito boa e desorganizada por parte da *Gaisler* [11];
- Trabalho remoto (problemas com conexão e atividades extras)
- Ausência de um monitor/debugger voltado para o *ReonV* (o *GRMON* não permite debugar coisas específicas de *RISC-V* e por ser código fechado também não permite adaptações para atender a alguma necessidade de desenvolvimento ou debug voltado para o *RISC-V*);

Uma parte significativa do descoberto no começo só foi de fato entendida próxima ao final do projeto, por isso algumas informações contidas nesse relatório sobre alguns pontos de atividades iniciais (como por exemplo o funcionamento da *Janela de Registradores*) estão mais completas e detalhadas do que o conversado nas primeiras reuniões.

5.6.2 Repositório

Atualmente o conteúdo deste trabalho se encontra em um repositório separado, "*forkeado*" do original, mas já se encontra em fase de preparação para inclusão (através de uma solicitação de *Pull Request*) de volta ao repositório original. Testes já foram feitos para garantir que não há problemas que possam prejudicar o uso, só faltando apenas iniciar o processo de inclusão (Pull Request) que será feito muito em breve.

Por hora, pode ser encontrado no link https://github.com/VitorME/ReonV [11].

5.6.3 Continuidade

O encerramento das atividades foram de forma a permitir continuidade futura tanto por parte do próprio autor, como de outros contribuintes, dado que o código deste projeto é aberto e dedicada a comunidade da Computação não só da Unicamp, mas de todos que desejam trabalhar com pesquisa sobre a arquitetura aberta *RISC-V*. Note mesmo que este relatório já deixa pistas importantes para próximos passos, como por exemplo, a estrutura do modelo inicial do *BP* e hipóteses sobre sua falha, além da evidência do problema na instrução *lui*.

6 Conclusões

Apesar de todas as adversidades os objetivos foram atingidos em sua parte mais importante, podem ter faltado detalhes ou mudado modelos iniciais para versões mais simples, mas todos os pontos mirados tiveram grandes correções e melhorias, com todo o impacto positivo dessas mudanças continuando presente, não necessariamente com o ganho inicialmente estimado, mas uma boa parte disso com certeza, conforme comprovado ao longo deste relatório.

Assim como na graduação e na vida, muito do nosso planejamento tem que ser mudado ou adaptado, mas mantendo a determinação conseguimos atingir os mesmo objetivos, talvez não tendo 100% do que queríamos, mas com certeza muito dele.

Mais especificamente, com relação aos resultados do projeto temos a *Janela de Registradores* removida, e uma melhoria significativa no tratamento de *branches* que apesar de não ser o *BP* pretendido, ainda gera um ganho bom e cumpre o foco do objetivo por melhorar o tratamento de branches no *ReonV* em busca de eficiência, a essência daquele.

Por fim, muito desse trabalho será agregado a um projeto maior do IC que pode inclusive impactar de forma bastante positiva em outros projetos e não só do Instituto de Computação da Unicamp. Fica assim um trabalho que além de positivo pode ser continuado para agregar ainda mais.

7 Agradecimentos

Por fim, agradeço à toda a comunidade do IC por todo o suporte ao longo de minha graduação, mas com agradecimentos especiais, devido a contribuição em ajuda para este projeto, a:

- **Prof. Dr. Rodolfo Jardim de Azevedo**, orientador deste projeto que me deu todo o suporte necessário, mesmo sendo bastante ocupado;
- **Lucas Castro**, aluno que iniciou o projeto, pelo projeto que herdei e pelo suporte que me deu ajudando a entender o funcionamento e andamento deste;
- **Ricardo Charf**, aluno que fez o projeto paralelamente junto a mim, mas cuidando da parte de software, também pela ajuda prestada em alguns momentos;

Referências

[1] Instituto de Computação Unicamp
http://ic.unicamp.br

[2] Arquitetura de Computadores
https://en.wikipedia.org/wiki/Computer_architecture

[3] RISC-V
https://riscv.org
https://en.wikipedia.org/wiki/RISC-V

[4] Leon3, processador SPARC v8
https://www.gaisler.com/index.php/products/processors/leon3
https://en.wikipedia.org/wiki/LEON
Manuais:
https://www.gaisler.com/index.php/downloads/leongrlib

[5] Arquitetura SPARC v8
https://en.wikipedia.org/wiki/SPARC

[6] ReonV, repositório original e documentação
https://github.com/lcbcFoo/ReonV

[7] Janela de Registradores (Register Windows)
https://en.wikipedia.org/wiki/Register_window

[8] Chamadas / Subrotinas
https://en.wikipedia.org/wiki/Subroutine

[9] Branch Predictor & Branch Prediction
https://en.wikipedia.org/wiki/Branch_predictor
https://pt.wikipedia.org/wiki/Branch_prediction

[10] Pipeline
https://en.wikipedia.org/wiki/Pipeline_(computing)
https://en.wikipedia.org/wiki/Instruction_pipelining

[11] Repositório do Projeto deste PFG
https://github.com/VitorME/ReonV

[12] Xilinx Vivado
https://www.xilinx.com/products/design-tools/vivado.html

[13] Xilinx XMD
https://www.xilinx.com/support/documentation/sw_manuals/xilinx2014_3/SDK_Doc/tasks/sdk_t_xmd_console.htm
Descrito no capítulo 3 do manual de ferramentas Xilinx
https://www.xilinx.com/support/documentation/sw_manuals/xilinx2014_1/ug1043-embedded-system-tools.pdf

[14] FPGA Digilent Nexys 4 DDR
https://reference.digilentinc.com/reference/programmable-logic/nexys-4-ddr/start

[15] GRMON3
https://www.gaisler.com/index.php/products/debug-tools/grmon3

[16] RISC-V Toolchain
https://github.com/riscv/riscv-gnu-toolchain

[17] Ubuntu, distribuição de Linux
https://www.ubuntu.com

[18] Xfce, Desktop Environment
https://www.xfce.org

[19] X2Go
https://en.wikipedia.org/wiki/X2Go
https://wiki.x2go.org/doku.php

[20] ASM RISC-V
https://rv8.io/asm.html
https://rv8.io/isa.html

[21] FPGA
https://en.wikipedia.org/wiki/Field-programmable_gate_array

[22] ISA
https://en.wikipedia.org/wiki/Instruction_set_architecture

[23] Branch
https://en.wikipedia.org/wiki/Branch_(computer_science)

[24] Endiannes
https://en.wikipedia.org/wiki/Endianness

[25] Gaisler
https://en.wikipedia.org/w/index.php?title=Gaisler_Research&redirect=yes