Universidade Federal do Rio Grande do Sul
Instituto de Informática

Bruno Ferreira Aires
Guilherme Schaab Fisch
João Vitor Moreira Dias
Manoel Narciso Reis Soares Filho
Richard Leal Ramos

Implementação de organizações MIPS nas versões monociclo e multiciclo e adaptação de versão com pipeline para inclusão de instruções

Porto Alegre - 2023

Bruno Ferreira Aires

Guilherme Schaab Fisch

João Vitor Moreira Dias

Manoel Narciso Reis Soares Filho

Richard Leal Ramos

Implementação de organizações MIPS nas versões monociclo e multiciclo e adaptação de versão com pipeline para inclusão de instruções

Trabalho apresentado no curso de graduação em Engenharia da Computação da Universidade Federal do Rio Grande do Sul como avaliação para a disciplina de Organização de Computadores ministrada pelo professor Luigi Carro.

Porto Alegre - 2023

Introdução

Neste artigo discorremos sobre nossas implementações de duas organizações de computadores, com arquitetura MIPS 32 bits, nas versões monociclo e multiciclo. Também apresentamos a adaptação de uma organização com pipeline para a inclusão de cinco novas instruções. A citada organização com pipeline original foi desenvolvida pelos professores Luigi Carro[1] e Antônio Beck[2] .

Para desenvolvimento das duas organizações e adaptação da terceira foi-se utilizado o software Logisim[3], que permite a elaboração e simulação de circuitos digitais. Em diversas situações também recorremos ao auxílio do montador MARS[4], na tarefa de codificar as instruções desejadas e também validar nossos programas de teste, para então verificar o funcionamento de nossos computadores implantados.

Particularidades de cada implementação serão brevemente explicadas no capítulos seguintes, mas unanimemente cada uma delas oferece suporte às instruções básicas de aritmética, transferência de dados, saltos e comparações, além da garantia de funcionamento das seguintes instruções: JR, ADDI, MULT, LBU e BLTZAL. Definições mais detalhadas acerca destas instruções destacadas são apresentadas no apêndice chamado “Conjunto de instruções destacadas para análise”, no final deste documento.

Nos capítulos “MIPS Monociclo” e “MIPS Multiciclo” uma visão geral de cada uma destas implementações será apresentada, e uma depuração mais detalhada será feita no funcionamento das cinco instruções em destaque, novamente para cada implementação. O capítulo “Adaptação do MIPS Pipeline” se atém à descrição das alterações incutidas na organização original a fins de implementar o conjunto de instruções

MIPS Monociclo

Criamos uma versão completa de um computador monociclo baseado na arquitetura MIPS. Foram implementadas as instruções básicas como LW(load word), SW(store word), ADD, entre outras, além de instruções mais específicas como por exemplo MFHI(Move from High), MFC0(Move from control), SUBI(SUB immediate), e logicamente as instruções requeridas no enunciado do trabalho, sendo elas BLTZAL, LBU, ADDI, MULT e JR. Uma especificação mais detalhada das instruções requeridas no trabalho podem ser consultadas no capítulo anterior e a lista de todas as instruções implementadas podem ser vistas na figura 2.

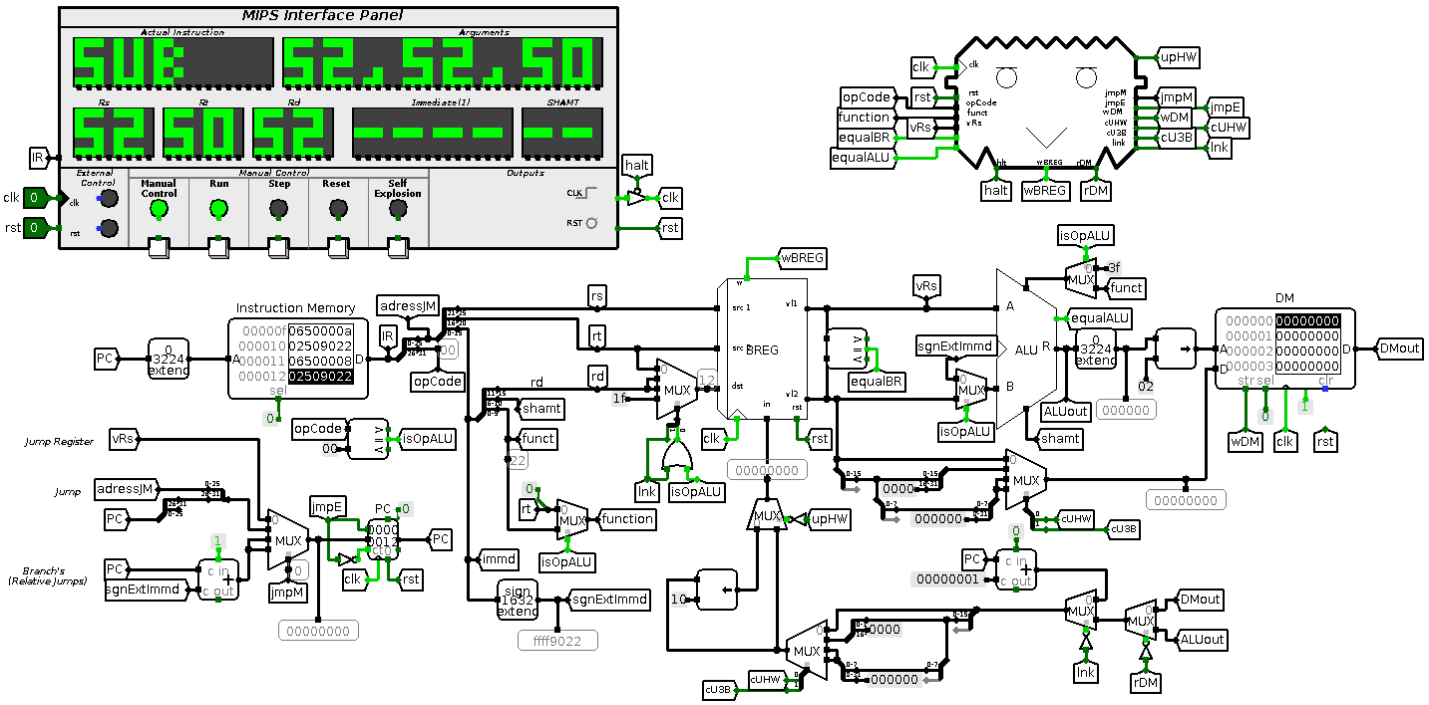


Figura 1: Corte panorâmico da implementação da versão do MIPS monociclo. Na parte superior esquerda se encontra o módulo de controle enquanto a parte operativa ocupa toda a metade inferior do circuito. No canto superior esquerdo é possível observar um painel que facilita a interação e depuração do computador.

Foge do escopo deste relatório apresentar uma lista prolixa do esquema de funcionamento de cada uma das instruções implementadas, e portanto discutiremos em detalhes somente sobre a implementação das cinco instruções destacadas.

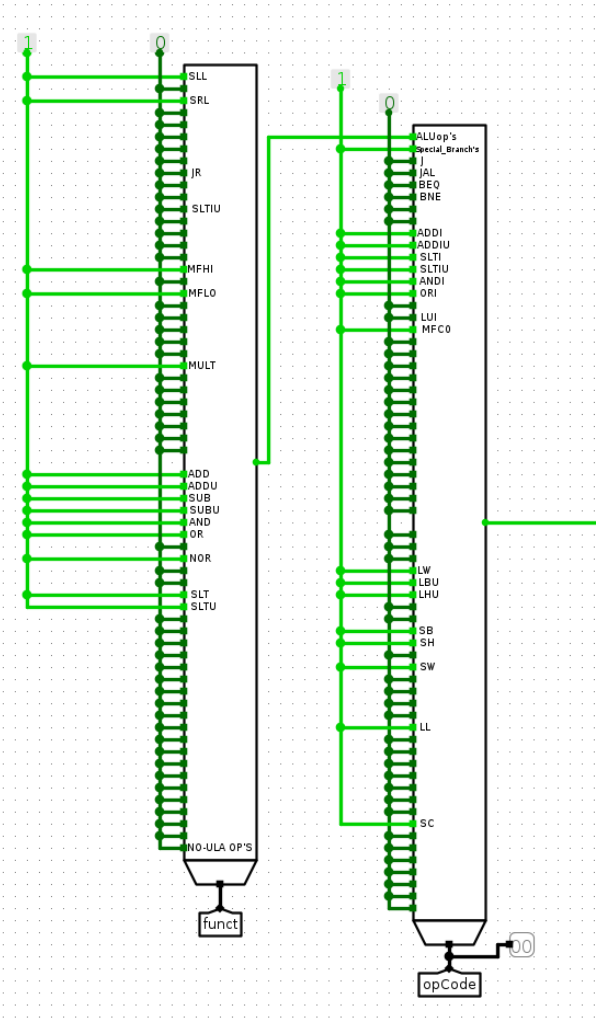


Figura 2: Multiplexadores usados em várias partes do computador que servem também como lista das instruções implementadas. Fica de ressalva que algumas destas instruções ainda estão em fase de ajuste e outras foram somente esboçadas no circuito, como é o caso das instruções SC(Store Conditional) e LL(Load Linked).

➢ JR - Jump Register

A instrução de Jump Register, possivelmente a mais simples das cinco, foi implementada basicamente utilizando-se de um multiplexador para a entrada do nosso Program Counter(PC), e a manipulação correta de dois sinais de controle, emitidos pela Unidade de Controle(UC), chamados de jmpM(JuMP Mode) e jmpE(JuMP Enable). A figura 3 nos mostra o PC e seu mecanismo de chaveamento,

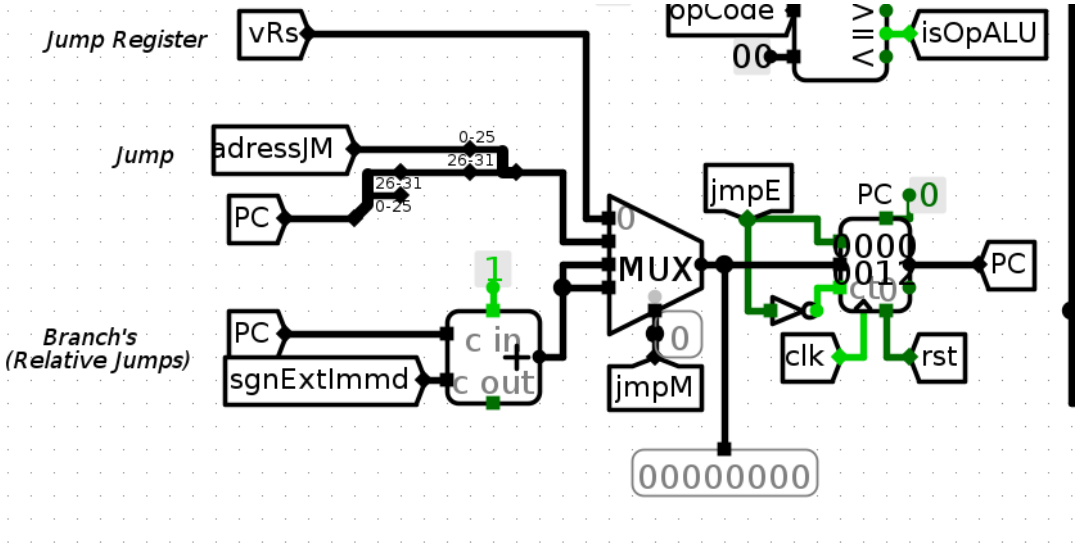


Figura 3: PC do MIPS monociclo e o multiplexador que permite selecionar o tipo de salto que programa executará, a depender do valor de jmpM(0 -> JR, 1-> J, 2 ou 3 -> Branch's). É importante destacar que só ocorrerá o salto determinado se o sinal de controle jmpE é ativo.

onde como observável pelo leitor, caso o sinal de controle jmpE esteja ativo o valor salvo por PC, que aponta o endereço - na Memória de Instruções - da próxima instrução a ser executada, será substituído, ocasionando um salto de execução na visão do usuário. A natureza desse salto é determinada pelo outro sinal de controle jmpM. Quando jmpM possui valor zero, então o valor obtido da primeira saída do Banco de Registradores, que é endereçada pelo valor Rs - Register Source, indicado pelos bits 25-21 da word de instrução -, é então carregado no PC. Uma segunda opção de salto se trata do salto provocado pela instrução Jump(J), onde os 26 bits menos significativos do PC são carregados pelo valor passado como argumento à instrução. Nesse caso o valor de jmpM emitido pela Unidade de Controle(UC) é 01. O último modo possível de salto corresponde aos causados pelas intruções de branch, quais sejam, BEQ, BNE e as instruções por mim chamadas de Branch's Especiais, as quais possuem OpCode igual a 000001. Nesse último modo o valor imediato, de 16 bits de largura, passado como argumento a tais instruções são adicionados ao PC através do somador indicado na figura. Esse último modo voltará a ser comentado quando detalharmos a implementação da instrução BLTZAL, uma vez que essa se enquadra no conjunto dos Branch's Especiais.

Explicado os modos de saltos possíveis fazemos um último esclarecimento quanto ao funcionamento ordinário do PC, ou seja, nos casos em que não devem ocorrer saltos, e, portanto o sinal de controle jmpE é desativado. Nesse caso ordinário o PC incrementará o seu valor automaticamente, uma vez que foi implementado com o uso de um contador, e portanto a cada borda de subida do sinal de clock, esse contador terá o seu valor incrementado, sendo que somente será carregado o valor de entrada quando o sinal de load(jmpE) for ativo. A figura 4 explicita taxativamente do que se trata o label vRS, sem a necessidade de comentários mais profundos, e a

figura 5 apresenta um corte da UC, que é o sub-circuito otimizado de determinação do valor de jmpM.

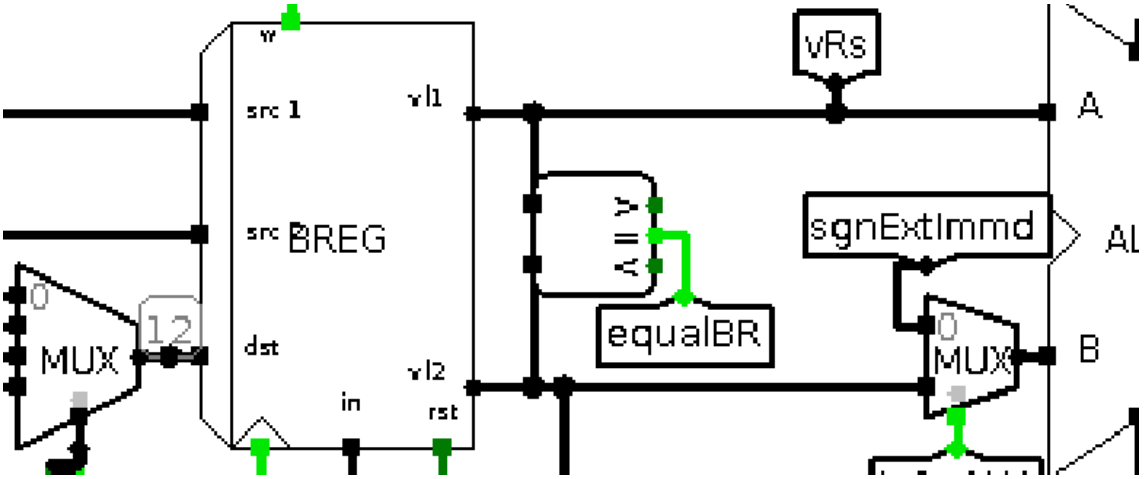


Figura 4: A label vRS, anteriormente citada, corresponde ao valor da primeira saída do BREG, endereçado pelo valor de Rs.

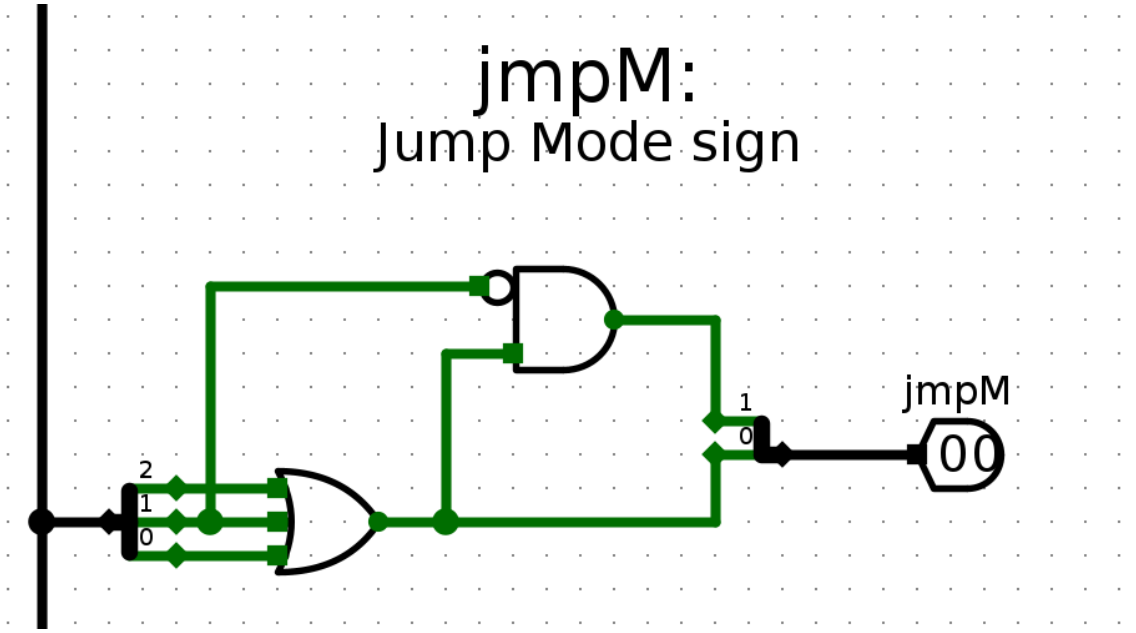


Figura 5: Sub-circuito de determinação do valor de jmpM. O sinal vindo da esquerda corresponde ao opCode da instrução atual, ou seja os 6 bits mais significativos da word de instrução.

Como se vê na figura 5, para evitar o uso de um multiplexador para determinar de forma extenuante o valor de jmpM para cada opCode, usamos uma porta AND, uma OR e outra NOT, para processar os 3 bits menos significativos de opCode. Explicamos aqui seu funcionamento detalhado: Como o valor de jmpM só tem significado nas instruções de salto, quando jmpE é ativo, logo basta restringirmos o domínio da função, implementada em

hardware, que determina jmpM para as instruções JR, SPECIAL_BRANCH's, J, BEQ, BNE e SPECIAL_BRANCH's. Este restrito conjunto de instruções correspondem aos opCode's 0, 1, 2, 4 e 5, nesta ordem. Em representação binária tais instruções equivalem a 000, 001, 010, 100 e 101. Portanto quando os 3 bits forem zero jmpM será 00, como devia de ser e como de fato será uma vez que a saída porta OR será 0 levando a saída porta AND também a ser. Atendendo essa instrução, nos restam dois modos de salto, de um lado os branch's(001, 100 e 101) requerem que jmpM seja 11 e do outro Jump exige que jmpM seja 01. Para diferenciar isso no bit 1 de jumM posicionamos então a porta NOT na entrada da AND, garantindo então que somente no caso da instrução Jump o bit jmpM_1 seja 0, já que do nosso restrito conjunto de opCodes essa instrução é a única que possui bit 1 ativo.

- **LBU - Load Byte Unsigned**

Essa instrução de load se assemelha a sua irmã mais famosa LW(Load Word), onde ambas carregam um valor da memória de dados em um registrador de BREG. Mais que uma semelhança, a mecânica de funcionamento destas duas instruções são idênticas em quase todas as suas abrangências: O valor de um registrador (Rs ou base) é somado, na ALU, aos 16 bits menos significativos da word(immediato) e esse valor indica o endereço da célula de memória de dados cujo valor residente, de 32 bits, será enviado para BREG, para ser salvo noutro registrador - de destino -, indicado por Rt. A disparidade de LBU para LW acontece na reta final, somente no instante de salvamento do valor da memória no registrador, onde no caso de LW o valor de 32 bits é salvo integralmente em BREG, e no caso de LBU esse valor de 32 bits passa por um filtro que zero seus 24 bits mais significativos. Em outras palavras, apenas os 8 bits menos significativos do valor provindo da memória serão salvos no registrador alvo, sendo todos os outros bits transformados em zero. Dadas as semelhanças entre a instrução LBU com a célebre LW, a explicação das etapas iniciais será achatada em poucas palavras: O registrador apontado por Rs é acessado e seu valor é somado na ALU ao valor imediato, após este ter passado pelo extensor de sinal, vale lembrar. O resultado da soma é então aplicado no indicador de endereço para leitura/escrita da memória, mas é o sinal de leitura que é ativo. O valor lido então é enviado para BREG, passando antes pelo supracitado filtro no caso de LBU. Os sinais de controles emitidos pela UC para seleção dos multiplexadores e controle do BREG, ALU e Memória podem ser mapeados em uma rápida análise do circuito.

A parte do filtro de entrada do BREG, projetado então para instruções de load parcial como LBU, pode ser vista na figura 6. O dado vindo da memória é transformado em duas versões, sendo a primeira com os 16 bits mais significativos transformados em zero(para a instrução LHW- load half word) e a segunda com os 24 bits menos significativos transformados em zero. Os sinais de cUHW e cU3B controlam o dado multiplexado para a entrada de BREG. cUHW significa Clear Uper Half Word, e é um comando da UC para que o filtro "limpe" metade do dado e cU3B(Clear Uper 3 Bytes) ordena que o filtro limpe

os 3 Bytes mais siginificativos). No caso de LHW apenas cUHW é enviado e no caso de LBU ambos são enviados pela UC.

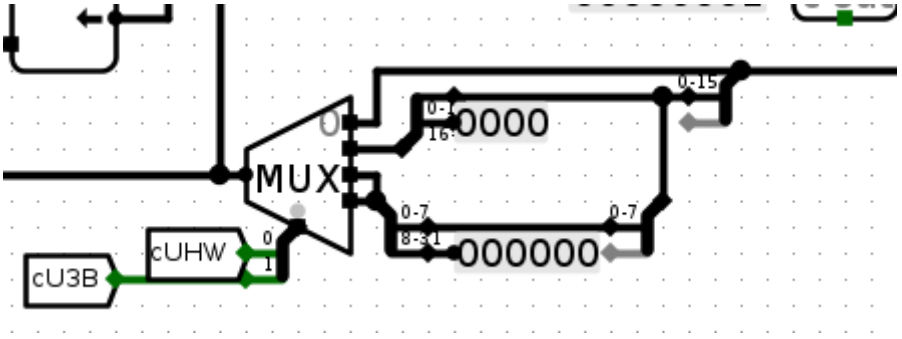


Figura 6: Filtro da entrada de BREG. Modo 0 transmite o valor integral, modo 1 zera a half-word mais significativa e modo 2/3 transmite o byte menos significativo intacto e o restante do dado de 32b zerado.

➢ **ADDI - Add Immediate**

Amada pelos programadores - de forma platônica caso se trate de um programador do Neander, é bem verdade -, a instrução realiza a adição do valor salvo em um registrador com um valor imediato, definido na própria codificação da instrução. Na nossa organização a instrução de ADDI segue um funcionamento similar à instrução ADD. A figura 7 exibe enfoca o multiplexador que seleciona qual valor será enviado à segunda entrada da ALU, sendo ou o valor vindo da segunda saída de BREG(comportamento desempenhado na instrução ADD) ou o valor immediato em 32 bits, com o sinal estendido (que é esperado na instrução ADDI). O sinal de seleção é determinado na própria parte operativa(figura 8) simplesmente verificando se o oPCode atual é zero. O conjunto de operações que requerem a segunda saída da ALU se restringe à instrução 0. Todas as demais instruções ou precisarão do valor imediato na ALU ou não dependerão da saída de ALU.

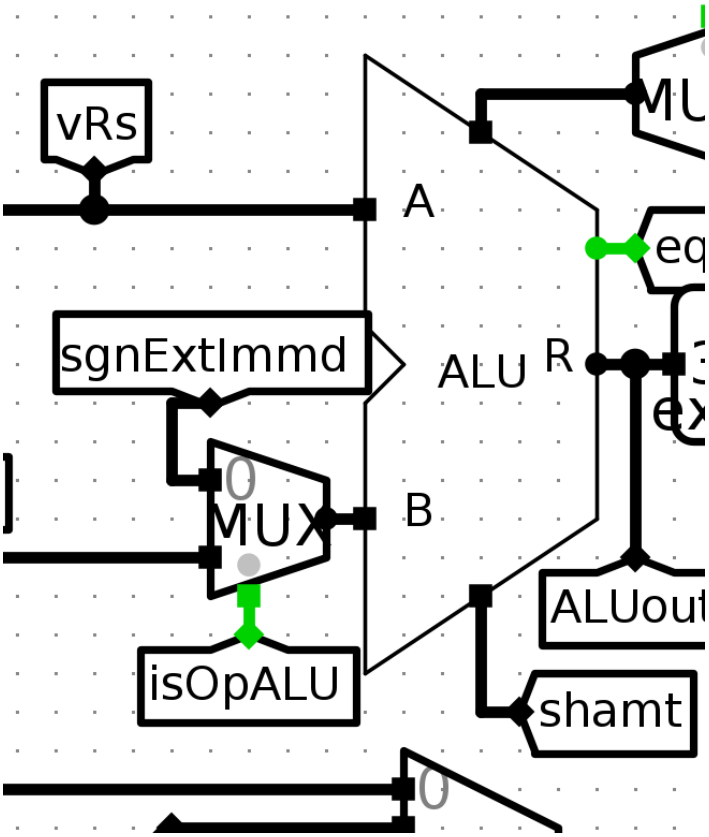


Figura 7: Multiplexador que controla qual valor será enviado à entrada B da ALU. O valor vindo do registrador Rt somente é o selecionado nas instruções de opCode zero.

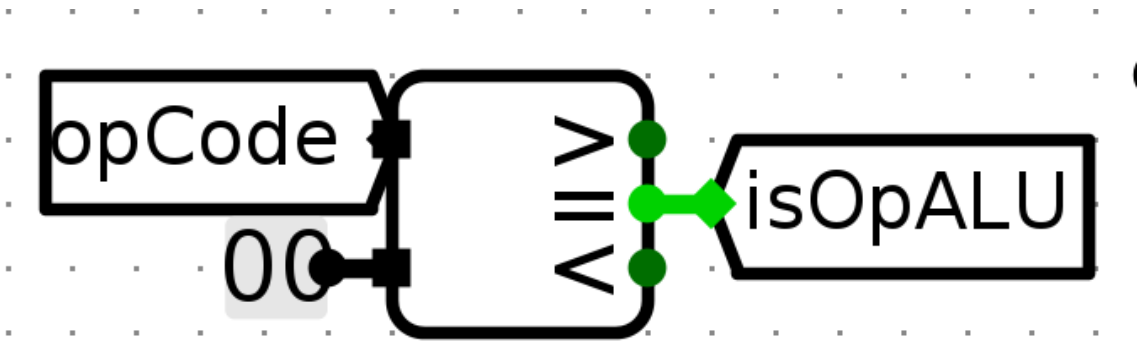


Figura 8: Determinação do sinal controlador 'isOpALU'. Será ativado unicamente durante as intruções de opCode zero.

O restante do funcionamento de ADDI é similar à ADD onde o valor calculado da soma, na saída de ALU, é gravado em um registrador de BREG. Entretanto diferentemente de ADD, que grava o resultado em Rd, a instrução ADDI envia a soma resultante para o registrador de BREG endereçado por Rt, e consequentemente um multiplexador no endereço de escrita de BREG se faz necessário. Como se vê na figura 9, o sinal controlador isOpALU ativado faz com que o registrador Rd seja o alvo da escrita. Por outro lado, no caso de ADDI, em que tal sinal controlador estará desativado, o registrador alvo passa

a ser então Rt, como desejado. O sinal de controle lnk(Link) será usado em outras instruções, como na BLTZAL, e portanto no tempo certo - que já é próximo tópico - receberá os devidos esclarecimentos. Uma vez definido Rt como registrador alvo para escrita, e tendo o valor de salvo em Rs somado ao imediato na entrada de BREG, basta ser ativo o sinal de escrita em BREG para finalizarmos o trajeto da instrução.

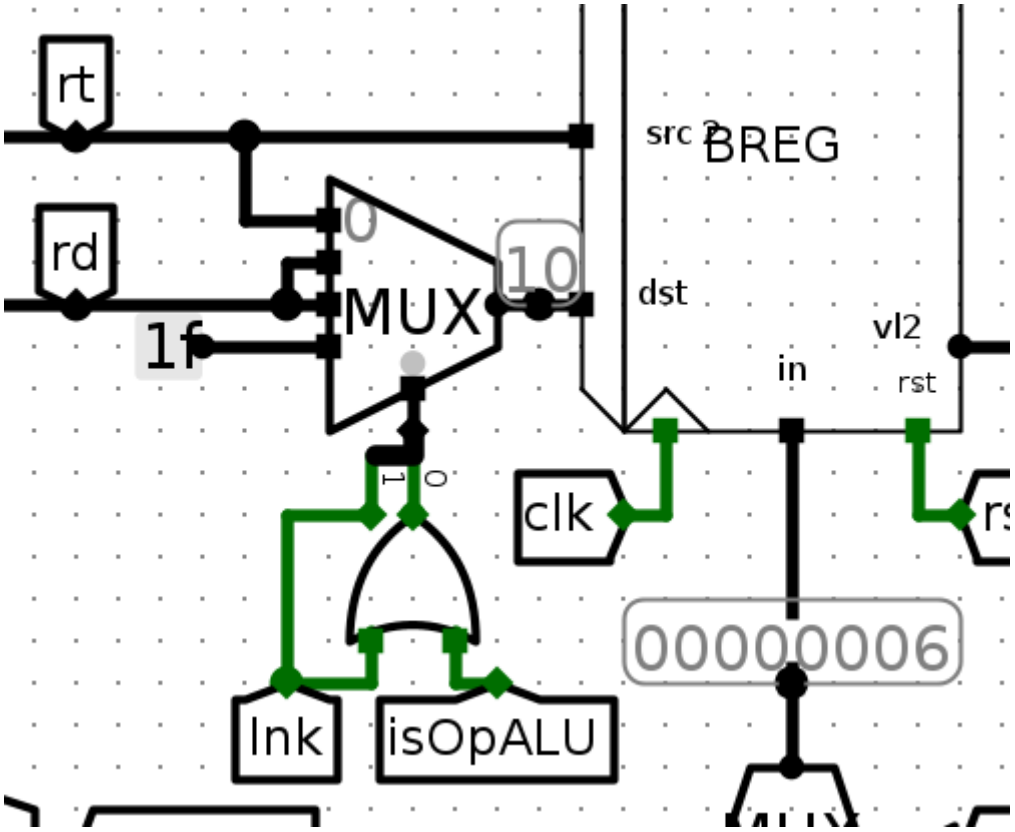


Figura 9: Multiplexação do registrador alvo da escrita em BREG feita pelos sinais 'isOpALU' e 'lnk'.

➢ **BLTZAL - Branch Less Than Zero And Link**

Pertencente ao clã dos Branchs Especiais, a instrução BLTZAL necessita de um segundo identificador, uma vez que essas instruções possuem o mesmo opCode 000001. O campo que outrora era destinado à determinação de Rt, agora funciona como uma forma de distinguir as operações de branch's especiais umas das outras, uma espécie de sub-opCode, sendo a insígnia de BLTZAL o valor 10000.

Quando configurada, a instrução de BLTZAL acessa o valor salvo no registrador apontado por Rs, e esse valor é enviado a um módulo interno da UC chamado de Branch Analyzer(figura 10).

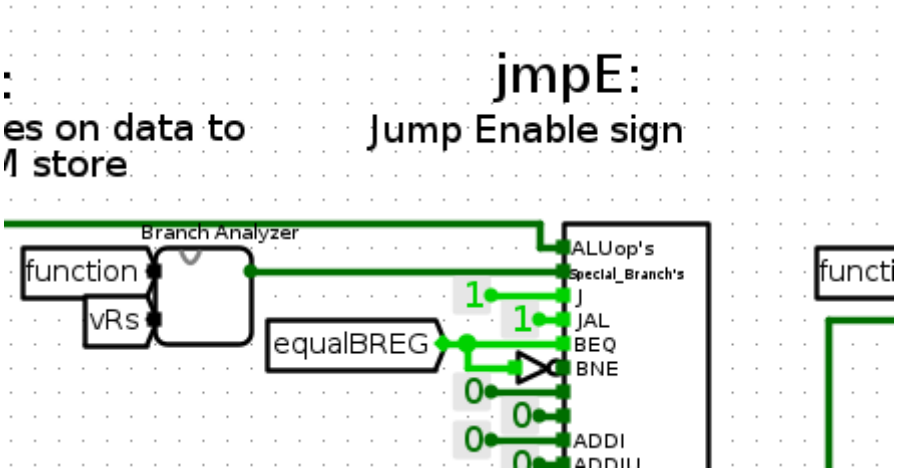


Figura 10: Módulo Branch Analyzer localizado dentro da Unidade de Controle(UC), usado para determinar o valor de jmpE nas instruções de Branch's Especiais. O módulo julga e sentencia a ocorrência do salto a depender do valor salvo em Rs o da do valor Rt, no circuito traduzido para a label 'function'.

Em conjunto com o valor Rt, aí transliterado para a label 'function', esse módulo decidirá se ocorrerá o salto ou não. Cada branch especial tem seu critérios próprios de determinação da ocorrência do salto, e no caso de BLTZAL, o salto - e linkagem - ocorrerá quando o valor salvo em Rs(vRs) for menor que zero. Já vimos anteriormente o mecanismo de funcionamento do PC na figura 3, e podemos aferir que o branch analyzer emitirá um sinal High em sua saída quando perceber que o salto é necessário, fazendo com que jmpE seja ativo. Simultaneamente, caso o salto seja validado, um sinal de controle chamado lnk(Link) é ativado pela UC. No tópico anterior, na figura 9, podemos verificar que o sinal lnk causará a multiplexação do valor 0x1f para o endereço de escrita de BREG. Em notação decimal equivale a definir como alvo de escrita o registrador 31, que é também chamado de Return Adress($ra). A ideia de sua concepção é servir como um armazenador do endereço de retorno após a chamada e fim de um procedimento. Todas as instruções que realizam o procedimento de linkagem salvam o valor do PC(já incrementado) no registrador $ra. A figura 11 mostra que existe um multiplexador controlado pelo sinal lnk, que é capaz de substituir o dado sendo enviado ao BREG pelo valor do PC incrementado.

A ideia da instrução BLTZAL e demais instruções de salto com linkagem é salvar o próximo endereço de instrução que seria executado caso não houvesse ocorrido o salto. Portanto ela e suas similares são usadas na chamada de procedimentos. A instrução JR, já comentada, é então usado de maneira complementar para retornar do procedimento, carregando no PC o valor salvo no registrador 31($ra).

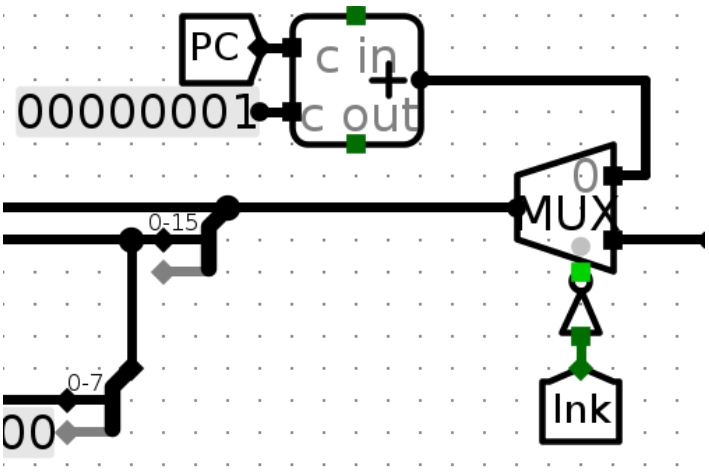


Figura 11: Multiplexador capaz de substituir o atual valor sendo enviado para escrita em BREG pelo valor do PC incrementado.

MULT - Multiply

A instrução de multiplicação foi implementada de forma que o comando MULT aciona o mecanismo de multiplicação da ALU, parametrizado com os argumentos fornecidos na instrução, mas o produto resultante desta multiplicação ficará salvo em registradores especiais, podendo ser resgatado através de outras instruções que serão comentadas. A figura 12 mostra uma seção interna da ALU(Arithimetic and Logic Unit). Quando a instrução de MULT é executada, o sinal registerInputs é ativado e os valores de entrada na ALU são salvos nos registradores especiais rA e rB.

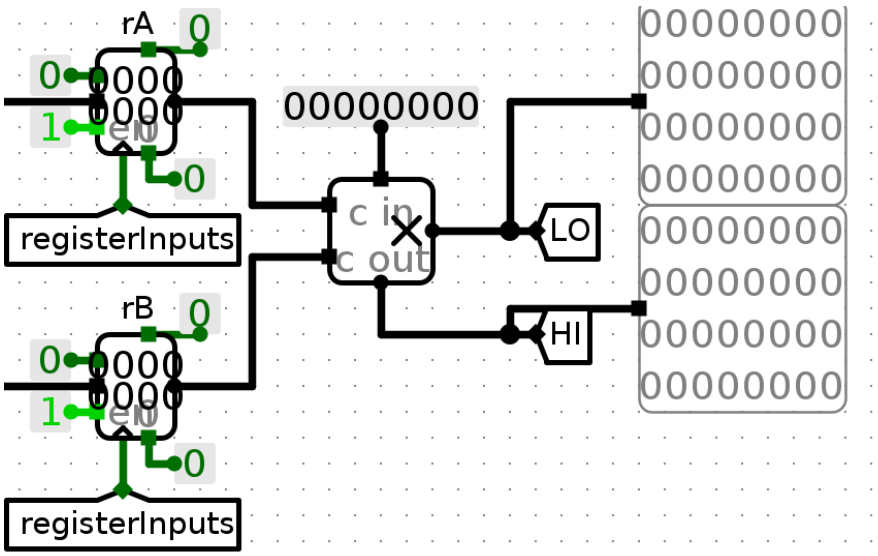


Figura 12: Corte interno da ALU onde é possível ver os registradores especiais rA e rB. Os comandos MFLO e MFHI transferem o valor de um destes registradores, respectivamente, para um registrador comum.

A partir do momento em que os valores A e B de 32 bits, no caso da operação de MULT proveniente do registradores Rs e Rt, são salvos, o multiplicador inicia seu trabalho, que quando concluído resultará em enviar os 4 Bytes menos significativos do produto para o barramento LO e os 4 Bytes mais significativos para o barramento HI(o produto é de 8 Bytes ou 64 bits). Em um outro momento o programador poderá utilizar as instruções MFLO - Move From LOw e MFHI - Move From HIgh, que conecta os barramentos LO e HI na entrada para escrita de BREG, ou seja transferir o produto resultante para registradores simples. O quão em seguida o programador pode chamar MFLO e MFHI após a execução da multiplicação dependerá da implementação física desta implementação lógica. Essa espera mínima é condicionada pelo tempo que esse multiplicador levará para produzir o resultado final estável nas saídas LO e HI.

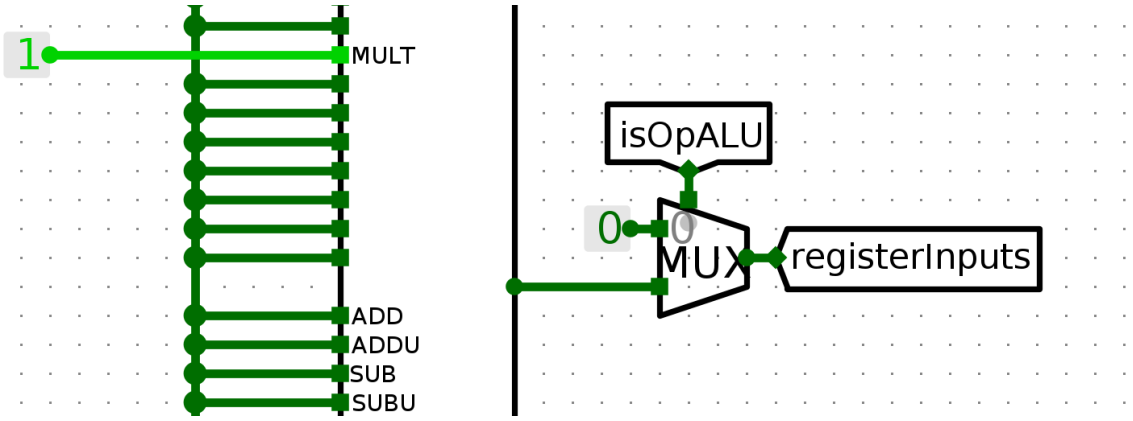


Figura 13: Detalhe da ALU exibindo o mecanismo de determinação do sinal de registerInputs. Numa expansão futura outras instruções poderão os registradores especiais rA e rB, como a intrução de divisão DIV(ainda não implementada).

Multiciclo

Fortemente baseada na implementação monociclo, com certas refatorações, algumas otimizações e o upgrade da Unidade de Controle, a versão multiciclo pode ser vista na figura 14. Em termos de refatoração o uso de labels foi priorizado para os sinais de controle, deixando por outro lado as ligações dos barramentos de dados explicitadas com fios, a fim de melhorar a legibilidade do circuito. Além disso, alguns componentes receberam símbolos melhorados, como os registradores comuns e a memória. Ainda em termo de refatoração a circuitaria que realizava a decodificação da palavra de instrução foi transferida para o interior do circuito do Instruction Register(IR), deixando a organização visualmente mais limpa. Já em termos estruturais, uma diferença gritante é o uso de uma memória única para dados e instruções.

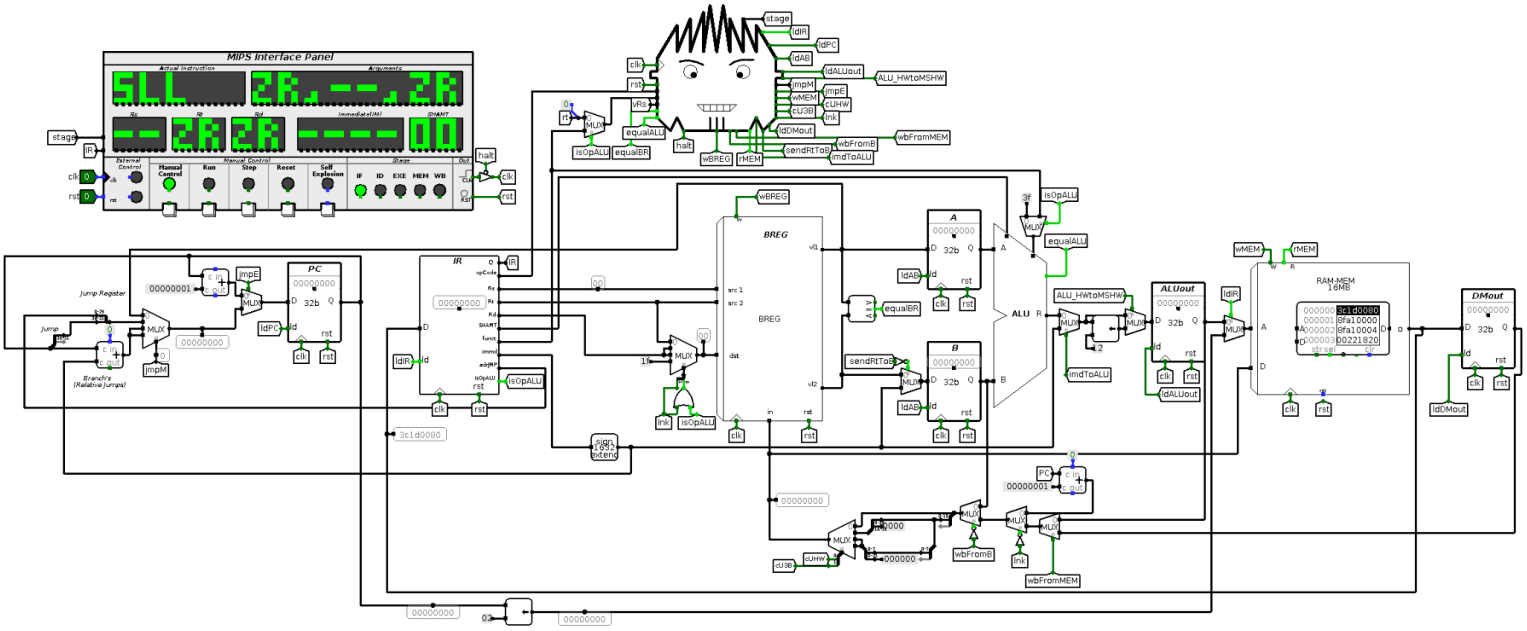


Figura 14: Implementação da organização MIPS multicycle. Entre diversas outras modificações podemos ver que o painel agora indica em qual estágio a execução da instrução se encontra.

A distinção fundamental entre a versão monociclo é que agora cada instrução utiliza vários ciclos de relógio para sua execução. Baseada em uma máquina de estados, a execução das instruções leva o sistema a cambiar entre estados diferentes que vão determinar as saídas da Unidade de Controle. Basicamente as instruções vão ser executadas em até cinco estágios sendo eles:

- Instruction Fetch (IF) :A busca de qual será a próxima instrução, busca essa que é feita na memória única do computador, apontada pelo PC.
- Instruction Decode (ID): Decodificação da instrução, encaminhando os valores adequados para os registradores A e B,

de entrada da ALU.

- Execution (EXE): Execução do cálculo pela ALU e salvamento no registrador ALUout ou realização de jump.
- Memory Operation(MEM): Escrita ou leitura de um dado na/da memória única.
- Write Back(WB): Escrita em BREG.

Na figura 15 é possível constatar que optou-se por utilizar uma série de multiplexadores na determinação do próximo estado a ser transicionado, a depender do estado atual e da instrução sendo executada. Fica registrado também que como a implementação da máquina multiciclo foi uma melhoria da monociclo, a emissão de alguns sinais de controle que já existiam na máquina monociclo, na versão multiciclo foi simplesmente adaptada para levar em conta o estado atual(Figura 16). Como exemplo o sinal rMEM(Read Memory) sempre é ativo no estágio de busca da instrução(IF) mas também é ativado no estágio de MEM quando, e somente quando, a instrução atual determina- o opCode serve de seletor do mux. O motivo da alternativa escolhida em contraste com uma máquina de estados tipo Moore, com as saídas determinadas pelo estado atual, foi em primeiro lugar a facilitação da adaptação do modelo monociclo para o multiciclo, também conhecida como pura evitação da fadiga, sinto confessar. Por outro lado, o fato de se ter apenas 5 estados, talvez tenha trago alguma simplificação no entendimento e depuração da máquina e na progressão de estados, em comparação com uma segunda versão que possuísse as dezenas de estados necessárias para uma máquina tipo Moore com todas as instruções implementadas. Da forma implementada a adição de instruções ou correção do estágio em que algum sinal é emitido podem ser feitas nas entradas dos multiplexadores da máquina de estados, ao invés de se ter de criar estados novos.

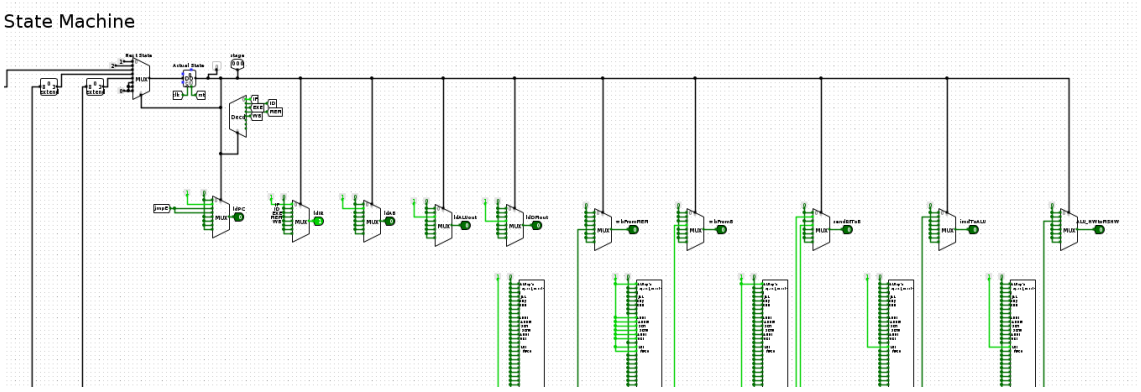


Figura 15: Máquina de estados da unidade de controle.

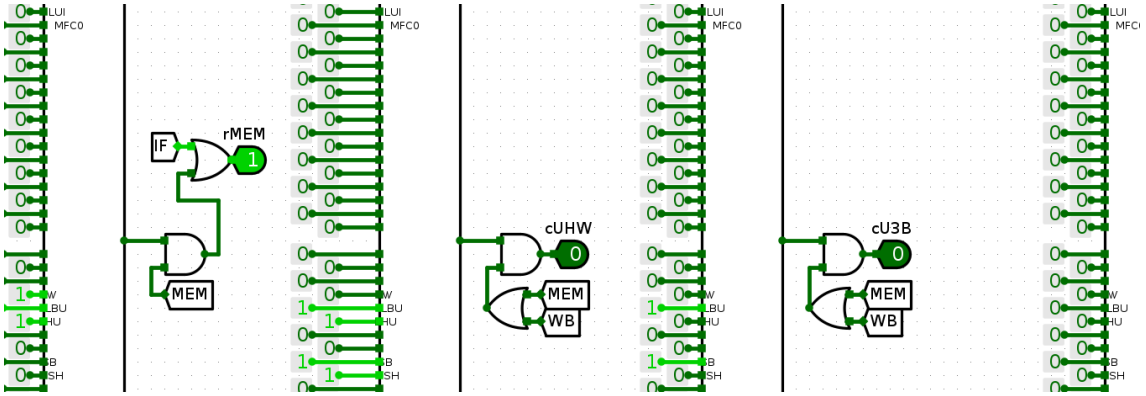


Figura 16: Adaptação da geração de sinais da máquina monociclo para a multiciclo.

A implementação das instruções básicas podem ser consultadas no projeto disponibilizado, sendo que as próximas linhas serão destinadas a uma análise sucinta da implementação das cinco instruções destacadas.

➢ **JR**

Antes de uma descrição dos estágios desta instrução, um adendo deve ser feito quanto à diferença na implementação do PC nesta máquina multiciclo. Como se vê na figura 17, nesta versão o PC foi implementado com a utilização de um registrador comum. Isso implica que toda alteração do valor de PC requer um sinal de load(ldPC).

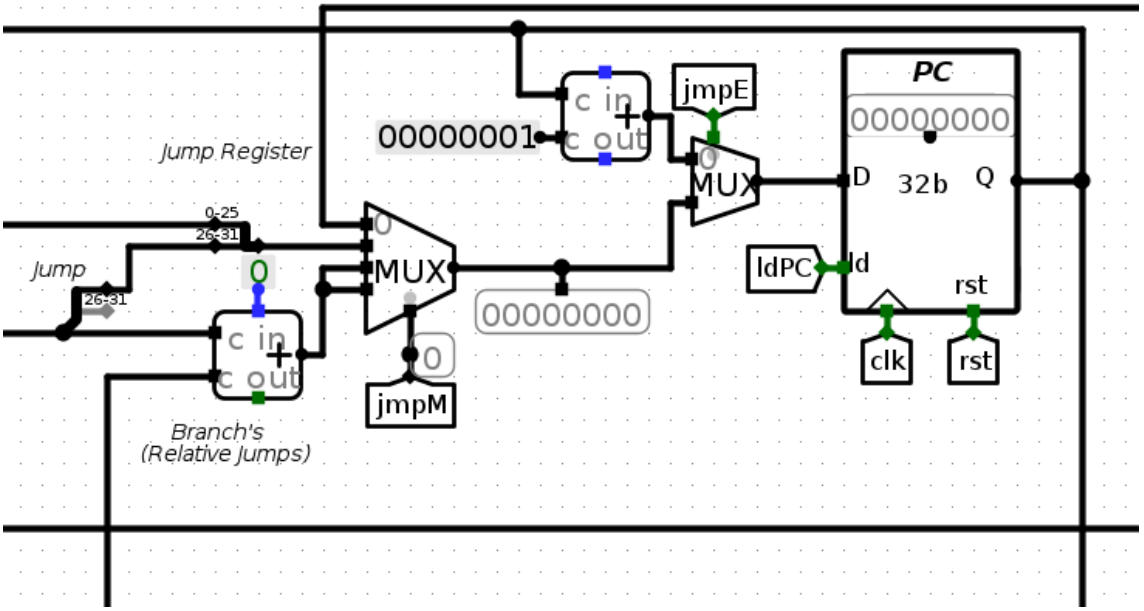


Figura 17: Implementação do PC com registrador comum e multiplexadores que determinam se o valor de entrada será o valor atual incrementado ou um outro endereço destino, no caso das instruções de salto.

Dito isto podemos verificar na instrução de JR, em algum momento, sendo o ‘momento’ entendido como um dos estágios da instrução, o valor de vindo do registrado escolhido será salvo no PC, e para isso o valor do sinal jmpM nesse estágio deverá ser 00 e os sinais ldPC e jmpE deverão ser ativados.

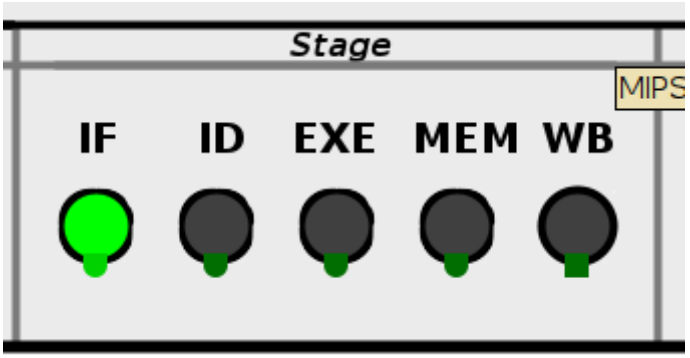


Figura 18: Indicador de estágio atual localizado no painel.

Voltando então o disco para o início, tudo começa com o instruction Fetch. Um valor da memória, apontado por PC é transferido para o Instruction Register. Na figura 19 vemos que os sinais de rMEM e ldIR, ambos ligados à memória, são ativados nesse estágio, assim como o de ldIR, controlador de IR. Esse é um estágio que ocorrerá exatamente da mesma forma no início da execução de cada instrução e portanto não será mais comentado.

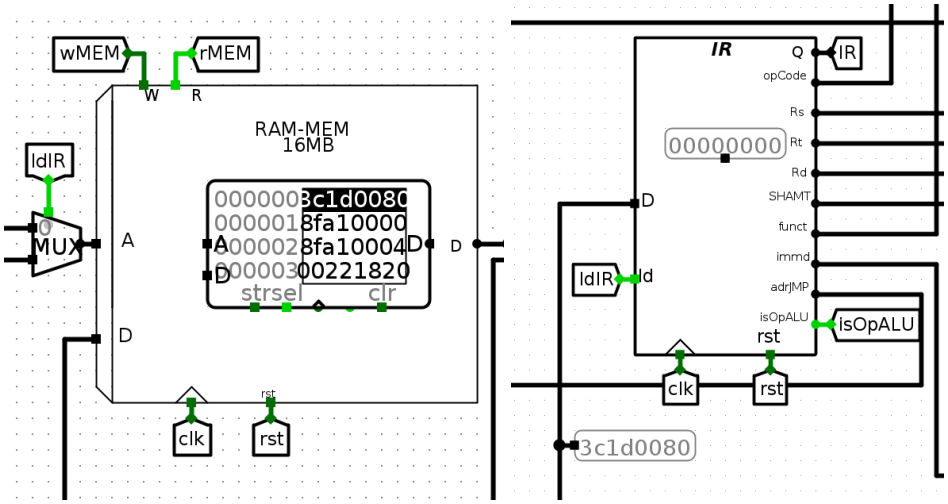


Figura 19: Memória, IR e seus sinais no momento do fetch.

No estágio seguinte temos a instrução já definida no IR, implicando que o endereço de leitura 1 de BREG já está definido como Rs e portanto a saída 1 carregará seu resultado que será enviado diretamente para a entrada de PC. Esperamos mais um ciclo de execução, apenas para garantir que o circuito já esteja estabilizado - na implementação física - e então no próximo estágio, o de execução(EXE) o os sinais de jump e load IR são ativados finalizando a instrução.

➢ **LBU**

A instrução LBU simplesmente foi adaptada ao do formato mono estágio para o formato multi estágios. Após IF, no estágio de ID, o valor do registrador A é carregado com valor de BREG[Rs] e B recebe o imediato com sinal extendido, e portanto ativa os sinais ldAB mas mantém desativado sendRtToB (figura20). No próximo estágio, EXE, a ALU soma esses dois valores e as envia para carga de ALUout, com a anuência do sinal ldALUout. No estágio de MEM o valor salvo em ALUout é usado como endereço para leitura de um dado de 32 bits que é enviado para a entrada de BREG antes passando pelo mesmo filtro visto na versão monociclo. No estágio final, o de WB o sinal de escrita em BREG (wBREG) é ativo findando a instrução.

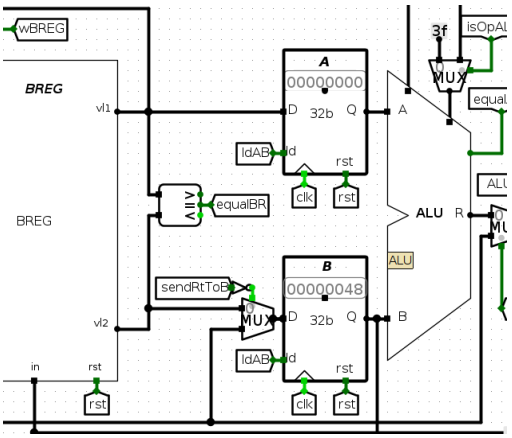


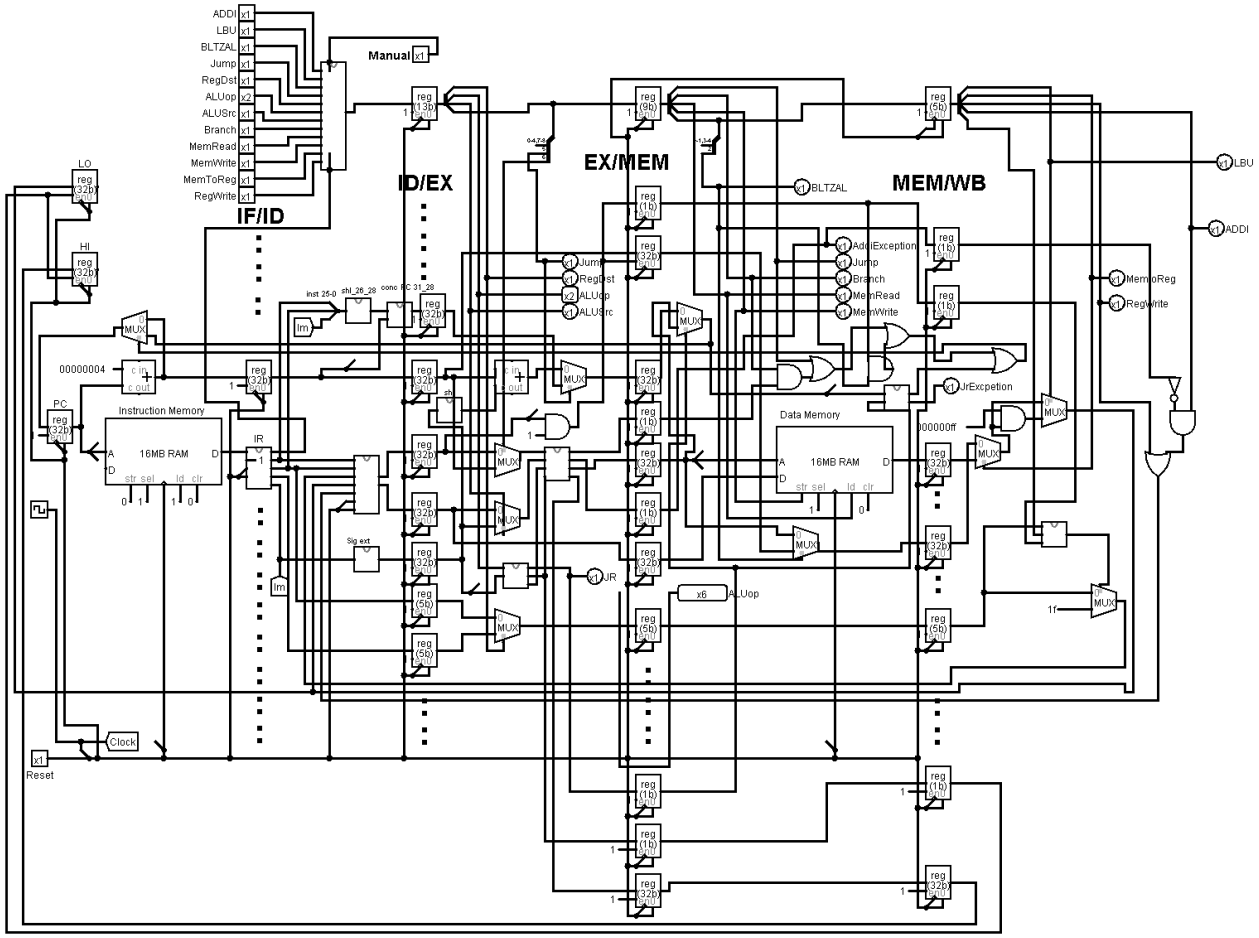
Figura 20: Registradores A e B, que armazenam argumentos de entrada da ALU.

➢ **ADDI, BLTZAL e MULT**

As três instruções destacadas finais usam os mesmo mecanismos que existiam na versão monociclo tendo sofrido a divisão em multi estágios, de forma similar às adaptações sofridas por JR e LBU.

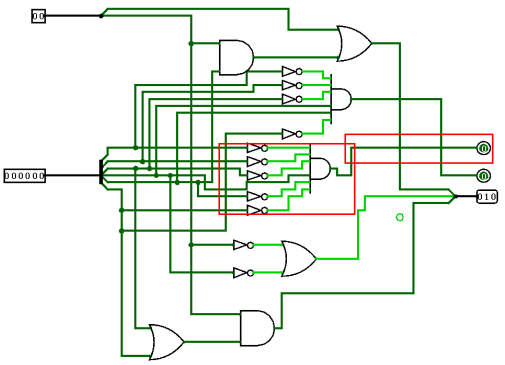
Pipeline

Bloco Operativo Completo:

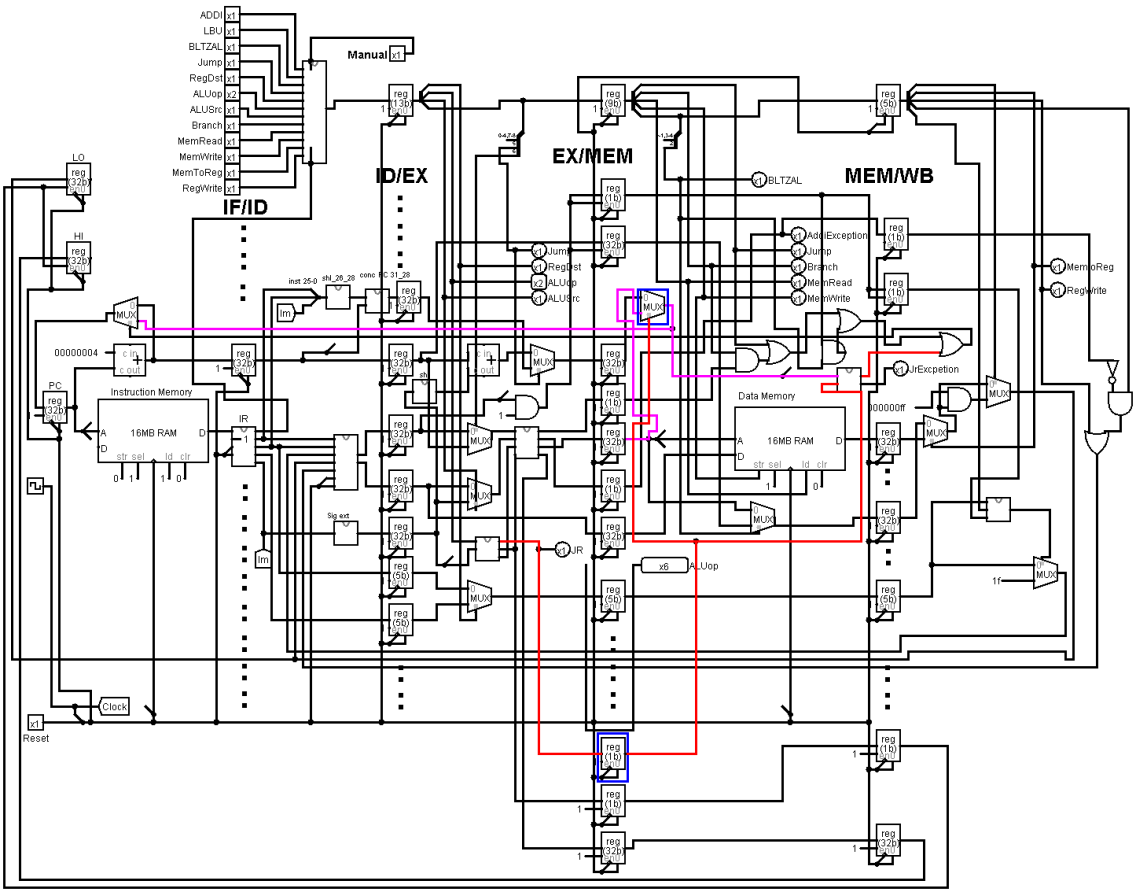


Instrução JR

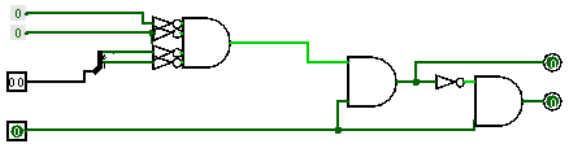
Para realizar a instrução JR, a ALU foi modificada para conseguir gerar um bit de controle que indica que a instrução do tipo R é a instrução JR a partir dos bits do campo funct. A escolha de adicionar uma saída a ALU Control, e não estender a os bit da saída já existente, foi tomada pela questão da facilidade. O bit é salvo em um registrador de um bit e, posteriormente, esse valor é utilizado para selecionar qual o próximo endereço do pc e verificar se efetivamente deve realizar o salto, com base em um comparador que verifica se os últimos bits do endereço são zeros



(ALU modificada)



(Instrução JR)



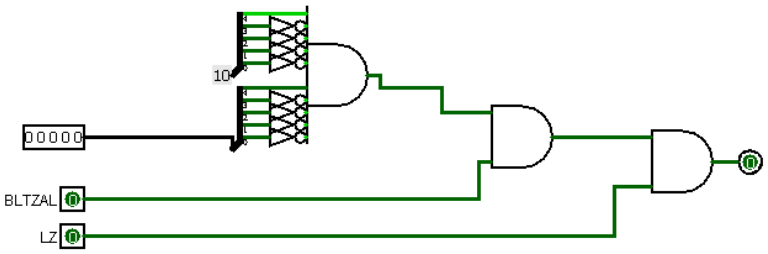
(Circuito que verifica se deve efetivamente saltar)

Bits de Controle:

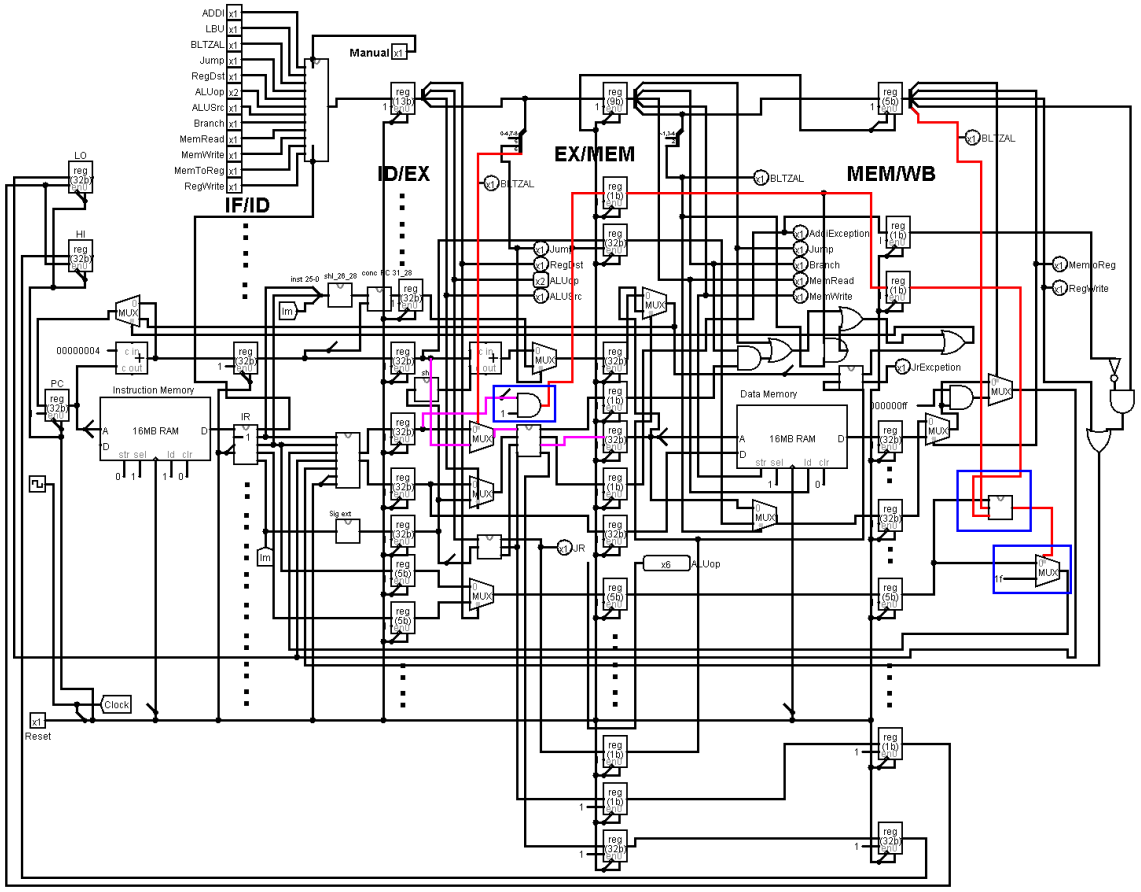
Addi	Lbu	Bltzal	Jump	RegDst	AluOp	ALUSrc	Branch	MemRead	MemWrite	MemToReg	RegWrite
0	0	0	0	1	10	0	0	0	0	0	1

Instrução BLTZAL

Foi adicionado um novo bit de controle chamado BLTZAL. Ele é passado pelos registradores do pipeline e é utilizado em um circuito que verifica se deve efetivamente saltar. Além disso, é feita a verificação se o bit de sinal do valor contido no registrador rs é 1 e essa verificação é passada pelo pipeline. No final da execução da instrução foi adicionado um mux que seleciona o registrador 31 para salvar o endereço de retorno.



(Circuito que verifica se deve salvar o endereço de retorno)

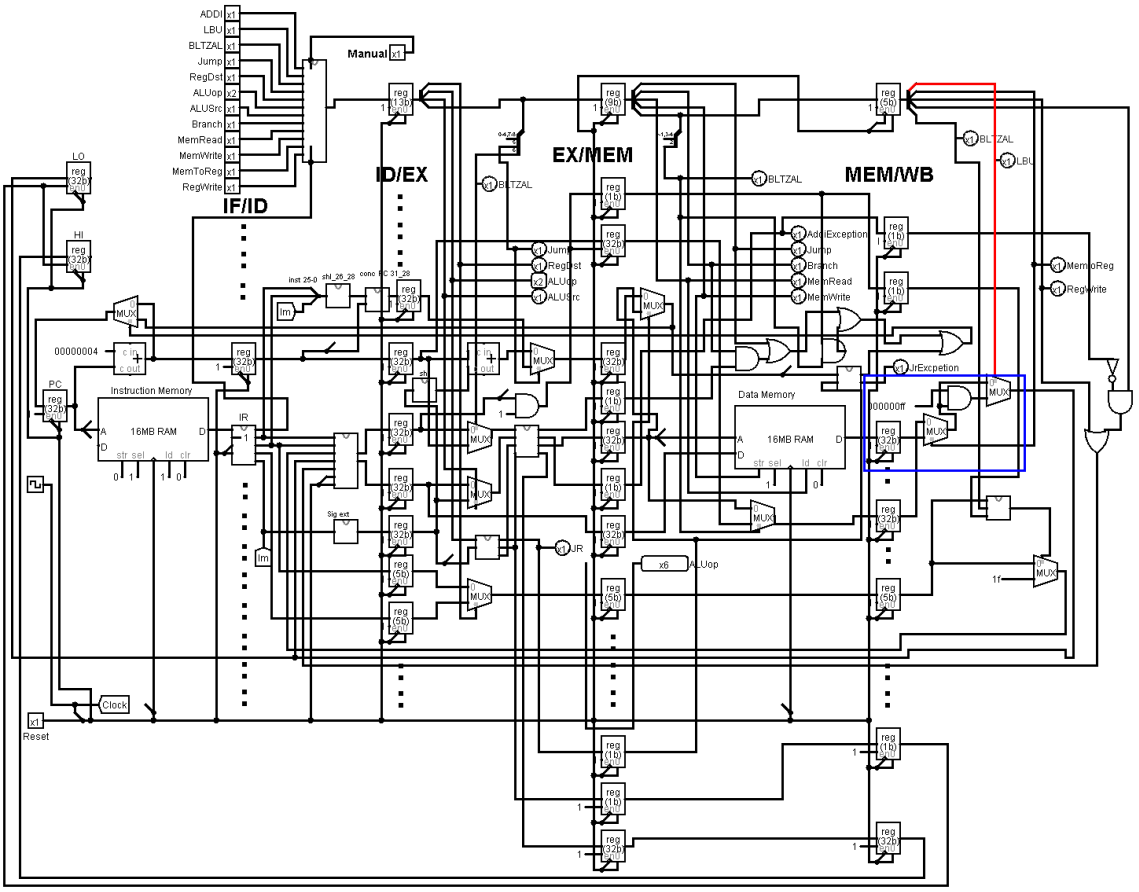


Bits de Controle:

Addi	Lbu	Bltzal	Jump	RegDst	AluOp	ALUSrc	Branch	MemRead	MemWrite	MemToReg	RegWrite
0	0	1	0	1	00	1	0	0	0	0	1

Instrução LBU

Acrescentou-se um bit de controle, chamado LBU e, além disso, foi adicionado uma máscara ao final da saída do mux que seleciona o valor que será gravado no banco de registradores. Essa máscara tem a finalidade de pegar somente o valor do byte do conteúdo da saída, fazendo com que os demais bits sejam zero. Dessa forma, torna-se possível salvar somente o byte do endereço da memória, fazendo com que os demais bits sejam zero.

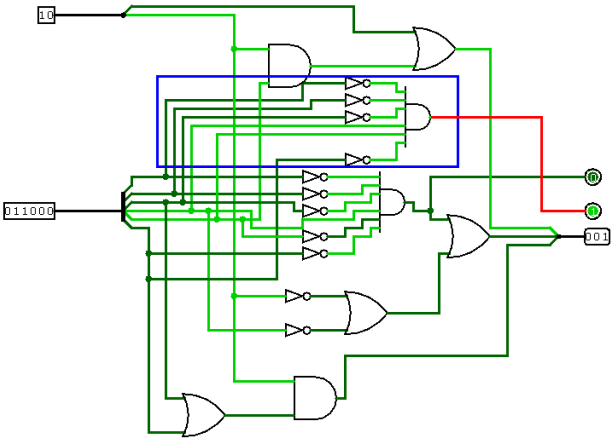


Bits de Controle:

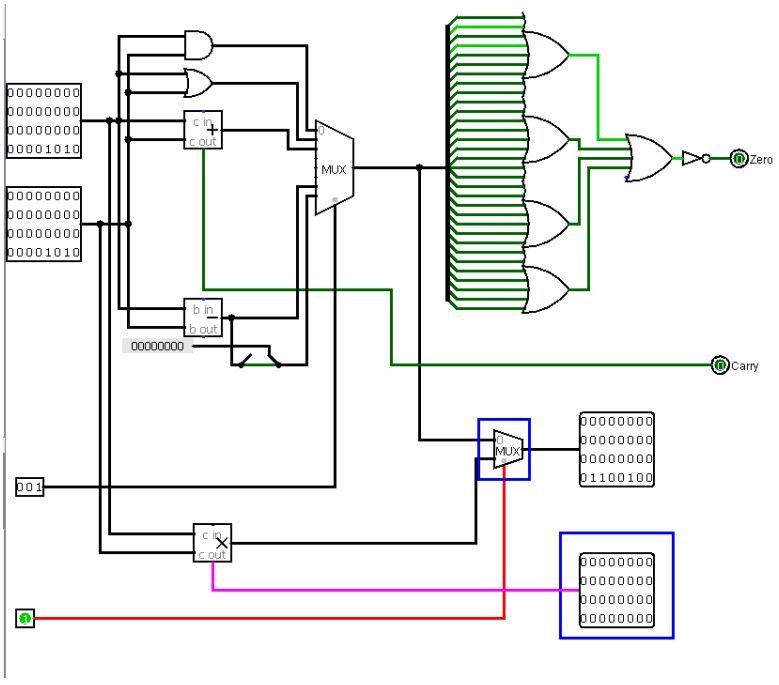
Addi	Lbu	Bltzal	Jump	RegDst	AluOp	ALUSrc	Branch	MemRead	MemWrite	MemToReg	RegWrite
0	1	0	0	0	10	1	0	1	0	1	1

Instrução MULT

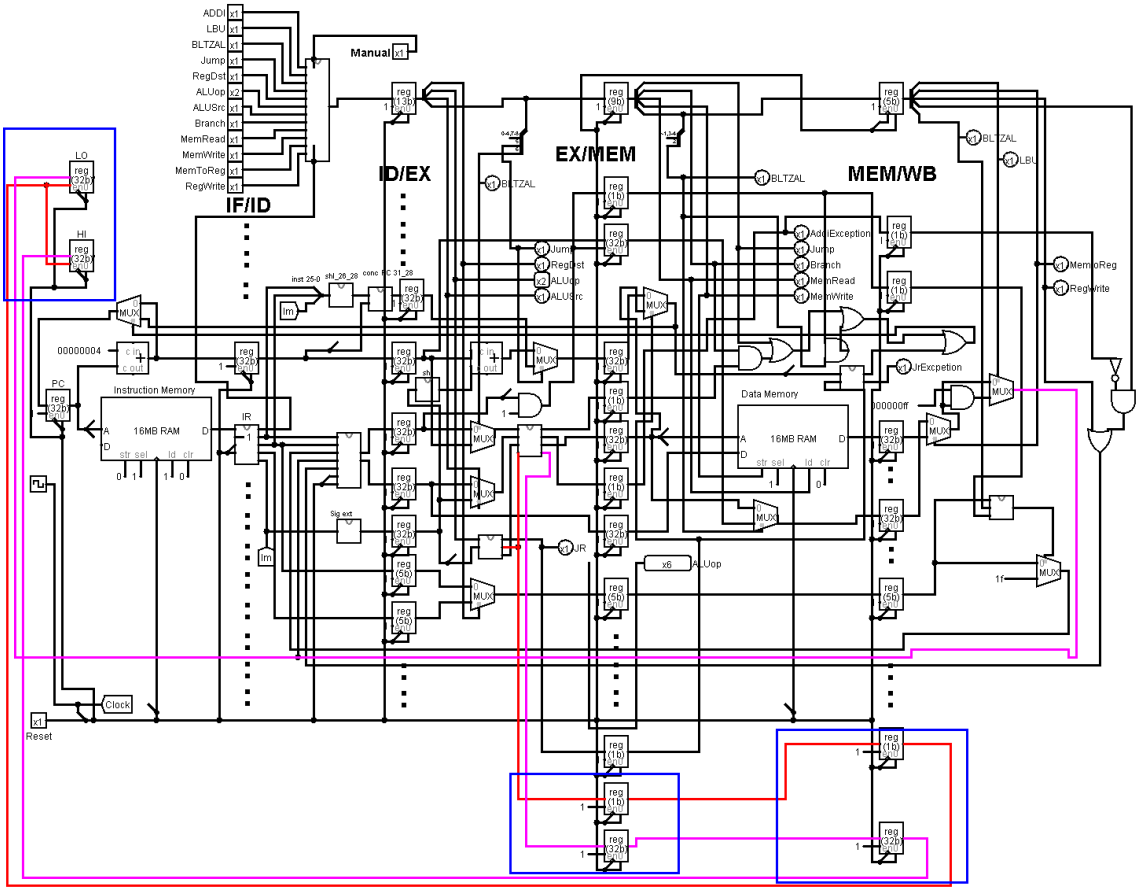
A Alu Control acabou sendo modificada para ser possível identificar uma operação do tipo Mult a partir do campo Funct. Por questão de escolha, novamente, a saída da Alu Control não foi estendida, mas sim acrescentado um bit de saída que indica a operação.



Além disso, foi adicionado uma entrada na Alu, para receber o sinal da operação de Mult, e adicionado uma saída para passar o valor mais dos bits mais significativos do resultado da multiplicação.



Foram adicionados dois registradores de propósito geral, LO e HI, e o pipeline foi estendido para receber um bit que indica ser uma operação do tipo Mult e o valor da parte mais significativa do resultado da multiplicação.

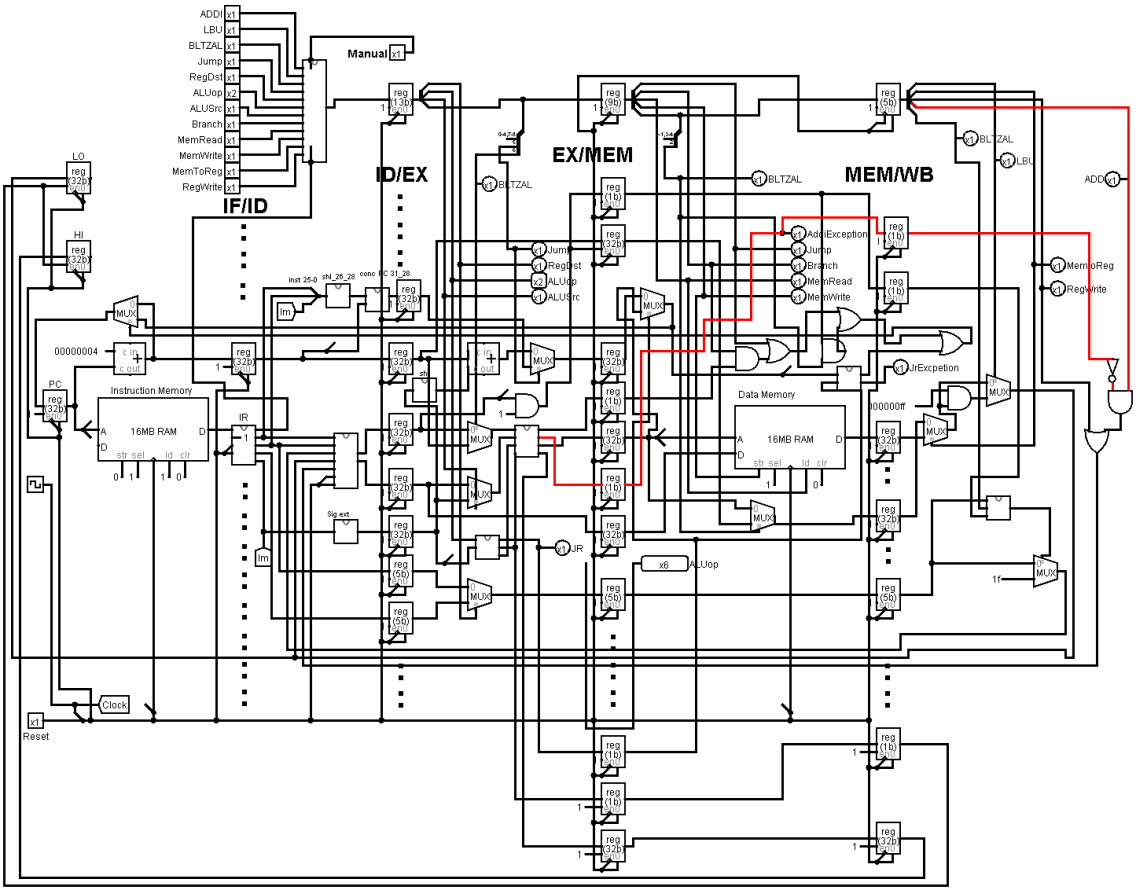


Bits de Controle:

Addi	Lbu	Bltzal	Jump	RegDst	AluOp	ALUSrc	Branch	MemRead	MemWrite	MemToReg	RegWrite
0	0	0	0	1	10	0	0	0	0	0	1

Instrução ADDI

Foi adicionado um bit de controle chamado LBU e uma saída na Alu que indica se ocorreu estouro na soma. Esse valor é passado pelo pipeline, que foi estendido, e é combinado, utilizando uma porta AND, com o valor do bit de controle LBU utilizado para indicar que o valor deve ser salvo no banco de registradores. Essa ação não ocorre quando há um estouro de representação na soma, e o registrador de destino permanece inalterado.

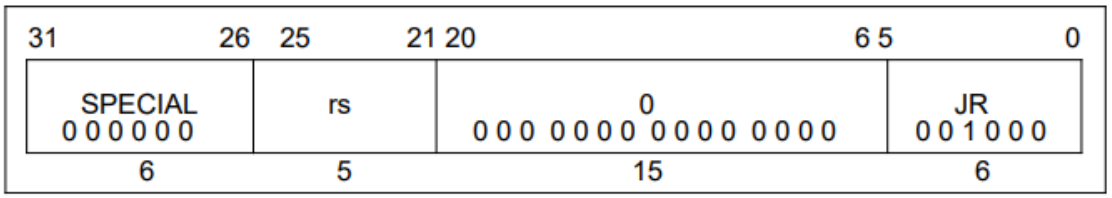


Bits de Controle:

Addi	Lbu	Bltzal	Jump	RegDst	AluOp	ALUSrc	Branch	MemRead	MemWrite	MemToReg	RegWrite
1	0	0	0	0	00	1	0	0	0	0	0

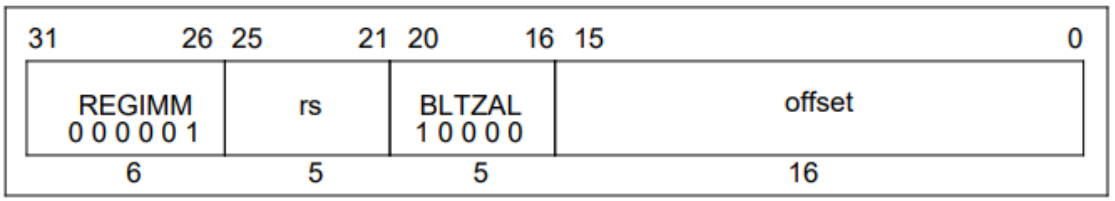
Apêndice: Conjunto de instruções destacadas para análise

JR **Jump Register** **JR**



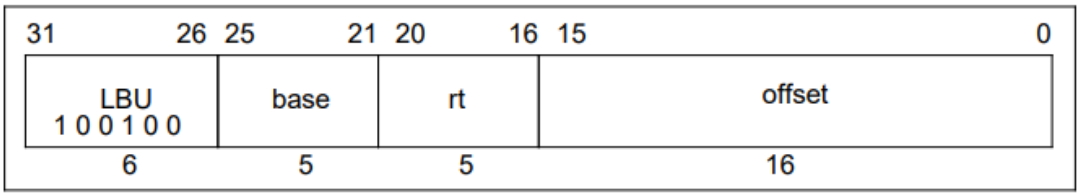
O programa salta incondicionalmente para o valor contido no registrador rs com o delay de uma instrução. Pelo fato das instruções serem orientadas a palavra, os últimos dois bits do valor do salto devem ser zeros. Caso não sejam, ocorre uma exceção depois que a instrução for buscada.

BLTZAL **Branch On Less Than Zero And Link** **BLTZAL**



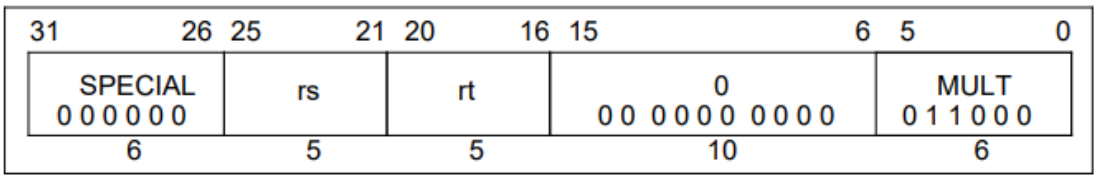
Executa um salto condicional baseando-se no valor contido no registrador rs é menor do que zero. Caso seja, executa um salto para o valor contido no offset, que é shiftado dois bits para a esquerda com o bit do sinal estendido. Além disso, o endereço de retorno, posterior à instrução do salto, é salvo no registrador 31.

LBU — Load Byte Unsigned — LBU



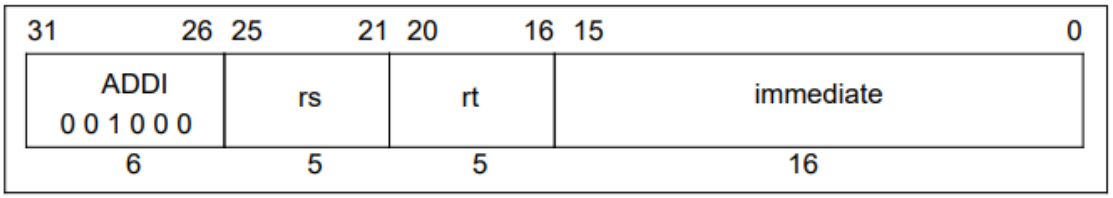
Carrega-se o byte do endereço da memória, indicado pelo endereço calculado a partir do valor contido no registrador base e o valor do offset com o bit de sinal estendido. O conteúdo da memória é estendido com o valor zero e salvo no registrador rt.

MULT — Multiply — MULT



Os conteúdos dos registradores rs e rt são multiplicados. Após a operação ser finalizada, a parte mais significativa do resultado é alocada ao registrador especial HI e a parte menos significativa é alocada ao registrador específico LO.

ADDI — Add Immediate — ADDI



O valor imediato tem o valor de sinal estendido e somado ao registrador rs para formar o resultado. O resultado é alocado ao registrador rt.

Se ocorrer overflow na soma dos valores, o valor do registrador rt não é modificado.

Referências

[1] http://lattes.cnpq.br/8544491643812450 e https://www.inf.ufrgs.br/~carro/

[2] https://www.inf.ufrgs.br/~caco/

[3] http://www.cburch.com/logisim/

[4] http://courses.missouristate.edu/kenvollmar/mars/