Implementação Simulada de um Subconjunto da Microarquitetura RISC-V de 32 Bits

Gabriel G. de Brito

July 27, 2024

1 Introdução

Esse trabalho apresenta o processo de implementação de um subconjunto de instruções da microarquitetura de processadores RISC-V de 32 bits em um simulador de lógica digital. Para provar a corretude da implementação, ela deve rodar um programa em Assembly que computa a sequência de fibonacci[?] em um vetor em memória.

Tanto a microarquitetura quanto o programa foram implementados com sucesso, inclusive com os requisitos sendo excedidos.

2 A Microarquitetura

Os requisitos do trabalho pedem que sejam implementados todos os registradores da especificação da microarquitetura e as seguintes instruções[?], no simulador de lógica digital em CMOS Digital[?]:

- Instruções Aritméticas sem Imediato
  - `add` Adição (+)
  - `sub` Subtração (−)
  - `xor` OU Exclusivo lógico (⊕)
  - `or` OU lógico (+)
  - `and` E lógico (·)
  - `sll` Shift lógico para esquerda (<<)
  - `slt` Menor que (<)
- Instruções Aritméticas com Imediato
  - `addi` Adição (+)
  - `xori` OU Exclusivo lógico (⊕)
  - `ori` OU lógico (+)
  - `andi` E lógico (·)
  - `slli` Shift lógico para esquerda (<<)
  - `slti` Menor que (<)
- Instruções de Manipulação de Memória
  - `lw` Carrega palavra
  - `sw` Salva palavra
- Instruções de Controle de Fluxo
  - `beq` Break se igual a (=)
  - `bne` Break se diferente de (≠)
  - `blt` Break se menor que (<)
  - `bge` Break se maior ou igual a (≥)
- Instruções de Salto
  - `jal` Salto para imediato
  - `jalr` Salto para registrador

Além disso, os requisitos pedem uma implementação em modelo de monociclo, e permite que o endereçamento seja feito palavra-a-palavra ao invés de byte-a-byte, por ser mais viável no simulador utilizado. A implementação descrita aqui utiliza endereçamento byte-a-byte.

3 Banco de Registradores

O banco de registradores é um módulo com seis entradas e duas saídas:

As entradas *RS1i* e *RS2i* selecionam quais registradores serão conectados às saídas *RS1* e *RS2*, respectivamente. A entrada *RSA* seleciona um

registrador para ter seu valor alterado, e a entrada *RDA* contém esse valor. Além disso, a entrada de 1 bit *WR* deve ser 1 para que haja escrita. A outra entrada de 1 bit, *CLK*, deve ser conectada ao clock da máquina.

As três entradas que endereçam registradores são entradas de 5 bits. Como cada registrador armazena 32 bits, as entradas e saídas de dados possuem esse tamanho.

Para selecionar os registradores para as saídas, usam-se dois multiplexadores. Para selecionar o registrador que será alterado, usa-se um demultiplexador no sinal de escrita.

Nota-se que o registrador endereçado por zero não é um registrador, mas sim uma constante. Isso se dá pois na microarquitetura esse registrador não é afetado por instruções de escrita, e sua leitura sempre deve resultar em zero.

4 Unidade Lógica e Aritmética

A Unidade Lógica e Aritmética (ULA) é um módulo com quatro entradas e duas saídas.

As entradas A e B possuem 32 bits cada, sendo os operandos a serem utilizados na operação. As entradas *OP3* e *OP7* possuem 3 e 7 bits cada, respectivamente, e selecionam a operação a ser realizada pela ULA. A maneira com que a seleção é feita é pensada para simplificar a lógica necessária em outras partes do processador, devido à maneira com que as instruções são codificadas.

A entrada *OP3* está diretamente ligada à seleção de um multiplexador, sendo o principal mecanismo de seleção da operação. A saída do somador da ULA é selecionada tanto quando *OP3* é 0 quanto 2. Porém, no caso de *OP3* ser 2, o bit mais significativo é utilizado para que a saída seja 0 ou 1, pois essa operação se trata de um "menor que". Subtraindo B de A, o resultado é negativo caso A seja menor que B, situação em que o bit mais significativo do resultado será 1.

Além disso, um circuito acima do somador é utilizado para controlar se o mesmo fará uma adição normal ou uma subtração. Quando *OP3* é 0, a operação pode ser tanto uma soma quanto uma subtração, dependendo do estado de *OP7*. No caso da soma, o último é 0, e no caso da subtração, é 32. Como a operação desejada é uma subtração tanto no caso da subtração comum como no "menor que", verificamos se *OP3* é 2 ou se *OP7* é 32 (esse último podemos verificar com apenas um bit do número, pois é a única operação que a ULA faz onde *OP7* não é zero). Caso afirmativo, o somador utiliza o complemento do número em B ao invés do original, e o valor 1 na entrada do *carry*. Devido às propriedades aritméticas do complemento de 2[?], o resultado é a subtração de A por B.

A saída de 32 bits R possui o resultado da operação em si, e a saída de 1 bit *Zero* é construída a partir de uma operação NOR em todos os bits do resultado. Dessa forma, essa saída será 1 apenas quando o resultado for exatamente 0.

5 Sinais

Certos aspectos do funcionamento do processador são controlados por um módulo chamado de processador de sinais. Esse componente possui uma entrada de 7 bits por onde recebe o *opcode* da instrução atual e outra entrada com os demais 25 bits. Esse módulo produz as seguintes saídas:

- 3 bits na saída *Func 3*, usado pela ULA na entrada *OP3* e no controle de fluxo;
- 7 bits na saída *Func 7*, usado pela ULA na entrada *OP7* e no controle de fluxo;
- 1 bit na saída *Load PC*, que indica se o banco de registradores deve receber como dado de escrita o valor do PC;
- 1 bit na saída *Load ALU*, que indica se o banco de registradores deve receber como dado de escrita o resultado da ULA;
- 1 bit na saída *Reg Store*, que indica se o sinal de escrita no banco de registradores deve ser ativo;
- 1 bit na saída *Mem Store*, que indica se o sinal de escrita da memória deve ser ativo;

- 1 bit na saída *Lau IMM*, que indica se a ULA deve receber na entrada B um valor imediato ou o valor da saída *RS2* do banco de registradores;

No caso em que ambos *Load PC* e *Load ALU* não estão em 1, o banco de registradores deve receber a saída da memória.

Como todas as instruções que devem ser implementadas tem *opcode* começando com $0b11$, os dois primeiros bits são ignorados. Quanto ao resto, a lógica é a seguinte:

- Caso o quinto bit seja 1, a instrução é aritmética de tipo R ou I. Nesses casos, ambos *Func 3* e *Func 7* estão codificados na instrução da mesma maneira que seus equivalentes na ULA, portanto basta propagá-los. Da mesma forma, o banco de registradores deve receber o resultado da ULA como dado de escrita, portanto *Load ALU* é configurado para 1;
- Caso o terceiro ou quinto bit seja 1, ou o sexto seja 0, a instrução precisa escrever no banco de registradores, então *Reg Store* é configurado de acordo;
- Caso o terceiro bit seja 1, a instrução é um salto, situação em que *Load PC* deve ser 1;
- Caso o sexto bit seja 1 e ambos quinto e sétimo 0, a instrução é um *store*, portanto *Load Mem* deve ser 1 para que o banco de registradores carregue dados da memória;
- Caso o terceiro bit seja 0 e ambos sexto e sétimo 1, a instrução é um *branch*. Nessa situação, a ULA deve realizar uma subtração de qualquer forma, portanto *Func 3* é configurado para 0 e *Func 7* para 32.
- Por fim, a ULA deve receber um imediato na entrada B ao invés da saída de *RS2* nas situações em que a instrução seja um salto, um *load*, um *store*, ou uma operação aritmética com imediato. Para que o sinal *Lau IMM* seja configurado de acordo, ele é ligado à uma porta NAND que recebe como inputs a negação dos terceiro e quarto bits, um OR entre o quinto e o sétimo, e o sexto.

6 Processamento de Imediato

Como há diferentes formatos de instrução, é necessário construir valores imediatos de maneira diferente para cada formato. O módulo de processamento de imediato recebe os 32 bits da instrução na entrada *Instruction* e retorna um imediato de 32 bits na saída *IMM*:

O módulo possui 4 circuitos separados para cada formato de instrução, e usa portas de transmissão para garantir que haja somente uma entrada conectada à saída. A montagem dos imediatos é extremamente simples, pois consiste em reorganizar a ordem dos bits nas regiões da instrução, adicionar uma constante 0 como o primeiro bit dos imediatos que sofrem shift (formatos B e J), e extender o imediato com sinal.

- A instrução é de tipo I caso seja uma instrução aritmética com imediato ou uma instrução de *load*, ou seja, com *opcode* $0b0010011$ ou $0b0000011$. Para averiguar, usa-se a expressão $(!O_3O_2 \oplus O_4)+!(O_5 + O_6)$. Para os *opcode* válidos no subconjunto implementado na CPU, essa expressão é verdadeira somente para as instruções de tipo I.
- A instrução é de tipo S somente no caso em que é um *store*. Essas são as únicas instruções com quinto e sétimo bit 0 e sexto bit 1, portanto isso é verificado.
- As instruções tipo B, da mesma maneira, são as únicas com sexto e sétimo bits 1 e terceiro e quinto bits 0.
- As instruções tipo J, igualmente, são as únicas com terceiro e quarto bits 1 e quinto bit 0.

Dessa maneira, a construção dos imediatos é simples e legível.

7 Controle de Fluxo

Um módulo separado processa sinais que controlam o fluxo de execução do programa, ou seja, indicam o comportamento do PC (*program counter*), um

registrador separado do banco de registradores que guarda o endereço da instrução atual.

O módulo possui as entradas *opcode*, de 7 bits, *Zero out*, de 1 único bit (conectada à saída *Zero* da ULA), *Alu out*, de 32 bits (conectada à saída da ULA), e *Func 3*, de 3 bits. É importante notar que a entrada *Func 3* não está ligada à saída *Func 3* do processador de sinais. Ela está ligada diretamente à região "func 3" da instrução, que está sempre no mesmo lugar.

A saída *PC Alu* indica se o próximo valor do PC deve ser copiado do resultado da ULA, e a saída *+ IMM* indica se o próximo valor do PC deve ser o resultado da soma do valor atual ao imediato. Ambas as saídas 0 indicam que o próximo valor deve ser o atual somado a 4, pois instruções tem 32 bits, portanto 4 bytes.

A lógica começa verificando se a instrução é uma instrução de salto ou *branch*. Para isso, verifica-se se os dois últimos bits dela estão em 1, pois esses tipos de instrução são os únicos implementados onde isso ocorre[**?**]. Caso falso, ambas as saídas automaticamente serão zero. Esse efeito é alcançado ao ligar o teste (que usa uma simples porta AND) à seleção de dois multiplexadores logo antes de cada saída, onde a opção 0 é ligada à constantes 0.

A saída *PC Alu* deve ser 1 somente quando a instrução é um `jalr`. Portanto, liga-se a saída à uma porta AND que verifica se os quarto e quinto bit são 0 e o terceiro é 1.

A saída *+ IMM* deve ser 1 caso a instrução seja um *branch* bem-sucedido ou um `jal`. Liga-se uma porta OR à saída, comparando ambos os testes. Caso a instrução seja `jal`, o quarto bit do *opcode* é 1, portanto o teste é feito simplesmente ligando esse bit à entrada da porta OR. Caso a instrução seja um branch, os dois últimos bits devem ser 1 e o terceiro deve ser 0. Liga-se tudo isso num AND de 4 entradas, junto com o teste do sucesso do *branch*.

Para testar o sucesso do *branch*, liga-se *Func 3* à seleção de um multiplexador. *Func 3* é 2, 5, 6 e 7 somente em situações não implementadas, portanto essas opções são ligadas em constantes 0. Outras situações estão listadas a seguir. Todas dependem do fato demonstrado na seção de **??** de que a ULA realizará uma subtração:

- *Branch* em igual (*Func 3* = 0): é realizado se a saída *Zero* da ULA é 1, portanto basta ligar essa saída diretamente na opção do multiplexador.
- *Branch* em diferente (*Func 3* = 1): é a negação da anterior.
- *Branch* em menor quê (*Func 3* = 3): é realizado caso o último bit do resultado da ULA seja 0, por conta das propriedades de complemento de 2[**?**]. Portanto liga-se a negação desse bit na opção do multiplexador.
- *Branch* em maior ou igual quê (*Func 3* = 4): é a negação da anterior.

8 Memórias

Os requisitos do trabalho não exigiam o endereçamento "byte-a-byte" nas memórias, porém sua implementação com os recursos do simulador é interessante. Como não é possível configurar os componentes de memória *built-in* para lerem/escreverem em mais de um byte por vez, foram criados módulos que abstraem o funcionamento de 4 chips de memória, de maneira a emular um único chip que endereça byte-a-byte, porém lê e escreve uma palavra por vez.

A entrada *S* é conectada diretamente à entrada *sel* de cada chip, para que funcione de maneira transparente. A entrada de endereço é tratada antes de cada chip, utilizando aritmética modular. Como cada chip guarda um byte de um valor de 4 bytes, os dois menores bits do endereço são verificados para adicionar 1 ao endereço dos chips em módulo menor que o endereçado.

A memória RAM funciona da mesma forma, porém a mesma lógica é utilizada para o valor de entrada. As entradas de *clock* e sinais de escrita e leitura são passadas de maneira transparente.

9 Circuito Principal

O circuito principal da CPU possui, além da ligação de todos os componentes, lógica que lida com sinais e o controle de fluxo do programa.

Todos os sinais que não são tratados internamente em algum módulo são tratados por meio de multiplexadores de 1 bit, como os sinais *Load PC* e *Load ALU*. Além disso, a instrução especial de *halt*, que foi requerida no trabalho, é implementada como uma porta AND entre os dois primeiros bits da instrução, ligada ao sinal de escrita do PC. Como o PC é implementado utilizando um módulo "registrador" do simulador, ele possui uma entrada que controla a escrita. Todas as instruções da especificação da microarquitetura que devem ser implementadas no trabalho possuem os dois primeiros bits como 1, portanto qualquer instrução sem os dois primeiros bits 1 irá desativar a escrita do PC, que ficará o tempo todo na mesma instrução.

10 O programa

O trabalho exigia um programa em Assembly que fosse montado e rodado na CPU projetada. Esse programa deve criar os 20 primeiros números da sequência de Fibonacci em um vetor iniciando na posição 0 da memória, baseando-se no programa em C a seguir:

```
void fib(int *vet, int tam, int elem1, int
elem2)
{
        int i;
        vet[0] = elem1;
        vet[1] = elem2;
        for (i = 2; i < tam; i++) {
                vet[i] = vet[i - 1] + vet[i
- 2];
        }
}

int main(void)
{
        int vet[20];
        fib(vet, 20, 1, 1);
}
```

O programa pode sofrer otimizações e pode executar passos adicionais para auxiliar a verificação da corretude, porém deve necessariamente utilizar uma função para `fib(int*, int, int, int)` e seguir as convenções de chamada de função.

O programa em Assembly desenvolvido foi o seguinte:

```
.text
        add a0, zero, zero
        addi a1, zero, 20
        addi a2, zero, 1
        addi a3, zero, 1
        jal ra, fib

        addi t0, zero, 80
        add t1, zero, zero
        print: beq t1, t0, exit
        lw t2, 0(t1)
        sw t2, 0(t1)
        addi t1, t1, 4
        jal zero, print

exit: halt

fib:    sw a2, 0(a0)
        sw a3, 4(a0)

        slli t0, a1, 2
        add t0, t0, a0
        addi a0, a0, 8

loop:   blt t0, a0, ret
        add t1, a2, a3
        sw t1, 0(a0)
        add a2, a3, zero
        add a3, t1, zero
        addi a0, a0, 4
        jal zero, loop

ret:    jalr zero, 0(ra)
```

Além do requerido, após a chamada da função, um loop carrega os valores criados no vetor para um registrador, e depois salva novamente na memória. Como não é possível verificar o resultado da computação no módulo de memória após o término do programa, isso auxilia na verificação da sua corretude. Entre as otimizações realizadas, destaca-se o uso de registradores como "cache" para os

elementos do vetor durante o loop, que serão possivelmente utilizados na iteração seguinte.

Para testagem e montagem do programa, foi utilizado o simulador RARS[?] e sua opção de "dump" binário. Após gerado o arquivo binário com as instruções, foi utilizado o seguinte programa em C para gerar quatro arquivos diferentes, para que cada um seja carregado por um dos chips de memória do módulo de ROM:

```
#include <stdio.h>

int main(void)
{
        unsigned char c;
        int i;
        FILE *files[4];
        files[0] = fopen("mod0.bin", "wb");
        files[1] = fopen("mod1.bin", "wb");
        files[2] = fopen("mod2.bin", "wb");
        files[3] = fopen("mod3.bin", "wb");

        if (!(files[0] && files[1] &&
files[2] && files[3]))
                return 1;

        i = 0;
        c = getchar();
        while (!feof(stdin)) {
                fputc(c, files[i % 4]);
                i++;
                c = getchar();
        }

        fclose(files[0]);
        fclose(files[1]);
        fclose(files[2]);
        fclose(files[3]);

        return 0;
}
```

Como o simulador não possui uma instrução de *halt*, para montar o programa essa instrução foi trocada pela instrução `add zero, zero, zero`. A escolha dessa instrução específica foi devido ao fato de ser composta inteiramente por bits 0 exceto pelo *opcode*. Assim foi possível, após a montagem do programa com o simulador, abrir o arquivo gerado em um editor de arquivos binários (no caso, o *Okteta*, do projeto KDE[?] e alterar o *opcode* para ser todo 0.

Com todos os arquivos prontos, foi possível executar o programa com sucesso na CPU projetada.

11 Conclusão

Os requisitos do trabalho foram completos com sucesso, inclusive sendo excedidos. A CPU projetada no simulador computou corretamente o programa requerido.