UNIVERSIDADE FEDERAL DE MINAS GERAIS

Lucca Alvarenga de Magalhães Pinto

Lucas Rafael Costa Santos

Lucas Ferreira Pedras

TRABALHO PRÁTICO 2 -

Organização de Computadores 1

BELO HORIZONTE, Dezembro de 2024

Sumário

1. Introdução

Este projeto tem como objetivo utilizar a Linguagem de Descrição de Hardware Verilog, junto com os conceitos ensinados ao longo da matéria de Organização de Computadores 1, para demonstrar o uso de um processador e suas instruções.

Dado uma implementação de um processador RISC-V de 5 estágios, disponibilizado em um arquivo .ipynb no Google Colab, a tarefa principal é modificar o caminho de dados fornecido para adicionar 4 novas operações no ambiente: operação lógica XORI imediata (‘xori’), desvio condicional igual (‘beq’), multiplicação (‘mul’) e divisão (‘div’).

Dessa forma, neste relatório será explicado as modificações feitas no arquivo original para que as novas instruções fossem executadas, assim como, a explicação de cada teste realizado para demonstrar o funcionamento de cada operação.

O enunciado do trabalho pode ser acessado pelo link: Enunciado. E o código final implementado pelo link: Implementação_Final.

2. Implementação

Com base no ambiente de execução disponibilizado, o código original apresenta uma implementação do RISC-V com algumas instruções já previamente implementadas:

- Operações do Tipo-R: (Operações Aritméticas e Lógicas) ADD, SUB, SLL, SLT, SLTU, XOR, SRA, OR, AND, SRL.
- Operações do Tipo-I: (Operações Imediatas) ADDI, SLLI, SLTUI, SRAI, SRLI, ORI, ANDI.
- Carregamento de dados: LW.
- Armazenamento de dados: SW.
- Desvios condicionais: BNE, BLT, BGE, BLTU, BGEU.

Entre essas instruções, algumas foram utilizadas nos testes das novas operações e suas implementações não foram alteradas. Em relação ao processador ele foi disponibilizado com 5 unidades:

- Fetch Unit: Responsável por buscar a instrução da memória. Possui 2 componentes principais: “PC” (Contador de Programa) que controla o endereço da próxima instrução a ser buscada, e a “instrução de memória” que armazena e fornece as instruções baseadas no endereço atual do “PC”.

- Decode Unit: Responsável por decodificar a instrução e preparar os sinais de controle. Seus componentes são “ControlUnit” que gera sinais de controle com base no “opcode”, e “Register_Bank” que armazena e fornece dados dos registradores.

- Execution Unit: Responsável por executar as operações aritméticas e lógicas. Possui a “ALU” (Unidade Aritmética Lógica) que realiza os cálculos baseados nos sinais de controle da “alucontrol” (determina a operação da ALU com base em “aluop” e “funct”).

- Memory Unit: Responsável por ler e escrever dados na memória com base nos sinais de leitura ou escrita.

- Writeback Unit: Responsável por escrever os resultados de volta nos registradores.

Nesse contexto, abaixo é explicado como cada operação nova foi implementada no processador:

2.1. XORI

A operação XORI é uma instrução do Tipo-I que realiza a operação lógica XOR entre o conteúdo de um registrador de origem e um valor imediato, armazenado o resultado em um registrador de destino.

Em um processador RISC-V de 5 estágios, ao executar a operação XORI ocorre as seguintes etapas: primeiro na unidade “Fecth” é feito a busca da instrução na memória; depois na unidade “Decode” a instrução é decodificada para identificar

os registradores de origem, destino e o valor imediato, dessa forma, os sinais de controle são gerados para determinar a operação da ALU e o uso do imediato; com isso na unidade “Execution” a ALU realiza a operação XOR entre o valor do registrador de origem e o imediato, o resultado é calculado e preparado para o próximo estágio; a unidade “Memory” é inativa para operação já que não há acesso à memória de dados; por fim, na unidade “Writeback” o resultado da ALU é escrito de volta no registrador de destino.

Sendo assim, os únicos módulos que são necessários modificações para implementar a instrução XORI são o módulo “ALU” e o módulo “alucontrol”. As modificações feitas no código em cada unidade são essas:

- A unidade “FechtUnit” não tem nenhuma modificação necessária no código. Essa parte do código apenas busca a instrução XORI na memória.
- Na unidade “DecodeUnit” no módulo “ControlUnit” já está implementado a opção quando o “opcode” verificar uma instrução do Tipo-I, ou seja, quando ele for igual a “0010011”, para assim identificar os sinais de controle: “alusrc ” igual a 1 identificando o uso de imediato, “regwrite” igual a 1 habilitando escrita no registrador de dados, “aluop” igual a 3 para identificar uma instrução imediata, e “ImmGen” gera o imediato estendido para 32 bits.

```
module ControlUnit ...
...
    7'b0010011: begin // TIPO-I (ANDI)
          alusrc   <= 1;
          regwrite <= 1;
          aluop    <= 3;
          ImmGen   <= {{20{inst[31]}},inst[31:20]};
    end
    endcase
  end
endmodule
```

- Na unidade “ExecutionUnit” no módulo “alucontrol” é definido o valor de “alucontrol” igual a 4 para XORI com base no “funct3”:

```
module alucontrol ...
...
  always @(*)
  begin
    case (aluop)
      ...
      3: case (funct3) // immediate
         4: alucontrol <= 4'd4; // XORI
         ...
         endcase
    endcase
  end
endmodule
```

- Ainda na unidade “ExecutionUnit” no módulo “ALU” é realizado a operação XOR:

```
module ALU ...
...
  always @(*) begin
      case (alucontrol)
        4: aluout <= A ^ B; // XOR
      ...
      endcase
  end
endmodule
```

- A unidade “MemoryUnit” não tem nenhuma modificação necessária no código, já que não é utilizado para XORI, pois não há acesso à memória de dados.
- A unidade “WritebackUnit” não tem nenhuma modificação necessária no código, apenas escreve o resultado da operação de volta no registrador de destino

Dessa forma, a operação XORI consegue ser feita no ambiente de execução fornecido corretamente.

2.2. BEQ

A operação BEQ é uma instrução de desvio condicional que verifica se dois registradores são iguais, se forem, o PC (Contador de Programa) é atualizado para desviar para um endereço especificado por um deslocamento imediato.

Em um processador RISC-V de 5 estágios, ao executar a operação BEQ ocorre as seguintes etapas: primeiro na unidade “Fecth” é feito a busca da instrução na memória, de forma que incrementa o PC para a próxima instrução; depois na unidade “Decode” a instrução é decodificada para identificar os registradores fontes, os valores dos registradores são lidos, o deslocamento imediato é extraído e sinalizado; com isso na unidade “Execution” os valores dos registradores são comparados, se forem iguais calcula o novo endereço de desvio, caso contrário o PC continua sequencial; a unidade “Memory” não é utilizada já que em BEQ não há leitura ou escrita na memória de dados; e a unidade “Writeback” também não é utilizada já que BEQ não envolve escrita em registradores.

Sendo assim, o único módulo que é necessário modificação para implementar a instrução BEQ é o módulo “ControlUnit”. As modificações feitas no código em cada unidade são essas:

- A unidade “FechtUnit” não tem nenhuma modificação necessária no código. O controle do PC para saltos condicionais é feito através do sinal “branch” e do cálculo do novo PC, ou seja, essa unidade já lida com o novo PC se o salto foi tomado.
- Na unidade “DecodeUnit” no módulo “ControlUnit” é necessário acrescentar a opção quando o “opcode” verificar uma instrução do tipo branch, ou seja, quando ele for igual a “1100011”, para assim identificar os sinais de controle para BEQ: “branch” igual a 1 indicando que a instrução é uma operação que altera o fluxo de execução do programa, “aluop” igual a 1 definindo que a operação na ALU é para comparar valores (subtração para comparação de igualdade), e “ImmGen” gera o deslocamento correto.

```
module ControlUnit ...
 ...
    7'b1100011: begin // BEQ
          branch   <= 1;
          aluop    <= 1;
          ImmGen   <= {{19{inst[31]}}, inst[31], inst[7], inst[30:25],
inst[11:8], 1'b0}; // Imediato de branch
        end
    endcase
```

```
  end
endmodule
```

- Na unidade “ExecutionUnit” no módulo “alucontrol” já está definido na ALU o código da operação para subtração para fazer a comparação, então não precisa ser modificado. Pelo mesmo motivo não tem modificação no módulo “ALU”.
- A unidade “MemoryUnit” não tem nenhuma modificação necessária no código, já que não é utilizado para BEQ, pois não há acesso à memória de dados.
- A unidade “WritebackUnit” não tem nenhuma modificação necessária no código, já que usando BEQ não há dados para escrever de volta.

Dessa forma, a operação BEQ consegue ser feita no ambiente de execução fornecido corretamente.

2.3. MUL

A operação MUL é uma instrução do Tipo-R, ela realiza a multiplicação de dois valores inteiros. Ela toma dois registradores de origem, faz a multiplicação entre eles e armazena o resultado em um registrador de destino.

Em um processador RISC-V de 5 estágios, ao executar a operação MUL ocorre as seguintes etapas: primeiro na unidade “Fecth” é feito a busca da instrução na memória; depois na unidade “Decode” a instrução é decodificada e identificada como MUL e os registradores de origem são lidos; na unidade “Execution” a ALU realiza a operação de multiplicação dos valores nos registradores e o resultado é preparado para ser escrito no registrador de destino; a unidade “Memory” é ignorada para operação MUL já que não envolve leitura ou escrita na memória; por fim, na unidade “Writeback” o resultado da multiplicação é escrito no registrador de destino.

Sendo assim, com a operação MUL é identificada pelos campos “opcode”, “funct3” e “funct7”, os únicos módulos que são necessários modificações para implementar a instrução são o módulo “alucontrol” e o módulo “ALU”. As modificações feitas no código em cada unidade são essas:

- A unidade “FechtUnit” não tem nenhuma modificação necessária no código. Essa parte do código apenas busca a instrução MUL na memória.
- Na unidade “DecodeUnit” no módulo “ControlUnit”, como MUL é uma instrução do Tipo-R, então o “opcode” igual a “011011” identifica já a operação, e essa parte já estava implementada.
- Na unidade “ExecutionUnit” no módulo “alucontrol” precisamos verificar se “funct7” é igual a “8b00000001” e se “funct3” é igual a “3b000” para configurar o valor de “alucontrol” para multiplicação. Dessa forma adicionamos essa parte no código (anteriormente “funct3” igual a 0 seria uma operação de ADD ou SUB, mas agora verificamos de acordo com “funct7” se é pra ser ADD ou SUB ou MUL):

```
module alucontrol ...
...
  always @(*)
  begin
    case (aluop)
      ...
      2: case (funct3)
           0: alucontrol <= (funct7 == 8'b00000000) ? 4'd2 : // ADD
                            (funct7 == 8'b01000000) ? 4'd6 : // SUB
                            (funct7 == 8'b00000001) ? 4'd10 : 4'd15; //MUL
           ...
         endcase
          endcase
  end
endmodule
```

- Ainda na unidade “ExecutionUnit”, no módulo “ALU” devemos acrescentar a opção para operação de multiplicação, caso “alucontrol” seja igual a 10:

```
module ALU ...
...
    always @(*) begin
      case (alucontrol)
         10: aluout <= A * B; // MUL
         ...
    endcase
  end
```

```
endmodule
```

- A unidade “MemoryUnit” não tem nenhuma modificação necessária no código, já que é ignorada pela operação de MUL porque não envolve leitura ou escrita na memória.
- A unidade “WritebackUnit” não tem nenhuma modificação necessária no código, apenas escreve o resultado da operação de multiplicação no registrador de destino.

Dessa forma, a operação MUL consegue ser feita no ambiente de execução fornecido.

2.4. DIV

A operação DIV é uma instrução do Tipo-R que realiza a divisão inteira de dois números, ou seja, ela divide o valor do registrador de origem 1 pelo valor do registrador de origem 2 e armazena o quociente no registrador de destino.

Em um processador RISC-V de 5 estágios, ao executar a operação DIV ocorre as seguintes etapas: primeiro na unidade “Fecth” é feito a busca da instrução na memória; depois na unidade “Decode” a instrução é decodificada e identificada como DIVe os registradores de origem são lidos para obter os operandos; na unidade “Execution” a ALU realiza a divisão dos valores nos registradores e o resultado é preparado para ser guardado; a unidade “Memory” não é utilizada já que DIV não acessa a memória; por fim, na unidade “Writeback” o resultado da divisão é escrito no registrador de destino.

Sendo assim, com a operação DIV precisa ser identificada pelos campos “opcode”, “funct3” e “funct7”, os únicos módulos que são necessários modificações para implementar a instrução são o módulo “alucontrol” e o módulo “ALU”. As modificações feitas no código em cada unidade são essas:

- A unidade “FechtUnit” não tem nenhuma modificação necessária no código. Essa parte do código apenas busca a instrução DIV na memória.
- Na unidade “DecodeUnit” no módulo “ControlUnit”, como DIV é uma instrução do Tipo-R, então o “opcode” é igual a “011011”, o que

identifica já a operação. Como esse tipo de operação já estava implementado, então não teve nenhuma modificação.

- Na unidade “ExecutionUnit” no módulo “alucontrol” precisamos verificar se “funct7” é igual a “8b00000001” e se “funct3” é igual a “3b100” para configurar o valor de “alucontrol” para divisão. Dessa forma adicionamos essa parte no código (anteriormente “funct3” igual a 4 seria uma operação de XOR, mas agora verificamos de acordo com “funct7” se é pra ser XOR ou DIV):

```
module alucontrol ...
 ...
  always @(*)
  begin
    case (aluop)
      ...
      2: case (funct3)
           ...
           4: alucontrol <= (funct7 == 8'b00000000) ? 4'd4 : // XOR
                            (funct7 == 8'b00000001) ? 4'd11 : 4'd15; // DIV
           ...
         endcase
       endcase
  end
endmodule
```

- Ainda na unidade “ExecutionUnit”, no módulo “ALU” devemos acrescentar a opção para operação de divisão, caso “alucontrol” seja igual a 11. Foi adicionado uma condição para verificar se o valor do divisor é 0, e caso seja, então a resposta deve ser -1 para indicar erro. Além disso, os sinais de entrada “A” e “B” foram definidos como vetores de 32 bits com sinal, para fazer DIV com números negativos.

```
module ALU (input [3:0] alucontrol, input signed [31:0] A, B, output reg
[31:0] aluout, output zero);
...
  always @(*) begin
      case (alucontrol)
        11: aluout <= (B != 0) ? A / B : -1;  // DIV
        ...
    endcase
  end
endmodule
```

- A unidade “MemoryUnit” não tem nenhuma modificação necessária no código, já que DIV não acessa a memória.
- A unidade “WritebackUnit” não tem nenhuma modificação necessária no código, apenas escreve o resultado da operação de divisão no registrador de destino.

Dessa forma, a operação DIV consegue ser feita no ambiente de execução fornecido.

3. Testes

O código base disponibilizado para elaboração do projeto permite a verificação dos comandos passados para o processador, por meio da identificação de seu código hexadecimal, de forma que gera a imagem dos ciclos para realização daquela tarefa descrita.

Assim, para verificar se as operações adicionadas estão corretas, foram feitos testes para cada instrução, e a análise de cada um dos testes foi feita com base nas imagens de ciclos geradas, mostrando a diferença do diagrama ao longo da execução. O tamanho de cada teste precisou ser reduzido, visto que caso o código de teste fosse muito grande, teriam diversas imagens de ciclos para serem explicadas, o que deixaria o trabalho com um número de páginas muito extenso. Para cada operação, o “Teste 1” será explicado mais detalhadamente, e o “Teste 2” será apenas para conferir o uso dela em outra situação, com outros valores, sua explicação será mais simples.

3.1. XORI

Testes relacionados a operação XORI:

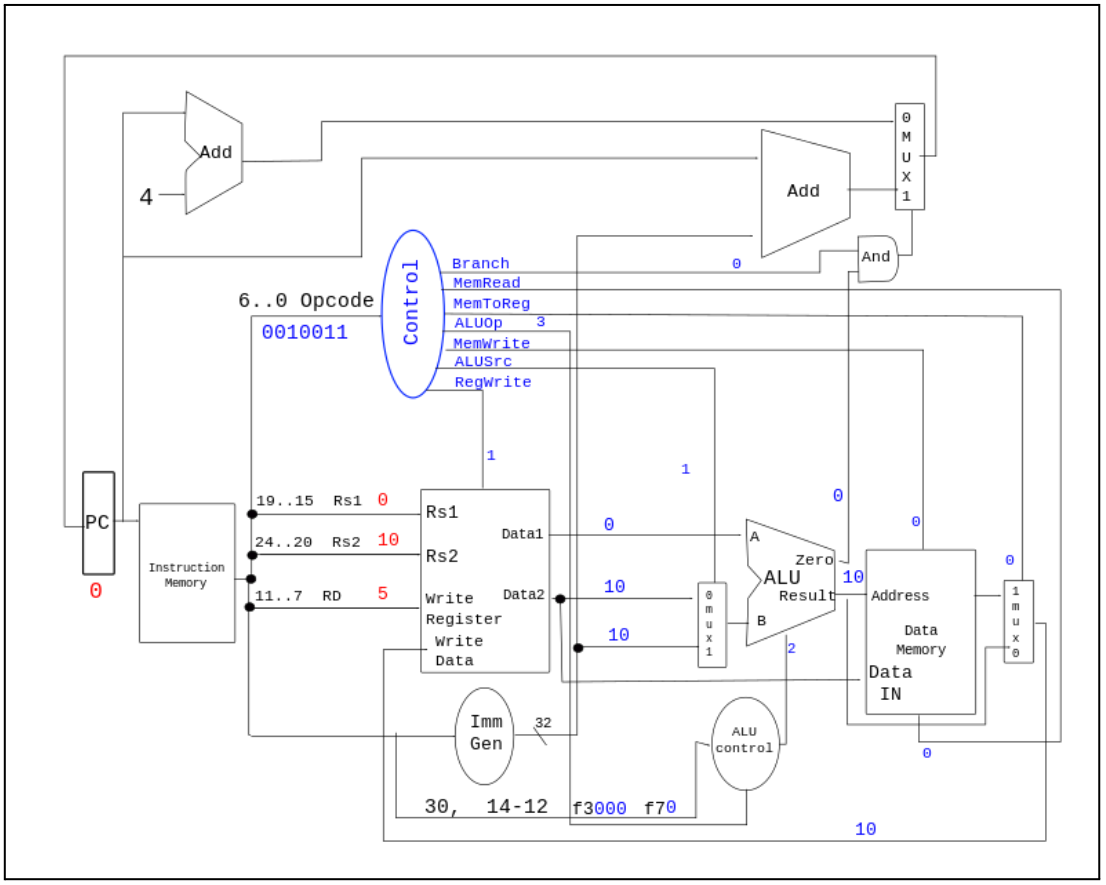
3.1.1. Teste 1:

Código referente ao teste:

```
li x5, 0b1010          # Carrega o valor 1010 (10 decimal) no registrador x5
xori x6, x5, 0b0011    # Realiza a operação AND entre x5 e 0011 (3 decimal)
                       # resultado em x6 = 1001 (9 decimal)
```

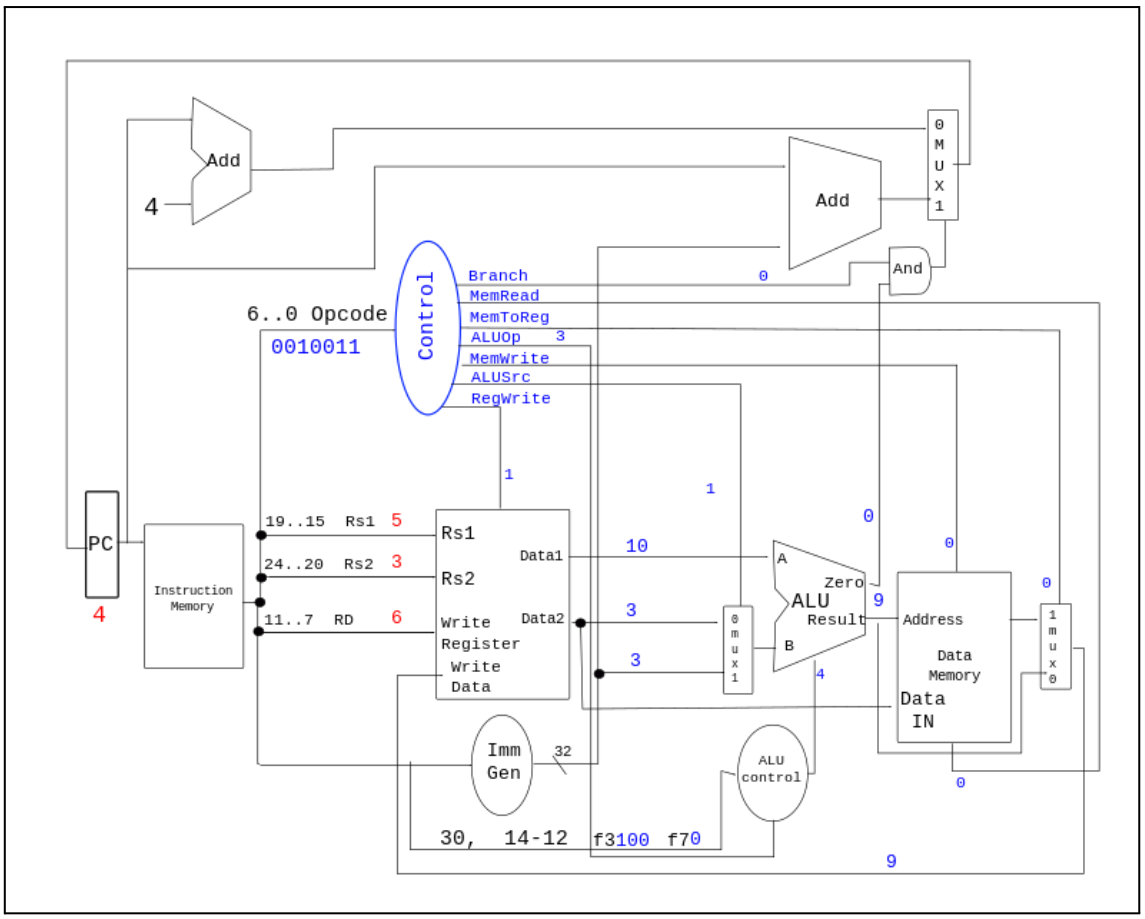
Código hexadecimal: 00a00293 0032c313

Imagem do ciclo 1:



No ciclo 1 é executado o comando “li x5, 0b1010”, uma pseudo-instrução que é traduzida para “addi x5, x0, 10”. Ocorre então a busca desta instrução (IF), PC é definido como “0x0000” e a instrução como “0x00a00293”. Depois disso a instrução é decodificada (ID), como é do Tipo-I seu “opcode” é “0010011”, o registrador “Rs1” é referente ao valor de X0, o “Rs2” ao valor imediato 10, e o “Rd” é referente ao X5; são definidos os sinais de controle “ALUSrc = 1” , “ALUOp = 11” (ADD), “RegWrite = 1”. Na parte da execução (EX), a ALU recebe “Rs1” (valor 0) e imediato (valor 10). Não há nenhuma operação de memória (MEM), o resultado 10 (0+10) da ALU é escrito em X5 (WB), e por fim, há a atualização do registro X5 = 10.

Imagem ciclo 2:



No ciclo 2 é executado o comando “xori x6, x5, 0b0011”. Ocorre a busca desta instrução (IF), PC é atualizado para “0x0004” e a instrução como “0x0032c313”. Depois disso, a instrução é decodificada (ID), como é do Tipo-I seu “opcode” é “0010011”, o registrador “Rs1” é referente ao valor de X5 e o “Rs2” ao valor imediato 3, são definidos os sinais de controle “ALUSrc = 1” , “ALUOp = 11” (ADD), “RegWrite = 1”. Na parte da execução (EX), a ALU recebe “Rs1” (valor 10) e imediato (valor 3). Não há nenhuma operação de memória (MEM), o resultado 9 (10 ANDI 3 = b1010 ANDI b0011 = b1001 = 9) da ALU é escrito em X6 (WB), e por fim, há a atualização do registro X6 = 9.

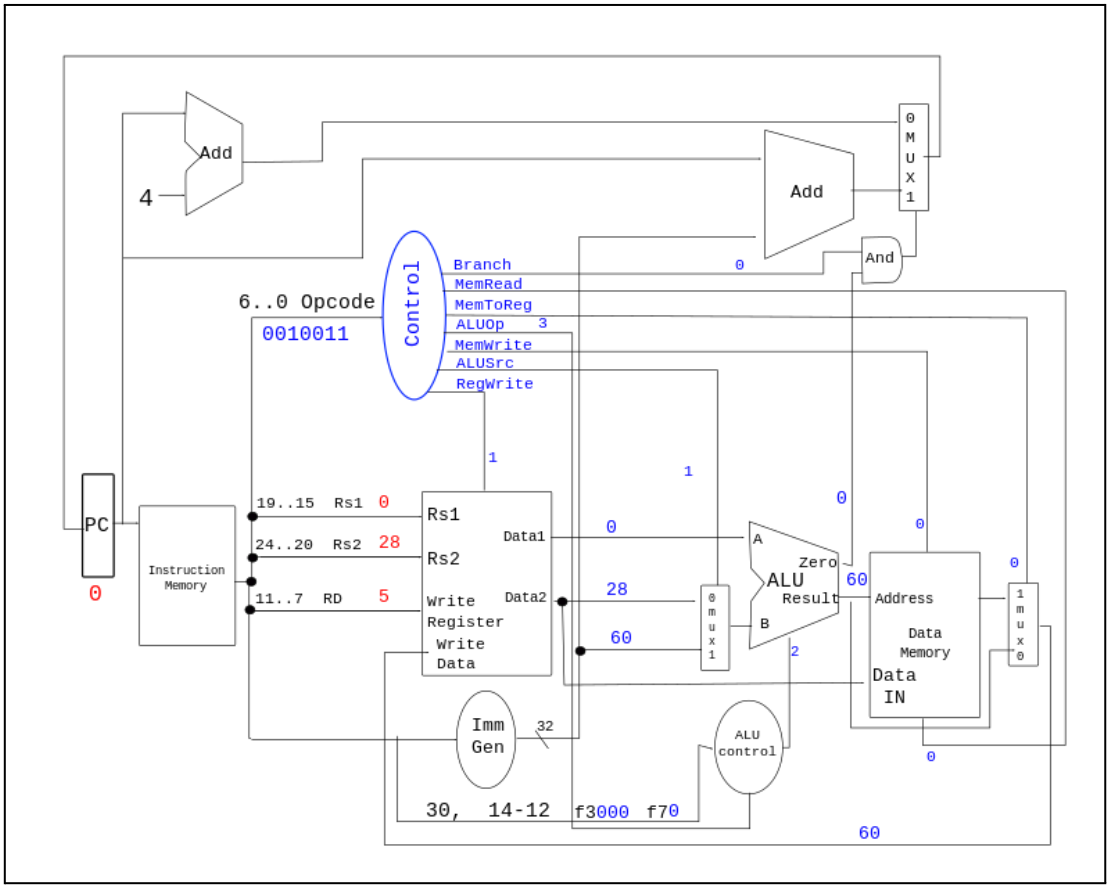
3.1.2. Teste 2:

Código referente ao teste:

```
li x5, 0x3C         # x5 = 0x3C (60 decimal, 00111100 binário)
xori x6, x5, 0x0F # x6 = x5 XOR 0x0F(15 decimal, 00001111 binário)
                  # x6 = 0x3C XOR 0x0F = 0x33 (51 em decimal)
```

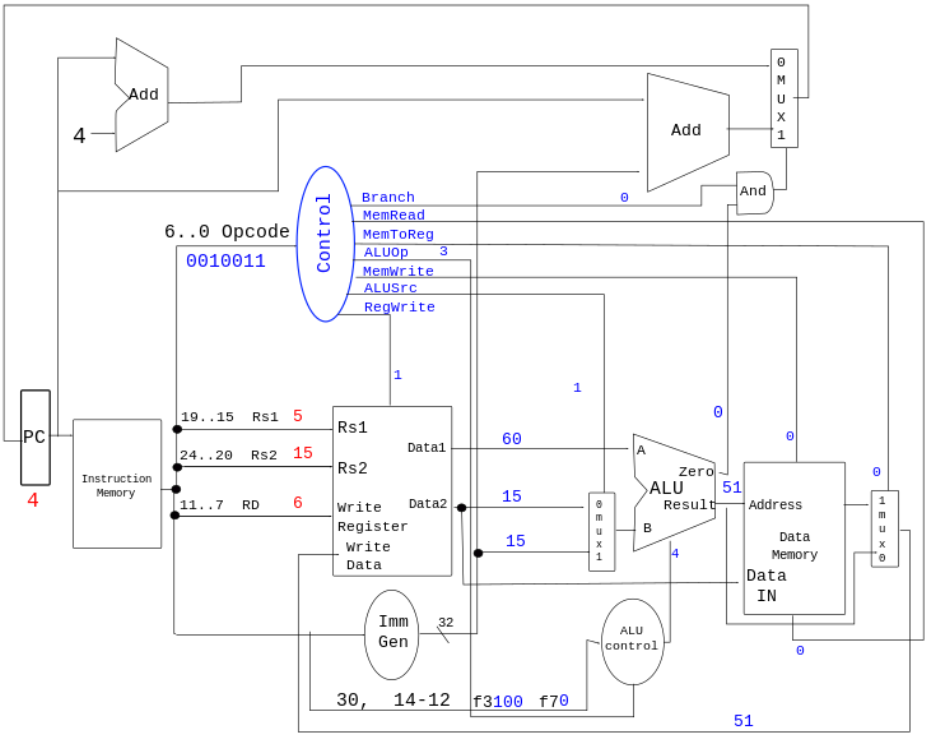
Código hexadecimal: 03c00293 00f2c313

Imagem ciclo 1:



No ciclo 1, é feito o comando “li x5, 0x3C”, (IF) PC é definido como “0x0000” e a instrução como “0x03c00293”. Como é do Tipo-I seu “opcode” é “0010011”, o registrador “Rs1” é referente a X0, “Rs2” valor imediato 60, são definidos sinais de controle “ALUSrc = 1” , “ALUOp = 11” (ADD), “RegWrite = 1”. Na execução (EX), ALU recebe “Rs1” (valor 0) e “Rs2 ”imediato (valor 60). Não há nenhuma operação de memória (MEM). Resultado 60 da ALU é escrito em X5 (WB).

Imagem ciclo 2:



No ciclo 2: Comando: “xori x6, x5, 0x0F”. Fetch (IF): PC = 0x0004; instrução como “0x00f2c313”. Decode (ID): Tipo-I (imediato); opcode = 0010011; registrador “Rs1” = X5, “Rs2” = 15, “Rd” = X6; sinais de controle: “ALUSrc = 1” , “ALUOp = 11” (ADD), “RegWrite = 1”. Execução (EX): ALU recebe “Rs1” (valor 60) e “Rs2 ”imediato (valor 15). MEM: Sem acesso de memória, Escrita de volta(WB): resultado da ALU (51) é escrito em X6, atualização do registro X6 = 51.

3.2. BEQ

Testes relacionados a operação BEQ:

3.2.1. Teste 1:

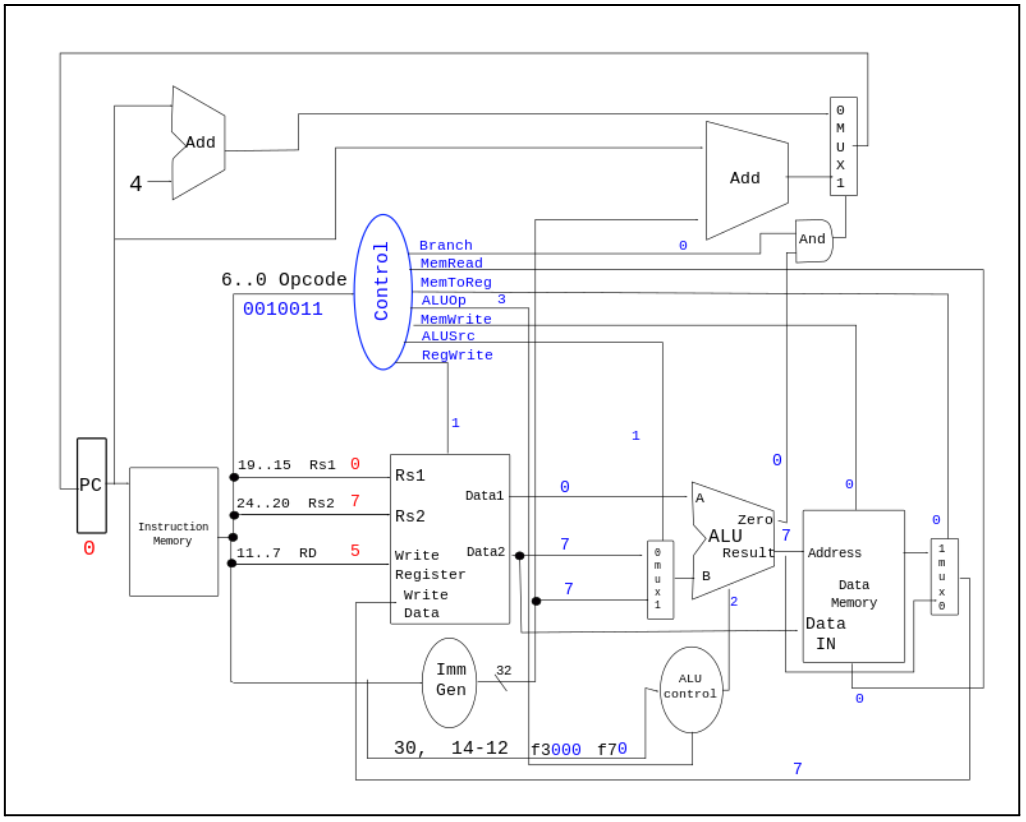
Código referente ao teste:

```
addi x5, x0, 7          # x5 = 7
addi x6, x0, 7          # x6 = 7

beq x5, x6, equal       # if(x5 == x6) vai para "equal"

addi x6, x0, 0          # se fosse executado, x6 = 0, porém não é executado
addi x5, x0, 0          # se fosse executado, x5 = 0, porém não é executado
```

```
equal: addi x5, x5, 4  # x5 = x5 + 4 = 7 + 4 = 11
                       # ou seja, no final temos x5 = 11 e x6 = 7

# Se não pular para "equal":
# x5 = 4
# x6 = 0
```

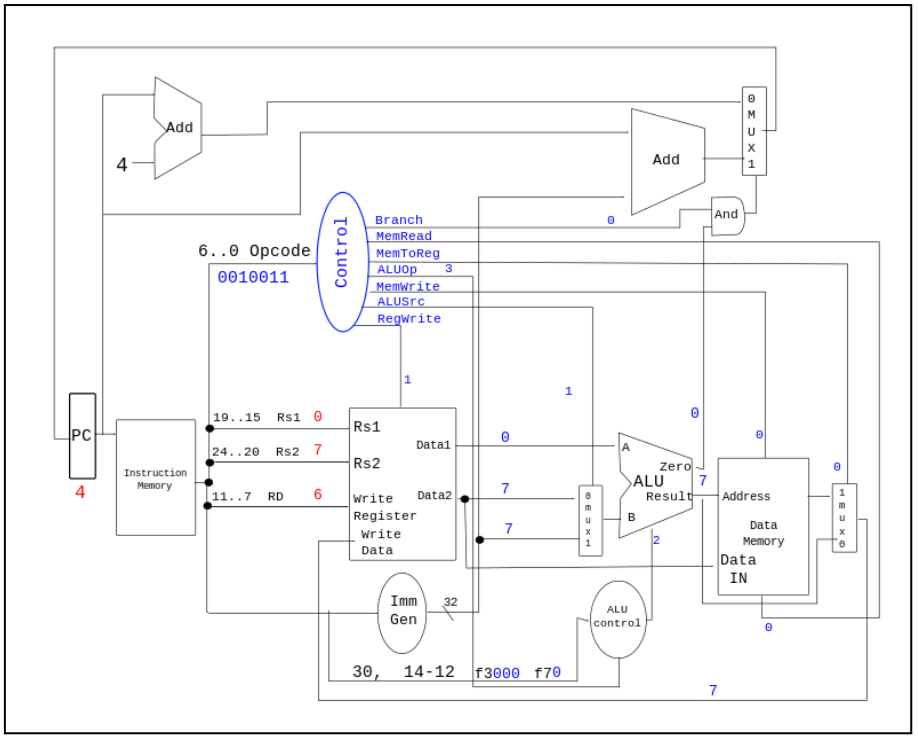
Código hexadecimal: 00700293 00700313 00628663 00000313 00000293 00428293

Imagem ciclo 1:



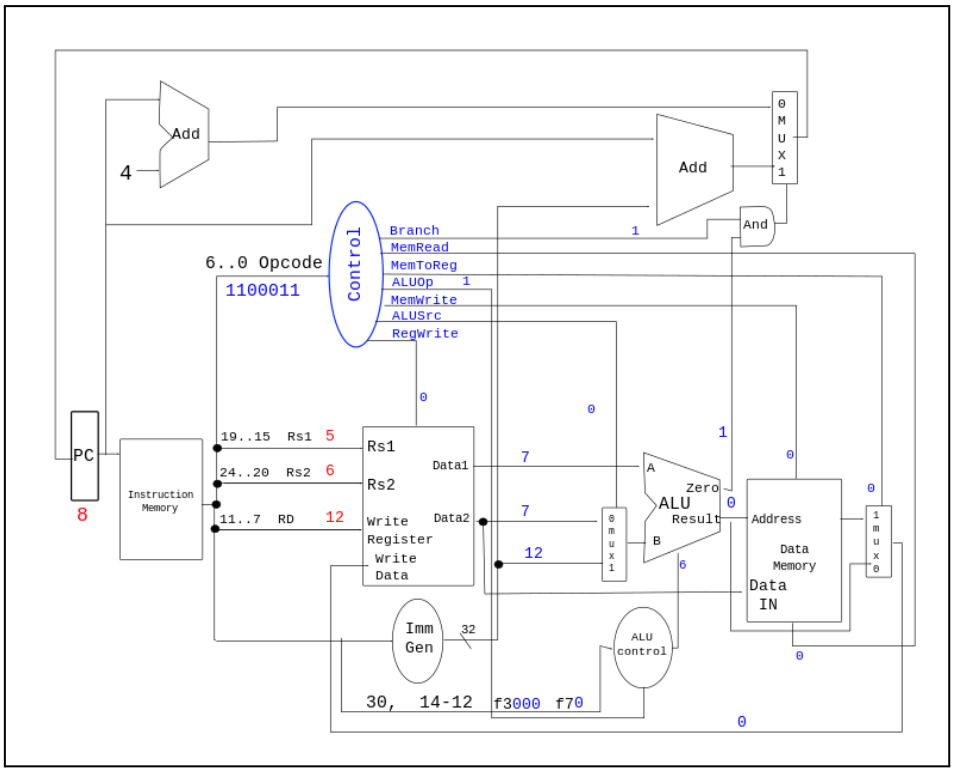
No ciclo 1, é feito o comando “addi x5, x0, 7”, (IF) PC é definido como “0x0000” e a instrução como “0x00700293”. Como é do Tipo-I seu “opcode” é “0010011”, o registrador “Rs1” é referente a X0, “Rs2” valor imediato 7, e “Rd” é o X5, são definidos sinais de controle “ALUSrc = 1” , “ALUOp = 11” (ADD), “RegWrite = 1”. Dessa forma, X5 = 7.

Imagem ciclo 2:



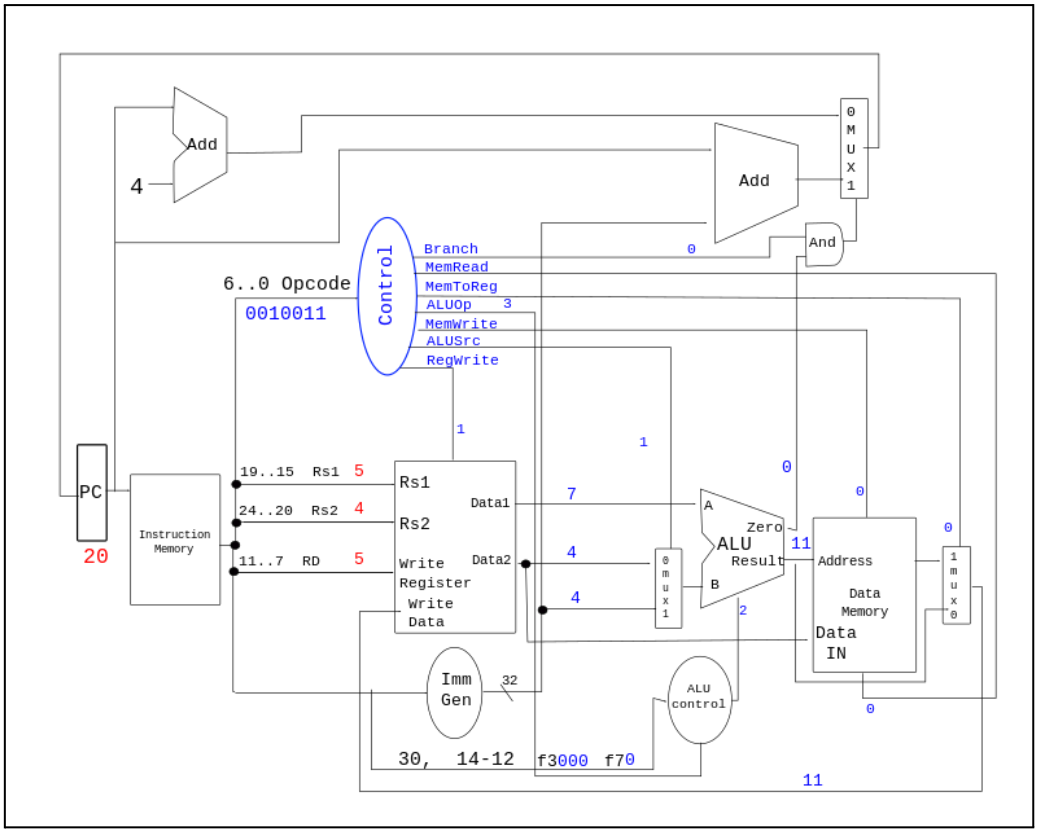
No ciclo 2, é feito o comando do ciclo, o valor de PC é somado em 4, o resultado é armazenado no registrador X6. Dessa forma, X6 = 7.

Imagem ciclo 3:



No ciclo 3, é feito o comando “beq x5, x6, equal”, (IF) PC é definido como “0x0008” e a instrução como “0x00628663”. Como é do tipo branch seu “opcode” é “1100011”, o registrador “Rs1” é referente a X5, “Rs2” valor imediato 6. Os valores dos registradores são comparados, por meio de uma subtração, para ver se são iguais. Como são iguais, então, o resultado da ALU é 0 e é feito o salto para o local de flag “equal”. Não há acesso à memória para BEQ, nem escrita de volta, pois não há escrita em registradores nesse comando.

Imagem ciclo 4:



No ciclo 4, é feito o comando “addi x5, x5, 4”, (IF) PC é definido como “0x0020” (já que foi feito o salto condicional) e a instrução como “0x00300293”. Como é do Tipo-I seu “opcode” é “0010011”, o registrador “Rs1” é referente a X5, “Rs2” valor imediato 4, são definidos sinais de controle “ALUSrc = 1” , “ALUOp = 11” (ADD), “RegWrite = 1”. Na execução (EX), ALU recebe “Rs1” (valor 7) e “Rs2 ”imediato (valor 4). Não há nenhuma operação de memória (MEM). Resultado 11 da ALU é escrito em X5 (WB).

3.2.2. Teste 2:

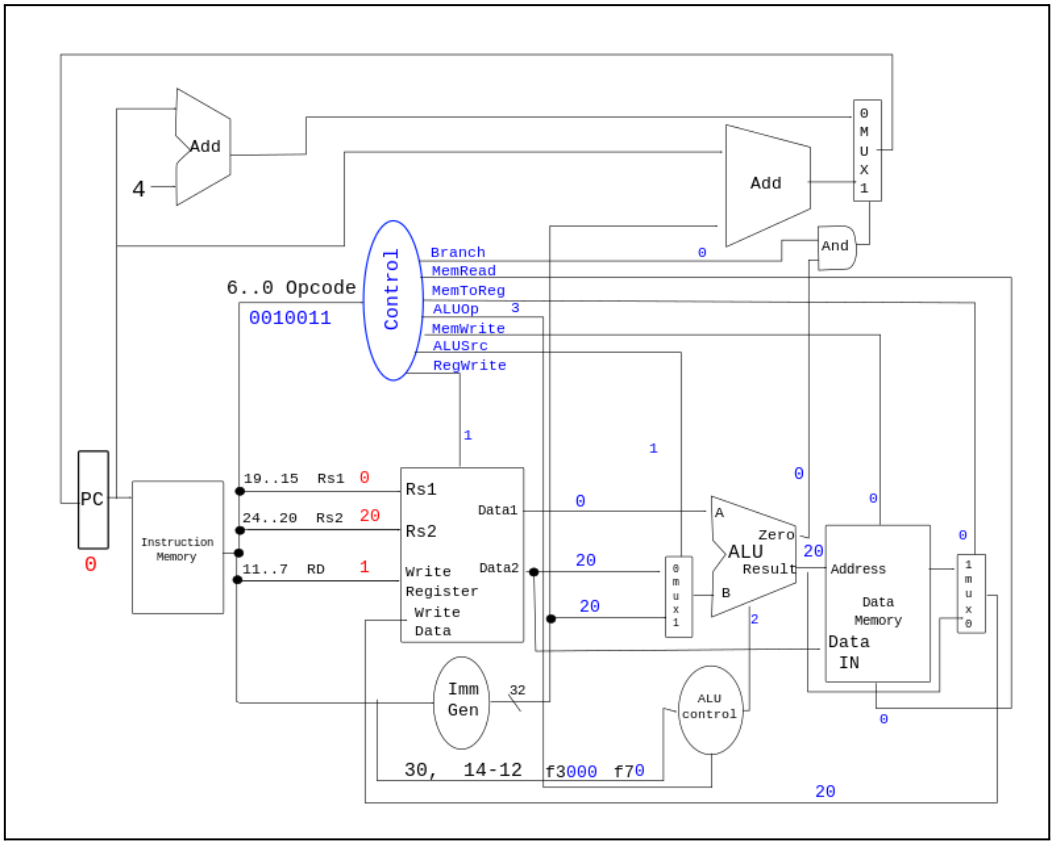
Código referente ao teste:

```
addi x1, x0, 20 # x1 = 20
addi x2, x0, 15 # x2 = 15

loop:
  beq x1, x2, fim  # if(x == y) => vá para "fim"
  addi x2, x2, 5
  beq x0, x0, loop # repetição do loop
fim:
  # x1 = 20, x2 = 20
```

Código hexadecimal: 01400093 00f00113 00208663 00510113 fe000ce3

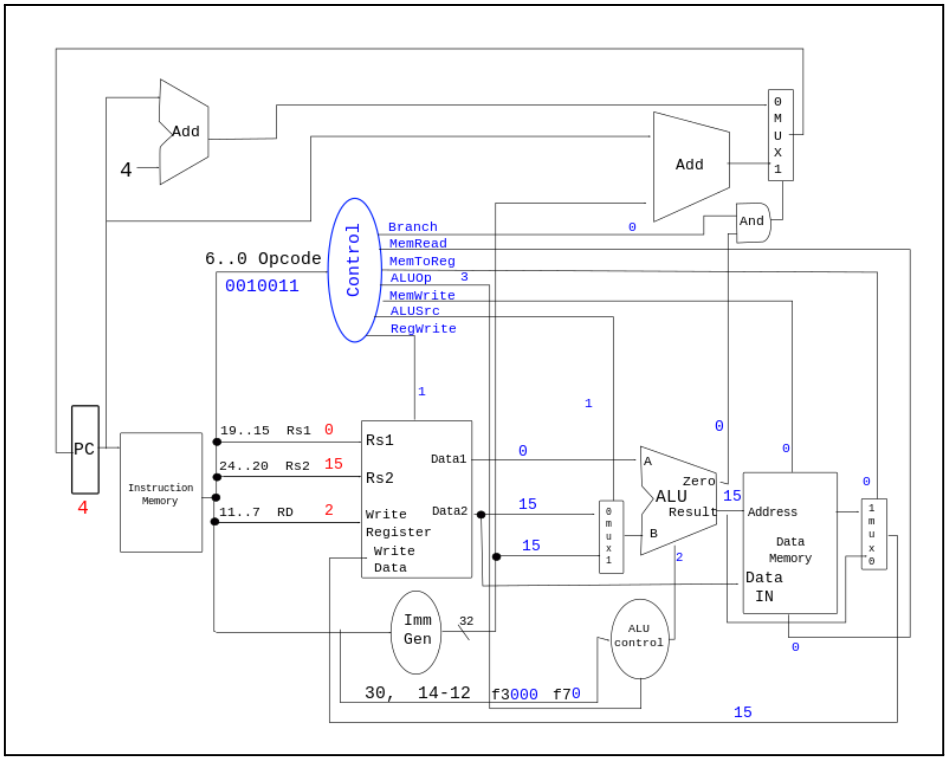
Como o código apresenta 6 ciclos no total, e vários deles são feitos apenas operações implementadas no código original, vamos focar mais nos ciclos em que é feita a nova operação implementada.

Imagem ciclo 1:



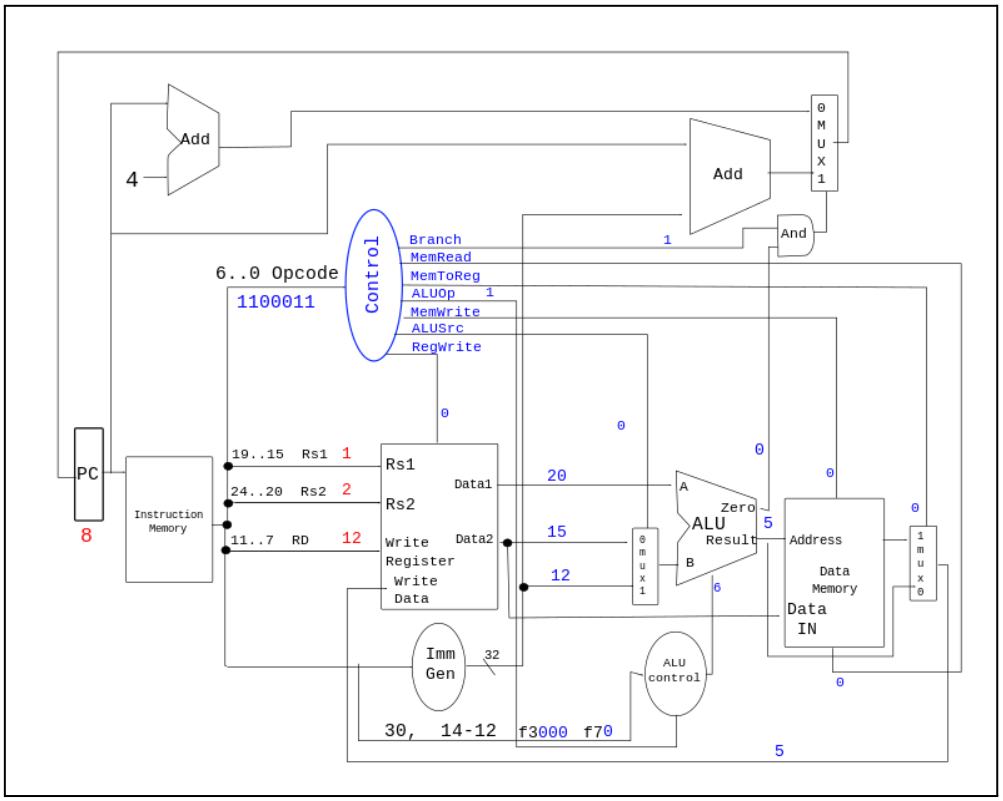
No ciclo 1, é feita a operação “addi x1, x0, 20”, ao final X1 = 20.

Imagem ciclo 2:



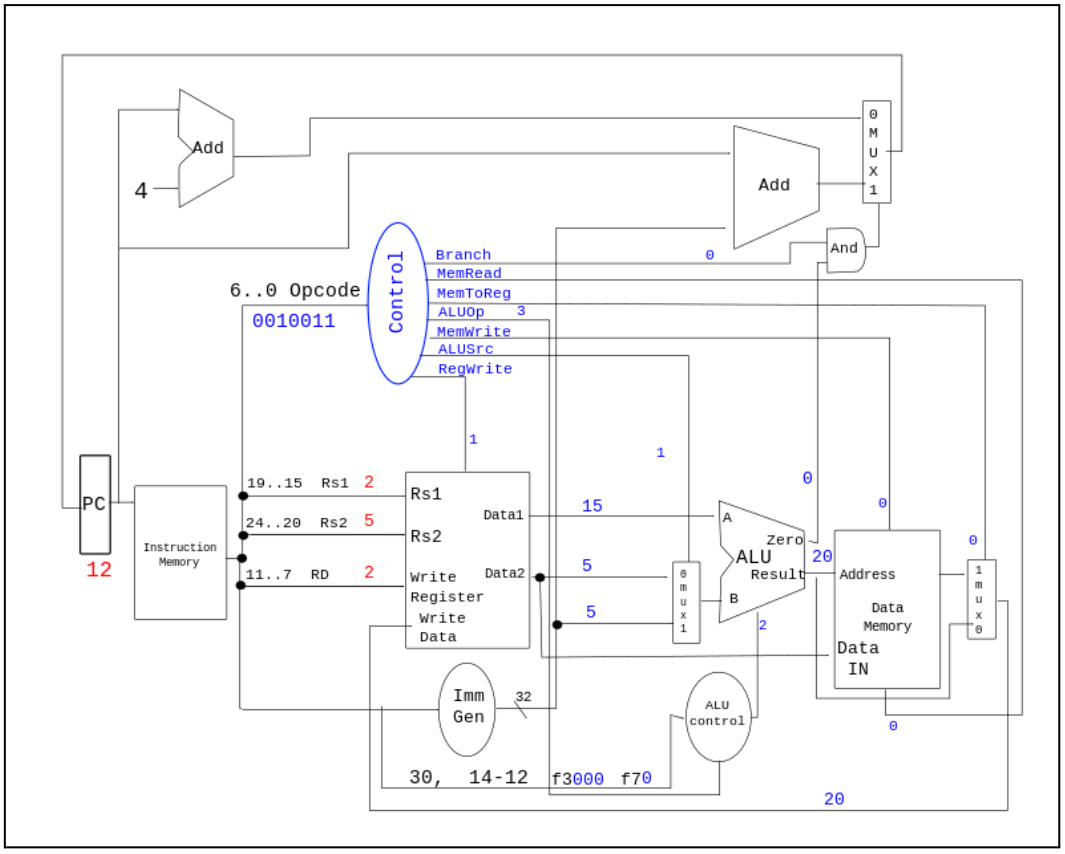
No ciclo 2, é feita a operação “addi x2, x0, 15”, ao final X2 = 15.

Imagem ciclo 3:



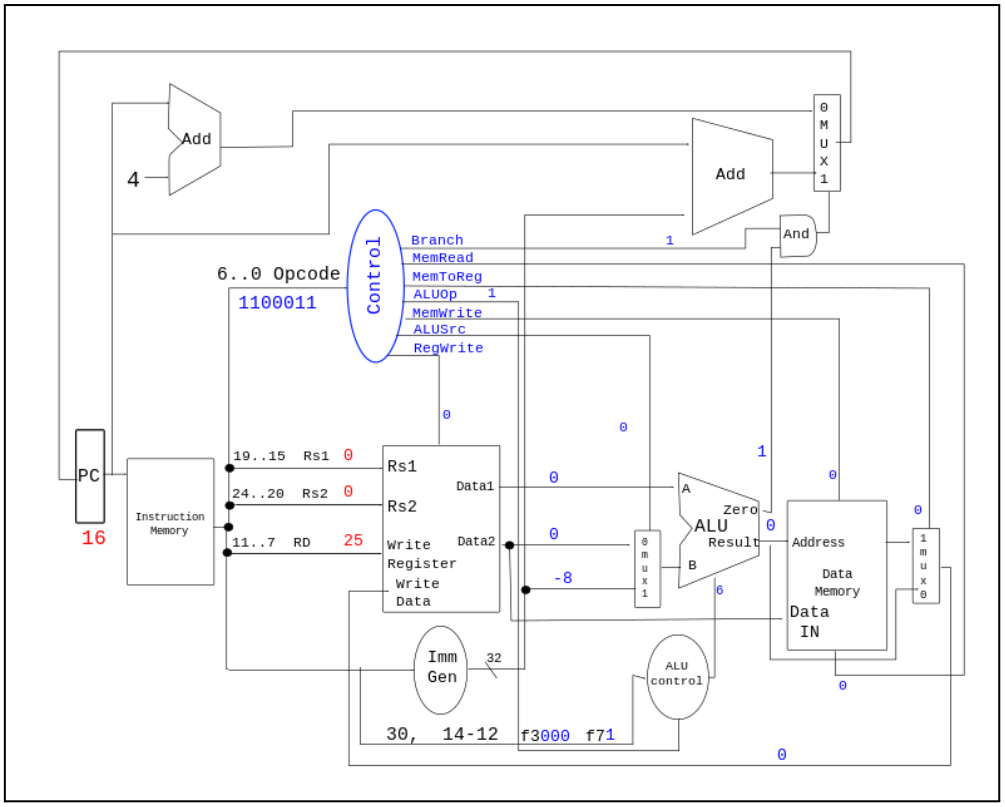
No ciclo 3: Comando: “beq x1, x2, fim”. Fetch (IF): PC = 0x0008; instrução como “0x00208663”. Decode (ID): tipo branch; opcode = 1100011; registrador “Rs1” = X1, “Rs2” = X2, offset = 12(endereço label “fim”); sinais de controle: “ALUSrc = 0” , “ALUOp = 01” (SUB), “RegWrite = 0”. Execução (EX): ALU recebe “Rs1” (valor 20) e “Rs2 ”imediato (valor 15), compara valores fazendo subtração deles, resultado diferente de zero (diferentes). MEM: Sem acesso de memória, Escrita de volta(WB): não há escrita em registradores nesse comando.

Imagem ciclo 4:



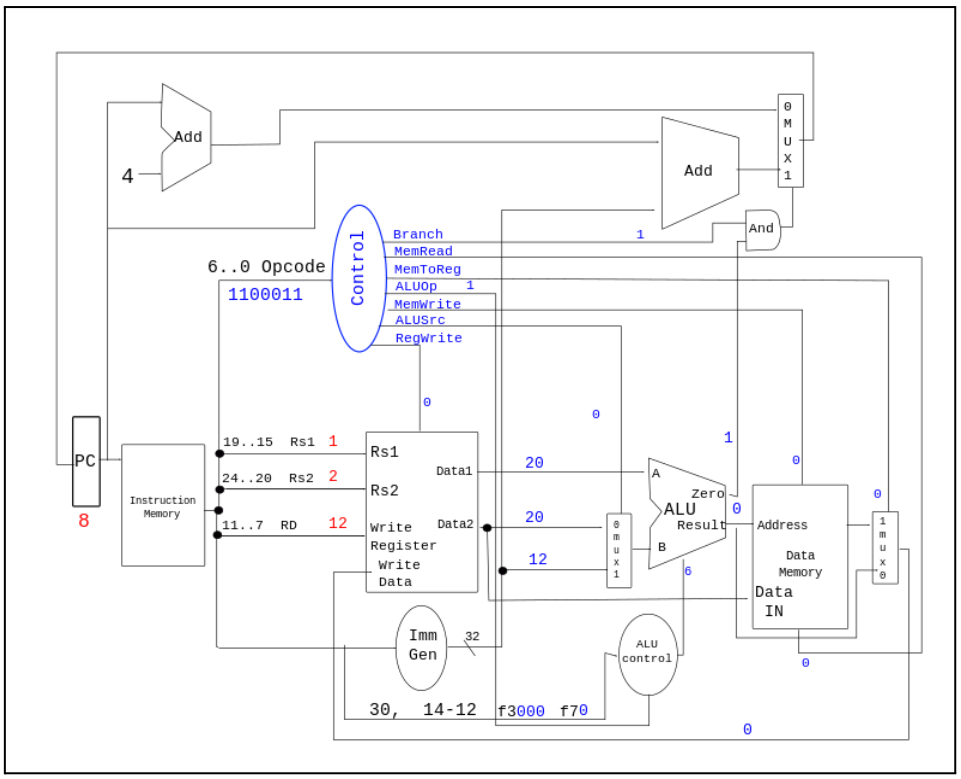
No ciclo 4, é feita a operação “addi x2, x2, 5”, ao final X2 = 20.

Imagem ciclo 5:



No ciclo 5: Comando: “beq x0, x0, loop”. Fetch (IF): PC = 0x0016; instrução como “0xfe000ce3”. Decode (ID): tipo branch; opcode = 1100011; registrador “Rs1” = X0, “Rs2” = X0, offset = -8 (endereço label “loop”); sinais de controle: “ALUSrc = 0” , “ALUOp = 01” (SUB), “RegWrite = 0”. Como os valores dos registradores são iguais, ocorre o salto, resultado ALU igual a 0.

Imagem ciclo 6:



No ciclo 6: Comando: “beq x0, x0, fim”. Fetch (IF): PC = 0x0008 (volta para o endereço “loop”); instrução como “00208663”. Decode (ID): tipo branch; opcode = 1100011; registrador “Rs1” = X1, “Rs2” = X2, offset = 12 (endereço label “fim”); sinais de controle: “ALUSrc = 0” , “ALUOp = 01” (SUB), “RegWrite = 0”. Como os valores dos são iguais, ocorre o salto para “fim”, resultado ALU igual a 0.

3.3. MUL

Testes relacionados a operação MUL:

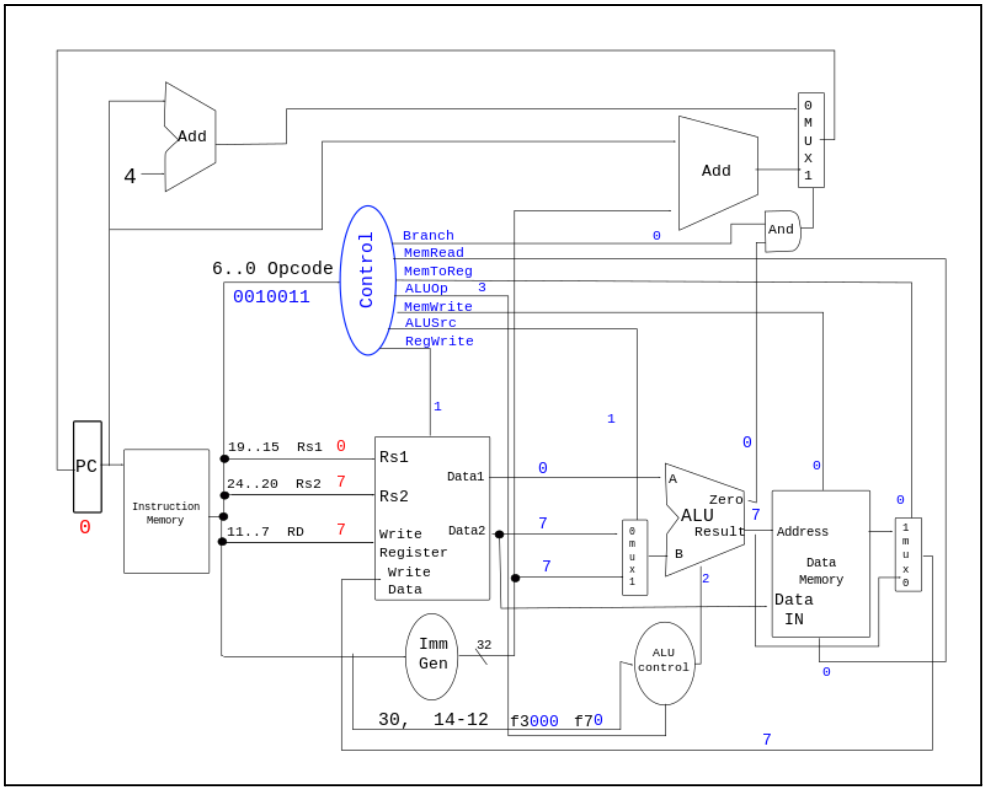
3.3.1. Teste 1:

Código referente ao teste:

```
addi x7, x0, 7   # Coloca valor 7 em x7
addi x2, x0, 2   # Coloca valor 2 em x2
mul x1, x7, x2   # Realiza multiplicação: 7 * 2 = 14;
                 # Resultado em x1
```

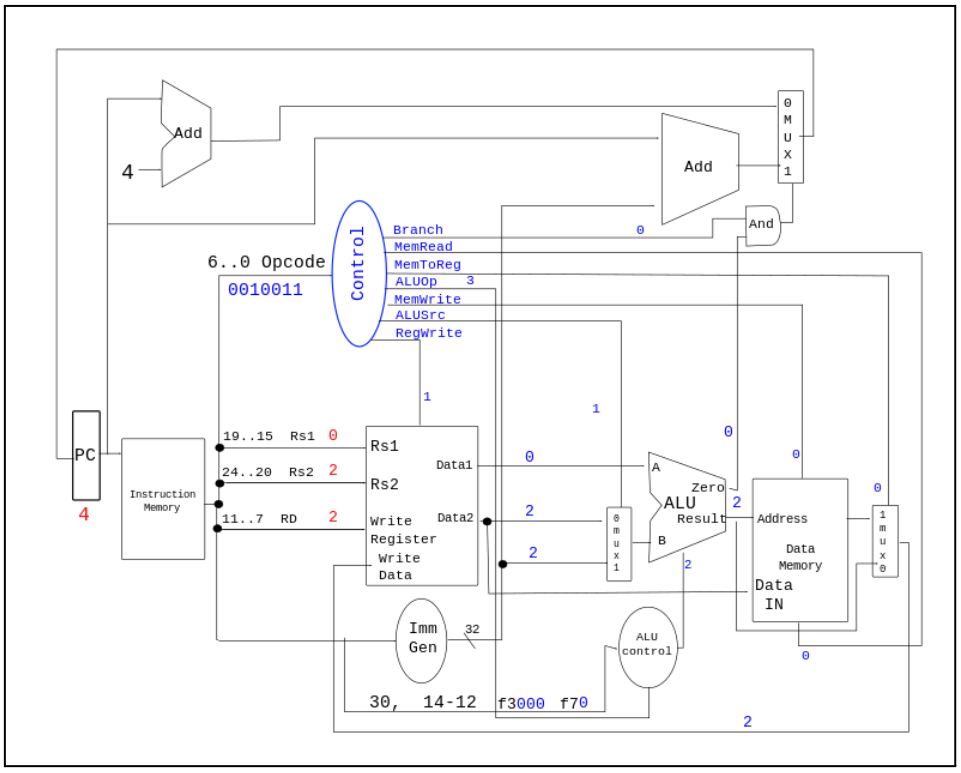
Código hexadecimal: 00700393 00200113 022380b3

Imagem ciclo 1:



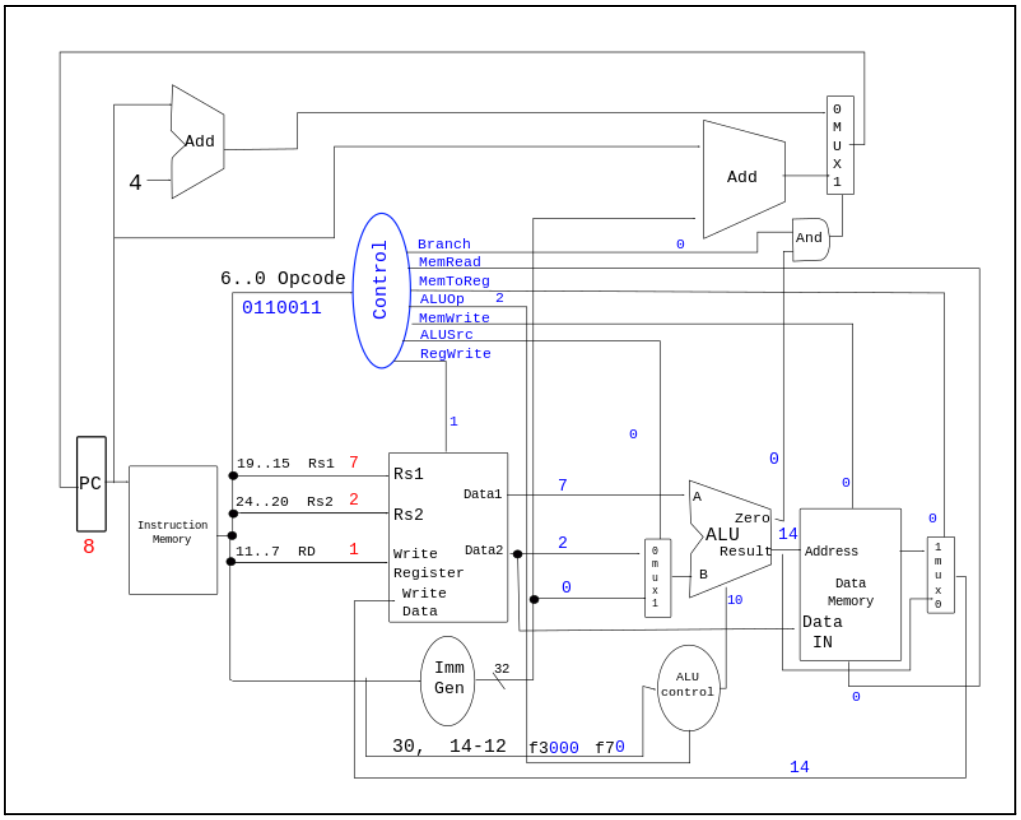
No ciclo 1, é feita a operação “addi x7, x0, 7”, ao final X7 = 7.

<u>Imagem ciclo 2:</u>



No ciclo 2, é feita a operação “addi x2, x0, 2”, ao final X2 = 2.

Imagem ciclo 3:



No ciclo 3, é feito o comando “mul x1, x7, x2”, (IF) PC é definido como “0x0008” e a instrução como “0x022380b3”. Como é do Tipo-R seu “opcode” é “0110011”, “funct7” é 0000001, “funct3” é 000, o registrador “Rs1” é referente a X7, “Rs2” referente a X2, “Rd” igual a X1, são definidos sinais de controle “ALUSrc = 0” , “ALUOp = 10” (MUL), “RegWrite = 1”. Na execução (EX), ALU recebe “Rs1” (valor 7) e “Rs2” (valor 2). Não há nenhuma operação de memória (MEM). Resultado 14 da ALU é escrito em X1 (WB).

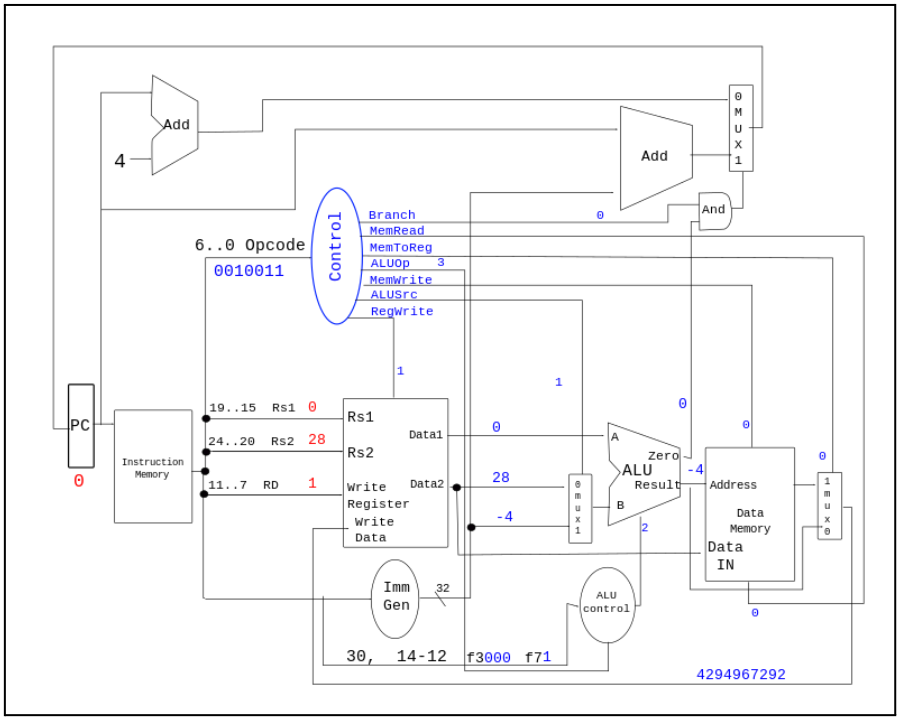
3.3.2. Teste 2:

Código referente ao teste:

```
addi x1, x0, -4   # x1 = -4
addi x2, x0, -2   # x2 = -2
addi x7, x0, -7   # x7 = -7

mul x7, x7, x1    # x7 = x7(-7) * x1(-4) = +28
mul x7, x7, x2    # x7 = x7(28) * x2(-2) = -56
```

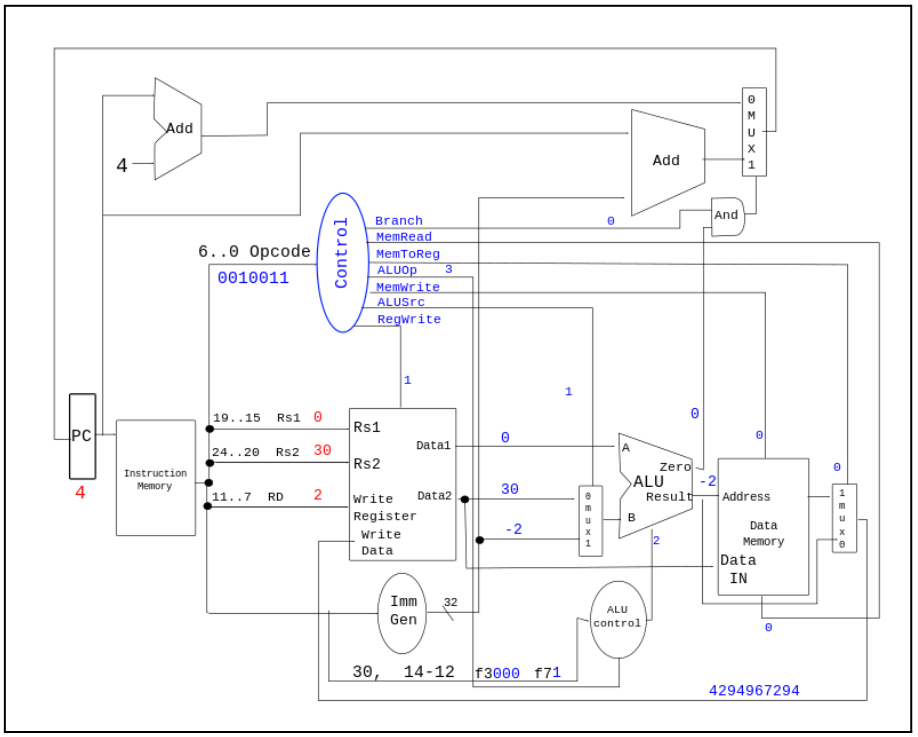
Código hexadecimal: ffc00093 ffe00113 ff900393 021383b3 022383b3

Imagem ciclo 1:



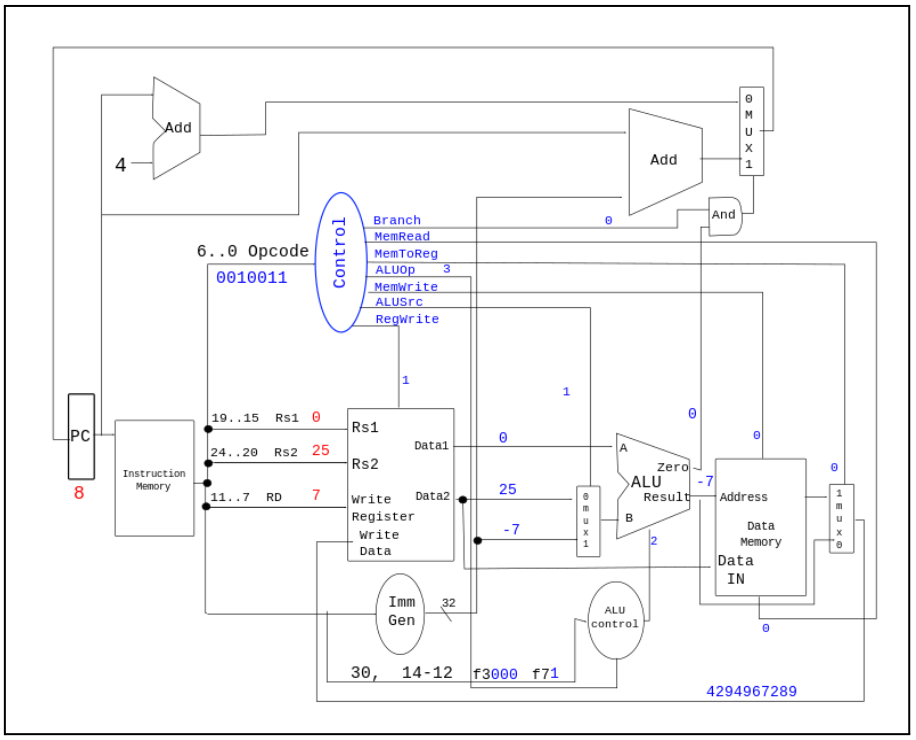
No ciclo 1, é feita a operação “addi x1, x0, -4”, ao final X1 = -4.

Imagem ciclo 2:



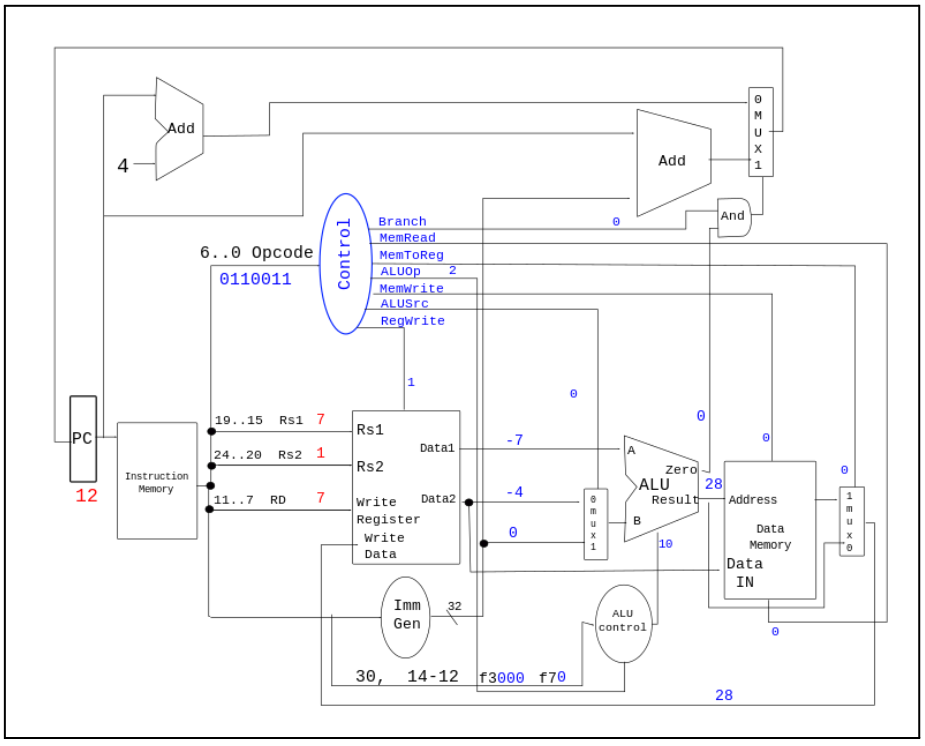
No ciclo 2, é feita a operação “addi x2, x0, -2”, ao final X2 = -2.

Imagem ciclo 3:



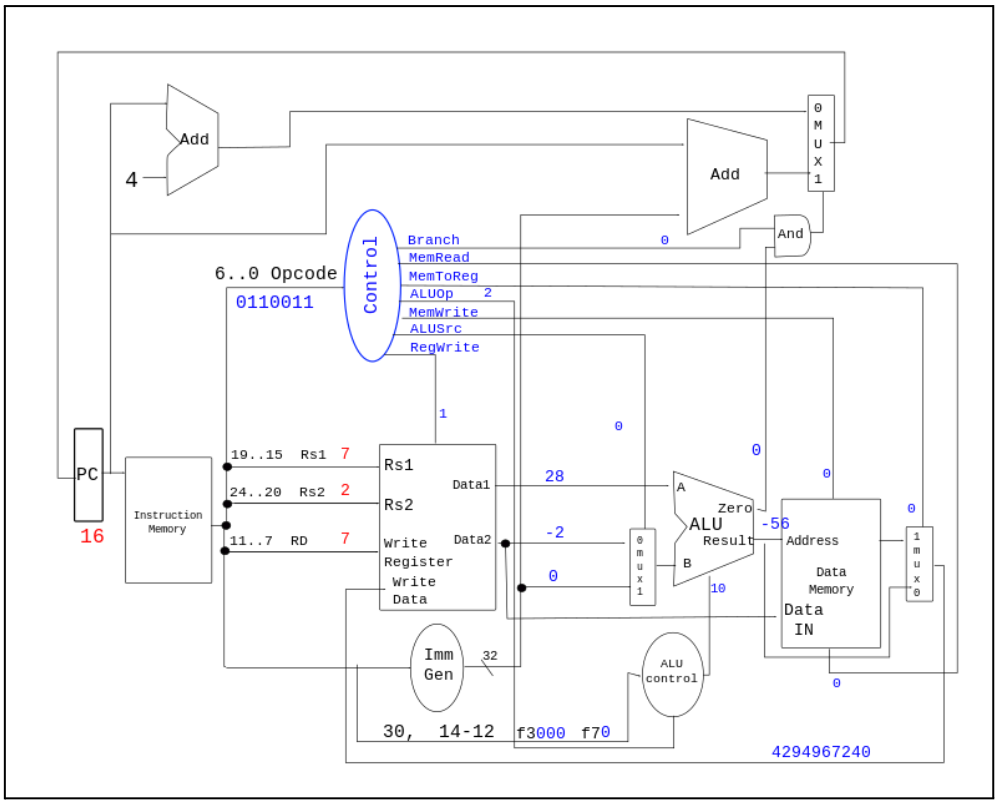
No ciclo 3, é feita a operação “addi x7, x0, -7”, ao final X7 = -7.

Imagem ciclo 4:



No ciclo 4, é feito o comando “mul x7, x7, x1”, (IF) PC é definido como “0x0008” e a instrução como “0x021383b3”. Como MUL é do Tipo-R seu “opcode” é “0110011”, “funct7” é 0000001, “funct3” é 000, o registrador “Rs1” é referente a X7, “Rs2” referente a X1, “Rd” igual a X7, são definidos sinais de controle “ALUSrc = 0” , “ALUOp = 10” (MUL), “RegWrite = 1”. Na execução (EX), ALU recebe “Rs1” (valor -7) e “Rs2” (valor -4). Não há nenhuma operação de memória (MEM). Resultado +28 da ALU é escrito em X7 (WB).

Imagem ciclo 5:



No ciclo 5, é feito o comando “mul x7, x7, x2”, (IF) PC é definido como “0x0008” e a instrução como “0x022383b3”. Como MUL é do Tipo-R seu “opcode” é “0110011”, “funct7” é 0000001, “funct3” é 000, o registrador “Rs1” é referente a X7, “Rs2” referente a X2, “Rd” igual a X7, são definidos sinais de controle “ALUSrc = 0” , “ALUOp = 10” (MUL), “RegWrite = 1”. Na execução (EX), ALU recebe “Rs1” (valor 28) e “Rs2” (valor -2). Não há nenhuma operação de memória (MEM). Resultado -56 da ALU é escrito em X7 (WB).

3.4. DIV

Testes relacionados a operação DIV:

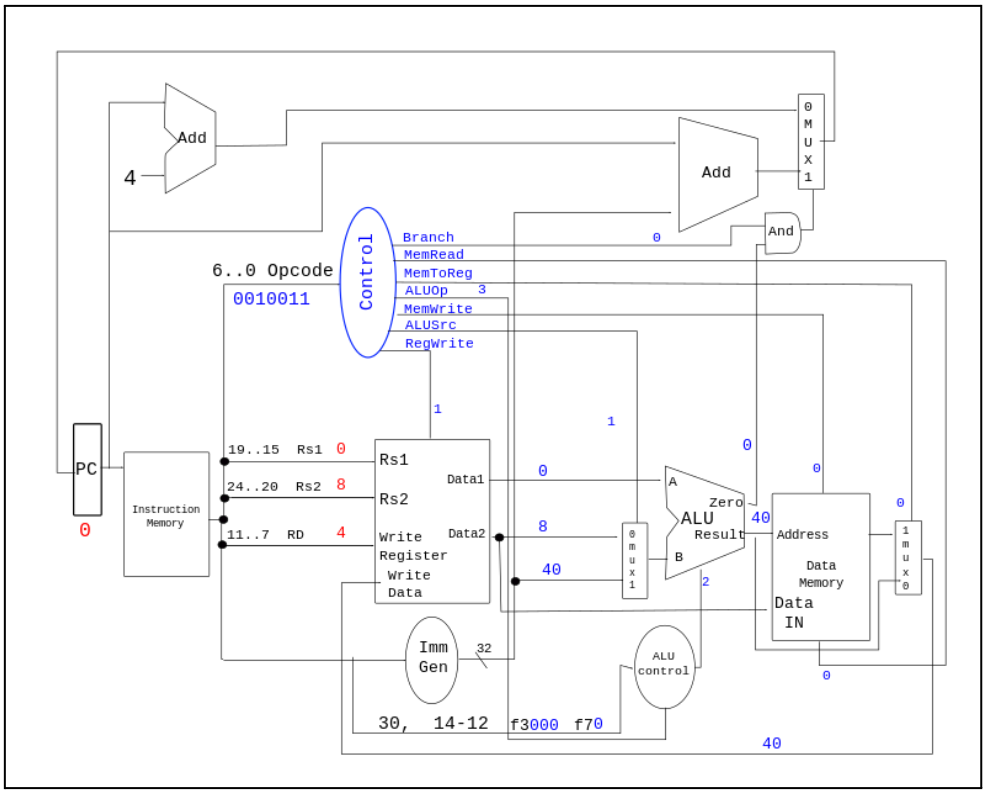
3.4.1. Teste 1:

Código referente ao teste:

```
addi x4, x0, 40   # x4 = 40
addi x2, x0, -2   # x2 = -2
div x7, x4, x2  # x7 = x4(40) / x2(-2) = -20
div x1, x7, x2  # x1 = x7(-20) / x2(-2) = 10
```

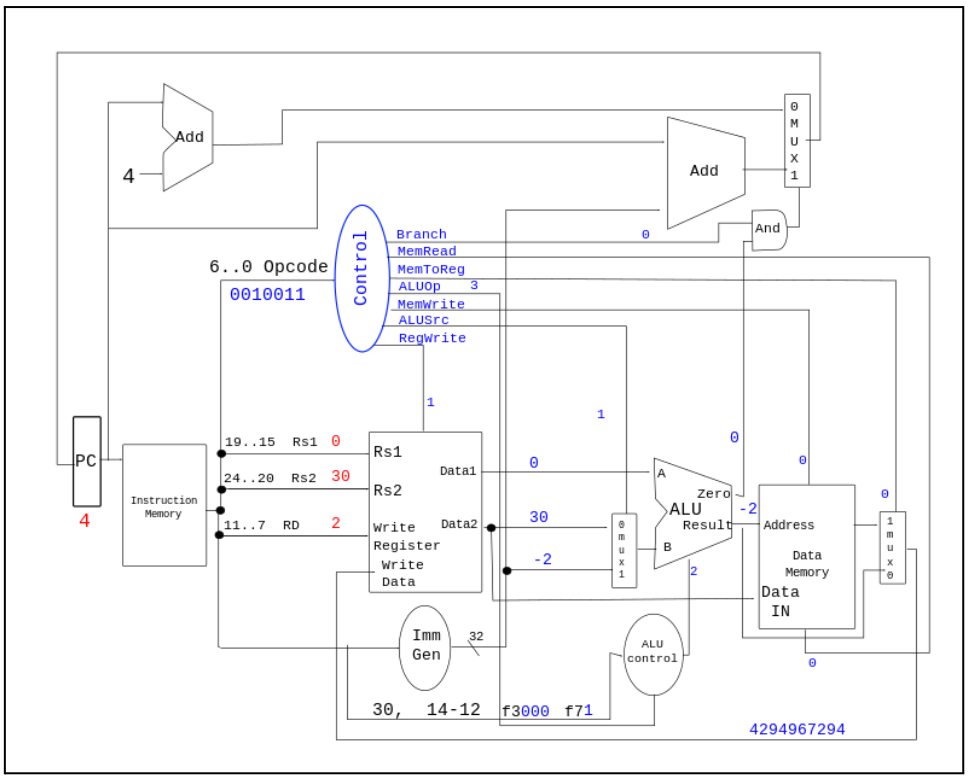
Código hexadecimal: 02800213 ffe00113 022243b3 0223c0b3

Imagem ciclo 1:



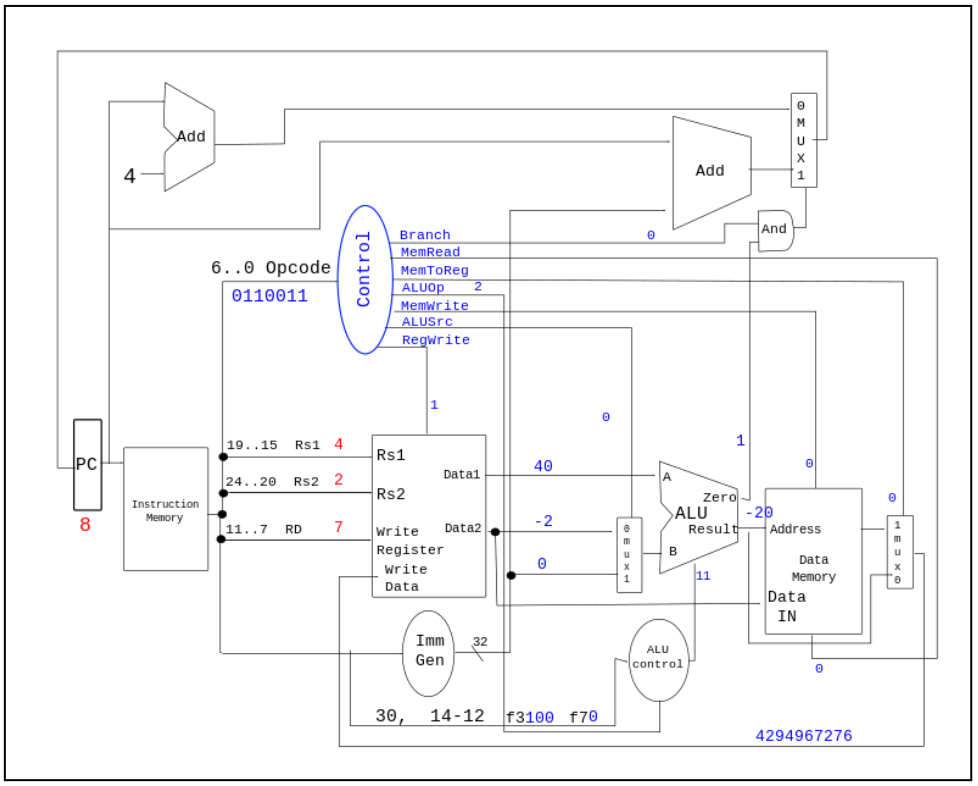
No ciclo 1, é feita a operação “addi x4, x0, 40”, ao final X4 = 40.

Imagem ciclo 2:



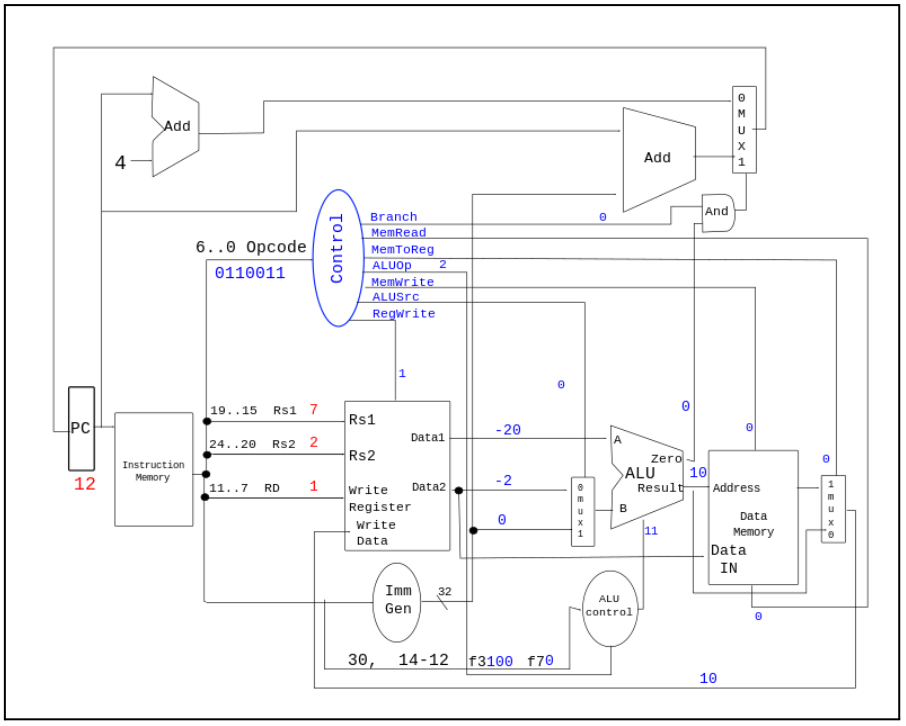
No ciclo 2, é feita a operação “addi x2, x0, -2”, ao final X2 = -2.

<u>Imagem ciclo 3:</u>



No ciclo 3, é feita a operação a “div x7, x4, x2”, (IF) PC é definido como “0x0008” e a instrução como “0x022243b3”. Como DIV é do Tipo-R seu “opcode” é “0110011”, “funct7” é 0000001, “funct3” é 100 (o que difere do MUL), o registrador “Rs1” é referente a X4, “Rs2” referente a X2, “Rd” igual a X7, são definidos sinais de controle “ALUSrc = 0” , “ALUOp = 10” (DIV), “RegWrite = 1”. Na execução (EX), ALU recebe “Rs1” (valor 40) e “Rs2” (valor -2). Não há nenhuma operação de memória (MEM). Resultado -20 da ALU é escrito em X7 (WB).

Imagem ciclo 4:



No ciclo 4, é feita a operação a “div x1, x7, x2”, (IF) PC é definido como “0x0012” e a instrução como “0x0223c0b3”. Como DIV é do Tipo-R seu “opcode” é “0110011”, “funct7” é 0000001, “funct3” é 100, o registrador “Rs1” é referente a X7, “Rs2” referente a X2, “Rd” igual a X1, são definidos sinais de controle “ALUSrc = 0” , “ALUOp = 10” (DIV), “RegWrite = 1”. Na execução (EX), ALU recebe “Rs1” (valor -20) e “Rs2” (valor -2). Não há nenhuma operação de memória (MEM). Resultado 10 da ALU é escrito em X1 (WB).

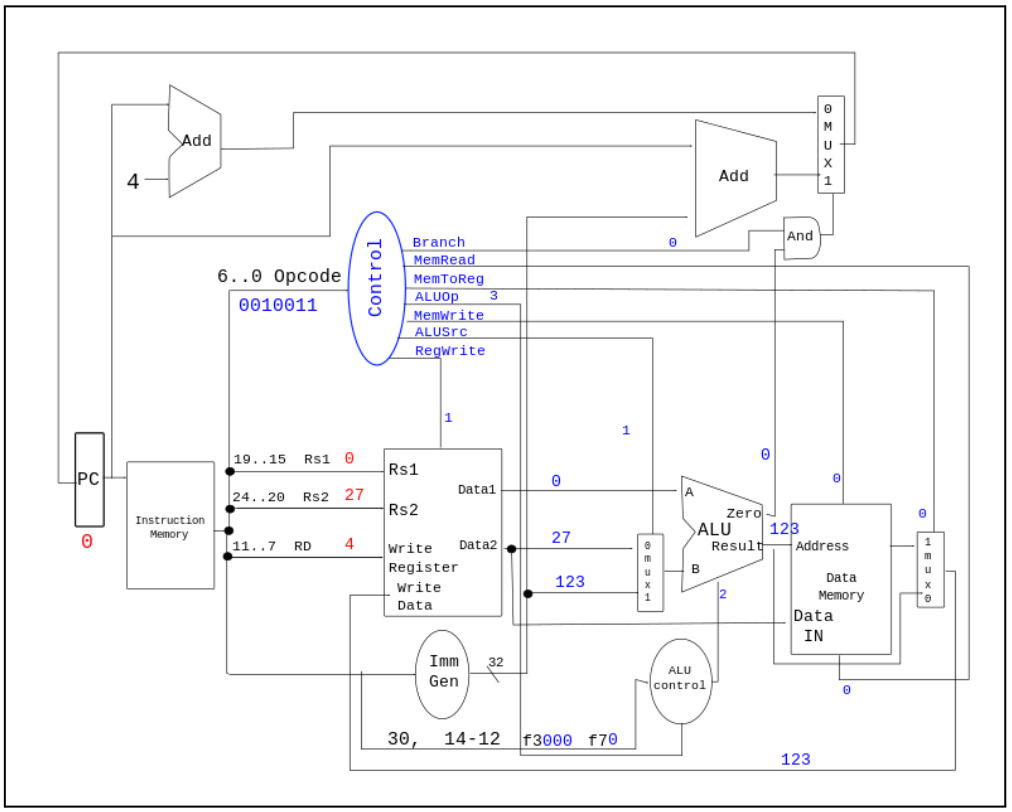
3.4.2. Teste 2:

Código referente ao teste:

```
addi x4, x0, 123    # x4 = 123
addi x2, x0, 20     # x2 = 20
div x7, x4, x2      # x7 = x4(123) / x2(20) = 6.15
div x1, x7, x0    # x1 = x7(6) / x0 = -1 (erro)
```

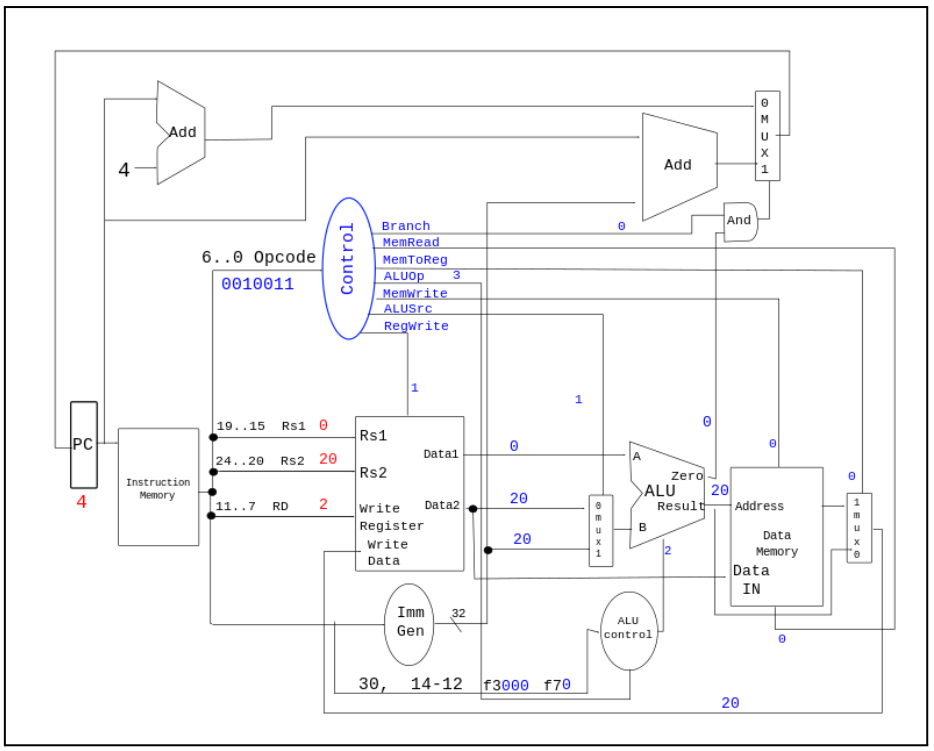
Código hexadecimal: 07b00213 01400113 022243b3 0203c0b3

Imagem ciclo 1:



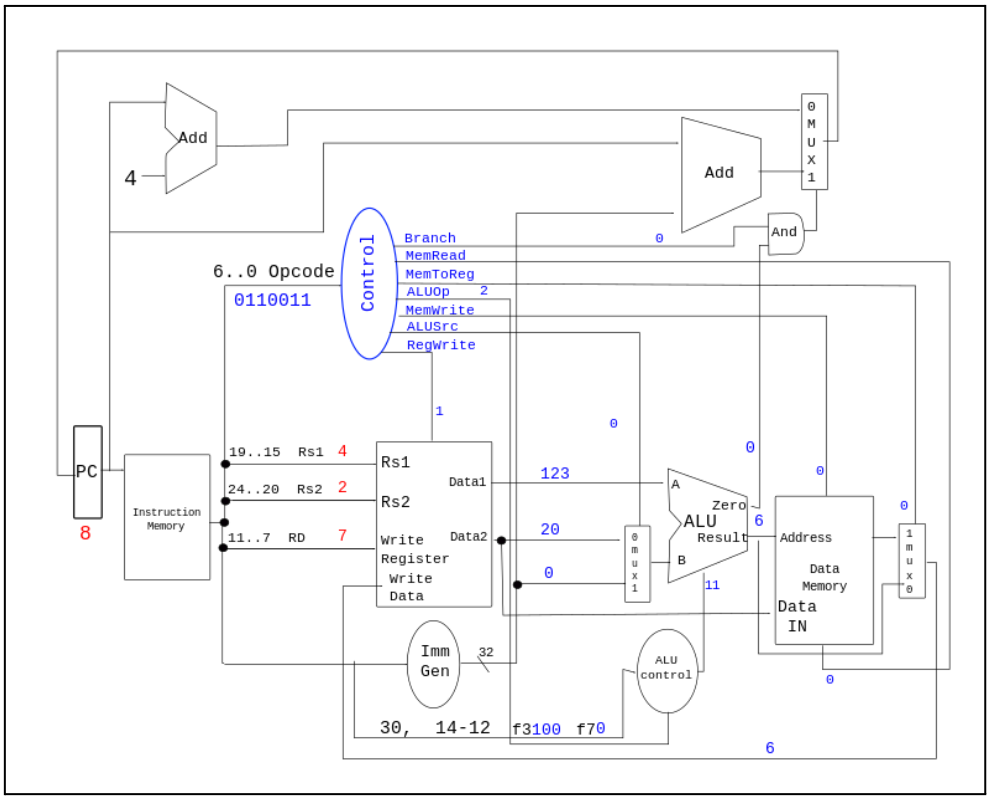
No ciclo 1, é feita a operação “addi x4, x0, 123”, ao final X4 = 123.

Imagem ciclo 2:



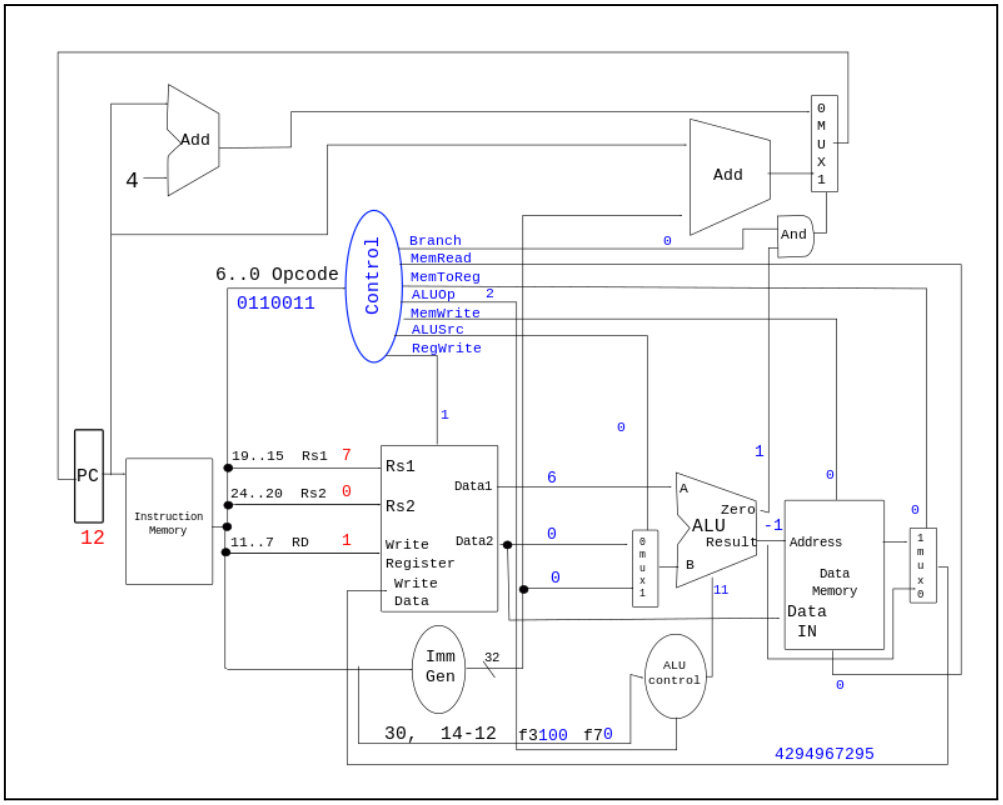
No ciclo 2, é feita a operação “addi x2, x0, 20”, ao final X2 = 20.

<u>Imagem ciclo 3:</u>



No ciclo 3, é feita a operação “div x7, x4, x2”, (IF) PC é definido como “0x0008” e a instrução como “0x022243b3”. Como DIV é do Tipo-R seu “opcode” é “0110011”, “funct7” é 0000001, “funct3” é 100, o registrador “Rs1” é referente a X4, “Rs2” referente a X2, “Rd” igual a X7, são definidos sinais de controle “ALUSrc = 0” , “ALUOp = 10” (DIV), “RegWrite = 1”. Na execução (EX), ALU recebe “Rs1” (valor 123) e “Rs2” (valor 20). Não há nenhuma operação de memória (MEM). O resultado da ALU é a parte inteira da divisão (valor 6), que é escrito em X7 (WB).

Imagem ciclo 4:



No ciclo 4, é feita a operação “div x1, x7, x0”, (IF) PC é definido como “0x0012” e a instrução como “0x0203c0b3”. Como DIV é do Tipo-R seu “opcode” é “0110011”, “funct7” é 0000001, “funct3” é 100, o registrador “Rs1” é referente a X7, “Rs2” referente a X0, “Rd” igual a X1, são definidos sinais de controle “ALUSrc = 0” , “ALUOp = 10” (DIV), “RegWrite = 1”. Na execução (EX), ALU recebe “Rs1” (valor 6) e “Rs2” (valor 0). Não há nenhuma operação de memória (MEM). O resultado da ALU é -1, mostrando o erro ao dividir um valor por 0, assim, X1 = -1(WB).

Considerações finais

O desenvolvimento deste projeto proporcionou uma forma de aprofundar nosso conhecimento em Verilog e como é feito o design de um processador RISC-V. O trabalho permitiu a visualização prática dos conceitos ensinados em Organização de Computadores 1, ao implementar instruções adicionais no processador de cinco estágios foi possível entender melhor a sua arquitetura.

O objetivo principal de adaptar o caminho de dados para incluir as instruções “XORI”, “BEQ”, “MUL” e “DIV” foi alcançado, assim como a explicação de cada modificação no código que permitiu sua execução. Os testes de cada instrução confirmaram o funcionamento correto dessas operações novas por meio da explicação das imagens demonstradas.

Referências

Omar Paranaiba Vilela Neto (2024). Slides virtuais da disciplina de Organização de Computadores 1. Disponibilizado via Moodle. Departamento de Ciência da Computação. Universidade Federal de Minas Gerais. Belo Horizonte.

RISC-V Instruction Set Specifications. Disponível em: RISC-V Instruction Set Specifications — riscv-isa-pages documentation (msyksphinz-self.github.io). Acesso em: dezembro de 2024.

PATTERSON, D. A.; HENNESSY, J. L. *Organização e Projeto de Computadores: Interface Hardware/Software*. Tradução da 5ª edição. Elsevier Editora Ltda, 2017.