Organização de Computadores B (2022/2) - Turma B

Primeiro Trabalho Prático - implementação de instruções em processador MIPS

Grupo 6
Arthur Hendges da Silva - 00332968
Eduardo Luís Marques - 00323594
Eduardo Raupp Peretto - 00313439
Hiram Artnak Martins - 00276484

Instruções Implementadas

XORI - XOR Imediato

Tipo I

XORI 001110	rs	rt	immediate
bits: 6	5	5	16

A instrução de XORI, em um processador MIPS, realiza a operação de um XOR (“ou” exclusivo) entre os bits contidos no registrador rs e o valor binário constante no campo “immediate” (estendido com sinal para 32 bits), salvando o resultado no registrador rt.

JALR - Jump And Link Register

Tipo R

SPECIAL 000000	rs	0 00000	rd	0 00000	JALR 001001
bits: 6	5	5	5	5	6

O programa realiza um jump para o endereço contido no registrador rs e salva o PC + 4 no registrador rd.

BGEZ - Branch on Greater than or Equal to Zero

Tipo I

REGIMM 000001	rs	BGEZ 00001	offset
bits: 6	5	5	16

A instrução BGEZ compara o valor contido no registrador recebido em rs com o valor zero. Se for igual ou maior do que zero, então toma-se o branch determinado pelo offset da instrução.

DIV - Signed Division

Tipo R

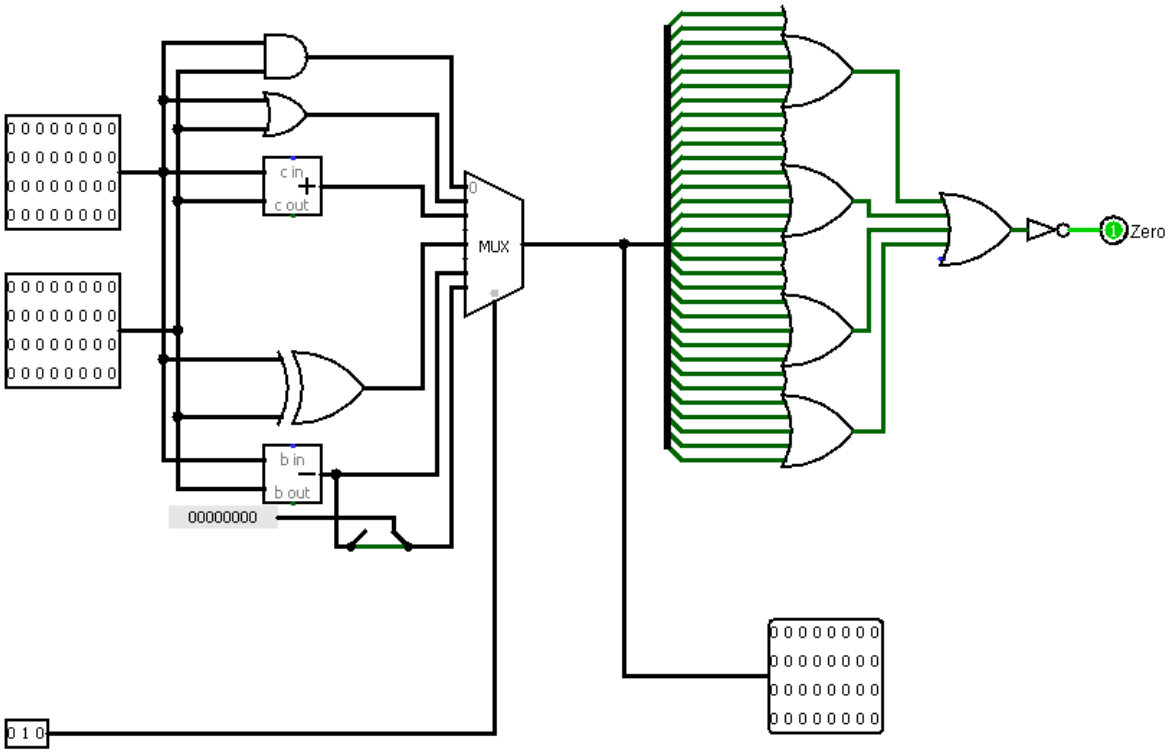
SPECIAL 000000	rs	rt	0000000000	DIV 011010
bits: 6	5	5	10	6

A instrução DIV divide o valor contido no registrador **rs** pelo valor contido no registrador **rt**. O quociente é guardado em um registrador especial chamado LO e o resto é guardado em um registrador especial chamado HI. A instrução DIV espera receber números com sinal. Essa instrução não gera exceções aritméticas.

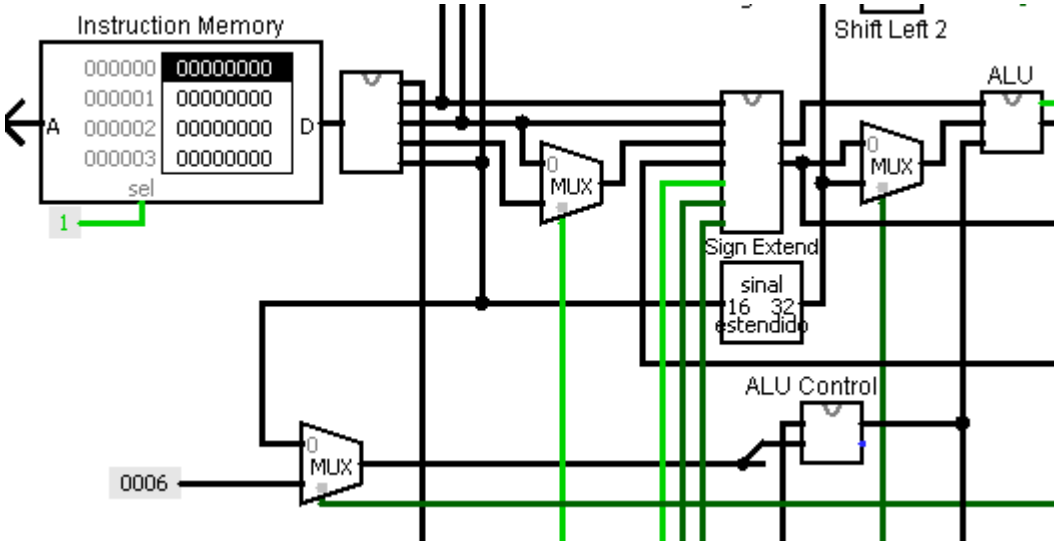
MIPS Monociclo

XORI

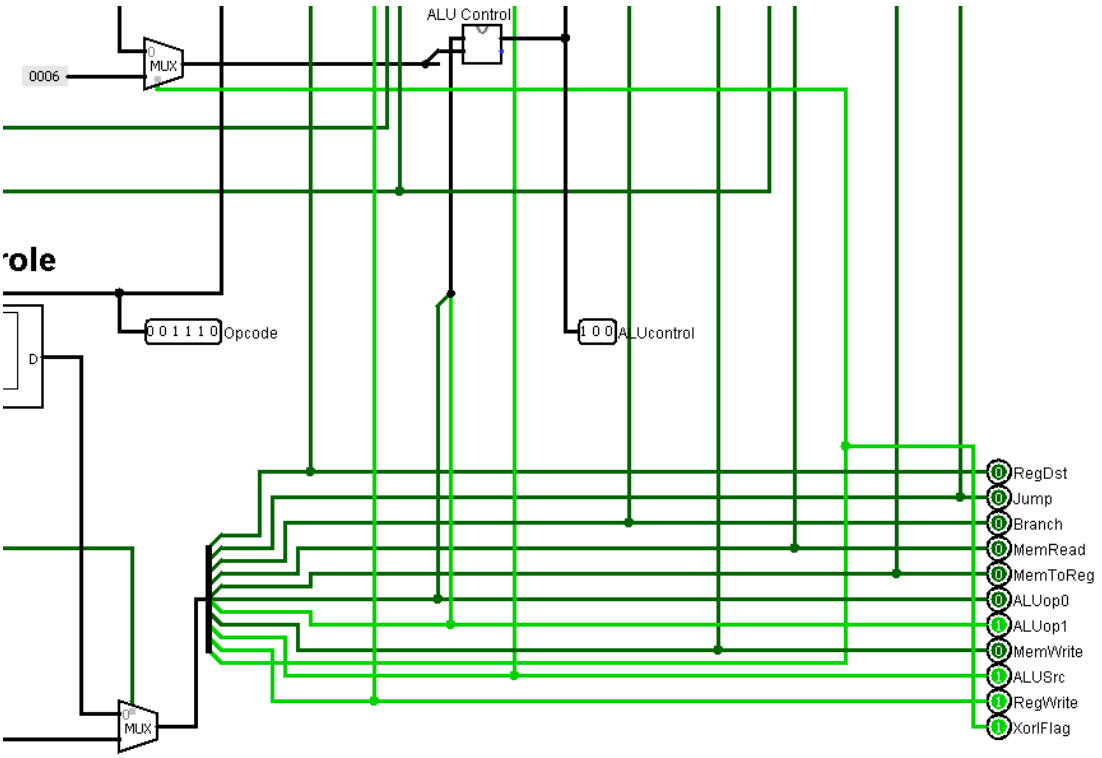
Para implementação da instrução XORI no MIPS Monociclo, a primeira necessidade foi adicionar uma porta lógica XOR dentro da ALU do circuito disponibilizado. Suas entradas são cada um dos operandos da ALU e sua saída é conectada na quarta entrada do Multiplexador de Seleção, tornando o seguinte circuito:



Para selecionarmos corretamente a operação na ALU quando quisermos realizar a instrução de XORI, foi adicionado um multiplexador no circuito principal com sua saída conectada à entrada de ALU Control e contendo duas entradas: o campo funct de instruções do tipo R (na entrada 0) e uma constante de 16 bits de valor 6, em decimal (na entrada 1):



O bit de seleção desse mux é o bit de controle “XorIFlag”, que quando ativado seleciona a entrada 1 para ser encaminhada à entrada de ALU Control, gerando os bits 100 (4, em decimal) de seleção na ALU e, portanto, selecionando a operação XOR:

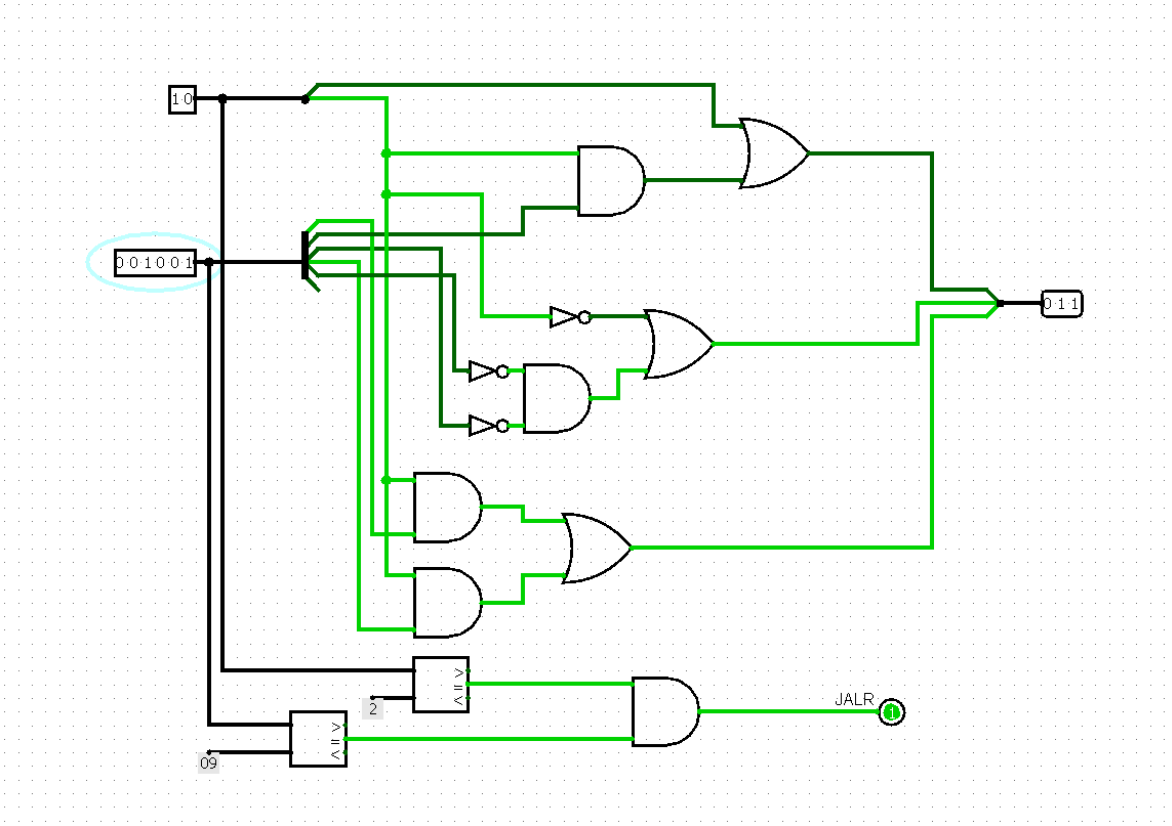


Esse novo bit de controle (XorIFlag), necessário para nossa implementação da instrução XORI, foi implementado no 11º bit do Bloco de Controle. Dessa forma, quando ele estiver ligado, o bit de XorIFlag também estará.

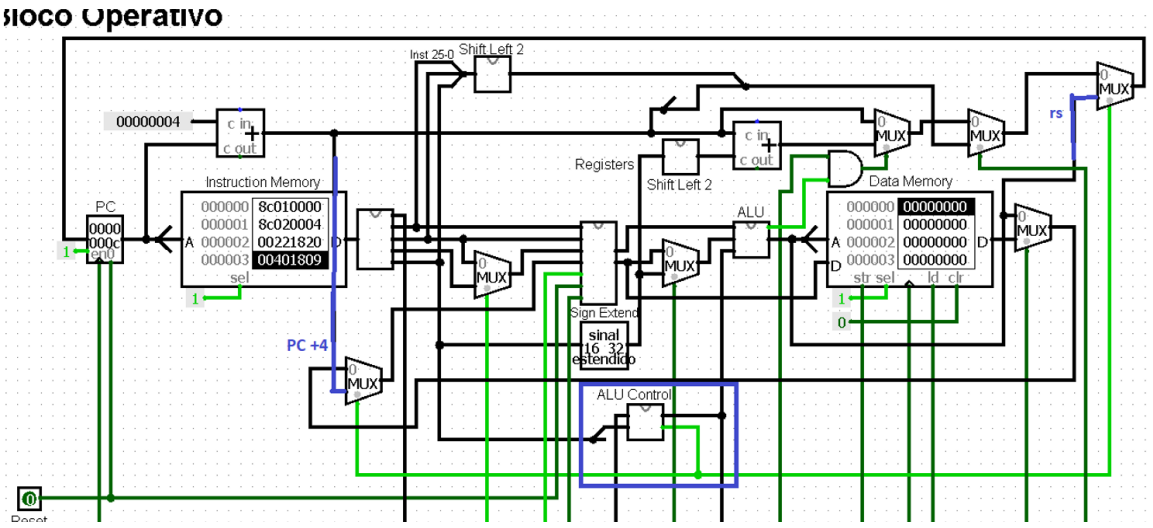
A última alteração necessária no circuito foi a adição dos bits de controle no endereço 0E, em hexadecimal, que corresponde ao opcode da instrução XORI (001110) em 6 bits. No bloco de controle, foi adicionado o valor 740 em hexadecimal, ou 11101000000 em binário de 11 bits. Os valores ligados em 1 correspondem, respectivamente, ao bit XorIFlag (já explicado anteriormente qual sua função), o bit RegWrite (já que iremos escrever em um registrador), o bit ALUSrc (selecionamos o segundo operando da ALU pelo opcode, campo immediate, estendido para 32 bits) e o bit ALUop1 (como ALUop0 é 0, pela lógica multiplexada explicada acima teremos os bits de seleção igual a 100 na ALU).

JALR

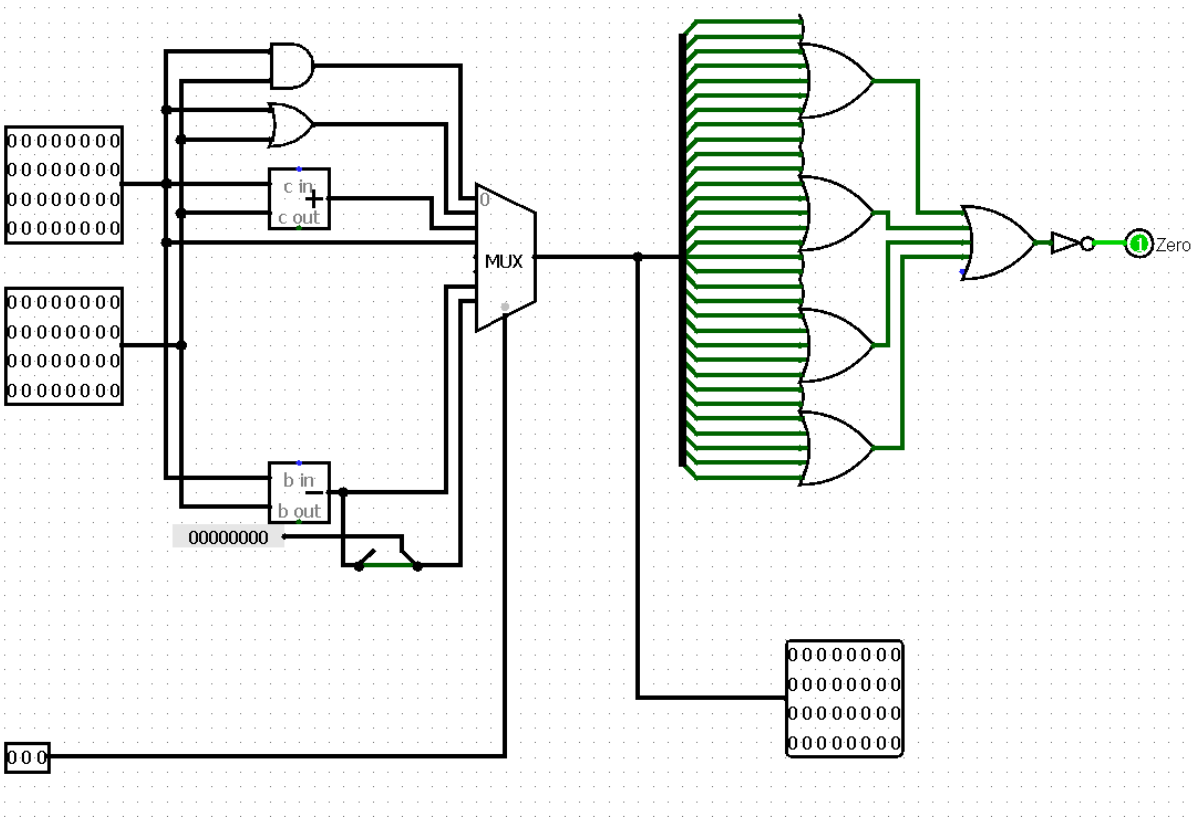
Para implementação da instrução JALR, primeiro tivemos que modificar a ALU_Control. Dentro dela foi adicionado o sinal JALR, toda vez que uma instrução do tipo R é executada com o campo funct igual a 001001 ele é emitido.



Esse sinal faz o controle de 2 multiplexadores, um deles envia o PC+4 para o dado de escrita e o outro envia o valor do registrador rs, presente na ALU, para o PC.

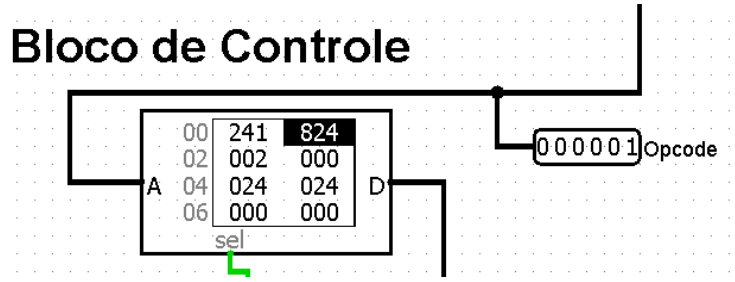


Além disso, foi adicionado um bypass dentro da ALU, assim toda vez que a ALU receber o sinal 011 ela retornará o dado do registrador rs.



BGEZ

Para implementação do BGEZ no monociclo, o caminho seguido foi inserir um novo sinal de controle, denominado ‘BranchOp’. A ativação desse sinal pela ROM que determina os estados no monociclo é dada quando o OpCode da instrução é igual a 000001.

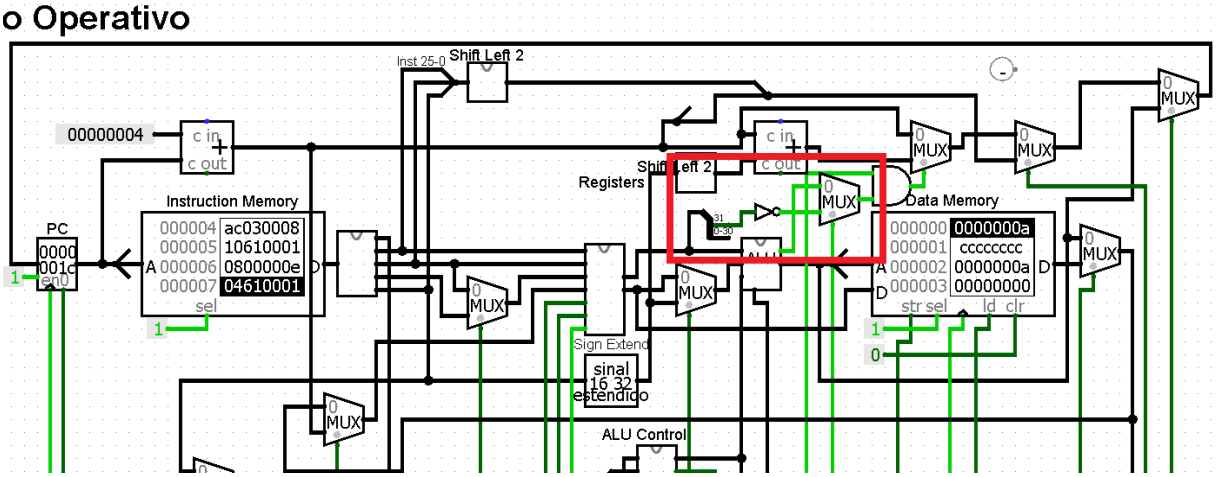


Como podemos ver na figura acima, o estado de controle em hexadecimal retornado pelo bloco de controle é 824, isto é, 100000100100 em binário. Sendo assim, os sinais que estarão ativos quando a operação for um BGEZ são:

- Branch
- AluOp0
- BranchOp

Em se tratando de como o Bloco Operativo utiliza esse novo sinal, o fluxo de execução segue similar à instrução ‘beq’, com uma alteração importante. Na instrução ‘beq’, o determinante para o circuito escolher se o branch deve ser ou não tomado é a saída Zero da ALU for positiva. No circuito aprimorado, agora há um Mux decidindo qual será o determinante pro branch ser tomado: a saída Zero da ALU ser positiva, ou, no caso adicionado, rs for maior ou igual a zero. O controle desse Mux é, portanto, o sinal ‘BranchOp’ adicionado.

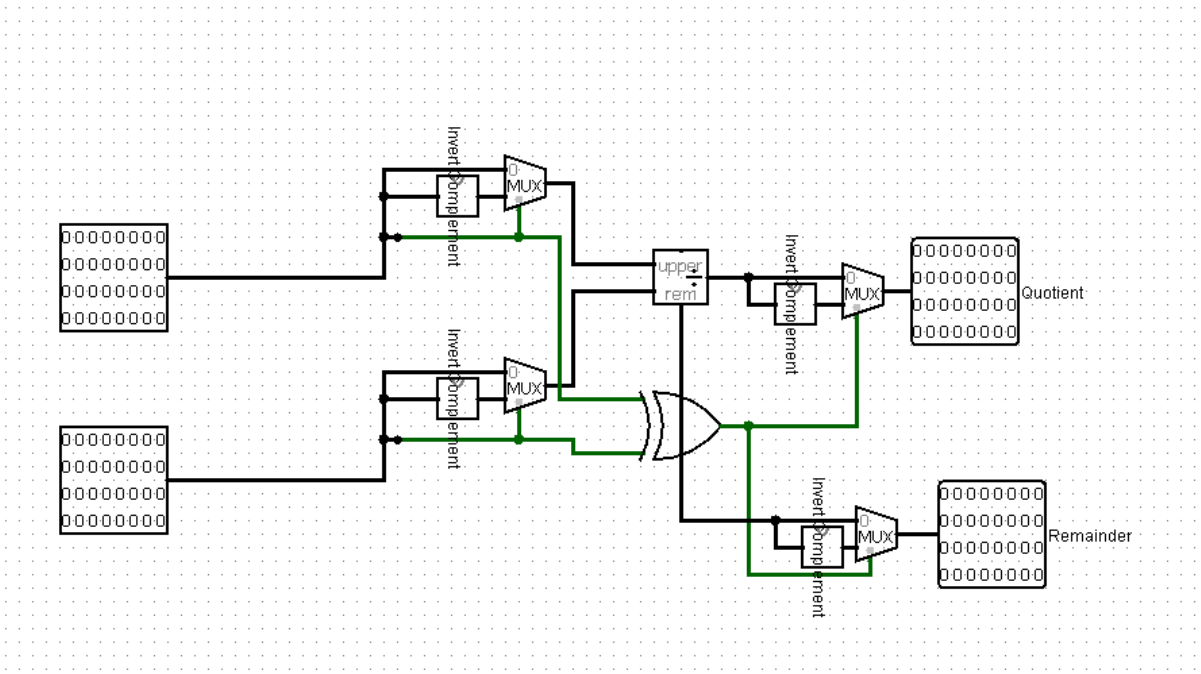
o Operativo



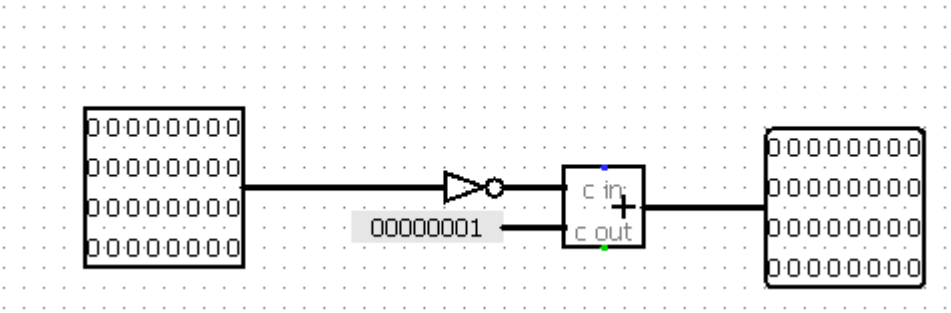
A lógica utilizada para determinar se o rs é maior ou igual a zero foi simples e eficiente: pega-se o bit mais significativo e nega-se sua saída. Se o resultado da negação for positivo, então o número é maior ou igual a zero. O restante do circuito dá-se da mesma maneira que um ‘beq’: PC receberá o valor da soma de PC com offset.

DIV

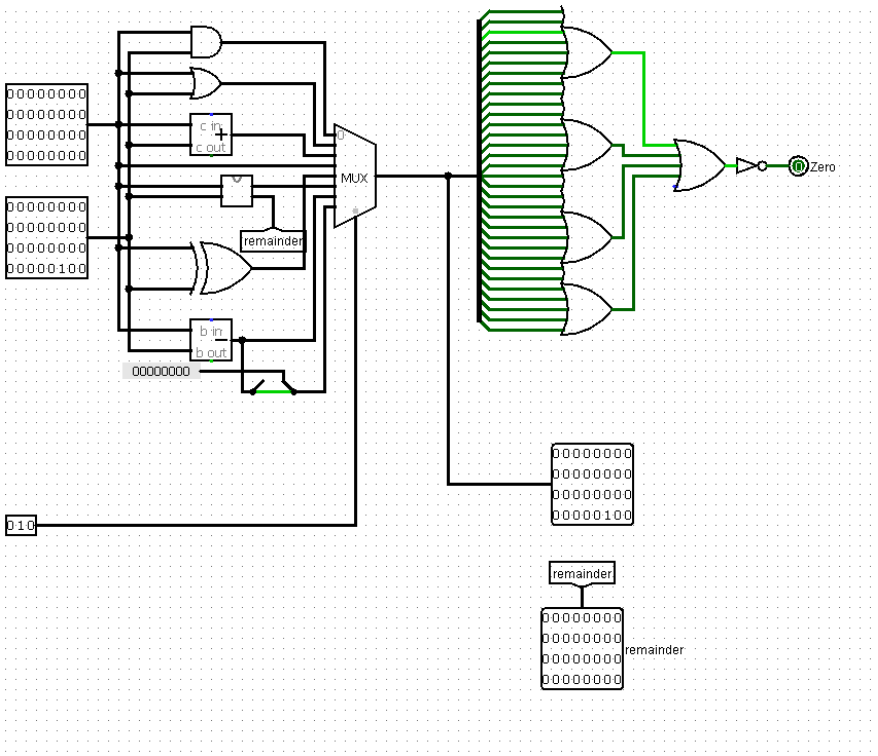
O primeiro componente necessário para a implementação da instrução DIV é um circuito combinacional que realiza divisão de números com sinal. Para tal, checamos o bit 31 das entradas com o objetivo de descobrir se os números são negativos. Nós invertemos o complemento de 2 para conseguir a versão positiva dos números negativos e realizamos a divisão entre eles. Nós temos agora o resultado da divisão do valor absoluto dos números. Uma porta XOR ligada aos bits 31 das entradas nos permite saber se apenas um dos valores de entrada for negativo. Se esse for o caso, nós invertemos as saídas novamente, gerando resultados negativos (em complemento de 2). O circuito completo consta a seguir:



Para inverter o complemento de 2, negamos o valor e somamos 1 ao resultado:

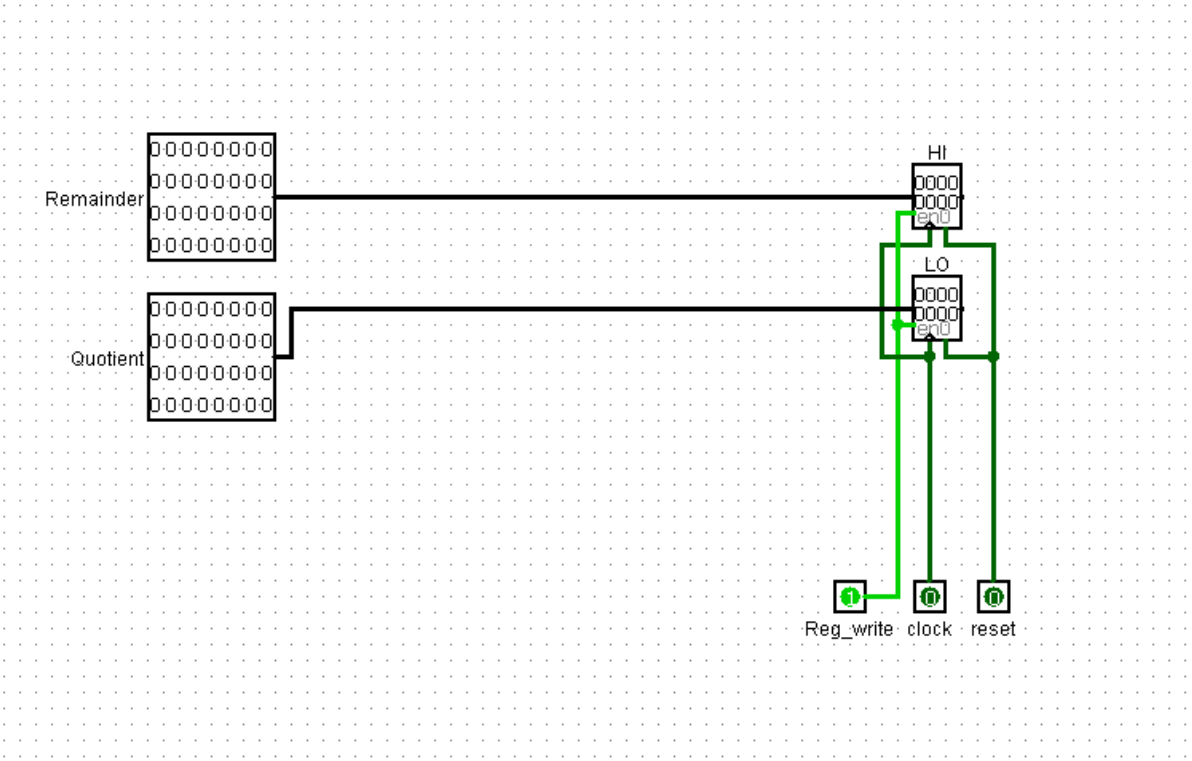


Essa instrução é então ligada na porta 101 do MUX da ALU, e uma segunda porta de saída é adicionada ao circuito, sem estar conectada ao MUX, já que a operação de divisão gera duas saídas: o quociente e o resto.

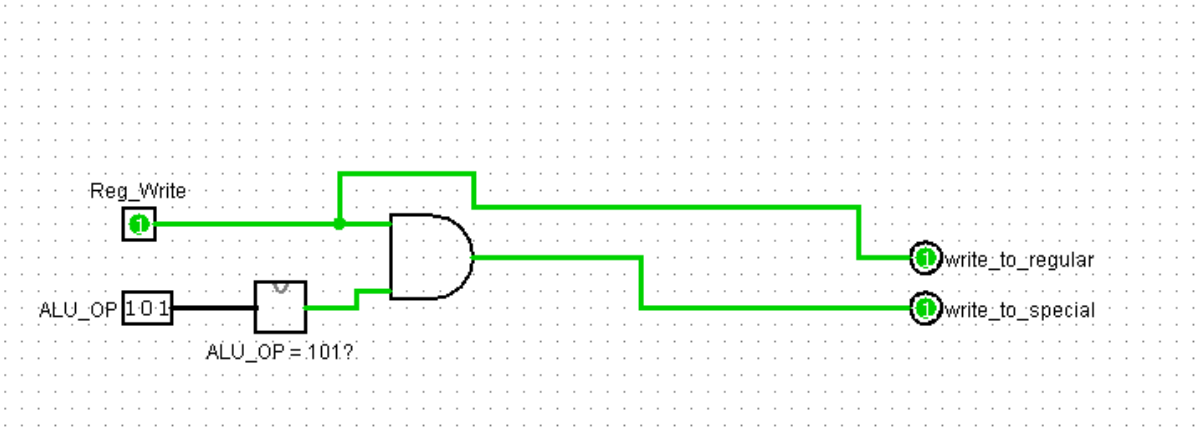


Como DIV é uma instrução do tipo **R**, o ALU CONTROL precisa ser modificado para que o func 011010 gere o código “101” quando o sinal de controle ALUOp1 estiver ligado. Felizmente, as modificações feitas no ALU CONTROL para a implementação de XORI e JALR já garantem esse resultado.

A próxima questão é como lidar com a escrita. A operação DIV não escreve nos registradores normais. De fato, ela escreve em dois registradores especiais ao mesmo tempo! O quociente é escrito em um registrador chamado “LO” e o resto é escrito em um registrador chamado “HI”. Nós criamos um banco de registradores especiais contendo esses dois registradores, assim facilitando a escrita.



Porém, nós precisamos saber quando escrever nesses registradores. Existem duas opções: mudar o controle, ou manipular o controle no caminho de dados. Com o objetivo de manter o controle das instruções do tipo R uniforme, foi escolhido o segundo método. Um circuito chamado “select bank” foi criado. Ele verifica se o sinal vindo da ALU OP é 101 e se o sinal de REG WRITE está ligado. Se sim, ele manda um sinal de escrita para o banco de registradores especiais. Note que não há necessidade de mascarar o REG WRITE mesmo quando queremos escrever para o HI/LO, já que a posição do registrador de destino na instrução DIV sempre contém zero, e escritas para o registrador zero são ignoradas.



O sub-circuito marcado como “ALU_OP = 101?” apenas utiliza uma porta lógica AND para verificar se a saída do ALU_CONTROL é 101. Esses passos

implementam a instrução DIV com sucesso no MIPS monociclo. Note que o “resto” que sai da ALU é ignorado por todas as outras instruções, já que ele está conectado apenas ao banco de registradores especiais.

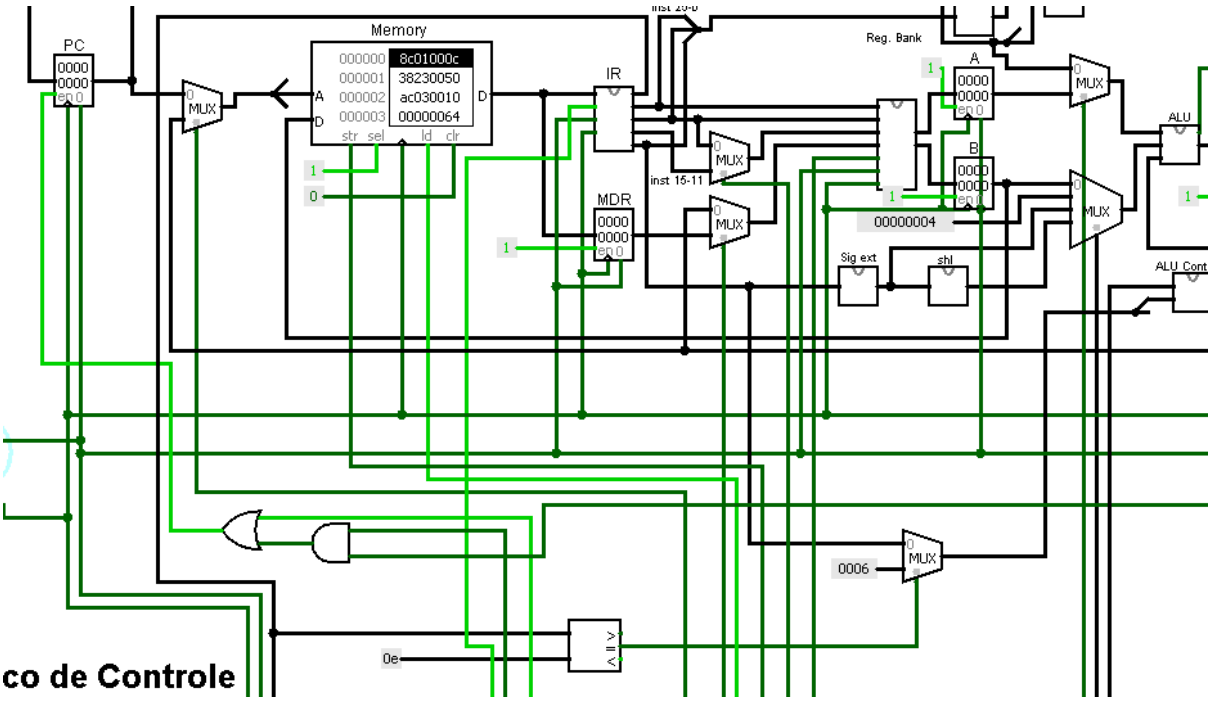
MIPS Multiciclo

XORI

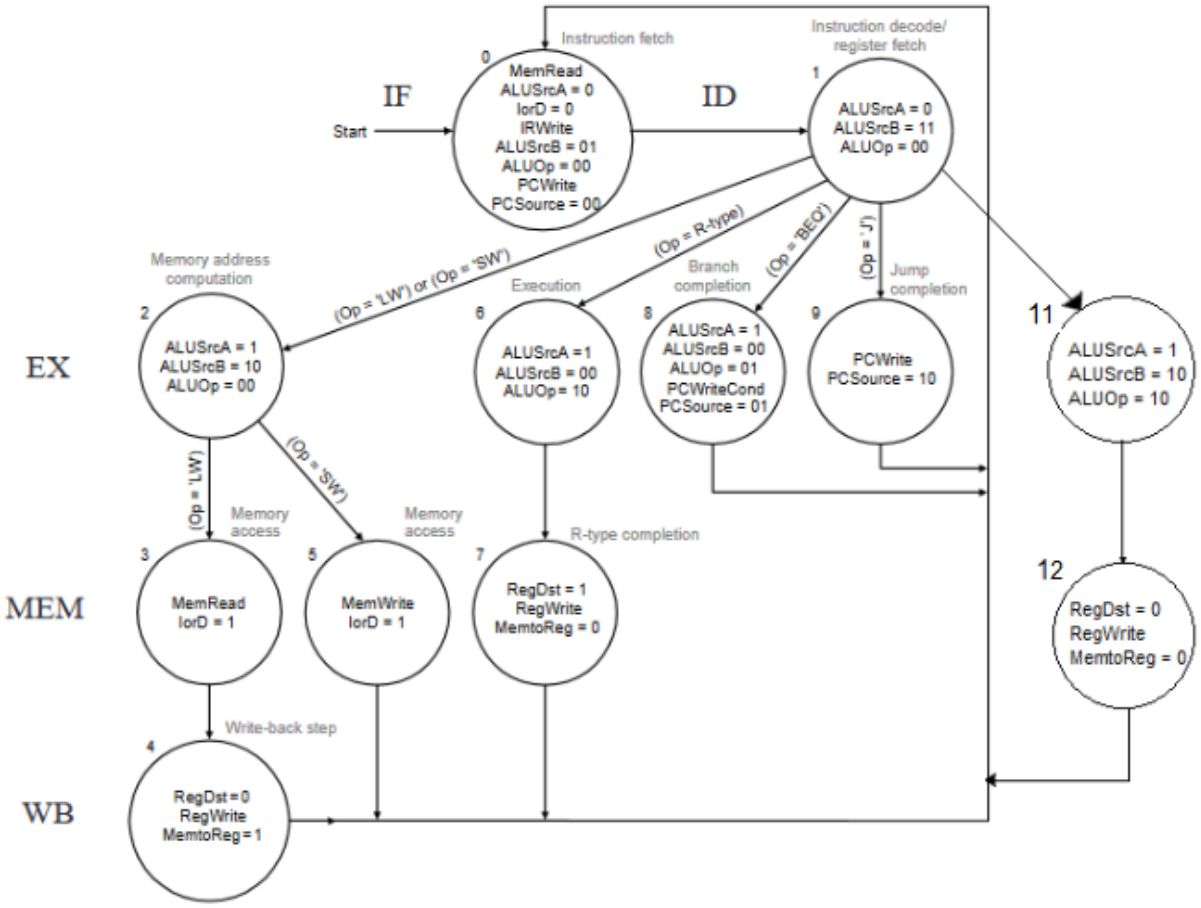
Para implementação da instrução XORI no MIPS Multiciclo, seguimos o mesmo passo de adicionar uma porta lógica XOR dentro da ALU do circuito disponibilizado. As modificações são as mesmas explicadas para o Monociclo.

Também utilizamos a mesma lógica multiplexada do Monociclo (explicada acima) para selecionarmos corretamente a operação na ALU. A única diferença consiste em, agora, o bit de seleção desse mux ser a saída de um comparador entre o valor lido no registrador de instruções (IR) e o valor opcode constante em hexadecimal da instrução XORI (0E). Achamos de mais simples implementação seguir dessa maneira, tendo em vista que os bits de controle poderiam sofrer grandes alterações nessa arquitetura caso adicionássemos uma *flag* como no Monociclo.

Da mesma forma, quando o valor em IR for igual a 0E, a saída do comparador é 1, que seleciona a entrada 1 (com a constante 1) para ser encaminhada à entrada de ALU Control, gerando os bits 100 (4, em decimal) de seleção na ALU e, portanto, selecionando a operação XOR:



Foi necessária a alteração da máquina de estados para suportar a instrução XORI. Dois estados novos foram adicionados, 11 e 12 (em decimal). No estado 11, o bit de controle ALUSrcA está em 1, indicando que usaremos o registrador A (lido do banco de registradores), o bit de controle ALUSrcB está em 10 (binário), indicando que usaremos o valor imediato lido do opcode, e o bit de controle ALUOp está em 10 (binário) pelos motivos explicados no Monociclo para uso da operação XOR. O estado 10 corresponde ao passo 3 da instrução XORI, após passar pelos estados e passos comuns à todas as instruções. No estado 12, passo 4, o bit de controle RegDst está em 0, já que guardaremos o resultado da operação no registrador em rt (do opcode), o bit de controle RegWrite está em 1, para escrevermos em um registrador, e MemtoReg está setado para 0, já que leremos o valor de ALU Out e não de MDR. Dessa forma, a máquina de estados atualizada para essa instrução fica a seguinte:



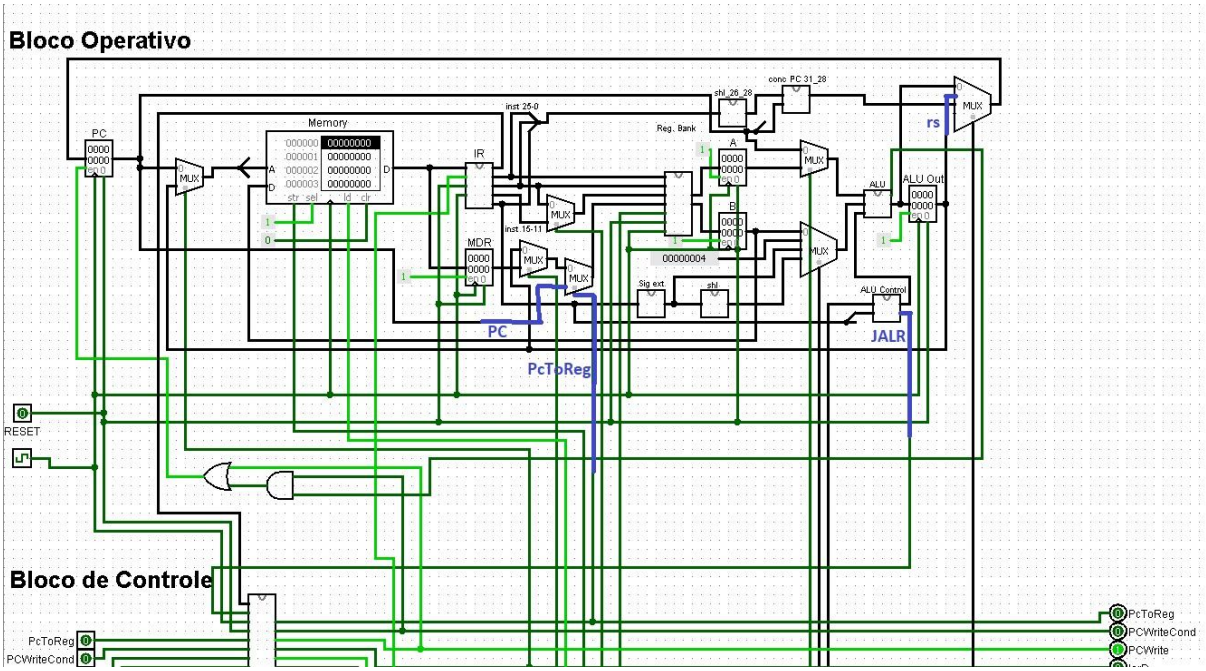
Traduzindo isso para o circuito, na FSM_ROM colocamos os valores 00068 e 00080, em hexadecimal, nos endereços (ou seja, estados) 11 e 12, respectivamente. Esses valores correspondem a 0000000000001101000 e 0000000000010000000 em binário, respectivamente, o que representa os bits ligados e desligados explicados acima.

No circuito de geração do próximo estado, a partir do endereço 0E0 (valor hexadecimal de 10 bits da instrução XORI), temos os estados que a instrução irá seguir. Após o estado 1, no endereço 0E0, a instrução segue para o estado 11 (B, em hexadecimal), salvo no endereço 0E1. O próximo estado (12, salvo como C em hexadecimal) está guardado no endereço 0EB.

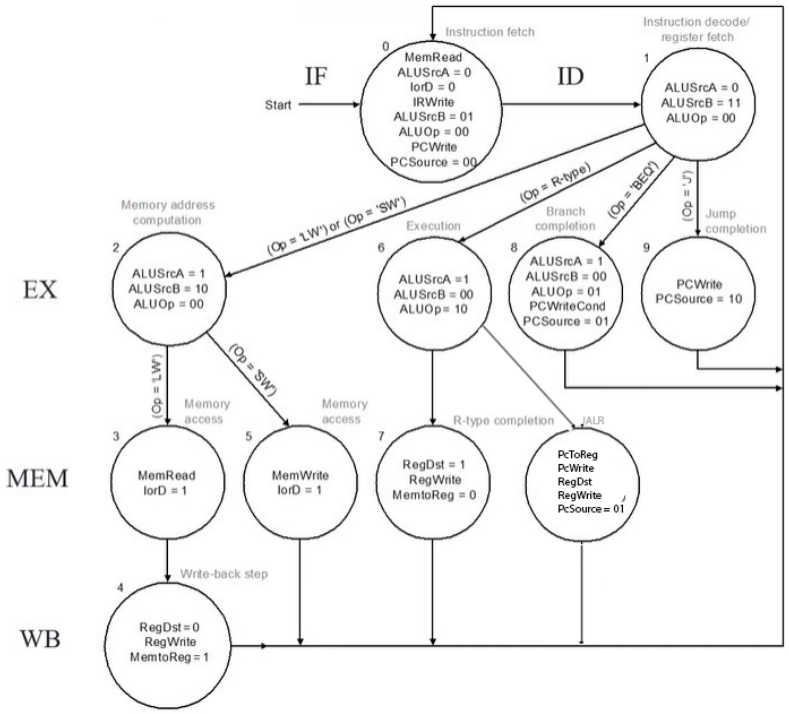
JALR

Como anteriormente, o bypass na ALU e o sinal JALR na ALU_Control foram implementados. O sinal JALR é enviado para a FSM_ROM que controla a troca de

estados, 6 para 10. Ademais, foi criado um sinal extra, PcToReg, que quando ativado sinaliza para um MUX que o dado de escrita no registrador deve ser o PC.

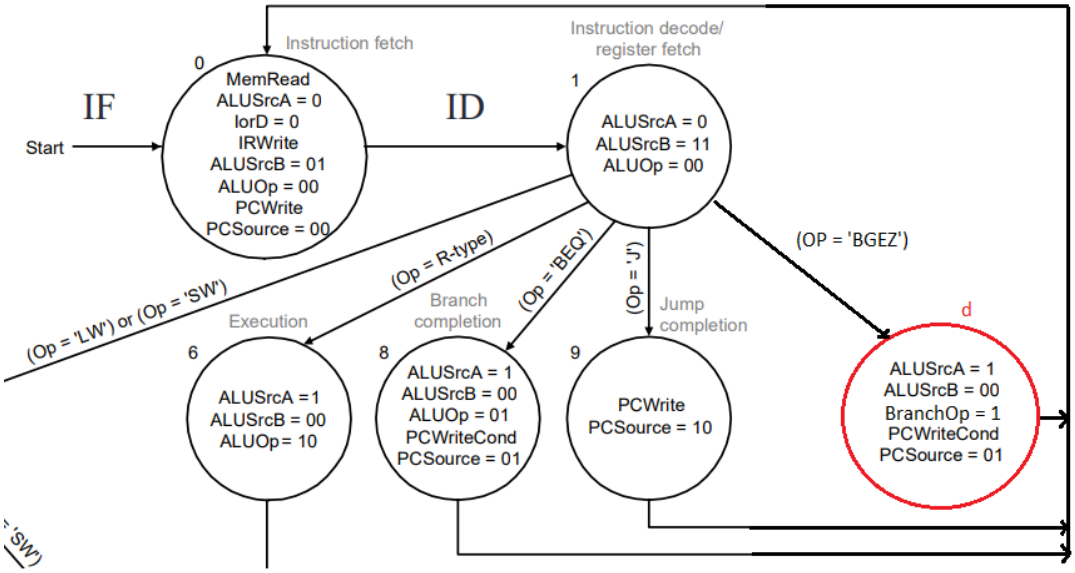


Esse estado 10 ativa os sinais PcToReg, PcWrite, RegDst, RegWrite e PcSource 01, assim conseguimos salvar o PC no registrador rd e atualizamos o PC com o valor do registrador rs.



BGEZ

Assim como no monociclo, foi implementado um novo sinal chamado ‘BranchOp’. A ativação do sinal no multiciclo é um pouco distinta, pois temos uma função de próximo estado junto a um mapeamento ROM para geração da saída. O determinante para função de próximo estado escolher qual será seu resultante, é a combinação do OpCode com o Estado anterior. O OpCode continua sendo 000001. Considerando esse OpCode, a sequência de estados é: se o estado anterior é 0, então o próximo estado é 1. Se o estado anterior é 1, então o próximo estado será ‘d’, um estado novo adicionado, conforme na figura abaixo:

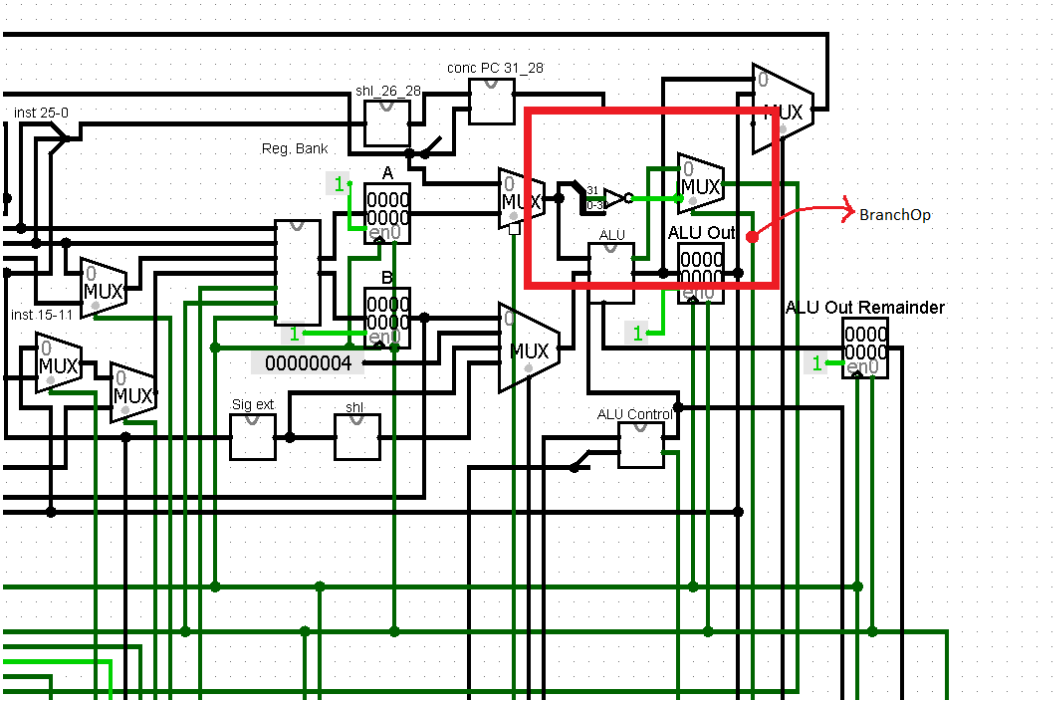


Como vemos, os sinais de controle que precisam estar ativos no estado ‘d’ são:

- AluSrcA
- BranchOp
- PCWriteCond
- PCSource = 01

Sendo assim, o estado ‘d’ foi preenchido com o hexadecimal 28041 no mapeamento do estado para os sinais.

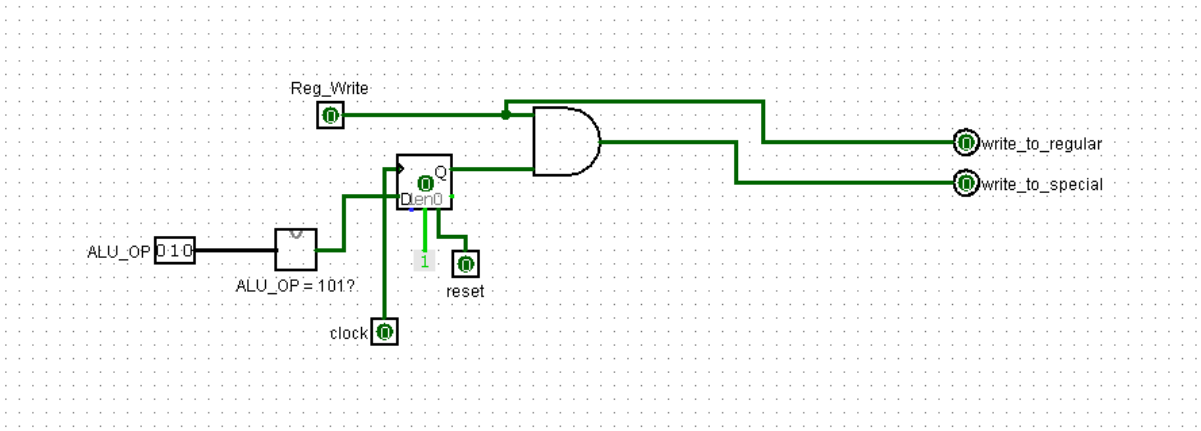
A maneira com que o Bloco Operativo usa esse sinal é similar ao monociclo. O início da instrução é similar ao ‘beq’, e novamente foi criado um Mux para decidir qual a lógica determinante para tomar ou não o branch: saída Zero da ALU for positiva (sinal 0 do Mux), ou se o bit mais significativo de rs for zero. Portanto, o controle desse Mux é dado pelo sinal ‘BranchOp’.



O restante do circuito dá-se da mesma maneira que um ‘beq’: PC receberá o valor da soma de PC com offset.

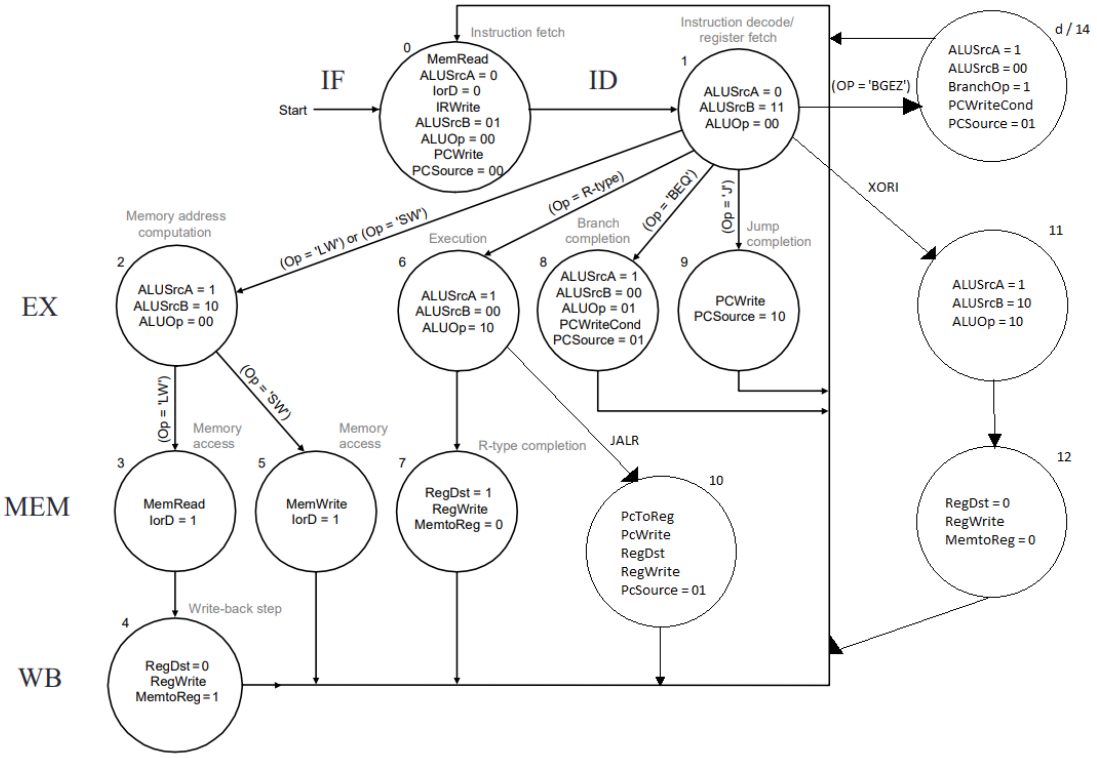
DIV

A maior parte do design feito para implementar a instrução DIV no MIPS Monociclo se mantém na organização Multiciclo. A diferença é que agora nós precisamos salvar o resultado do “resto” que sai da ALU em um registrador, e reter a informação da última operação realizada pela ALU (para ativarmos os registradores especiais caso tenha sido uma divisão). Para tal, simplesmente adicionamos um flip-flop tipo D dentro do circuito “select bank”.



As outras modificações foram triviais. Agora o quociente e o resto que entram nos registradores especiais vem dos registradores intermediários do circuito, e isso completa a versão multiciclo da operação DIV.

Em suma, ficamos com a seguinte máquina de estados para nosso MIPS Multiciclo:



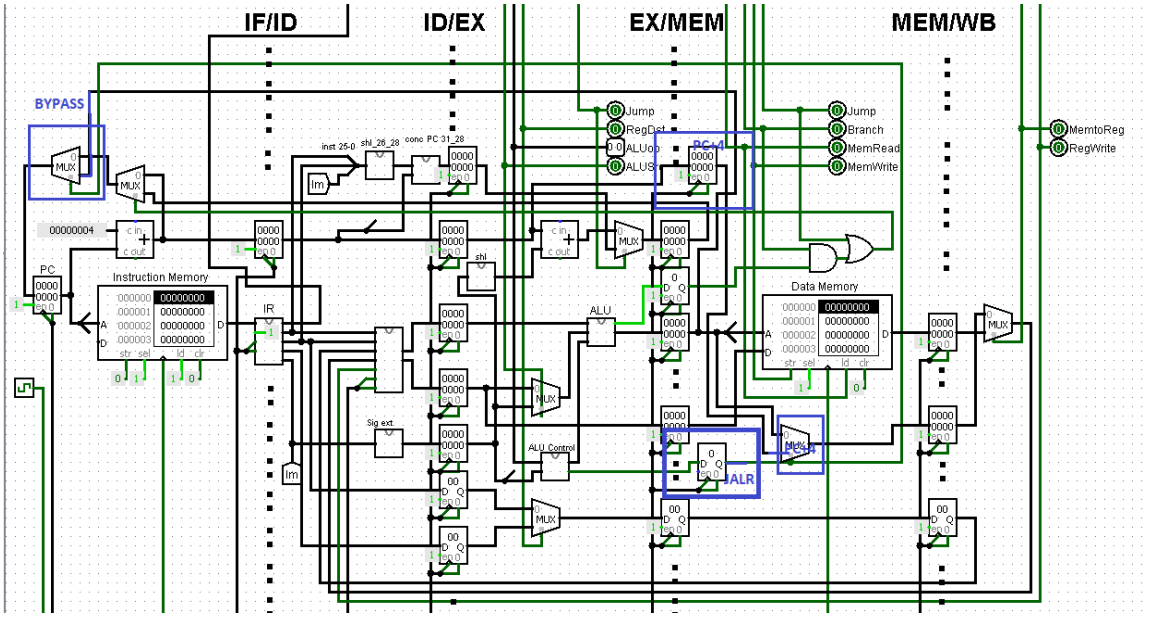
MIPS Pipeline

XORI

Para implementação da instrução XORI no MIPS Pipeline, os passos foram bastante similares à implementação no MIPS Monociclo. Adicionamos a porta XOR da mesma forma na ALU, utilizamos o multiplexador advindo do Instruction Register e a constante 6 e adicionamos o bit de controle XorIFlag para selecionar o mux. No circuito disponibilizado, o valor binário necessário para a instrução XORI é 10010100001 (ativando e desativando os mesmos bits explicados no Monociclo), equivalente à 4A1 em hexadecimal, que foi salvo no endereço 0E do bloco de controle.

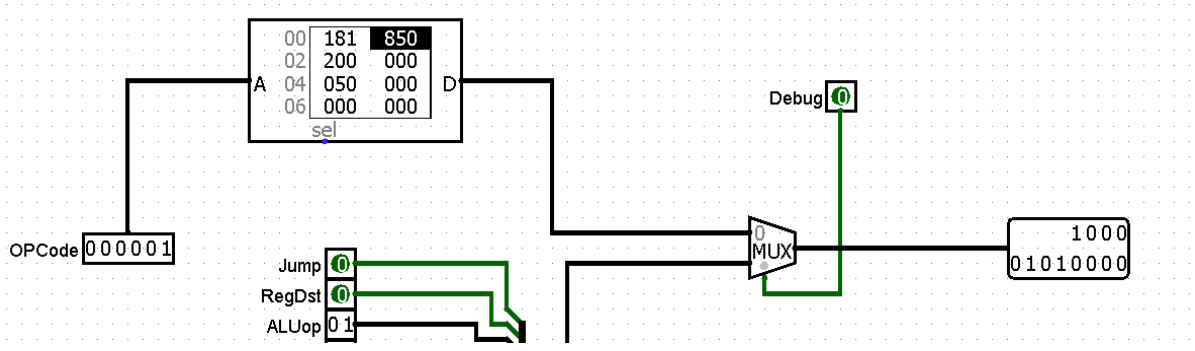
JALR

A implementação das versões anteriores, como o bypass na ALU e o sinal JALR de controle dentro da ALU_Control foram mantidos. Esse sinal foi passado para o próximo estado do pipeline (EX/MEM), nesse mesmo estado foi adicionado outro registrador que guarda o PC+4 e um multiplexador que aliado do sinal JALR faz o controle do dado que vai para a escrita do registrador. Além disso, o sinal JALR é mandado para outro multiplexador que faz o controle do dado que vai para o PC, ou o bypass da ALU, ou a saída padrão do pipeline.

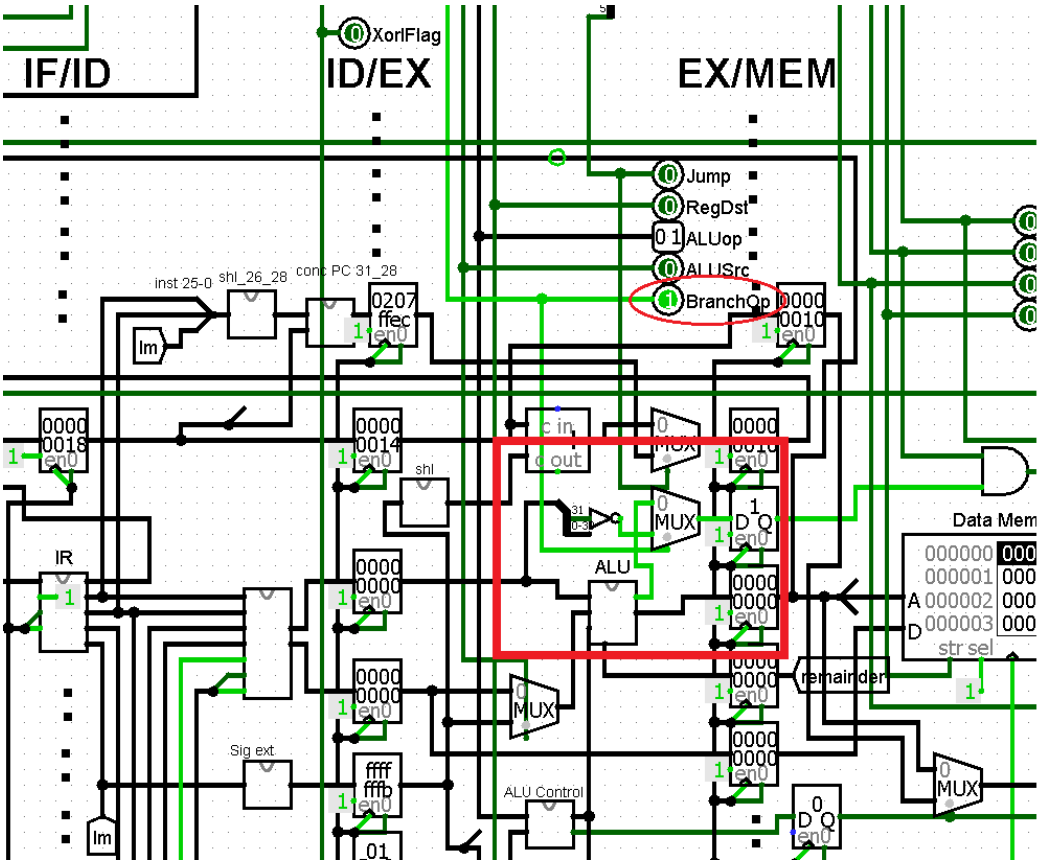


BGEZ

A implementação do BGEZ para a versão Pipeline foi muito similar às anteriores em monociclo e multiciclo. Foi adicionado o sinal ‘BranchOp’, que neste caso será ligado quando o OpCode (entrada da ROM de controle) for 000001. Além do sinal BranchOp, os sinais necessários são ALUOp = 01 e Branch. Portanto, o número em hexadecimal adicionado na posição 01 da ROM é 850.



No bloco de controle, a alteração foi feita no estágio de execução: assim como nas outras implementações, um multiplexador foi adicionado para escolher a lógica para tomar-se ou não o branch: saída Zero da ALU for positiva (‘beq’) ou o bit mais significativo for zero. Portanto, o controle desse Mux é dado por esse novo sinal de controle (BranchOp).



O restante do circuito dá-se como no ‘beq’, isto é, se o branch deve ser tomado, PC receberá o novo valor dado pela soma com o offset da instrução.

DIV

Novamente podemos reaproveitar boa parte do design feito para as organizações anteriores. Agora, o “resto” que sai da ALU precisa ser preservado por dois ciclos e o mesmo é válido para a saída do ALU CONTROL. Diferente do MIPS multiciclo, nós guardamos a saída do ALU CONTROL em registradores intermediários, para espelhar o padrão geral da organização. O circuito “select bank”, então, voltou a ser o mesmo do monociclo, já que ele recebe o ALU CONTROL preservado pelo segundo registrador intermediário. O resto do design é preservado do Monociclo (tirando, é claro, o detalhe que o quociente passa por dois registradores intermediários).