Trabalho Prático 02 – Caminho de Dados do RISC-V
CCF 252 – Organização de Computadores I

Guilherme Augusto Schwann Wilke[1], **Kayo Gregore dos Santos de Jesus**[1]

[1]Curso de Ciência da Computação – Universidade Federal de Viçosa – Campus Florestal

guilherme.wilke@ufv.br, kayo.jesus@ufv.br

Resumo. *Este Documento descreve a aplicação do segundo trabalho prático da disciplina de Organização de Computadores 1 do curso de Ciência da Computação da Universidade Federal de viçosa - Campus Florestal.*

1. Introdução

O objetivo deste trabalho consiste em implementar uma versão simplificada de um caminho de dados do processador RISC-V em Verilog (linguagem de descrição de hardware). O caminho de dados deve ler um código assembly RISC-V já traduzido em binário e executar suas instruções, após isso, deve exibir no terminal o valor contido em cada um dos registradores presentes na memória de registradores do mesmo.

Para a realização desta atividade foram utilizados recursos ensinados em sala de aula, juntamente com o auxílio de materiais encontrados em sites e fóruns na internet. O ambiente usado para desenvolvimento foi a IDE Visual Studio Code com extensões para desenvolvimento em linguagem Verilog. O compilador escolhido para compilar o código verilog foi o Icarus Verilog. Todo o trabalho foi desenvolvido e testado em um notebook com sistema operacional Arch Linux.

2. Execução do Caminho de Dados

Para executar a simulação do caminho de dados RISC-V, primeiro é necessário instalar o compilador Icarus Verilog em sua máquina Linux, o que pode ser feito ao inserir o comando **sudo apt-get install iverilog** em seu terminal de comandos caso seu sistema utilize apt-get como gerenciador de pacotes ou **sudo pacman -S iverilog** caso o gerenciador de pacotes usado seja o pacman.

Após isso, é necessário carregar os arquivos de entrada contendo as instruções RISC-V em binário para a pasta src, onde está localizado o arquivo datapath.v.

O caminho de dados está configurado para a leitura de um arquivo chamado ”inputDP.txt”, portanto, para executar outro conjunto de instruções desejado é necessário renomear o arquivo contendo o conjunto de instruções em binário para ”inputDP.txt” ou modificar dentro do arquivo ”datapath.v” a linha contendo esse nome (descrita na subseção).

Para executar a simulação do caminho de dados, é preciso (dentro da pasta src) executar o comando **make all** no terminal, que compilará o código verilog para sua execução. Após isso, para executar o arquivo compilado, é preciso executar o comando **make run** no terminal, que executará o código compilado e exibirá o resultado dos valores armazenados nos registradores após simulação de todo o código presente no arquivo de entrada.

O arquivo de entrada deve estar de acordo com as especificações RISC-V, respeitando os limites de tamanho para inteiros e a numeração dos registradores, caso contrário, o caminho de dados terá problemas para executá-lo.

3. Desenvolvimento

Para o desenvolvimento do caminho de dados modularizado, cada módulo do caminho de dados fora descrito e salvo em diferentes arquivos verilog, os quais são: **pc.v, add.v, alu.v, aluControl.v, control.v, dataMemory.v, datapath.v, immGen.v, instructionMem.v, mux.v, registerMem.v**.

Cada módulo é descrito a seguir, de acordo com seu funcionamento.

3.1. Program Counter (pc.v)

O módulo Program Counter, descrito no arquivo pc.v, recebe como entrada os fios de clock e reset do caminho de dados, assim como um fio de 32 bits que representa o endereço da instrução a ser encaminhada por output do módulo (que consequentemente será conectada à memória de instruções).

O PC basicamente verifica a cada borda de subida do clock se o sinal de reset é 0 ou 1, caso seja 0, ele encaminha para seu output o valor 0 em 32 bits, caso seja 1, ele apenas encaminha o endereço recebido por input para seu output.

3.2. Instruction Memory (instructionMem.v)

O módulo Instruction Memory, descrito no arquivo instructionMem.v, recebe por input o endereço em 32 bits da instrução a ser lida e encaminhada para os demais módulos e o nome do arquivo contendo todas as instruções em binário (deve ser uma string de até 200 bits em tamanho). O módulo encaminha por output um sinal de bit único que sinaliza o fim da execução de todas as instruções do arquivo de entrada e um valor de 32 bits representando a instrução lida atualmente no arquivo de entrada.

Dessa forma, o módulo instructionMem.v identifica quando o input que representa o nome do arquivo de entrada é modificado e usa-o para abrir esse arquivo e ler seu conteúdo, de acordo com o endereço da instrução a ser lida que fora passado para ele por seu input, cada linha é lida e encaminhada para o output do módulo toda vez que o mesmo identifica alguma mudança no valor de endereço da instrução a ser lida, além disso, caso o endereço exceda a quantidade de instruções dispostas no arquivo de entrada (ou seja, quando todas as instruções forem executadas) o módulo define o output de instrução como 0 em 32 bits e o sinal de fim de arquivo como 1.

3.3. Control (control.v)

O módulo Control é responsável por administrar todos os sinais de controle do datapath, que definem o que certos módulos irão fazer de acordo com a instrução a ser executada. Para isso, ele recebe por input os últimos 7 bits da instrução lida (correspondentes a seu opcode) e encaminha por output 7 sinais de controle, sendo 6 fios únicos e um conjunto de dois fios.

Os sinais de controles são definidos como branch (responsável por definir se a próxima instrução a ser lida é a diretamente abaixo da instrução atual ou se é alguma

outra especificada por um comando de branch), memRead (responsável por controlar se a memória de dados encaminha ou não um valor lido dentro da memória para seu output), memToReg (responsável por definir se o valor a ser escrito no registrador alvo da instrução, caso necessário, é proveniente da memória de dados ou diretamente do resultado do ALU), memWrite (responsável por controlar se o valor recebido por input na memória de dados será ou não escrito em alguma posição da memória), aluSrc (responsável por controlar se o segundo valor envolvido na operação do ALU é proveniente de um registrador ou de algum imediato presente na instrução), regWrite (responsável por controlar se o valor de 32 bits recebido por input na memória de registradores deve ou não ser armazenado em algum dos registradores) e aluOp (responsável por definir, juntamente com o funct3 da instrução a ser executada, qual operação será realizada no ALU).

Para modificar esses sinais de controle de acordo com as intruções lidas, o módulo control verifica quando o valor do opcode recebido por input é modificado e, de acordo com um conjunto de instruções case, define quais valores serão atribuídos a cada sinal.

3.4. ALU Control (aluControl.v)

O módulo ALU Control é responsável por definir qual operação será realizada no ALU de acordo com o sinal de controle aluOp, o funct3 e o funct7 da instrução RISC-V a ser executada. Para isso, o módulo aluControl recebe como input o sinal de controle de 2 bits aluOp e um valor de 4 bits correspondentes ao funct3 e funct7 da instrução a ser executada, quando qualquer um desses valores é modificado, uma função case os analisa e atribui um determinado valor de 4 bits como output do módulo, que representa qual operação será executada pelo ALU.

3.5. Register Memory (registerMem.v)

O módulo de memória dos registradores, descrito no arquivo registerMem.v, é responsável por armazenar os valores de todos os 32 registradores do caminho de dados RISC-V e administrá-los conforme as instruções lidas pela memória de instruções. O módulo recebe por input um fio de clock, o sinal de controle regWrite, 3 fios de 5 bits representando o endereço dos registradores envolvidos na instrução a ser executada e um fio de 32 bits representando a informação a ser escrita em um dos registradores, caso seja necessário. Além disso, o módulo encaminha por output os valores em 32 bits que representam os dados dos registradores lidos durante a execução da instrução (representados geralmente como rs1 e rs2).

Para armazenar o valor de todos os 32 registradores, o módulo registerMem guarda um vetor de 32 valores de 32 bits e, assim que instanciado, define o valor de todos as suas posições como 0. Uma escolha de implementação foi a de identificar toda vez que o valor armazenado no registrador de posição 0 sofrer alguma alteração, ele automaticamente define novamente seu valor para 0, uma vez que em RISC-V o registrador x0 tem valor nulo (ou zero) constante.

Para encaminhar os registradores lidos durante uma instrução, o módulo identifica sempre que os valores de endereço dos registradores a serem lidos que recebe por input ou o valor do registrador disposto nesse endereço é modificado e encaminha o novo valor do registrador desejado para seu output. É importante ressaltar que isso ocorre duas vezes, uma para cada registrador lido durante a execução da instrução (rs1 e rs2).

Além disso, para escrever um novo valor em um registrador desejado em uma instrução, o módulo verifica a cada borda de subida de clock se o sinal de controle regWrite é 1, caso seja, ele então define (de acordo com o endereço do registrador alvo especificado no input) o novo valor desse registrador como o valor de 32 bits recebido por input.

3.6. Immediate Generator (immGen.v)

O módulo Immediate Generator é responsável por receber a instrução de 32 bits da memória de instruções e identificar, de acordo com o tipo da instrução, qual o imediato presente nessa instrução. Para isso, ele recebe por input um valor de 32 bits correspondente a instrução inteira lida e encaminha por output um valor de 32 bits correspondente ao imediato presente nessa instrução.

Para fazer a identificação do imediato presente na instrução o módulo immGen verifica quando o sinal de input for modificado e usa seus últimos 7 bits (correspondentes ao opcode da instrução) para definir como o imediato está disposto dentro da instrução usando uma função case e, de acordo com o opcode identificado, reunindo os bits correspondentes do imediato de dentro da instrução para encaminhá-los em seu output. Caso o opcode identificado não for reconhecido dentro do padrão de instruções implementado, o módulo encaminha um valor x em 32 bits para seu output.

3.7. ALU (alu.v)

O módulo ALU é responsável por receber 2 valores como entrada e executar uma operação definida na instrução entre eles. Para isso, o módulo recebe como entrada dois valores de 32 bits e um sinal de 4 bits proveniente do módulo aluOp, que representa qual operação será executada, quando qualquer um desses valores é alterado, o alu verifica qual instrução é especificada pelo sinal de 4 bits e a executa entre os valores de entrada, atribuindo seu resultado a um valor de 32 bits em seu output. Além disso, para instruções de branch, há um sinal de output de apenas um bit que identifica quando o resultado da operação é igual a zero e define o valor 1 nessa condição, cujo valor é 0 em qualquer outra.

3.8. Data Memory (dataMemory.v)

O módulo Data Memory é responsável por guardar os valores da memória de dados do caminho de dados RISC-V, para isso, ele armazena um vetor de 512 posições (valor escolhido durante a implementação) de 32 bits cada. Além disso, o módulo recebe como input os sinais de clock, memRead e memWrite e dois valores de 32 bits, correspondentes ao endereço da memória que será manipulada e os dados a serem escritos na memória, caso necessário.

Quando qualquer um dos inputs do módulo é modificado, a memória verifica o valor do sinal de controle memRead e, caso esse seja 1, atribui o valor correspondente ao valor armazenado no endereço especificado pelo input ao seu output.

Para escrever valores dentro da memória de acordo com as instruções executadas, o módulo verifica a cada borda de descida de clock se o sinal de controle memWrite é positivo e atribui o valor especificado em um de seus inputs à posição de endereço de memória especificado em um de seus inputs.

3.9. Adder (add.v)

O módulo Adder apenas recebe dois valores de 32 bits como input e atribui a soma desses valores a seu output, também de 32 bits.

3.10. Multiplexador (mux.v)

O módulo Multiplexador recebe dois valores de 32 bits e um sinal de controle como input. Caso esse sinal de controle seja 0, o módulo atribui o primeiro valor de 32 bits ao seu output, mas, caso esse sinal de controle seja 1, ele atribui o segundo valor de 32 bits a seu output.

3.11. Datapath (datapath.v)

O módulo Datapath é o módulo responsável por reunir todos os outros módulos e conectá-los, formando o caminho de dados RISC-V em si. Esse módulo recebe como input um sinal de clock e um sinal de reset, além de um valor de 200 bits correspondente ao nome do arquivo de entrada contendo todas as instruções RISC-V em binário. Além disso, o módulo datapathR5 tem como output um fio de 1 bit que identifica quando o caminho de dados terminou a execução de todas as instruções do arquivo de entrada.

É importante ressaltar que esse módulo cria 2 instâncias de módulos Adders e 3 instâncias de módulos Multiplexadores. As instâncias de módulos Adders correspondem as adições do endereço da instrução sendo atualmente executada pelo caminho de dados com o valor 4 (que a transforma no endereço da instrução imediatamente abaixo dela) ou com o valor definido por alguma instrução de branch. As instâncias de módulos Multiplexadores correspondem ao controle de decisão de quais valores de entrada utilizar nos módulos ALU, Register Memory e PC de acordo com os sinais de controle do caminho de dados.

Dentro do arquivo que descreve o módulo datapathR5, também está descrito um módulo de testbench que executa o teste do caminho de dados utilizando algum arquivo especificado nele como arquivo de entrada das instruções RISC-V. Esse módulo testbench inicia com valores clock e reset definidos como 0, e, a cada uma unidade de tempo o sinal de clock é invertido, ou seja, se era 0 vira 1 e vice-versa, além disso, após 5 unidades de tempo a partir do início do testbench, o sinal de reset é definido como 1, marcando assim o começo da execução das instruções RISC-V contidas no arquivo de entrada.

Assim que o sinal de output do módulo datapath correspondente ao fim da execução de todas as instruções do arquivo de entrada é definido como 1, o testebench exibe no terminal os valores armazenados em cada um dos registradores da memória de registradores do caminho de dados tanto em sua forma decimal quanto em sua forma binária (consultar a seção Resultados para um exemplo). Após isso, o testbench é finalizado e a execução do caminho de dados é encerrada.

É importante destacar que o nome do arquivo de entrada é definido assim que o módulo testbench é iniciado, de acordo com a linha do arquivo verilog que o representa, essa decisão de implementação permite ao usuário modificar qual arquivo de entrada ele deseja executar com facilidade diretamente do arquivo datapath.v.

4. Resultados

Para verificação de resultados, foram utilizados 2 arquivos de entrada contendo instruções RISC-V em binário diferentes, chamados inputDP.txt e inputDP2.txt, a execução de cada um deles está documentada a seguir.

Após inserido o arquivo inputDP.txt (Figure 1) na pasta src e executado os comandos make all (para compilar o código) e make run (para execução do código), foi registrado o resultado no terminal exibido na Figure 2. Em seguida, ao inserir o arquivo inputDP2.txt (Figure 4) na pasta src, modificar a linha do arquivo datapath.v correspondente ao nome do arquivo de entrada e executar os comandos make all (para compilar o código) e make run (para execução do código), registrado o resultado no terminal exibido na Figure 5.

Para verificar se os resultados estão corretos, eles foram comparados com resultados registrados ao executarmos os mesmos conjuntos de instruções no simulador online contido na especificação do trabalho, que estão exibidos, respectivamente, nas imagens Figure 3 e Figure 6.

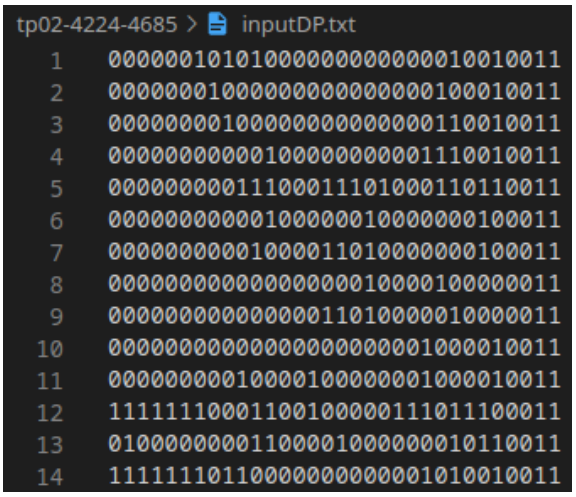


Figure 1. Arquivo de entrada inputDP.txt

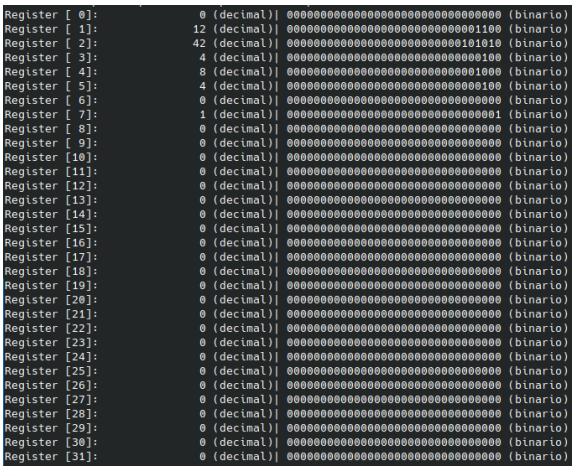


Figure 2. Resultado no terminal da execução das instruções contidas no arquivo inputDP.txt

Init Value	Register	Decimal	Hex	Binary
0	x0 (zero)	0	0x00000000	0b00000000000000000000000000000000
0	x1 (ra)	12	0x0000000c	0b00000000000000000000000000001100
0	x2 (sp)	42	0x0000002a	0b00000000000000000000000000101010
0	x3 (gp)	4	0x00000004	0b00000000000000000000000000000100
0	x4 (tp)	8	0x00000008	0b00000000000000000000000000001000
0	x5 (t0)	4	0x00000004	0b00000000000000000000000000000100
0	x6 (t1)	0	0x00000000	0b00000000000000000000000000000000
0	x7 (t2)	1	0x00000001	0b00000000000000000000000000000001
0	x8 (s0/fp)	0	0x00000000	0b00000000000000000000000000000000
0	x9 (s1)	0	0x00000000	0b00000000000000000000000000000000
0	x10 (a0)	0	0x00000000	0b00000000000000000000000000000000
0	x11 (a1)	0	0x00000000	0b00000000000000000000000000000000
0	x12 (a2)	0	0x00000000	0b00000000000000000000000000000000
0	x13 (a3)	0	0x00000000	0b00000000000000000000000000000000
0	x14 (a4)	0	0x00000000	0b00000000000000000000000000000000
0	x15 (a5)	0	0x00000000	0b00000000000000000000000000000000
0	x16 (a6)	0	0x00000000	0b00000000000000000000000000000000
0	x17 (a7)	0	0x00000000	0b00000000000000000000000000000000
0	x18 (s2)	0	0x00000000	0b00000000000000000000000000000000
0	x19 (s3)	0	0x00000000	0b00000000000000000000000000000000
0	x20 (s4)	0	0x00000000	0b00000000000000000000000000000000

Figure 3. Resultado registrado no simulador online (apenas os 20 primeiros registradores) para a execução das instruções contidas no arquivo inputDP.txt

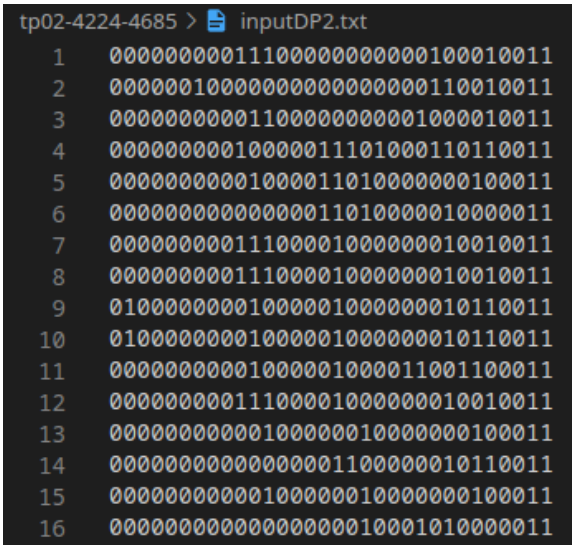


Figure 4. Arquivo de entrada inputDP2.txt

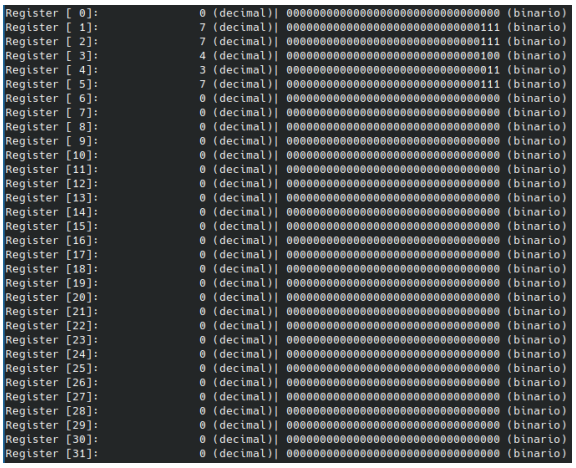
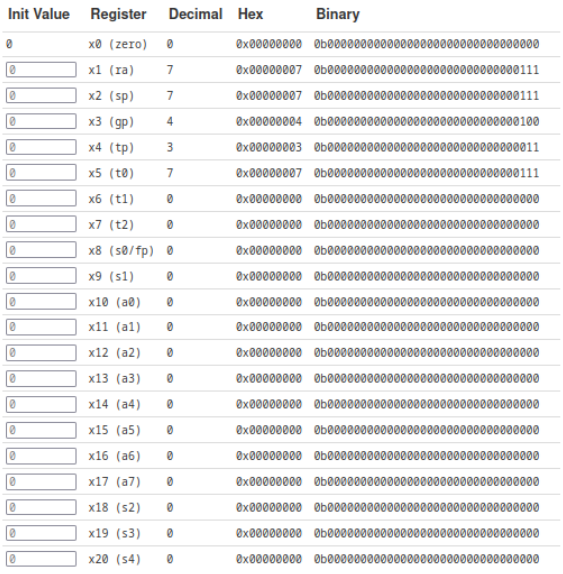


Figure 5. Resultado no terminal da execução das instruções contidas no arquivo inputDP2.txt



Init Value	Register	Decimal	Hex	Binary
0	x0 (zero)	0	0x00000000	0b00000000000000000000000000000000
0	x1 (ra)	7	0x00000007	0b00000000000000000000000000000111
0	x2 (sp)	7	0x00000007	0b00000000000000000000000000000111
0	x3 (gp)	4	0x00000004	0b00000000000000000000000000000100
0	x4 (tp)	3	0x00000003	0b00000000000000000000000000000011
0	x5 (t0)	7	0x00000007	0b00000000000000000000000000000111
0	x6 (t1)	0	0x00000000	0b00000000000000000000000000000000
0	x7 (t2)	0	0x00000000	0b00000000000000000000000000000000
0	x8 (s0/fp)	0	0x00000000	0b00000000000000000000000000000000
0	x9 (s1)	0	0x00000000	0b00000000000000000000000000000000
0	x10 (a0)	0	0x00000000	0b00000000000000000000000000000000
0	x11 (a1)	0	0x00000000	0b00000000000000000000000000000000
0	x12 (a2)	0	0x00000000	0b00000000000000000000000000000000
0	x13 (a3)	0	0x00000000	0b00000000000000000000000000000000
0	x14 (a4)	0	0x00000000	0b00000000000000000000000000000000
0	x15 (a5)	0	0x00000000	0b00000000000000000000000000000000
0	x16 (a6)	0	0x00000000	0b00000000000000000000000000000000
0	x17 (a7)	0	0x00000000	0b00000000000000000000000000000000
0	x18 (s2)	0	0x00000000	0b00000000000000000000000000000000
0	x19 (s3)	0	0x00000000	0b00000000000000000000000000000000
0	x20 (s4)	0	0x00000000	0b00000000000000000000000000000000

Figure 6. Resultado registrado no simulador online (apenas os 20 primeiros registradores) para a execução das instruções contidas no arquivo inputDP2.txt

5. Conclusão

O trabalho foi devidamente executado e implementado de acordo com o especificado, cumprindo todos os requisitos da primeira entrega e executando corretamente todas as 7 instruções designadas ao grupo.

A execução e implementação do trabalho contou com o uso da linguagem de descrição de hardware Verilog, juntamente com o uso do ambiente de desenvolvimento VSCode, assim como o uso da plataforma GitHub para versionamento de código.

O programa foi devidamente testado e roda normalmente sob as circunstâncias descritas na seção **Execução do Programa**.

No geral, a execução do trabalho auxiliou os integrantes do grupo a fixar os conhecimentos da matéria de Organização de Computadores 1, assim como colocar em prática as atividades vistas em sala.

6. Referências

Documentação fornecida sobre o trabalho.

Patterson, D.A.; Hennessy J.L. Computer Organization and Design: RISC-V Edition, 2ª Edição, Editora Morgnan Kaufman, 2021.