Trabalho Prático - Organização de Computadores 2

DCC/UFMG - Prof. Omar - 2021/1

Integrantes:

Arthur Souto Lima
Pablo Correa Costa
Hilário Corrêa da Silva Neto
Victor Vieira Brito Amaral Pessoa

1 - Introdução

Este trabalho visa implementar, na linguagem Verilog, duas CPU Risc-V superescalares, paralelas e independentes (em módulos separados). Pode-se considerar que o projeto é dividido em duas partes. Numa primeira, a implementação sugerida pelo professor de um processador Risc-V, bem como um projeto em verilog encontrado pelo grupo de uma CPU MIPS com superescalar, foram os pontos de partida para transformar o processador fornecido em um superescalar do tipo I4. Na segunda, esta implementação do I4 é utilizada e adaptada, com relativamente poucas alterações, para a criação de um superescalar de tipo I2O2.

Na primeira parte, o trabalho principal consistiu em implementar um scoreboard independente, um módulo issue e na adaptação do datapath da execução para as unidades paralelas para instruções de soma, de multiplicação e de leitura e escrita na memória. Já na segunda, foram necessárias, além da eliminação de módulos desnecessários que apenas sincronizavam os pipelines, algumas alterações que permitiriam o writeback (e o commit, se pensarmos na divisão típica de funções de superescalares, apesar de, em I2O2, não haver distinção entre os dois passos) fora de ordem. Isso foi feito utilizando um encaminhamento direto entre o último submódulo de cada caminho onde a operação era, efetivamente, calculada e o módulo de writeback. No caso das operações de leitura e escrita, o submódulo M1 já poderia realizar o encaminhamento, encerrando a execução em dois ciclos de clock, e, no das operações de soma, o módulo X0 já era terminal. Os módulos necessários para a efetuação de multiplicações não sofreram alterações. Por fim, o scoreboard foi devidamente adaptado tendo em vista a nova disposição dos módulos. As seções 3 e 4 anexas descrevem detalhadamente estes passos.

2 - Links Úteis

O código dos processadores, também entregue juntamente com este relatório, se encontra disponível num repositório GitHub[1]. Além disso, para maior comodidade,

[1] https://github.com/arthursl12/TP_OC2_

os testes estão apresentados também num notebook Colab[2], possíveis de serem executados com auxílio da extensão sugerida nas orientações do trabalho [1].

3 - Implementação

(Decisões de implementação, alterações feitas, falar dos estágios novos da execução, falar do issue...)

3.1 - Fluxo de Trabalho

Este projeto foi desenvolvido inicialmente segundo a sugestão da orientação dada, programando em células de notebooks Colab e executando os códigos por meio da extensão referenciada [1]. Contudo, com o avançar do projeto, os módulos se tornaram demasiado grandes, o que inviabilizou a continuidade dessa abordagem. Portanto, o projeto passou a ser desenvolvido localmente nas máquinas dos integrantes e sincronizado via repositório compartilhado no GitHub.

Para essa compilação e execução local, seguimos o exato mesmo processo feito na extensão [1], isto é, instalamos o Icarus Verilog [2] no ambiente. Com isso, foi possível compilar os processadores e suas testbenches. Em seguida, com os códigos assembly *vvp* gerados, podemos executá-los para obter as formas de onda em arquivos *vcd*. Finalmente, a fim de aferir os resultados, pode-se usar um programa visualizador desse tipo de informação a partir desses arquivos. No caso, foi utilizado o GTKWave [3], que possui uma interface gráfica intuitiva e simples de ser usada. Em momentos adiante, os testes apresentados foram obtidos dessa maneira.

3.2 - Apresentação no Colab

Com intuito de propiciar mais comodidade para aqueles que desejam ter contato o projeto, há a disponibilidade de um Colab[3] que perfaz os passos supracitados nesse ambiente. A extensão [1] instala o Icarus Verilog. Além disso, clonamos o repositório com o código-fonte dos processadores em uma das células para que eles possam ser compilados e executados. De posse das formas de onda, utilizamo-nos da visualizador das formas de onda da extensão para exibir os resultados dos testes.

3.3 - Módulo Fetch

O módulo Fetch, representado no arquivo *Fetch.v*, é responsável pela busca e leitura das instruções no módulo de memória. Ele passa as informações da instrução para o bloco Decode (*if_id_instruc*) e atualiza o Program Counter (*if_id_nextpc*) com PC = PC+4.

---

[2] https://colab.research.google.com/drive/15bMN5D__goXvq7orqdB6rHpDwjzQg35g?usp=sharing
[3] https://colab.research.google.com/drive/15bMN5D__goXvq7orqdB6rHpDwjzQg35g?usp=sharing

3.4 - Módulo Decode

O módulo Decode, no arquivo *Decode.v*, fica responsável pelo funcionamento de instruções de desvio e pela decodificação das outras instruções do RISC-V. Ele recebe o módulo de Controle *Control.v*, do qual recebe assincronamente os endereços de memória.

3.5 - Módulo de Controle

O módulo de controle é um dos mais importantes do processador. É ele quem envia sinais de gerenciamento para os outros módulos. Em termos de implementação, ele possui seu próprio arquivo, o *Control.v*, mas é criado como submódulo do módulo Decode. Algoritmicamente, temos um comando que casa padrões da instrução, ou seja, uma string de bits que amalgama *funct7*, *func3* e *opcode* obtidos da instrução e cada padrão gera uma série de sinais específicos para os outros módulos do processador. São exemplos de sinais como o código de operação da ALU, quantidade de operandos, isto é, se a operação usa imediato ou não, e se haverá escrita de registradores ao final.

3.6 - Módulo Issue

A primeira coisa que o módulo *Issue.v* faz é receber o endereço dos registradores do Decode e comunica com o ARF (Architecture Register File) da mesma forma que o *Decode.v* faz. Todos esses dados são necessários para a implementação dos módulos Scoreboard e Hazard Detector, presentes nos arquivos *Scoreboard.v* e *HazardDetector.v* respectivamente. Com isso, ele realiza a detecção dos Hazards no *HazardDetector.v* com auxílio da tabela do *Scoreboard.v* e envia as instruções corretas para as unidades de execução.

3.7 - Módulo Writeback

No *Writeback.v* ocorre o recebimento dos sinais e os tipos das operações feitas no módulo de Execução. Ele interpreta qual operação recebe e decide onde no ARF vai escrevê-la, quando for necessário.

3.8 - Scoreboard

O Scoreboard, presente no módulo Issue, possui um módulo próprio *Scoreboard.v*, onde recebe dos estágios Decode e Issue as informações dos registradores que serão operados e alguns sinais de controle anti-hazards para serem usados no Hazard Detector.

O módulo propriamente consiste na montagem de uma "tabela" que armazena, para cada um dos registradores, se existe alguma instrução com escrita pendente nesse registrador em específico. Isso indicará que é preciso realizar um encaminhamento. Caso exista alguma instrução pendente, ele armazena a unidade

funcional na qual essa instrução está e seu estágio, apontando onde realizar o encaminhamento.
Nas colunas remanescentes, os bits são marcadores informando a presença ou não de alguma instrução em determinado estágio do pipeline do módulo de execução ou se a mesma já está sendo encaminhada ao write back.
Com o virar do clock e portanto movimentação das instruções pelo pipeline os bits são 'shiftados' para a direita, atualizando assim a movimentação das informações. Este processo se repete a cada ciclo de clock e nos registrados onde stalls não ocorrem o 'shifta' acontece até ocorrer dos bits informativos sumirem, o que por sua vez significaria que a escrita pelo writeback já aconteceu e portanto o registrador não mais está pendente.

3.9 - Hazard Detector

HazardDetector.v é um módulo que recebe os registradores do Decode e do Issue e lê o Scoreboard para determinar se ele deve fazer alguma alteração em caso de Hazard. Essa alteração pode ser um stall a ser incluído antes ou depois do Issue assincronamente.

4 - Alterações na Execução

O estágio de Execução, presente entre o Issue e o WriteBack, foi dividido em 3 módulos: Multiplicação, Operações em Memória e outras operações de ALU. Na implementação, foram criados os módulos *Mult.v, Mem.v* e *Alu.v* para representar cada uma das partes, respectivamente.

4.1 - Multiplicação

O estágio de multiplicação é o mais crítico dos caminhos de execução. Por ser o único com quatro subestágios, representa o pior caso dentro do conjunto de instruções implementado. No caso do processador I4, ele é o determinante da duração de todas as demais instruções, implementadas com subestágios internos desnecessários para permitir a execução em ordem do processador. Já no I2O2, é ele quem causa a maior parte das inversões de ordem de término de instruções - e seus possíveis desdobramentos desastrosos.
A divisão em quatro módulos internos é unida por um módulo externo, que tem como função acionar o primeiro dos módulos (M0) e inicializar os valores das entradas do mesmo, além de conectar os demais módulos.
O primeiro módulo, Y0, tem como função lógica apenas decidir se o resultado da multiplicação será positivo, negativo ou zero, e o faz utilizando a função signal do verilog, para verificação dos sinais dos operandos, e comparando cada um deles com o literal zero.
O segundo, Y1, tem como função transformar cada operando negativo da multiplicação e transformá-los em positivos, para fins de permitir um circuito de

multiplicação mais simples. Ele faz isso checando se cada operando é menor que zero e, se for, inverte todos os bits do valor e soma um (visto que inteiros, em Risc-V, são representados em complemento de dois).
No terceiro, Y2, a multiplicação em si é efetuada, utilizando o operador * do verilog, e o resultado é armazenado num registrador interno de 64 bits. Todos os 64 são transmitidos para o módulo final.
No quarto, por fim, é feita a checagem de overflow, a possível correção de sinal (caso o resultado seja negativo) e a formatação da saída para a escrita no banco de registradores. Primeiro, na entrada de 64 bits vinda de Y2, os 32 bits menos significativos são armazenados em um barramento interno e os 32 mais significativos, mais o 33º (note que ele corresponde ao bit mais significativo dos 32 de baixo, mas, como a entrada está, nesse circuito, ainda obrigatoriamente positiva, se ele for igual a 1 já é indício de que houve overflow). O barramento com os 33 bits mais significativos é comparado com um barramento de mesmo tamanho que guarda o literal zero. Caso eles não sejam iguais, é detectado o overflow e a operação de escrita no banco de registradores não acontece. Após a checagem de overflow, caso o sinal calculado em Y0 para o resultado da multiplicação seja negativo, os 32 bits mais baixos são tratados como um número cujo sinal será trocado, em procedimento idêntico ao realizado em Y1. Em caso de não ter ocorrido overflow, os 32 bits mais baixos serão encaminhados para a saída do módulo, junto do endereço do registrador de destino e do sinal que determina se haverá ou não escrita no writeback, e o valor será guardado.

4.2 - Memória

O módulo de memória foi refinado em um módulo *Mem.v* que integra dois submódulos *Mem_0.v* e *Mem_1.v*. No processador I4, em tese, existiriam mais dois módulos de memória para que se adequasse ao modelo em ordem desse processador, porém no código a espera do Clock é feita no próprio *Mem.v* sem a criação de módulos adicionais para esses dois estágios, já que não há outra funcionalidade para eles.
O arquivo *Mem_0.v* recebe o sinal do processamento do Issue e realiza o cálculo do endereço de memória e passa o resultado para o bloco *Mem_1.v*.
O arquivo *Mem_1.v*, por sua vez, realiza a escrita ou a leitura, quando necessário, dos dados no endereço recebido.

4.3 - Operações da ALU

O módulo da ALU possui apenas um estágio relevante *Alu.v*, no qual são executadas as operações Lógico-Aritméticas mais simples: Adição, Subtração, AND, OR entre outras. No processador I4, os 3 estágios de passagem de sinal, que não possuem valor significativo e que são removidos no I2O2, são tratados no arquivo *AluMisc.v*.

5 - Testes

O método de execução e aferição dos testes já foi explicitado em seções anteriores. Outra consideração a ser feita é que os processadores devem receber as instruções em hexadecimal, logo há a necessidade de utilizarmos um montador RISC-V para que transformemos da linguagem próxima à natural assembly em código que a máquina consiga compreender e interpretar.

Felizmente, dentre as orientações fornecidas para esse projeto, foi disponibilizado um montador RISC-V [4], o qual auxiliou na testagem dos processadores como um todo. Para ele fornecemos os códigos assembly dos programas de teste e ele nos retornava as instruções, por linha, em binário em um arquivo e em hexadecimal em outro. Dado os nossos processadores, usamos as instruções em hexadecimal.

Neste relatório, os resultados dos testes serão apresentados aqui via prints de uma lista de sinais exibidos pelo GTKWave [3]. As instruções assembly que foram fornecidas para os processadores em cada teste também estarão aqui. O leitor é convidado a consultar o Colab[4], para que não tenha a necessidade de executar localmente os testes.

5.1 - Teste 1

O primeiro teste é uma única instrução de soma com imediato. Ela é importante para mostrar o funcionamento básico dos processadores.

```
addi x10, x0, 3
```

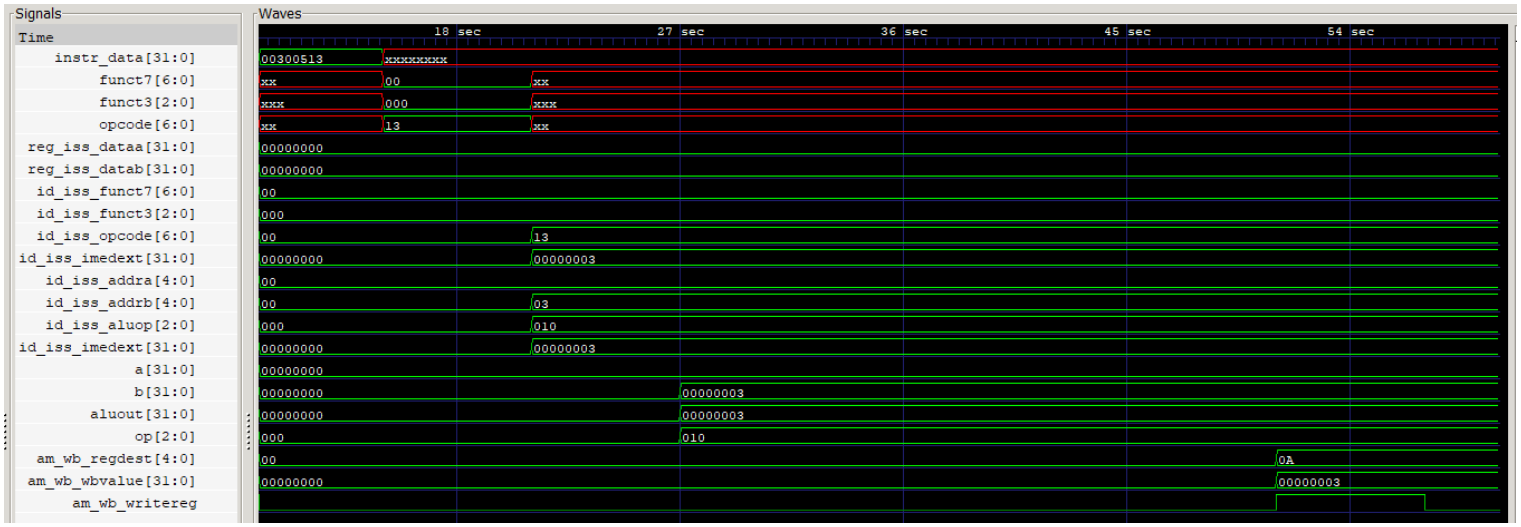

Figura 1 - Teste 1 no I4

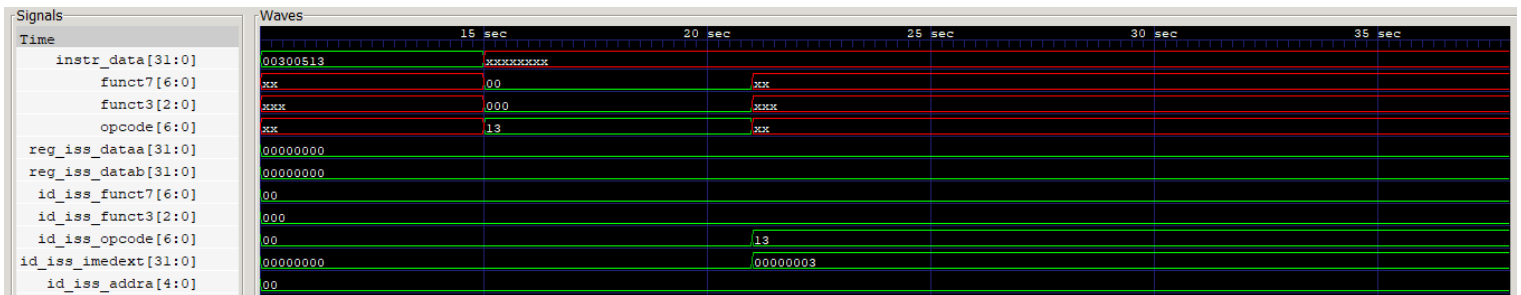

Figura 2 - Teste 1 no I2O2

Nesse teste, é possível ver a instrução sendo processada em cada etapa: ela sendo lida no Fetch, tendo suas informações extraídas no Decode, sendo passada para o Issue que faz um processamento para avaliar para qual unidade funcional de execução a execução será enviada, ela é calculada efetivamente na ALU e, finalmente, seja escrito no registrador *x10* no Writeback.

No I4, vemos, ainda, claramente um "gap" entre a ALU e o Writeback, que representa, exatamente, os outros três estágios do estágio de execução presentes nesse processador para esse tipo de instrução. Como já comentado em outros momentos, esses três estágios não fazem qualquer computação, apenas repassam sinais a cada ciclo.

5.2 - Teste 2

Esse segundo teste é uma evolução natural do primeiro, agora com mais instruções apenas.

```
addi x10, x5, 7
addi x11, x10, 8
add x12, x10, x11
```

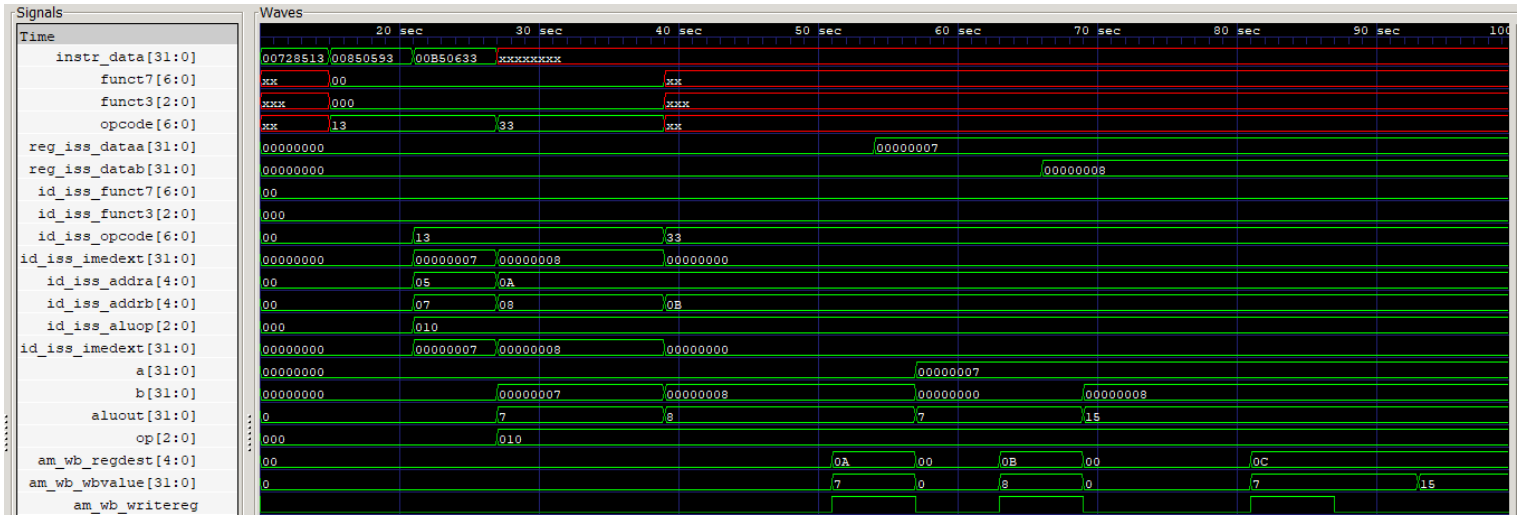

Figura 3 - Teste 2 no I4

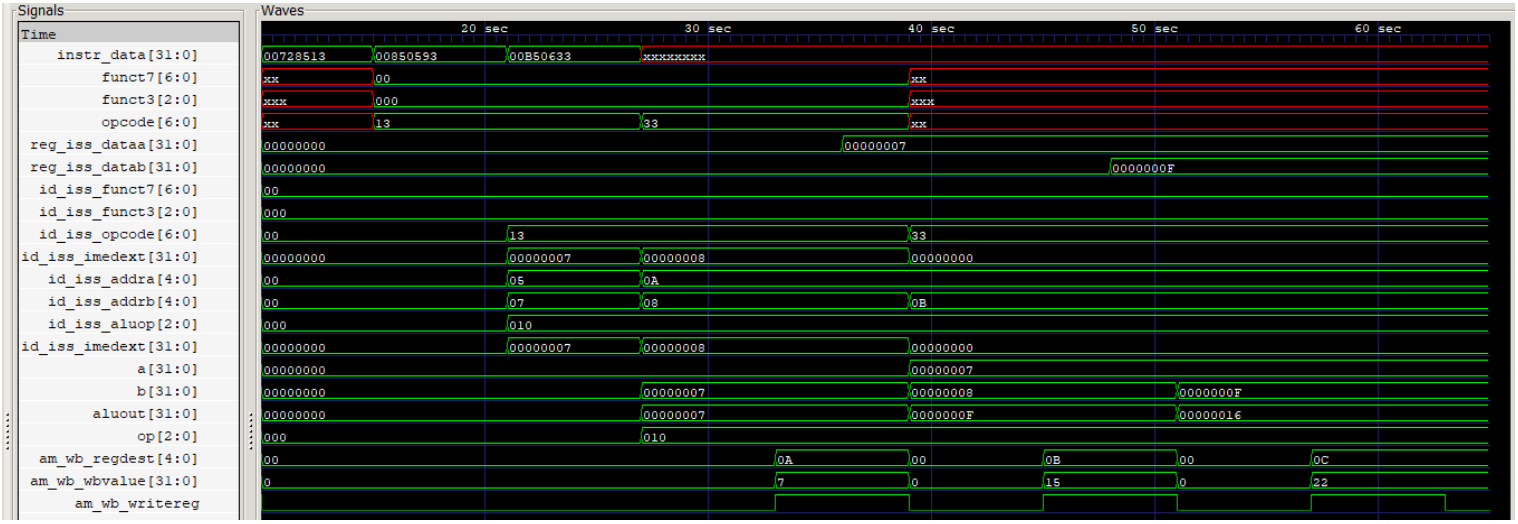

Figura 4 - Teste 2 no I2O2

5.3 - Teste 3

Para testar o segundo caminho de dados da execução, temos que executar instruções de *load* e *store*. Exatamente esse é o objetivo desse terceiro teste. Aqui temos mais instruções pois é mais interessante salvar e recuperar valores diferentes de zero, por isso a presença também de somas.

```
addi x13, x0, 3
addi x14, x0, 0
lw x10, 0(x14)
lw x11, 4(x14)
add x12, x11, x10
sw x13, 12(x0)
sw x12, 8(x0)
```

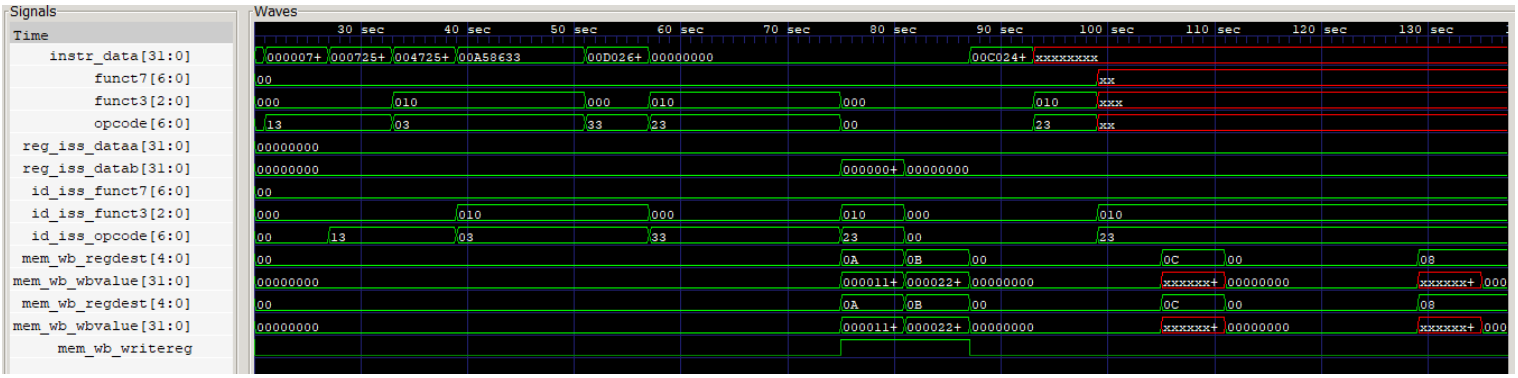

Figura 5 - Teste 3 no I4

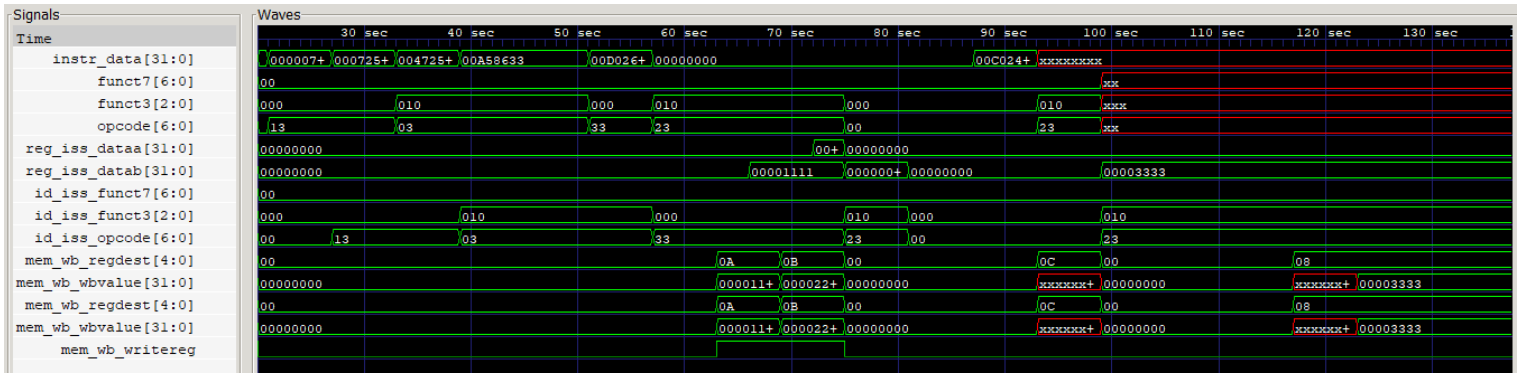

Figura 6 - Teste 3 no I2O2

O interessante nesse teste, além das formas de onda, nas quais vemos as informações sendo repassadas para os módulos de memória e ali computadas, é observar as memórias de dados em si, comparando o antes e o depois.

As memórias de dados também são incorporadas de início nas testbenches desse teste 3 e, ao final, é feito uma exportação delas. Assim, esses resultados estão disponíveis para conferência nos arquivos das testbenches e dos resultados. A memória inicial está na pasta das testbenches, no arquivo *riscv_tb3_lw_reg_data.hex*, enquanto o seu estado final está na pasta dos resultados, ou seja, *build*, no arquivo *riscv_tb3_lw_reg_data_out.hex*.

Para consolidar, exibiremos a comparação para o processador I2O2:

```
1   // 0x00000000
2   00001111
3   00002222
4   xxxxxxxx
5   xxxxxxxx
6   xxxxxxxx
7   xxxxxxxx
8   xxxxxxxx
9   xxxxxxxx
10  xxxxxxxx
11  xxxxxxxx
12  xxxxxxxx
13  xxxxxxxx
14  xxxxxxxx
15  xxxxxxxx
16  xxxxxxxx
17  xxxxxxxx
18  // 0x00000010
19  xxxxxxxx
20  xxxxxxxx
21  xxxxxxxx
```

Figura 7 - Memória Inicial

```
1   // 0x00000000
2   00001111
3   00002222
4   00003333
5   00000003
6   xxxxxxxx
7   xxxxxxxx
8   xxxxxxxx
9   xxxxxxxx
10  xxxxxxxx
11  xxxxxxxx
12  xxxxxxxx
13  xxxxxxxx
14  xxxxxxxx
15  xxxxxxxx
16  xxxxxxxx
17  xxxxxxxx
18  // 0x00000010
19  xxxxxxxx
20  xxxxxxxx
21  xxxxxxxx
```

Figura 8 - Memória Final,

após o Teste 3 no I2O2

As instruções leem as duas primeiras posições de memória, ou seja, as posições 0 e 4 a partir do endereço 0 (guardado em x14) e soma os valores lidos. O resultado dessa adição é guardada na terceira posição (12) desse mesmo endereço. No quarto endereço guardamos uma soma de imediato que resultou em 3. É interessante notar que essas duas últimas operações foram feitas na ordem contrária, indicando que o processador consegue calcular corretamente as posições de memória a serem escritas.

No teste 3 do I4, o resultado é quase o mesmo, exceto na linha 4, em que, por conta de um erro ainda não solucionado, o processador não está sendo capaz de ler e somar corretamente os valores das duas primeiras posições de memória e, após isso, guardar a soma novamente. Ele está guardando 0, no momento.

5.4 - Teste 4

No quarto e último teste de funcionamento geral dos processadores, vamos avaliar o módulo de multiplicação, o terceiro caminho de dados de execução disponível.

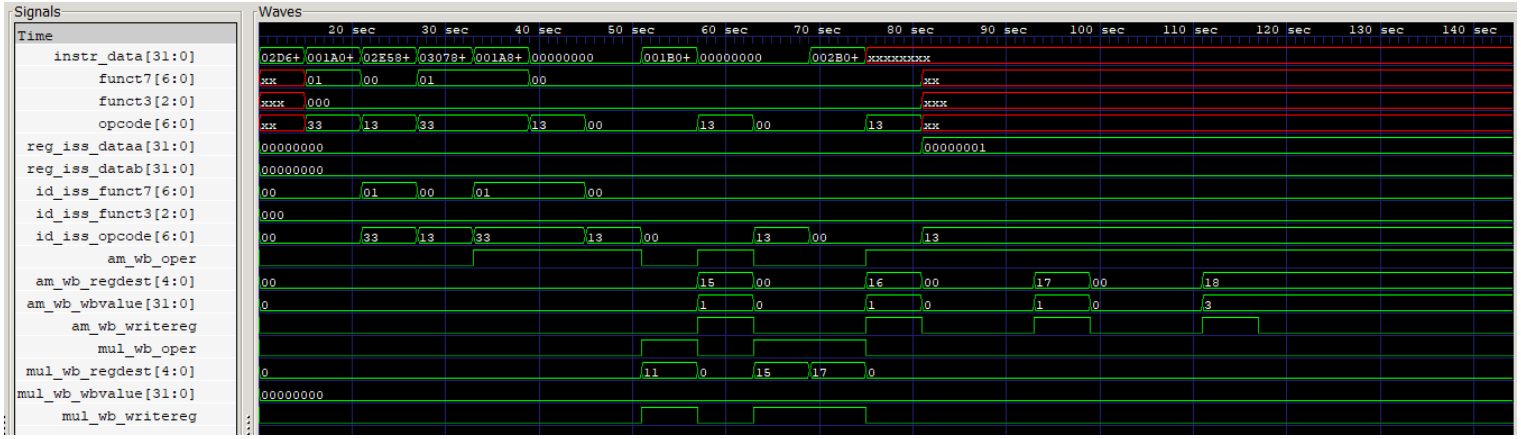

Figura 9 - Teste 4 no I4

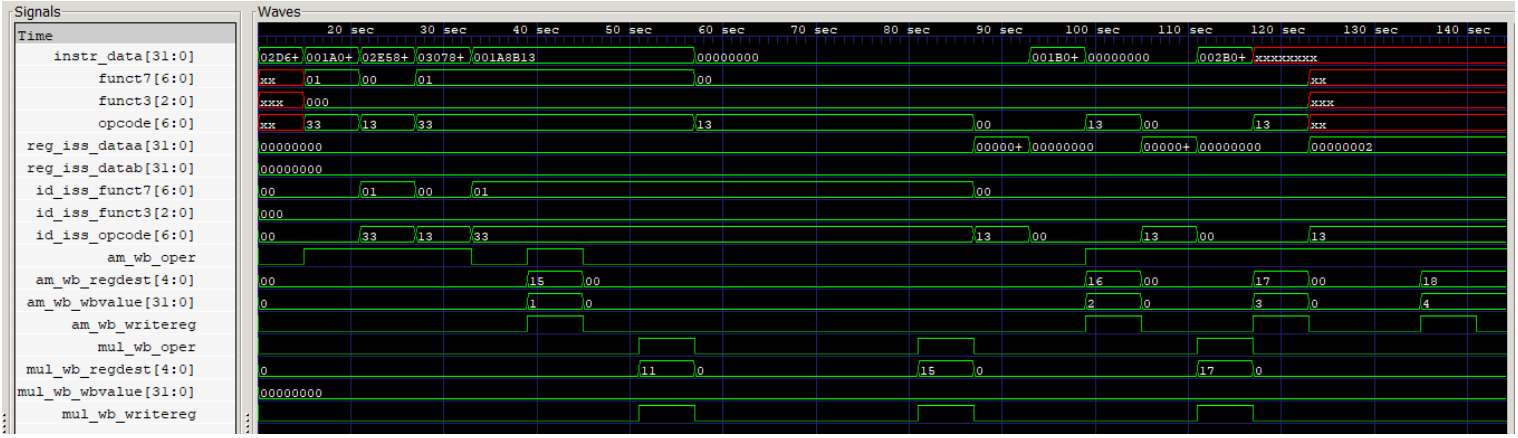

Figura 10 - Teste 4 no I2O2

Nos testes, vemos que há o sinal de operação de multiplicação ativo, o que indica que o módulo *Issue* conseguiu identificar corretamente as instruções desse tipo e passar os dados para os módulos corretos de execução. Eles fazem o processamento e repassam para o writeback consolidar os resultados dos cálculos.

6 - Conclusão

Ao final do projeto, é possível tecer algumas considerações finais. Nota-se que o fluxo de desenvolvimento de um trabalho em Verilog guarda diferenças significativas com relação a de um em linguagens de mais alto nível. Pode-se elencar a linha de raciocínio, mais voltada para o hardware com seus ciclos de clock, a compilação e depuração, que necessitam de passos intermediários não tão comuns em outras situações de programação.

Em termos de plataformas de desenvolvimento, o ambiente Colab, com auxílio das extensões sugeridas [1], se mostrou muito profícuo à primeira vista. Contudo, com o crescimento do projeto, foi necessário um ferramental de controle de versões e de poder de edição mais aprimorado. Portanto, num segundo momento o desenvolvimento foi local usando o Icarus Verilog [2]. Apesar disso, é possível apresentar os resultados e executar os processadores sem quaisquer problemas nesse ambiente, como já citado em outras seções.

Finalmente, convém ressaltar a ímpar oportunidade de se debruçar sobre um projeto de tal magnitude, mesmo que os objetivos não tenham sido plenamente atingidos. Todo o estudo e preparação extraclasse proporcionados por esta tarefa servirão certamente de grande experiência para situações futuras com Verilog e relacionadas à Organização de Computadores como um todo.

7 - Referências

[1] Canesche, Michael and Braganca, Lucas and Neto, Omar Paranaiba Vilela and Nacif, Jose A and Ferreira, Ricardo. **Google Colab CAD4U: Hands-on Cloud Laboratories for Digital Design**. In: 2021 IEEE International Symposium on Circuits and Systems (ISCAS). Pág 1-5. 2021.

[2] Stephen Williams, **Icarus Verilog**. https://github.com/steveicarus/iverilog

[3] **GTKWave**. Código-fonte: https://github.com/gtkwave/gtkwave. Binários: http://gtkwave.sourceforge.net/

[4] **RISC-V Assembler**. https://github.com/kcelebi/riscv-assembler