Universidade Federal do Rio Grande do Sul
Instituto de Informática
Departamento de Informática Aplicada
INF01113 – Organização de Computadores B - Turma B
Prof. Luigi Carro

Heloísa de Carvalho Rosa Marques - 00334243
Laura Keidann Rodrigues da Silva - 00217870
Matheus de Moraes Costa - 00297121
Pedro Henrique Bouvié Roewer - 00330062

Trabalho 1: MIPS

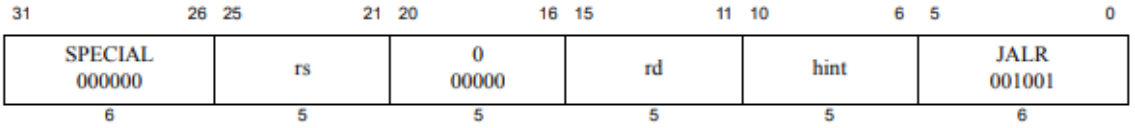
Grupo	Jump	Branch	Load	Arit/Log	Arit/Log com Imediato
1	JALR	BLTZ	LBU	SRA	SLTIU

Instruções: JALR, BLTZ, LBU, SRA e SLTIU

Por estarmos em um quarteto, a SLTIU não foi implementada.

Instruções:

JALR



Jump And Link Register

Formato: JALR rs (rd = 31 implied) MIPS32 JALR rd, rs

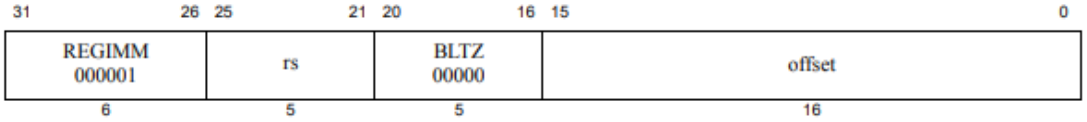
Descrição:

Armazene o endereço de retorno no registrador GPR rd. Esse endereço de retorno é o da segunda instrução que vem depois da instrução de desvio, indicando onde a execução prossegue após a chamada de uma função.

Operação:

```
I:  temp ← GPR[rs]
    GPR[rd] ← PC + 8
I+1:if (Config3_ISA = 0) and (Config1_CA = 0) then
        PC ← temp
    else
        PC ← temp_GPRLEN-1..1 || 0
        ISAMode ← temp_0
    endif
```

BLTZ



Branch on Less Than Zero

Formato:

BLTZ rs, offs

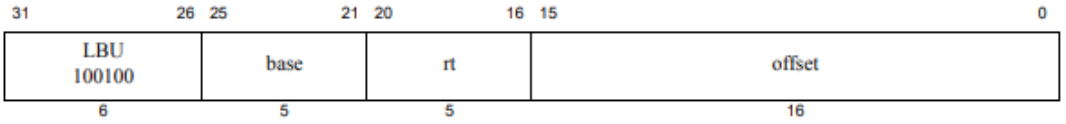
Descrição:

Um deslocamento de 18 bits com sinal (o campo de deslocamento de 16 bits deslocado 2 bits para a esquerda) é adicionado ao endereço da instrução que vem após o desvio (não o próprio desvio), no slot de atraso do desvio, para formar um endereço de destino relativo ao contador de programa (PC). Se o valor contido no registrador GPR rs for menor que zero (bit de sinal igual a 1), o desvio será feito para o endereço de destino efetivo depois que a instrução no slot de atraso for executada.

Operação:

```
I:      target_offset ← sign_extend(offset || 0^2)
        condition ← GPR[rs] < 0^GPRLEN
I+1:    if condition then
            PC ← PC + target_offset
        endif
```

LBU



Load Byte Unsigned

Formato:

LBU rt, offset(base)

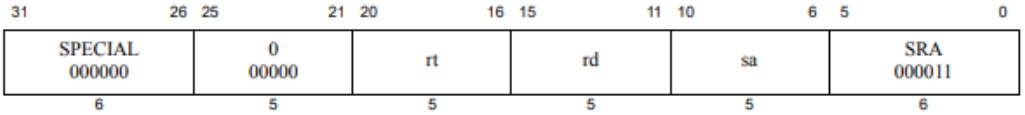
Descrição:

O conteúdo do byte de 8 bits no endereço de memória especificado pelo endereço efetivo é carregado, estendido com zeros, e armazenado no registrador GPR rt. O deslocamento com sinal de 16 bits é somado ao valor do registrador GPR base para criar o endereço efetivo.

Operação:

```
vAddr ← sign_extend(offset) + GPR[base]
(pAddr, CCA) ← AddressTranslation (vAddr, DATA, LOAD)
pAddr ← pAddr_PSIZE-1..2 || (pAddr_1..0 xor ReverseEndian^2)
memword ← LoadMemory (CCA, BYTE, pAddr, vAddr, DATA)
byte ← vAddr_1..0 xor BigEndianCPU^2
GPR[rt] ← zero_extend(memword_7+8*byte..8*byte)
```

SRA



Shift right arithmetic

Formato:

SRA rd, rt, sa

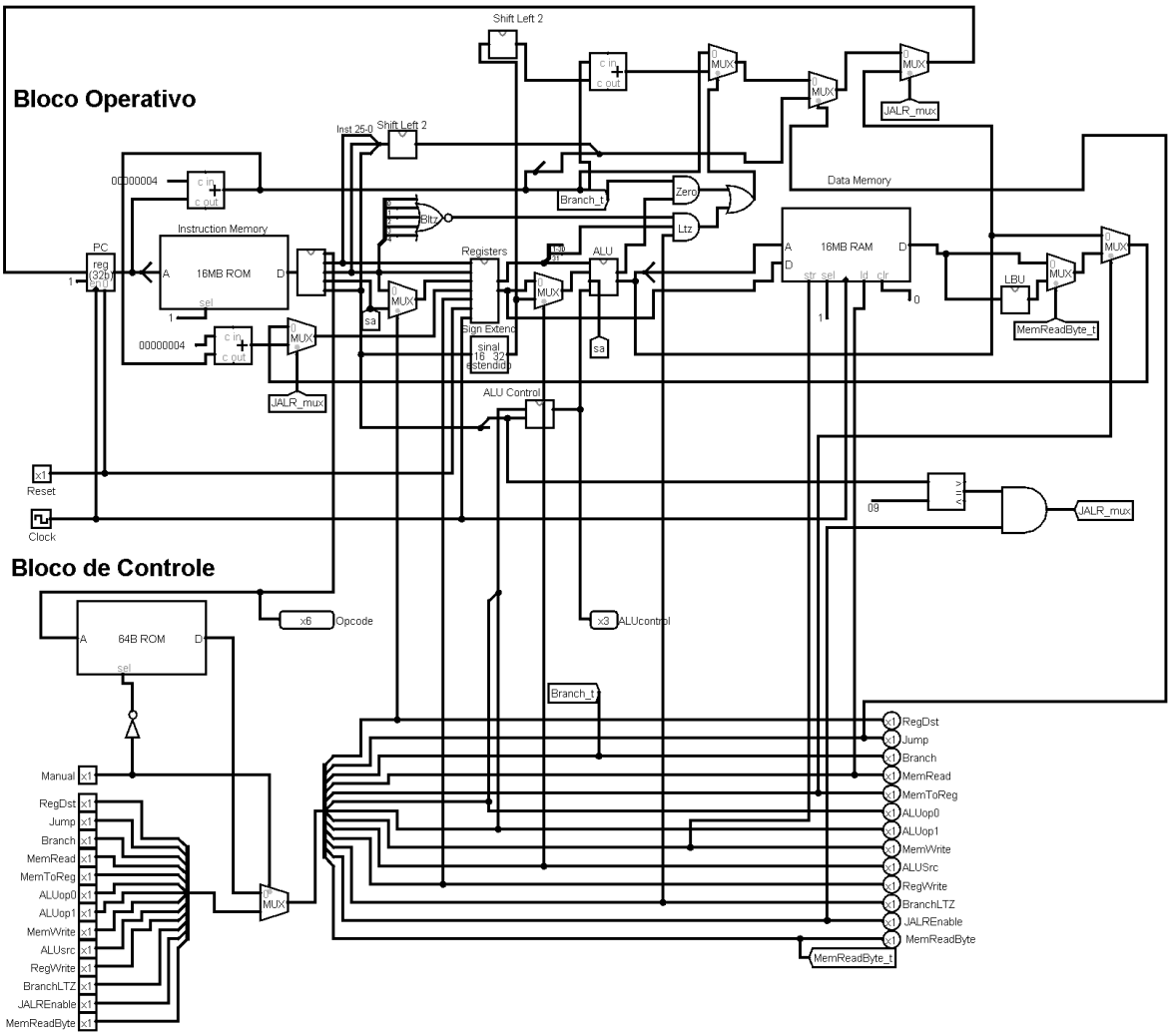
Descrição:

O valor dos 32 bits menos significativos do registrador GPR rt é deslocado para a direita, replicando o bit de sinal (bit 31) nos bits que ficam vazios; o resultado desse deslocamento é armazenado no registrador GPR rd. A quantidade de deslocamento de bits é determinada por sa.

Operação:

```
ss ← sa
temp ← GPR[rt]31)^s || GPR[rt]31..s
GPR[rd] ← temp
```

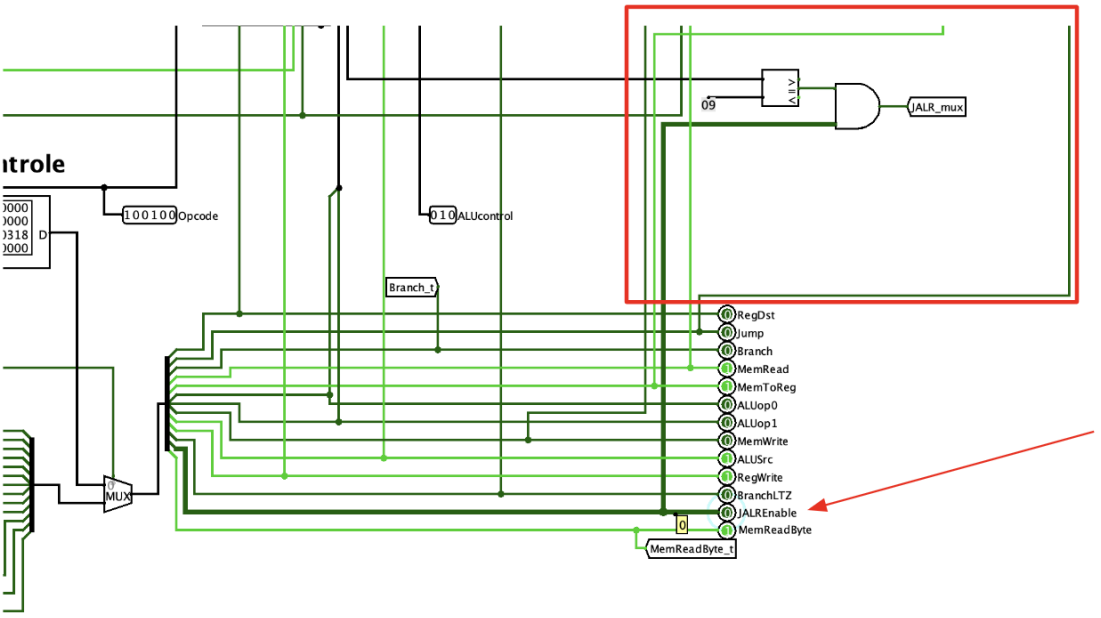
1) Monociclo



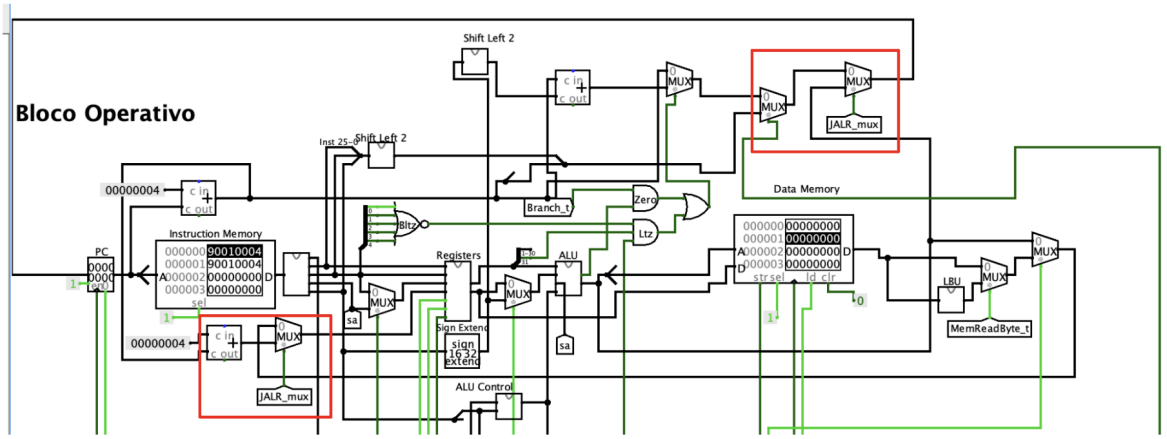
Visão geral

JALR NO MONOCICLO

A flag “JALREnable” foi adicionada ao bloco de controle, e é chamada com o opcode “000000”. Para identificar a instrução, checamos se o campo “funct” lido realmente consta “001001”, e se a flag citada está ativa.



Com esta verificação feita, o sinal JALR_mux é ativo e usado para garantir que o endereço atual de PC incrementado vá para o banco de registradores, e o endereço novo de PC chegue até ele.



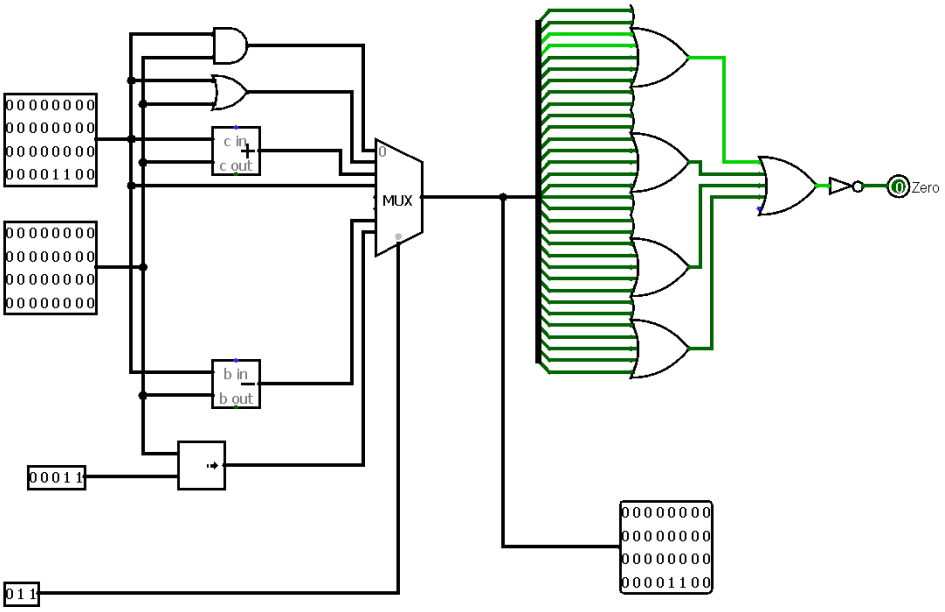
Sinais de controle

“0101001000001” = 0a41

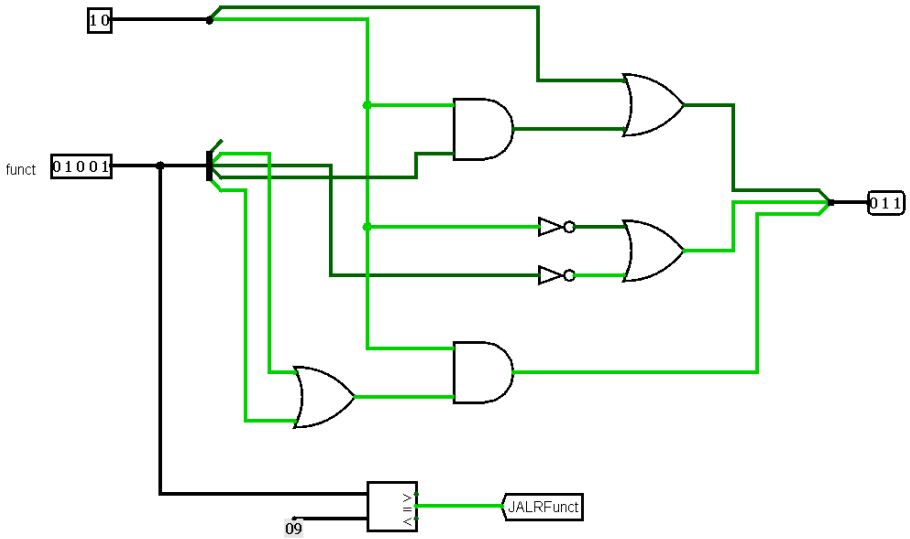
RegDst	Jump	Branch	MemRead	MemToReg	ALUOp
X					01

MemWrite	ALUSrc	RegWrite	BranchLTZ	JALREnable	MemReadByte
		X		X	

Na ULA, foi feito um caminho para que o endereço a ser posto no PC passe.

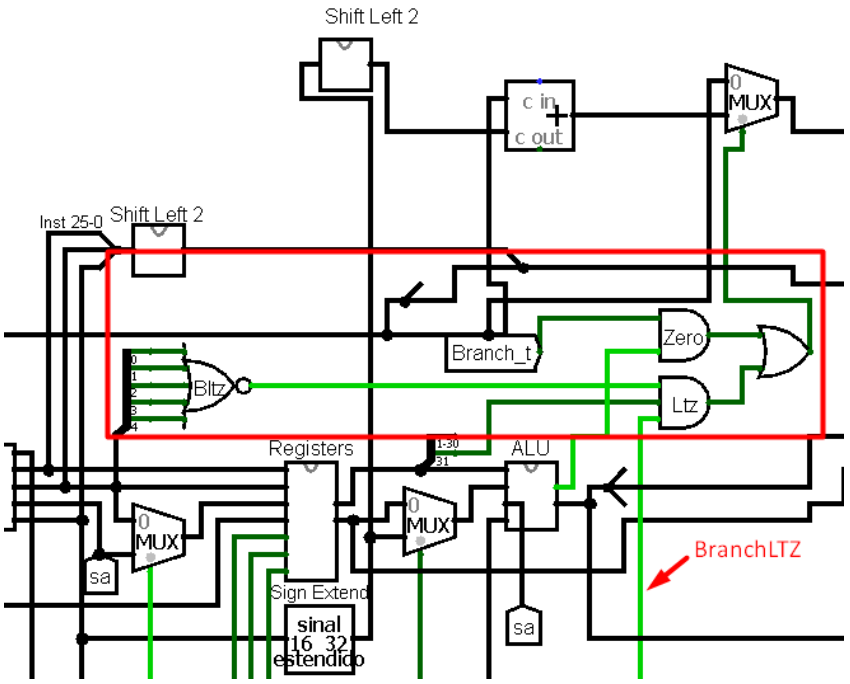


Para seleção correta no mux, a ULA recebe “011” de ALU_Control.



BLTZ NO MONOCICLO

Para BLTZ, foi criado uma flag “BranchLTZ”, e as seguintes adições indicadas na imagem foram feitas:



Observa-se que a porta AND "Ltz" recebe: i. O bit mais significativo do valor armazenado no registrador indicado pela instrução; ii. Os bits 21..16 da instrução, que devem ser zero; iii. O sinal de BranchLTZ; Esse sistema garante que, caso "less than zero" seja verdade, o endereço do jump a ser feito será posto em PC (0 no mux superior direito da imagem acima). Caso contrário, PC recebe simplesmente PC+4.

Sinais de controle

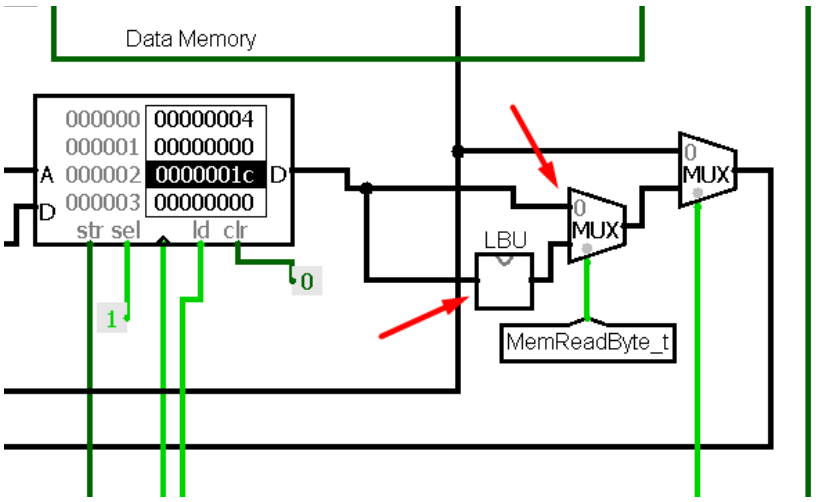
"0010000000001" = 0401

RegDst	Jump	Branch	MemRead	MemToReg	ALUOp
X					00

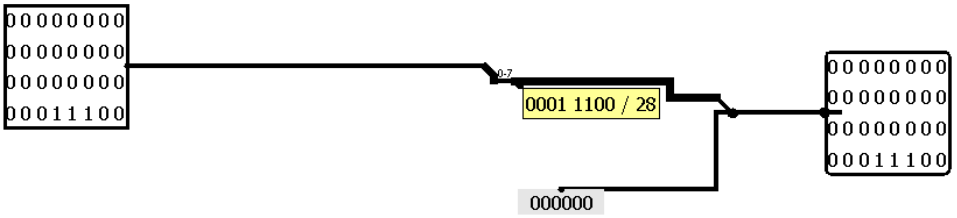
MemWrite	ALUSrc	RegWrite	BranchLTZ	JALREnable	MemReadByte
			X		

LBU NO MONOCICLO

Nesta implementação, na saída da memória de dados, adicionamos um novo mux para passagem do "byte" desejado. O sinal de controle "MemReadByte" foi criado para controle deste mux.

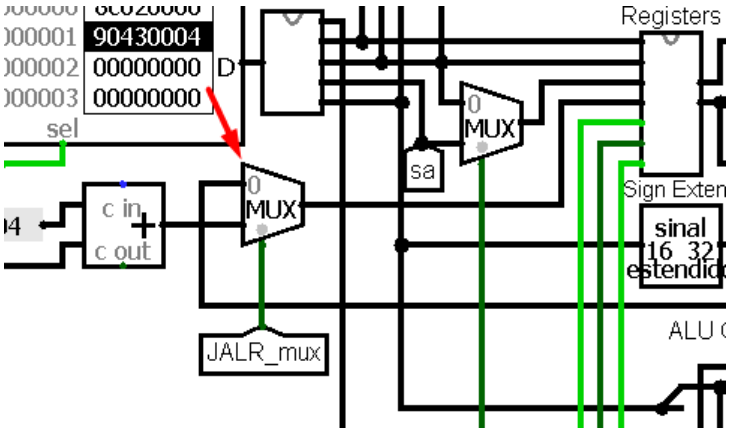


Para leitura apenas dos 8 bits menos significativos vindos da informação da memória, foi utilizado um distribuidor. Em seguida, são concatenados zeros ao byte (como é um valor unsigned, não passa por "signed extension") para formar um dado de 32 bits, pronto para ser armazenado no registrador de destino.



Operação realizada no bloco LBU indicado na imagem anterior.

Observa-se que, antes de chegar nos registradores, o dado passa pelo mux também usado pela instrução JALR.



Sinais de controle

"1001100011000" = 1318

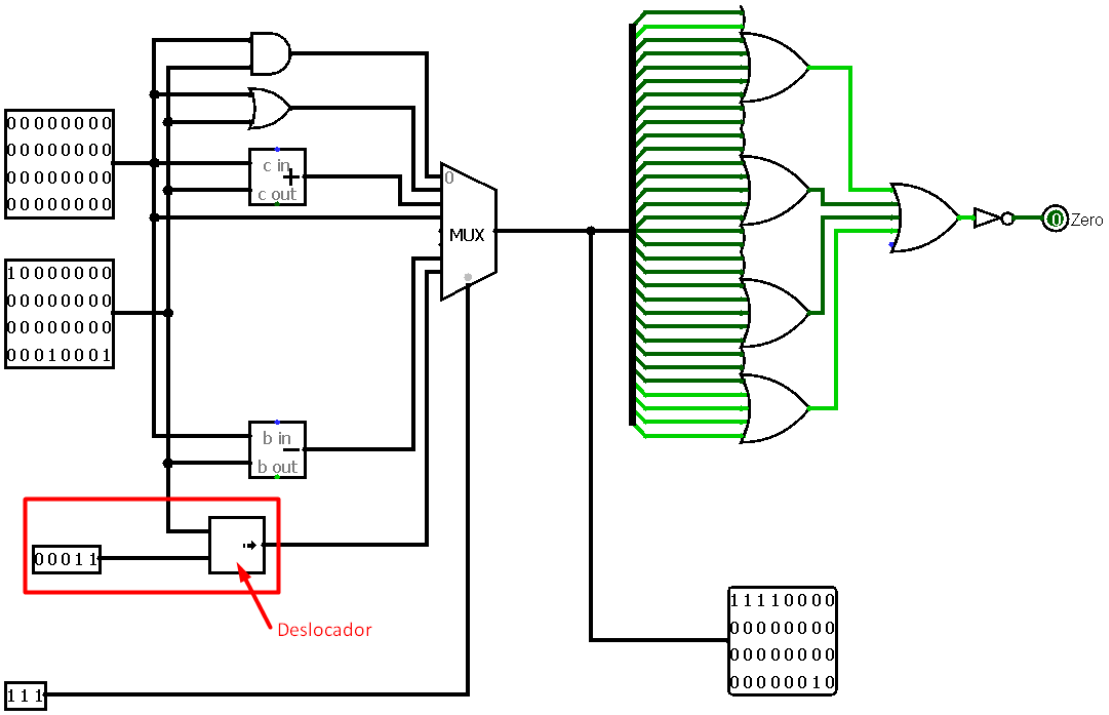
RegDst	Jump	Branch	MemRead	MemToReg	ALUOp

			X	X	00

MemWrite	ALUSrc	RegWrite	BranchLTZ	JALREnable	MemReadByte
	X	X			X

SRA NO MONOCICLO

No caso desta instrução, foi adicionada uma operação com deslocador e uma nova entrada de 5 bits (“sa”) na ULA. Estes 5 bits vem dos bits 12..7 da instrução, que indicam o deslocamento a ser feito.



Observa-se que o sinal de controle do mux da ULA “111” é produzido no bloco “ULA_Control” pelo campo “funct” da instrução (“000011”).

Sinais de controle

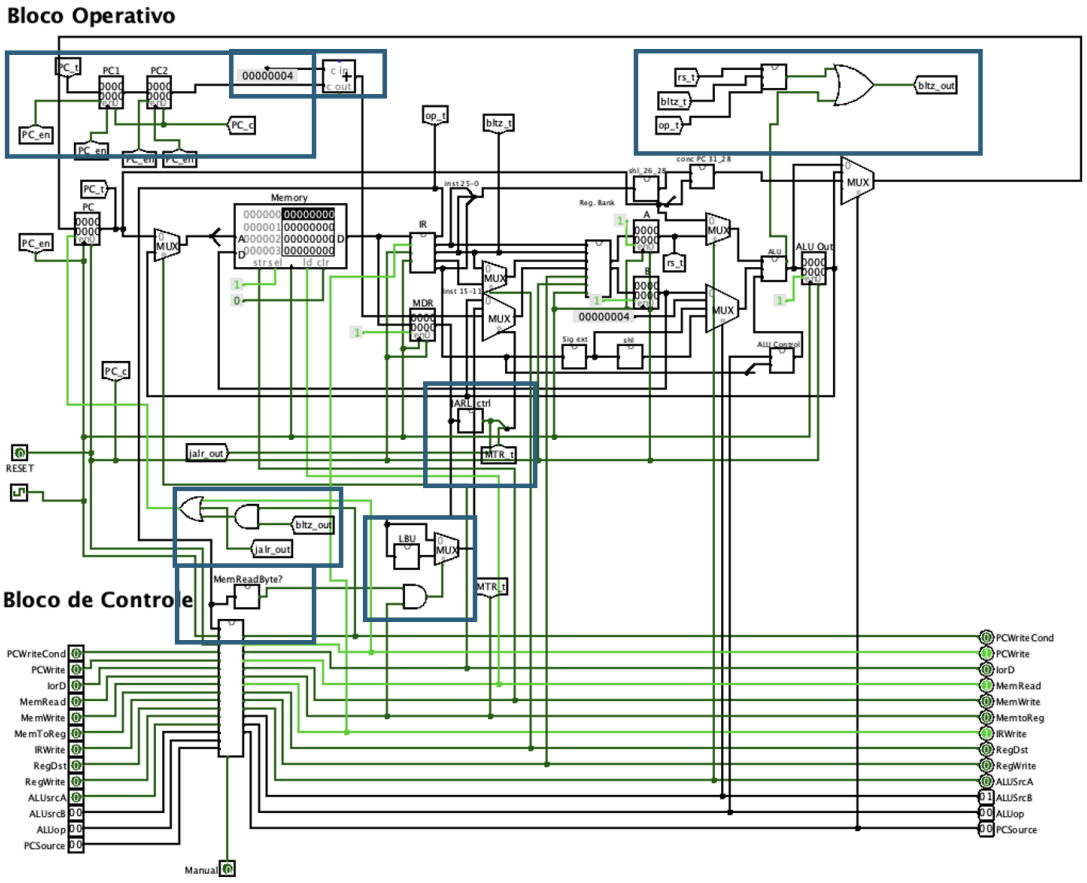
“0101001000001” = 0a41

RegDst	Jump	Branch	MemRead	MemToReg	ALUOp
X					01

MemWrite	ALUSrc	RegWrite	BranchLTZ	JALREnable	MemReadByte
		X		X	

PS: Apesar da flag “JALREnable” ativa, já que se trata de uma instrução do “tipo R”, ela não influencia no resto do circuito, pois o campo funct também é checado para realização das operações de JALR.

2) Multiciclo

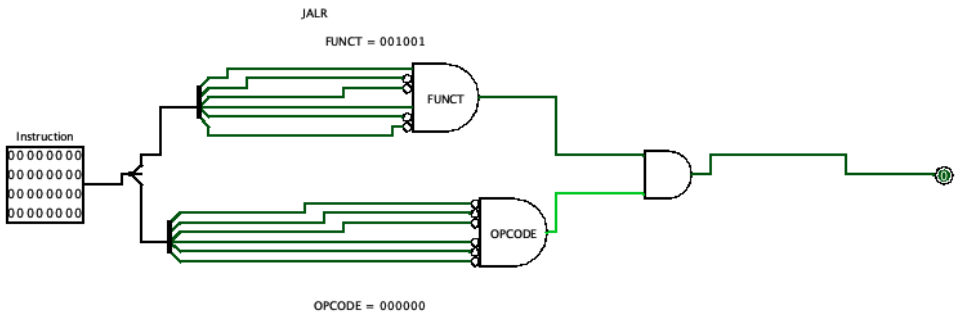


Multiciclo: visão geral

JALR NO MULTICICLO

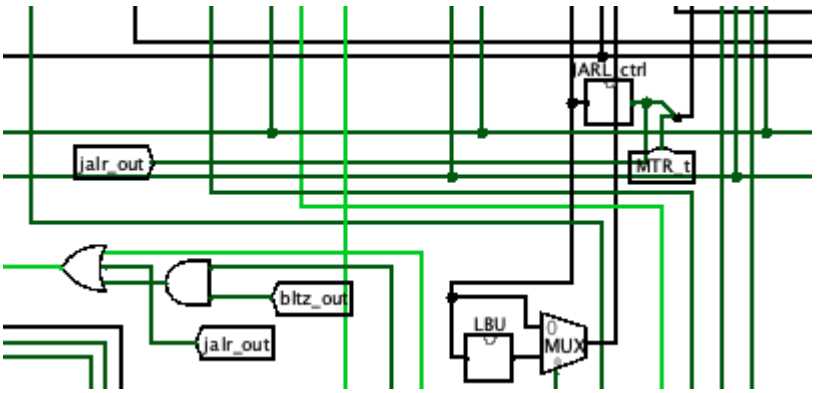
O multiciclo exigiu mudanças na implementação da JALR.

Criamos uma estrutura JALR_CTRL para definir o sinal de controle da instrução a partir do OPCODE e do campo FUNCT.



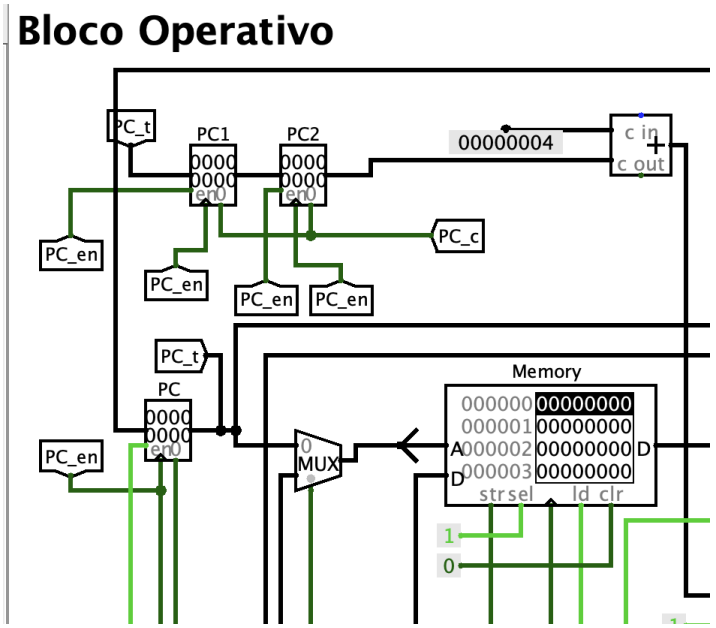
JALR_CTRL

Passamos o sinal de saída, jalr_out, para a OR que decide o que será escrito no PC.



Posição do circuito JALR_ctrl e Sinal jalr_out

Por fim, em função da presença de mais de um ciclo, criamos um buffer de PC para lidar com o atraso da instrução.

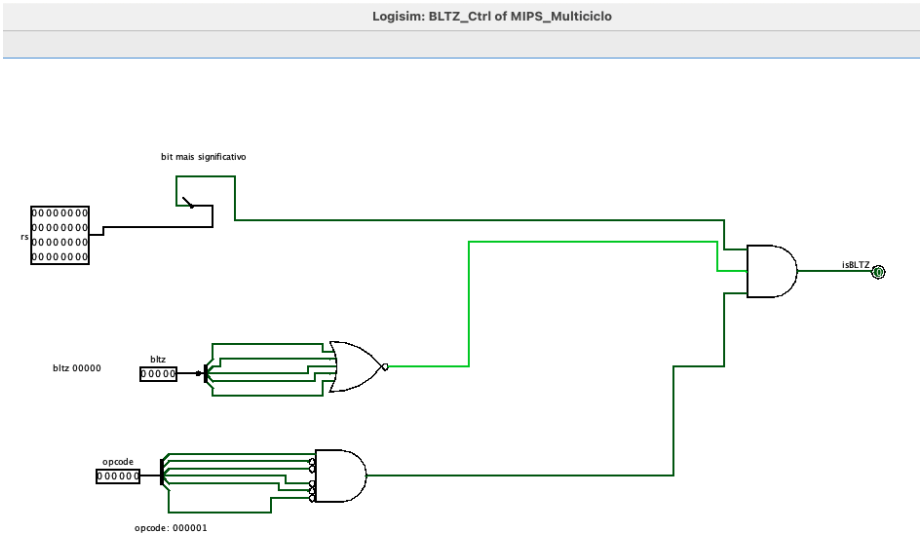


Buffer de PC

BLTZ NO MULTICICLO

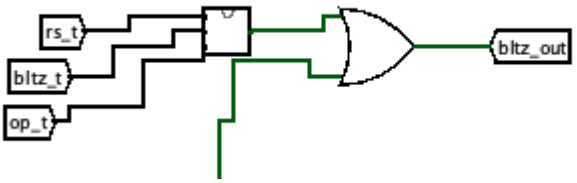
BLTZ_Ctrl

De forma análoga, criamos o circuito BLTZ_Ctrl para definir se o sinal de BLTZ estaria ligado ou não. Este circuito recebe os bits REGIMM e BLTZ da instrução para verificar se é a instrução BLTZ. Ele também valida que o bit mais significativo de rs é 1, indicando um valor negativo. Se essas três condições forem verdadeiras, o sinal isBLTZ sai ativado.



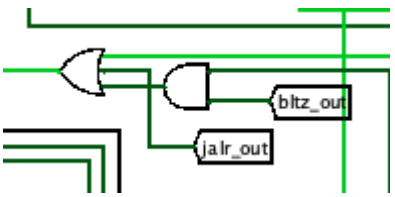
BLTZ_Ctrl

A saída isBLTZ passa por uma OR junto com o sinal zero da ALU, para definir se, em algum dos casos, BLTZ ou BEQ, se trata de uma instrução de branch. Ou seja, apesar do nome, bltz_out também é ativada no caso de a ALU ter ativado o sinal zero, apontando que talvez seja necessário fazer um BEQ.



OR com o resultado de BLTZ_Ctrl e o sinal Zero da ALU.

Assim como na JALR, adicionamos o valor de bltz_out para ver se o valor de PC deve ser modificado, por se tratar de uma instrução de branch.

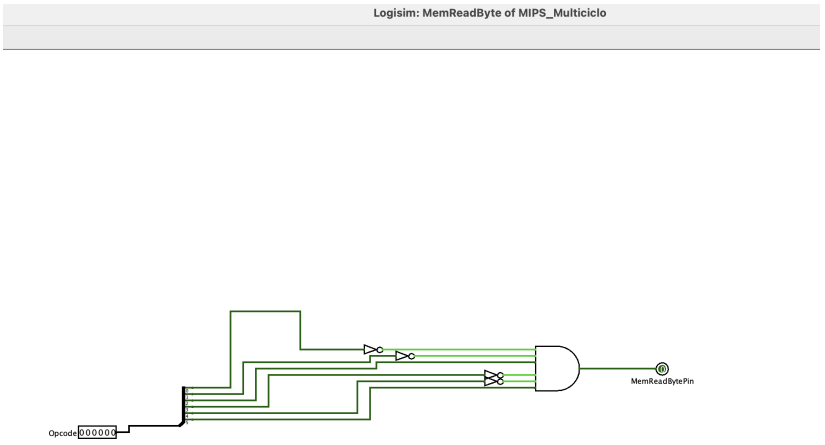


Sinal bltz_out na AND com PCWriteCond

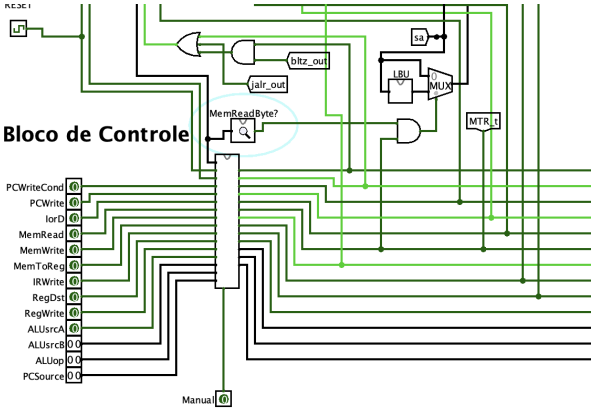
LBU NO MULTICICLO

Utilizamos a mesma implementação do Monociclo, com a mesma operação do circuito que chamamos de LBU para fazer a manipulação dos bytes. Para a definição dos estados, consideramos que LBU é uma operação semelhante à Load Word (LW) e segue os mesmos passos.

Criamos o circuito MemReadByte para ver se a instrução possui o opcode da LBU e gerar o sinal MemReadByte.

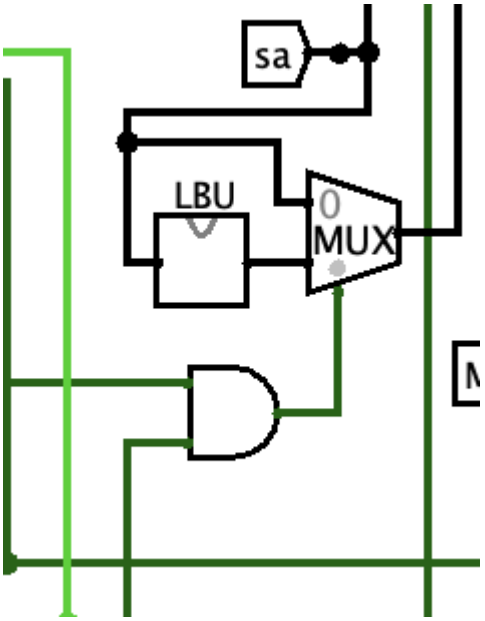


Circuito MemReadByte

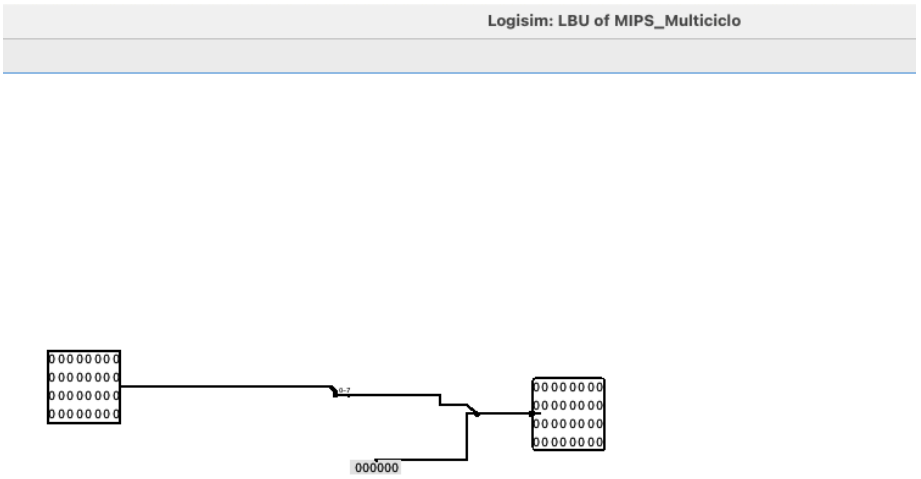


Posição do circuito MemReadByte

Para a implementação, juntamos os sinais MemReadByte e MemtoReg em uma porta lógica AND que ativa o MUX que decide qual valor será passado para o MUX conectado ao banco de Registradores. Assim, se MemReadByte e MemtoReg estiverem ativos, trata-se da instrução LBU e o byte unsigned será carregado. Caso apenas MemtoReg seja ligado, trata-se de uma LW.



LW ou LBU

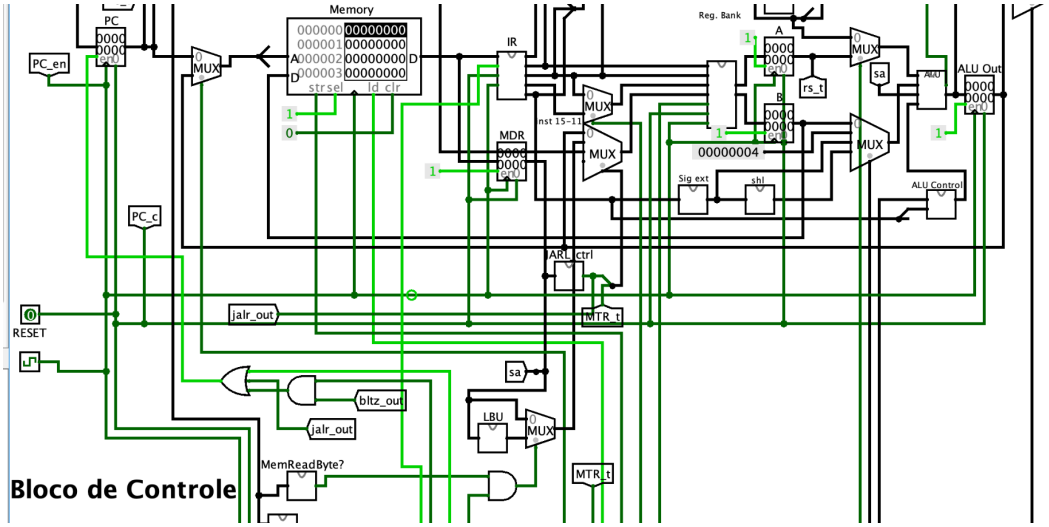


Circuito interno da LBU (mesma lógica do Monociclo)

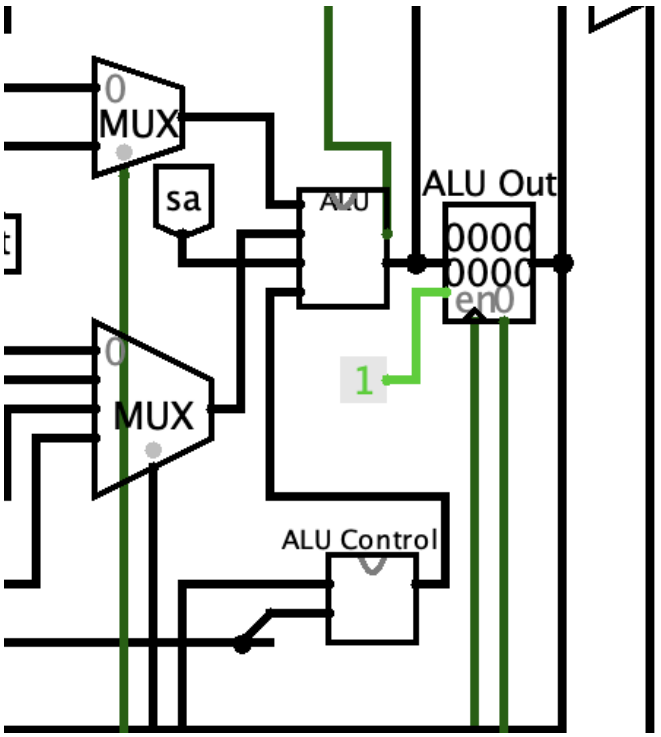
SRA NO MULTICICLO

A instrução SRA não exigiu grandes mudanças no multiciclo, pois foi implementada apenas com mudança na ALU. Assim, adicionamos a mesma alteração na ALU, mas precisamos identificar que seria uma instrução SRA no ciclo certo. Por isso, separamos os bits 6-10 da saída do MDR.

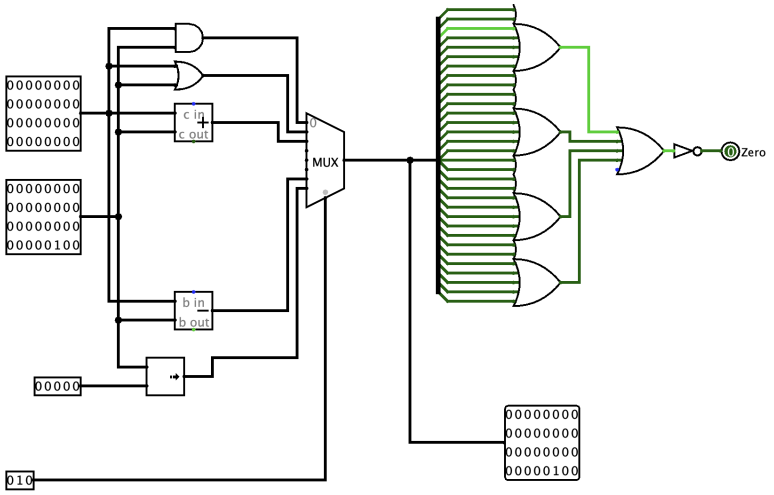
Através de um túnel (que chamamos de *sa*), passamos esses bits como entrada para a ALU, onde implementamos o Shift Right Arithmetic.



Posição do túnel para instrução SRA



Entrada dos bits pelo túnel *sa* na ALU



Lógica do Shift Right Arithmetic na ALU

SLTIU: não foi implementada

Máquina de Estados

Não foi preciso fazer grandes mudanças na máquina de estados. Houve apenas a criação dos novos sinais de controle. SRA é uma instrução do tipo R, por isso segue o mesmo fluxo das outras instruções com o mesmo opcode. Da mesma forma, a instrução JALR também não acarretou modificações por se tratar de uma instrução do tipo R e poder seguir com as mesmas etapas, apenas ativando o sinal dentro do circuito.

Como juntamos os sinais de BLTZ e BEQ para realizar a branch, não foi necessário modificar a máquina de estados para incorporar o BLTZ. Já no caso da instrução LBU, fizemos uma bifurcação no fluxo da operação LW, pois adicionamos um mux que avalia se o sinal MemReadByte está ativo, junto com MemtoReg, o que indica que se trata de uma instrução LBU. Se apenas o sinal MemtoReg estiver ligado, trata-se de uma LW. Assim, a única diferença está na etapa Write-Back para as duas instruções de Load.

Visão global

Regras:

- Aparece apenas o nome, = 1
- Não aparece o nome, = 0
  (cuidado, pode ser don't care)

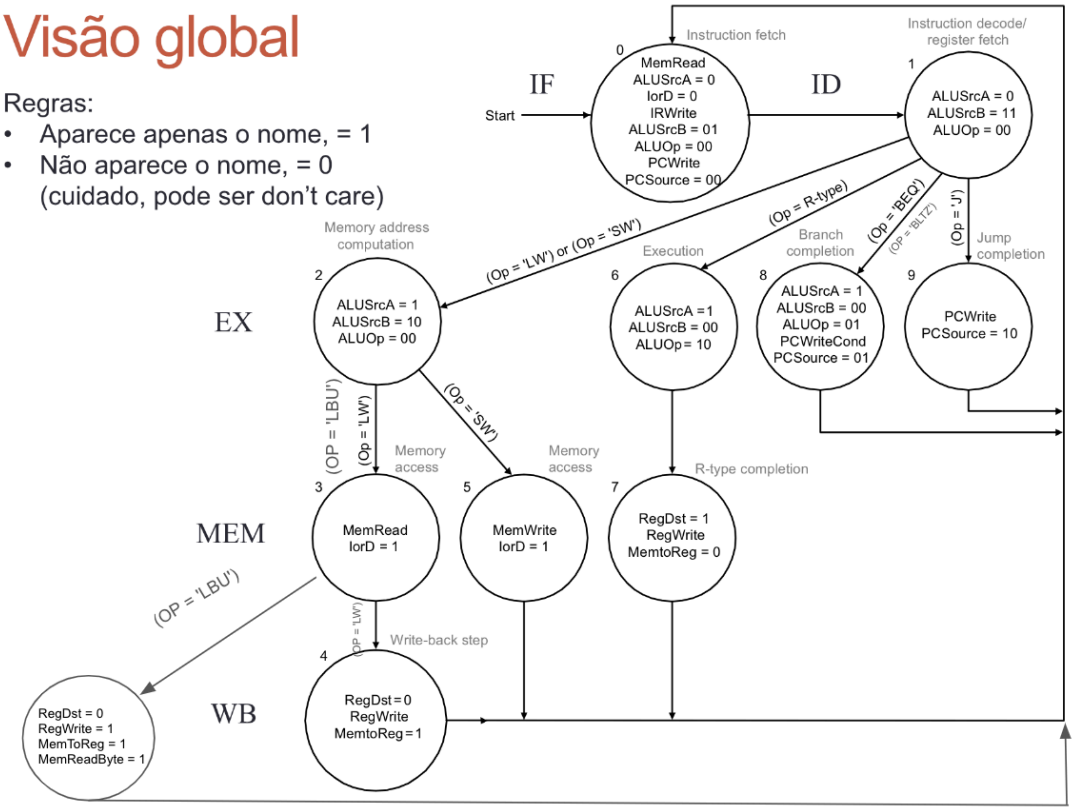
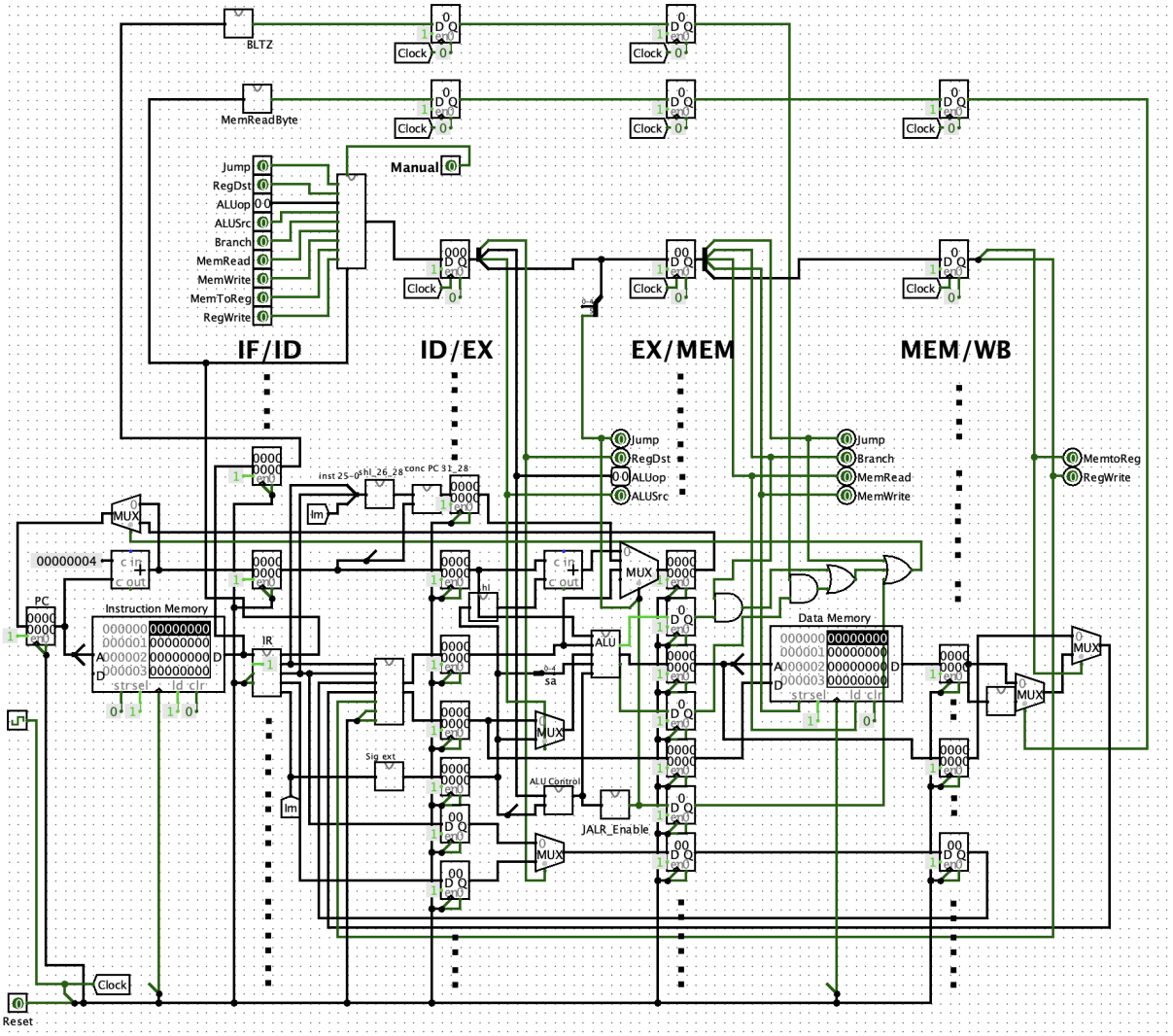
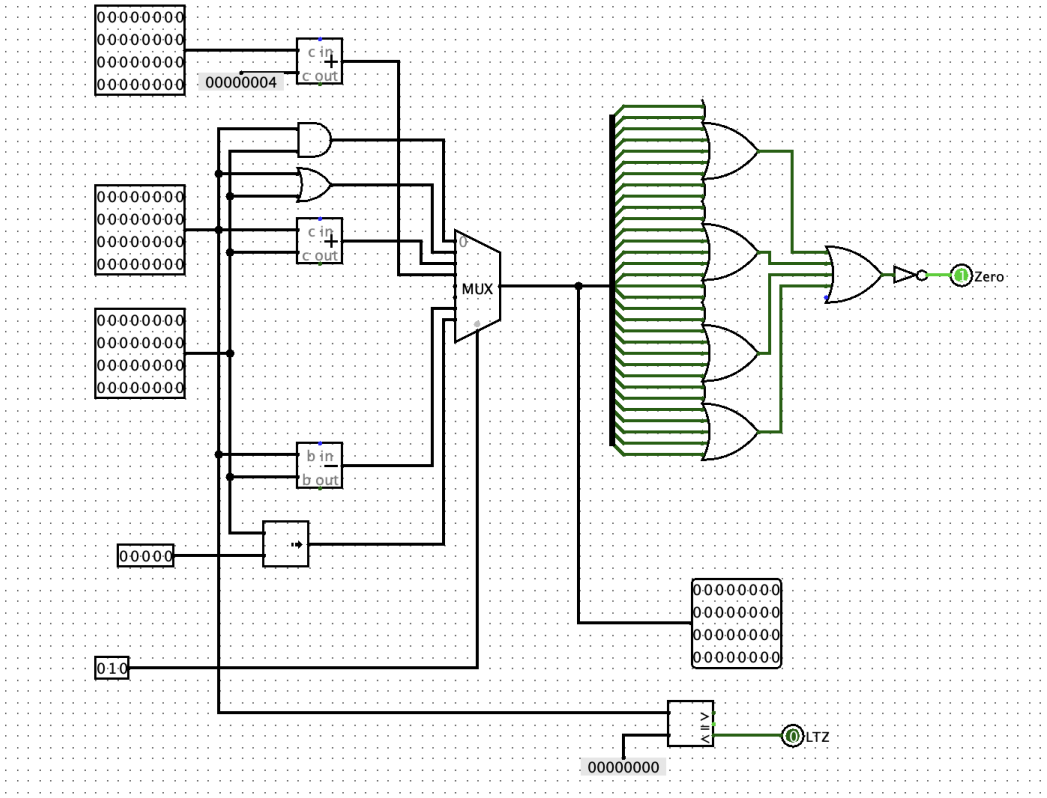


Diagrama de estados modificado

3) Pipeline

Visão geral

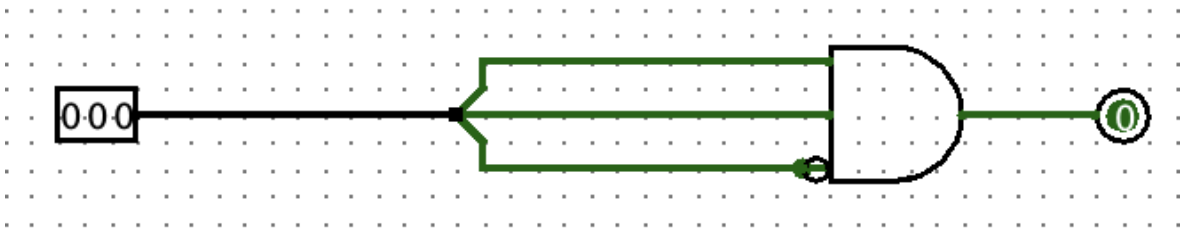




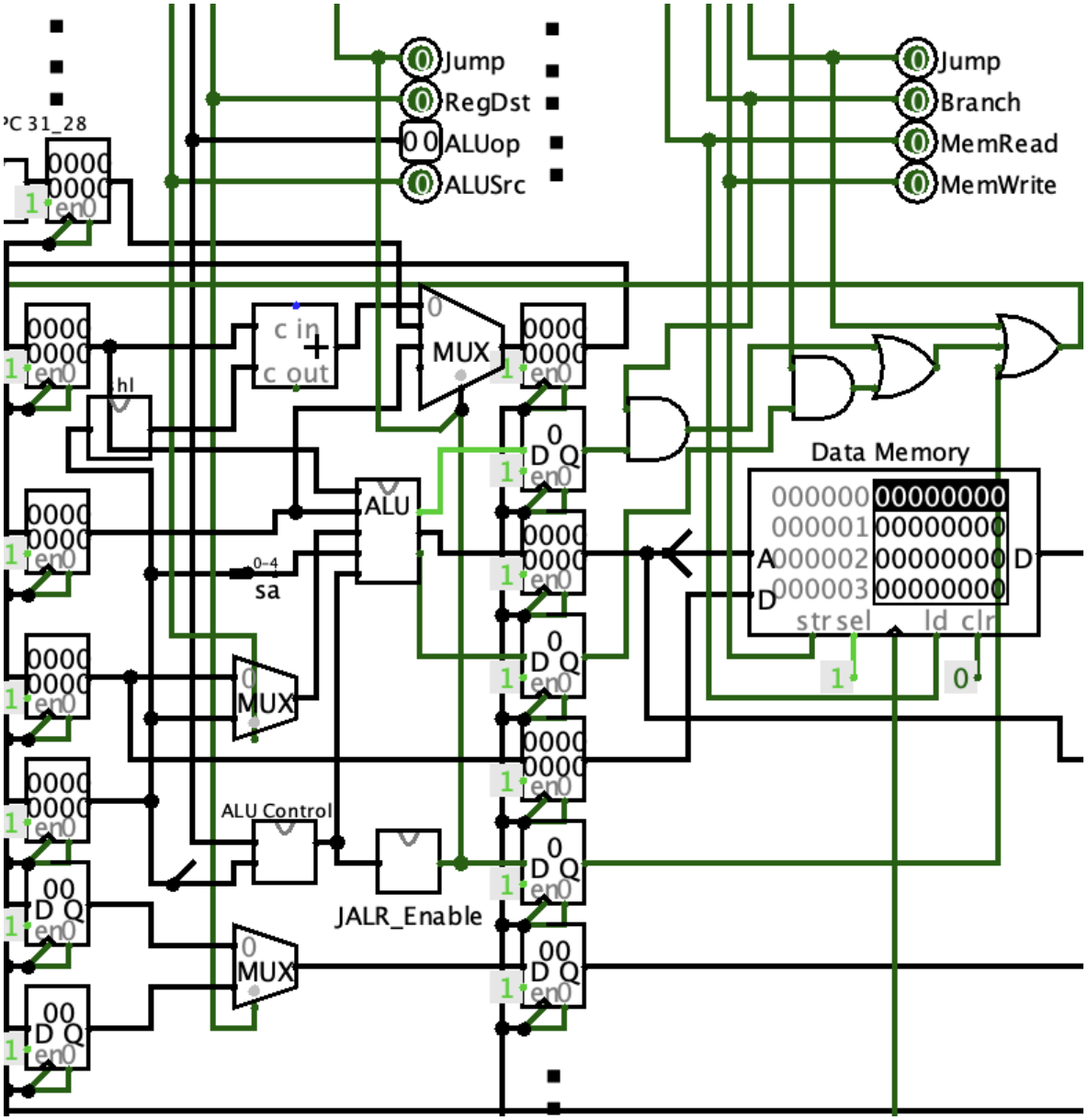
ULA Pipeline

JALR NO PIPELINE

Para implementar o JALR no Pipeline, foram feitas duas alterações. Primeiro, foi adicionado um incremento de PC + 4 na ULA para sincronizar com o estágio seguinte . Segundo, um circuito e um registrador para JALR_Enable, para que possa propagar na próxima barreira temporal, onde uma porta OR propaga um sinal para o Mux que controla o próximo valor de PC.



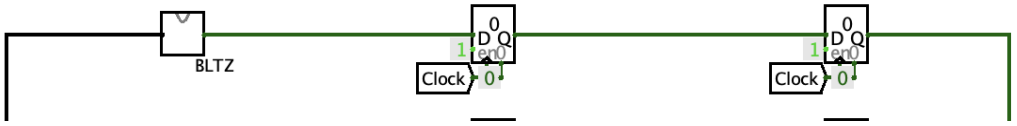
JALR_Enable



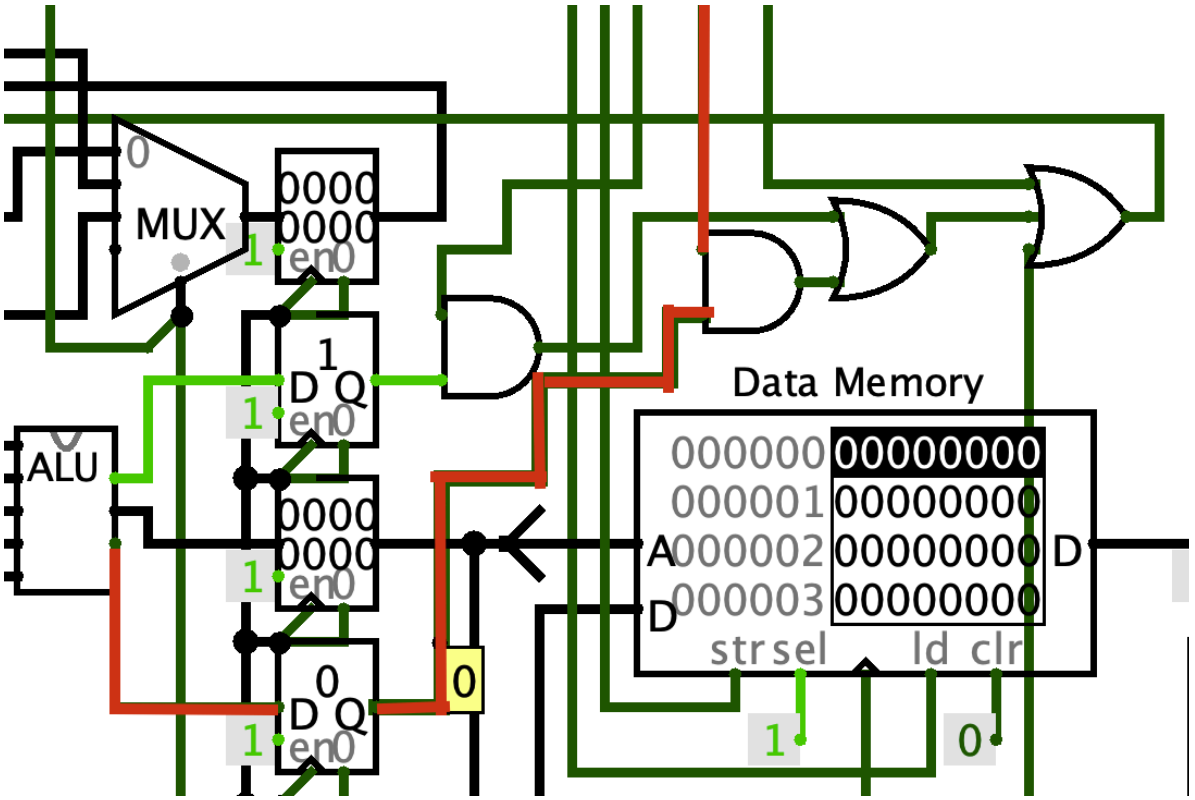
Componentes adicionados para instrução JALR

BLTZ NO PIPELINE

Foi inserido na ULA um sinal de LessThanZero que propaga para a próxima barreira temporal. O sinal é comparado em uma porta AND com o sinal vindo dos registradores adicionais BLTZ.



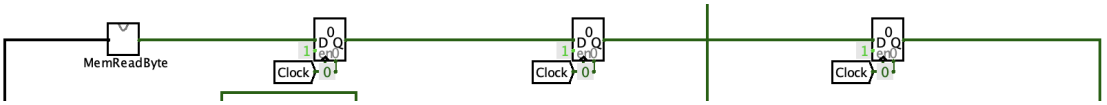
Registradores adicionais para instrução BLTZ



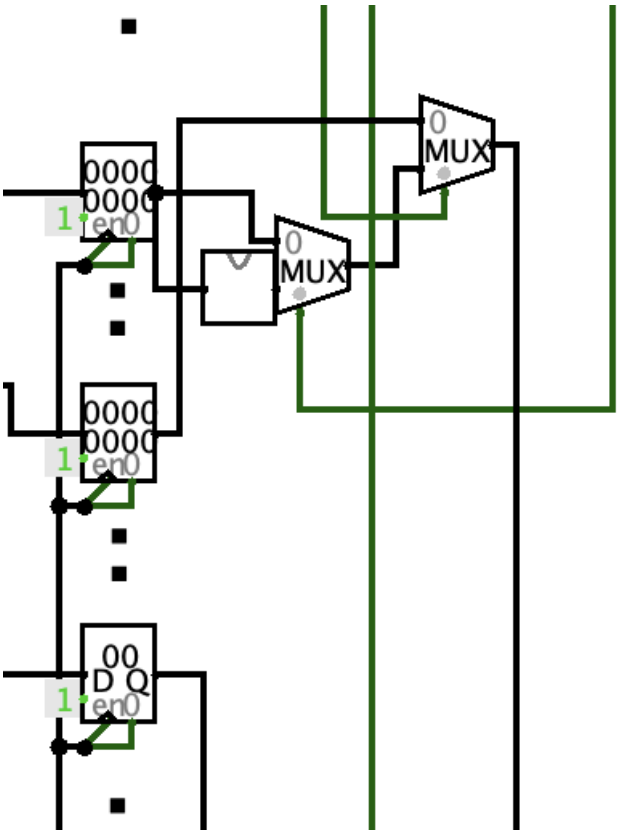
Controle da ULA (LessThanZero) e sinal BLTZ

LBU NO PIPELINE

Foram utilizados os mesmos componentes do Monociclo, com a adição de barreiras temporais para o sinal MemReadByte. O sinal controla o Mux que determina se a saída é palavra inteira ou somente um byte.



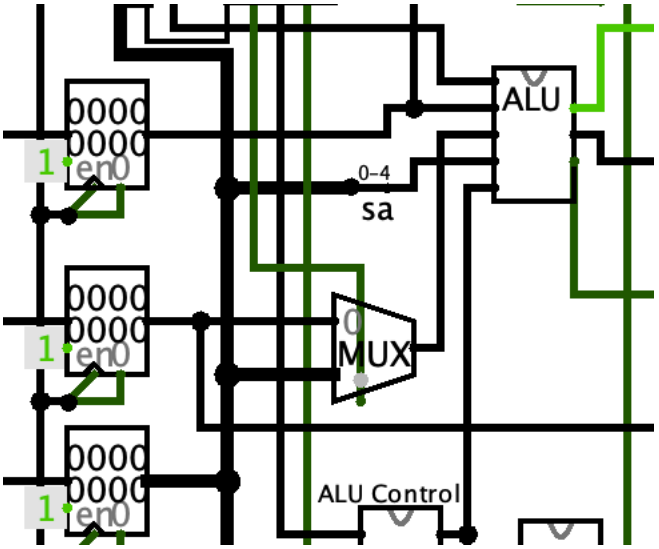
Registradores adicionais para instrução LBU



LBU com MUX para saída de escrita

SRA NO PIPELINE

Utilizou a mesma modificação da ULA que Monociclo e Multiciclo. O valor do shift amount é retirado dos bits 0-4 do registrador na barreira temporal.



Entrada do Shift Amount na ULA

SLTIU: não foi implementada