Universidade Federal do Rio de Janeiro
Escola politécnica
Departamento de Engenharia Eletrônica e de Computação
Disciplina: Arquitetura de Computadores

Prática dos Projetos RISC-V

Marcelo Firmino Ribeiro
Ricardo Amaral de Lasmar Abreu Rei
Umberto Augusto de Almeida Pinto Pereira

Professor: Diego Leonel Cadette Dutra

15 de julho de 2025

Sumário

1 Introdução

Este relatório visa apresentar o que foi feito pelo grupo durante a execução das atividades da Prática dos Projetos RISC-V, o que inclui o Somador Vetorial, o Projeto RISC-V e o Projeto RISC-V com SIMD. Por conveniência, abordaremos cada uma dessas práticas em diferentes capítulos ao longo do relatório, com seções de objetivo, implementação, resultados e conclusões. Ao final desses capítulos, teremos um capítulo final de conclusão, no qual iremos sintetizar os principais aprendizados adquiridos pelo grupo durante a elaboração dessas atividades.

Para a realização de todas as práticas aqui descritas, foi utilizado o simulador Digital [1], conforme recomendado pelo docente no site da disciplina [2]. Nele, criamos módulos VHDL nos quais escrevemos códigos e os testamos com as ferramentas disponíveis. Os arquivos de circuito gerados no simulador (arquivos do tipo .dig) estão disponíveis no GitHub referenciado [3]. O arquivo `SomadorVetorial.dig` é referente à prática AVX - Somador Vetorial (2). Já o arquivo `CPUsemSIMD.dig` é o projeto de uma CPU RISC-V sem instruções vetoriais (3). Por fim, o arquivo `CPUcomSIMD.dig` é a última etapa das práticas RISC-V e corresponde à CPU com instruções vetoriais (4).

2 Atividade Prática AVX - Somador Vetorial

2.1 Objetivo

Esta primeira tarefa tem como objetivo implementar um somador vetorial. O módulo deve receber vetores de 32 bits que podem representar:

- Oito números de 4 bits;
- Quatro números de 8 bits;
- Dois números de 16 bits;
- Um número de 32 bits.

Com essas entradas, o módulo deve ser capaz de somar ou subtrair os números. A seleção da operação (soma ou subtração), assim como a indicação de como está dividido o vetor de bits, ocorre por meio de uma entrada seletora. Sendo assim, o módulo deve possuir 4 entradas:

- 2 entradas de 32 bits, sendo os vetores de bits;
- 1 entrada de 1 bit que seleciona a operação;
- 1 entrada de 2 bits que indica como os vetores de bits estão divididos.

Como saída, esse módulo deve possuir apenas o resultado da operação, um vetor de bits que estará dividido da mesma forma que os vetores de bits da entrada.

2.2 Implementação

Para a implementação desse módulo, nós nos inspiramos nos circuitos integrados 74S181 e 74S182. O primeiro é uma ALU que realiza operações com números de 4 bits e o segundo computa os carry-in's para que seja possível interligar várias ALU's, de forma a possibilitar somas com números de mais bits. Ambos os CIs fazem o cálculo dos carries com a técnica de carry-lookahead, o que é essencial quando se busca um bom desempenho.

Dessa forma, antes de começar a escrever nosso código VHDL, estudamos o funcionamento desses dois circuitos integrados e analisamos formas de implementar a mesma lógica.

Então criamos, inicialmente, duas entidades diferentes, cada uma com o objetivo de reproduzir o funcionamento de um dos circuitos integrados citados. Em outras palavras, criamos uma entidade capaz de realizar uma soma para números de 4 bits - aqui chamada de ALU -, que utiliza a técnica de carry-lookahead e também fornece saídas G (gerador de carry) e P (propagador de carry), que são usadas pela outra entidade para permitir a interligação de mais de uma ALU. A segunda entidade, como dito, computa os carry-in's de cada ALU a partir de um carry-in e dos sinais G e P.

Com isso feito, criamos uma terceira entidade, responsável por receber as entradas, interligar os módulos necessários das outras duas entidades e fornecer as saídas.

Deve-se ressaltar que os carry-in's de cada ALU são sempre calculados como se as entradas representassem um número de 32 bits cada. Entretanto, uma pequena lógica combinacional é usada para que as ALUs só recebam carry-in's quando realmente for necessário.

Antes de falar quais são os casos em que os carry-in's serão necessários em cada ALU, vamos explicar como fizemos a subtração. Caso seja ela a operação escolhida, devemos fazer $A - B$, o que equivale a fazer $A + (-B)$.

Portanto, para a subtração, primeiro encontramos o complemento a 2 dos números da entrada B. Isso pode parecer um pouco difícil, tendo em vista que a entrada B pode representar diferentes quantidades de números. Porém, há uma forma simples de se calcular o complemento a dois de um número quando se quer fazer uma soma com ele:

1. Inverte todos os bits desse número;

2. Faz-se o carry-in do módulo somador ser igual a 1.

Dessa forma, os dois passos para se chegar ao complemento a dois de um número são realizados em apenas 1 tempo de porta extra (uma porta NOT). Aqui, devemos esclarecer que a inversão dos bits de B é realizada na entidade principal, a terceira citada nesta seção.

Como a entrada B pode representar diferentes quantidades de números, precisamos de uma pequena lógica combinacional para verificar quais somadores precisarão de um carry-in no caso de uma subtração. Por exemplo, se formos fazer uma subtração e os vetores de bits de entrada estiverem representando 8 números de 4 bits, todas as ALU's precisarão que seu carry-in seja 1.

Com isso exposto, agora podemos evidenciar quais são os casos em que a entrada carry-in de uma ALU deve estar em nível lógico alto:

1. A ALU está computando a soma de bits mais significativos de um número e os bits menos significativos geraram um carry-out;

2. A ALU está computando a subtração dos bits menos significativos de um número.

Percebe-se que ambas as situações são excludentes, pois, no segundo caso, a ALU deve estar realizando a operação sobre os bits menos significativos e, no primeiro caso, não.

A partir desses pensamentos, criamos um único código em VHDL - o que significa um único bloco no simulador Digital - que correspondesse a tudo que planejamos. O arquivo de circuito que contém o módulo com o código VHDL encontra-se no arquivo "SomadorVetorial.dig", presente no GitHub [3].

2.3 Resultados

Após adicionarmos entradas e saídas no simulador, pudemos realizar alguns testes no nosso código. Todos os testes tiveram resultados de acordo com o esperado, tanto para variações na seleção de operação quanto para mudanças na indicação do tamanho do vetor.

2.4 Conclusões

Essa prática permitiu que os integrantes do grupo entendessem melhor as operações vetoriais, assim como os desafios por trás de sua implementação de forma eficaz. Sendo assim, esse trabalho trouxe grande aprendizado para o grupo, auxiliando na compreensão de aspectos teóricos importantes da disciplina e também na familiarização com aspectos práticos e simulações.

3 Projeto de CPU RISC-V

3.1 Objetivo

Nesta tarefa, temos como objetivo fazer a simulação de uma CPU RISC-V com microarquitetura baseada em uma pipeline de 5 estágios. A CPU deve ser capaz de processar as seguintes instruções: add, addi, auipc, sub, and, andi, or, ori, xor, xori, sll, slli, srl, srli, lw, lui e sw, jal, jalr, beq e bne. Para isso, deve ser usado o simulador Digital [1], criando blocos de código VHDL e usando alguns outros blocos (em especial os de memória) que estejam disponíveis no simulador.

3.2 Implementação

Para realizar esse trabalho, nos baseamos na pipeline explicada ao longo do capítulo 4 do livro-texto da disciplina [4] e fizemos alguns acréscimos para que nossa CPU pudesse processar todas as instruções requisitadas. Sendo assim, usamos os mesmos 5 estágios do livro: Busca de Instrução (IF), Decodificação de Instrução (ID), Execução da Operação (EX), Acesso à Memória (MEM) e Escrita no Register File (WB), mas com algumas alterações.

Entre 2 estágios consecutivos, implementamos um registrador de pipeline para armazenar as informações necessárias de um estágio para os estágios seguintes. Cada um será chamado de "registrador de pipeline A/B"onde A

é a sigla do estágio anterior e B é a sigla do estágio seguinte. Por exemplo, entre os estágios Busca de Instrução e Decodificação da Instrução, está o registrador de pipeline IF/ID.

Antes de explicarmos os estágios e seus componentes com mais detalhes, vamos ressaltar algumas características globais da nossa CPU. As memórias de dados e de instruções são endereçadas apenas com 16 bits. Com isso, não temos memórias grandes demais a ponto de atrapalhar a simulação. Entretanto, a manipulação de endereços (operações aritméticas e/ou escrita/leitura em registradores) é feita com 32 bits, dos quais os 16 menos significativos são usados como endereço nas memórias de dados e de instruções.

Ainda sobre as memórias, devemos ressaltar que os endereços da nossa CPU são endereços de palavras (word address), e não endereços de bytes. Optamos por fazer dessa maneira para facilitar o uso do bloco ROM do simulador Digital [1] (nossa memória de instruções do estágio IF (3.2.2)), que fornece apenas 1 dado na sua saída (e, no caso das instruções, esse dado possui 32 bits). Para manter a uniformidade no circuito, o endereço da memória RAM, no estágio MEM (3.2.4), também é um word address.

Os ciclos de clock foram definidos de forma que, na primeira metade, o clock estará em nível lógico alto. Ou seja, no momento em que há uma borda de subida do clock, os registradores de pipeline mudam seus valores e um novo ciclo se inicia. A escrita nos registradores do register file e na memória de dados ocorre, então, no momento em que há uma borda de descida do clock. Em outras palavras, durante o semi-ciclo inicial do ciclo de clock, os sinais de escrita estão chegando aos seus destinos e, no semi-ciclo final, os sinais de leitura estão saindo de suas origens.

A última característica da nossa CPU que devemos ressaltar está relacionada com a verificação das condições em instruções de branch (`beq` e `bne`). Implementamos uma fowarding unit apenas no estágio EX (3.2.3), de forma que não há fowarding para o estágio ID, onde ocorre a verificação da condição do branch. Sendo assim, para a nossa CPU, é responsabilidade do software (compilador) inserir nops entre uma instrução de branch e sua instrução anterior, caso essa instrução anterior escreva em um registrador que deve ser verificado pelo branch.

As instruções de operações lógicas ou aritméticas foram implementadas como na CPU do livro-texto [4], bem como os branches, o load e o store. As instruções `jalr e jal` serão explicadas na seção referente ao estágio IF (3.2.1). Já as implementações das instruções `auipc e lui` serão detalhadas na seção do estágio ID (3.2.2).

Abaixo, falaremos sobre cada um dos estágios implementados, dando foco nas diferenças em relação ao livro-texto, explicando suas motivações e, principalmente, sua implementação.

3.2.1 Busca de Instrução (IF)

É o estágio onde uma instrução é buscada na memória de instruções para que ela possa ser processada pela CPU.

Esse estágio possui os seguintes componentes:

1. Registrador para guardar o PC atual;
2. Memória ROM para leitura das instruções;
3. Somador de PC+1, para calcular o próximo PC;
4. MUX para selecionar qual o próximo PC;
5. Portas lógicas para implementar stall, flush e seleção do próximo PC.

As lógicas de stall, flush (no registrador de pipeline IF/ID) e seleção de próximo PC foram feitas sem consultar o livro, uma vez que a pipeline do livro não processa as instruções `jal` e `jalr`.

Um sinal `stall` é gerado no estágio ID (3.2.2) e indica que a pipeline não deve andar, ou seja, o PC e o registrador de pipeline IF/ID não podem mudar. Sendo assim, o registrador PC possui uma entrada que recebe diretamente o sinal `stall` para controlar se ele deve ser modificado ou não. Já o registrador de pipeline IF/ID possui uma entrada com o mesmo propósito, mas que recebe um sinal um pouco diferente em decorrência da forma que implementamos a instrução `jalr`, explicada no parágrafo a seguir.

A instrução `jalr`, na nossa CPU, leva 2 ciclos de clock. Optamos por fazer isso pois ela precisa calcular 2 valores: o próximo PC (que é a soma de 2 registradores e, então, precisa da ALU no estágio EX (3.2.3)) e o valor PC+1 (que seria o PC seguinte se não houvesse um salto). Portanto, a instrução `jalr` primeiro realiza e operação de soma dos dois registradores (no estágio EX) e envia esse resultado para o PC. No ciclo seguinte, a ALU realiza a soma PC+1 (e esse valor passará pelos demais estágios e será armazenado em um registrador do register file). Sendo assim, a instrução `jalr` impede que o registrador de pipeline IF/ID mude por 1 ciclo de clock, já que ela permanecerá por dois ciclos no estágio ID.

Portanto, as entradas PC_write e IF/ID_write recebem os sinais:

- PC_write = stall
- IF/ID_write = stall OR JALR

Onde JALR é um sinal que indica que o ID está no primeiro ciclo com a instrução `jalr` e, portanto, no próximo ciclo, o estágio ID ainda deve estar com a instrução `jalr`. Além disso, quando PC_write e/ou IF/ID_write estão em nível lógico alto, não ocorre escrita.

A instrução `jal`, por sua vez, é o mesmo que um branch com a condição sempre verdadeira. Portanto, sua implementação é quase idêntica à de um branch, mas sem a parte de verificação de condição.

Já a entrada IF/ID_flush do registrador de pipeline IF/ID, quando em nível lógico alto, faz com que todos os valores desse registrador sejam zerados na próxima escrita (inserindo um nop). Isso deve ocorrer para saltos, condicionais ou não, uma vez que, enquanto se calcula o próximo PC e a condição do salto (caso haja uma), o estágio IF já buscou uma nova instrução que não deve ser executada caso ocorra um salto. Portanto, sempre que ocorrer um salto (uma instrução de branch com condição verdadeira, uma instrução `jal` ou uma instrução `jalr`), a entrada IF/ID_flush deverá estar em nível lógico alto. Ou seja:

- IF/ID_flush = JALR2cycle OR BranchTaken OR JAL

Onde JAL é um sinal que indica que há uma instrução `jal` no estágio ID, BranchTaken é um sinal que indica se ocorreu um salto condicional e JALR2cycle indica que o estágio ID está, pelo segundo ciclo de clock consecutivo, com uma instrução `jalr`. Cada um desses sinais indica, resumidamente, que está ocorrendo um salto.

Por fim, devemos explicar os sinais de seleção para o próximo PC. Existem 3 opções possíveis: PC+1, um resultado vindo do estágio ID e um resultado vindo do estágio EX. Precisamos, então, de duas chaves de seleção S1 e S2. Caso ambas estejam em nível lógico baixo, o próximo PC será PC+1. Caso S1 esteja em nível lógico alto, o próximo PC será o resultado vindo do estágio EX (caso de `jalr`). E, caso S2 esteja em nível lógico alto, o próximo PC será o resultado vindo do estágio ID (caso de branches e `jal`). Portanto, as chaves de seleção seguem a seguinte lógica:

- S1 = JALR2cycle
- S2 = BranchTaken OR JAL

3.2.2 Decodificação de Instrução (ID)

O estágio de Decodificação de Instrução (ID) é o segundo dos 5 estágios do nosso pipeline. Sua principal responsabilidade é interpretar a instrução que foi buscada no estágio anterior (IF) e preparar todos os dados e sinais de controle necessários para os estágios seguintes.

Este estágio é composto pelos seguintes componentes:

1. Control;
2. Hazard Detection Unit;
3. Mux ControlSignals;
4. FlipFLop JALR 2ndCycle;
5. Immediate Generator;
6. Somador PC + Imediato (PC Plus Imm);
7. MUX para selecionar qual imediato será passado adiante;
8. Register File;
9. Verificador de Registradores (Check Equal).

Dentre os componentes desse estágio, o Control é o componente mais importante e mais complexo, pois ele constitui o núcleo lógico de um processador, operando como um centro de comando que traduz as instruções de máquina em operações concretas no datapath. A implementação em VHDL apresentada para este projeto detalha essa lógica complexa de forma combinacional, onde as saídas são uma função direta das entradas em qualquer instante. As entradas primárias da entidade Control são a própria instrução de 32 bits, Instr, e um sinal de estado específico, JALR 2ndCycle. A partir da instrução, a unidade extrai os campos opcode e funct3, que são os elementos essenciais para a decodificação. O sinal JALR 2ndCycle é uma entrada de estado que informa à unidade de controle que a instrução `jalr` está em seu segundo ciclo de execução, permitindo a geração do conjunto de sinais adequado para completar a operação, uma abordagem de projeto que lida eficientemente com a natureza de dois passos (cálculo do endereço e salto) que definimos para esta instrução. Para verificar se estamos no segundo ciclo

da instrução `jalr`, usamos o FlipFLop JALR 2ndCycle. Ele recebe como entrada o sinal JALR do controle (que indica que a instrução `jalr` está em seu primeiro ciclo) e funciona como um flip-flop tipo D, ou seja, no ciclo seguinte sua saída estará em nível lógico alto, de forma a indicar o segundo ciclo de `jalr` no estágio ID.

A partir dessas entradas, a unidade gera um conjunto de sinais de saída que orquestram o processador. Esses sinais são agrupados logicamente de acordo com o estágio do pipeline em que serão utilizados. Para o estágio de execução, os EXSignals determinam o comportamento da ALU. Este feixe inclui o ALUSrcA, que seleciona a primeira entrada da ALU (seja o valor do primeiro registrador, o PC ou uma constante zero), o ALUSrcB, que seleciona a segunda entrada (seja o valor do segundo registrador ou o valor imediato), e o ALUOp, que instrui a ALU sobre qual operação realizar (uma adição para cálculo de endereço, uma operação baseada nos campos funct3 e funct7 para o Tipo-R, etc.). Para o estágio de memória, os MEMSignals consistem no MemRead e MemWrite. O MemRead é ativado ('1') exclusivamente para instruções load (opcode "0000011"), enquanto o MemWrite é ativado apenas para instruções store (opcode "0100011"), garantindo que os acessos à memória de dados ocorram corretamente. A lógica de controle também precisa lidar com as instruções do Tipo-U, como lui e auipc, que são essenciais para carregar valores de 32 bits. Para a instrução lui (load upper immediate), cujo objetivo é posicionar uma constante de 20 bits na parte superior de um registrador, a unidade de controle prepara a ALU para somar o valor imediato (já deslocado para a esquerda pelo Immediate Generator) a zero. Já para a instrução auipc (add upper immediate to pc), a unidade configura a ALU para somar o mesmo tipo de imediato ao valor do Program Counter (PC), sendo uma operação crucial para a geração de endereços relativos ao PC. Em ambos os casos, é a decodificação precisa no estágio ID que garante que as entradas corretas sejam selecionadas para a ALU no estágio seguinte.

No estágio final, o de Write-Back, os WBSignals gerenciam a escrita de resultados no banco de registradores. Este feixe é composto pelo RegWrite e pelo MemToReg. O RegWrite é um sinal permissivo fundamental, ativado para todas as instruções que modificam um registrador, como as do Tipo-R, Tipo-I, load e as de salto com link (`jal` e `jalr`), e desativado para stores e branches. O sinal MemToReg, por sua vez, atua como um seletor: ele é ativado apenas para a instrução load, direcionando o dado vindo da memória para o banco de registradores; para as demais operações que escrevem um resultado, como as aritméticas, esse sinal permanece desativado, selecionando

o resultado vindo da ALU.

Além disso, o Control produz sinais de saída específicos para gerenciar o fluxo de controle do programa. Os sinais BranchEQ, BranchNEQ, JAL e JALR são flags que indicam à lógica de atualização do PC qual tipo de salto está ocorrendo - ou qual tipo de condição deve ser verificada -, permitindo o cálculo correto do próximo endereço de instrução. Já o sinal ImmSrc é gerado para controlar o valor correto de imediato que deve ser passado adiante, diferenciando entre o valor saído do Immediate Generator e uma constante 1 para realizar as somas PC+1 em casos de `jal` e `jalr`.

O MUX ControlSignals existe para implementarmos nops em caso de load-use hazards. A lógica por trás do hazard é a mesma encontrada no livro. Nosso código verifica se a instrução no estágio EX é um load (usando o sinal ID/EX_MemRead) e se o registrador no qual ela vai escrever (ID/EX_RegisterRd) é o mesmo que a instrução seguinte, no estágio ID, precisa ler (IF/ID_RegisterRs1 ou IF/ID_RegisterRs2). Se essa dependência for encontrada, a unidade ativa a saída StallPipeline, que comanda o processador a parar por um ciclo para esperar o dado ficar pronto, garantindo que a execução continue correta. Caso não haja um stall, o MUX irá permitir que os sinais de controle passem para o próximo estágio do pipeline. Contudo, se a Hazard Unit ativar o sinal de stall, o MUX bloqueia os sinais de controle originais e, em vez disso, envia um conjunto de zeros para a saída. Ao zerar os sinais de controle, a instrução que está entrando no estágio EX se torna uma operação nula, ou uma "bolha", que não altera nenhum estado do processador. É dessa forma que o pipeline é efetivamente paralisado por um ciclo, implementando a pausa necessária para resolver o hazard.

Já o Immediate Generator foi feito pensando em como as instruções RISC-V, como `addi, lw, sw` e as de salto, usam um valor imediato. No entanto, a arquitetura RISC-V codifica esses imediatos de maneiras distintas dependendo do formato da instrução (Tipo-I, Tipo-S, Tipo-B, etc.), espalhando seus bits por diferentes posições na palavra de instrução. Por isso, o Immediate Generator recebe a instrução completa. Ele então extrai os bits corretos, os reorganiza na ordem apropriada e realiza a extensão de sinal para convertê-los em um número de 32 bits, pronto para ser usado pela ALU.

Por fim, o PC Plus Imm e o Check Equal são componentes mais simples que foram criados para acelerar o processamento de instruções de salto. O PC Plus Imm é um somador simples cuja única função é calcular o endereço de destino. Ele recebe o valor atual do PC e um valor imediato vindo do Immediate Generator, e sua saída (Next PC) é a soma dos dois. Este resul-

tado é o endereço para onde o programa vai saltar se a condição do salto for atendida - ou caso não o salto não seja condicional (`jal`). Já o Check Equal realiza a comparação necessária para os saltos condicionais. Ele recebe os valores de dois registradores, Rs1 e Rs2, e os compara. Se os valores forem idênticos, a saída IsEqual se torna '1', caso contrário, ela se torna '0'. Essa saída é então usada pela lógica de controle do salto (portas lógicas) para decidir se a condição do `beq` ou `bne` foi satisfeita.

O Register File é idêntico ao do livro e, por isso, não o detalheremos neste relatório. Para implementá-lo, nos baseamos no bloco de registradores disponível no Digital [1]. Entretanto, não utilizamos diretamente o bloco pois o número máximo de bits de dados possível era 24. Sendo assim, exportamos o bloco como VHDL e alteramos, diretamente no código, o número de bits de dados para 32.

3.2.3 Execução da Operação (EX)

Esse estágio pode ser visto contendo 3 componentes fundamentais: ALUControl, ALU e Forwarding Unit além dos MUX's responsáveis por encaminhar as entradas para a ALU, sendo estes os ALUSrcA e ALUSrcB. As entradas podem ser de um registrador, de um imediato ou então de um resultado da própria ALU caso seja preciso. Esse último é o caso no qual a Forward Unit é necessária, usando também os MUX's de ForwardA e ForwardB, para que, caso uma entrada requeira um valor previamente calculado pela ALU, ele possa ser enviado para ela sem causar um stall na pipeline.

A ALUControl é o componente que recebe do Control o tipo de operação que a ALU deve realizar, por meio do sinal ALUOp de 2 bits, possuindo 4 tipos:

- 00 - para instruções do tipo Branch.
- 01 - para instruções do tipo R-Type.
- 10 - para instruções do tipo I-Type.
- 11 - para instruções que usam a ALU para soma.

Além disso, também recebe o funct3 e o funct7, caso a instrução tenha - caso contrário, o ALUControl recebe os bits que estão na posição que o funct3 e o funct7 estariam, mas não os utiliza. Com essas 3 entradas, é possível gerar uma saída de 4 bits, ALUCtl, que irá informar à ALU qual tipo de operação

ela deve realizar.

A ALU foi projetada para receber dois operandos de 32 bits, ALUScrA e ALUSrcB, e um sinal de controle de 4 bits que determina a operação a ser executada, chamado de ALUCtl. Por fim, gera uma saída ALUOut também de 32 bits.

A ALU é responsável por realizar as operações necessárias de todas as instruções requisitadas para esse trabalho, sendo as operações:

- AND, $A\ and\ B \rightarrow ALUOut$
- OR, $A\ or\ B \rightarrow ALUOut$
- ADD, $A\ add\ B \rightarrow ALUOut$
- SUB, $A\ sub\ B \rightarrow ALUOut$
- XOR, $A\ xor\ B \rightarrow ALUOut$
- SLL, $A\ shift\_left(amount = B) \rightarrow ALUOut$
- SRL, $A\ shift\_right(amount = B) \rightarrow ALUOut$

A Forward Unit é necessária para arquiteturas de CPU em pipeline, pois uma instrução pode precisar do resultado de uma instrução anterior que ainda não foi escrito de volta no banco de registradores. A Forwarding Unit monitora os operandos das instruções nos estágios do pipeline para determinar se o resultado de uma operação (nos estágios `MEM` ou `WB`) é necessário como operando para a instrução atualmente no estágio `EX`. Além disso, há ainda a possibilidade de se ter um resultado no estágio `WB` e também no `MEM`, de tal forma que o resultado encaminhado deve sair do `MEM` pois este é o mais recente. Se um conflito for detectado, a unidade gera sinais de controle para adiantar ("forward") o resultado diretamente para a entrada da ALU, contornando o banco de registradores.

3.2.4 Acesso à Memória (MEM)

O estágio de Acesso à Memória (MEM) possui a responsabilidade de interagir com a memória de dados, executando as operações de leitura ou escrita que foram solicitadas pela instrução.

Este estágio é composto por somente um componente:

- RAM.

A RAM funciona como a memória principal do nosso processador. Ela usa apenas os 16 bits menos significativos do sinal `Addr` como endereço para localizar uma posição, como já explicamos. Novamente, exportamos o código em VHDL que descreve a RAM disponível no simulador e fizemos as alterações necessárias para o nosso caso, já que a ferramenta Digital [1] não possuía um bloco de memória com dados de 32 bits.

Sendo assim, quando uma instrução load precisa ler um dado, o sinal MemRead (gerado pelo Control no estágio ID 3.2.2) comanda a RAM. Ela recebe o endereço de 32 bits, mas usa apenas os 16 menos significativos para encontrar a localização correta e copia o valor de 32 bits que está armazenado lá para a saída `Dout`. Esta, por sua vez, é enviada para o registrador de pipeline MEM/WB, para que no estágio seguinte o dado possa ser enviado ao componente Register File. No caso de uma instrução de store, o sinal MemWrite (também vindo do Control no estágio ID 3.2.2) indica que o dado que está sendo recebido pela RAM deve ser escrito no endereço descrito pelos 16 bits menos significativos do endereço que está sendo fornecido ao bloco de memória.

3.2.5 Escrita no Register File (WB)

Nesse estágio temos apenas um componente: o Multiplexador `MemToReg`. Este mux é o componente central do estágio WB, atuando como um "tomador de decisão". Sua função é selecionar qual dado será efetivamente escrito no banco de registradores, com base no tipo da instrução que está sendo finalizada. As opções, conforme descrito na seção referente ao estágio ID (3.2.2), são: escrever o resultado da ALU ou escrever um dado buscado na memória. O sinal de seleção do MUX vem do control.

3.3 Resultados

Para testar nossa CPU, criamos um código simples em assembly que nos permitisse verificar todo tipo de situação possível (saltos, operações lógicas ou aritméticas, etc.). Então, encontramos as versões em binário de cada instrução e seu equivalente em hexadecimal. Esses são os valores presentes na memória de instruções do arquivo `CPUsemSIMD.dig` que enviamos juntamente com este relatório e que está também presente no GitHub referenciado [3].

O código, em assembly, que escrevemos é o seguinte:

```
addi    x1,x0,8     ;x1 recebe 8
ori     x2,x1,4     ;x2 recebe 12
andi    x3,x2,4     ;x3 recebe 4
xori    x4,x2,10    ;x4 recebe 6
slli    x5,x1,2     ;x5 recebe 32
beq     x0,x1,10    ;untaken, continua normalmente
srli    x6,x1,2     ;x6 recebe 2
sw      x1,1(x0)    ;M[1] recebe 8
add     x7,x2,x3    ;x7 recebe 16
and     x8,x2,x4    ;x8 recebe 4
or      x9,x2,x4    ;x9 recebe 14
xor     x10,x2,x4   ;x10 recebe 10
sub     x11,x2,x1   ;x11 recebe 4
sll     x12,x3,x6   ;x12 recebe 16
bne     x3,x8,10    ;untaken, continua normalmente
srl     x13,x5,x6   ;x13 recebe 8
lw      x14,x0,1    ;x14 recebe 8
beq     x3,x8,3     ;taken, vai para PC+6=24
addi    x19,x0,1    ;para ver se vai virar nop
addi    x19,x0,1    ;para ver se vai virar nop
addi    x19,x0,1    ;para ver se vai virar nop
addi    x19,x0,1    ;para ver se vai virar nop
addi    x19,x0,1    ;para ver se vai virar nop
bne     x2,x3,3     ;taken, vai para PC+6=30
addi    x19,x0,1    ;para ver se vai virar nop
addi    x19,x0,1    ;para ver se vai virar nop
addi    x19,x0,1    ;para ver se vai virar nop
addi    x19,x0,1    ;para ver se vai virar nop
addi    x19,x0,1    ;para ver se vai virar nop
auipc   x15,1       ;x15 recebe PC+2^12=PC+4096=4126
lui     x16,2       ;x16 recebe 2^13=8192
jal     x17,3       ;x17 recebe PC+1=33 e vai para PC+6=38
addi    x19,x0,1    ;para ver se vai virar nop
addi    x19,x0,1    ;para ver se vai virar nop
addi    x19,x0,1    ;para ver se vai virar nop
addi    x19,x0,1    ;para ver se vai virar nop
addi    x19,x0,1    ;para ver se vai virar nop
jalr    x18,x5,12   ;x18 recebe PC+1=39 e vai para 32+12=44
addi    x19,x0,1    ;para ver se vai virar nop
addi    x19,x0,1    ;para ver se vai virar nop
addi    x19,x0,1    ;para ver se vai virar nop
addi    x19,x0,1    ;para ver se vai virar nop
addi    x19,x0,1    ;para ver se vai virar nop
addi    x20,x1,-4   ;x20 recebe 4
```

Para efetuar os testes, inserimos os valores hexadecimais de cada instrução

no endereço da memória de dados correspondente à linha da instrução no código acima. Então, criamos, no simulador, um botão de reset e um botão para simular o clock. Optamos pelo uso de um botão ao invés do clock disponível no Digital [1] pois queríamos passar por cada instrução com bastante calma e analisarmos o efeito que cada uma causava na CPU.

Com isso, fomos capazes de perceber alguns erros.

O principal problema que tivemos foi com a utilização de 'process' nos códigos em VHDL. Percebemos que deveríamos usar lógicas puramente combinacionais em certos blocos ao invés de usar um 'process' que verificasse alguma entrada para produzir a saída equivalente. Os nossos problemas estavam relacionados a saídas que correspondiam a entradas antigas, e não às entradas atuais do bloco em questão. Após essa percepção, modificamos os códigos que causavam esse problema para usarem apenas lógicas combinacionais.

Também encontramos outros erros, muito menos problemáticos, como fios ligados em posições erradas no simulador (o que é muito difícil de perceber apenas visualmente). Após todas as correções necessárias, nossa CPU se comportou da maneira que desejávamos.

3.4 Conclusões

Com esse trabalho, pudemos entender alguns dos desafios por trás da criação de um computador. Apesar de nos basearmos na CPU do livro-texto [4], tivemos que pensar em como implementar algumas instruções que não eram processadas na pipeline simples do Capítulo 4, especialmente as instruções `auipc, lui, jal e jalr`.

Além disso, aprofundamos nossos conhecimentos acerca da linguagem de descrição de hardware VHDL e pudemos, também, conhecer melhor o simulador Digital [1]. Esses conhecimentos serão de enorme importância no restante das nossas formações e nas nossas carreiras como engenheiros eletrônicos e da computação.

4 Projeto de CPU RISC-V com SIMD

4.1 Objetivo

Nesta etapa, temos como objetivo criar uma CPU que suporte a algumas instruções vetoriais, sendo elas: `add, addi, sub, sll, slli, srl e srli`. Aqui, ressaltamos que o docente esclareceu em uma aula de dúvidas que não há necessidade de implementar a versão vetorial da instrução `auipc`.

Em outras palavras, essa etapa do trabalho é uma junção das duas anteriores, já que inicialmente fizemos um somador/subtrator vetorial (descrito no capítulo 2) e, após, fizemos uma CPU que não comporta tal tipo de instrução (descrito no capítulo 3). Devemos, então, definir os valores em binário e, consequentemente, em hexadecimal para cada uma das instruções vetoriais e alterar o control, o gerador de imediato, o ALU control e a ALU da CPU da etapa anterior (aquela que não suporta instruções vetoriais) para que nossa nova CPU seja capaz de processar as instruções vetoriais requisitadas.

4.2 Implementação

Para a implementação das instruções vetoriais faz-se necessário que a nossa CPU seja capaz de diferenciar quando é uma instrução vetorial ou não, além de também identificar o tamanho dos vetores para a operação: um vetor de 32 bits; dois vetores de 16 bits; quatro vetores de 8 bits; ou oito vetores de 4 bits. Dessa forma, mapeamos as sete instruções vetoriais requeridas para a execução desse trabalho de acordo com as seguintes tabelas:

Instrução	**funct7**	**funct3**	**opcode**
add_vet	000 0000	0VV	111 0000
sub_vet	000 0000	1VV	111 0000
sll_vet	000 0000	0VV	111 0001
srl_vet	000 0000	1VV	111 0001

Tabela 1: Tabela de instruções para R-Type, onde VV indica o tamanho dos vetores

Por se tratar de instruções R-Type, elas seguem o seguinte formato:

funct7	**rs2**	**rs1**	**funct3**	**rd**	**opcode**

Tabela 2: Descrição R-Type

Instrução	**funct3**	**opcode**
addi_vet	0VV	111 1000
slli_vet	0VV	111 1001
srli_vet	1VV	111 1001

Tabela 3: Tabela de instruções para I-Type, onde VV indica o tamanho dos vetores

Por se tratar de instruções I-Type elas seguem o seguinte formato:

immediate	**rs1**	**funct3**	**rd**	**opcode**

Tabela 4: Descrição I-Type

Foi decidido que, para o mapeamento dessas sete instruções, utilizaríamos o Most Significant Bit (MSB) do funct3 para diferenciar `add` de `sub`, que possuem o mesmo opcode, além de também diferenciar `sll` de `srl` que possuem o mesmo opcode entre si, mas um opcode diferente comparado ao de add e sub. Já a diferenciação entre R-Type e I-Type está no quarto bit do opcode. Ademais, o funct3 também carregará a informação do tamanho dos vetores nos seus dois bits menos significantes (LSB), de acordo com a tabela a seguir:

N°de Vetores X Tamanho em Bits	**funct3**
8×4	X00
4×8	X01
2×16	X10
1×32	X11

Tabela 5: Identificação do tamanho dos vetores de operação a partir do funct3 da instrução.

Esse mapeamento foi feito de maneira a reduzir o número de alterações necessárias na nossa CPU, o que ficará mais evidente nas subseções abaixo, onde explicamos as modificações feitas nos componentes.

4.2.1 Control

Foi preciso mudar o Control, pois a ALU passou a executar novas operações que antes não existiam. Para acomodar essas novas funcionalidades, o sinal de controle `ALUOp`, que efetivamente diz à ALU qual cálculo realizar, foi expandido de 2 para 3 bits, aumentando o número de operações que poderiam ser codificadas. Consequentemente, a lógica de decodificação do Control foi atualizada para reconhecer os opcodes específicos das novas instruções vetoriais. Ao identificar uma dessas instruções, o Control agora envia os novos códigos de `ALUOp` para a ALU. Além disso, a lógica para os demais sinais, como `ALUSrcA, ALUSrcB e RegWrite`, também foi estendida para incluir os novos opcodes, garantindo que eles se comportem de maneira apropriada, de forma muito similar a como as instruções do Tipo-R já eram tratadas.

4.2.2 ALUControl e ALU

No nosso ALUControl, as modificações foram mínimas, pois a verificação se é ou não uma operação vetorial vem do Control através do ALUOp. Dessa forma, as únicas modificações feitas foram: a adição de dois novos cases; a entrada ALUOp é de 3 bits, diferentemente do caso anterior (CPU sem instruções vetoriais) em que ALUOp era de apenas 2 bits. Os cases adicionados foram:

Para soma ou subtração:

```
when "100" => -- mapeamento do Control para ADD, ADDI e SUB
    ALUCtl <= "0011"; -- saída para a ALU
```

E para quando a operação é um shift lógico:

```
when "101" =>  -- mapeamento do Control para SLL, SRL, SLLI, SRLI
    ALUCtl <= "0111";
```

Deve-se ressaltar que, para a ALU, não faz diferença se a instrução é R-Type ou I-Type, tendo em vista que ela apenas recebe as entradas através da ALUSrcA e ALUSrcB e realiza a operação que a ALUControl envia (seja a entrada B um valor vindo de um registrador ou um imediato).

Já para a ALU, foram feitas grandes modificações:

- Recebimento direto do funct3 na ALU.
- Novos cases para lidar com os dois novos valores possíveis da entrada ALUCtl.

- Case 1: Integração Somador Vetorial com CPU sem SIMD.
- Case 2: Criação do código para os Shifts Vetoriais.

Para o primeiro case bastou integrarmos o código feito na primeira etapa descrita nesse relatório (2). Com essa integração, fomos capazes de obter já as instruções de add, addi e sub vetoriais.

A execução dessas instruções é feita conforme foi descrito na seção correspondente (2.2): Cada entrada (A e B) é quebrada em vetores de tamanhos fixos especificados pelo funct3 e, em seguida, é feita a operação entre os respectivos vetores da entrada A com a entrada B. Caso seja uma operação de soma com 2 vetores de 16 bits, por exemplo, ocorrerão 2 somas: vetor 1 de A + vetor 1 de B; vetor 2 de A + vetor 2 de B.

Para o segundo case essa lógica também é aplicada, mas agora para o caso de SHIFT, em que cada vetor de A possui uma quantidade de shifts definida por cada vetor de B. Ou seja, B também será dividido em vetores e cada um deles será pareado com um vetor de A.

4.3 Resultados

Para esse trabalho, não criamos um código de testes como fizemos no caso da CPU sem instruções vetoriais (3.3). Como modificamos apenas alguns blocos, testamos eles separadamente para todas as instruções vetoriais requisitadas e atestamos seus funcionamentos corretos. Com isso feito, inserimos tais blocos na CPU da etapa anterior (3). Assim, criamos o arquivo `CPUcomSIMD.dig` com as alterações descritas ao longo deste capítulo do relatório.

Para testar o funcionamento total da nossa nova CPU, usamos o mesmo código e o mesmo procedimento da etapa anterior (descritos na seção 3.3). Como o código de teste não possuía instruções vetoriais, adicionamos, ao final do mesmo, duas das que implementamos. Dessa forma, se ambas funcionassem corretamente, poderíamos ter certeza de que as demais também funcionariam, uma vez que os blocos afetados por essas operações já foram testados individualmente.

Como os pequenos erros já tinham sido corrigidos durante os testes da CPU sem suporte para operações vetoriais, os testes para a CPU dessa etapa foram bem mais rápidos e seus resultados estavam de acordo com o esperado.

4.4 Conclusões

Esta etapa do trabalho nos apresentou mais um grande desafio: determinar a codificação de uma instrução. As limitações impostas pelo tamanho da instrução (32 bits) e pelas demais instruções já existentes apresentaram obstáculos ao grupo. Entretanto, devido ao baixo número de instruções que devem ser suportadas pela nossa CPU, conseguimos definir de maneira eficiente as codificações para as instruções vetoriais requisitadas.

Dessa forma, nosso senso crítico e nossa capacidade de solução de problemas foram estimulados de forma a nos tornar melhores engenheiros, visto que tais competências serão essenciais nas nossas carreiras.

5 Conclusão

Os projetos descritos nesse relatório forneceram um grande aprendizado aos membros do grupo. Inicialmente, o trabalho com o somador/subtrator vetorial (2) nos ajudou a pensar em detalhes importantes para a eficiência de um projeto. Num segundo momento, o desenvolvimento da CPU sem operações vetoriais (3) nos mostrou como pensar em grandes projetos e nos fez entender a necessidade de planejar antes de executar. Por fim, o trabalho na CPU com operações vetoriais (4) nos permitiu entender melhor como aproveitar projetos já existentes e modificá-los para incorporar as novas características necessárias.

Como discutido nas seções de conclusões em cada capítulo, todas as habilidades técnicas desenvolvidas com esses projetos serão de grande importância em nossas formações como engenheiros eletrônicos e de computação. Porém, acreditamos que outros aspectos tão essenciais quanto as habilidades técnicas também foram aprimorados pelo grupo. O mais evidente deles foi o trabalho em equipe, que exige um bom planejamento conjunto para que todos sejam capazes de contribuir sem sobrecarga e para que todas as contribuições se integrem corretamente.

O planejamento, já citado, também foi uma habilidade exercitada pelo grupo. Documentamos nossas decisões, dividimos tarefas e registramos tudo aquilo que poderia ser necessário posteriormente. Com isso, foi muito mais fácil integrar os diferentes blocos e escrever este relatório, uma vez que todas as informações já estavam anotadas.

Em suma, desenvolvemos e aprimoramos habilidades essenciais para o res-

tante de nossas formações acadêmicas e também para nossas futuras carreiras como engenheiros.

6 Referências Bibliográficas

[1] Helmut Neemann. *Digital*. URL: https://github.com/hneemann/Digital (acesso em 19/04/2025).

[2] Diego Dutra. *Laboratório de Computação Paralela e Sistemas Móveis - EEL580*. URL: https://www.compasso.ufrj.br/disciplinas/eel580 (acesso em 23/04/2025).

[3] Marcelo Firmino, Ricardo Amaral e Umberto Augusto. *Projetos-RISC-V*. URL: https://github.com/UmbertoAugusto/Projetos-RISC-V.

[4] D. A. Patterson e J. L. Hennessy. *ComputercOrganization and Design RISC-V Edition: The Hardware Software Interface*. 2ª ed. Morgan Kaufman, 2021. ISBN: 978-0-12-820331-6.