Implementação da Arquitetura ReduxV em Verilog

Fernando de Barros Castro

Setembro 2024

1 Introdução

O projeto de processador monociclo implementa uma ISA do tipo load-store com instruções de 8 bits. As instruções são armazenadas em uma memória separada da RAM, como modelo havard de processador.

2 Program Counter

- Falha da implementação inicial: O sinal JMP vindo do decode e indicando um salto incondicional usado inicialmente é desnecessário, já que a lógica do módulo foi reduzida a mudar o PCSRC para 110 (PC = PC + 1) caso pcsrc indique um branch e o bit Zero for 0 (R[r0] != 0).

3 Decode

- Tabela de Sinais de Controle:

		ULASRC	PCSRC	IMM	REGSRC	MW	RW
brzr	0000	0000	000	0	0	0	0
ji	0001	0000	101	0	0	0	0
ld	0010	0000	110	0	1	0	1
st	0011	0000	110	0	0	1	0
addi	0100	0100	110	1	0	0	1
inc	0101	1100	110	0	0	0	1
brzi	0110	0000	100	1	0	0	0
slf	0111	1110	110	0	0	0	1
not	1000	0000	110	0	0	0	1
and	1001	0001	110	0	0	0	1
or	1010	0010	110	0	0	0	1
xor	1011	0011	110	0	0	0	1
add	1100	0100	110	0	0	0	1
sub	1101	0101	110	0	0	0	1
slr	1101	0110	110	0	0	0	1
srr	1111	0111	110	0	0	0	1

- Implementação feita com uma ROM

4 Unidade Lógica Aritmética

- Detalhes:

  1. Um deslocamento de bits maior ou igual a oito gera a saída 0.
  2. O sinal *ULASRC* é de quatro bits, sendo os três menos significativos usados para controlar o multiplexador e o quarto para determinar se a operação é especial, ou seja, é uma das três instruções adicionais. A construção foi feita assim já que essas instruções não demandam nenhuma unidade extra na ULA.

5 Algoritmo Base

```
sub r0, r0 // zera R0
sub r1, r1 // zera R1
sub r2, r2 // zera R2
sub r3, r3 // zera R3
addi 7     // r0 = 7
addi 7     // r0 = 14
add r0, r0 // r0 = 28
addi 7     // r0 = 35
add r2, r0 // r2 = 35
sub r0, r0 // zera r0
addi 1     // r0 = 1
add r3, r0 // r3 = 1
addi 5     // r0 = 6
addi 5     // r0 = 11
slr r0, r3 // r0 = 22
slr r0, r3 // r0 = 44
add r1, r0 // r1 = 44
sub r0, r0 // zera r0
addi 2     // r0 = 2
LOOP:
slr r0, r3 // r0 *= 2
slr r0, r3 // r0 *= 2
addi 5     // r0 += 5
addi 5     // r0 += 5
store r1, r0 // MEM[r0] = r1
ji 2        // PC += 2
ji -6       // PC -= 6
addi -5     // r0 -= 5
addi -5     // r0 -= 5
srr r0, r3  // r0 /= 2
srr r0, r3  // r0 /= 2
brzr r0, r2 // if (r0 == 0) PC = 35
sub r1, r3  // r1 -= 1
sub r0, r3  // r0 -= 1
ji -8       // PC -= 8
FORA:
ji 0        // PC = PC
```

6 Fibonacci e Instruções adicionadas

```
# int ant = 1;
# int atual = 1;
# int aux;
# int *vet = 32;
#
# vet[0] = 1;
# vet[1] = 1;
# for (int i = 2; i < 10; i++) {
#       aux = ant + atual;
#       v[i] = aux;
#       ant = atual;
#       atual = aux
# }

sub r0, r0  // zera r0
sub r1, r1  // zera r1
sub r2, r2  // zera r2
inc r1, r1  // r1 += 1
inc r2, r2  // r2 += 1
addi 2      // r0 += 2
slf r0, r0  // r0 = r0 * 16
st r1, r0   // MEM[r0] = r1
addi 4      // r0 += 4
st r1, r0   // MEM[r0] = r1
addi 4      // r0 += 4
LOOP:
sub r3, r3  // zera r3
add r3, r1  // r3 += r1
ji 2        // PC += 2
ji -3       // PC -= 3
add r3, r2  // r3 += r2
sub r1, r1  // zera r1
add r1, r2  // r1 += r2
sub r2, r2  // zera r2
add r2, r3  // r2 += r3
st r2, r0   // MEM[r0] = r2
ji 2        // PC += 2
ji -8       // PC -= 8
addi -8     // r0 -= 8
addi -8     // r0 -= 8
addi -8     // r0 -= 8
addi -8     // r0 -= 8
sub r3, r3  // zera r3
inc r3, r3  // r3 += 1
ji 2        // PC += 2
ji -8       // PC -= 8
srr r0, r3  // r0 /= 2
srr r0, r3  // r0 /= 2
inc r0, r0  // r0 += 1
addi -5     // r0 -= 5
addi -5     // r0 -= 5
```

```
brzi 11     // if (r0 == 0) PC += 11
ji 2        // PC += 2
ji -8       // PC -= 8
addi 5      // r0 += 5
addi 5      // r0 += 5
slr r0, r3  // r0 *= 2
slr r0, r3  // r0 *= 2
inc r3, r3  // r3 += 1
slf r3, r3  // r3 *= 16
add r0, r3  // r0 += 32
ji -8       // PC -= 8
FORA:
ji 0        // PC = PC
```

- O algoritmo acima calcula os dez primeiros elementos da sequência de Fibonacci e os guarda em um vetor de inteiros na memória a partir do endereço 32. O desafio consiste na implementação usando quatro registradores, sendo que três deles (r0, r1, r2) não podem ter seus valores alterados e o último serve como auxiliar. Dessa forma, as instruções INC e SLF se complementam para ajudar nessa tarefa, já que o incremento pode ser feito em qualquer registrador e o shift permite escrever nos quatro bits mais significativos mudando apenas os menos significativos. Além disso, a instrução BRZI ajuda muito já que escrever o endereço do rótulo FORA a cada iteração do loop iria deixar o código bem maior e menos eficiente.

7 Assembly para Teste das Instruções

```
sub r0, r0    // r0 = 0
sub r2, r2    // r2 = 0
sub r3, r3    // r3 = 0
addi 1000     // r0 = -8
not r1, r0    // r1 = 7
sub r1, r1    // r1 = 0
add r0, r1    // r1 = -8
xor r0, r0    // r0 = 0
inc r0, r0    // r0 = 1
inc r1, r1    // r1 = 1
st r0, r0     // MEM[1] = 1
ld r1, r1     // r1 = MEM[1]
slf r0, r0    // r0 = 16
brzr r1, r0   // uma iteracao no laco
sub r1, r1    // r1 = 0
ji 1110
inc r1, r1    // r1 = 1
brzi 0011     // cinco iteracoes no laco
srr r0, r1    // 16->8->4->2->1->0
ji 1110
slr r1, r1    // r1 = 2
or r1, r1     // r1 = r1
and r1, r0    // r1 = 0
ji 0
```

- Testbench: Os algoritmos estão no diretório 'memory_files' e para alternar entre eles, a linha 9 de redux_V_TB deve ser alterada:
  - Fibonacci: readmemh("./memory_files/fibonacci.mem", DUT.ci.rom, 0, 47);
  - Algoritmo Base: readmemh("./memory_files/algoritmo_base.mem", DUT.ci.rom, 0, 34);
  - Teste Instruções: readmemh("./memory_files/teste_instrucoes.mem", DUT.ci.rom, 0, 23);