Universidade Federal de Minas Gerais
Departamento de Ciência da Computação

TRABALHO PRÁTICO 2
ORGANIZAÇÃO DE COMPUTADORES 1

Alexander Thomas Mol Holmquist 2018066255
Daniel Souza de Campos 2018054664
Letícia da Silva Macedo Alves 2018054443

Outubro/2020

Sumário

1. Introdução

Esse trabalho consiste na resolução de 9 problemas propostos para compreender melhor o funcionamento de um caminho de dados (*Datapath*) de ciclo único. Cada problema descreve uma instrução presente, ou não, na arquitetura RISC-V que deve ser implementada pelo caminho de dados e, portanto, o mesmo deverá ser alterado para que todas essas instruções funcionem de modo correto.

Para a realização desse trabalho, foi utilizada uma ferramenta disponibilizada pelo DCC similar à ferramenta *DigitalJS* que, dado como entrada um programa escrito na linguagem *Verilog*, é possível sintetizar o hardware descrito e vê-lo funcionando. Além disso, a ferramenta já vinha com uma versão inicial do *datapath* implementada de acordo com a arquitetura RISC-V, o que facilitou o progresso do trabalho. Para acessar essa ferramenta, foi necessário utilizar o programa OpenVPN com configurações já ajustadas para conexão com o DCC.

Para o controle de versão do código desenvolvido, foi usado o Github, sendo que o trabalho está disponível em: <https://github.com/Pendulun/TrabalhoOc1>.

2. Resolução dos Problemas

A seguir, apresentamos as decisões de projeto tomadas na implementação de cada uma das operações que deveriam ser incluídas no caminho de dados.

2.1. Problema 1 - Instrução ORI

A operação **ori (Bitwise or immediate)** possui a forma *ori rd, rs1, imm*. Ela realiza uma operação lógica de OR bit a bit entre o valor contido em *rs1* (registrador de origem) e um valor imediato (*imm*). O número gerado pela operação é guardado, posteriormente, no registrador de destino *rd*.

Para incluí-la ao caminho de dados, observamos que, sendo ela uma operação do tipo I, possui *opcode 0010011*, assim como a operação *addi*, por exemplo, que já estava implementada no caminho de dados fornecido. Assim, *ori* também possui *aluop = 0*, atribuído no módulo *ControlUnit*.

Dessa forma, no módulo *alucontrol*, para distinguir a operação *ori* das demais de *aluop 0*, comparamos o *funct3*. Caso *funct3 = 6* (110, em binário), então trata-se de uma operação *ori*. Logo, seu *alucontrol* deve receber sinal *1*, para que na *ALU* seja realizada a operação *A or B* (r1 or imm) para a saída *aluout*.

***Sequência de testes:**

1. nop;
2. ori x1,x2,10;

2.2. Problema 2 - Instrução SLLI

A operação **slli (Shift Left Logical Immediate)** possui a forma *slli rd, rs1, imm*. Ela move todos os bits de *rs1* para a esquerda uma quantidade de casas, que é determinada

pelo valor imediato (*imm*). Esse número “shiftado” é guardado, posteriormente, no registrador de destino *rd*.

Cada shift para a esquerda representa a multiplicação do número atual por dois. Além disso, as casas de bits menos significativas (mais à direita) vão sendo preenchidas com 0 a cada shift para a esquerda.

Para incluir a operação *slli*, observamos que, sendo ela uma operação do tipo I, possui *opcode 0010011*. Assim, possui *aluop = 0*, atribuído no módulo *ControlUnit*.

Dessa forma, no módulo *alucontrol*, para distinguir a operação das demais de *aluop 0*, comparamos o *funct3*. Caso *funct3 = 1* (001, em binário), então trata-se de uma operação *slli*.

Além disso, adicionamos no módulo *ALU* um novo sinal de *alucontrol*. Assim, para *alucontrol = 3*, *aluout* receberá *rs1 << imm*. Ou seja, o recurso A “shiftado” B vezes para a esquerda. Por esse motivo, para *slli*, o *alucontrol* no módulo *alucontrol* deve receber sinal *3*.

***Sequência de testes:**

1. nop;
2. slli x1,x5,3;

2.3. Problema 3 - Instrução LUI

A operação **lui (Load Upper Immediate)** possui a forma *lui rd, imm*. É uma operação tipo U. Ela construirá um valor de 32 bits no registrador de destino *rd* da seguinte forma: Copia a constante de 20 bits da instrução (*imm*) nos bits *[31:12]* do *rd* e coloca *0* nos *[11:0]* bits restantes de *rd*.

Para incluí-la no caminho de dados, no módulo *ControlUnit* adicionamos o caso *opcode = 0110111*. Nesse caso, *alusrc*, *regwrite* e *aluop* recebem os sinais *1*, *1* e *3*, respectivamente.

Além disso, o valor imediato será composto como “*ImmGen <= {inst[31:12]}*”, isto é, como o descrito acima. Como *alusrc = 1*, na *ALU* será usado *imm* como o recurso B. No módulo *alucontrol*, foi adicionado o caso *aluop = 3* para atender à operação. Assim, *alucontrol* receberá sinal *4* e na *ALU* será gerada a saída *aluout <= {B,12'b0}* (construindo o valor de 32 bits no *rd* como o especificado).

***Sequência de testes:**

1. nop;
2. lui x5,5;

2.4. Problema 4 - Instrução LWI

A operação **lwi (Load With Increment)** não está presente na base original do conjunto de instruções do RISC-V. Ela possui a forma *lwi rd, rs1, rs2*. Sua interpretação é: o registrador de destino *rd* receberá o conteúdo que está na posição de memória *“valor em rs1 + valor em rs2”*. Isto é, os conteúdos dos registradores de origem (*rs1* e *rs2*) serão usados para calcular o endereço da memória do qual virá o valor a ser carregado em *rd*.

Para adicioná-la ao caminho de dados, consideramos seu formato de instrução como o de uma instrução tipo R, isto é:

Assim, mesmo que *lwi* seja, a rigor, um tipo de operação Load, por questão de padronização, optamos pelo formato do tipo R, já que utilizaremos *rs1* e *rs2*.

No módulo *ControlUnit* adicionamos um novo caso de *opcode*. Escolhemos *opcode = 0001010* (que ainda não havia sido usado neste conjunto de instruções) para a operação *lwi*. A configuração de sinais para esse caso ficou definida como: *alusrc = 0*, *memtoreg = 1*, *regwrite = 1* e *memread = 1*.

Como possui *aluop = 0*, atribuído no módulo *ControlUnit*, para distinguir a operação das demais de *aluop 0*, comparamos o *funct3*. Caso *funct3 = 5* (101, em binário), então trata-se de uma *lwi*. Este valor atribuído para o *funct3* foi escolhido por nós (já que ainda não estava em uso para nenhuma outra instrução do caso *aluop = 0*) para atender a necessidade de implementação da função.

Com essas especificações, *alucontrol* receberá sinal *5*. Dessa forma, no módulo *ALU* um novo sinal de *alucontrol* foi adicionado e *aluout* receberá *(A + B) << 2*, sendo A = *rs1*, B = *rs2*.

Uma observação é que a soma é "shiftada" duas vezes para a esquerda pois o endereço de memória acessado deve ser múltiplo de 4, já que os dois últimos bits menos significativos do *aluout* serão descartados na busca de posição de memória.

***Sequência de testes:**

1. nop;
2. lwi x1, x2, x4;

2.5. Problema 5 - Instrução SWAP

A operação ***swap* (literalmente, "trocar")** não está entre as instruções padrões do RISC-V. A função tem formato *swap rs1, rs2*. RTL: *reg[rs1] = reg[rs2]; reg[rs2] = reg[rs1]*. O *opcode* e a decodificação da instrução foi de acordo com os testes providos (*opcode = 2*). Como a *ControlUnit* não reconhece *case(opcode) = 2*, adicionou-se este caso, atribuindo *alusrc = 1*, *regwrite = 1*.

O problema então surge de, na etapa de *writeback*, realizar escrita em dois registradores. Para isso, procedemos adicionando saída no módulo *mips*, e propagando essa modificação (módulos *decode*, *writeback* e *Register_Bank*). Então, conectamos a saída *data2* de *decode* ao módulo *writeback*, e faz-se a conexão interna em *writeback*. Depois, precisamos fazer com que *writedata2* (a nova saída) chegue no banco de registradores, e haja uma escrita em *memory[read_reg1]*. Para isso, criamos novo sinal *regwrite2* em *decode*, e o conectamos a *ControlUnit* e *Register_Bank*; depois adaptamos estes módulos de acordo, notavelmente adicionando um bloco condicional com as atribuições dentro do bloco "*always*" no módulo *Register_Bank*, para realizar a conexão interna.

Após isso, o processador adotou, aparentemente sem hazard de dado, a função *swap*.

***Sequência de testes:**

1. nop;
2. swap x4, x5;
3. sub x2, x5, x4;
4. add x4, x2, x2;
5. swap x12, x6.

2.6. Problema 6 - Instrução SS

A operação ***ss* (store sum)** não está entre as instruções padrões do RISC-V. Tem o formato *ss rs1, rs2, imm*, e interpretação *Mem[Reg[rs1]] = Reg[rs2] + imm*. O *opcode* utilizado seguiu o dos testes do monitor (*opcode = 8*), assim também a decodificação dos 32 bits da instrução. O formato da instrução é como a "*type S*" do RISC-V. Depois de adicionar gerenciamento com base no *opcode* em *ControlUnit*, nos deparamos com o problema de que, na presente configuração, não é possível realizar a soma entre o registrador *rs2* e o imediato.

Assim, precisamos adicionar um caminho entre *rs2* e a primeira entrada da *ALU*, passando por um multiplexador. Como sinal de controle para este multiplexador, definimos *alusrc1*, e renomeamos *alusrc* para *alusrc2*. Dessa forma, *alusrc1* agora controla a entrada de "cima" da *ALU*, e *alusrc2* a de "baixo"(multiplexador antigo).

Porém, depois de conseguir fazer a soma rs2 + imm, temos o problema de a entrada *address* em *memory* estar definida como *aluout*, o que faria ela receber, necessariamente, o resultado da soma acima. Decidiu-se, então, adicionar um output no módulo *execute* - *aluout1* - para conter o *Reg[rs1]* solicitado por essa instrução. O antigo *aluout* foi renomeado *aluout2*, e um multiplexador para cada entrada de *memory*, *address* e *writedata*, foram adicionados. Esse novo *aluout1*, até esse ponto, simplesmente passa o valor *Reg[rs1]* para frente.

***Sequência de testes:**

1. nop;
2. ss x1, x5, 0;
3. ss x1, x5, 5;
4. ss x9, x2, 13.

2.7. Problema 7 - Instrução BLT

A operação ***blt* (branch on less than)** é definida no "R32I Base Instruction Set" no formato *blt rs1, rs2, imm*. Os bits são codificados como instrução tipo B, sendo (*funct3 = 100*) e (*opcode = 1100011*). Esse *opcode* já estava coberto no código. O que não estava coberto é, primeiro a detecção de quando a subtração utilizada para o branch tem resultado diferente de *zero*.

Assim, acrescentou-se o sinal *lt*(less than) como output da *ALU*, para indicar que *A - B < 0*. Note que com *lt* e *zero*, é possível fazer todo tipo de comparação (*lt*, *leq*, *eq*, *geq*, *gt*), fazendo combinações lógicas entre esses dois sinais. Depois disso, deve-se notar que, em RISC-V, somente as instruções de branch têm *opcode 99*, e, portanto, se o sinal *branch* na *ControlUnit* for igual a 1, certamente é um tipo de branch, que, por sua vez, é uma instrução do tipo B. Instruções do tipo B sempre apresentam *funct3* nas mesmas posições.

Portanto, podemos fazer um *switch-case* no módulo *fetch* que decide qual tipo de branch é, com base nos 3 bits extraídos das posições [14:12] de *inst*. Dentro desse *switch-case*, ainda é feita checagem do sinal *branch* e são utilizados *zero* e *lt* para determinar o *new_pc*.

***Sequência de testes:**

1. nop;
2. ori x8, x0, 2;

3. ori x9, x0, 3;
4. blt x8, x9, 8;
5. addi x8, x9, 2;
6. addi x8, x9, 5;
7. blt x8, x9, -12.

2.8. Problema 8 - Instrução BGE

A operação **bge (branch on greater than or equal)** é definida no "R32I Base Instruction Set" no formato *bge rs1, rs2, offset*. Da mesma forma que o problema anterior, ela tem *opcode = 1100011* mas com *funct3 = 101*.

Devido ao jeito em que a instrução *blt* foi implementada, um *switch-case* no *fetch* decide qual tipo de branch está sendo realizado. Dessa forma, foi apenas necessário adicionar a opção em que o *funct3 = 101*. Caso seja o branch em questão, é feito uma operação lógica em cima dos sinais vindos da *ALU*, *lt* e *zero* que analisa, primeiramente, se é de fato um branch e se os sinais *lt* (less than) **ou** *zero* são verdadeiros, definindo o valor correto para o novo *pc*, conforme a funcionalidade da instrução.

***Sequência de testes:**

1. nop; (pc=0)
2. bge x5,x4, 12; (hexadecimal: 0042d663) (pc=4)
3. ori x8, x0, 2; (hexadecimal: 00206413) (pc=8)
4. ori x9, x0, 3; (hexadecimal: 00306493) (pc=12)
5. addi x8, x9, 2; (hexadecimal: 00248413) (pc=16)
6. addi x8, x9, 5; (hexadecimal: 00548413) (pc=20)

Nesse teste, com x5=5 e x4=4, o branch deverá ser tomado para a instrução 6, pulando todas as outras, pois o novo *pc* será igual a *pc + 4 + 12*, da forma que está implementado no código.

2.9. Problema 9 - Instrução J

A operação **j (Jump)** é considerada uma pseudo instrução de formato *j x0, offset* variante da instrução *jal*. Ela possui *opcode = 1101111* e sem *funct3*.

Para identificar a instrução *jump*, foi adicionado ao *switch-case* da *ControlUnit* o valor do *opcode* da instrução.

Do mesmo jeito que para as instruções *branch*, ficou decidido adicionar outro sinal de *output* da *ControlUnit*, nomeado *jump*, para servir de *flag* no módulo *fetch* para apoiar a escolha do novo valor de *PC*. Essa *flag* só será igual a *1* quando for uma instrução *jump* e, caso isso seja verdadeiro, o novo valor de *PC* deverá ser calculado com base no pc + 4 e o *offset* passado na instrução.

***Sequência de testes:**

1. nop; (pc=0)
2. j x0, 12; (hexadecimal: 00c0006f) (pc=4)
3. ori x9, x0, 3; (hexadecimal: 00306493) (pc=8)
4. addi x8, x9, 2 ; (hexadecimal: 00248413) (pc=12)
5. j x0, 8; (hexadecimal: 0080006f) (pc=16)
6. lui x6, 5; (hexadecimal: 5337) (pc=20)
7. lui x5, 1997; (hexadecimal: 7CD2B7) (pc=24)

Nessa sequência de instruções, o primeiro *jump* deverá saltar para a instrução de *PC=16*. Além disso, de modo análogo, o segundo *jump* deverá saltar para *PC=24*.

3. Conclusão

Começamos com um processador bem simples, ciclo único, que comportava instruções como *add*, *sub*, *beq*, *lw*, *sw*, etc. Então, procurando expandir o leque de instruções disponíveis, descobrimos que algumas se “encaixavam” naturalmente no código.

Para o Caminho de Dados se adaptar, bastava incluir novos valores possíveis para *opcode* e *funct3*, e especificar sinais de controle - já existentes - adequados para a instância.

Em alguns casos, porém, mostrou-se inevitável a adição de novos componentes (e.g. multiplexadores), conexões entre módulos, etc. Nesses casos, nos deparamos com a realidade de que o processador é um todo (um chip integral), e toda e qualquer decisão, além de satisfazer os requisitos funcionais, deve ser feita seguindo ao menos dois princípios: não encurtar a funcionalidade existente (*backwards compatibility*), e facilitar futuras expansões (*extensibility*).

Mais concretamente, fica evidente como a complexidade do hardware necessariamente aumenta ao adicionar instruções, e isso explica o surgimento das arquiteturas minimalistas, como a RISC-V, em contraste com modelos complexos como a CISC. O escopo de aprendizado inclui, ainda, o entendimento de como um processador funciona, já que pudemos visualizar em tempo real a propagação dos sinais.