Trabalho Prático 2 - Organização de Computadores

Gabriel Martins Miranda, João Carlos Ferraz, Luiz Felipe Gondim

Universidade Federal de Minas Gerais (UFMG)
Belo Horizonte - MG - Brasil
Matrículas: 2023028307, 2023087990, 2023028188

Introdução e Objetivos

Este trabalho prático da disciplina Organização de Computadores 1 teve como objetivo promover uma maior familiarização dos alunos com a linguagem Verilog, por meio da implementação de um processador RISC-V de 5 estágios. Para isso, foi disponibilizado um notebook contendo uma implementação básica do processador e a visualização do caminho de dados, com a proposta de que os alunos realizassem modificações no código a fim de adicionar novas instruções e corrigir o funcionamento incorreto de algumas instruções existentes. O arquivo modificado foi o proprio notebook disponibilizado e para fins de organização, só será mostrada a imagem do caminho de dados dos testes da instrução implementada. Para mais, os módulos modificados foram o Execution Unit e Decode Unit.

1 Desenvolvimento e Soluções

1.1 Instrução `add Rd, Rs1, Rs2`

O objetivo desta instrução é somar os valores dos registradores `Rs1` e `Rs2`, armazenando o resultado no registrador `Rd`.
Exemplo: `add x10, x15, x20` → `[x10] = [x15] + [x20]`

A solução foi dada de forma relativamente simples. Foi feita uma modificação na célula de execução no seguinte trecho:

```
2: case (funct3)
     0: alucontrol <= (funct7 == 1) ? /*MUL*/ 4'd10 : /*SUB*/ 4'd6;
```

Esse código foi substituído por:

```
2: case (funct3)
     0: alucontrol <= (funct7 == 7'b0000001) ? 4'd10 : // MUL
                      (funct7 == 7'b0100000) ? 4'd6  : // SUB
                      4'd2;   // ADD
```

O problema era que qualquer instrução diferente de `MUL` era tratada como `SUB`, mesmo que fosse `ADD`. Isso foi corrigido com uma verificação adicional do campo `funct7`.

Foram realizados três testes para verificar o funcionamento da instrução:

Teste 1: `add x10, x15, x20`
Valores iniciais: `x15 = 10`, `x20 = 5`, `x10 = 0`
Valor esperado: `x10 = 15`

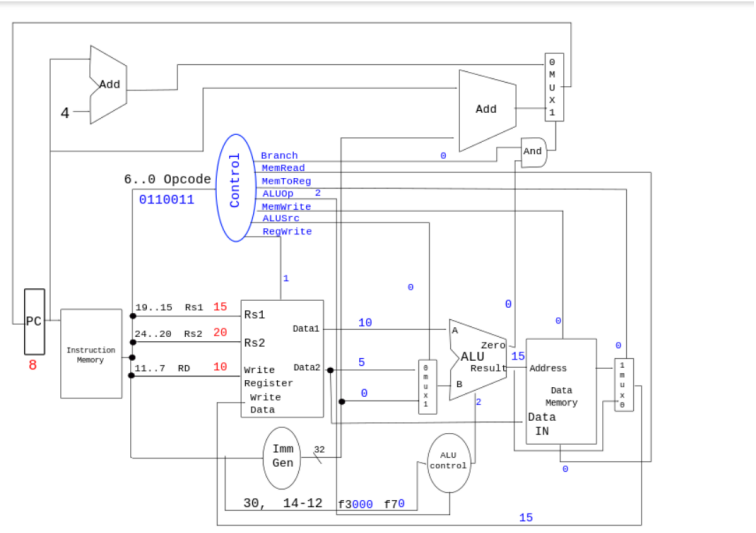

Figura 1: Resultado da instrução `add x10, x15, x20`

Teste 2: `add x11, x12, x13`
Valores iniciais: `x12 = 10`, `x13 = -5`, `x11 = 0`
Valor esperado: `x11 = 5`

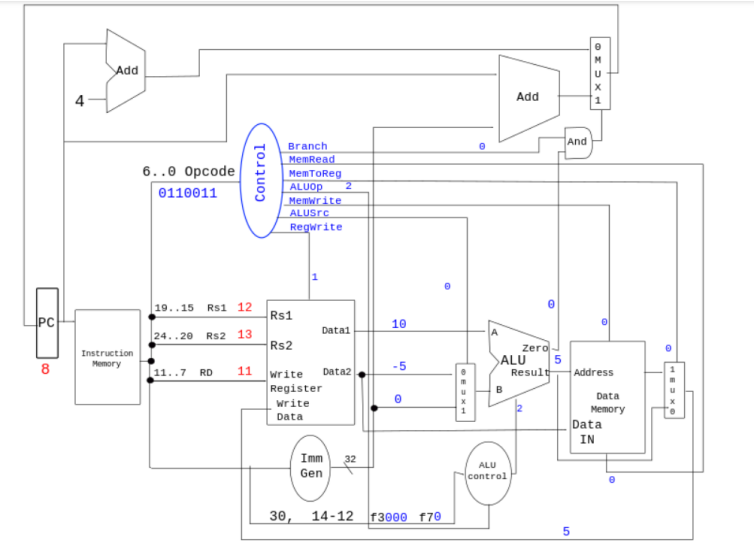

Figura 2: Resultado da instrução `add x11, x12, x13`

Teste 3: `add x14, x16, x17`
Valores iniciais: `x16 = 10`, `x17 = 10`, `x14 = 10`
Valor esperado: `x14 = 20`

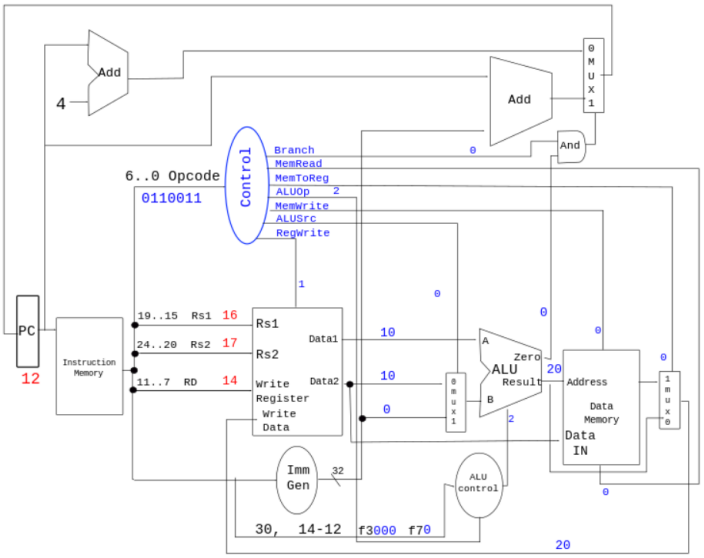

Figura 3: Resultado da instrução `add x14, x16, x17`

1.2 rem Rd, Rs1, Rs2

Essa instrução armazena no registrador destino o resultado de XmodY, sendo X e Y os valores armazenados em Rs1 e Rs2 respectivamente.

Para implementa-la eu modifiquei o módulo alucontrol da execution unit no seguinte trecho:

```
2: case (funct3)
    6: alucontrol <=  4'd1; // OR
```

Esse código foi substituído por:

```
2: case (funct3)
    6: alucontrol <= (funct7 == 1) ? 4'd11 : 4'd1; // REM ou OR
```

Além disso acrescentei no modulo ALU:

```
case (alucontrol)
    11: aluout <= (B != 0) ? $signed(A) % $signed(B) : 32'd0; // REM
```

Em suma, a distinção entre OR e REM depende do campo funct7:

- funct7 == 0000000 → OR
- funct7 == 0000001 → REM

So precisei adicionar esse if simples (funct7 == 1) que identifica instruções da extensão M, como MUL, DIV, REM, etc., que todas usam funct7 = 1.

Agora segue um teste para a instrução:

Teste 1: `rem x3, x2, x1`
Valores iniciais: `x2 = 14`, `x1 = 5`
Valor esperado: `x3 = 4`

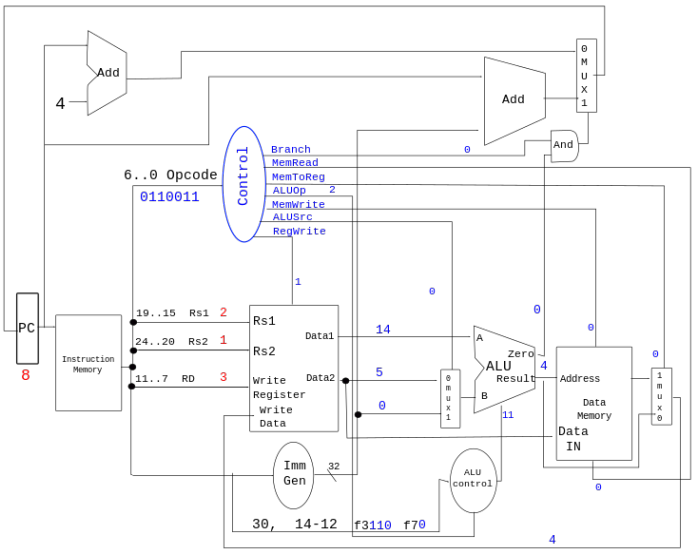

Figura 4: Resultado da instrução `rem x3, x2, x1`

Teste 2: `rem x3, x2, x1`
Valores iniciais: `x2 = 2047`, `x1 = 177`
Valor esperado: `x3 = 100`

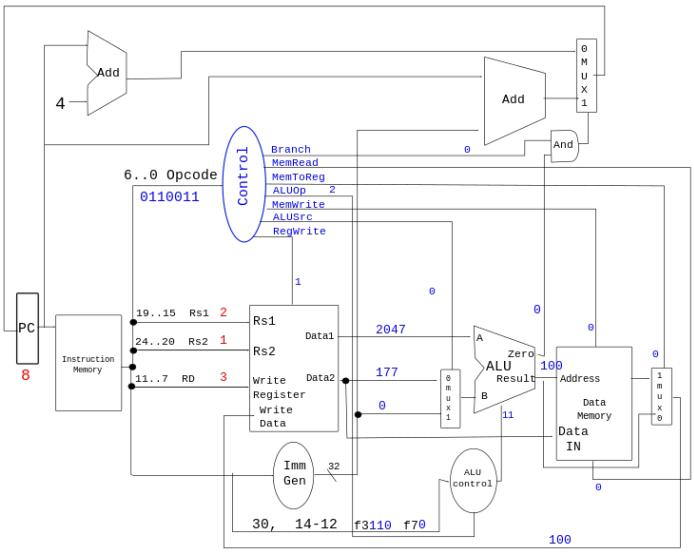

Figura 5: Resultado da instrução `rem x3, x2, x1`

1.3 ori rd, rs1, imm

Esta instrução realiza a operação `OR` bit a bit entre o valor de `Rs1` e o valor imediato `imm`, e armazena o resultado em `Rd`.

Para implementá-la, foi necessário incluir o subcaso a seguir no caso 3 do módulo alucontrol da

`Execution Unit`.

```
6: alucontrol <= 4'd1; // ORI
```

Isso foi feito para que o sistema identifique corretamente a instrução imediata em questão e atribua o valor correto para alucontrol (4'd1.).

Os testes feitos foram:

Teste 1

```
li x2, 0b10101010
ori x3, x2, 0b01010101
```

Valor esperado: `x3 = 255`

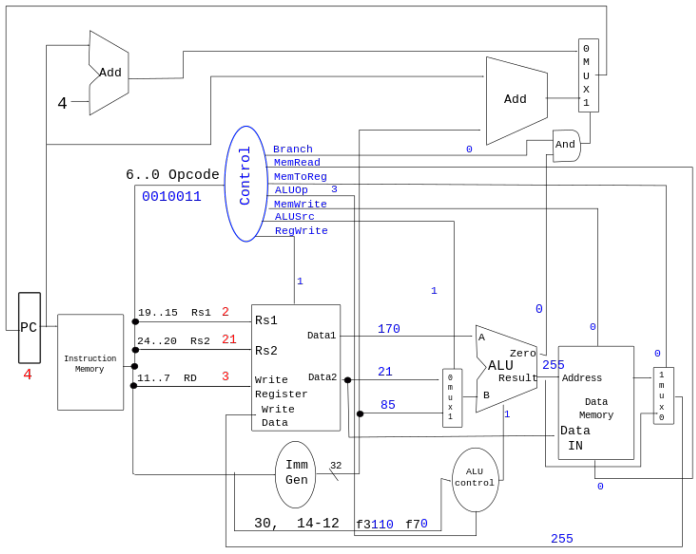

Figura 6: Resultado da instrução `beq x1, x2, pula`

Teste 2

```
li x5, 0b11110000 // 240
ori x6, x5, 0
```

Valor esperado: `x3 = 240`

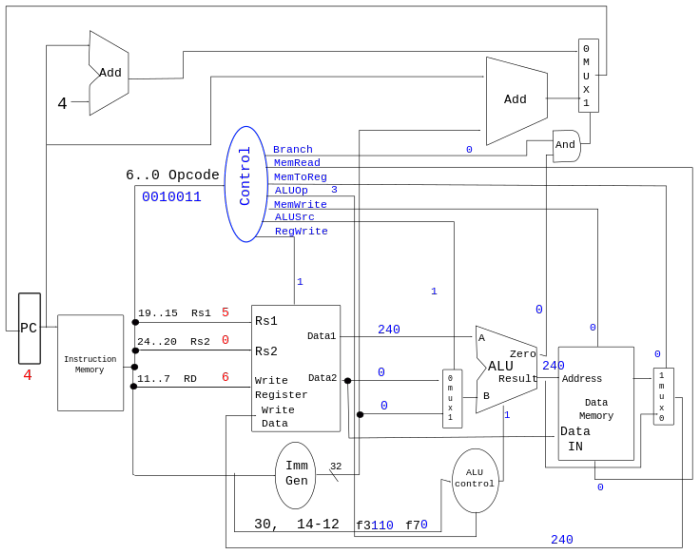

Figura 7: Resultado da instrução `beq x1, x2, pula`

1.4 beq Rs1, Rs2, label

A instrução em questão verifica se os valores de `Rs1` e `Rs2` são iguais, em caso afirmativo, o programa desvia para o endereço identificado pelo label informado.

Para adicioná-la ao programa, foi necessário apenas adicionar o case a seguir em `Control Unit`.

```
7'b1100011: begin // beq
    branch   <= 1;
    aluop    <= 1; // SUB
    ImmGen   <= {{20{inst[31]}}, inst[7], inst[30:25], inst[11:8], 1'b0};
end
```

Isso foi necessário para que o programa consiga identificar o opcode do beq (1100011) e setar os valores das variáveis de controle corretamente.

Em relação aos testes realizados, foram feitos os seguintes:

Teste 1

```
li x1, 10
li x2, 10
beq x1, x2, pula
li x3, 99
pula:
li x3, 123
```

Valor esperado: `x3 = 123`

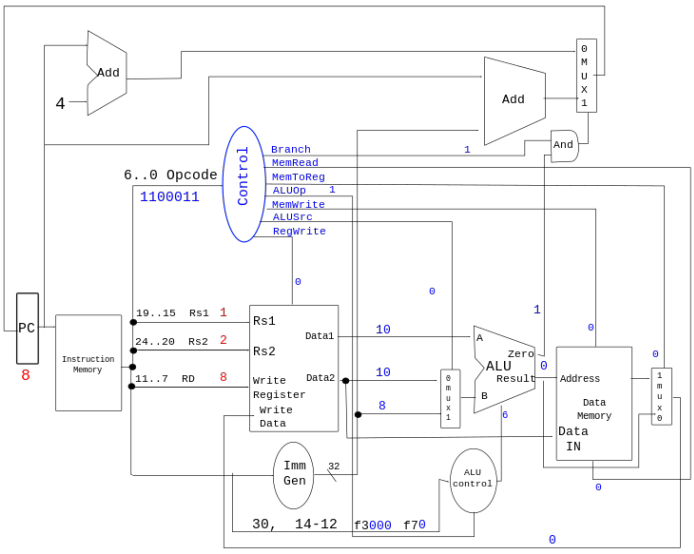

Figura 8: Resultado da instrução `beq x1, x2, pula`

Teste 2

```
li x1, 10
li x2, 9
beq x1, x2, pula
li x3, 99
pula:
li x4, 123
```

Valor esperado: `x3 = 99`

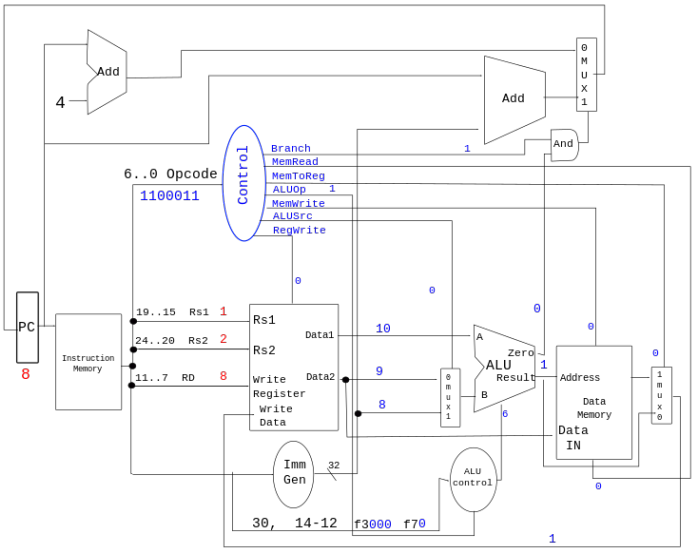

Figura 9: Resultado da instrução `beq x1, x2, pula`

2 Conclusão

Como era de se esperar, todas as modificações concentraram-se no módulo de execução, uma vez que este era justamente o estágio com maior propensão a conter erros. Em essência, as alterações giraram em torno do componente ALUControl, que, em determinados casos, estava ignorando ou confundindo as operações a serem executadas.

Além disso, foi necessário implementar casos específicos para lidar com instruções que não estavam previamente previstas. O maior desafio consistiu em identificar exatamente onde, no caminho de dados, o problema estava ocorrendo e, a partir disso, localizar e corrigir a parte do código responsável pela falha.