# Trabalho Prático 2
# Organização de Computadores

Integrantes:
Lucas Affonso Pires RM: 2023028420
Mateus Faria Zaparoli Monteiro RM: 2023028560
Ricardo Shen RM: 2023028102
Prof. Omar Paranaiba Vilela Neto
Universidade Federal de Minas Gerais, Janeiro, 2025

## 1. Introdução

O objetivo deste trabalho prático é implementar algumas operações, sendo elas: mul, div, xori e beq, em um caminho de dados de uma implementação do RISC-V em um notebook python. Tal trabalho foi desenvolvido em meio à matéria de organização de computadores, na qual foi trabalhado o funcionamento do RISC-V, caminho de dados, pipeline e assembly, e outros tópicos relevantes abordados neste trabalho prático. Aqui foi necessário entender a implementação fornecida pelo professor para, com base nos conhecimentos de caminho de dados adquiridos na disciplina, descobrir onde é necessário modificar os componentes de tal caminho fornecido para que ele possa realizar as operações citadas acima. Para as 3 primeiras instruções foi necessário alterar a execution unit, mais especificamente a “alu control” e a “alu”, já para a operação de branch foi necessário alterar a fetch unit, no módulo fetch, e a decode unit no módulo de controle, ControlUnit.
Uma parte relevante do trabalho foi desenvolver testes para verificar a corretude da implementação das operações pedidas, a visualização fornecida no notebook também foi essencial para verificação da implementação e para correção de erros.

## 2. Desenvolvimento

a. **Implementando a operação de “multiplicação” MUL**:
   Reconhecemos o código funct 7 (1 binário) da multiplicação como definido pela arquitetura através da alucontrol. A alucontrol seleciona o código de multiplicação, reconhecido pela ALU, que executa e retorna o produto pela saída aluout.

b. **Implementando a operação de “divisão” DIV**:
   A implementação do DIV é quase idêntica à do MUL. Dentro do módulo alucontrol, identificamos os códigos funct 3 (4) e funct 7 (1) do DIV, retornando o sinal alucontrol para a ALU. A ALU seleciona a operação a executar de acordo com o código alucontrol e, para o caso do DIV, retorna a divisão no aluout.

c. **Implementando a operação de “ou excludente com imediato” XORI**:
   Diferentemente das duas operações descritas acima, que são separadas a partir de um mesmo sinal aluop (2) para instruções do tipo algébrico ou

lógico, a operação XORI entra na categoria (3) das operações de imediatos. Em seguida, identificamos XORI com seu código funct 3 (4) e devolvemos o sinal para a ALU que seleciona a operação XOR. A ALU portanto retornará o ou excludente entre um imediato e o registrador dado executando isso bit a bit.

d. **Implementando a operação de "branch on equal" BEQ**

Para a implementação do branch primeiro verificamos quais partes do caminho de dados fornecido precisariam ser modificadas para que tal operação passasse a funcionar, verificou-se então que a "controlunit" precisava enviar os sinais corretos uma vez identificada a operação de branch fosse identificada, e a fetch unit deveria corretamente calcular o novo "PC" uma vez que o sinal de zero na "Alu" e o de "branch" fossem simultaneamente iguais a 1, além disso o novo "PC" deveria ser a soma do pc antigo com o imediato que vinha da memória de instrução.

Assim, o primeiro passo foi adicionar um caso a mais na "controlunit" para quando o "opcode" fosse um código de branch, no caso "7'b1100011", e dentro desse caso passamos os valores das variáveis para que uma branch fosse corretamente executada, que no caso é deixar o sinal de "branch" igual a 1, o de "aluop" igual a 1 também, pois esse código realiza uma subtração na "Alu", que serve para verificar se os dois valores comparados da branch são iguais e disparar um 1 no sinal do zero que sai da "Alu" indicando que os valores são realmente iguais, esse sinal de zero entra numa porta "and" junto com o sinal de "branch" para que a branch seja executada caso ambos sejam 1.

Além disso, foi necessário alterar a variável "new_pc" pois o "sigext" que carrega o imediato para ser somado com o "PC" estava sendo deslocado duas vezes para a esquerda " << 2" porém o sinal de imediato já vinha com 32 bits uma vez que no "controlunit" passamos ele para 32 bits quando a operação de branch fosse identificada.

Assim com essas modificações a instrução de branch está funcionando corretamente de acordo com testes feitos.

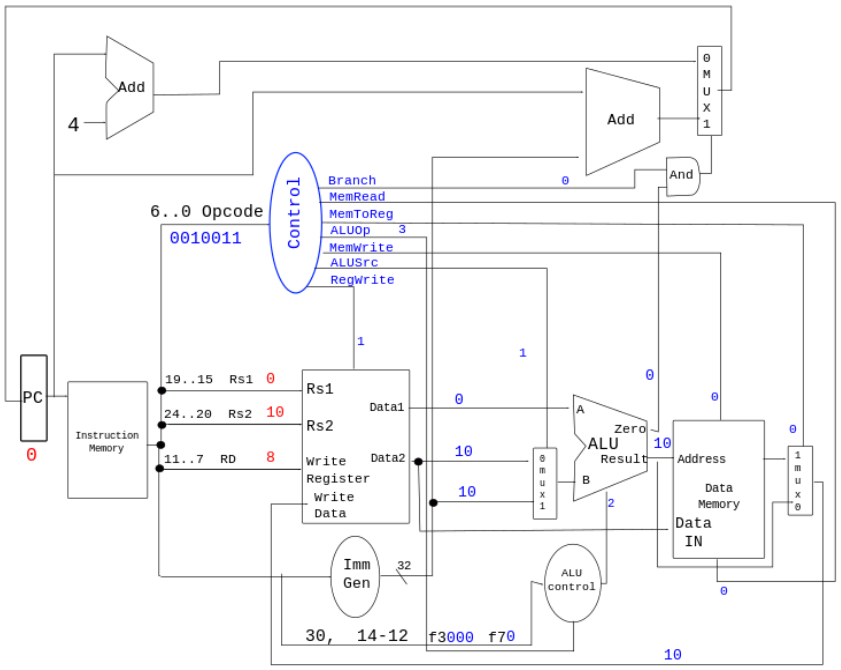
# 3. Testes

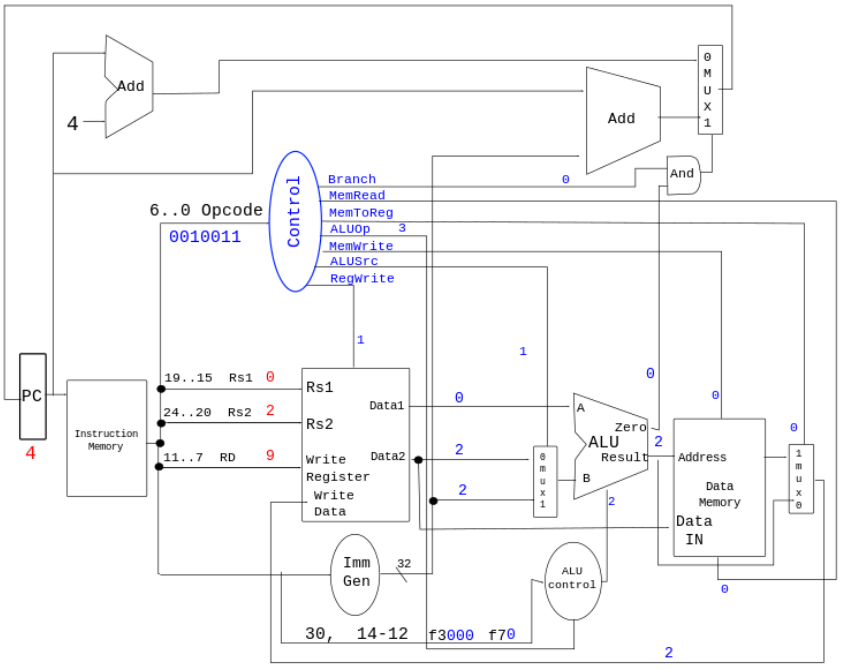
Os testes foram criados visando observar o comportamento das alterações e verificar se as operações implementadas abordam todas as possíveis possibilidades de operações, garantindo o funcionamento correto para todo e qualquer caso, seja ele padrão ou um caso de borda. Os testes serão explicitados abaixo, cada um referente à sua operação.

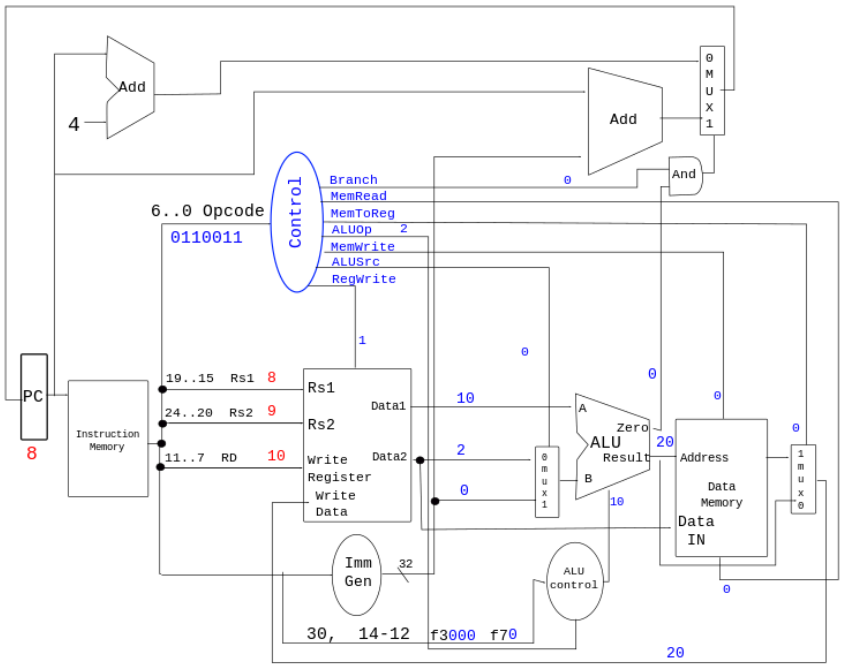
a. Testes referentes a operação **MUL**:

Teste 1: Para verificar se a multiplicação de dois inteiros está funcionando foi implementado o seguinte teste que verifica se a multiplicação de 10 por 2 está correta. O teste retornou 5 o que está correto. O código Assembly será especificado abaixo:

addi s0, zero, 10
addi s1, zero, 2
mul a0, s0, s1

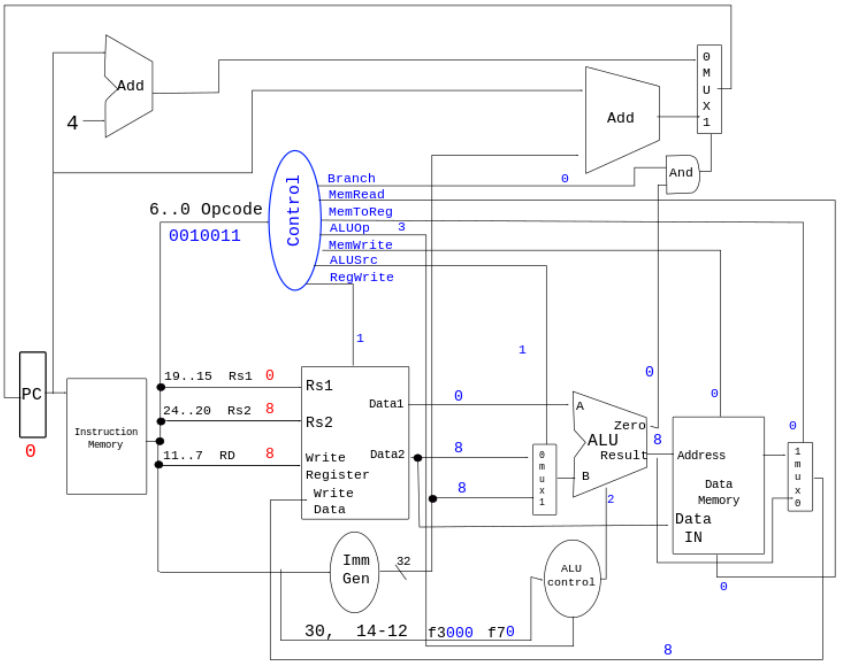


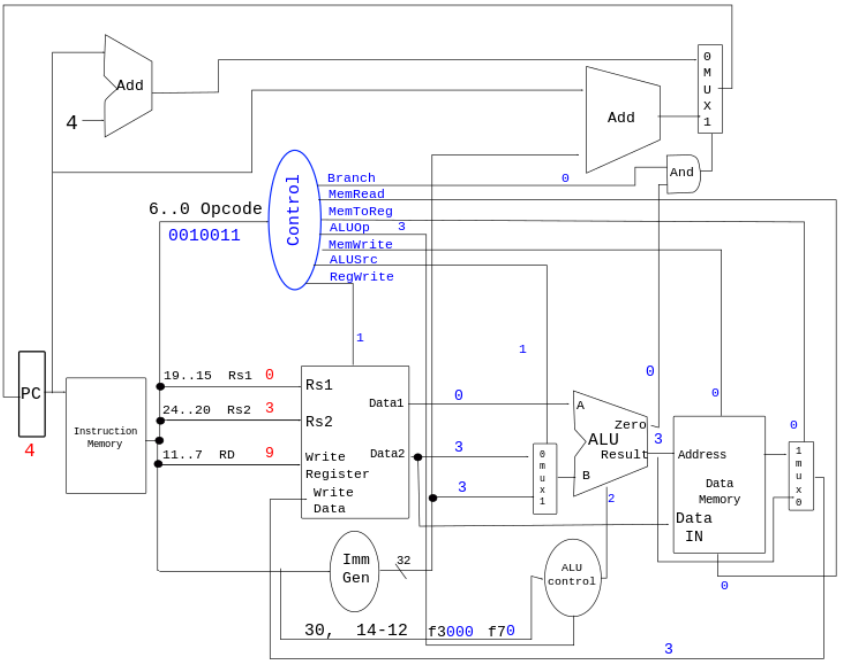



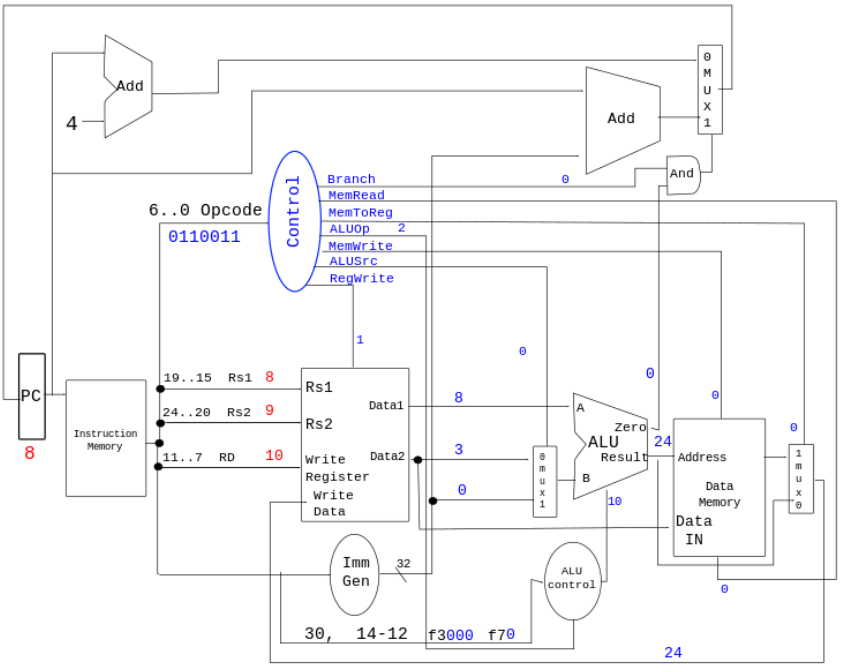



Teste 2: Similarmente ao teste 1, neste verifica-se se a multiplicação de 8 por 3 está correta. O teste retornou 24 a ser guardado em a0, o que mostra mais uma vez que a multiplicação está funcionando corretamente. Código Assembly abaixo:

addi s0, zero, 8
addi s1, zero, 3
mul a0, s0, s1



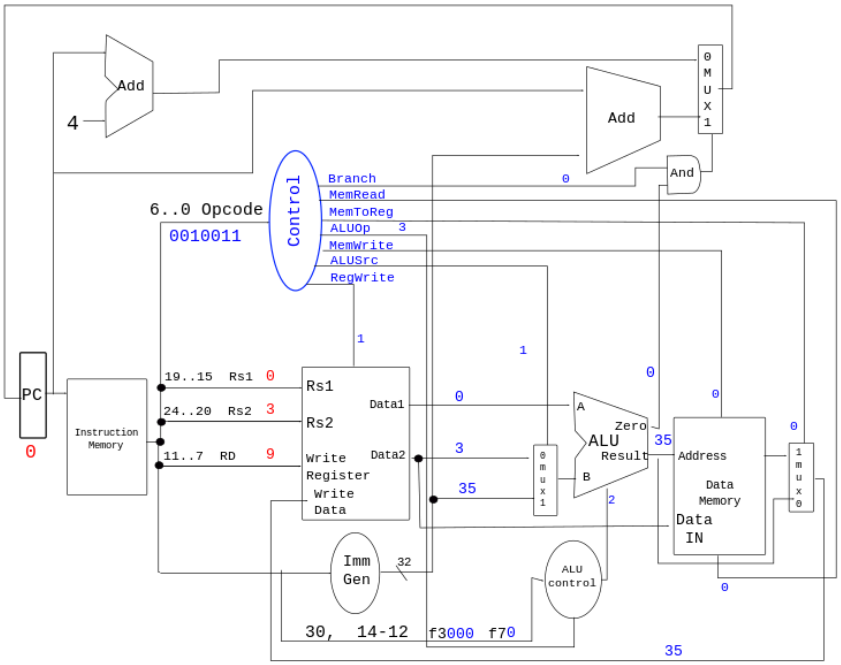


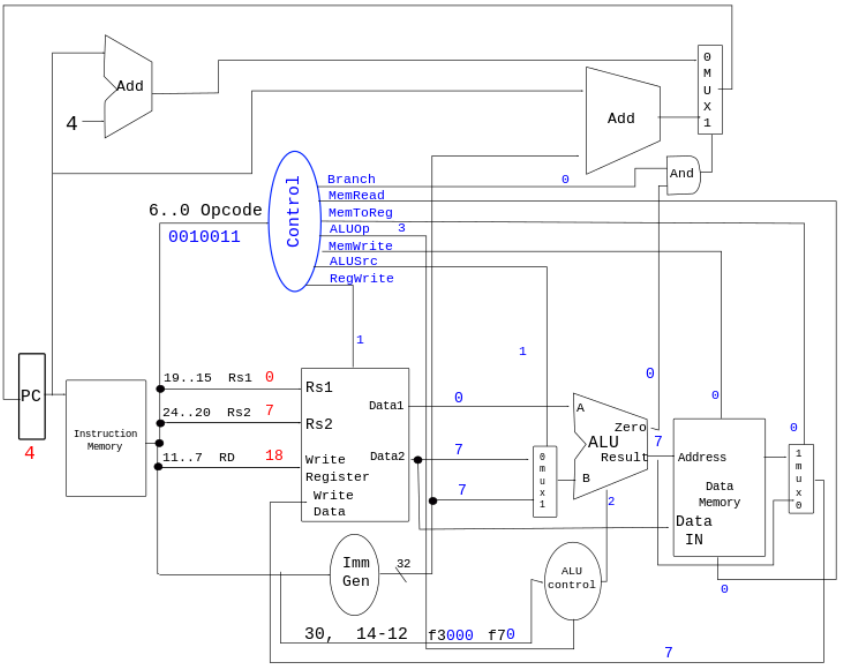



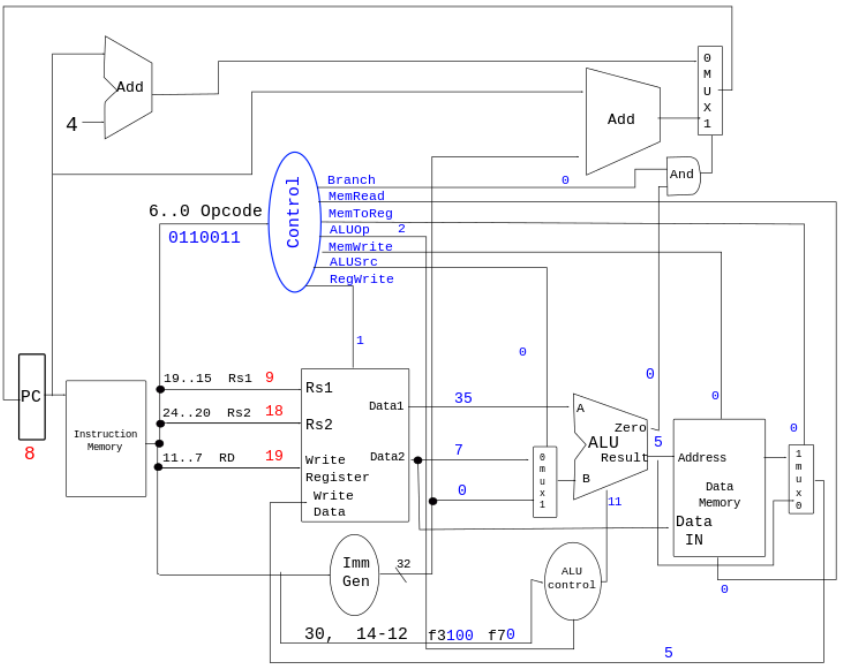

b. Testes referentes a operação **DIV**:

Teste 1: Para verificar se a operação de divisão está correta, foi realizado o seguinte teste que verifica se a divisão de 35 por 7 está correta. O teste retornou corretamente o valor 5, demonstrando que a divisão está funcionando Código Assembly abaixo:

```
addi s0, zero, 35
addi s1, zero, 7
div a0, s0, s1
```

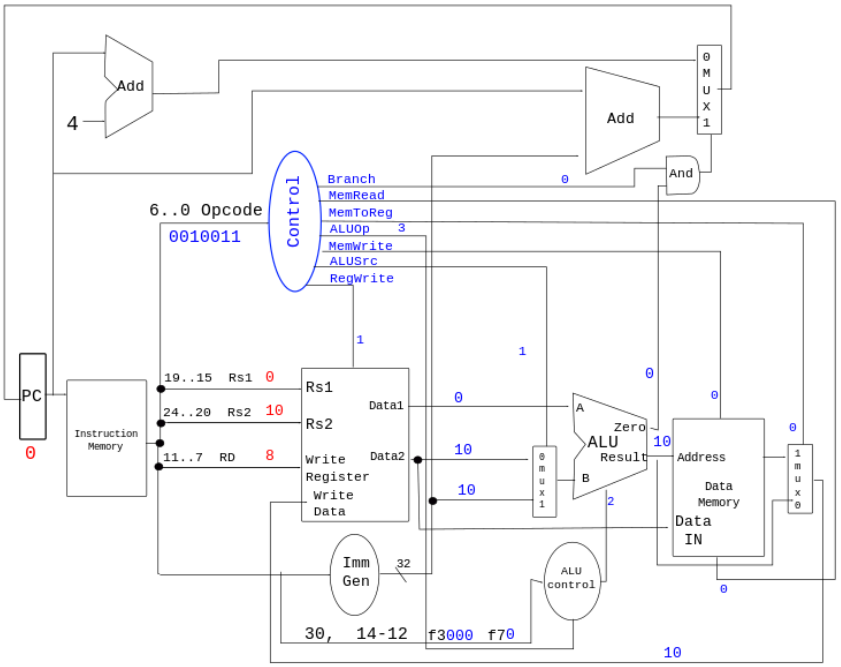

Add
4
Add
0
M
U
X
1
And
Branch
MemRead
MemToReg
ALUOp
MemWrite
ALUSrc
RegWrite
Control
6..0 Opcode
0010011
PC
0
Instruction Memory
19..15 Rs1
24..20 Rs2
11..7 RD
Rs1
Rs2
Write Register
Write Data
Data1
Data2
A
B
ALU
Zero
Result
Address
Data Memory
Data IN
Imm Gen
32
ALU control
30, 14-12
f3000 f70
35

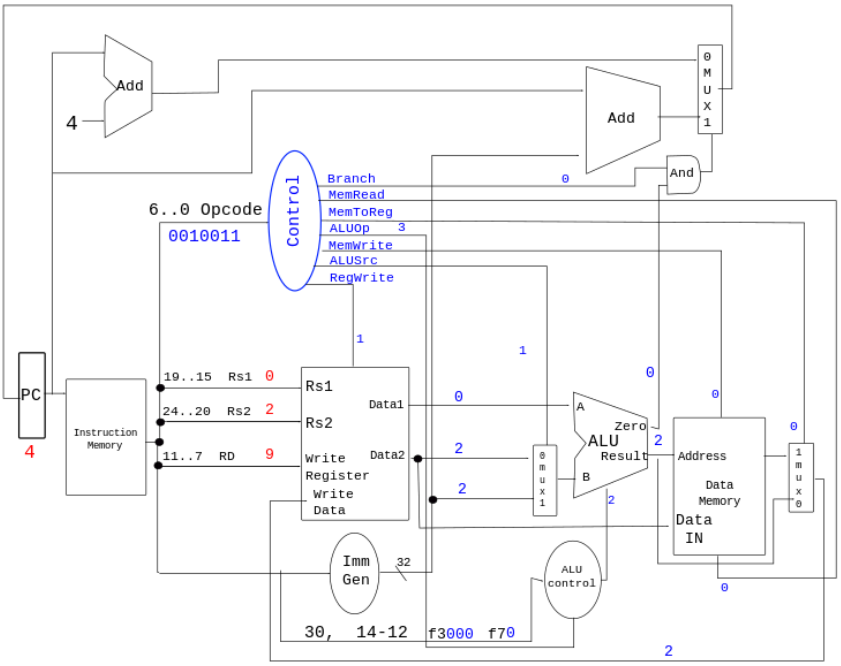

Add
4
Add
0
M
U
X
1
And
Branch
MemRead
MemToReg
ALUOp
MemWrite
ALUSrc
RegWrite
Control
6..0 Opcode
0010011
PC
4
Instruction Memory
19..15 Rs1
24..20 Rs2
11..7 RD
Rs1
Rs2
Write Register
Write Data
Data1
Data2
A
B
ALU
Zero
Result
Address
Data Memory
Data IN
Imm Gen
32
ALU control
30, 14-12
f3000 f70
7

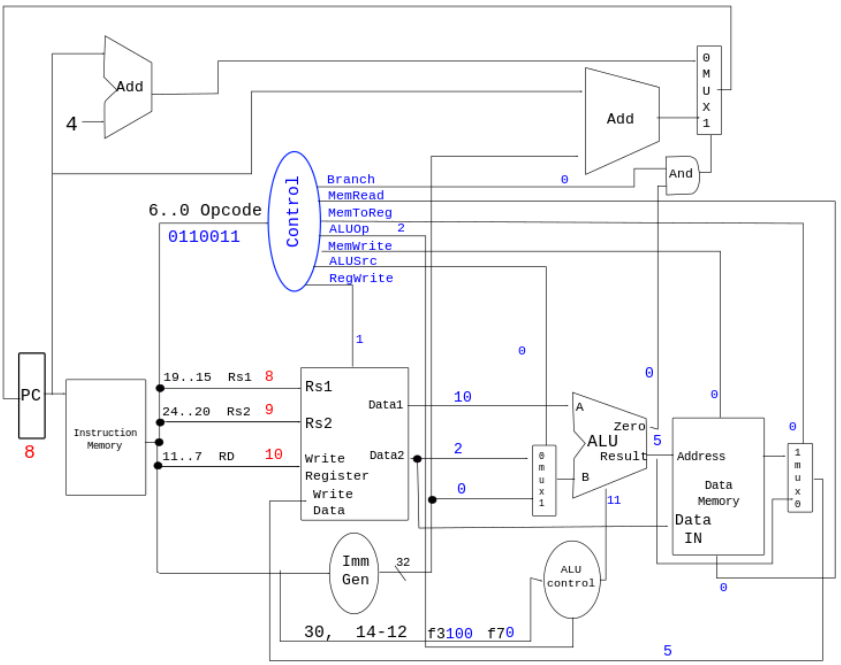



Teste 2: Similarmente ao teste 1, verificamos se a divisão está correta de 10 por 2. O programa retornou o valor 5 a ser guardado em a0, o que mostra que a divisão está funcionando corretamente dado o código Assembly abaixo:

addi s0, zero, 10
addi s1, zero, 2
div a0, s0, s1



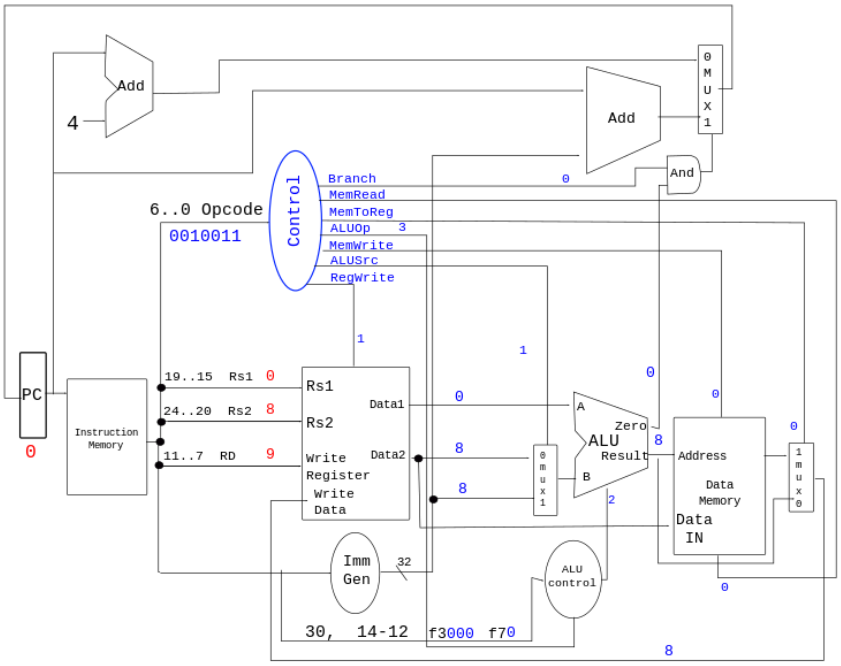


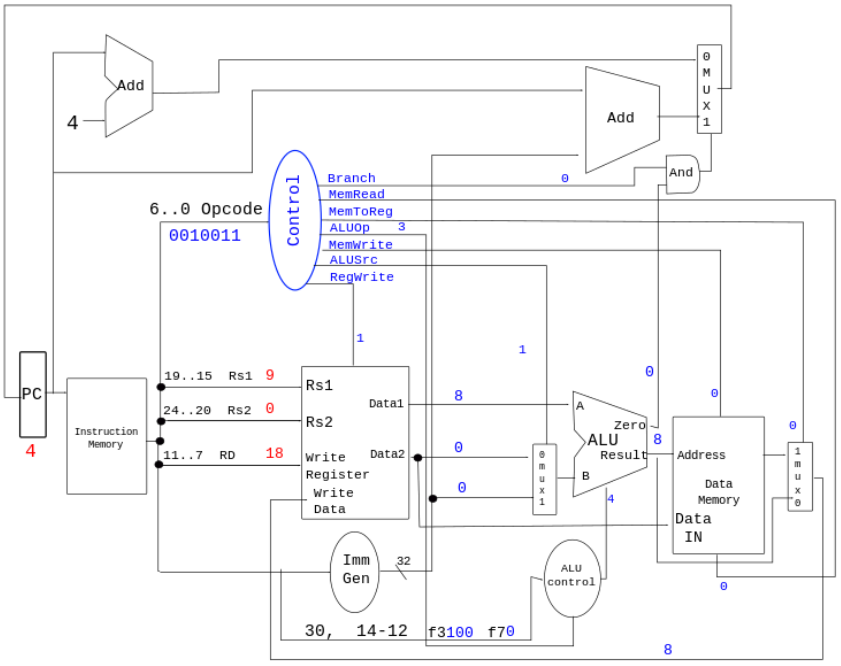




c. Testes referentes a operação **XORI**:

Teste 1: O teste foi criado com a função de verificar se a operação XORI foi implementada corretamente, primeiramente verificando se funciona para “s1 = 8” e um ou excludente com 0 que deve retornar 8, o que de fato acontece como pode ser visto na segunda figura abaixo, indicando que a operação funcionou corretamente. Código Assembly abaixo:

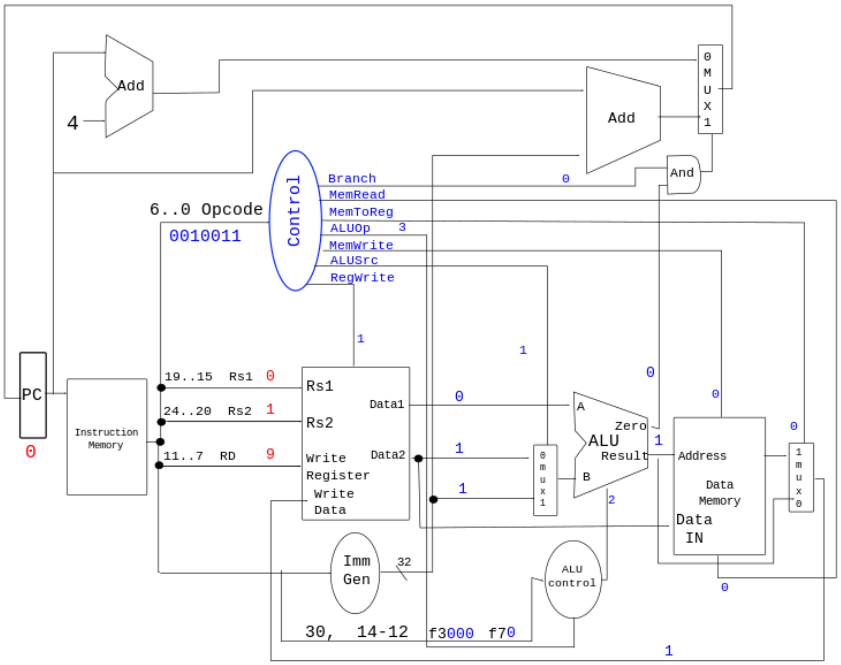
addi s1, zero, 8
xori s2, s1, 0

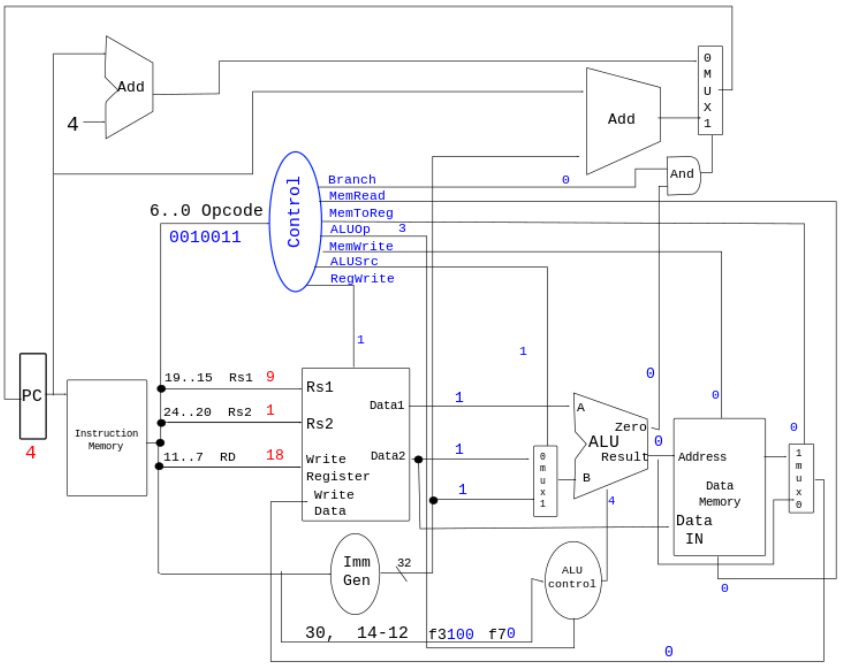



Teste 2: Similarmente, o teste 2 foi criado para o XORI entre o valor 1 e outro valor 1, o que deve retornar um valor 0 a ser guardado em s2, como aconteceu conforme previsto verificou-se que esta operação opera corretamente. Código Assembly abaixo:

addi s1, zero, 1
xori s2, s1, 1

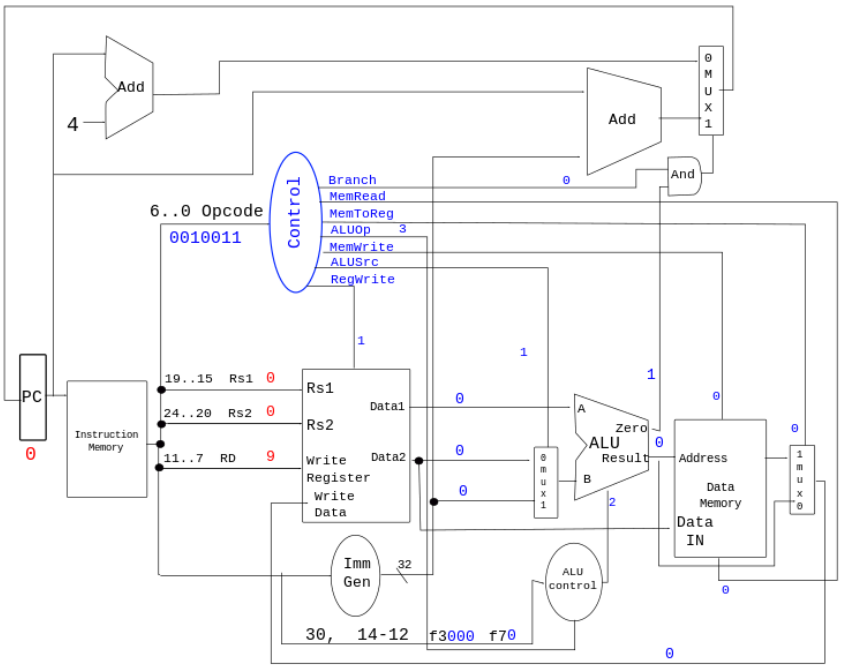


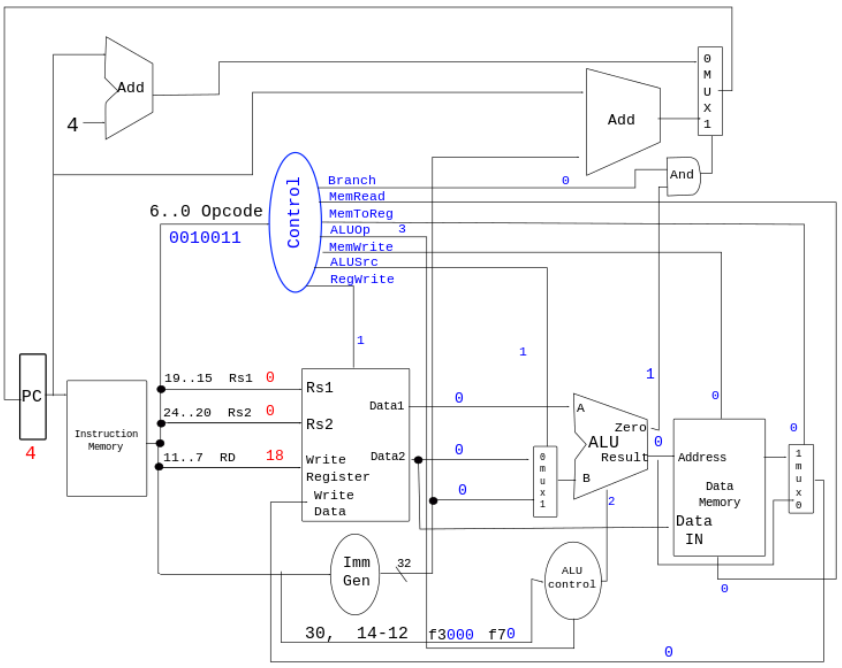



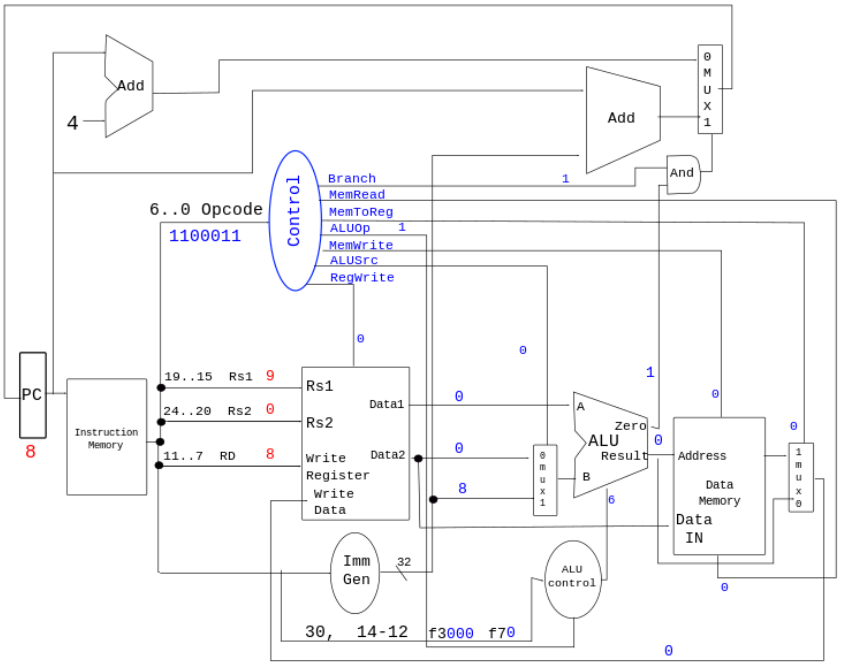

d. Testes referentes a operação **BEQ**:

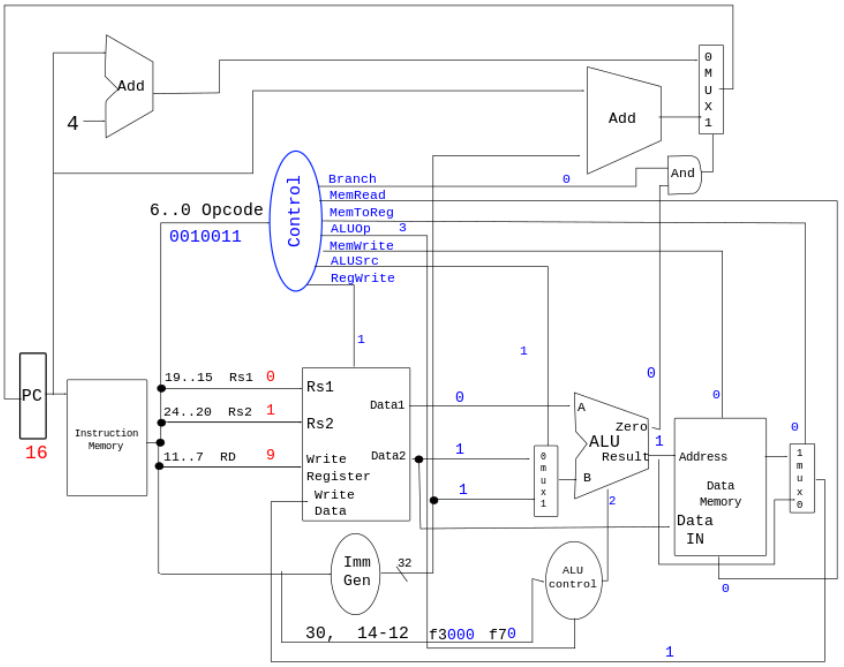
Teste 1: O teste foi criado com o objetivo de verificar se a operação de Branch está sendo executada corretamente, para isso verifica-se se a execução do branch faz com que somente os comandos necessários sejam executados enquanto os outros “addi s2, zero, 5” são pulados, nesse caso. Código Assembly abaixo:

```
addi s1, zero, 0
addi s2, zero, 0
beq s1, zero, ZERO
addi s2, zero, 5
ZERO:
addi s1, zero, 1
```



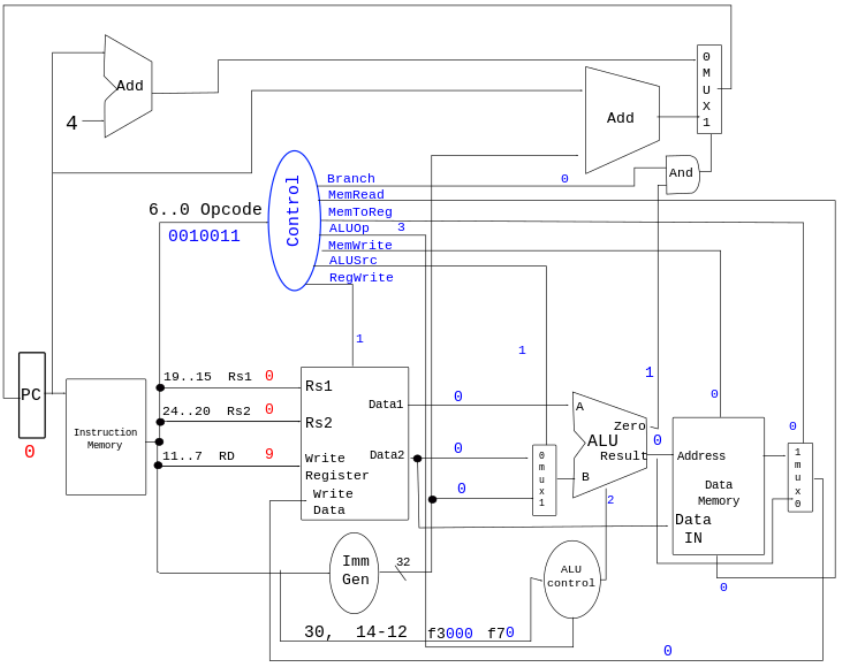


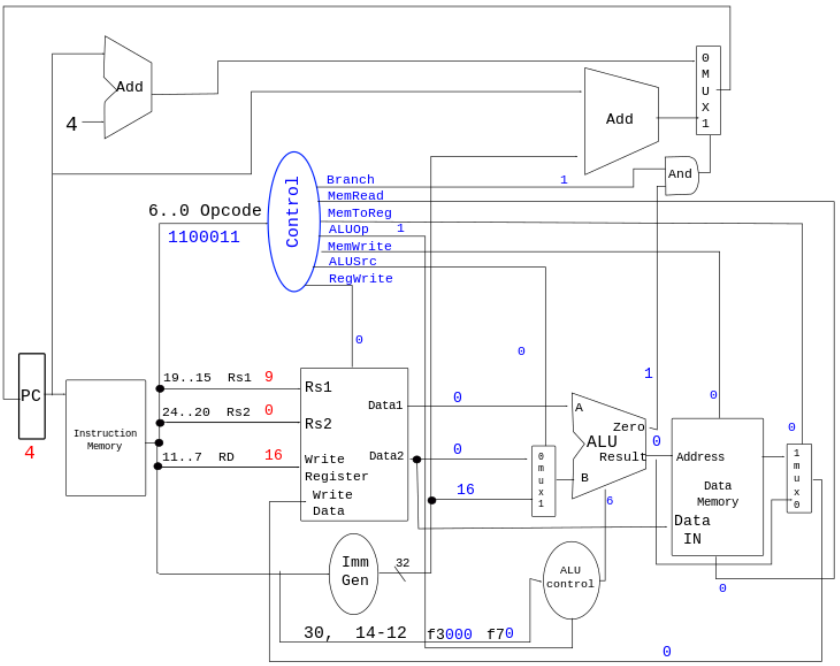



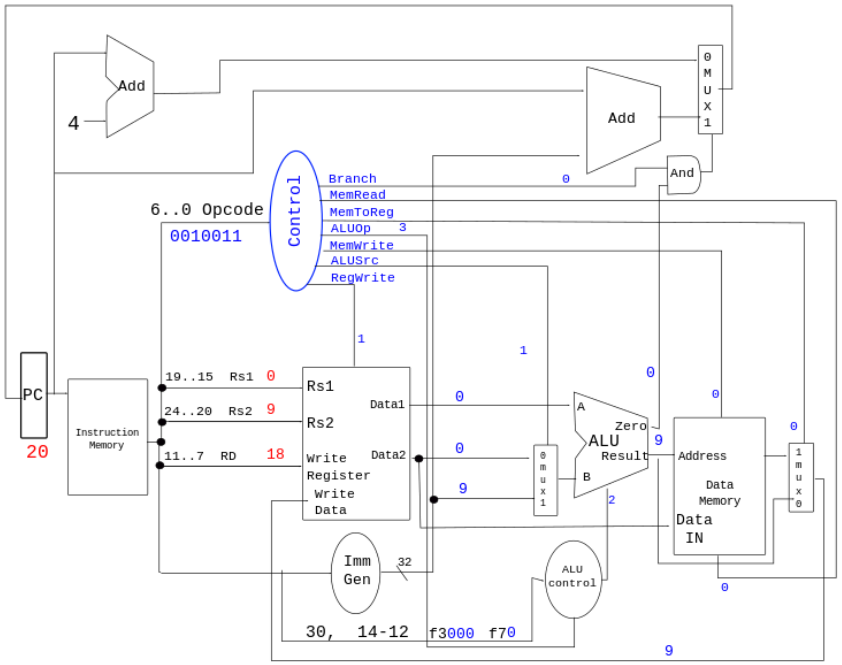

Teste 2: De maneira semelhante, o teste 2 tenta verificar se o branch funciona corretamente, dessa vez realizando mais operações de branch ao longo da operação. Nesse teste a ordem das linhas esperada era 1, 2, 6, 7, 8, 3, 4, 5, 9, tal ordem foi observada assim como pode se observar nas 9 imagens abaixo, mostrando que a operação de branch está corretamente implementada e funcionando ainda que muitas branches sejam executadas em sequência. Código Assembly abaixo:

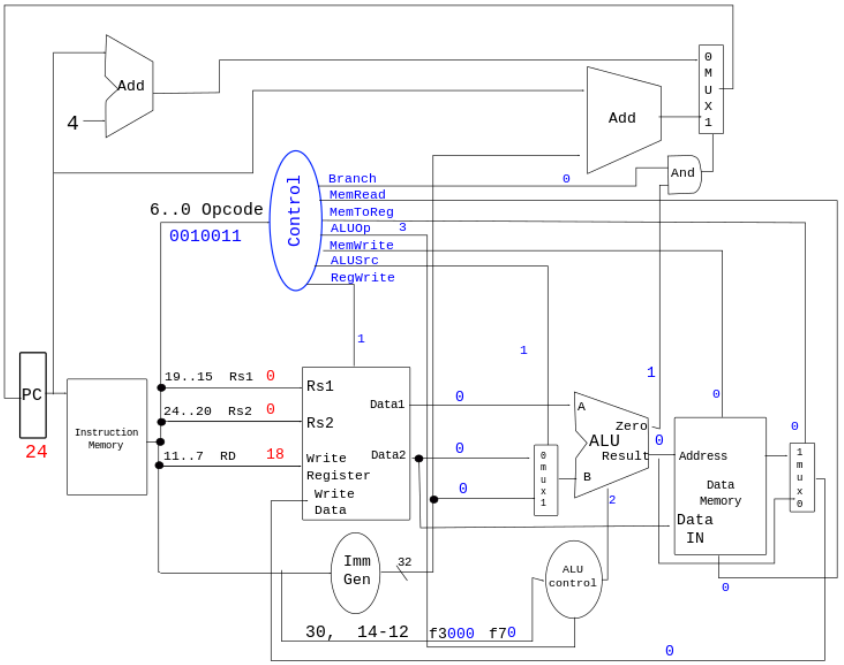
```
addi s1, zero, 0
beq s1, zero, EQ
```

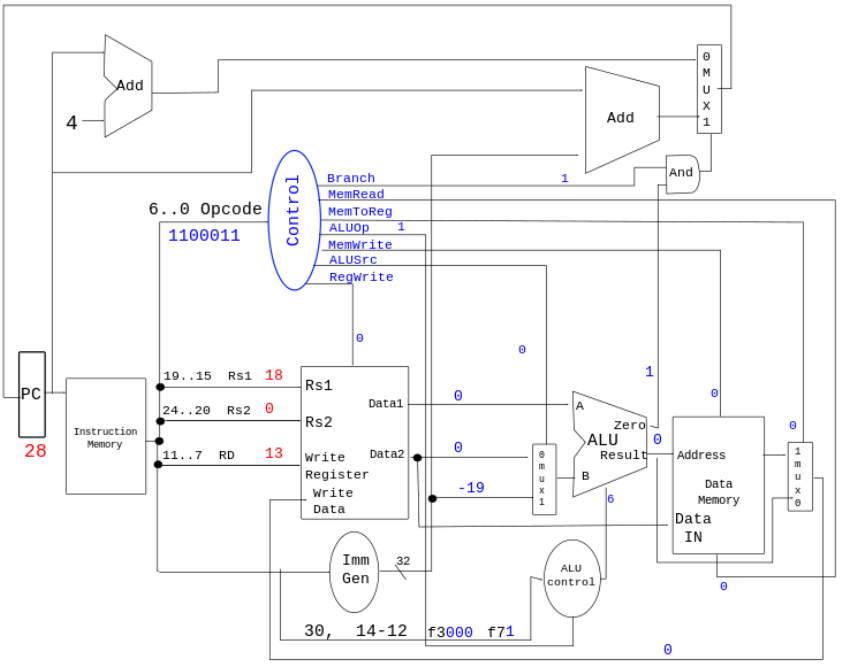
volta: addi s2, zero, 7
addi s3, zero, 8
beq s1, zero, fim
EQ: addi s2, zero, 9
addi s2, zero, 0
beq s2, zero, volta
fim: addi s3, zero, 17

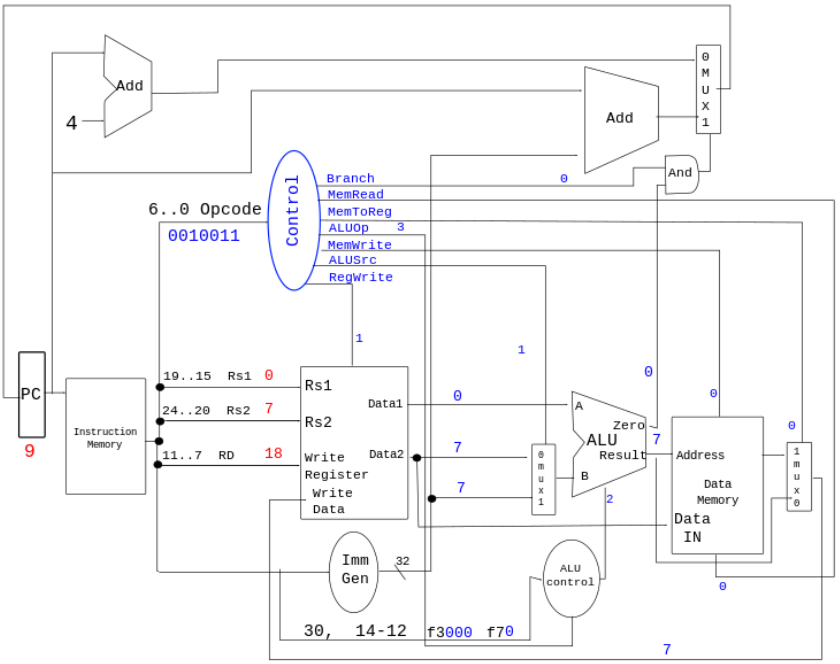


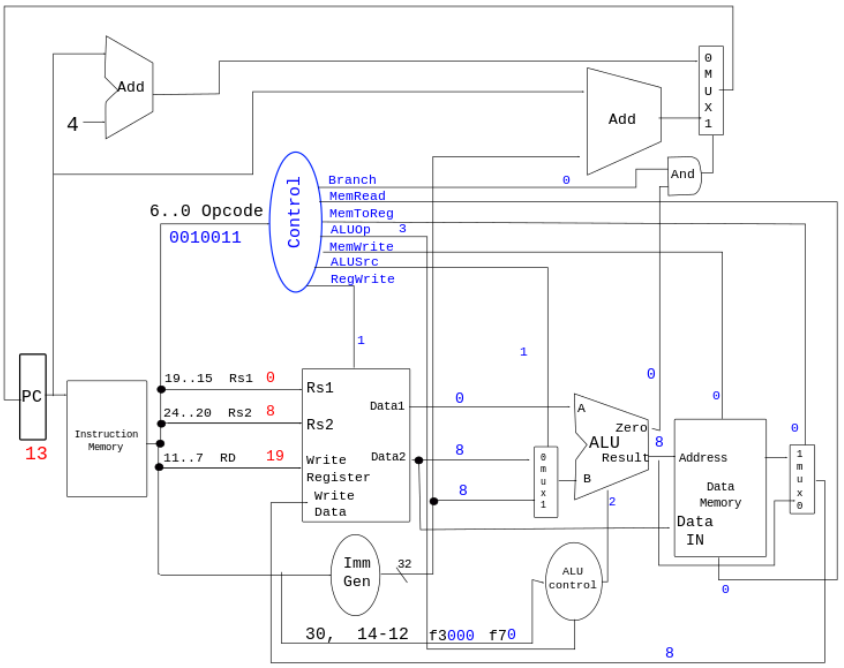



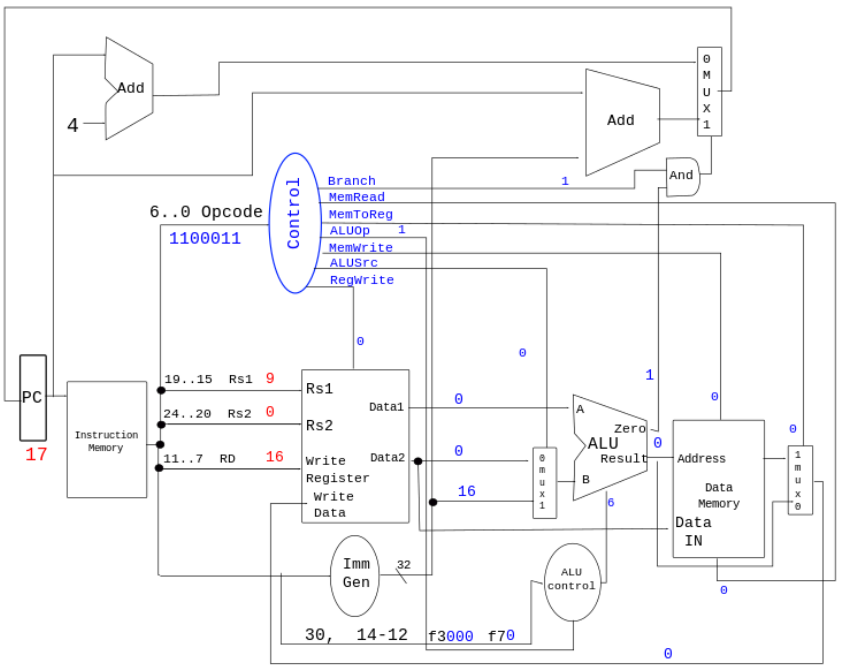

Add
4
Add
0
M
U
X
1
And
Branch
MemRead
MemToReg
ALUOp
MemWrite
ALUSrc
RegWrite
Control
6..0 Opcode
0010011
PC
20
Instruction Memory
19..15 Rs1
24..20 Rs2
11..7 RD
18
Rs1
Rs2
Write Register
Write Data
Data1
Data2
A
B
ALU
Zero
Result
Address
Data Memory
Data IN
Imm Gen
32
ALU control
30, 14-12
f3000 f70

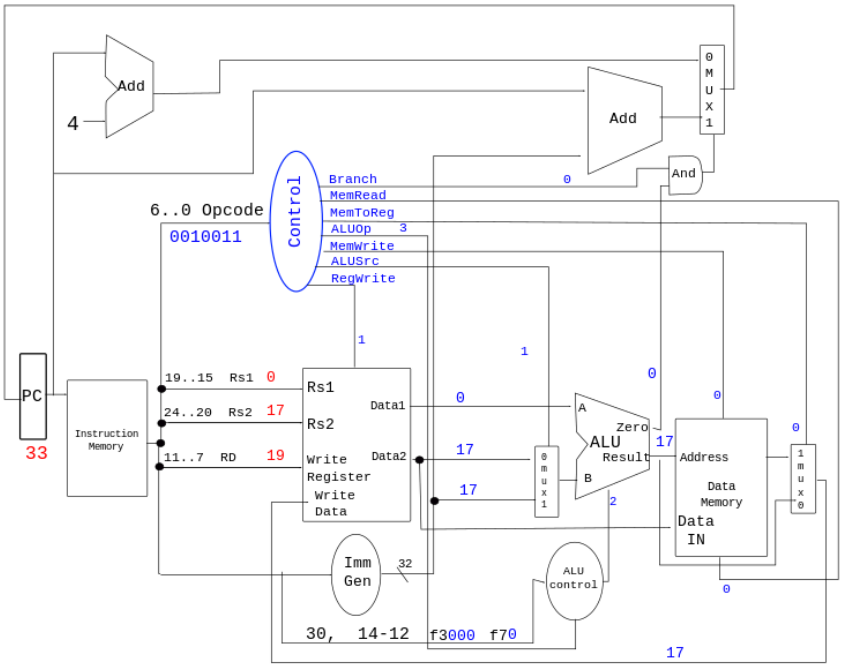

Add
4
Add
0
M
U
X
1
And
Branch
MemRead
MemToReg
ALUOp
MemWrite
ALUSrc
RegWrite
Control
6..0 Opcode
0010011
PC
24
Instruction Memory
19..15 Rs1
24..20 Rs2
11..7 RD
18
Rs1
Rs2
Write Register
Write Data
Data1
Data2
A
B
ALU
Zero
Result
Address
Data Memory
Data IN
Imm Gen
32
ALU control
30, 14-12
f3000 f70


Add
4
Add
0
M
U
X
1
And
Control
Branch
MemRead
MemToReg
ALUOp
MemWrite
ALUSrc
RegWrite
6..0 Opcode
1100011
1
1
PC
28
Instruction
Memory
19..15 Rs1 18
24..20 Rs2 0
11..7 RD 13
Rs1
Rs2
Write
Register
Write
Data
Data1
Data2
0
0
0
-19
A
B
ALU
Result
Zero
0
1
0
0
Address
Data
Memory
Data
IN
0
1
m
u
x
0
6
Imm
Gen
32
ALU
control
30, 14-12 f3000 f71
0


Add
4
Add
0
M
U
X
1
And
Control
Branch
MemRead
MemToReg
ALUOp
MemWrite
ALUSrc
RegWrite
6..0 Opcode
0010011
0
3
1
1
PC
9
Instruction
Memory
19..15 Rs1 0
24..20 Rs2 7
11..7 RD 18
Rs1
Rs2
Write
Register
Write
Data
Data1
Data2
0
7
7
A
B
ALU
Result
Zero
7
0
0
Address
Data
Memory
Data
IN
0
1
m
u
x
0
2
Imm
Gen
32
ALU
control
30, 14-12 f3000 f70
7


Add
4
Add
0
M
U
X
1
And
Branch
MemRead
MemToReg
ALUOp
MemWrite
ALUSrc
RegWrite
6..0 Opcode
0010011
Control
PC
13
Instruction Memory
19..15 Rs1
24..20 Rs2
11..7 RD
Rs1
Rs2
Write Register
Write Data
Data1
Data2
A
B
ALU
Zero
Result
Address
Data Memory
Data IN
1
m
u
x
0
Imm Gen
32
ALU control
30, 14-12
f3000 f70


Add
4
Add
0
M
U
X
1
And
Branch
MemRead
MemToReg
ALUOp
MemWrite
ALUSrc
RegWrite
6..0 Opcode
1100011
Control
PC
17
Instruction Memory
19..15 Rs1
24..20 Rs2
11..7 RD
Rs1
Rs2
Write Register
Write Data
Data1
Data2
A
B
ALU
Zero
Result
Address
Data Memory
Data IN
1
m
u
x
0
Imm Gen
32
ALU control
30, 14-12
f3000 f70




Também foi testado o caso de a branch não ser tomada, o que funcionou corretamente e a instrução abaixo dela foi executada normalmente, sem o branch ter sido tomado

# 4. Conclusão

O trabalho consistiu portanto da modificação de módulos e unidades do RISC-V fornecido em Verilog para que a implementação passasse a suportar as operações de multiplicação (mul), divisão (div), ou excludente com imediato (xori), e branch on equal “ramo em igualdade” (beq). Para iniciar a resolução do trabalho, após a leitura do enunciado, analisamos o código fornecido de maneira geral, e realizamos alguns testes com as operações já implementadas como a de soma (add) e de subtração (sub) para nos familiarizarmos com o modo de testes e de visualização fornecidos.

Após isso, para iniciar a implementação da operação “mul” identificamos, na imagem do caminho de dados que é exibida no final do notebook, quais sinais eram liberados e o que acontecia no caminho de dados quando uma operação de mul é executada, com isso e com base nos conhecimentos adquiridos nas aulas de Organização de Computadores identificamos que a operação de mul por ser uma operação tipo R é bem parecida, em sua implementação com a operação “add” por exemplo, assim, para prosseguir com a implementação identificamos onde era implementada a operação add e partimos daí para implementar a mul. Primeiro implementamos a operação “ * ” na Alu e depois criamos um novo código no “Alu control” para chamar a multiplicação, o que causou erro pois o funct3 tem apenas 3 bits e colocamos a operação de funct 3 com número 8 o que causou erro, para resolver isso usamos o conhecimento de que o funct 7 é diferente para o “add” e para

o “mul” assim, caso o funct 3 fosse 0 verificamos se o funct 7 é zero e caso seja 1 executamos a operação de mul.

O mesmo aconteceu para a implementação do “div”, que, após implementarmos a operação de divisão na Alu, mudamos o alucontrol para  caso o funct 7 fosse 1 e o funct 3 fosse 4, executamos a div, porém, caso o funct 7 fosse 0 executamos a operação “XOR”. Semelhantemente fizemos com o xori, caso o aluop fosse 3, o que significa trabalhar com o imediato, e o funct 3 fosse 4, antes tinha-se uma operação Nop nesse código, ou seja, ele não executava operações, assim substituímos essa operação pelo xori, habilitando nosso RISC-V a realizar tal operação.

Para a implementação do branch, como citado na parte de desenvolvimento, o mais trabalhoso foi entender que sinais precisariam ser enviados pelo controle e fazer com que o pc fosse atualizado corretamente uma vez que já enviávamos o imediato com 32 bits, “estendido” e não precisávamos transladar o valor do “sigext” duas vezes para a esquerda, o que estava sendo feito como padrão na implementação fornecida, esse foi um grande desafio na implementação e acreditamos que caso não mudássemos o imediato no módulo de controle quando se chama uma branch, poderíamos manter a linha de “new_pc” conforme fornecida que seria equivalente.

Por fim, finalizamos destacando a pertinência deste trabalho em relação ao que foi trabalhado na disciplina, pois, além de consolidar os conhecimentos a respeito do caminho de dados, do funcionamento da alu, do pc e do módulo de controle ainda foi importante para consolidar o conhecimento em assembly, utilizado para criar casos de teste, e em verilog, linguagem utilizada para a resolução do trabalho.