Relatório - Tomasulo

Projeto de OAC II

Allan Almeida Santos (8993497), Pedro Donini Linan (8993048),
Pedro Duarte Bairao (8992840) e Henrique Costabile Filho (8992670)

7 de novembro de 2019

1 Introdução

Mesmo após o tempo de acesso ao armazenamento ter sido satisfatoriamente reduzido através do uso de técnicas de buffer e sobreposição, e a unidade de instrução ter sido canalizada para operar a uma taxa próxima de uma instrução por ciclo, permaneceu a necessidade de otimizar o desempenho real de operações aritméticas, especialmente de ponto flutuante. Na tentativa de equilibrar a execução com a emissão enfrenta-se dois problemas principais. Primeiro, operações individuais não são rápidas o suficiente para permitir a execução serial simples. Segundo, é difícil conseguir a execução mais rápida possível em uma unidade de execução universal. Em outras palavras, os circuitos projetados para multiplicar e adicionar não fariam nenhuma função tão rápido quanto duas unidades, cada uma limitada a um tipo de instrução.

O primeiro passo para superar esses obstáculos foi a divisão da função de execução em duas partes independentes, uma área de execução de ponto fixo e uma área de execução de ponto flutuante. Isso alivia a restrição física e torna a execução paralela possível.

O assunto deste projeto, então, é o método usado para obter execução de instruções de ponto flutuante no IBM System / 360 Model 91 que lida com múltiplas unidades de execução, conhecido como algoritmo de Tomasulo.

Pode parecer que alcançar a operação simultânea de duas unidades (adição e multiplicação) não difere substancialmente da sobreposição fixa-flutuante. No entanto, neste último caso a arquitetura limita cada uma das classes de instrução ao seu próprio conjunto de acumuladores e isso garante independência. No primeiro caso, existe apenas um conjunto de acumuladores, o que implica em sequências especificadas pelo programa de operações dependentes. Agora não é mais simplesmente uma questão de classificar cada instrução como ponto fixo ou ponto flutuante, uma classificação que independia das instruções anteriores. Entretanto, é uma questão de determinar o relacionamento de cada instrução com todas as instruções anteriores incompletas. Simplificando, o objetivo deve ser preservar as precedências essenciais, permitindo a maior sobreposição possível de operações independentes.

Este objetivo é alcançado com Tomasulo através de um barramento de dados comum (CDB). Ele torna possível simultaneidade máxima com o mínimo esforço (geralmente nenhum) pelo programador ou, mais importante, pelo compilador. Ao mesmo tempo, o hardware necessário é pequeno e logicamente simples. O CDB pode funcionar com qualquer número de acumuladores e qualquer número de unidades de execução. Isso é possivel atraves do renomeamento de registradores, que permite contornar os conflitos de escrita depois de leitura e escrita depois de escrita (WAR e WAW). Para tal, o Tomasulo usa uma Map Table, que indica para cada registrador do computador uma Tag que representa a dependência de dados deste registrador a alguma unidade funcional. Em resumo, fornece uma implementação para a exploração automática e eficiente de múltiplas unidades de execução do hardware.

2 Arquitetura do Circuito Tomasulo

A Figura 1 mostra o circuito Tomasulo desenvolvido nesse projeto. Cada elemento será descrito individualmente em termos de funcionalidade e sinais de entrada e saída no próximo capítulo. O circuito é composto por uma fila first-in-first-out que armazena instruções do programa, um decodificador de instruções, banco de registradores e tabela de correspondência, reservation stations (RS's) para as três unidades funcionais, duas de adição/subtração e uma de ponto flutuante de multiplicação, e também para acesso à memória.

Quando uma instrução é emitida, o Decoder separa os elementos codificados e os

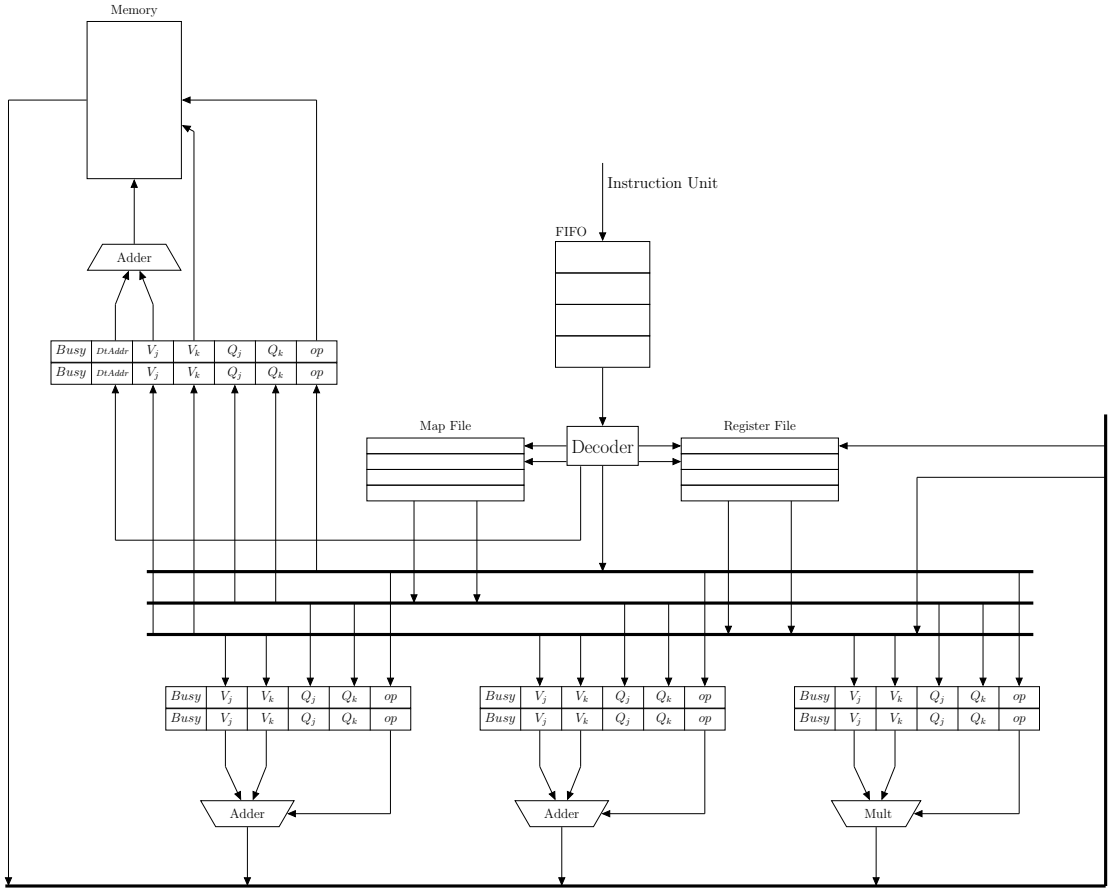

Figure 1: Circuito completo do Tomasulo

distribui. Ele aciona a leitura dos dois operandos na Register File (RegFile), cujos valores são armazenados nos V_j and V_k das RS's. Os mesmos endereços são lidos na

Map Table (MapTable), que possui a correspondência entre registradores e endereços nas RS's, chamados de tags. Se, para um dado registrador, o valor do seu endereço na Map Table for zero, significa que não há dependência de dados e, logo, os operandos estão prontos na RegFile. O operando decodificado é armazenado diretamente nas RS's.

As RS's são responsáveis por armazenar os elementos decodificados pelo Decoder e os dados vindos da RegFile e da MapTable. Cada RS possui um controlador associado que determina se alguma instrução está pronta para execução, i.e., seus operandos já foram calculados, qual linha da RS está vazia e quando uma instrução já fora calculada. Com exceção da RS associada à memória, a instrução a ser calculada pela unidade funcional é a primeira a ter seus operandos calculados. Na RS associada à memória, as instruções são executadas sequencialmente, para evitar conflito de dados na memória.

Os dados calculados pelas unidades funcionais ou carregados da memória são enviados a um bus, também chamado de Common Data Bus (CDB). Esse bus também possui um controlador associado o qual faz o controle dos dados enviados pelas unidades funcionais. Os resultados das operações são escritas na RegFile e são também disponibilizados diretamente para as RS's.

3 Descrição dos Componentes do Circuito

Nessa seção serão descritos cada um dos componentes importantes do circuito. Suas funções, ligações e particularidades são detalhadas a seguir.

3.1 FIFO

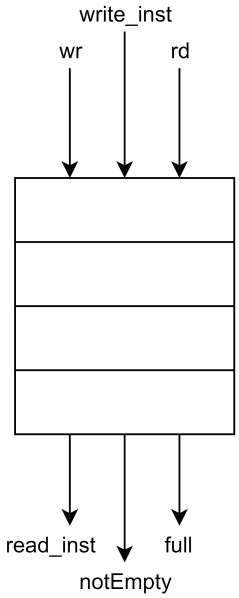

Figure 2: Esquema da FIFO e seus sinais.

Antes da execução, as instruções são trazidas para a uma estrutura de buffers do tipo FIFO, que armazena e acumula as instruções à serem executadas antes de serem expedidas pelo Decoder. As instruções chegam através de um sinal *write_inst* e são gravadas na FIFO quando o sinal *wr* estiver ativo e houver espaço na FIFO. Quando ela estiver cheia, a saída *full* será ativada. Caso existam instruções na fila, ou seja, o sinal de saída *notEmpty* está ativo, o Decoder demanda a próxima instrução ao acionar o sinal *rd* de leitura da FIFO. A instrução é então colocada na saída *read_inst* para que o Decoder a receba.

3.2 Decoder

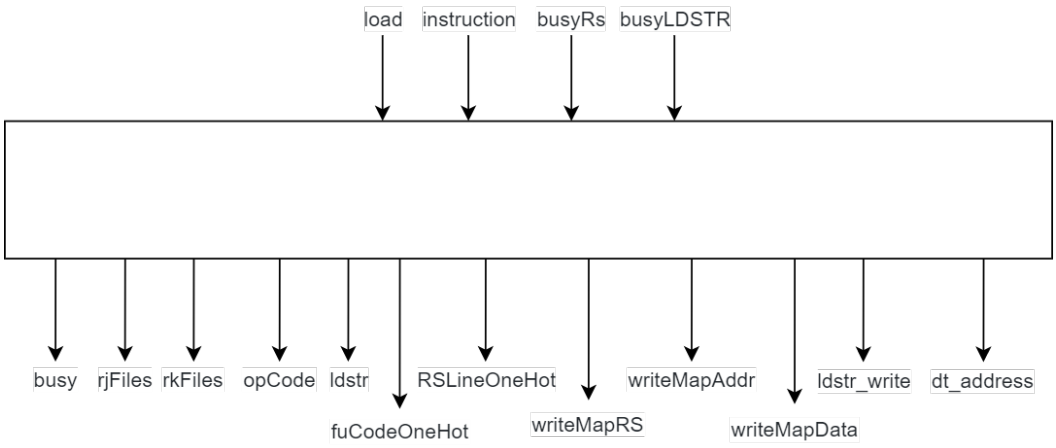

Figure 3: Esquema do Decoder e seus sinais.

O Decoder recebe a instrução da FIFO via o sinal de entrada *instruction*. Caso a FIFO tenha alguma instrução a ser lida, o sinal de entrada do Decoder *load* é setado em alto, iniciando o processo de decodificação da primeira instrução da fila. Foram utilizados para testes nesse projeto as instruções de soma, multiplicação, load e store usando os seguintes opCodes:

- 10001011000: Soma
- 11001011000: Multiplicação
- 11111000000: Store
- 11111000010: Load

Os endereços dos registradores de operandos e destino são decodificados usando o padrão da arquitetura Armv8. Os registradores dos operandos correspondem aos sinais *Rj* e *Rk*. O registrador de destino corresponde ao endereço de escrita na MapTable *writeMapAddr*.

Caso não haja nenhuma RS disponível para armazenar a instrução, verificado pelos sinais de entrada *busyRS* e *busyLDSTR*, o sinal de saída *busy* é setado em alto de

Table 1: Valor dos dados de saída do Decoder

	Soma	Multiplicação	Load	Store
writeMapRS	1	1	1	0
ldstr	0	0	0	1
ldstr_write	0	0	1	1

forma a FIFO não adiantar a fila. O valor dos sinais de saída *writeMapRS*, *ldstr* e *ldstr_write* são dependentes apenas do operação da instrução. *writeMapRS* refere-se à escrita na MapTable. Excetuando a operação de store, que não resulta na alteração dos estado dos registradores, todas as instruções produzem escrita na MapTable. O valor de opCode de load e store são descritos pelo sinal *ldstr* e, caso a instrução seja uma operação de memória, o sinal *ldstr_write* é setado em alto. A Tabela 1 apresenta os valores desses sinais em função da operação da instrução decodificada.

RSLineOneHot é um sinal de saída com mesmo número de bits que a soma de linhas de todas as RS's. Apenas um desses bits estará em alto e representa a linha de uma RS que será carregada. Cada linha possui uma tag, que é composto pela identificação da unidade funcional (hard-coded) e o número da linha na RS. O sinal *writeMapData* possui o valor da tag da RS na qual a instrução será escrita. Esse sinal será o dado de entrada para escrita na MapTable. E, por fim, o sinal *dt_address* é o valor do imediato que será somado ao valor de um registrador para cáculo do endereço efetivo nas operações de memória. Esse valor é escrito diretamente na RS.

3.3 Register File

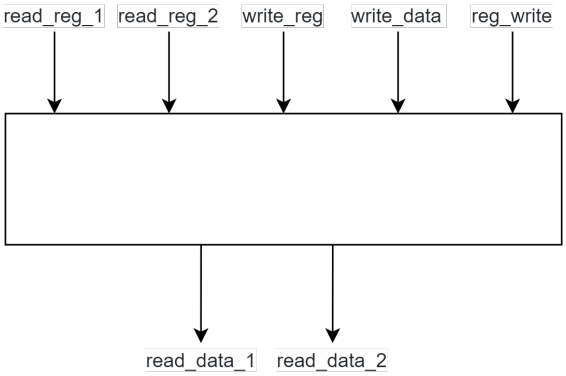

Figure 4: Esquema do Register File e seus sinais.

A Register File é um vetor de Registradores que armazena os valores dos parâmetros das instruções executadas pelo processador. Ela possui dois sinais de entrada para a leitura de registradores em paralelo: *read_reg_1* e *read_reg_2*. Então os valores dos registradores requisitados são colocados nas saídas *read_data_1* e *read_data_2* respectivamente. Para realizar a escrita nos registradores o sinal de controle de escrita *reg_write* deve ser acionado, então o sinal *write_reg* recebe o número do registrador ao qual o valor do sinal *write_data* será gravado.

3.4 Map Table

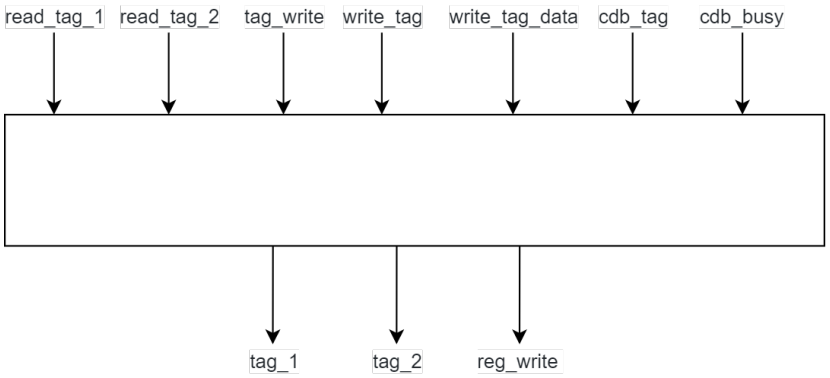

Figure 5: Esquema da Map Table e seus sinais.

Dentro da estrutura do Tomasulo, nós temos a necessidade de armazenar as Tags, que indicam qual Reservation Station fornecerá o valor de um determinado registrador. Para tal é utilisado a Map Table, um vetor de registradores, similar a Register File, que armazena uma Tag para cada registrador equivalente. Caso a Tag seja nula (ou zero) isso significa que o valor daquele registrador está correto e este não esta esperando nenhuma Reservation Station. Caso contrario, o valor da Tag representa a Reservation Station responsável por calcular o valor esperado pelo tal registrador.

A Map Table apresenta dois valores de entrada *read_tag_1* e *read_tag_2* para pedir a leitura de duas Tags em paralelo, os valores das Tags requisitadas são colocados nas saídas *tag_1* e *tag_2* respectivamente. Para realizar a escrita em uma tag, de forma similar ao RegFile, o sinal de controle de escrita *tag_write* deve ser acionado, então o valor de *write_tag_data* é registrado no registrador *write_tag*. Além disso a Map Table possui sinais vindo do CDB que indicam se uma Reservation Station já calculou os valores de um registrador indicado por uma Tag, isto significa que essa Tag já pode ser zerada. Esse comportamento é controlado pelos sinais *cdb_busy*, que indica que o CDB recebeu um valor, e *cdb_tag* que indica a Tag presente no CDB. Assim, o registrador que possuir o mesmo valor desta Tag vinda do CDB já pode ser liberado.

3.5 Reservation Station (Unidades funcionais)

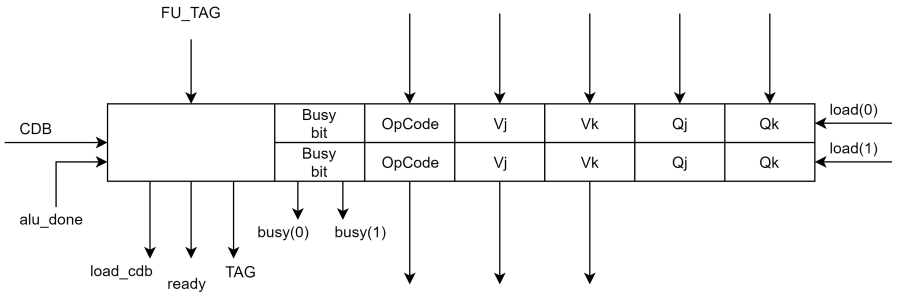

Figure 6: Esquema de uma Reservation Station e seus sinais.

As Reservation Stations (RS) foram implementadas com duas linhas por Unidade Funcional (UF). Cada linha possui os campos tradicionais de uma RS:

- o Busy bit, que indica se a linha armazena uma instrução que não foi executada;
- o OpCode, que determina a operação da UF;
- os valores dos operandos da instrução V_j e V_k;
- as Tags de valores que não estão prontos Q_j e Q_k.

Os valores V_j e V_k são zero caso os valores dependam de uma instrução que não foi completa e são substituídos pelo valor quando ele aparecer na entrad vinda do CDB com a Tag indicada no campo Q respectivo do valor.

Cada linha possui seu sinal de load dedicado que vem do Decoder, o responsável pela alocação de RS para instruções. Os sinais de Busy bit são enviados para o Decoder para a mesma finalidade.

A entrada FU_TAG é definida no circuito completa para funcionar como o Tag das RS de uma UF específica. Por exemplo, definiu-se que a UF Add1 seria 01, Add2 seria 10 e Mult seria 11. Sempre existe uma Tag 00 que é usada para indicar que não há mapeamento em nenhuma UF.

Quando uma instrução tem seus dois operandos prontos, seu OpCode e seus operandos são enviados para a UF e o sinal de ready dispara a UF (pois simulou-se um atraso para a execução de um operação).

Quando a UF termina seu cálculo ela aciona o sinal alu_done e a RS se responsabiliza por enviar o sinal de load_cdb para carregar o valor no CDB. Ela mantém sua saída e o sinal de load até que o valor seja carregado no CDB, lidando assim com a concorrência de outras UF. Uma vez no CDB, a linha da RS é liberada para receber novas instruções.

3.6 Unidades Funcionais

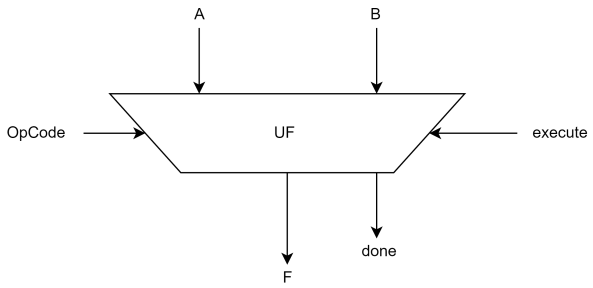

Figure 7: Esquema de uma Unidade funcional e seus sinais.

Foi decidido implementar três UF para o Tomasulo: dois somadores/subtratores e um multiplicador. Assim, pode-se testar o comportamento do algoritmo com duas UF de mesma função e uma de função diferente. Isso permitiu verificar o algoritmo de alocação do Tomasulo.

Todas UF possuem o esquema da Figura 3 com:

- dois operandos de entrada, A e B;
- um seletor de operação, OpCode;
- uma saída de resultado, F;
- um sinal de disparo de uma operação para simular um atraso no componente que vem da RS (ready), execute;
- um sinal que indica o fim de uma operação, done que vai para a RS (alu_done);

Os sinais de controle que foram colocados a mais, têm como objetivo simular o atraso de uma UF real fazendo com que as operações demorem 3 ciclos para se completar. Assim, a simulação feita do circuito se aproxima mais do funcionamento de um circuito real.

3.7 Reservation Station (Memória)

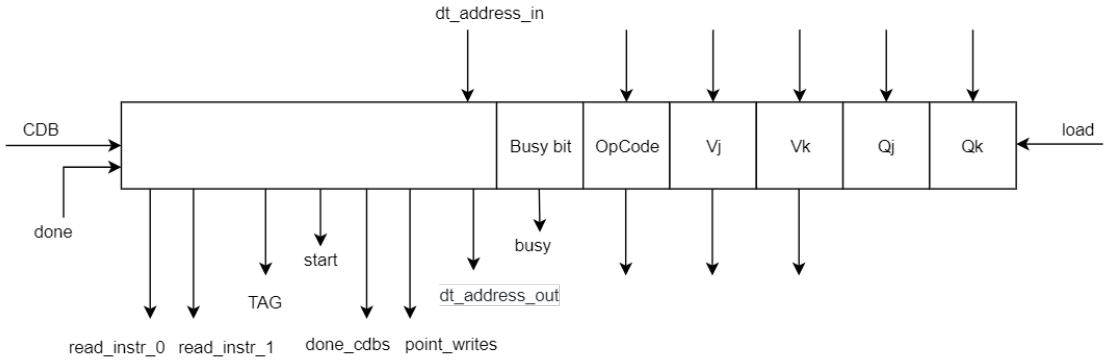

Figure 8: Esquema da RS de Memória e seus sinais.

A RS associada à memória possui um campo a mais em relação às outras RS's, chamado *dt_address_in*, o qual armazena o valor do imediato a ser adicionado ao valor *Vj* para cálculo do endereço efetivo. Além disso, o sinal de saída *start* indica para o Adder de endereço de memória efetivo que a primeira instrução da fila já possui seus operandos prontos. O sinal de entrada *busy* indica que o Adder ainda está processando os operandos. Os demais sinais de entrada e saída são idênticos aos das RS's das unidades funcionais.

3.8 Adder (Cálculo de endereço efetivo)

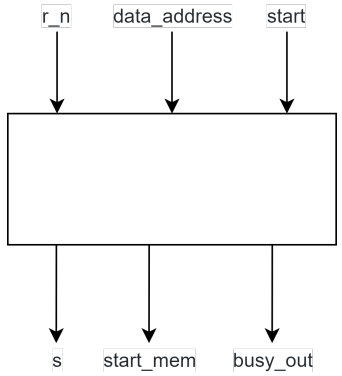

Figure 9: Esquema do Adder e seus sinais.

O componente Adder é responsável por calcular o valor efetivo do endereço de memória solicitado. Para isso, quando o sinal de controle *start* for ativado ele recebe o valor *r_n* da RS, ao qual soma o valor imediato *data_address*. Durante a operação, o sinal *busy_out* se mantém ativo, e ao término desta, o endereço calculado é colocado na saída *s* e o sinal *start_mem* é ativado.

3.9 Memória

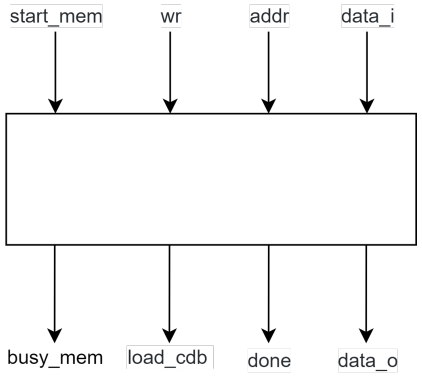

Figure 10: Esquema da Memória e seus sinais.

O bloco de memória desta arquitetura foi implementado como um vetor de palavras que armazenam as informações dos programas do computador. Ela permite a gravação e leitura de palavras como uma RAM. Ela possui em sua entrada dois sinais vindos do Adder, que realiza o cálculo do endereço efetivo usado pela unidade de memória: o sinal *start_mem* que indica que que o endereço do sinal *addr* já fora calculado. Em uma operação de escrita, osinal de controle de leitura/escrita *wr* deve estar ativo para que então o valor do sinal *data_i* seja armazenado no endereço *addr*. Já no processo de leitura, o sinal *wr* deve estar em zero e o sinal de saída *data_o* receberá o valor do endereço indicado por *addr*. Além disso, a memória possui outros sinais que permitem sua interface com os outros componentes, como o *busy_mem* que indica se a memória está ocupada realizando alguma operação, o sinal *done* que indica para a RS de memória que a operação requisitada já foi realizada e o *load_cdb*

que é o sinal de controle de escrita no CDB.

3.10 Common Data Bus (CDB)

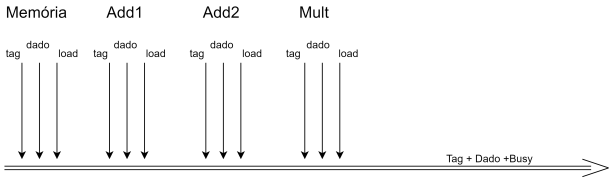

Figure 11: Esquema do CDB e seus sinais.

O CDB foi feito de forma que cada UF e a memória enviassem três entradas para ele: uma com a tag da RS à qual o valor está ligado, outra com o dado a ser enviado e um sinal de load para indicar que há um valor a ser carregado no CDB.

O CDB olha cada load buscando um valor a ser carregado e a cada ciclo ele começa a varredura por um load diferente para garantir que não haverá monopólio do CDB. Os sinais de load das UF são provenientes das RS ligadas a elas.

Carregado um dado no CDB, ele vai ser enviado a todas RS (para que valores sendo esperados já sejam adiantados) e ao banco de registradores para guardar um valor caso seja necessário. Tudo isso é possível, pois além do dado, o CDB também leva a Tag da RS de origem desse resultado. O sinal de busy é usado para indicar que existe um valor válido no CDB.

4 Testbench do Circuito Tomasulo

Para testarmos o circuito Tomasulo desenvolvido nós implementamos um testbench com uma sequência de instruções que colocaria a prova o funcionamento dos diferentes componentes, a interface entre eles, assim como a capacidade do Tomasulo de gerenciar e executar as múltiplas instruções em paralelo e resolver os conflitos de dados.

O nosso teste se baseia na seguinte sequência de instruções, um programa simples que forçaria o circuito a lidar com situações como as RS cheias e execuções fora de ordem :

```
1. LD    R0, (R3+504)
2. ADD   R0, R0, R2
3. MULT  R0, R0, R2
4. STR   R0, (R3+504)
5. LD    R1, (R3+504)
6. ADD   R4, R3, R3
7. ADD   R1, R1, R1
8. ADD   R1, R1, R1
9. ADD   R1, R1, R1
10. MULT  R1, R2, R1
11. MULT  R1, R2, R1
12. MULT  R1, R2, R1
13. MULT  R1, R2, R1
14. STR   R1, (R3+505)
15. LD    R0, (R3+505)
16. LD    R2, (R3+505)
```

Sendo que as instruções de ADD e MULT são escritas da seguinte forma: "ADD rd, rs, rt", onde rd é o registrador de destino e rs e rt são os operandos. Já as instruções de LD e STR, são escritas como "LD rd, (rs+Immed)", onde rd é o destino e o endereço de memória é igual à soma do conteúdo de rs e do imediato Immed.

4.1 Resultados

Usando o ModelSim para analisar as funções de onda geradas pela simulação nós obtemos:

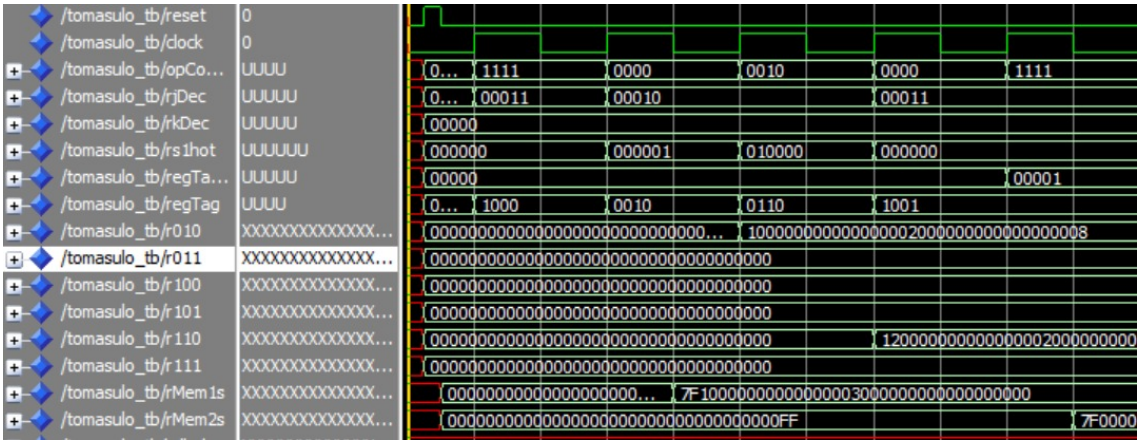

Figure 12: Testbench parte 1

Na primeira parte (Figura 12) nós vemos a emissão das instruções 1, 2, 3, 4, 5 feitas pelo Decoder. O sinal de OpCode e os registradores operandos e destino de cada instrução sendo expedidos. Em seguida podemos ver os sinais chegando nas Reservation Stations respectivas, primeiramente na RS de Store (Tag = 1000), em seguida na de Add (Tag = 0010) e depois a de Mult (Tag = 0110).

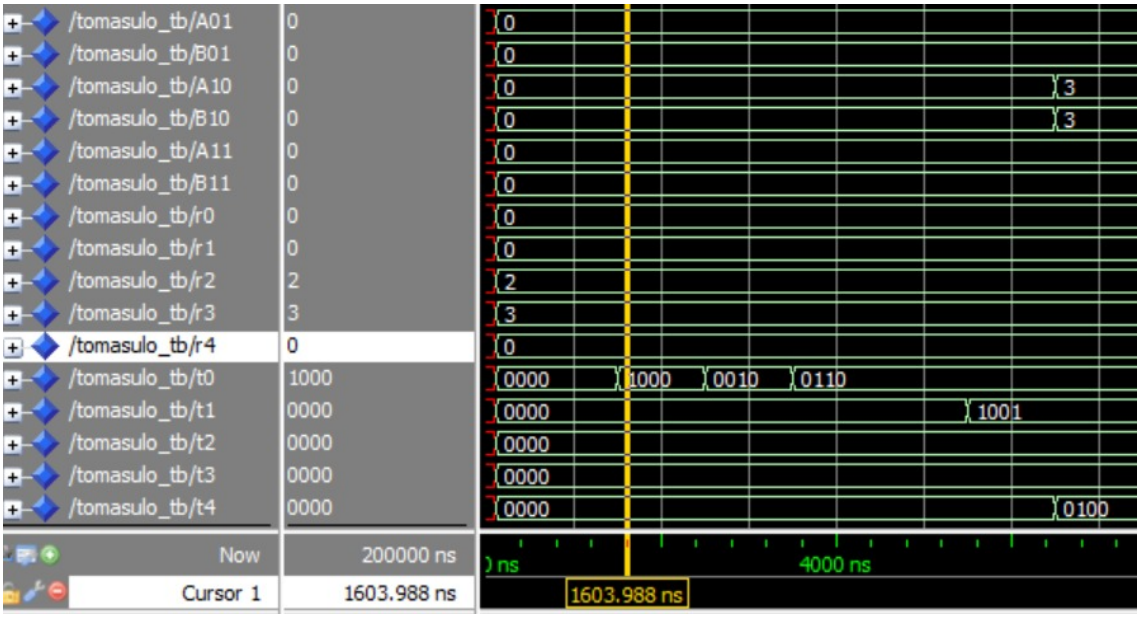

Figure 13: Testbench parte 2

Em seguida nós podemos ver os Tags relativos as primeiras instruções sendo

escritas na MapTable. A Tag do R0 recebe o valor 1000 (RS de LD/STR) em seguida o valor 0010 (RS Add) e depois 0110 (RS Mult). Assim a Tag de R0 registra e guarda apenas o último valor que seria escrito nele.

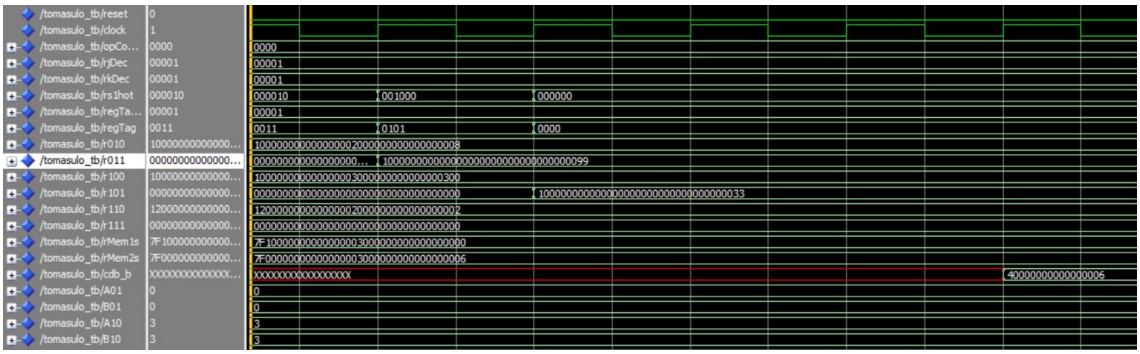

Figure 14: Testbench parte 3

Nesta parte da simulação (Figura 14) nós vemos a 6a instrução, ADD R4, R3, R3 sendo executada. O R3 possui o valor 3 e podemos vê-la na 1a saída do Decoder, entrando na RS100 (Tag = 4) (RS de Add), sendo executada pela FU10 e finalmente sendo carregada no CDB (sinal 400...06) onde 4 é a Tag da RS (100 = 4) e 6 é o resultado da operação da soma R3 + R3 = 3 + 3.

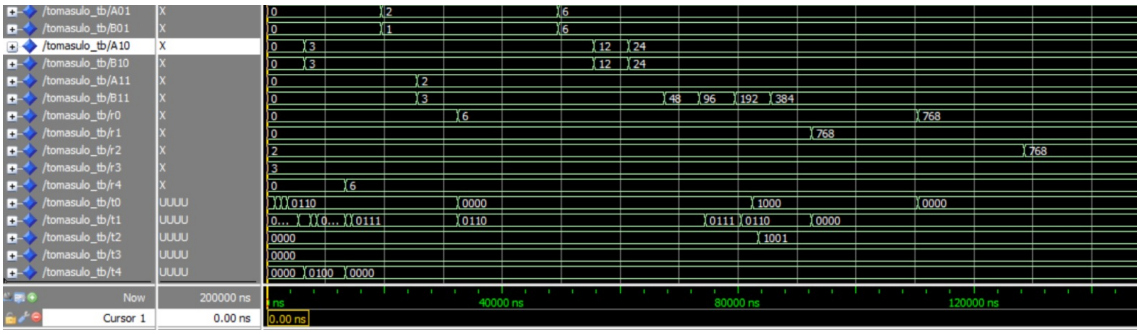

Figure 15: Testbench parte 4

Na Figura 15 nós podemos ver globalmente os sinais das entradas A e B de cada FU a cada instrução assim como os valores calculados e então armazenados nos registradores nos sinais R0, R1, R2 e R4. Podemos ver que a instrução 6. ADD R3 + R3 = R4 é executada antes das instruções 2,3,4 e 5, uma vez que estas estão

esperando a memoria carregar a instrução 1. Com o mecanismo de renomeação de registradores a arquitetura é capaz de adiantar o cálculo das outras instruções que dependem de R1 mesmo que este esteja aguardando a memoria na instrução 5, assim o cálculo das instruções 7 a 13 é feito, como podemos ver nos valores de entrada das FU, mesmo que o valor de R1 ainda não tenha sido salvo. Ao final da simulação nos constatamos que os registradores R0, R1 e R2 todos possuem o resultado das operações: 768.

5 Considerações Finais

Com este projeto de elaboração do algoritmo Tomasulo nós pudemos desenvolver nossas capacidades em arquitetura de Hardware assim como aprofundar nosso entendimento do processo de escalonamento dinâmico e conflito de dados, visando a otimização do processamento e execução de instruções. A arquitetura desenvolvida se mostrou eficaz, podendo lidar com situações de execução fora de ordem e contornando os conflitos de dependência através do uso das Reservation Stations, Tags e o CDB. Para o programa de teste proposto para nossa simulação nós obtivemos uma melhoria significante se comparado com o tempo total teórico que levaria para este mesmo programa ser executado em uma arquitetura sem nenhuma técnica de escalonamento.