MC851 Projeto em Computação

Relatório Entrega 1

Resumo: planejamento e desenvolvimento de um processador RISC-V de 32 bits, com pipeline, que suporte o conjunto RV32I.

Equipe "RISC-VI":
RA 169374, Daniel Paulo Garcia
RA 182783, Lucca Costa Piccolotto Jordão
RA 185447, Paulo Barreira Pacitti
RA 198435, Guilherme Tavares Shimamoto
RA 216116, Gabriel Braga Proença
RA 221859, Mariana Megumi Izumizawa

1. Introdução

O *design* e desenvolvimento de um sistema computacional pode ser muito complexo, e muito caro também. Graças a tecnologias de HDL (*Hardware Description Language*) e *chips* FPGA (*Field Programmable Gate Arrays*), podemos prototipar *chips* de forma barata e rápida. Mesmo com experiências com RISC-V em MC404 (Organização Básica de Computadores e Linguagem de Montagem), com FPGA e HDL em MC613 (Laboratório de Circuitos Digitais) e MC732 (Projeto de Sistemas Computacionais), construir um SoC (*System-on-a-Chip)*, a equipe entende o projeto como um desafio bastante complexo e têm estudado e revisado conceitos dessas disciplinas. Nesta primeira entrega, apresentamos o planejamento de desenvolvimento realizado de um sistema computacional com arquitetura RISC-V de 32 bits que têm suporte ao conjunto RV32I.

2. Planejamento

O grupo fez o *design* do sistema computacional com arquitetura RISC-V na forma de um processador com *pipeline* RISC-V de cinco estágios [1]: IF (*Instruction Fetch*), ID (*Instruction Decode*), EX (*Execute*), MEM (Memory Data) e WB (*Write Back*).

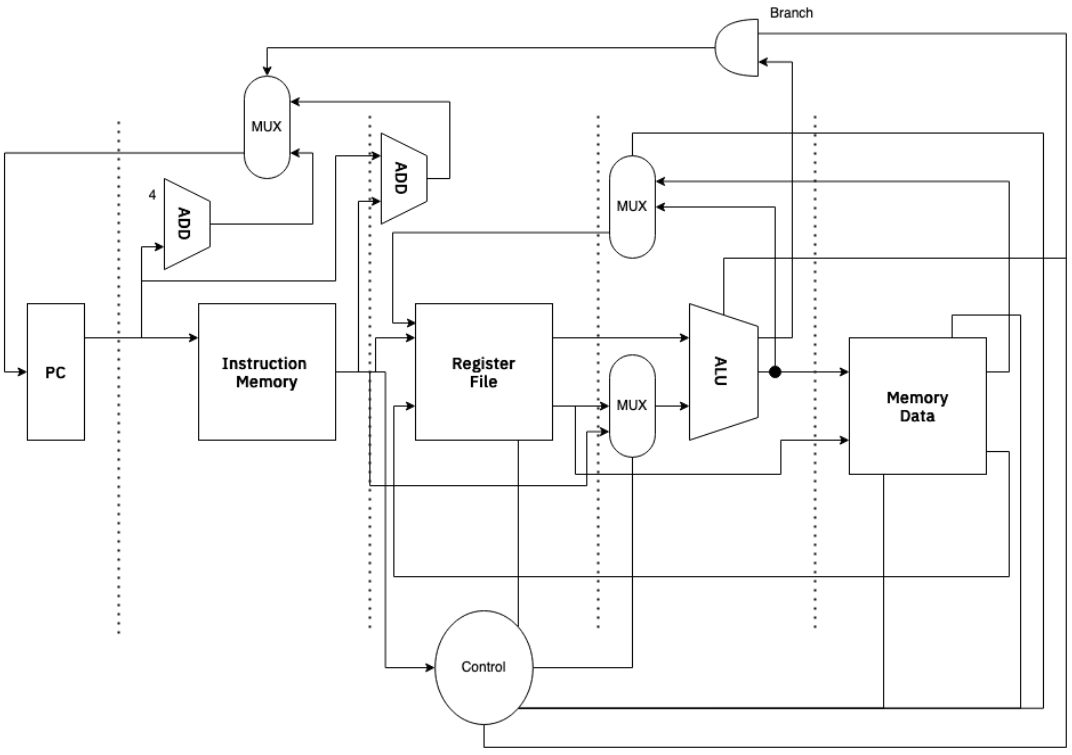

Figura 1: *Datapath do sistema computacional com arquitetura RISC-V com pipeline de 5 estágios a ser implementada pelo grupo.*

Dessa forma, a equipe se dividiu em 3 duplas para o desenvolvimento dos estágios do *pipeline*: Gabriel e Lucca ficaram com os estágios IF e ID, Mariana e Paulo com o EX, e Daniel e Guilherme com o MEM e WB.

Por falta de familiaridade com Verilog, a proposta foi implementar módulos simples como a ALU e o RegFile e após isso integrar os mesmos com seus respectivos estágios. Foi determinado que todo módulo implementado deve ter um *testbench* (suíte de testes) associado, a fim de prevenir entraves no futuro durante a integração de módulos. Visto que o

debugging de uma HDL é muito mais dificultoso do que de uma linguagem de programação comum, o uso desta técnica é essencial para o desenvolvimento dos sistemas computacionais.

3. Resultados

Como resultado do desenvolvimento, construímos módulos responsáveis por cada unidade do processador, separados por cada estágio da pipeline.

O módulo do *Register File* foi implementado utilizando como entrada os registradores a serem lidos, o registrador no qual o dado será escrito, o dado que será escrito e um *write_enable* para garantir que o módulo possui permissão para escrita. Como saída, temos os dois registradores que foram lidos da entrada. Importante ressaltar que caso o registrador a ser escrito seja o *x0*, a escrita não é realizada pois seu dedo sempre será zero.
O módulo de decodificação foi desenvolvido particionando a instrução de entradas bit a bit. O desafio maior da implementação é a diferença que existe entre diferentes tipos de operações, já que um range de bits da instrução pode conter algo diferente dependendo do *opcode* lido. Inicialmente fizemos o particionamento de acordo com o opcode do tipo R, e desenvolvemos a unidade de controle para acionar o próximo estágio de acordo com a operação selecionada (*e.g. add, sub, XOR etc).*

No estágio EX, foi implementado o módulo da ALU já com todas as instruções de aritmética, lógica, de comparação e de deslocamento de bits. Foi também implementado parte dos registradores de *pipeline* necessários no estágio EX, porém, o grupo identificou uma melhor organização para se implementar a *pipeline* e seus registradores de transição, então esta implementação foi suspensa.

```
`include "define.v"

module alu_module (
    input       alu_input_enable,
    input  [3:0]   alu_input_op,
    input  [31:0]  alu_input_a,
    input  [31:0]  alu_input_b,

    output [31:0]  alu_output_result
);

reg [31:0]   alu_register_result;

always @ (alu_input_op or alu_input_a or alu_input_b)
begin
    alu_register_result <= 32'h0;

    if (alu_input_enable)
    begin
        case (alu_input_op)
        // ARITHMETIC OPS
            `ALU_ADD: alu_register_result <= (alu_input_a + alu_input_b);
            `ALU_SUB: alu_register_result <= (alu_input_a - alu_input_b);
        // LOGIC OPS
            `ALU_AND: alu_register_result <= (alu_input_a & alu_input_b);
            `ALU_OR:  alu_register_result <= (alu_input_a | alu_input_b);
            `ALU_XOR: alu_register_result <= (alu_input_a ^ alu_input_b);
        // SHIFT OPS
            `ALU_SLL:  alu_register_result <= alu_input_a << alu_input_b[4:0]; // need to set a
range otherwise it will binary extend the number
            `ALU_SRL:  alu_register_result <= alu_input_a >> alu_input_b[4:0];
            `ALU_SRA:  alu_register_result <= $signed(alu_input_a) >>> alu_input_b[4:0];
        // COMPARE OPS
            `ALU_SLT:  alu_register_result <= (alu_input_a < alu_input_b ? 1 : 0);
            `ALU_SLTU:  alu_register_result <= ($signed(alu_input_a) < $signed(alu_input_b) ? 1
: 0);
        endcase
    end
end

assign alu_output_result = alu_register_result;
endmodule
```

Figura 2: *código implementado em Verilog para o módulo da ALU.*

No estágio MEM, o desenvolvimento foi iniciado criando o módulo de RAM, sendo um componente simples e responsável pela leitura e escrita dos dados do processador. Além disso, para testar a funcionalidade do módulo foram criados *testbenches* validando os cenários de leitura e escrita. Sendo que, a criação desses testes foi importante para o grupo por ter introduzido testes simulando o clock do processador.

Em sequência, após algumas discussões e com o auxílio do professor, o grupo optou pela implementação de um módulo de *Memory Control*, responsável pelo controle do acesso de memória dos estágios IF e MEM. Logo, o módulo foi criado dividindo a memória do processador em RAM e ROM, o grupo decidiu realizar essa divisão separando as primeiras 1024 posições para ROM e as outras 1024 posições da memória para RAM. Visando facilitar a escrita e a leitura dos dados no processador, cada espaço da memória foi criado para armazenar 32 bits. Ademais, temos que a parte mais importante da criação do módulo é justamente o controle de acesso à memória. Para isso, o módulo possui duas saídas *wait_if* e *wait_mem*, que são responsáveis por "habilitar" ou "desabilitar" os registradores de pipeline entre os estágios de IF/ID e MEM/EX. Dessa forma, o estágio de IF é obrigado a esperar enquanto o MEM está acessando a memória e vice-versa. Por fim, como inicialmente todas as instruções do estágio IF necessitam acessar a memória, a prioridade de acesso é do estágio MEM, ou seja, primeiro é verificado se MEM está acessando a memória, se sim *wait_if* é 1, impedindo que os registradores de pipeline de IF/ID continuem a pipeline nesse estágio. Em sequência, se não existe uma operação de memória em MEM, é verificado se IF está tentando acessar a memória, se sim *wait_mem* é 1, impedindo os registradores de pipeline de MEM/EX de continuarem transmitindo dados. Com o conhecimento adquirido nos *testbenches* do módulo de RAM, foram criados testes para validar a leitura e a escrita de dados, além das saídas de controle *wait_if* e *wait_mem*.

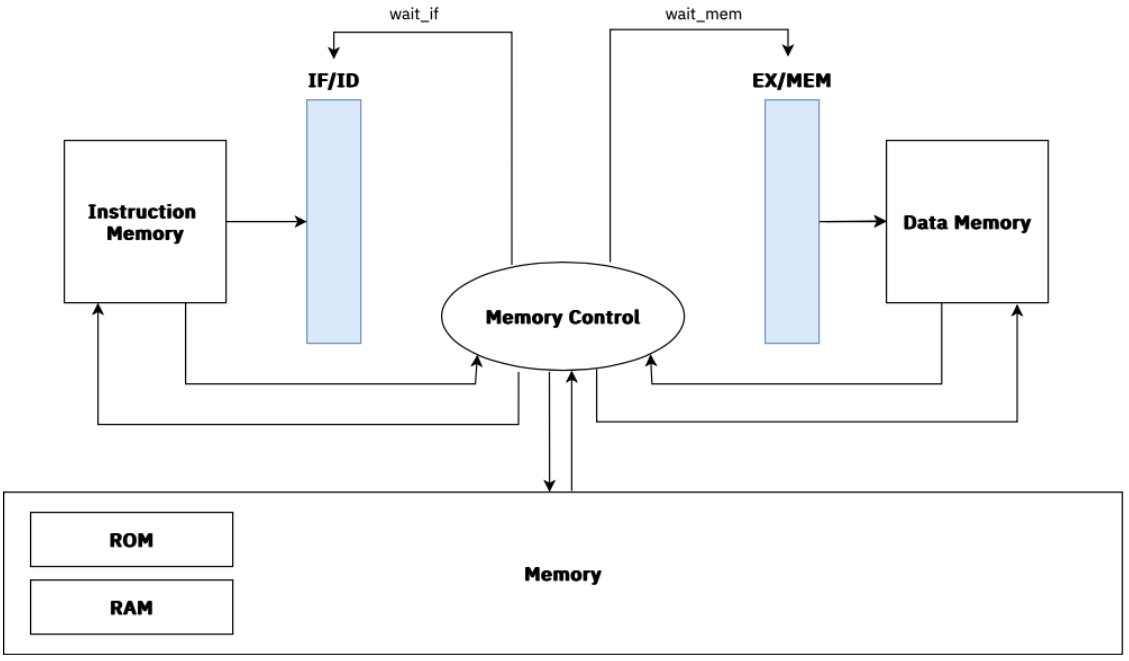

Figura 3: *Diagrama do controlador de memória.*

A equipe enfrentou dificuldades durante o projeto, tanto no processo de *design* quanto no de implementação. Por não ter experiência com Verilog e de projetar uma microarquitetura, o grupo depreendeu muito do tempo nos estudos e aprendizado dessas técnicas. Apesar de ter sido entregues módulos bem implementados, como a ALU e o RegFile, a equipe ainda sente dificuldades nessas áreas. O grupo não entregou o *subset* de instruções RV32I no prazo estipulado, mas visto que na apresentação da disciplina os prazos já foram determinados como apertados e principalmente com o fim de acompanhamento e não de entrega de resultados, a equipe entende que conseguiu se esforçar ao máximo para esta entrega, mesmo que parcial.

4. Futuro

Para a próxima entrega, a equipe irá priorizar integrar os módulos desenvolvidos na *pipeline* de cinco estágios a fim de concluir o subset RV32I de instruções especificadas na arquitetura RISC-V. Após isso, irá se concentrar em integrar periféricos, melhorar módulo de memória e na implementação de *caches*. Em paralelo a isso, também fará a implementação de instruções do *subset* RV32M.

A equipe identificou que, visto a complexidade do projeto, deveria criar medidas que aumentassem a produtividade do grupo (mais resultados em menos tempo). Dessa forma, além do encontro semanal de sexta, foi estipulado uma outra reunião semanal para *reports* e resolução de problemas de forma colaborativa. Foi observado que, por diferença no ritmo de aprendizado, alguns membros tinham mais domínio sobre determinada área do projeto do que outros. Para isso, foi determinado que a colaboração deve ser mais intensa para que ocorra menos tempo ocioso entre o aparecimento de um problema e sua solução.

5. Conclusão

Apesar das muitas dificuldades do grupo no desenvolvimento do projeto, toda a equipe está bastante interessada em resolver estes desafios para a construção de um processador com arquitetura RISC-V. Projetos como esse são mais restritos no mercado, e a universidade é um ótimo espaço para se ter esse aprendizado e experiência. Com a reorganização da implementação em Verilog do sistema computacional, e novas revisões no *design* do datapath, a equipe espera ser mais produtiva e consciente do que deve ser implementado, mas principalmente como deve ser desenvolvido.

Referências

1. D. A. Patterson and J. L. Hennessy, *Computer Organization and Design RISC-V Edition: The Hardware Software Interface*, 1st ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2017.