Implementação simulada da arquitetura Sagui em pipeline

Gabriel G. de Brito

22 de abril de 2024

1 Introdução

Esse trabalho apresenta a implementação no simulador Logisim Evolution[1] da arquitetura Sagui em *pipeline*. A implementação foi um sucesso e todos os requisitos foram atendidos[2].

2 O Sagui

A figura 1 apresenta o diagrama do processador.

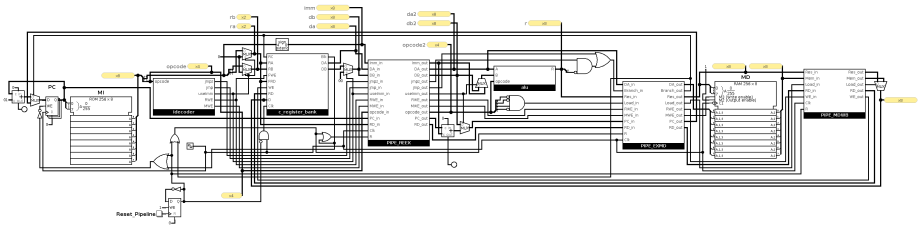

Figura 1: Diagrama do Sagui.

O circuito principal do Sagui, além de unir os circuitos internos, realiza o controle do *pipeline* e da execução do programa, colocando uma instrução sem efeitos colaterais quando um salto ocorre ou o banco de registradores bloqueia a execução de uma instrução.

Os circuitos internos possuem as seguintes responsabilidades:

- *idecoder*: Cria sinais de controle para identificar se a instrução realizará um salto condicional ou incondicional, se irá utilizar registradores ou imediato e se escreverá no banco de registradores ou na memória.

[1] `https://github.com/logisim-evolution/logisim-evolution`

[2] Os requisitos foram superados, inclusive, pois o trabalho permitia a implementação monociclo.

- *c_ register_ bank*: Implementa um banco de registradores com bloqueio de leitura.
- *alu*: Implementa a Unidade Lógica Aritmética (ULA).
- *PIPE_ REEX*, *PIPE_ EXMD*, e *PIPE_ MDWB*: separam os 4 estágios do *pipeline*.

3 Unidade Lógica Aritmética

A figura 2 apresenta o diagrama da Unidade Lógica Aritmética.

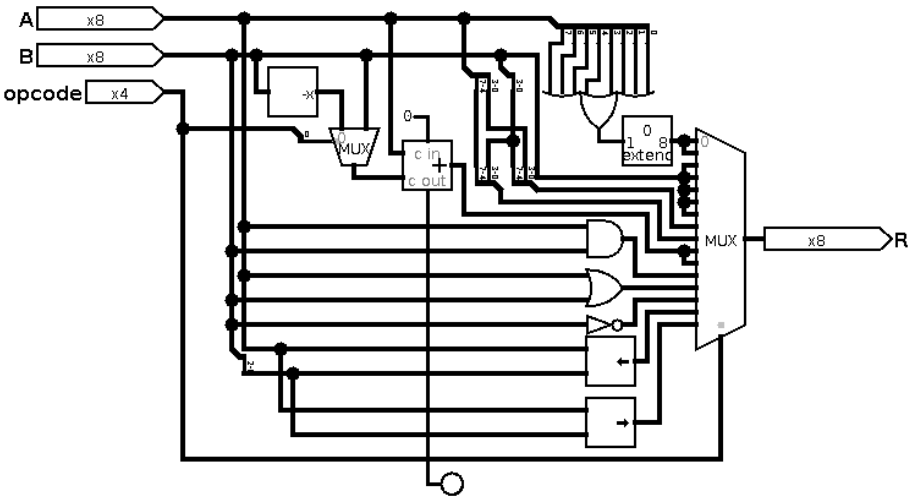

Figura 2: Diagrama da Unidade Lógica Aritmética.

Ela seleciona a operação a ser realizada através do próprio *opcode* da instrução, pois a arquitetura possui poucas instruções e a maioria delas utiliza o resultado da ULA. As possíveis operações são, portanto:

1. Comparação da entrada A com 0: *opcodes* 0 e 1, para os saltos condicionais. Propaga 1 caso A diferente de 0.
2. Propagação da entrada B: *opcodes* 2 a 6. São instruções que não necessitam de operações nas entradas da ULA, porém utilizam o valor de B para realizar saltos, mover o valor de B para outro registrador e acessar a memória.
3. Concatenação da região alta de B com a baixa de A: implementa a instrução *movh* para o *opcode* 7.
4. Concatenação da região baixa de B com a alta de A: implementa a instrução *movl* para o *opcode* 8.

5. Soma: *opcodes* 9 e 10. Realiza a soma ou a subtração, no caso em que multiplexadores trocam a entrada 0 do *carry* do somador para 1 e a entrada B para o complemento do valor de B.

6. As restantes operações são implementadas para cada um do resto dos *opcodes*: E lógico, OU lógico, negação da comparação descrita no item 1, *shift* para a esquerda e *shift* para a direita.

4 Decodificação das instruções

O circuito *idecoder* realiza a decodificação das instruções utilizando pura lógica binária[3]. A figura 3 apresenta o módulo *idecoder*.

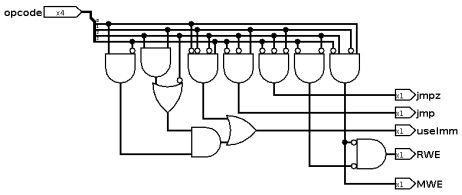

Figura 3: Decodificação das instruções.

5 *Pipeline*

O *pipeline* possui 4 estágios. Sendo eles:

1. *Fetch*, *decode* e busca de dados;

2. Execução;

3. Acesso à memória;

4. *Write back*.

O estado intermediário entre os estágios é salvo em módulos nomeados na forma *PIPE_ xxxx*. Um exemplo é apresentado na figura 4.

Quando o estado precisa ser "zerado", ou seja, uma instrução sem efeitos colaterais precisa ser salva, os registradores que podem causar um efeito colateral são salvos com um valor conveniente, através de um multiplexador.

[3]O ato de criar e preencher uma memória de controle é genericamente visto como mais simples que o ato de criar lógica binária para decodificar instruções, porém o autor discorda. O primeiro ato desperta mais preguiça nele do que o segundo.

6 Programa de teste

Para testar a implementação, dois programas em linguagem Assembly foram criados. Para carregá-los na máquina, geraram-se dois arquivos com as instruções montadas utilizando-se a funcionalidade de *dump* do emulador EGG[4]. Os programas são distribuídos junto à implementação e esse documento, com os nomes de "test.txt"e "soma-vetor.txt". Os respectivos arquivos para serem carregados no simulador são "test-bd.txt"e "soma-vetor-bd.txt".

Como o montador do emulador utilizado não corrige as etiquetas, visto que a maioria dos saltos precisa ser realizada a partir de registradores com endereços, carregados via duas instruções *movl* e *movh*, todos os endereços foram computados manualmente. Para facilitar a tarefa, um script em linguagem AWK foi criado (distribuído com o nome de "addr.txt"). Qualquer sistema UNIX com uma implementação compatível da linguagem pode executá-lo. Para tal, usa-se a seguinte linha de comando: `cat <arquivo assembly> | awk -f addr.txt`.

Para iniciar a máquina, primeiro deve-se inserir instruções sem efeito colateral no *pipeline*. Para isso, liga-se o modo "Reset Pipeline"com o botão *Reset_pipeline* no circuito do Sagui e executa-se 1 ciclo de *clock*. Após o desligamento do modo "Reset Pipeline"a máquina estará pronta, e os ciclos subsequentes executarão o programa da memória de instrução.

[4] `https://github.com/gboncoffee/egg`

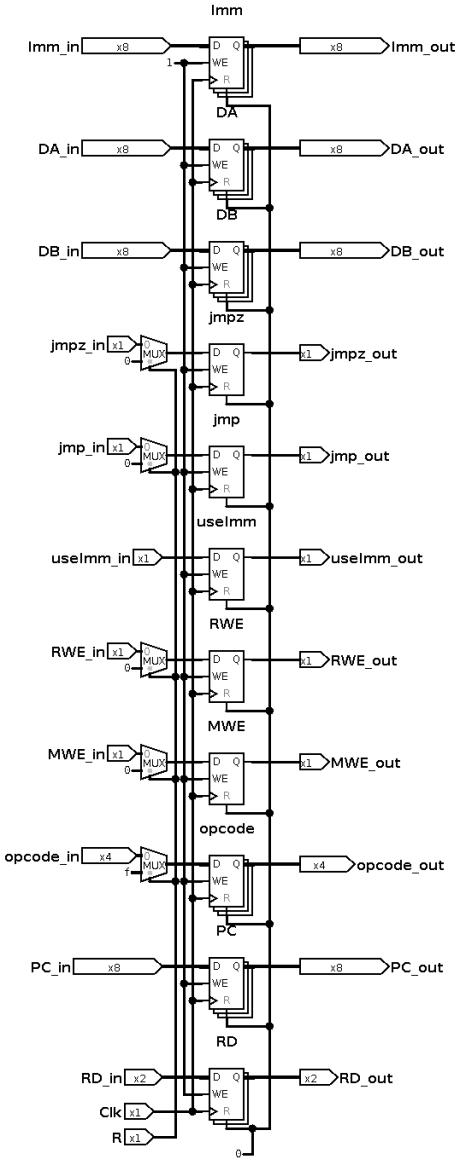

Figura 4: Circuito intermediário "*PIPE_REEX*", entre os estágios 1 e 2

.