Relatório: Trabalho RISC-V

Aluno: Romeu Tamarozi Bernardes

Tabela de controle completa no arquivo “controle.txt”, neste diretório.

1. Introdução:

Neste trabalho, o aluno tinha de completar um modelo de processador baseado em RISC-V, no simulador Digital. Foi passado pelo professor o processador suportando algumas instruções. O objetivo do aluno era implementar outras sete instruções obrigatórias e uma à sua escolha.

2. Resolução nos circuitos: na ordem que eu implementei as instruções

1 - sll, srl e sra:

Como especificado no documento do trabalho, era permitido utilizar o componente *barrel shifter* já pronto no Digital, que pode ser alterado para realizar o *shift left logical* (sll), o *shift right logical* (srl) e o *shift right arithmetical* (sra). Então para implementar as instruções sll, srl e sra, eu coloquei 3 *barrel shifters*, cada um configurado para realizar uma das instruções, na ULA, e conectei o sinal A nas entradas “in” dos *shifters* e os seis bits menos significativos do sinal B nas entradas “shift”. Então eu configurei o binário do sinal de controle para essas instruções, cada uma tendo um sinal “aluFunc” único (cf. “controle.txt”).

Para selecionar qual sinal sairá do “R” da ULA, eu conectei a entrada “F” (que seleciona a funcionalidade da ULA) a três ANDs, cada um checando para o sinal “aluFunc” de uma instrução. Cada um desses ANDs tem a saída ligada ao seletor de um dos três MUXs que estão ligados em cadeia e selecionam o sinal de saída de um dos *shifters*.

Assim, quando o sinal de “F” for o de sll, a ULA retornará o resultado da operação *shift left logical*, quando o sinal de “F” for o de srl, a ULA retornará o resultado da operação *shift right logical*, quando o sinal de “F” for o de sra, a ULA retornará o resultado da operação *shift right arithmetical* .

2 - jal:

A instrução jal (*jump and link*) soma o sinal “const” ao sinal “PC”, que é a instrução atual, e torna essa soma o novo número da instrução, o novo “PC”. Então quando a instrução for jal é necessário que o valor de entrada “D” de PC seja “PC + const”. Assim, eu coloquei um MUX que seleciona entre o sinal “PC + 1” (a próxima instrução) e o sinal “PC + const” (a instrução para qual se deseja saltar no jal), cada uma saindo do resultado de um *full adder*. O sinal de seleção desse MUX será gerado pela memória de Controle com 2 bits e chama-se “selPC”, sendo que eu o adicionei à última coluna da tabela de controle, o que significa que todos os sinais de instrução que já estavam implementados tiveram de ser multiplicados por quatro na tabela do Digital (pois dois bits ao final de um número binário significa dois *shifts* à esquerda).

Na instrução jal, também se armazena o valor “PC + 1” em um registrador. Assim, foi preciso adicionar mais uma entrada ao MUX de seleção da entrada “C” dos registradores que receba “PC + 1”. Portanto, também foi necessário tornar o sinal “selC” um sinal de 2 bits, colocando-se “PC + 1” na entrada 3 (10).

3 - jalr:

A instrução jalr (*jump and link register*) torna a próxima instrução a instrução “r(a) + const” (ou seja, o registrador passado na instrução mais a constante). Assim, é necessário conectar a uma entrada do MUX de seleção de “PC” o valor “r(a) + const”. Essa soma será realizada pela ULA (o que significa que o sinal “aluFunc” será 000, que corresponde à adição) e o resultado “R” será enviado para a entrada 3 (10) do MUX. O sinal “selPC”, então, será 10 para a instrução jalr.

Assim como em jal, em jalr o valor “PC + 1” deve ser guardado em um registrador “r(c)”. Então o sinal “selC” será 10 também para jalr.

4 - beq:

O comando beq (*branch on equal*) decide a quantidade de instruções puladas a partir do resultado da subtração dos valores em “r(a)” e em “r(b)”. Se r(a) - r(b) = 0, a próxima instrução será “PC + const”; senão, a próxima instrução será “PC + 1”.

Então para implementar o beq foi necessário utilizar a saída “zero” da ULA, e usar o sinal “aluFunc” igual a 001 (de subtração). Se o sinal “zero” for 0 (ou seja, o resultado da operação não for 0), o MUX que seleciona o número da próxima instrução soltará o valor de “PC + 1”; se o sinal for 1 (ou seja, o resultado da operação for 0), a próxima instrução deverá ser “PC + const”. Por isso, foi preciso adicionar um MUX que escolhe o bit mais à direita de “selPC” entre o bit dado pela memória Controle e o bit “zero”. Pois quando o sinal “zero” for 0, “PC + 1” (que está na entrada 00 do MUX) deverá ser selecionado, e quando o sinal “zero” for 1, “PC +

const” (que está na entrada 01 do MUX) deverá ser selecionado. Esse MUX que escolhe o bit mais à direita de “selPC” receberá no seletor um novo bit de controle chamado “selBeq” (na primeira coluna da tabela em “controle.txt”), que será 1 apenas quando a instrução for beq, tornando o bit à direita de “selPC” o bit “zero”.

5 - blt:

O comando blt (*branch on lower than*) também decide a quantidade de instruções puladas a partir do resultado da subtração dos valores em “r(a)” e em “r(b)”. Se r(a) - r(b) < 0, a próxima instrução será “PC + const”; senão, a próxima instrução será “PC + 1”.

Portanto é necessário usar o sinal “neg” da ULA (que será 1 quando o resultado da operação for menor que 0, e 0 no caso contrário) e também o sinal “aluFunc” igual a 001 (subtração). A implementação de blt é, então, similar à de beq, mas utilizando o sinal “neg” ao invés de “zero”. Um MUX foi adicionado, escolhendo entre o valor de saída do MUX de seleção de beq e o sinal “neg”. Como sinal de seleção, esse MUX recebe um novo bit da memória Controle, chamado “selBlt”, que só será 1 quando a instrução for blt. Assim, quando a instrução for blt, o bit mais à direita de “selPC” será igual ao sinal “neg”.

6 - slt:

Na parte 2.1 do trabalho (de implementar uma nova instrução) eu escolhi a instrução slt (*set on lower than*). Nessa instrução passam-se três registradores, “r(a)”, “r(b)” e “r(c)”. Se o valor de “r(a)” for menor que o valor em “r(b)”, “r(c)” receberá o valor 1; senão, “r(c)” receberá o valor 0.

Para aplicar essa lógica eu adicionei um MUX de um bit de seleção que recebe 0 na entrada 0, 1 na entrada 1 e o sinal “neg” da ULA (o que significa que o sinal “aluFunc” terá também o valor 001 de subtração aqui) como sinal de seleção. Assim, quando o resultado da ULA for negativo, o MUX soltará 1, e quando o resultado da ULA não for negativo, o MUX soltará 0. A saída desse MUX está conectada à entrada 4 (11) do MUX de seleção do valor “C” dos registradores. Então, o sinal “selC” será 11 quando a instrução for slt.

3. Mudanças na ROM Controle:

A memória ROM de controle possuía, no projeto inicial passado pelo professor, 8 locais de memória preenchidos cada um com uma instrução, e 7 bits para cada instrução.

Ao final do meu trabalho foram adicionadas 8 novas instruções, cada uma preenchendo um local da memória Controle, o que totaliza 16 instruções que preenchem os 16 espaços da memória Controle.

Os bits de dados de cada instrução foram aumentados para 12. Os bits novos foram: um novo sinal de 1 bit para controle da instrução beq (selBeq), um novo sinal de 1 bit para controle da instrução blt (selBlt), um sinal de 2 bits novo para controle da entrada de PC (selPC) e a adição de mais 1 bit para o sinal “selC”, que originalmente possuía apenas 1 bit.

Todas as alterações do circuito de controle foram feitas na ROM Controle, através da função de mudança por tabela que o Digital possui. Para realizar essas mudanças eu montei a tabela verdade dos sinais de controle para cada instrução no arquivo “controle.txt”, convertendo os valores binários de cada linha em hexadecimal para aplicação na tabela do Digital.

Evidentemente também o distribuidor que recebe o sinal de controle teve de ser alterado.

4. Mudanças no arquivo “build_riscv_inst.py”:

A pasta do projeto disponibilizado pelo professor possui um arquivo Python chamado “build_riscv_inst.py”, que converte códigos Assembly para hexadecimal, em arquivos .hex que podem ser lidos pelo Digital e aplicados à memória interna. Esse código, porém, só realiza as conversões para as instruções já implementadas e às obrigatórias (sll, srl, sra, beq, blt, jal, jal), não fazendo conversões para as novas instruções da seção 2.1 do trabalho.

Então, para poder testar a instrução slt que eu havia implementado, eu alterei “build_riscv_inst.py” para que ele realizasse conversões também para slt. Foram apenas duas pequenas mudanças: na declaração do dicionário “switch” (linha 22) eu adicionei a chave ‘slt’ com valor 15 (pois a instrução slt ocupa a posição 0xF da memória de controle) e na sequência de desvios condicionais que selecionam os valores utilizados para cada instrução (a partir da linha 51) eu adicionei uma condição para a variável “opc” igual a 15 (ou seja, quando a instrução lida for slt), que então definirá “rc” igual a “instr[1]”, “ra” igual a “instr[2]”, “rb” igual a “instr[3]”.

5. Novos códigos Assembly de teste:

Eu escrevi 5 novos códigos Assembly para realizar testes em algumas etapas do desenvolvimento do meu projeto. Para fins de organização eu os juntei todos em um novo diretório chamado “novosTestes”, que está dentro do diretório “exemplos”. Cada um desses novos testes contém na primeira linha um comentário que explica brevemente a função do Assembly em questão. Assim, eu não achei necessário adicionar os códigos aqui neste relatório, e apenas irei indicar que o diretório “novosTestes” seja consultado diretamente para visualização deles. A pasta “novosTestes” também possui os arquivos .hex de cada Assembly para aplicação na MI do processador.

6. Referências:

Para referência eu utilizei apenas as anotações que eu realizei durante as aulas de Arquitetura de Computadores.