RISC-V de 16 instruções em VHDL: SD16_RISC-V

Trabalho Sistemas Digitais 2025/2

Nome do aluno: _____. Matricula: _____

Vamos projetar, sintetizar e validar por simulação um processador RISC-V (SD16_RISC-V) com apenas **16 instruções (16 opcodes de instruções)** para fins didático.

- Numero de instruções; 16
- Memoria: única de programa e de dados de largura de 16 bits e a profundidade da memória é de 256 posições.
- Registradores: banco de registradores com 8 registradores (r0 a r7) de largura de 16 bits
- PC: 8 bits
- Largura de instrução: 16 bits. Cada instrução cabe em um endereço de memória.

Tipo da instrução (R, I, S, B, J, U, N), onde cada instrução tem formato de 16 bits com **opcode nos 4 bits nos bits menos significativos.**

Note que a ordem dos registradores é sempre rd, rs1 e rs2 para fins de simplificação para que todas as instruções caibam em 16 bits.

Tabela de Instruções RISC-V com 4 LSBs como opcode

CODE rd rs1 rs2 immed/offset

Opcode (bin)	Mnemônico	Tipo	Descrição	Exemplo (* note que os números estão todos em hexadecimal)
0000	ADD	R	rd ← rs1 + rs2	ADD r3, r5, r6. (R3 <= R5 + R6)
0001	SUB	R	rd ← rs1 - rs2	SUB r3, r5, r6. (R3 <= R5 - R6)
0010	AND	R	rd ← rs1 & rs2	AND r3, r5, r6. (R3 <= R5 and R6)
0011	OR	R	rd ← rs1 | rs2	OR r3, r5, r6. (R3 <= R5 or R6)
0100	XOR	R	rd ← rs1 ^ rs2	XOR r3, r5, r6. (R3 <= R5 xor R6)
0101	ADDI	I	rd ← rs1 + imm	ADDI r3, r1, 5 (R3 <= r1 + 5)
0110	ANDI	I	rd ← rs1 & imm	ANDI r3, r1, 5 (R3 <= r1 and 5)
0111	ORI	I	rd ← rs1 | imm	OR r3, r1, 5 (R3 <= r1 and 5)
1000	LW	I	rd ← Mem[rs1 + offset]	LW r3, r5, 5 (R3 <= mem(r5+5)
1001	SW	S	Mem[rs1 + offset] ← rd	SW r3, r5, 5 (mem(r5+5) <= r3
1010	BEQ	B	if rs1==rd pc ← pc + offset	BEQ r3, r5, 4 (if r3 = r5, PC <= pc + 4)
1011	BNE	B	if rs1!=rd pc ← pc + offset	BNE r3, r5, 4 (if r3 != r5, PC <= pc + 4)
1100	JAL	J	pc ← pc+offset rd <= pc+1;	JAL r1, 2 (PC <= PC + 2, r1 <= pc+1)
1101	LUI	U	rd ← “000000”& imm (6 bits)&”0000"	LUI r0, 5 registrador r0 recebe 5 <<<4 = 50
1110	NOP	N	Nenhuma operação	PC <= PC +1
1111	HLT	N	Para a execução	PC <= PC

Campos na Instrução (simplificado para 12 bits mais significativos)

Rd Rs1 Rs2 imm code - (de maneira geral) mas cada instrução tem um pouco de variação conforme a seguir.

Campo	Descrição
`rs1`	Registrador fonte 1 (numero dele ocupa 3 bits na instrução r0- r7)
`rs2`	Registrador fonte 2 (numero dele ocupa 3 bits na instrução r0- r7, para instruções tipo R/S/B)
`rd`	Registrador destino (numero dele ocupa 3 bits na instrução r0- r7, em R/I/U/J types)
`imm`	Valor imediato (offset, constante etc. ocupa 6 bits neste caso pois a instrução toda cabe em 16 bits)

TIPOS DE INSTRUÇÃO de 16-bits E SUAS ESTRUTURAS no SD16_RISC-V

1. Tipo R (Register)

Instruções: `ADD`, `SUB`, `AND`, `OR`, `XOR`

- **ADD rd, rs1, rs2**

```
[15-13] rd | [12-10] rs1 | [9-7] rs2 | [6-4] unused | [3-0] opcode
```

2. Tipo I (Immediate)

Instruções: `ADDI`, `ANDI`, `ORI`, `LW`

- **ADDI rd, rs1, immediate**

```
[15-13] rd | [12-10] rs1 | [9-4] immediate | [3-0] opcode
```

3. Tipo S (Store)

Instrução: `SW`

- **SW rd, rs1, offset**

```
[15-13] rd | [12-10] rs1 | [9-4] offset | [3-0] opcode
```

4. Tipo B (Branch)

Instruções: `BEQ`, `BNE`

```
[15-13] rd | [12-10] rs1 | [9-4] offset | [3-0] opcode
```

5. Tipo J (Jump)

Instrução: `JAL`

```
[15-10] unused | [9-4] offset | [3-0] opcode
```

6. Tipo U (Upper Immediate)

Instrução: `LUI`

```
[15-13] rd | [13-10] unused | [9-4] immediate | [3-0] opcode
```

7. Tipo N (Nulo/Controle)

Instruções: `NOP`, `HLT`

```
[15-4] zeros | [3-0] opcode
```

```
-- RISC-V Simplificado com 16 Instruções usando Máquina de Estados
-- SD16_RISC-V
-- Estilo RTL com memória unificada (dados + instruções)
-- Autor: Versão didática para ensino


library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity RiscV16 is
    Port (
        clk     : in  STD_LOGIC;
        rst     : in  STD_LOGIC;
        saida_temp : OUT STD_LOGIC_VECTOR(15 downto 0);
        halted  : out STD_LOGIC
    );
end RiscV16;

architecture rtl of RiscV16 is

    type state_type is (FETCH, DECODE, EXECUTE, WRITEBACK, HALT);
```

```
    signal state : state_type := FETCH;

    type mem_type is array(0 to 255) of STD_LOGIC_VECTOR(15 downto 0);
    type reg_file_type is array(0 to 7) of STD_LOGIC_VECTOR(15 downto 0);  -- 8 registradores (r0-r7)

    signal PC       : STD_LOGIC_VECTOR(7 downto 0) := (others => '0');
    signal IR       : STD_LOGIC_VECTOR(15 downto 0);
    signal Regs     : reg_file_type := (others => (others => ‘0'));
— inicialização da memoria
    signal Mem      : mem_type := (others => (others => ‘0'));

    signal running  : STD_LOGIC := '1';

    signal opcode   : STD_LOGIC_VECTOR(3 downto 0);
    signal rd, rs1, rs2 : INTEGER range 0 to 7;
    signal imm      : STD_LOGIC_VECTOR(7 downto 0);
    signal offset   : SIGNED(5 downto 0);
     signal alu_out  : STD_LOGIC_VECTOR(15 downto 0);
    signal write_en : STD_LOGIC := '0';
    signal write_addr : INTEGER range 0 to 7;

begin

    process(clk, rst)
    begin
        if rst = '1' then
             -- flag
             running <= ‘1';
            -- registers
            PC <= (others => '0');
            Rd <= (others => '0');
            Rs1 <= (others => '0');
            Rs2 <= (others => '0');
            Imm < (others => '0');
```

```
    Offset <= (others => ‘0');
    IR <= (others => '0');
    Opcode <=  (others => ‘0');
     alu_out <= (others => ‘0’);
     write_en <= '0';
     write_addr <= (others => ‘0’);


    -- inicializa register file com zero em todas as posições
     regS <= (others => (others => ‘0'));

    state <= FETCH;


elsif rising_edge(clk) then

   if running = '1' then
      case state is

         when FETCH =>
            IR <= Mem(to_integer(unsigned(PC)));
            PC <= std_logic_vector(unsigned(PC) + 1);
            state <= DECODE;

         when DECODE =>
             — define a posição de cada operando na instrução
           opcode <= IR(3 downto 0);
           rd     <= to_integer(unsigned(IR(15 downto 13)));
           Rs1    <= to_integer(unsigned(IR(12 downto 10)));
          Rs2    <= to_integer(unsigned(IR(9 downto 7)));
           imm    <= IR(9 downto 4);    -- **** 6 bits
           offset <= signed(IR(9 downto 4)); -- **** 6 bits

            state <= EXECUTE;
```

```
when EXECUTE =>

   case opcode is
      when "0000" => alu_out <= std_logic_vector(signed(Regs(rs1)) + signed(Regs(rs2)));  -- ADD
      when "0001" => alu_out <= std_logic_vector(signed(Regs(rs1)) - signed(Regs(rs2)));  -- SUB
      when "0010" => alu_out <= Regs(rs1) and Regs(rs2); -- AND
      when "0011" => alu_out <= Regs(rs1) or Regs(rs2);  -- OR
      when "0100" => alu_out <= Regs(rs1) xor Regs(rs2); -- XOR
      when "0101" => alu_out <= std_logic_vector(signed(Regs(rs1)) + signed(('0' & imm))); -- ADDI
      when "0110" => alu_out <= Regs(rs1) and ("0000000000" & imm); -- ANDI
      when "0111" => alu_out <= Regs(rs1) or    ("0000000000" & imm);  -- ORI
      when "1000" => alu_out <= Mem(to_integer(unsigned(Regs(rs1)(7 downto 0)) + unsigned(imm(5 downto 0)))); -- LW
      when "1001" => -- SW
         Mem(to_integer(unsigned(Regs(rs1)(7 downto 0)) + unsigned(imm(5 downto 0)))) <= Regs(rd);
         state <= FETCH;
      when "1010" => -- BEQ
         if Regs(rs1) = Regs(rd) then
            PC <= std_logic_vector(signed(PC) + signed(offset(5 downto 0)));
         end if;
         state <= FETCH;
      when "1011" => -- BNE
         if Regs(rs1) /= Regs(rd) then
            PC <= std_logic_vector(signed(PC) + signed(offset(5 downto 0)));
         end if;
         state <= FETCH;
      when "1100" => -- JAL
         alu_out <= std_logic_vector(unsigned(PC));
         PC <= std_logic_vector(signed(PC) + signed(offset(5 downto 0)));
      when "1101" => alu_out <= “000000" & imm&"0000"; -- LUI
     when "1110" => state <= FETCH; -- NOP
      when "1111" => state <= HALT; -- HLT
      when others => null;
   end case;
```

```
                if opcode /= "1001" and opcode < "1010" then
                    write_en <= '1';
                    write_addr <= rd;
                    state <= WRITEBACK;
                end if;

            when WRITEBACK =>

                if write_en = '1' then
                    Regs(write_addr) <= alu_out;
                end if;
                write_en <= '0';
                state <= FETCH;

            when HALT =>
                running <= '0';

        end case;
      end if;
    end if;
  end process;

  halted <= not running;
  saida_temp <= alu_out;

end rtl;
```

EXEMPLOS DE PROGRAMAS

Exemplo 1: Soma de dois valores e grava-se na memória

Relembrando…

ADDI [15–13] rd | [12–10] rs1 | [9-4] immediate | [3–0] opcode)

SW [15–13] rd | [12–10] rs1 | [9-4] offset | [3–0] opcode

LUI [15–13] rd | [12-10] unused | [9–4] immediate | [3–0] opcode

Primeiro registrador é sempre o destino (rd)

LUI r0, 0 r0 <= 0 000 000 000000 1101 => 0000001000001101 => 0000_0000_0000_1101 hex 000D

ADDI r1, r0, 3 r1 <= r0+3 001 000 000011 0101 => 0010000000110101 => 0010_0000_0011_0101 hex: 2035

ADDI r2, r0, 5 r2<=r0 +5 010 000 000101 0101 => 0100000001010101 => 0100_0000_0101_0101 hex 4055

ADD r3, r1, r2 r3<=r1+r2 011 001 010 000 0000 => 011001010000000 => 0110_0101_0000_0000 hex 6500

LUI r0, 32 r0 <= 32 000 000 100000 1101 => 0000001000001101 => 0000_0010_0000_1101 hex 020D

SW r0, r3, 10 mem(r0+offset)<=r3 ; gravar o valor 8 no endereço de memoria 32+10 então mem(42) <= 8;

000 011 001010 1001 => 0000110010101001 => 0000_1100_1010_1001 hex 0CA9

HLT

Inicialização da memória no VHDL

Mudar a iniciação para um exemplo:

signal Mem : mem_type := (others => (others => ‘0'));

Para:

```
signal Mem : mem_type := (
   0 => x"…..",
   1 => x"…..",
   2 => x"…..",
   3 => x"……",
   4 => x"……",
   others => (others => '0')
);
```

Exemplo 2: Multiplicação por 2 usando soma

Carrega `5` em `r1`

Soma `r1 + r1` e armazena em `r2` (equivalente a `r1 * 2`)

Armazena o resultado (`10`) em `mem[20]`

Finaliza

Endereço	Instrução (Mnemônico)	Descrição	Binário (16 bits)	Hexadecimal
0	`ADDI r1, r0, 5`	r1 ← 5	`0010000001010101`	`2055`
1	`ADD r2, r1, r1`	r2 ← r1 + r1	`0100010010000000`	`4900`
2	`LUI r0, 20`	r0 ← 20 (endereço base para SW)	`0000000101001101`	`014D`
3	`SW r2, r0, 0`	Mem[r0 + 0] ← r2	`0100000000001001`	`4009`

4	HLT	Fim da execução	0000000000001111	000F

Substitua a declaração:

```
vhdl
signal Mem : mem_type := (others => (others => '0'));
```

Por:

```
signal Mem : mem_type := (
    0 => x"2055",  -- ADDI r1, r0, 5
    1 => x"4900",  -- ADD r2, r1, r1
    2 => x"014D",  -- LUI r0, 20
    3 => x"4009",  -- SW r2, r0, 0
    4 => x"000F",  -- HLT
    others => (others => '0')
);
```

Exemplo 3

Este programa realiza o seguinte:

1. Carrega `10` em `r1`
2. Carrega `10` em `r2`
3. Compara `r1` com `r2` usando `BEQ`
4. Se forem iguais, salta (com `JAL`) para uma rotina que carrega um valor da memória

5. Caso contrário, segue direto para `HLT`

Código do Programa

Endereço	Mnemônico	Descrição	Binario / HEX
0	`ADDI r1, r0, 10`	r1 ← 10	
1	`ADDI r2, r0, 10`	r2 ← 10	
2	`BEQ r1, r2, +2`	Se r1 == r2, pula 2 instruções	
3	`HLT`	Fim do programa (não será executado)	
4	`JAL 4`	Salta para endereço `0x08`	
5	`NOP`	Espaço vazio	
6	`NOP`	Espaço vazio	
7	`NOP`	Espaço vazio	
8	`LW r3, r0, 10`	r3 ← Mem[10]	
9	`HLT`	Fim do programa	
10	(dado: 50)	Conteúdo da memória acessado por LW	

Passos do trabalho de Sistemas Digitais

1) Sintetizar e simular a descrição do processador RISC-V e verificar o seu funcionamento para 3 programas diferentes sugeridos por você, explique em poucas palavras o que o programa faz.

Mostre o testbench aqui:

Programa 1:

Forma de onda da simulação do programa 1

Programa 2:

Forma de onda da simulação do programa 1

Programa 3:

Forma de onda da simulação do programa 3

Voce precisou corrigir o código? Se sim, cole aqui o código novo corrigido

2) Mostre os dados de área do processador RISC-V sintetizado no FPGA Artix. Se não for o Artix, indique o FPGA: _____

LUTs =

fps =

3) Complete o numero de clock cycles (c.c.) para executar cada uma das instruções usando array como memória.

Opcode (bin)	Mnemônico	Tipo	**Numero de clock cycles para execução**
0	ADD	R	
1	SUB	R	
10	AND	R	
11	OR	R	
100	XOR	R	
101	ADDI	I	
110	ANDI	I	
111	ORI	I	
1000	LW	I	
1001	SW	S	
1010	BEQ	B	

1011	BNE	B	
1100	JAL	J	
1101	LUI	U	
1110	NOP	N	
1111	HLT	N	

Quantos ciclos de relógio demora para executar todo o programa 1 = _____ ? E o programa 2? _____ E o programa 3? _____.

4) Substitua o array de memória por uma memória embarcada do tipo BRAM. O programa agora deve ser carregado por .coe

Isso implica em ter sinal para address, data_out, data_in e wr da memória.

Note que a BRAM demora um clock cycle para fornecer a saída da memória.

Então a maquina de estados deverá ser ajustada para funcionar com a BRAM.

Cole aqui o novo VHDL

Simule um dos programas a cima, indique o programa e e mostre a forma de onde nova.

5) Complete o numero de clock cycles (c.c.) para executar cada uma das instruções usando BRAM

Opcode (bin)	Mnemônico	Tipo	Numero de clock cycles para execução
0	ADD	R	
1	SUB	R	
10	AND	R	
11	OR	R	
100	XOR	R	
101	ADDI	I	
110	ANDI	I	
111	ORI	I	
1000	LW	I	
1001	SW	S	
1010	BEQ	B	

1011	BNE	B	
1100	JAL	J	
1101	LUI	U	
1110	NOP	N	
1111	HLT	N	

Quantos ciclos de relógio demora para executar todo o programa 1 = _____ ? E o programa 2? _____ E o programa 3? _____.

6) Voce ja pensou que a descrição da organização deste SD16_RISC-V poderia ser por parte operativa (registradores, ULA) e parte de controle (maquina de estados)?

Redescreva o mesmo processador usando essa abordagem, mostre como usou a memória e simule para pelo menos 1 programa.

A) codigo VHDL

B) programa testado

C) simulação