CENTRO FEDERAL DE EDUCAÇÃO TECNOLÓGICA DE MINAS GERAIS

TRABALHO PRÁTICO III

O Algoritmo de Tomasulo

Executando fora de ordem

Arthur Estevão de Souza Machado

Marcelo Lopes de Macedo Ferreira Cândido

Engenharia de Computação

Laboratório de Arquitetura e Organização de Computadores II

28 de outubro de 2018

1 INTRODUÇÃO

Na computação, o constante incremento da performance é uma regra que não deve ser quebrada. Mesmo com o estabelecimento dos processadores *single-core* já com o *pipeline* avançado, a busca por aumentar a performance desses módulos não parou. A alternativa adotada foi o escalonamento dinâmico de instruções, a ser abordado mais profundamente na subseção 1.1.

1.1 O ALGORITMO DE TOMASULO

O escalonamento dinâmico visa a reorganização das instruções, de modo a se esquivar dos *stalls* gerados pelas dependências envolvidas no código desdobrado. Ele consegue decidir qual instrução executará, diferentemente do escalonamento estático (feito pelos compiladores), que organiza previamente o código, antes de executá-lo. Além disso, o escalonamento dinâmico permite a simplificação do compilador e que o código que foi compilado para um determinado *pipeline* seja executado eficientemente em outro [2]. Um algoritmo capaz de realizar isso é o de Tomasulo [3].

Esse algoritmo, criado por Robert Tomasulo em 1967 na IBM, permite a execução (operações lógico-matemáticas) fora de ordem e em paralelo e o uso mais eficiente das unidades funcionais do *hardware*. Além disso, evita dependências de dados através da renomeação de registradores, estações de reserva para todas as unidades funcionais, possui uma barramento (tratado por barramento comum de dados, ou *CDB*, em inglês) que permite o *broadcast* (transmissão simultânea via rede) dessas unidades para as estações. O algoritmo de Tomasulo se organiza em três etapas, como será mostrado, em alto nível, nas próxima sub-subseções, sem se considerar as diferenças entre os tipos de instruções.

1.1.1 PRIMEIRO ESTÁGIO - ISSUE (EMISSÃO)

Nessa etapa, a instrução é pega de um módulo chamado fila de instruções. Se seus operandos estiverem disponíveis nos respectivos registradores, eles são emitidos juntos para a instrução para a respectiva estação de reserva. Do contrário, A instrução é emitida sozinha.

1.1.2 SEGUNDO ESTÁGIO - EXECUTE (EXECUÇÃO)

Se todos os registradores de uma determinada instrução que está em uma estação de reserva estão disponíveis, os operandos são encaminhados para a respectiva unidade funcional para a execução. Caso algum operando não esteja disponível ainda, monitora-se o CDB a sua espera e, assim que ele aparecer, realiza-se o processo descrito acima.

1.1.3 TERCEIRO ESTÁGIO - WRITE-RESULT

A essa altura, os resultados das unidades funcionais estão disponíveis. Se o CDB estiver disponível, os dados são escritos nele e, consequentemente, encaminhados para as estações e registradores que os esperam. Nessa etapa também ocorrem as escritas à memória [4].

1.2 A PRÁTICA

Essa prática consiste em implementar o algoritmo de Tomasulo utilizando a linguagem de descrição de *hardware* `Verilog`. Nesse relatório serão apresentados o funcionamento e os detalhes da implementação, bem como as simulações a nível de registradores que comprovem seu funcionamento. Além disso, serão apresentadas as dificuldades, sugestões e comentários que surgiram ao longo do projeto.

2 FUNCIONAMENTO E DETALHES DA IMPLEMENTAÇÃO

Aqui serão apresentados o funcionamento de cada módulo do *hardware* implementado para o Tomasulo (pelo menos os principais), bem como as principais decisões de projeto adotadas ao longo da implementação.

2.1 DECISÕES GERAIS

A dupla adotou o padrão de se colocar uma porta *clear* em todos os módulos, de modo que, quando ela é ativada, as variáveis dos módulos são mudados para valores estipulados via código. Isso foi escolhido na esperança de se poder usar o código na placa Altera existente no relatório, visto que o bloco `initial` não funciona adequadamente nela.

O número de *bits* para as instruções foi escolhido tendo-se em vista os campos desejados para elas:

- código da operação: três *bits* (esperava-se implementar as instruções de desvio);
- registrador destino: três *bits*;
- registrador fonte: três *bits*;
- deslocamento: sete *bits*;

2.2 UNIDADE DE INSTRUÇÕES / MEMÓRIA DE DADOS

Memória RAM LPM padrão do *Altera Quartus 13.0sp*, com 48 posições de 16 *bits*. O número de posições foi escolhido assim para que se pudesse colocar instruções e dados de forma suficientemente espaçada na memória.

2.3 *Program Counter*

Basicamente um contador que, ao receber o sinal de incremento a cada subida de *clock*, aumenta o número armazenado em um, fornecendo o endereço para a unidade de instruções. O sinal de incremento está diretamente ligado à fila de instruções, sendo de sinal alto enquanto a fila não estiver cheia.

Seu registrador que armazena o endereço de memória possui apenas cinco *bits*. Esse tamanho foi determinado dessa maneira para evitar que qualquer programa acessasse posições maiores que 48 ($2^6 = 64 > 48$).

2.4 FILA DE INSTRUÇÕES

Mecanismo de oito posições que funciona com a ajuda de dois apontadores. O primeiro apontador, `frente`, marca a posição do item mais antigo da fila, e o segundo, `trás`, marca a do mais novo. Quando um item é adicionado à fila, `trás` é incrementado. Se um item é retirado, `frente` é incrementado.

Além disso, para garantir se uma instrução foi lida ou não, a dupla criou o vetor `iValid`, que registra se a instrução nas posições da fila são válidas ou não. Com a ajuda disso, o mecanismo

também consegue avisar se está cheio e vazio. Se todos as posições de `iValid` estão com o sinal alto, a fila está cheia, pois todas as posições estão válidas. Analogamente, se todas as posições estão com sinal baixo, a fila está vazia, pois não há posição válida.

O tamanho das posições da fila é o mesmo tamanho da instrução, 16 *bits*. Já o número de posições foi escolhido arbitrariamente

2.5 REGISTRADOR DE INSTRUÇÕES

Simplesmente um registrador que guarda a última instrução despachada para livre uso pelo *hardware* do Tomasulo. Seu tamanho segue o tamanho da instrução.

2.6 BANCO DE REGISTRADORES

Consiste em um módulo com oito variáveis com 16 *bits* do tipo `reg` para armazenamento de dados. Há também oito, do mesmo tipo, com quatro *bits*, para armazenamento de dependências (representadas pelo nome da estação que irá escrever naquele registrador).

O banco consegue comunicar ao exterior até dois dados ou até duas dependências (caso a posição requerida pelo exterior possua). Se uma posição tem dependência, o banco a comunica sem comunicar o dado existente naquela dependência. A escrita de dado e dependência pode ocorrer simultaneamente.

2.7 ESTAÇÃO DE RESERVA *Add*/*Sub*

A dupla decidiu criar um módulo para estação de reserva que representasse apenas uma linha da tabela vista em Hennessy e Patterson [1]. Ou seja, o módulo possui os mesmos campos como base:

- `Name (ID)`: quatro *bits*;
- `busy`: um *bit*;
- `opCode`: três *bits*;
- `Vj`: 16 *bits*;
- `Vk`: 16 *bits*;

- `Qj`: quatro *bits*;
- `Qk`: quatro *bits*;

Além dos campos mostrados na obra citada no parágrafo anterior, existem outras portas auxiliares. Cada estação possui um número de identificação, que serve para o registro de dependências no banco de registradores. Além disso, o módulo da estação possui as portas

- `start`: quando o sinal fica ativado durante a subida do `clock`, a estação passa a trabalhar;
- `busy`: quando a estação passa a trabalhar, ficando **ocupada**, esse sinal é ativado;
- `despacho`: avisa que os dados necessários para a instrução já estão disponíveis e prontos para despachar;
- `confirma`: sinal de entrada, vindo da Unidade Aritmética correspondente, que a operação relativa aqueles dados já foi realizada e a estação pode ser liberada.

sendo aqui listadas as principais. Mais informações podem ser encontradas no código que acompanha esse relatório.

2.8 UNIDADE ARITMÉTICA *Add/Sub*

Responsável por realizar operações de soma/subtração de acordo com os dados fornecidos por alguma das estações de reserva do mesmo tipo. Ela demora três ciclos para a realização da operação, como especificado no enunciado da prática, a partir da recepção do sinal alto para início das operações através da sua porta `start`.

Suas portas possuem as seguintes dimensões:

- start: um *bit*;
- ID_RS_in: quatro *bits*;
- Dado1: 16 *bits*;
- Dado2: 16 *bits*;
- OP_Rd: três *bits*;

- Resultado: 23 bits, sendo 16 para o resultado da operação, quatro para o identificador da estação de reserva responsável pela operação e três para identificar o registrador de destino;
- confirmacao: um *bit*;
- busy: um *bit*;
- desWrAS: um *bit*.

2.9 ESTAÇÃO DE RESERVA *Load/Store*

A mesma configuração das estações de reserva para operações de *Add/Sub*, tendo um campo para o deslocamento (de sete *bits*), como indicado por Hennessy e Patterson [1].

2.10 UNIDADE ARITMÉTICA *Load/Store*

Responsável por realizar os cálculos de endereço base mais deslocamento para as operações de *load/store*. Diferentemente da Unidade aritmética *Add/Sub*, essa demora apenas dois ciclos para completar seu processo de operação. Segue as mesmas configurações da unidade de operações *add/sub*.

2.11 DECODIFICADOR DE CONFIRMAÇÃO

A partir de uma unidade aritmética, recebe o identificador da estação de reserva que mandou a operação para essa unidade. O decodificador decodifica o identificador para uma *string* de ativação, cujo *bit* ativado corresponde a estação de reserva que chamou a unidade aritmética. Isso serve para mostrar que a unidade terminou a operação.

Os tamanhos de suas portas são os seguintes:

- `ID`: quatro *bits*, pois são oito estações e o número `4'b0` não pode representar nenhuma, pois significa que não há dependência naquele registrador;
- `confima`: oito *bits*.

2.12 ÁRBITRO DO CDB

Responsável por decidir quem, dentre as unidades aritméticas, escreverá no banco de registradores primeiro. Ele determina isso analisando qual instrução veio primeiro, por meio das linhas das instruções, que são armazenadas no *hardware*. Também é capaz de ativar o banco de registradores para escrita.

3 SIMULAÇÕES

A fim de apresentar o funcionamento do algoritmo, será utilizado o ambiente de simulação ModelSim integrado ao Quartus II. Para isso, é necessário ressaltar alguns pontos fundamentais no funcionamento do algoritmo.

1. Uma instrução após estar disponível, gera a requisição dos dados contidos nos registradores trabalhados.

2. Com a requisição efetivada, verifica-se a existência de uma estação de reserva para a respectiva instrução.

3. Havendo estação disponível os dados requisitados são encaminhados para a estação de reserva destino. Caso não exista, aplica-se o stall nessa etapa e em todas anteriores.

4. Na estação de reserva é examinado se há alguma dependência dos registradores que serão utilizados. Caso exista a dependência, o dado é esperado para ser atualizado através do CDB.

5. Possuindo todos os dados disponíveis, estes são encaminhados para a Unidade Aritmética referente a instrução.

6. Após realizar o cálculo dos dados, o resultado é encaminhado ao CDB Arbiter que deverá tratar a escrita no CDB.

7. Aplica-se a rotina referente à última etapa da instrução.

3.1 CONFLITO DE DADOS

Para apresentar o tratamento do algoritmo em caso de dependência de dados, será inserida a sequência de operações:

OFFSET 7 bits- RS 3 bits -RD 3 bits - OP 3 bits
0000000-111-000-011 == LD R0, 0(R7)
0000000-000-000-001 == ADD R0, R0
0000000-001-000-010 == SUB R0, R1

Essa sequência de operações apresenta os conflitos RAW (read after write), WAW (write after write) e WAR (write after read). O algoritmo de Tomasulo possibilita minimizar estes conflitos e com isso proporcionar uma maior vazão de dados reduzindo a ociosidade do processador.

3.1.1 INSTRUÇÃO LD R0, 0(R7)

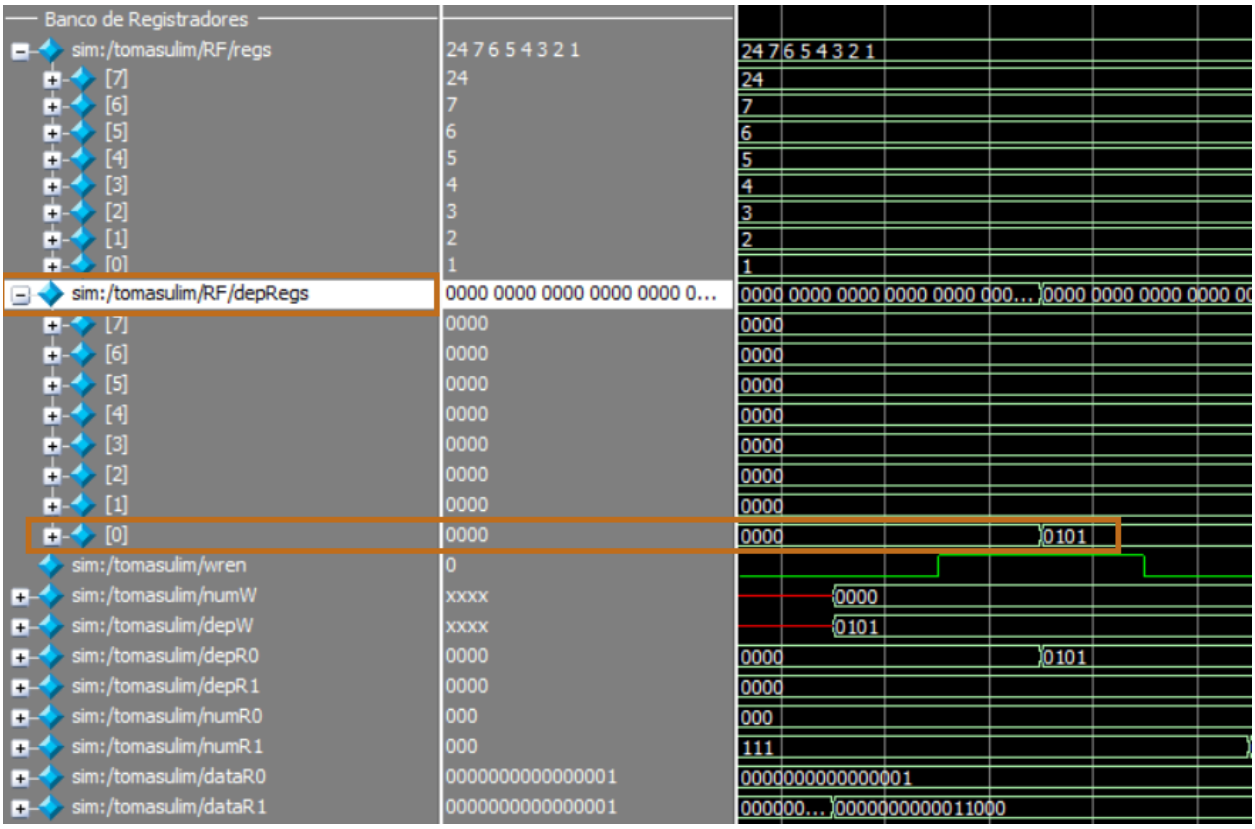

Figura 1: Fazendo a requisição de dados da instrução. Em seguida, coloca o registrador destino dependente da estação de reserva.

A instrução é encaminhada para a estação com identificação 0101 (5 em decimal), que é uma estação para load e store. Além disso, são encaminhados os valores contidos nos registradores e suas respectivas dependências (depRegs). Como depRegs inicialmente é 0000 para todos os registradores, nenhum registrador é considerado dependente de uma instrução.

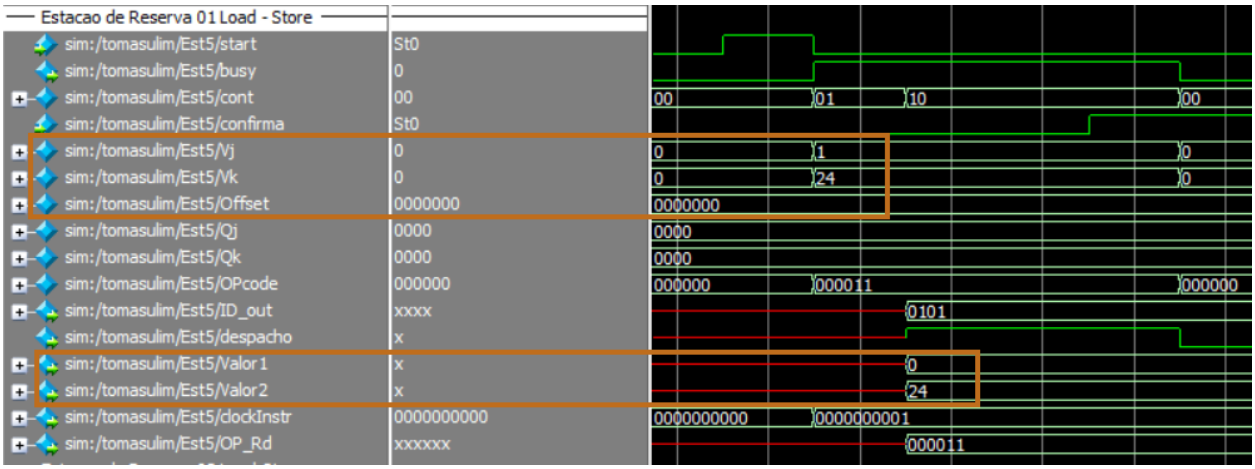

Figura 2: Os valores são armazenados em VJ (R0), VK (R7) e OFFSET. Já as dependências em QJ e QK e como não há nenhuma, os valores estão corretos e podem ser enviados para a Unidade Aritmética.

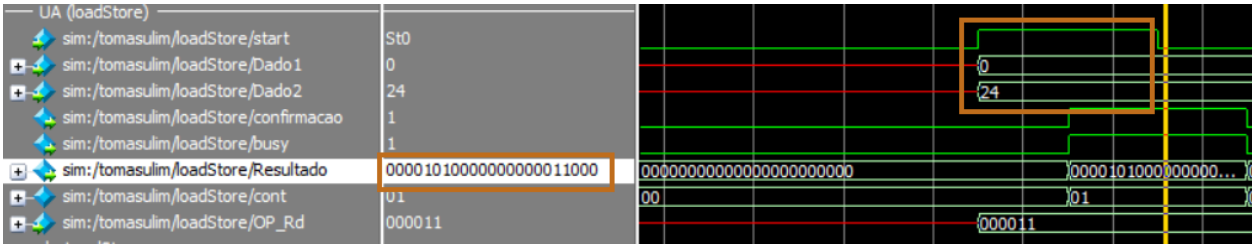

Figura 3: Os recebidos (OFFSET e R7) são somados para cálculo do endereço que será trabalhado.

Vale ressaltar, que o resultado da Unidade Aritmética possui 23 bits, dos quais 16 representam o cálculo do endereço, 4 representam a Estação de Reserva que gerou a instrução, e 3 representam o Registrador destino. Portanto:

Resultado = RD (3 bits) - Identificador da estação (4 bits) - Endereço (16 bits)

É possível perceber que o endereço calculado está correto, pois o valor contido no registrador R7 era 24 somado com 0 do OFFSET resultou em 11000 = 24 que pode ser encaminhado para a memória.

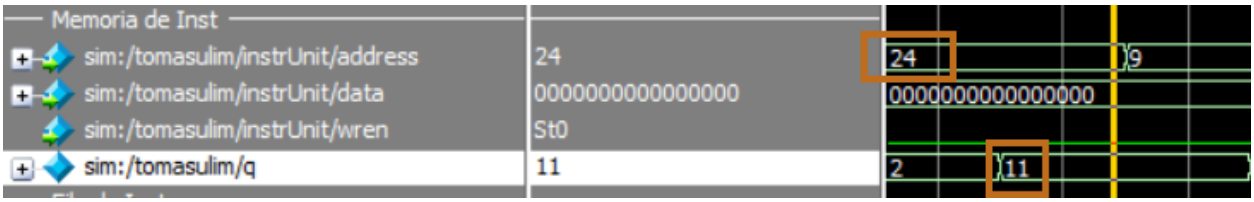

Figura 4: O endereço calculado pela Unidade Aritmética é usado pela memória e o dado referente é obtido

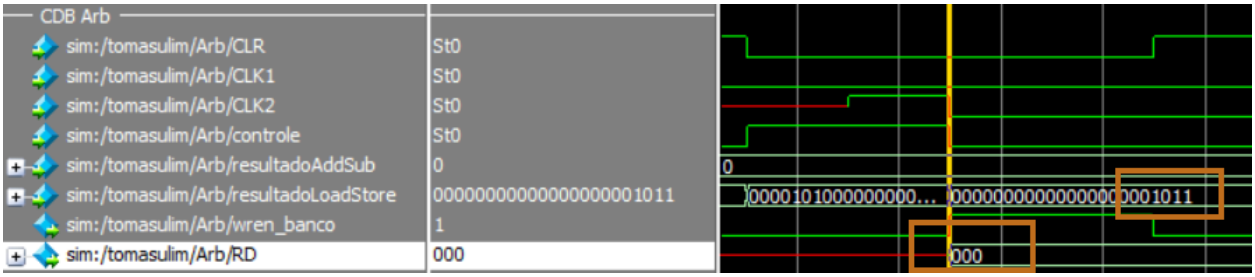

Figura 5: O dado buscado da memória é enviado ao CDB Arbiter que prepara a escrita no registrador destino.

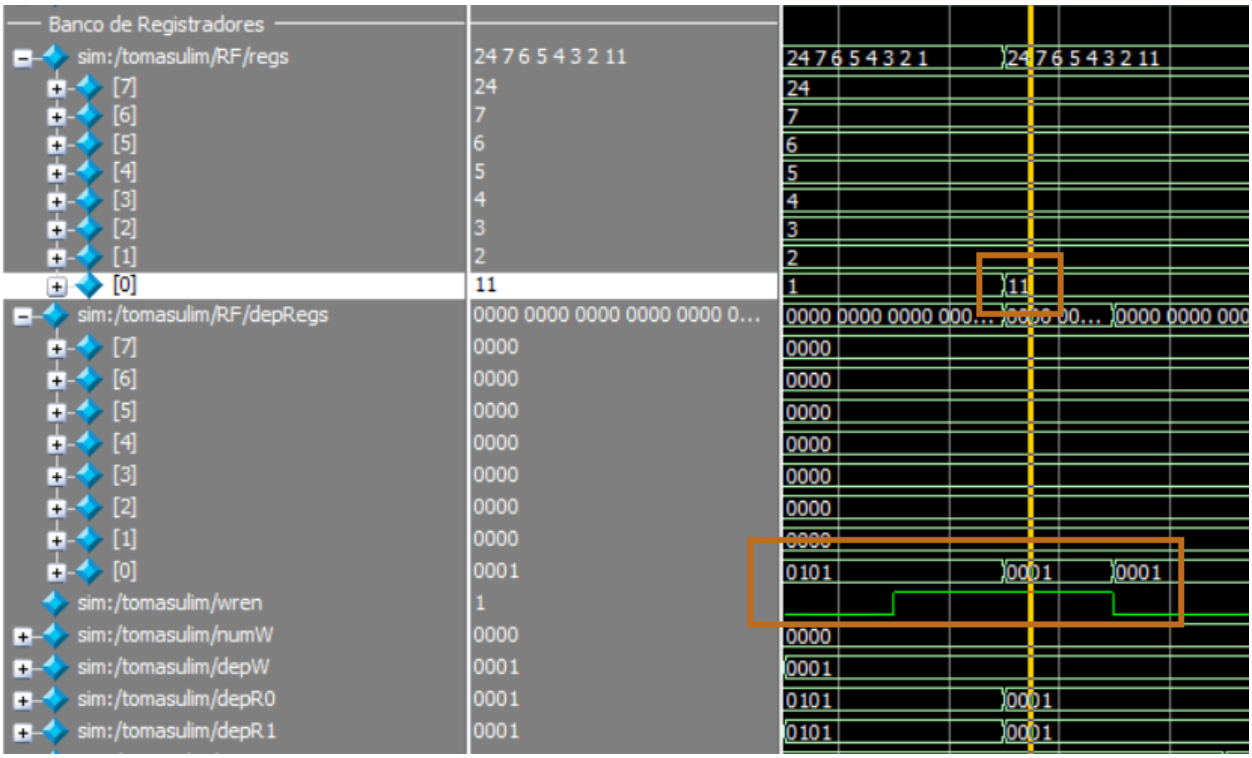

Figura 6: A atualização do dado é feita através da escrita no CDB e tornando o habilita escrita do banco de registradores igual a 1.

Vale ressaltar que neste passo, a instrução seguinte já foi iniciada, por isso a atualização de dependência do registrador 0 foi de 5 (0101) para 1 (0001), já que a Estação de Reserva 1 trabalha com operações de adição e subtração, e estava vazia.

3.1.2 INSTRUÇÃO ADD R0, R0

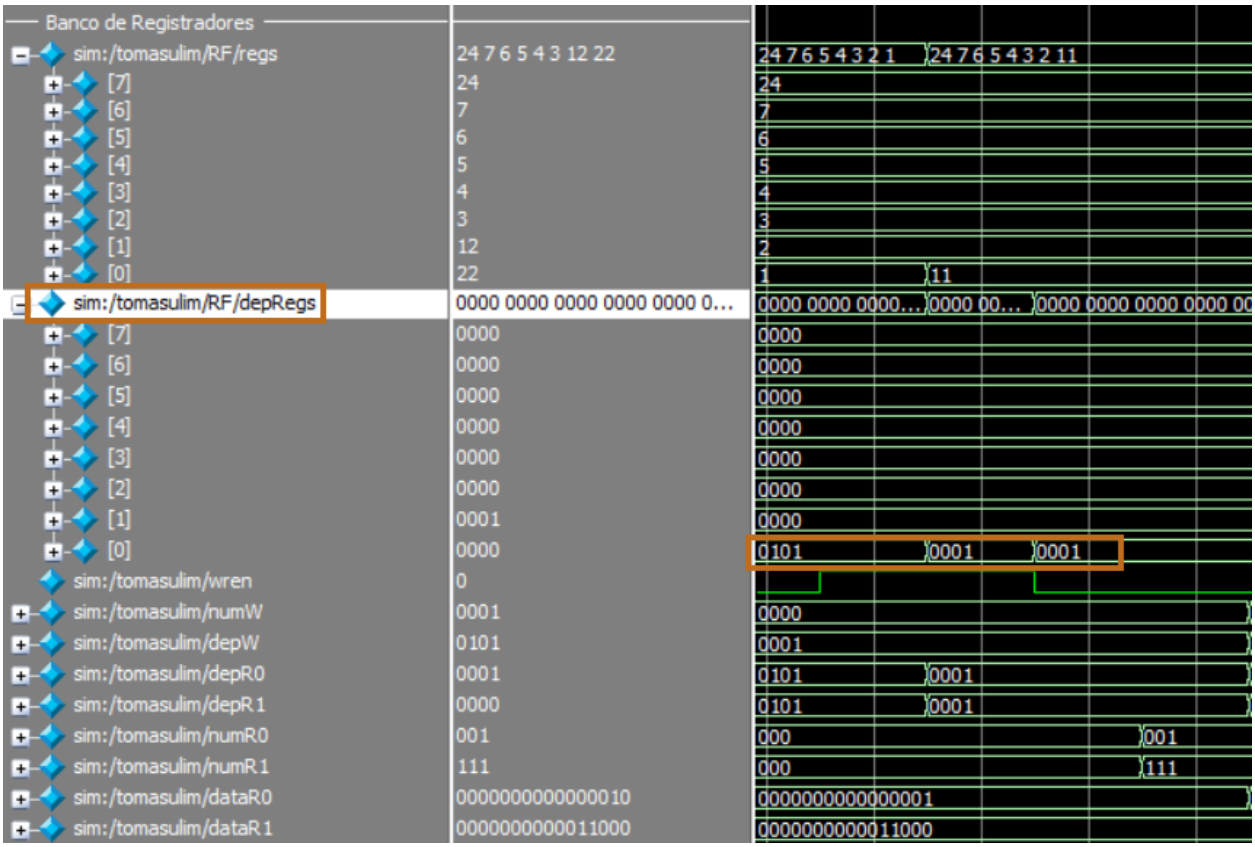

Figura 7: Fazendo a requisição de dados da instrução. Em seguida, coloca o registrador destino dependente da estação de reserva.

A instrução é encaminhada para a estação com identificação 0001 (1 em decimal), que é uma estação para adição e subtração. Além disso, são encaminhados os valores contidos nos registradores e suas respectivas dependências (depRegs). Como existe dependência, a estação 1 deve esperar o resultado de R0 ser atualizado no CDB.

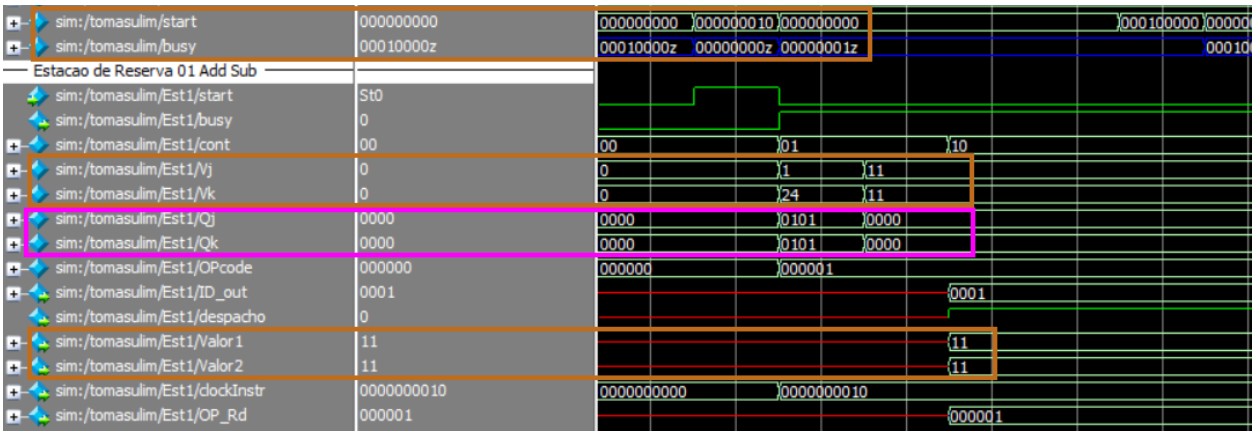

Figura 8: Os valores são armazenados em VJ (R0), VK (R0). Já as dependências em QJ e QK.

É possível verificar que o valor contido nos registradores é atualizado pelo CDB e a dependência deixa de existir (QJ e QK iguais a 0). Com isso, os dados podem ser encaminhados para a Unidade Aritmética de inteiros.

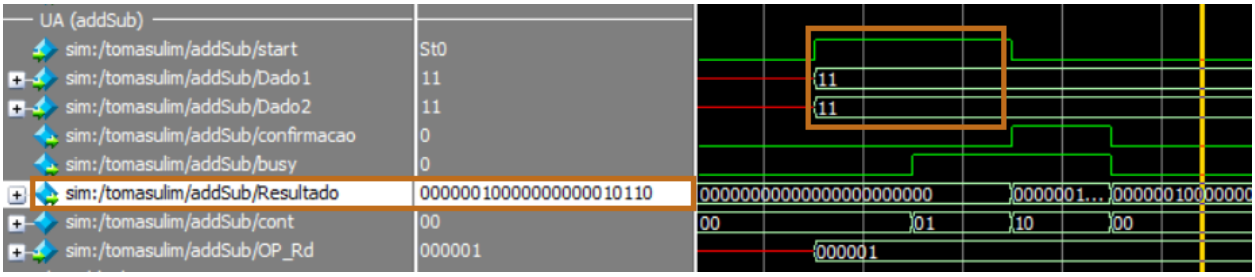

Figura 9: Os recebidos (Ambos iguais a 11) são somados para obtenção do resultado final (22 = 10110). Em seguida, este dado é enviado para o CDB Arbiter.

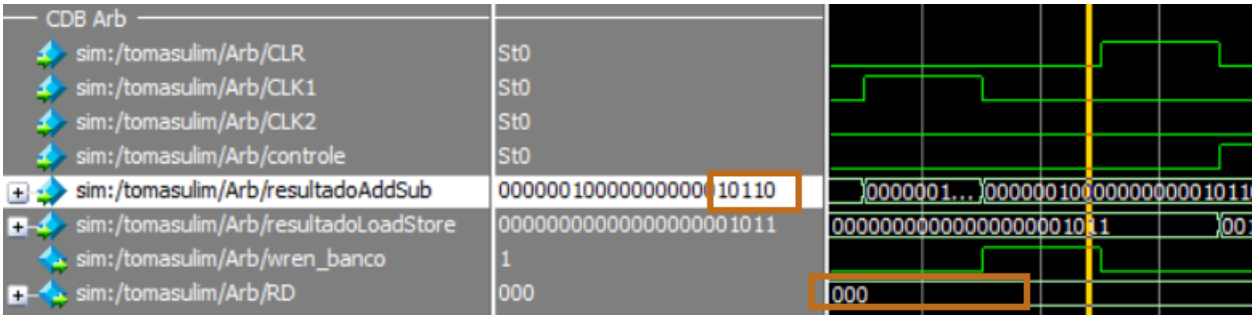

Figura 10: O dado encaminhado (10110), é preparado para ser escrito no registrador destino.

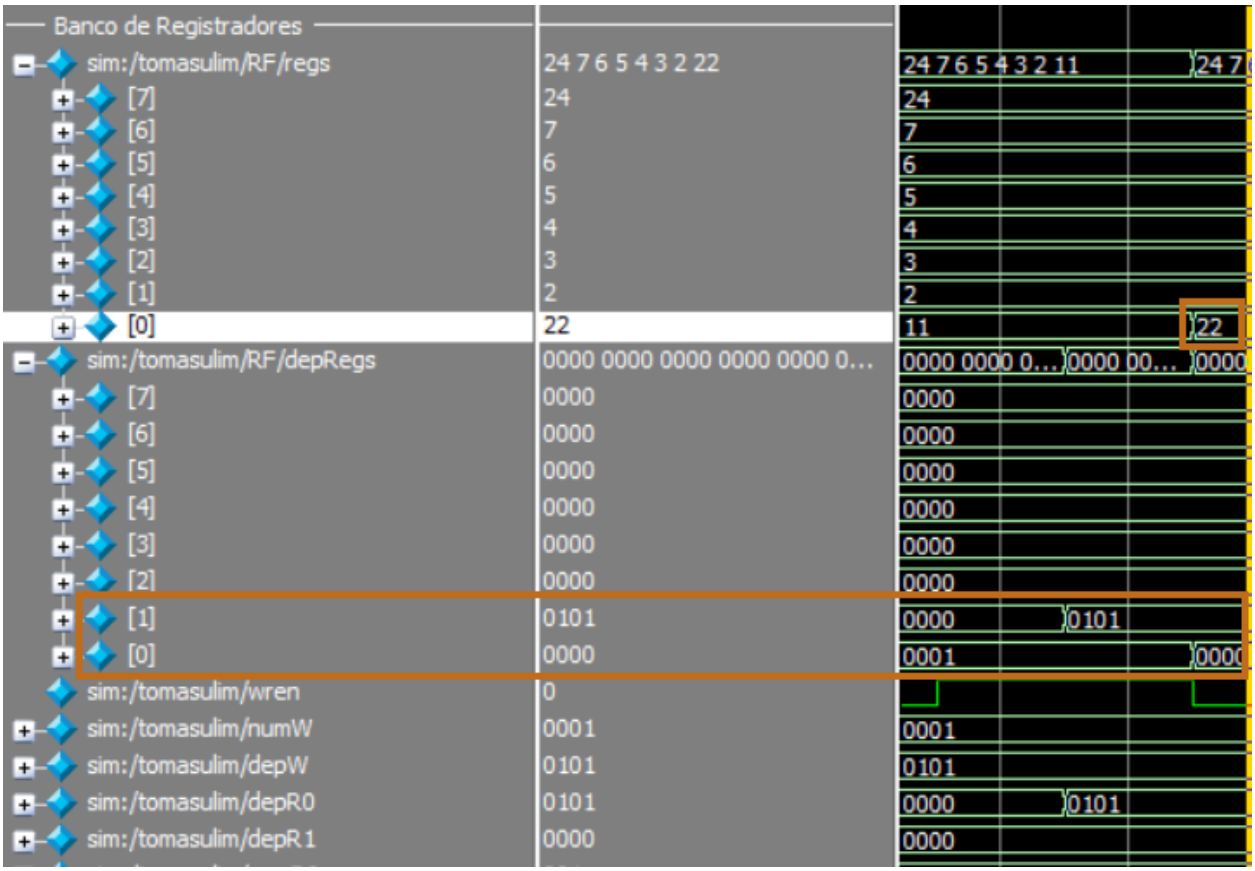

Figura 11: A atualização do dado é feita através da escrita no CDB e tornando o habilita escrita do banco de registradores igual a 1.

3.1.3 INSTRUÇÃO SUB R0, R1

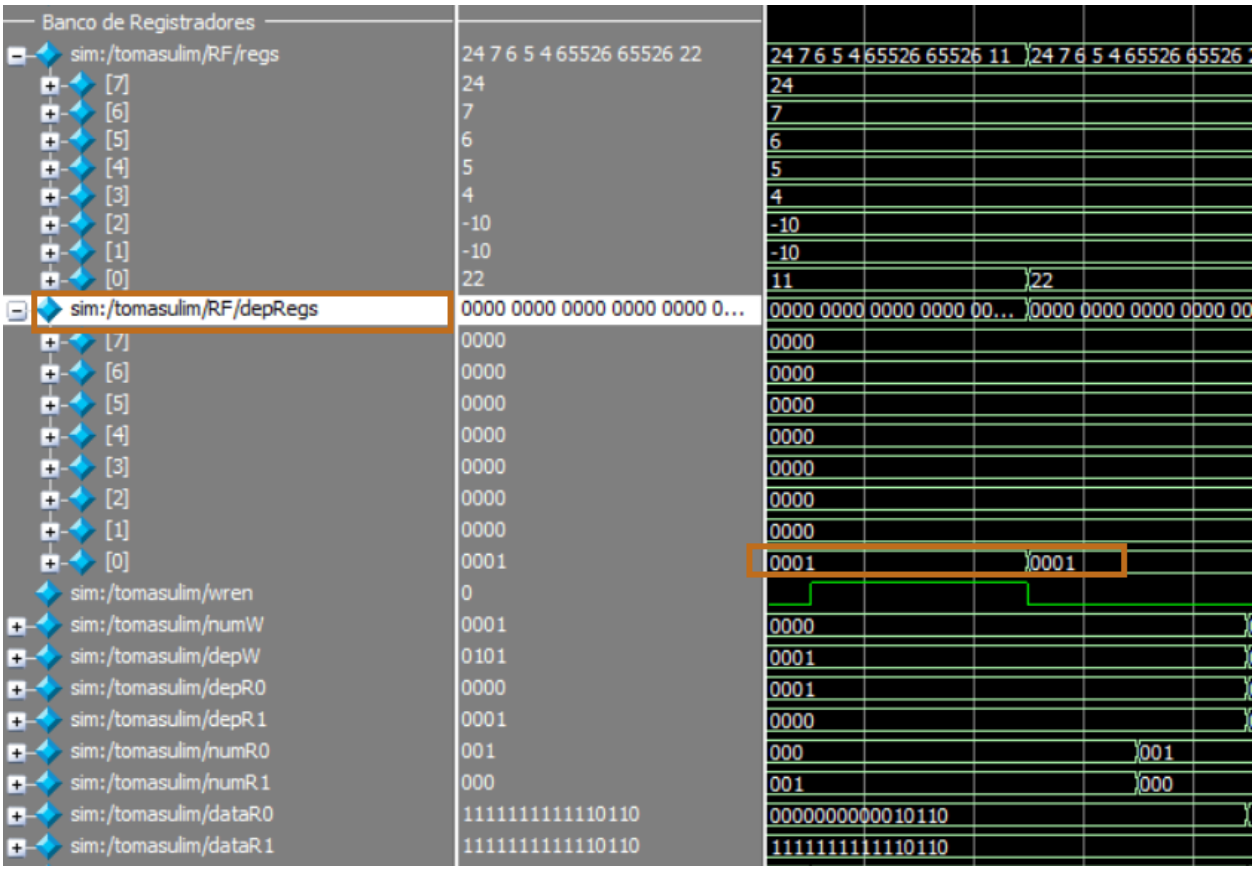

Figura 12: Fazendo a requisição de dados da instrução. Em seguida, coloca o registrador destino dependente da estação de reserva.

A instrução é encaminhada para a estação com identificação 0001(1 em decimal), que é uma estação para adição e subtração. Além disso, são encaminhados os valores contidos nos registradores e suas respectivas dependências (depRegs).

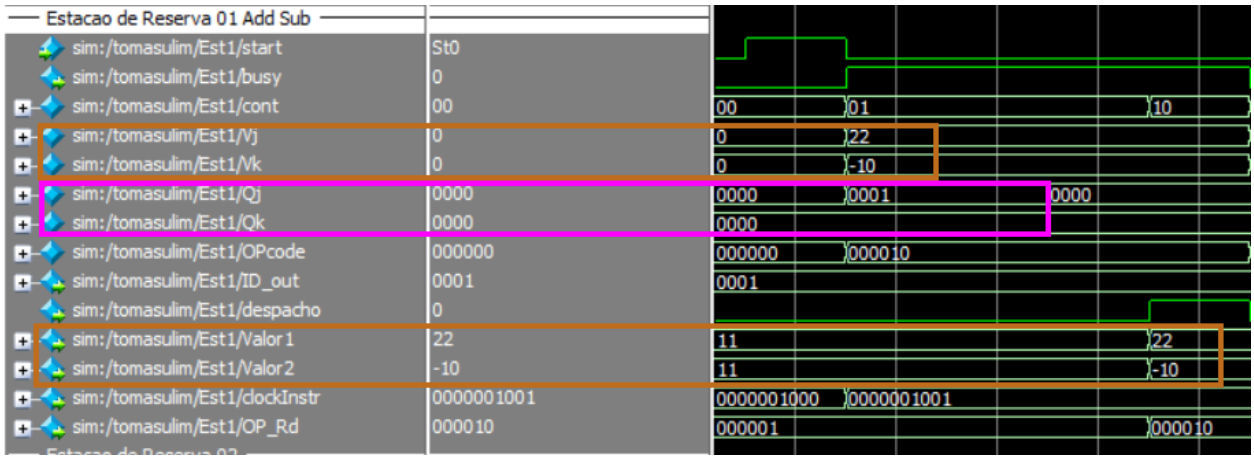

Figura 13: Os valores são armazenados em VJ(R0), VK(R1). Já as dependências em QJ e QK. Com a atualização dos valores os dados podem ser enviados para a Unidade Aritmética.

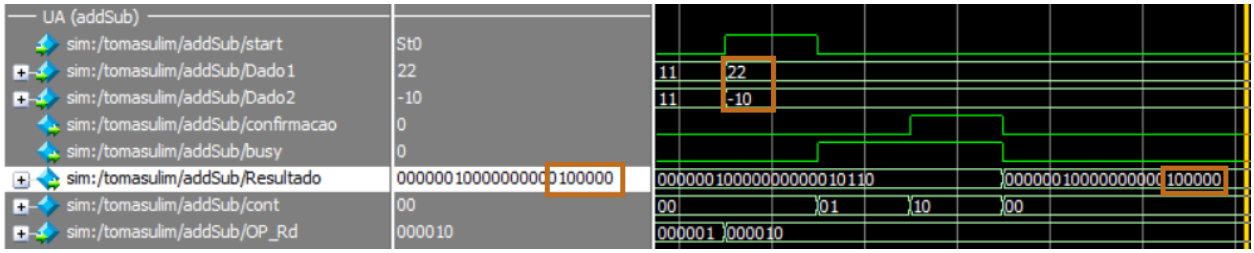

Figura 14: Os recebidos (R0 e R1) resultam na operação R0 - R1 que tem como valor final 32 (100000).

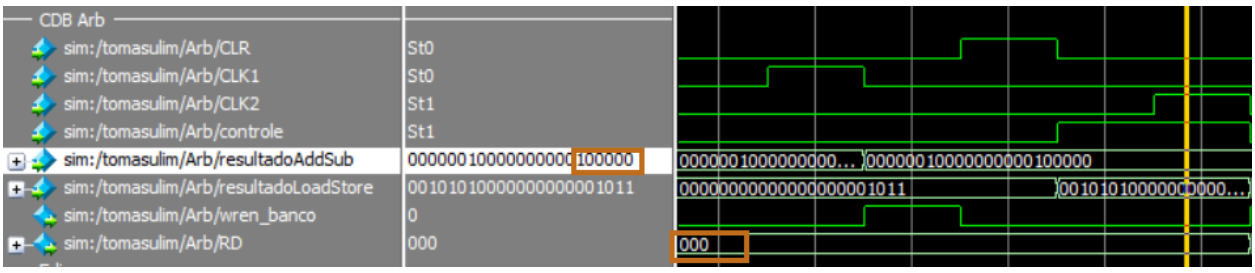

Figura 15: O dado calculado é enviado ao CDB Arbiter que prepara a escrita no registrador destino.

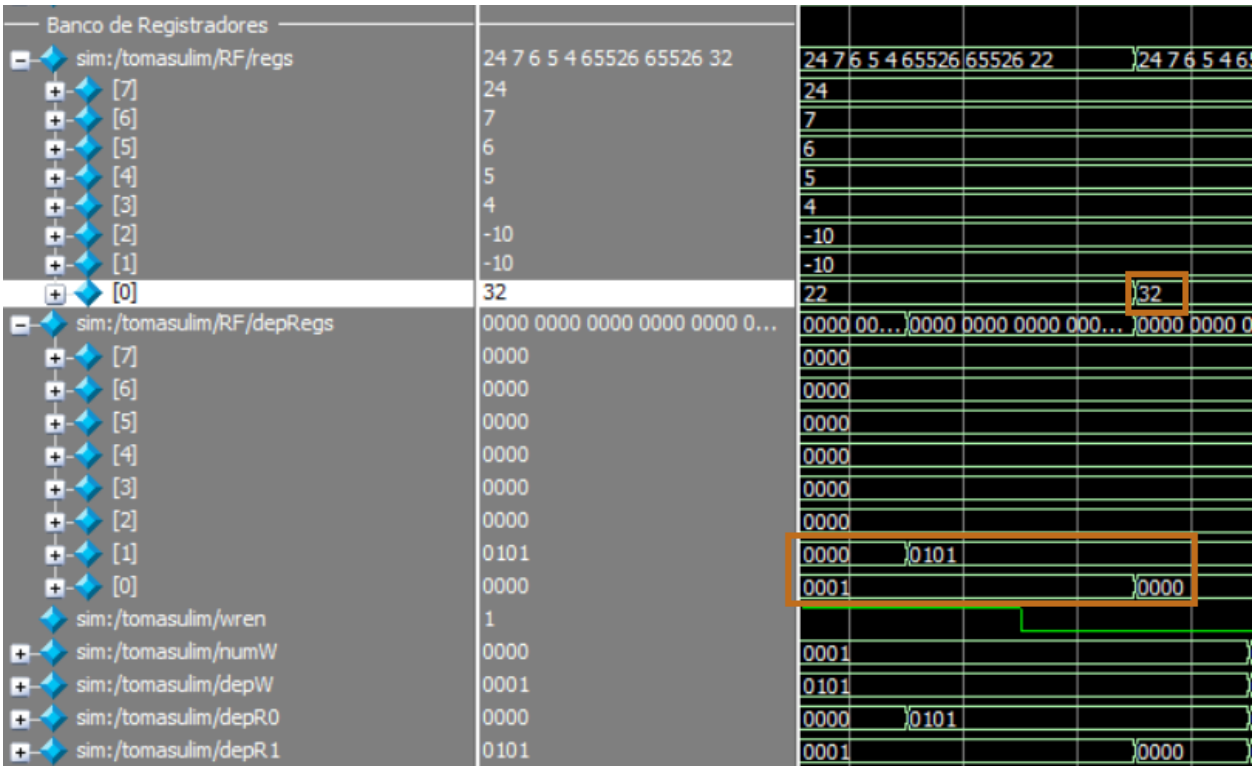

Figura 16: A atualização do dado é feita através da escrita no CDB e tornando o habilita escrita do banco de registradores igual a 1.

3.2 CONFLITO NO CDB

Como não foi observado nenhum caso em que as duas unidades aritméticas emitissem um resultado no mesmo momento, foi simulado um caso em que isso iria ocorrer. A política adotada para casos como este, se baseou em escrever no CDB a instrução mais antiga, em que essa identificação de tempo de instrução acompanha em todas as etapas trabalhadas.

A variável clk_instAS é referente à instrução de ADD/SUB que chegou no CDB Arbiter e a clk instLS é referente à instrução de LOAD/STORE. No caso, como a instrução de Add/Sub possui menor clock instruction, o resultado referente é encaminhado para o CDB.

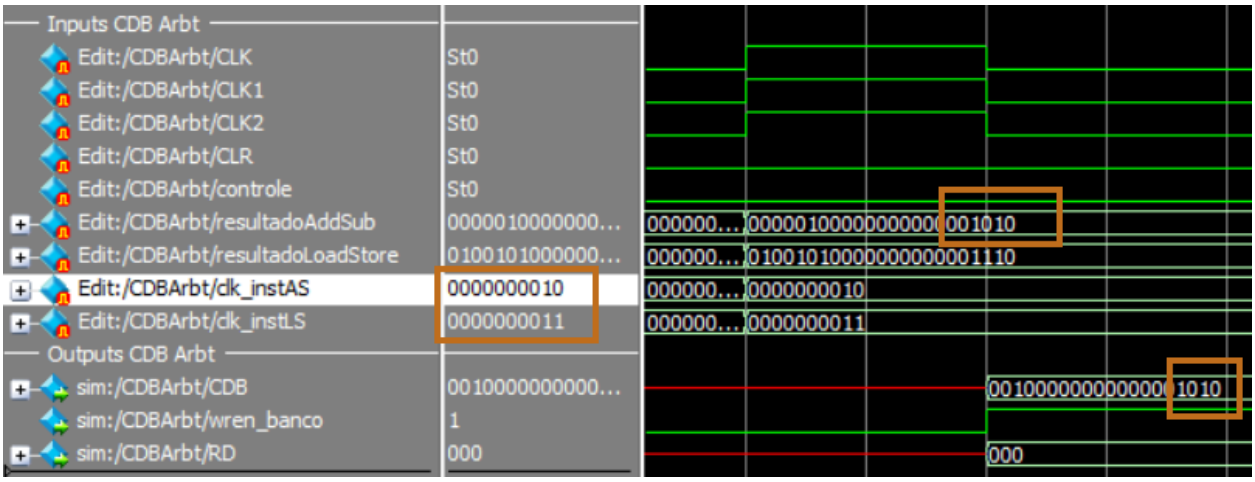

Figura 17: Exemplo da política adotada ao acontecer um conflito no CDB.

3.3 CONFLITO ESTRUTURAL

Para simular um conflito estrutural, foram retiradas as estações 2,3,4 das instruções de adição e subtração. As instruções carregadas na fila foram:

1. ADD R0, R7 (1 + 24 = 25)
2. ADD R0, R0 (25 + 25 = 50)
3. SUB R1, R7 (2 - 24 = -22)

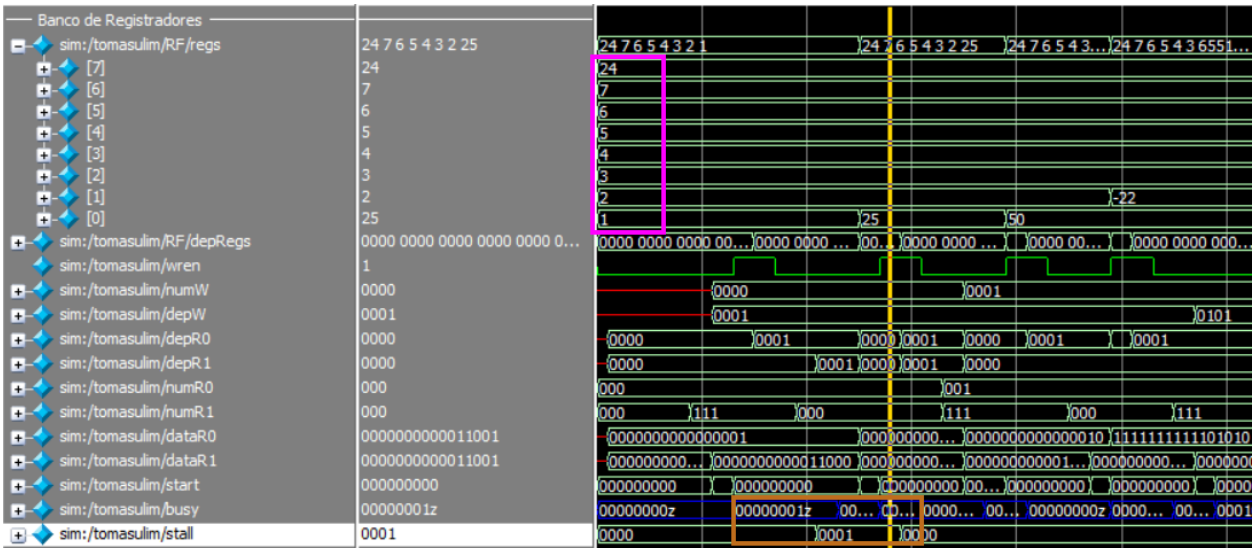

Figura 18: Exemplo de Hazard Estrutural na estação de reserva de Add/Sub.

Em destaque na Figura 18, temos os valores iniciais dos registradores, apresentados em rosa e o stall, em laranja, realizado por não existir estação de reserva disponível, visto que foram retiradas as demais. Ao ocorrer o stall, todas as etapas anteriores sofrem o mesmo efeito para que nenhum dado seja perdido ou mal manipulado. De fato, o stall ocorreu corretamente, já que os resultados esperados foram inseridos após cada operação.

3.4 TESTE GERAL

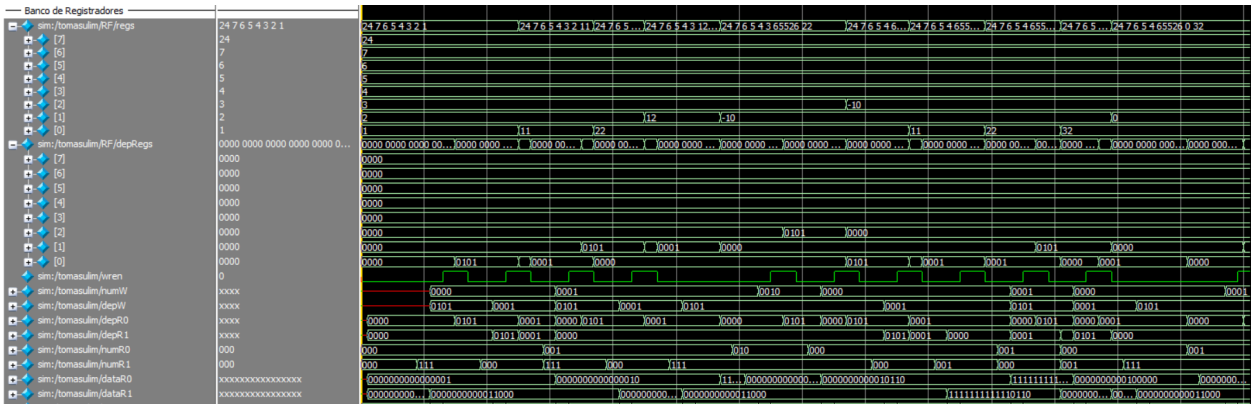

Figura 19: Print contendo o teste geral do algoritmo.

4 DIFICULDADES, SUGESTÕES E CONSIDERAÇÕES FINAIS

Como dificuldades a dupla notou que não são fornecidas informações suficientes para a implementação do algoritmo. O livro não apresenta detalhes refinados sobre a implementação e o aluno acaba tendo que inferir ou pesquisar, o que pode não atender o desejado tendo-se em vista a proposta da prática.

A dupla recomenda que desde o início fiquem definidos todos os detalhes relacionados à prática, como a totalidade do enunciado e o código que será utilizado para avaliação. Além disso, também há a recomendação de que se sofistique o enunciado com mais detalhes, como foi visto na segunda prática.

A implementação desse algoritmo possibilitou um maior aprendizado sobre *design* lógico, bem como entender mais como funciona o algoritmo para a disciplina teórica. Implementar um projeto como esse é vencer um desafio.

REFERÊNCIAS

[1] D. A. Patterson and J. L. Hennessy. *Computer Architecture: A quantitative approach.* Elsevier, 2011.

[2] G. M. Prabhu. Computer architecture tutorial: Dynamic scheduling techniques. `http://web.cs.iastate.edu/~prabhu/Tutorial/PIPELINE/dynamSchedTech.html`, oct 2018.

[3] A. Tong. Dynamic scheduling. `https://www.cs.umd.edu/~meesh/cmsc411/website/projects/dynamic/intro.html`, oct 2018.

[4] Wikipedia contributors. Tomasulo algorithm — Wikipedia, the free encyclopedia. `https://en.wikipedia.org/w/index.php?title=Tomasulo_algorithm&oldid=865217699`, 2018. [Online; accessed 23-October-2018].