Relatório sobre o Primeiro Trabalho Prático

Nome: Bernardo C. Zietolie **Matrícula:** 00550164
Nome: Guilherme Ismael Flach **Matrícula:** 00324108
Nome: Igor Ianicell De Latorre **Matrícula:** 00305018
Nome: Miguel Lemmertz Schwarzbold **Matrícula:** 00342191
Nome: Ricardo Zanini De Costa **Matrícula:** 00344523

Monociclo

A implementação das instruções no MIPS monociclo exigiu alterações no bloco operativo e no bloco de controle. Dessa forma, a estratégia adotada foi priorizar a simplicidade do datapath, mesmo que isso implicasse a necessidade de mais bits de controle.

As principais alterações no bloco operativo foram a adição de multiplexadores e o aumento dos bits de seleção dos multiplexadores já existentes no circuito. Também foram incluídas novas operações na ULA, como XOR, divisão e identidade. As operações XOR e divisão são necessárias para as instruções XORI e DIV. A operação de identidade, por sua vez, auxilia a instrução BGEZAL, pois fornece o primeiro operando para a ULA. Além disso, foram adicionadas portas lógicas AND para determinar se um desvio deve ser realizado, atuando no bit de seleção do multiplexador de branch. Por fim, para implementar a instrução LB, foi criado um circuito que recebe o valor lido da memória, os dois bits menos significativos da saída da ULA e um bit de controle, que indica se o valor a ser lido da memória corresponde a uma palavra ou a um byte. Ou seja, se o bit de controle indicar que o valor lido é um byte, os dois bits menos significativos da saída da ULA determinarão qual byte da palavra será carregado.

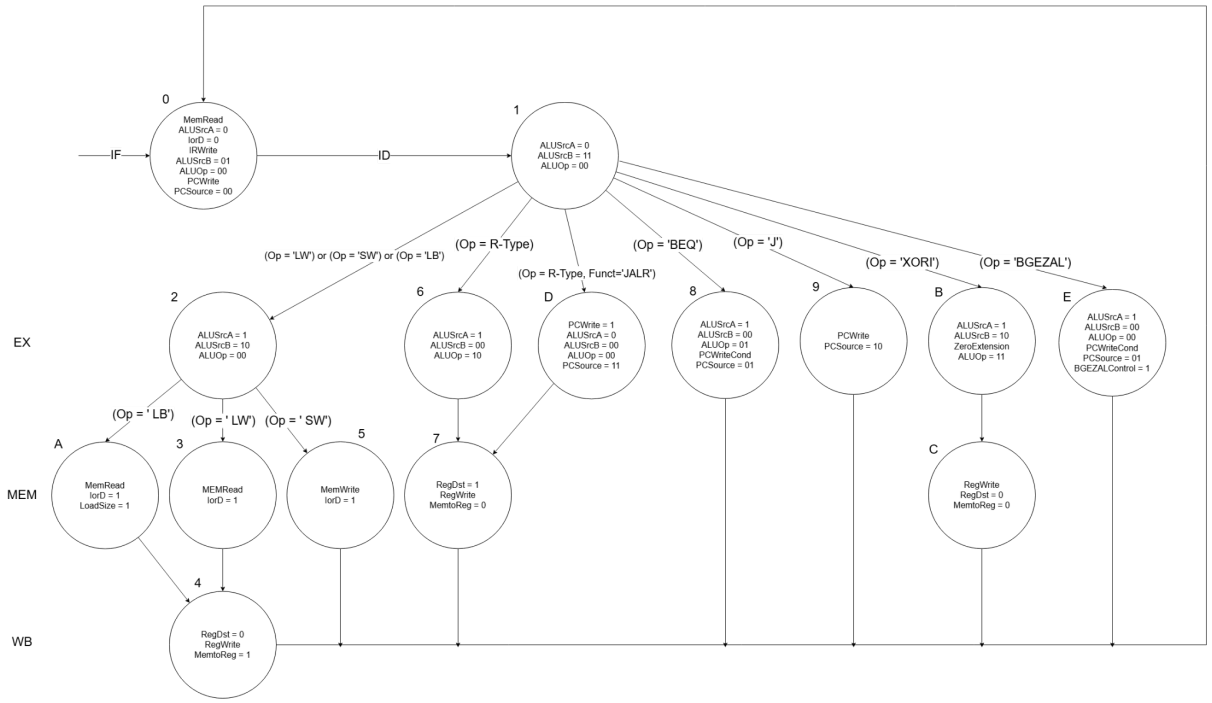
As mudanças no bloco de controle envolveram a adição de 5 novos bits de controle, a modificação do ALUControl para selecionar as operações adicionadas na ULA e a criação de um circuito específico para operações com o opcode 000000.

Os bits de controle adicionados foram: **ByteOrWord**, **PcToReg**, **PcToR31**, **JumpFromR** e **BGEZAL**.

- **ByteOrWord** controla se o valor a ser carregado da memória é um byte ou uma palavra.
- **PcToReg** sinaliza que o valor de PC + 4 deve ser gravado no registrador indicado pela instrução.
- **PcToR31** indica que o valor de PC + 4 deve ser gravado no registrador 32.
- **JumpFromR** sinaliza que o jump deve ser feito para o endereço armazenado no registrador *rd*.
- **BGEZAL** indica que o desvio condicional deve ser realizado quando o valor lido for não negativo.

Por fim, o circuito **TypeRControl** fornece os bits de controle quando o opcode da instrução é 000000. Nesse caso, os bits de controle são determinados pelo campo *funct*. Assim, a instrução JALR, que compartilha o mesmo opcode de outras instruções aritméticas, fornece os bits de controle necessários para realizar o jump e salvar o endereço de PC + 4 no banco de registradores.

Multiciclo



DIV

Uma vez que DIV é uma instrução do tipo R sem grandes efeitos colaterais, sua implementação não necessitou de grandes mudanças. Para a parte operativa, apenas foram adicionados dois registradores, *LO* e *HI*, dentro da ALU, para guardar o quociente e resto da operação, respectivamente. Para fazer o controle da escrita desses registradores, adicionou-se um sinal de controle “interno”, que é ativado enquanto a ALU é coordenada a realizar uma operação de divisão.

Já na parte de controle, alterou-se o bloco *ALU_Control*, para gerar a saída 101 quando *ALU_OP* e *FUNCT* fossem 10 e 011011, respectivamente. Por fim, cogitou-se criar uma saída na ALU que bloqueasse a escrita no banco de registradores quando a operação de DIV fosse realizada, mas isso não se mostrou necessário, uma vez que a arquitetura do MIPS define que o campo equivalente ao RD (que as outras instruções do tipo R usam como destino), na instrução DIV, é deixado como 00000. Assim, como o registrador R0 é sempre 0, tentar realizar uma escrita nele não gera impacto real. Por fim, para evitar o sinal Zero da ALU, definiu-se que a saída da ALU durante a operação de DIV seria FFFFFFFF.

XORI

Dentro da parte operativa foi necessário fazer duas coisas:

- Incluir o XOR dentro da ALU;
- Criar um caminho com extensão de zero para transportar o imediato;

Para a primeira, foi adicionada uma porta XOR de 32 bits para realizar a operação, acessada dentro do MUX da ALU na posição 011. Na parte operativa, também foi criado um novo sinal de *ALUOp*, 11, que força a ALU a fazer uma operação XOR, usada para realizar essa instrução.

Também, foi preciso criar um caminho com extensão de zero, já que outras instruções que usam imediatos (como LW e JMP), fazem extensão de sinal nesses bits. Junto com isso, criou-se o sinal *ZeroExtension*, que, quando em 1, faz com que valores imediatos passem por extensão com zero, ao invés de uma extensão de sinal.

Por fim, também foram criados dois novos estados na unidade de controle: **B** e **C**, em B o dado imediato é lido na ALU com sinal estendido por zero, juntamente com o valor do registrador indicado em rs, e é enviado um sinal para forçar um XOR; em C, o resultado da ALU é guardado no registrador indicado em rt.

LB

Para implementar-se o LB, reaproveitou-se boa parte do datapath da instrução LW, apenas com uma modificação que permite escolher se a saída da memória para o Memory Data Register deve ser de 32 bits ou de 8 (com sign extension) bits. Essa escolha é controlada pelo sinal *LoadSize*: 1 para 8 bits, 0 para 32. Após isso, pode-se continuar normalmente, como uma instrução de LW. Desse modo, na FSM da UC, foi colocado um novo estado, **A**, que é essencialmente igual ao estado 3 (usado pelo LW), é lido um dado da memória e colocado em MDR, mas que é lido com apenas 1 byte com sign extension. Após isso, o fluxo faz um writeback de memória normal.

JALR

Sendo uma instrução do tipo R, é natural que o JALR siga o datapath desse tipo de função. Porém, no estado de execução da função, foi necessário utilizar sinais de controles com valores diferentes (*PCWrite* = 1, *ALUSrcA* = 0, *ALUOp* = 00, *PCSource* = 11). Assim, *PCWrite* = 1 permite fazer o jump, *PCSource* = 11 permite que o valor do registrador rs seja levado diretamente para o PC (conectamos a saída do registrador A no MUX controlado pelo *PCSource*), e *ALUSrcA* = 0 e *ALUOp* = 0 permitem que o valor correto seja escrito no registrador no próximo passo. O estado seguinte, então, é o mesmo das instruções do tipo R comuns.

Para diferenciar JALR no Bloco de Controle, verificamos se o campo *funct* da instrução é igual a 001001. Se for, ele usa o estado D no lugar do estado 6 (que seria o normal para o tipo R).

BGEZAL

Para esta instrução, foi criado o sinal de controle *BGEZALControl* = 1, que decide três MUX. No MUX que decide o registrador de escrita, o sinal faz com que seja escolhido o registrador 31, que deve receber o endereço de retorno da

subrotina. No MUX que decide o valor a ser escrito no registrador, o sinal faz com que seja escolhido o valor do PC.

No MUX que decide o código de controle da ULA, forçamos que o código passado seja 100, pois, na ULA, esse é o código que permite executar um AND entre o valor do registrador rs e uma constante de 32 bits com o bit mais significativo valendo 1. Assim, se o valor de rs for maior ou igual a zero, a ULA acionará o sinal *Zero*, o qual indicará que deve-se escrever no registrador 31 e no PC. Caso contrário, o sinal *Zero* não será acionado, e nada será feito.

Pipeline

O foco para a implementação das novas instruções no MIPS pipeline foi a minimização das mudanças no processador. Foi criada uma nova ROM na unidade de controle para armazenar as flags das novas instruções. Isto foi feito para simplificar a criação das novas flags, pois foi julgado mais simples do que modificar o tamanho da ROM já existente. A ALU foi também alterada para efetuar as novas operações (xor e div) e para conter os registradores HI e LO, portanto é sincronizado o relógio com o resto do processador. A unidade de controle da ALU também foi atualizada, com flags para ignorar o comportamento padrão e direcionar a ALU a executar as instruções novas, que lidam com imediatos e registradores fora do banco.

XORI

XORI	Rs	Rt	Immediate
001110	xxxxx	xxxxx	xxxxxxxxxxxxxxxx
6	5	5	16

A instrução foi implementada inserindo um portão XOR na ALU. Como a instrução não é do tipo SPECIAL e não contém campo FUNCT, a unidade de controle central redireciona a unidade de controle da ULA para ignorar o

comportamento padrão e executar o XOR com os valores retirados do imediato da instrução.

DIV

SPECIAL	Rs	Rt	0	DIV
000000	xxxxx	xxxxx	0000000000	011010
6	5	5	10	6

A instrução DIV foi implementada como uma função especial da ALU. Implementações mais eficientes dividiram a instrução em múltiplos estágios da pipeline, mas em nome da simplicidade a escolha foi de inserir um bloco divisor na ALU. Isso gera uma perda de eficiência mas torna a implementação mais coesa.O código FUNCT da instrução é passado pela pipeline até a unidade de controle da ALU, que faz uma comparação direta para perceber se a instrução é um DIV. Caso seja, guarda os valores nos registradores HI e LO, presentes dentro da ALU. A unidade de controle garante que os valores só sejam escritos em HI e LO se a instrução for do tipo SPECIAL, com a função DIV.

BGEZAL

REGIMM	base	BGEZAL	offset
000001	xxxxx	10011	xxxxxxxxxxxxxxxx
6	5	5	16

A instrução BGEZAL foi baseada fortemente no caminho de dados da instrução BEQ. Foi criado uma flag Greather Or Equal To Zero na ALU, e um sinal de controle define qual das flags é utilizada para definir se um branch é feito ou não. A posição da memória no delay slot é passado pela pipeline, e quando é a hora de efetuar a linkagem, é somado 4 (para obter o endereço da instrução após o delay slot) e inserido no banco de registradores, com a seleção do registrador 31 feita com uma constante. A unidade de controle garante que na hora da linkagem a escrita nos registradores é feita no registrador 31, com o valor esperado de memória.

JALR

SPECIAL	Rs	—	Rt	—-	JALR
000000	xxxxx	000000	xxxxx	000000	001001
6	5	5	5	5	6

O JALR não teve grandes problemas para ser implementado, ele apenas faz um pulo incondicional para o conteúdo de Rs, salvando o PC atual + 4 em Rd. Para sua implementação foi necessário a adição de registradores nas camadas do pipeline, e a criação de uma flag “JALRControl” que decide o valor armazenado em Rd, que diz para qual endereco o pulo incondicional deve acontecer. Para essa nova flag foram adicionados dois novos registradores. Por fim, foi feita uma modificação no Controle para que a instrução JALR tenha suas flags definidas com base no campo “funct”.

LB

LB	base	Rt	offset
100000	xxxxx	xxxxx	xxxxxxxxxxxxxxxx
6	5	5	16

A instrução LB funciona de maneira muito semelhante com a instrução LW, com a diferença de carregar-se um Byte da memória ao invés de uma palavra completa. Para realizar essa função sem que fosse necessária uma modificação interna na própria memória, a instrução continua a buscar uma palavra inteira internamente, mas faz um shift de 24 bits na palavra para que apenas o Byte buscado permaneça na palavra. Após isso os dados sofrem uma extensão de sinal e são enviados para o banco de registradores para o seu armazenamento por meio de um multiplexador com a flag “MemToRegShifted24” que pertence ao 5° bit dos sinais extras adicionados no controle. Após isso o Byte é armazenado no registrador Rt finalizando a instrução. Todas as flags acionadas foram: “AluSRC”, “MemRead”, “RegWrite” e “MemToRegShifted24”. (OBS: Como no logisim as memórias têm limite

de endereçamento de 24 bits ocorre que o LB implementado só consegue acessar o primeiro Byte de palavras, mas supondo uma memória normal com endereçamento de 32 bits isso não ocorreria).