Allender Vilar de Alencar

Relatório de Estágio

Campina Grande, Brasil

Dezembro de 2019

Allender Vilar de Alencar

Relatório de Estágio

Relatório de Estágio Supervisionado submetido à Coordenação de Graduação em Engenharia Elétrica da Universidade Federal de Campina Grande como parte dos requisitos necessários para a obtenção do grau de Engenheiro Eletricista.

Universidade Federal de Campina Grande - UFCG

Centro de Engenharia Elétrica e Informática - CEEI

Departamento de Engenharia Elétrica - DEE

Coordenação de Graduação em Engenharia Elétrica - CGEE

Orientador: Marcos Ricardo Alcântara Morais

Campina Grande, Brasil

Dezembro de 2019

Agradecimentos

Agradeço primeiramente aos cientistas do passado, porque sem os seus estudos insanos de anatomia, possivelmente, não estaria aqui para realizar esta tese e o meu frágil vínculo de vida estaria estilhaçado no universo.

À minha mãe, Ana, e aos meus irmãos, Alluska, Aryadne e Aristótenes, pelo apoio e ajuda em grande parte da minha vida. A todos os membros da minha família, seja de sangue, ou não, e em especial, à minha tia e meus avós por parte de mãe que deram bastante suporte ao longo da vida.

A José Vinicius, Erivan e todos aqueles que me ajudaram durante a graduação à permanecer firme frente aos problemas, resolvendo alguma questão ou ensinando-me algum assunto.

Aos meus amigos que desferem as críticas mais incisivas, ajudando-me assim, a ter não somente uma nova visão sobre as coisas, mas também, formas de como me aprimorar e me entender.

Ao professores do curso de Engenharia Elétrica que cada qual com sua convicção luta firme e com muito amor ao curso para aprimorá-lo e ajudar os nossos queridos estudantes. Em especial, ao meu oriantador, Marcos, por ser uma pessoa muito incrível e inteligente, sendo assim alguém quem me inspiro.

A todos os que tornam a dor existencial mais suave e leve, como um doce, que enquanto dura, tem textura indescritível de paz e amor, e no fim, um tom amargo-cintilante.

"Tu, amante da verdade, não fiques enciumado dos que são absolutos e impacientes! Nunca a verdade andou de braço dado com um ser absoluto"

Friedrich Nietzsche

Resumo

Com o avanço da tecnologia e o limite físico que pode alcançar o tamanho dos transistores dentro de um chip, a lei de Moore começa a sofrer uma diminuição e a busca por outras formas de desempenho voltam a ter mais força. Neste cenário, temos o coprocessador vetorial, que pode ser utilizado em conjunto com um processador para aumentar a capacidade de processamento, com uma boa efficiência energética.

Palavras-chave: Moore, circuitos, Nanotecnologia, Coprocessador Vetorial, RISCV.

Abstract

With the advance of technology and the physical limit of transistor can reach, the Moore law start to lose the growth and researchs for another form of perfomance back with strength. In this scenario, we have the vector processor, which can be used together with a processor to improve the processing capacity, with a good energy effiency.

Keywords: Moore, Circuits, Nanotechnology, Vector Processor, RISCV.

Lista de ilustrações

Lista de abreviaturas e siglas

HPC	*High-performance computing*
VMPIS	*MIPS vector extension*
ISA	*Instruction set architecture*
MVL	*Maximum vector length*
AXI	*Advanced eXtensible Interface*
SLDU	*Slide Unit*
RoCC	*Rocket Custom Coprocessor*

Sumário

1 Introdução

O primeiro computador foi construído em *Iowa State College* no ano de 1942. Esse computador era composto por poucas poucas peças, mas, com o avanço das tecnologias foi possível torná-los mais complexos, menores e eficientes devido a pesquisas realizadas na área (JúNIOR, 2012).

Segundo (West; Harris, 2011), Os inventores do transistor ganharam o prêmio nobel da física no ano de 1956, sendo eles, Bardeen, Brattain e seus supervisor Willian Shockley. Essa que foi a invenção que revolucionou a forma de computação de dados.

Com a invenção do transistor, temos uma industria que busca conseguir instrumentar tamanhos cada vez menores de transistores no silício, pois acabam se tornando cada vez menos custosas as unidades. Os dois tipos principais de transistores são: junção bipolar (*bipolar junction transistors*) e metal-oxido-semicondutor (*metal-oxide-semiconductor field effect transistors*).(Harris; Harris, 2013)

Na atualidade, temos bilhões de transistores em um único chip, isso se deve à diminuição extremamente rápida das dimensões do transistor e de criação de técnicas ao longo dos anos, possibilitando assim, uma inovação na humanidade, pois, pode-se alterar a microarquitetura e com isso inserir novas possibilidades de computar dados.(Patterson; Hennessy, 2017)

Uma dessas possibilidades de computação é a *single instruction*, *mutiple data* (*SIMD*), que significa que em uma instrução o processador ou coprocessador irá realizar operações em uma certa quantidade de dados tendo um certo nível de paralelismo, dessa forma não irá a cada operação necessitar de uma instrução diferente.(Hennessy; Petterson, 2012) E com isso, não fará *fetch* e *decode* de várias instruções, diminuindo assim, o gasto energético, sendo mais interessante em sistema *low power* como por exemplo, dispositivos celulares. (Cavalcante et al., 2019)

1.1 *Embedded* Lab.

A empresa concedente do estágio foi a própria Universidade Federal de Campina Grande, realizado em convênio com o laboratório do *Embedded* intitulado *XMEN* e tem por objetivo produzir chips e elevar o reconhecimento que a microeletrônica tem tanto da universidade quanto do país. O laboratório fica localizado conforme a imagem abaixo.

Figura 1 – Localização do laboratório.

Fonte: Próprio Autor.

Os projetos são focados desde o *Design* até o *Backend*, ou seja, toda a descrição do chip a ser enviada para a *Foundry*.

1.2 Objetivos

O objetivo deste estágio quando iniciou era somente simular modelos RISCV com extensão vetorial mas foi com o tempo se moldando e foi realizado com parceria do laboratório *Embedded* estão listados a seguir as principais atividades:

- Aprender sobre o funcionamento de instruções vetoriais e dos coprocessadores vetoriais;
- Compilar e simular instruções vetoriais para RISCV com extensão 0.7.1;
- Buscar implementações existentes e seus desempenhos.

2 Revisão bibliográfica

Esta seção apresenta os tópicos teóricos mais importantes sobre o funcionamento dos coprocessadores vetoriais e será baseada em essência na teoria do livro (Hennessy; Petterson, 2012) e na documentação da extensão vetorial RISCV (ASANOVIC et al., 2019).

Arquiteturas vetoriais pegam elementos de dados da memória, alocam esses elementos nos registradores internos, operam com um paralelismo configurável, e após isso, salvam os elementos que foram operados de volta na memória principal do sistema, na condição de que não serão utilizados nas próximas operações. Uma única intrução é capaz de realizar operações com vários elementos da memória.

Várias técnicas são implementadas para esconder a latência e varia com a largura de banda da memória do sistema, sendo possível que o compilador utilize os registradores internos como buffer para evitar ao máxima ter que salvar na memória principal do sistema, devido a latência.

Como a ideia das arquiteturas vetoriais são similares, nesta revisão bibliográfica será apresentada a teoria do VMIPS.

2.1 VMIPS

VMIPS é a extensão vetorial para a porção escalar MIPS, consiste assim, do conjunto de intruções da arquitetura (*ISA*).

Os principais componentes do conjunto de instruções do VMIPS são os seguintes:

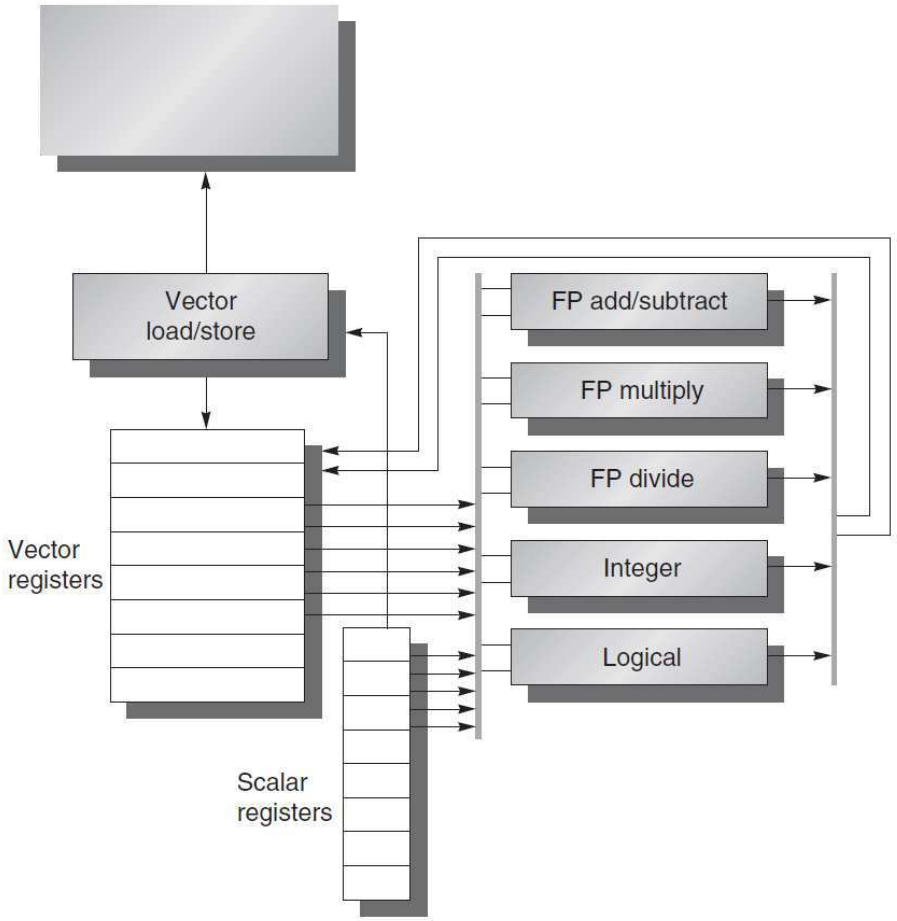
- Registrador vetorial - Os elementos dos registradores têm tamanho variáveis. O VMIPS tem oito registradores desse tipo, sendo que cada registrador vetorial contém 64 elementos e cada elemento é de 64 bits de largura. com ligação *crossbar* temos a conexão entre os registradores e as unidades funcionais vetoriais.

- Unidade funcional vetorial - O VMIPS contém cinco unidades funcionais, elas são totalmente em *pipeline* e a cada ciclo pode-se começar uma nova operação na unidade, também têm unidades de controle para detecção de *harzards*. Na figura 7

- Unidade de carregar/salvar vetor - Esta unidade carrega ou salva elementos dos vetores na memória, sendo totalmente em pipeline, consegue a cada ciclo de clock transferir uma palavra, após uma latência inicial.

- Conjunto de registradores escalares - Pode ser usado como entrada para computar dados (por exemplo o SAXPY), como também para acessar elementos da memória, ou seja, para alcançar endereços.

Figura 2 – Estrutura da arquitetura vetorial VMIPS.



Fonte: (Hennessy; Petterson, 2012).

2.2 Funcionamento de um coprocessador vetorial

Nesta seção será abordada um exemplo de SAXPY ou DAXPY, essa sigla têm significados matemáticos iguais, a única diferença é que na primeira a procisão é singular e na segunda a precisão é dupla. Essas siglas representam o seguinte cálculo:

$$Y = a * X + Y \tag{2.1}$$

ou seja, será somado o vetor **Y** com o vetor **X** escalonado com o valor da constante a. Ou seja, por fim, SAXPY é a vezes x mais y. O laço a ser executado está descrito no código em assembly abaixo.

Figura 3 – DAXPY no VMIPS.

```
        L.D       F0,a          ;load scalar a
        DADDIU    R4,Rx,#512    ;last address to load
Loop:   L.D       F2,0(Rx)      ;load X[i]
        MUL.D     F2,F2,F0      ;a × X[i]
        L.D       F4,0(Ry)      ;load Y[i]
        ADD.D     F4,F4,F2      ;a × X[i] + Y[i]
        S.D       F4,9(Ry)      ;store into Y[i]
        DADDIU    Rx,Rx,#8      ;increment index to X
        DADDIU    Ry,Ry,#8      ;increment index to Y
        DSUBU     R20,R4,Rx     ;compute bound
        BNEZ      R20,Loop      ;check if done
```

Here is the VMIPS code for DAXPY.

```
        L.D       F0,a          ;load scalar a
        LV        V1,Rx         ;load vector X
        MULVS.D   V2,V1,F0      ;vector-scalar multiply
        LV        V3,Ry         ;load vector Y
        ADDVV.D   V4,V2,V3      ;add
        SV        V4,Ry         ;store the result
```

Fonte: (Hennessy; Petterson, 2012).

Para RISCV é possível encontrar vários exemplos no próprio documento, esse que é disponibilizado no github específico de documentação: https://github.com/riscv/riscv-v-spec/blob/master/v-spec.adoc. Neste documento é possível encontrar a estrutura do SAXPY:

Como existe uma forma específica e geral de escrita do laço para cálculo com vetores, será possível que os compiladores adquiram com o tempo essa interpretação direto do .c, não precisando escrever o assembly para executar as funcionalidades vetoriais.

Figura 4 – SAXPY no RISCV.

```
# void
# saxpy(size_t n, const float a, const float *x, float *y)
# {
#   size_t i;
#   for (i=0; i<n; i++)
#     y[i] = a * x[i] + y[i];
# }
#
# register arguments:
#     a0      n
#     fa0     a
#     a1      x
#     a2      y

saxpy:
    vsetvli a4, a0, e32, m8
    vlw.v v0, (a1)
    sub a0, a0, a4
    slli a4, a4, 2
    add a1, a1, a4
    vlw.v v8, (a2)
    vfmacc.vf v8, fa0, v0
    vsw.v v8, (a2)
    add a2, a2, a4
    bnez a0, saxpy
    ret
```

Fonte: (ASANOVIC et al., 2019).

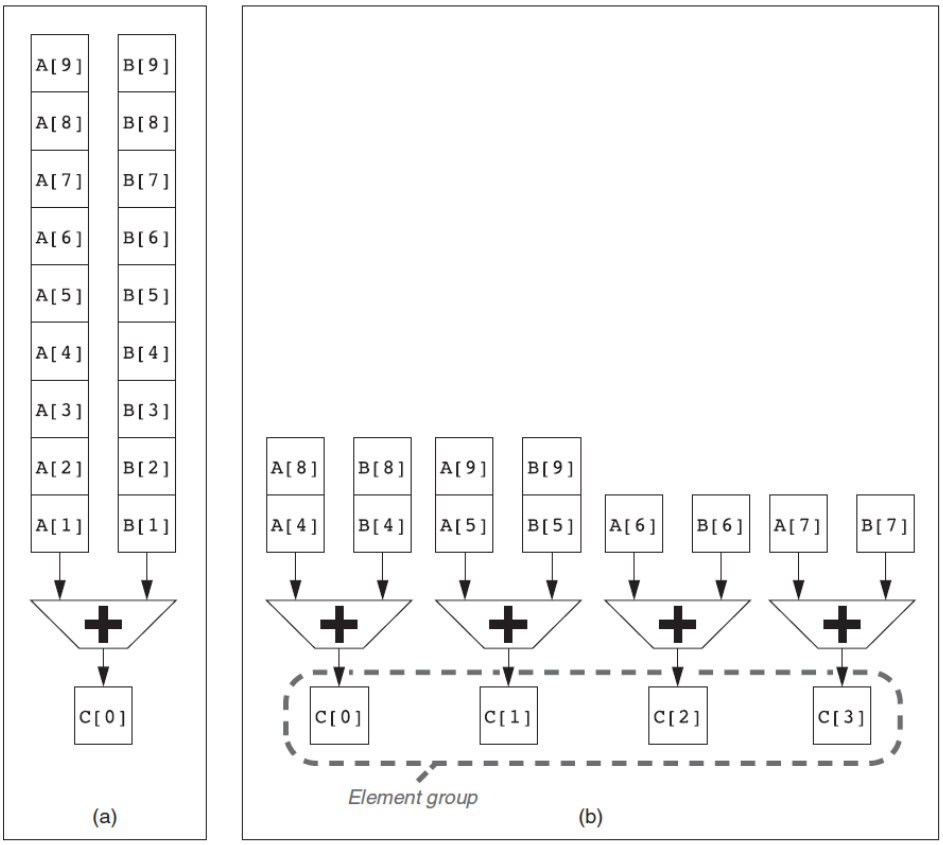
Com esse código, fica evidente que há uma grande redução na transferência de instruções, pois, uma única instrução vetorial é equivalente a várias instruções em processadores escalares e com isso há uma diminuição no gasto energético, pois ocorrem menos *fetch* e *decode*.

2.3 Múltiplas *lanes*

Em uma curta instrução é possível executar mais de uma operação, isso se deve ao fato de que nas arquiteturas vetoriais existem as chamadas *lanes* e sistemas de controle próprios, elas consistem em pistas as quais são possíveis realizar operações a cada ciclo de

clock, ou seja, em pipeline e cada lane é independente das outras, ou seja, o paralelismo é a nível de dados. No caso dessas pistas que são para executar soma (somadores), podemos ter o seguinte exemplo ilustrado na figura abaixo.

Figura 5 – *Lanes* de somadores



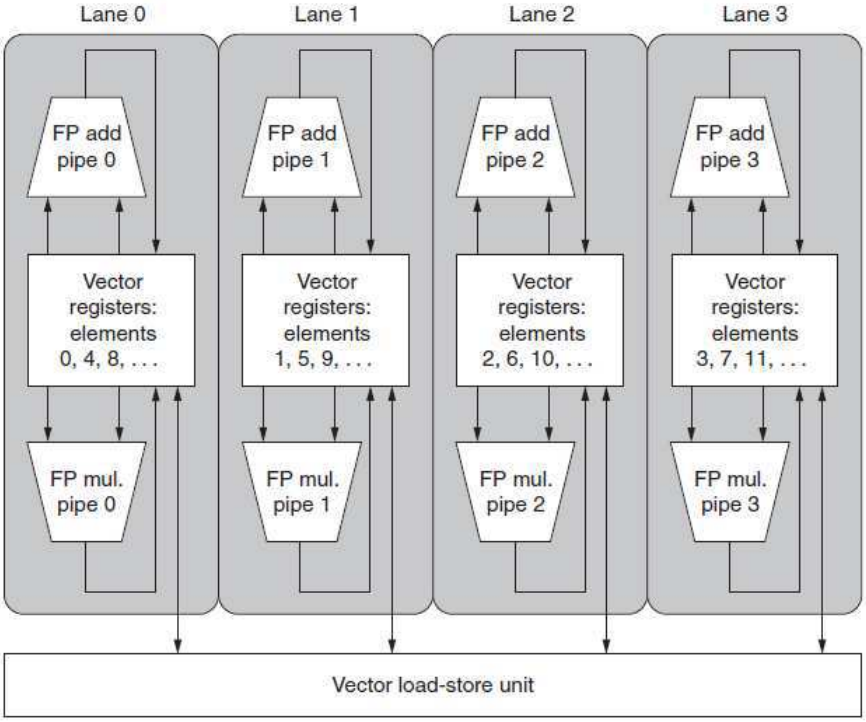
Fonte: (Hennessy; Petterson, 2012).

Dessa figura 5, temos a comparação de (a) para (b), no processador escalar (a), temos nove ciclos para completar a tarefa das somas, já no caso do processador vetorial (b), temos a possibilidade de paralelismo a nível de dados, dessa forma só precisariam de dois ciclos para completar a operação. Dessa forma, a quantidade de *Lanes* está diretamente associado à aceleração e quantidade de operações em pontos flutuantes, e normalmente é limitado pela largura de banda da memória, pois, a quantidade de dados fluindo é muito grande e pode gerar gargalo.

A estrutura de uma unidade vetorial leva consigo não somente um somador mas também um múltiplicador como mostra a figura 6.

A divisão dos registradores são feitos sequencialmente e ciclicamante no espaço, ou seja, varia de 0 até 3, após isso reinicia a contagem de 4 até 7, assim sucessivamente, até

Figura 6 – Estrutura unidade vetorial



Fonte: (Hennessy; Petterson, 2012).

completar a alocação de todos os registradores vetoriais. Há necessidade de um controle mais robusto para resolver os Harzards, evitando assim erros de sobrescrever o registrador e perda de dados. Além do somador e do multiplicador, tem também a unidade de carregar/salvar, que também tem que ter um controle. Essa alocação permite a execução de cálculos sem precisar comunicar com as outras faixas, entretanto, caso seja requerida pode ser feita.

Após ter a estrutura pronta e o controle, aumentar a quantidade de faixas é uma das formas de melhorar desempenho sem aumento considerável de dificuldade, como são independentes só precisam de um certo controle e podem ser parametrizados e gerados. Entretanto, deve-se observar o *Trade off* a ser feito para tal.

2.4 Lidando com vetores com quantidade de elementos não múltiplas de 64

Geralmente a quantidade de elementos dos vetores não serão iguais a quantidade de elementos ou tamanho do registrador vetorial, assim para lidar com essa diferença deve-se

realizar uma técnica descrita a seguir. Se tivermos um vetor da forma:

$$for(i = 0; i < n; i = i + 1)$$
$$Y[i] = a * X[i] + Y[i];$$

Como o laço varia de 0 até n-1, temos que está intrinsicamente ligado a n, o tamanho do vetor. E isso deve ser realizada o *strip mining*, que nada mais é que dividir o tamanho total do vetor a ser calculado em ralação ao tamanho dos vetores utilizados na microarquitetura, nesta seção está descrita para a arquitetura VMIPS e depois RISCV. Primeiramente temos o MVL (*Maximum vector length*), esse determina o tamanho máximo do vetor, ou seja, a quantidade de elementos (Note que pode variar dependendo da precisão utilizada, nesse VMIPS temos 64*64, em cada registrador). A técnica basicamente é dividir n por MVL e realizar a parte inteira normalmente e realizar a parte fracionária sem levar em consideração o tamanho padrão registrador vetorial, somente calculando o necessário, para poupar energia, pois poderia simplesmente encher de zeros e calcular, descartando depois os valores não utilizáveis.

Figura 7 – *strip mining* VMIPS

```
low = 0;
VL = (n % MVL); /*find odd-size piece using modulo op % */
for (j = 0; j <= (n/MVL);  j=j+1) { /*outer loop*/
    for (i = low; i < (low+VL);  i=i+1) /*runs for length VL*/
        Y[i] = a * X[i] + Y[i] ; /*main operation*/
     low = low + VL; /*start of next vector*/
     VL = MVL; /*reset the length to maximum vector length*/
}
```

Fonte: (Hennessy; Petterson, 2012).

(n % MVL), calcula o tamanho da parte fracionária, ou seja, de quantas vezes serão chamadas as operações de DAXPY da parte fracionária do *strip mining*, após isso, temos que ele reseta o tamanho do vetor "VL"e daí em diante serão todos do tamanho "MVL", ou seja, a parte inteira da divisão (n/MVL), e assim, esse laço trata da diferença de tamanhos no VMIPS. Exemplo, se tivermos um vetor de tamanho 135, temos que 135 % 64 = 7, então esse laço iria calcular primeiro esses 7 e depois faria duas vezes o laço de 64, completando o vetor, nesse caso o compilador reconhece e consegue otimizar e utilizar o coprocessador vetorial.

Na figura 8, temos um exemplo para registradores de tamanho VLEN = 256b da extensão vetorial do RISCV. Dessa forma a sua divisão por SEW dará a quantidade de

Figura 8 – *strip mining* RISCV

```
VLEN=256b, SLEN=128b
Byte     1F1E1D1C1B1A19181716151413121110 F E D C B A 9 8 7 6 5 4 3 2 1 0

SEW=8b, LMUL=1, VLMAX=32

v1       1F1E1D1C1B1A19181716151413121110 F E D C B A 9 8 7 6 5 4 3 2 1 0

SEW=16b, LMUL=2, VLMAX=32

v2*n       17  16  15  14  13  12  11  10   7   6   5   4   3   2   1   0
v2*n+1     1F  1E  1D  1C  1B  1A  19  18   F   E   D   C   B   A   9   8

SEW=32b, LMUL=4, VLMAX=32

v4*n           13      12      11      10       3       2       1       0
v4*n+1         17      16      15      14       7       6       5       4
v4*n+2         1B      1A      19      18       B       A       9       8
v4*n+3         1F      1E      1D      1C       F       E       D       C

SEW=64b, LMUL=8, VLMAX=32

v8*n                   11              10               1               0
v8*n+1                 13              12               3               2
v8*n+2                 15              14               5               4
v8*n+3                 17              16               7               6
v8*n+4                 19              18               9               8
v8*n+5                 1B              1A               B               A
v8*n+6                 1D              1C               D               C
v8*n+7                 1F              1E               F               E
```

Fonte: (ASANOVIC et al., 2019).

elementos por registrador vetorial, ou seja, no caso SEW = 8b, temos 32 elementos e sendo o LMUL o agrupamento de quantos vetores podem ser trabalhados no momento e seu valor igual a 1, então pela figura abaixo, então pode-se trabalhar os 32 elementos do grupo. No caso de SEW = 32 b, temos 8 elementos por registrador e sendo o LMUL 4, pode-se trabalhar 8 elementos ao mesmo tempo. Dessa forma o *strip mining* fica dependente de todas essas variáveis, e quando os registradores forem configurados terão a possibilidade de irem calculando uma certa quantidade a cada ciclo de clock, assim, o laço antes feito em VMIPS também servirá para uma microarquitetura RISCV, entretanto, a configuração terá de ser feita manualmente, dado que ainda não há compiladores contendo essas otimizações para RISCV.

No caso do *strip mining* de elementos de 64 bits, temos nessa microarquitetura menor quatro elementos por registrador, sendo assim, necessário uma maior quantidade de ciclos de clock parecida devido a quantidade de *Lanes* fornecidas, ou seja, caso a quantidade de *Lanes* do VMIPS fosse maior, poderia calcular até 64 elementos em um único ciclo de clock, acelerando muito o processo em relação a somente 4 *Lanes*.

Figura 9 – LMUL RISCV

vlmul		LMUL	#groups	VLMAX	Grouped registers
0	0	1	32	VLEN/SEW	vn (no group)
0	1	2	16	2*VLEN/SEW	vn, vn+1
1	0	4	8	4*VLEN/SEW	vn, ..., vn+3
1	1	8	4	8*VLEN/SEW	vn, ..., vn+7

Fonte: (ASANOVIC et al., 2019).

3 Atividades desenvolvidas

Baseado nas atividades propostas no termo de estágio e algumas sugestões do orientador, as próximas seções descrevem o que foi realizado no estágio.

3.1 Estudo das arquiteturas vetoriais

No início do estágio teve um estudo mais aprofundado para o entendimento do funcionamento de algumas microarquiteturas e arquiteturas vetoriais. O livro *Computer Architecture A Quantitative Approach* foi a base para entendimento das arquiteturas, após o capítulo de paralelismo a nível de dados do mesmo, foi estudada a a descrição da extensão vetorial RISCV, juntamente com os *papers* de descrição do *ARA* e o *Hwacha* (microarquiteturas RISCV que têm certa documentação aberta). Parte desse estudo já foi descrito na revisão bibliográfica.

3.2 Simulação de instruções vetoriais

Ao buscar um simulador de instruções vetoriais, primeiramente o que conseguia executar a extensão vetorial 0.7.1 foi o OVPSIM, mas devido a falta de materiais que explanassem sobre a compilação das intruções vetoriais, e até mesmo no próprio *Github* da OVPSIM ter uma nota dizendo que devido a complexidade da compilação só foi descrito que foi utilizando alguns compiladores cruzados e *binutils*, realizei os passos que estavam descritos no documento oficial do simulador e não obtive nenhum resultado sobre as instruções específicas, mesmo instalando alguns dos pacotes disponibilizados por eles no site e buscando por várias fontes diferente não foi possível realizar em primeira mão essa simulação. Por fim, ao perguntar diretamente no fórum da OVPSIM também obtive um redirecionamento e não foi explicado novamente o processo, assim, após um certo tempo ficou decidido que não seria mais utilizado esse simulador, o que resultado no uso do *spike*.

Como havia possibilidade de utilizar um ramo do *RISCV Binutils* para executar a compilação das intruções vetoriais em assembly (Note que elas são muito complexas e no momento os compiladores não conseguem extrair de um código C para assembly intruções vetoriais RISCV), e com várias semanas de pesquisas e tentativas foi possível compilar do assembly e executar instruções vetoriais riscv 0.7.1. Abaixo segue um breve tutorial, feito pelo engenheiro de compilação kito-cheng, de como instalar o compilador, e depois tem um tutorial de como executar um exemplo.

Uma das formas é usando o compilador cruzado disponível no github com os seguintes passos:

- git clone https://github.com/riscv/riscv-gnu-toolchain -b rvv-0.7.x
- cd riscv-gnu-toolchain-rvv
- git submodule update –init –recursive
- mkdir build cd build
- export RISCV=/path/to/install
- ../configure –prefix=$RISCV
- make
- make install

Após isso será necessário instalar o RISCV-pk, seguem os passos:

- mkdir build
- cd build
- ../configure –prefix=$RISCV –host=riscv64-unknown-elf
- make
- make install

Por fim, deve-se instalar o simulador (spike):

- https://github.com/riscv/riscv-isa-sim.git
- cd riscv-isa-sim

- apt-get install device-tree-compiler
- mkdir build
- cd build
- ../configure –prefix=$RISCV
- make

sudo make install

uma forma simples de testar o funcionamento do simulador é escrever um alô mundo em .c e depois compilá-lo:

- riscv64-unknown-elf-gcc -o hello hello.c
- spike pk hello

Agora pode-se escrever códigos em assembly utilizando a extensão vetorial 0.7.1 e compilá-los um bom exemplo é um arquivo pronto com o linker para executar diretamente, servindo de base para escrever algum código, são disponibilizados pelo *Institute for System Programming, Russian Academy of Sciences* no link: https://github.com/ispras/riscv-avs/tree/master/testsuite/synthetics/rv32v. Para rodar:

- git clone https://github.com/ispras/riscv-avs.git
- cd riscv-avs/testsuite/synthetics/rv32v
- riscv64-unknown-elf-gcc -march=rv64gcv -nostdlib -nostartfiles \
- -Trv32v_simple_selfcheck_0000.ld \
- -o rv32v_simple_selfcheck_0000.o rv32v_simple_selfcheck_0000.s
- spike –isa=RV64IMAFDCV -l -p1 rv32v_simple_selfcheck_0000.o

Atentar-se aos nomes dos arquivos, pois podem ter sido alterados em alguma atualização no momento em que esse documento esteja sendo lido.

Com isso, fica possível realizar simulação de instruções em alto nível da última versão estável da extensão vetorial (0.7.1).

3.3 Microarquiteturas vetoriais

Faltando pouco tempo para o término do estágio foi que começou a buscar as implementações, como a única disponível em código aberto é a hwacha, essa que foi a única que pôde ser estudada um pouco mais profundamente, entretanto, não foi possível em tempo hábil simular a mesma e observar a efficiência, entretanto na documentação consta tais dados. Será dada uma breve visão geral das microarquiteturas já documentadas, utilizando as próprias documentações como referências.

3.4 Ara

Segundo (Cavalcante et al., 2019), essa microarquitetura foi implementada na *GLOBALFOUNDRIES* em tecnologia de 22FDX FD-SOI alcançando um desempenho de 33 DP-GFLOPS (Giga operações em pontos flutuantes de precisão dupla) e uma eficiência energética de 41 DP-GFLOPS/W, nas condições de *Tipical corner* (TT/0.80 V/25 C). De fato, Não fica claro a condição de uma quantidade maior de cálculo em pontos flutuantes podem ser realizadas somente aumentando a potência, ou seja, qual seria o *Trade off* e como deveria ser utilizada as áreas de chips para contruir computadores maiores? Essas perguntas poderão ser respondidas em trabalhos futuros. Notoriamente é possível alocar essas microarquiteturas embarcadas com o processador e ajudar no cálculo com um baixo consumo, mas deve-se observar o quanto será necessário e quanto consumirá dentro de uma dada temperatua e tensão de alimentação.

Na figura 10a temos a arquitetura do Ara, ao utilizar o processor escalar Ariane como base, as instruções são despachadas pelo mesmo, na parte do *Ara front end*, ou seja é uma pequena mudança realizada no Ariane para realizar a decodificação da instrução. Uma pequena diferença é que o *Dispacher* funciona de forma especulativa.

O *Sequencer* consegue observar o funcionamento como um todo do coprocessador, tem a função de despachar para diferentes partes do Ara, podendo ter até oito instruções paralelas no mesmo. Também tem a função de resolver os *Harzards*, ou seja, como tem a visão geral do Ara pode comunicar ao Ariane que está cheio ou terá de esperar.

A *SLDU* está conecatado a todo o banco de registradores e tem a função de dada a necessidade de realizar *vector shuffle* ou *vector slide*, realizá-la para poder operar nos novos vetores gerados.

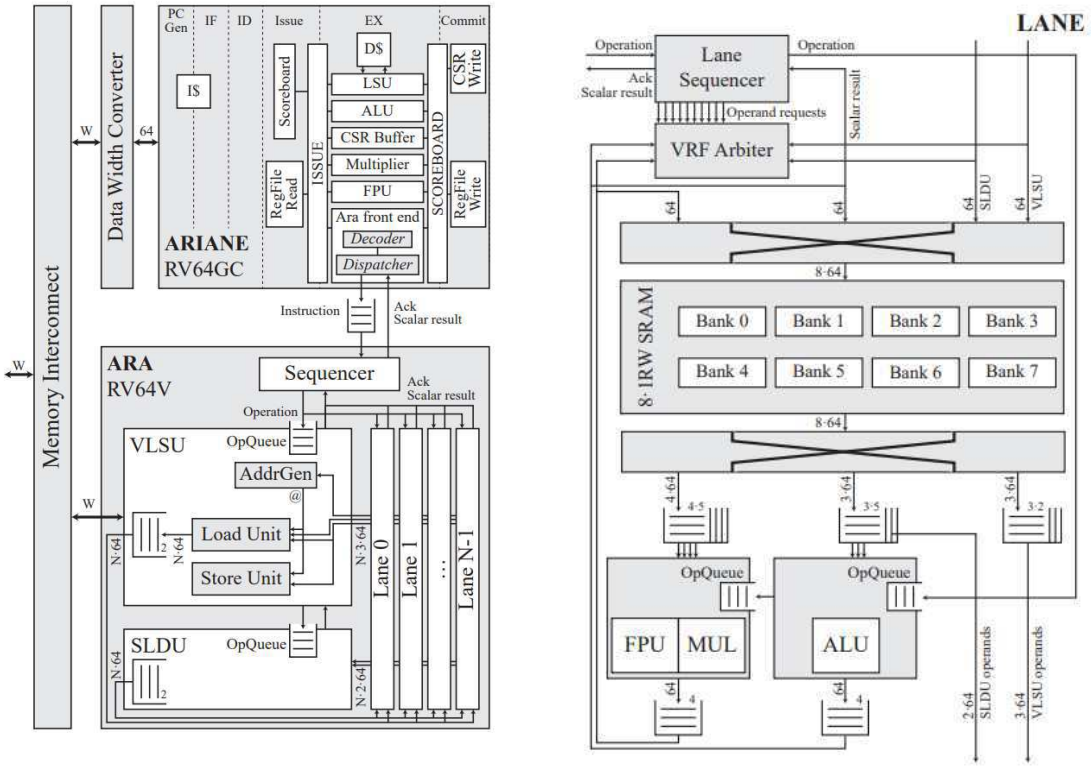
Unidade de carregar/salvar utiliza barramento AXI e tenta manter uma taxa de transmissão de 2 B/DP-FLOP.

As *Lanes* são de formato padrão segundo a figura 10b, dessa forma podem ser

criadas em função do valor de N, ou seja, gera o circuito de forma parametrizada.

Figura 10 – Microarquitetura do Ara.

(a) Microarquitetura do Ara junto com Ariane. (b) *Lanes* do Ara.



Fonte: (Cavalcante et al., 2019).

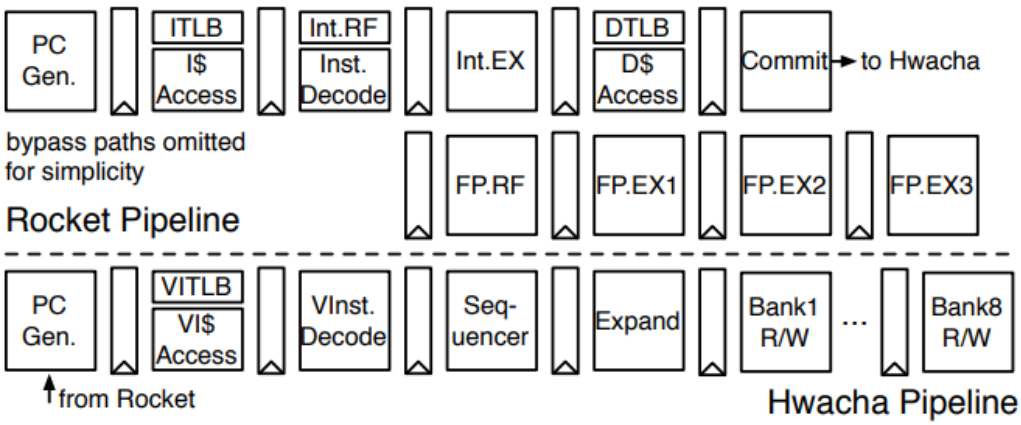
3.5 Hwacha

Esta seção é baseada na publicação (LEE et al., 2015). Hwacha tem uma implementação documentada (LEE; WATERMAN; ASANOVIC, 2014) alcançando 16,7 DP-GFLOPS/W que é cerca de 0,4073 da eficiência energética do Ara, note que a tecnologia de implementação também é diferente (neste caso 45 nm) e funciona a 1,3 GHz em 0,65 V com 3mm de área. Não é possível comparar com o Ara devido a falta de valores de custo, ou seja, mesmo que tenha uma eficiência menor poderia ainda ser mais viável economicamente, além de ser código aberto. o Ara utiliza uma interface de comunicação do processador com o coprocessador parecida com o RoCC interface, neste caso, é a própria RoCC implementada.

na figura 11 temos a *pipeline* do *core* escalar da Rocket juntamente com o coprocessador vetorial, a parte escalar é 64 bits e tem 6 estágios de pipeline e somente ao fim

dela é que envia dados ou instruções para o coprocessador *Hwacha*.

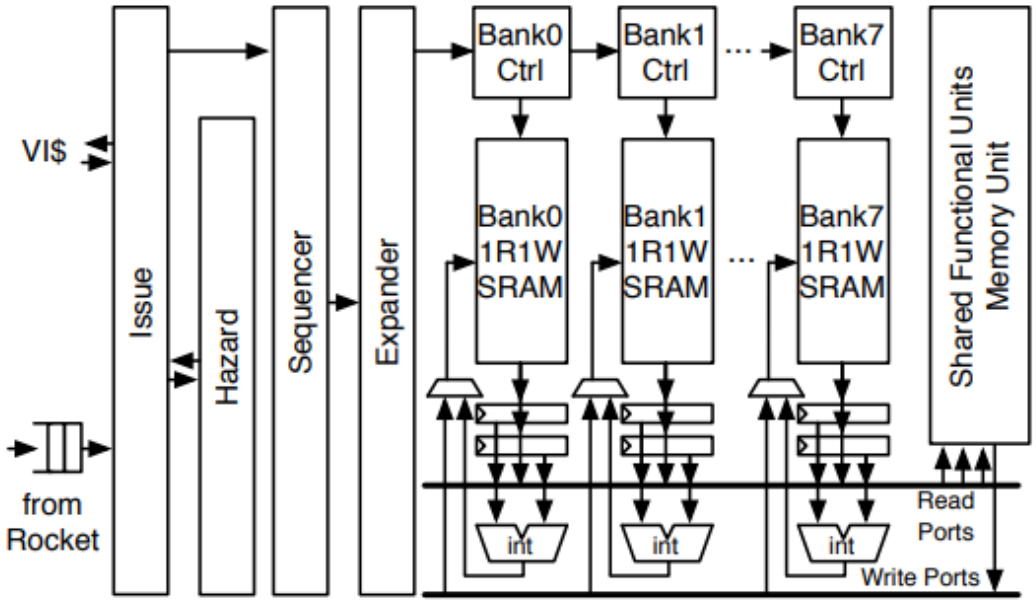
Figura 11 – *Rocket scalar* com *Hwacha*.



Fonte: (LEE; WATERMAN; ASANOVIC, 2014).

A diferença básica da implementação do *Hwacha* em relação ao *Ara* é que o primeiro não tem a quantidade de faixas parametrizado como o segundo (em função do valor de N), mas sim 8 faixas fixas e disso tem uma implementação que tem a metade da quantidade de faixas que tem na implementação do Ara, e essa quantidade de faixas quanto maior, melhor será a eficiência energética, até chegar em um limite que é o do gargalo provocado pela memória principal do sistema.

Figura 12 – *Hwacha* microarquitetura.



Fonte: (LEE; WATERMAN; ASANOVIC, 2014).

4 Considerações finais

O estágio supervisionado ofertado pela Universidade Federal de Campina Grande teve papel fundamental na complementação da formação, pela possibilidade de aprofundar mais nos conhecimentos em microeletrônica e linux, pois conhecimentos mais específicos como variáveis do sistema não são vistos no curso e nessa ocasioão foi possível buscar mais esses conhecimentos mais fechados, mesmo que não vá ser utilizado no futuro.

O orientador teve um forte papel no suporte para completar algumas tarefas, dado aulas, dicas e até mesmo tentando fazer a atividade junto comigo.

Para atividades futuras fica realmente dissecar uma microarquitetura ou implementá-la usando o conjunto de instruções RISCV, para conseguir dessa forma transformar a universidade em referência em microeletrônica, infelizmente não teve tempo suficiente este estágio para a implementação de tal.

O cronograma foi basicamente concluído, pois a ideia geral do estágio era trazer uma forma de avaliar as instruções vetoriais do RISCV e com o compilador funcionando é possível escrever o código em "C"e as partes vetoriais invocar o assembly necessário.

Referências

ASANOVIC, K. et al. *Vector Extension, v0.7.1.* [S.l.]: GitHub, 2019. `https://github.com/riscv/riscv-v-spec/releases/tag/0.7.1`. Citado 4 vezes nas páginas 5, 8, 12 e 13.

Cavalcante, M. et al. Ara: A 1 ghz+ scalable and energy-efficient risc-v vector processor with multi-precision floating point support in 22 nm fd-soi. *ArXiv*, Nov 2019. ISSN 2331-8422. Citado 3 vezes nas páginas 1, 18 e 19.

Harris, D. M.; Harris, S. L. *Projeto Digital e Arquitetura de Computadores.* [S.l.]: Elsevier, Inc., 2013. Citado na página 1.

Hennessy, J. L.; Petterson, D. A. *Computer Architecture A Quantitative Approach.* 5. ed. [S.l.]: Elsevier, Inc., 2012. Citado 7 vezes nas páginas 1, 5, 6, 7, 9, 10 e 11.

JúNIOR, G. G. dos S. *Robust design of deep-submicron digital circuits.* Tese (Doutorado) — Télécom ParisTech, 2012. Citado na página 1.

LEE, Y. et al. *The Hwacha Vector-Fetch Architecture Manual, Version 3.8.1.* [S.l.], 2015. Disponível em: <http://www2.eecs.berkeley.edu/Pubs/TechRpts/2015/EECS-2015-262-.html>. Citado na página 19.

LEE, Y.; WATERMAN, A.; ASANOVIC, A. A 45nm 1.3ghz 16.7 double-precision gflops/w risc-v processor with vector accelerators. *ArXiv*, Oct 2014. Citado 3 vezes nas páginas 19, 20 e 21.

Patterson, D. A.; Hennessy, J. L. *Computer Organization and Design The Hardware/Software Interface: RISC-V Edition.* 1. ed. [S.l.]: Elsevier, Inc., 2017. Citado na página 1.

West, N. H. E.; Harris, D. M. *CMOS VLSI Design A Circuits and Systems Perspective.* 4. ed. [S.l.]: Addison-Wesley, 2011. Citado na página 1.