Segundo trabalho prático de OC I
2020/1

Ábner de Marcos Neves, 2018054605

Leonardo de Almeida Brito, 2018046696

Departamento de Ciência da Computação da
Universidade Federal de Minas Gerais

1. Introdução

O trabalho a seguir consiste na implementação das instruções *ori, slli, lui, lwi, swap, ss, blt, bge, jump* em um caminho de dados de risc-v com ciclo único pré-existente, com o intuito de nos familiarizarmos com a linguagem de descrição de hardware Verilog.

2. Implementação das instruções

1. ORI - *bitwise or immediate*

A instrução *ori* é codificada com o mesmo *opcode* da *addi*, que já estava implementada no processador. Por isso, elas precisam ser diferenciadas pelo *funct3*, no bloco de execução. A partir desse momento, portanto, enquanto o sinal enviado à ALU pela *addi* é referente à operação de soma, a *ori* envia o sinal da operação *or*, que também já possuía uma implementação pré-existente no *datapath*.

A diferença entre essas duas instruções, *ori* e *or*, é que, na primeira, um registrador é comparado com um imediato, em vez de outro registrador, mas esse problema já é resolvido logo no bloco de decodificação, atribuindo-se ao sinal de controle *alusrc* o valor 1. A implementação dessa instrução pôde, então, tirar proveito dessas duas outras.

2. SLLI - *shift left logical immediate*

Tal qual a instrução *ori*, a *slli* possui o mesmo *opcode* da *addi* e precisou ser diferenciada dessas outras duas pelo seu *funct3*. Dentro da ALU, entretanto, não havia ainda a operação requisitada pelo *slli* e foi necessário acrescentar uma nova opção para ela, que computa o valor de $A << B$, em que A é o conteúdo do registrador *rs1* e B, o imediato.

3. LUI - *load upper immediate*

Essa instrução aloca o imediato nos bits mais significativos do registrador de destino, completando com zeros os menos significativos. A princípio, a implementação escolhida já resolvia o problema no bloco de decodificação, concatenando o imediato lido com 12 zeros do lado direito, mas, mais tarde, a fim de que o processo se aproximasse mais daquele retratado nos testes disponibilizados, essa tarefa foi delegada à ALU.

4. LWI - *load with immediate*

Para a implementação da função Load with Increment , não foram necessários o acréscimo de nenhum outro bloco funcional, assim como a adição de nenhum outro caminho. Como a função *lwi* não é oficial na especificação do risc-v, utilizou-se um *opcode* novo no valor de 0000001, para simular essa função. Além dela, o *funct3* seria correspondente ao de *add/addi* no valor de 000, para executar o comando de somar os dois registradores.

Para o carregamento do resultado na memória (*load*), utilizou-se os mesmos sinais de controle que a função *load*, que nesse caso são *memtoreg*, *regwrite* e *memread*, com exceção dos sinais *alusrc* e *ImmGen* que são para funções do tipo I (imediato). Portanto, a única mudança foi a criação de um novo *opcode* que acionaria sinais específicas do *control* para essa instrução.

5. SWAP

Devido à possibilidade de executar atribuições não bloqueantes com Verilog, a troca de conteúdo entre dois registradores não demanda um auxiliar, entretanto exige que o processador escreva nesses dois registradores paralelamente.

Para isso, foi criado um sinal de controle *swap* que avisa ao banco de registradores quando realizar esse comportamento. O bloco de decodificação fica encarregado de mantê-lo consistente, atribuindo o valor de verdadeiro durante a execução de uma instrução *swap* e falso durante outras instruções.

No caso de esse sinal ser verdadeiro, ocorre a atribuição de *read_data1* ao registrador *rs2* paralelamente à de *read_data2* ao registrador *rs1*. Se o sinal for falso, então o banco de registradores procede normalmente.

6. SS - *store sum*

A implementação dessa nova função não oficial exigiu adequações maiores do que aquelas apresentadas para *lwi.* Assim como essa, a função *store sum* teve um *opcode* criado no valor de 0000100, recebendo os sinais de comando da função *store (memwrite)*, além do *ImmGen* para trabalhar com valores imediatos.

Como na função *store* o registrador substituído pelo valor imediato é o *reg2*, por meio do *alusrc*, não se utilizou esse comando no *opcode* de *ss*, sendo criado um sinal de controle *storeSum*, para fazer a troca do *reg1* pelo valor imediato.

Além disso, para que ocorra o salvamento na memória do resultado (*aluout*) da soma entre o reg2 e o valor imediato foi necessário criar um novo caminho (*writedataIn*) até a entrada *writedata* no bloco *memory,* assim como um outro (*addressIn*) que saísse de *reg1* até o *address* desse mesmo bloco. Para a implementação desses novos caminhos, foi necessário a adição de dois *mux* controlados pelo sinal de *storeSum* que decide qual saída que irá para as respectivas entradas do bloco *memory*.

7. BLT - *branch on less than*

A *branch on equal* já possuía implementação no *datapath* e confere se A e B são iguais subtraindo um pelo outro e verificando se o resultado é igual a zero, caso em que o *branch* especificado será tomado.

Por causa dessa implementação, também já existia o sinal *branch*, indicador de quando a instrução é desse tipo, e o sinal *zero*, que é verdadeiro quando a saída da ALU é igual a zero. Assim, o bloco *fetch* atribui o valor de *new_pc* baseado nesses sinais: se ambos são verdadeiros, então o *offset* especificado pelo imediato deve ser levado em conta para o cálculo do endereço da próxima instrução. Se não, *new_pc* apenas recebe normalmente o valor de *pc + 4*.

Para aproveitar esse mecanismo pré-existente e implementar a função *branch on less than*, bastava, então, que os sinais *branch* e *zero* ficassem consistentes também às condições dessa instrução e mais nada precisaria ser modificado.

Uma vez identificado o *opcode* da *blt*, que é o mesmo da *beq*, o sinal *branch* torna-se verdadeiro já no bloco de decodificação. Agora, sendo diferenciadas pelo *funct3*, a *beq* continua seu processo normalmente e a *blt* envia um novo controle à ALU para que seja calculada a seguinte operação:

$$aluout <= A < B \ ? \ 32'd0 : 32'd1.$$

Assim, *aluout* será igual a zero quando a condição da *blt* for satisfeita e, consequentemente, o sinal *zero* será verdadeiro, mantendo-se consistente a essas duas instruções do tipo *branch*.

8. BGE - *branch on greater than or equal*

A *branch on greater than or equal* tem funcionamento análogo às outras duas *branches* explicadas no item anterior. Exceto que a ALU processa a operação

$$aluout <= A >= B \ ? \ 32'd0 : 32'd1.$$

9. J - *jump*

A pseudoinstrução *jump* pode ser codificada como uma *jump and link* em que *rd = x0*. Como o *offset* nessas instruções é sempre um múltiplo de 2 *bytes*, é necessário que ele seja multiplicado por 2, a fim de que fique alinhado com os endereços de memória, que são múltiplos de 4. Isso é feito logo na decodificação, acrescentando um 0 no final do imediato.

Além disso, foi criado um novo sinal de controle *jump* para indicar quando se trata desse tipo de instrução. Esse sinal é ligado ao bloco *fetch* e, uma vez verdadeiro, *new_pc* recebe *pc + sigext*, em que *sigext* é o imediato, agora um múltiplo de 4, estendido para 32 bits.

Como *jump* é o único sinal de controle ativado por essa instrução, nenhum valor será armazenado como endereço de retorno, o que está de acordo com a proposta da pseudoinstrução *jump*.