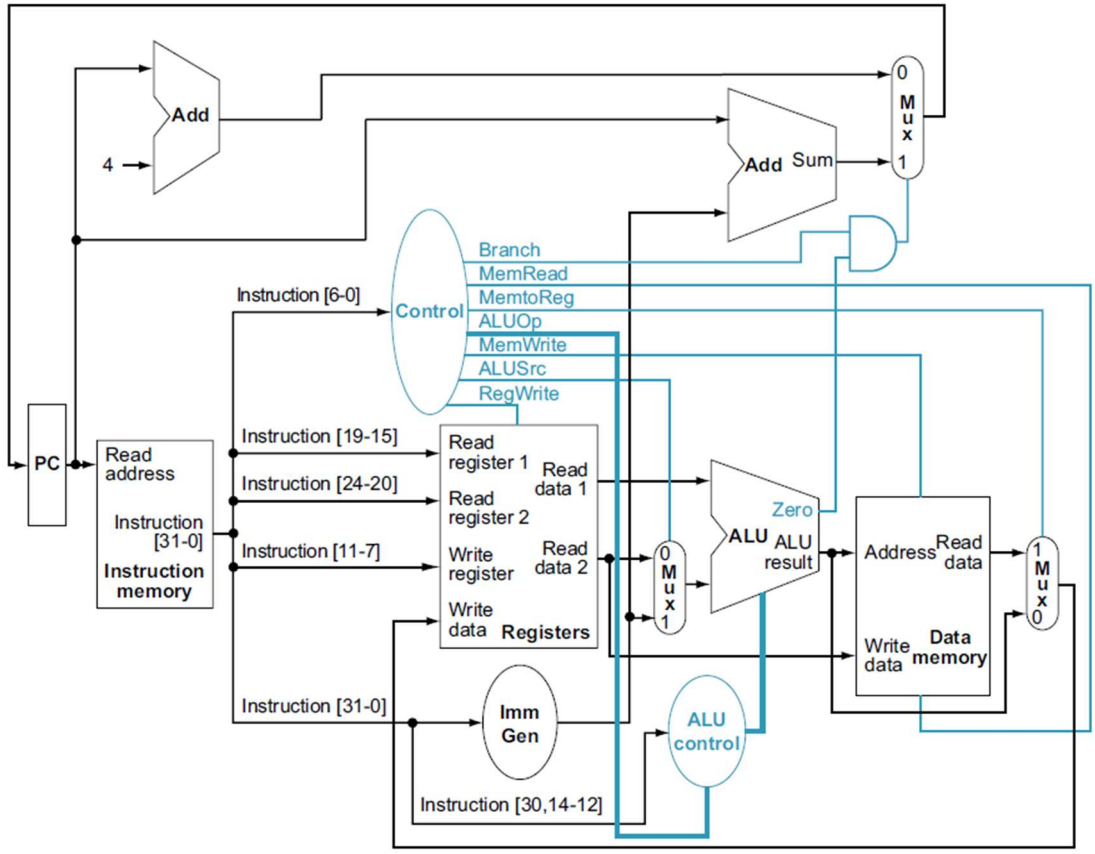
UNIVERSIDADE DE BRASÍLIA
DISCIPLINA: ORGANIZAÇÃO E ARQUITETURA DE COMPUTADORES
TURMA: C
2º SEMESTRE DE 2021

RISC-V UNICICLO

FELIPE FARIAS DA COSTA – 190027592

Descrição

O projeto consiste na construção de um processador RISC-V utilizando a linguagem VHDL. Para a compilação das entidades e realização dos testes, foi utilizado o ModelSim. O diagrama inicial do caminho de dados do RISC-V é apresentado na figura abaixo:



Porém, algumas modificações precisam ser feitas para que possamos implementar todas instruções desejadas.

Instruções

Lógico-Aritméticas: ADD, SUB, AND, OR, XOR, SLT, SLTU

Lógico-Aritméticas com imediato: ADDi, ANDi, ORi, XORi, SLLi, SRLi, SRAi, SLTi, SLTUi, LUI, AUIPC

Subrotinas: JAL, JALR

Saltos: BEQ, BNE, BLT, BGE, BGEU, BLTU

Memória: LW, SW

As instruções destacadas exigiram mudanças no caminho de dados para que pudessem ser implementadas.

Controle

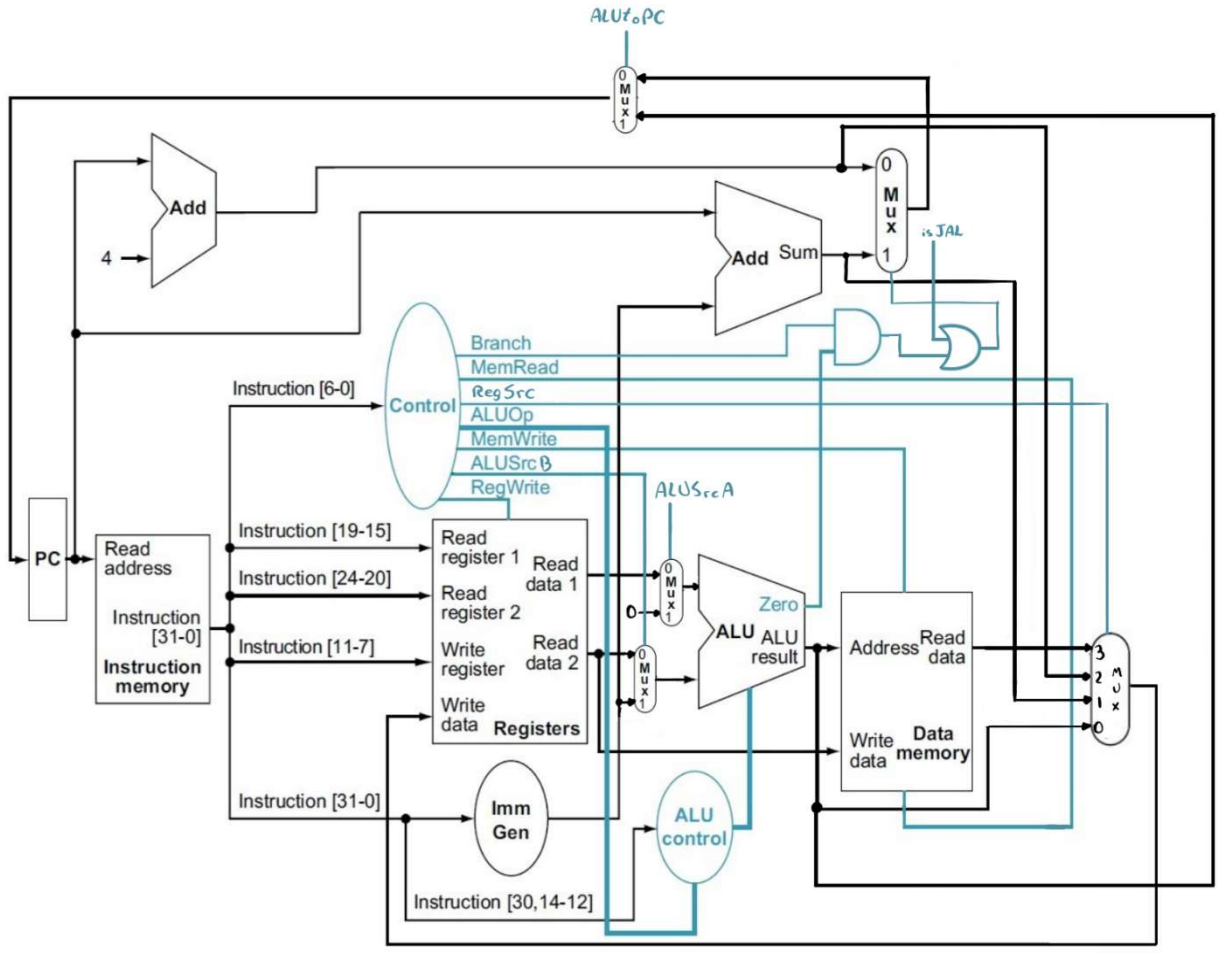
Para a implementação do controle para todas as instruções apresentadas, foram necessárias inclusões de novos sinais e modificações a outros já presentes. *MemtoReg* agora possui 2 bits e foi renomeado para *RegSrc*, *ALUSrc* agora se chama *ALUSrcB,* e foram adicionados os sinais *ALUSrcA, isJAL e ALUtoPC.* Os novos sinais controlam novos multiplexadores para decidir quais dados devem seguir pelo *datapath*.

Sinais de controle ativos para cada tipo de instrução

Saltos: ALUOp=”01”, branch=’1’
LW: ALUOp=”00”, memRead=’1’, regSrc=”11”, ALUSrcB=”1”, regWrite=’1’
SW: ALUOp=”00”, memWrite=’1’, regSrc=”00”, ALUSrcB=”1”
Lógico-Aritméticas: ALUOp=”10”, regWrite=’1’
Lógico-Aritméticas com imediato (sem LUI e AUIPC): ALUOp=”10”, regSrc=”00”, regWrite=’1’, aluSrcB=’1’
LUI: ALUOp=”00”, regSrc=”00”, aluSrcA=’1’, aluSrcB=’1’, regWrite=’1’
AUIPC: regSrc=”01”, regWrite=’1’
JAL: regSrc=”10”, regWrite=’1’, isJAL=’1’
JALR: regSrc=”10”, regWrite=’1’, ALUtoPC=’1’

Implementação

O novo caminho de dados para implementar todas as instruções descritas é apresentado abaixo:



Controle da ULA

A ULA precisa, dos valores de *ALUop* (controle) e dos campos *funct3* e *funct7* (instrução), para que possa enviar a operação correta para a ULA realizar. O código resumido é mostrado abaixo:

```
library IEEE;
use IEEE.std_logic_1164.all;
use work.riscv_pkg.all;

entity alu_control is
port (
    ALUOp      : in std_logic_vector(1 downto 0);
    funct3     : in std_logic_vector(2 downto 0);
    funct7     : in std_logic;
    ALUControl : out std_logic_vector(3 downto 0));
end alu_control;

architecture arch_alu_control of alu_control is
begin
    process(ALUop, funct3, funct7)
    begin
        case ALUOp is
            when "00" =>  -- ADD (lw, sw)
                ALUControl <= ALU_ADD; -- ADD
            when "01" =>  -- Operacao de salto
                if (funct3 = iBEQ3) then -- branch equal
                    ALUControl <= ALU_SEQ; -- SEQ
                elsif (funct3 = iBNE3) then -- branch not equal
                    ALUControl <= ALU_SNE; -- SNE
                 elsif ...
                else
                    report "Valor de funct3 invalido" severity error;
                end if;
            when "10" =>  -- R-Type
                if (funct3 = iADDSUB3) then
                    if (funct7 = iADD7) then ALUControl <= ALU_ADD; -- ADD
                    else                     ALUControl <= ALU_SUB; -- SUB
                    end if;
                elsif (funct3 = iSLL3) then
                    ALUControl <= ALU_SLL;      -- shift left logical
                elsif ...
                else
                    report "Valor de funct3 invalido" severity error;
                end if;
             when "10" =>  ... -- I-Type
            when others => report "Valor de ALUOp invalido" severity error;
        end case;
    end process;
end arch_alu_control;
```

Controle

A implmentação do controle geral do Risc-V é simples, possui cláusulas “when” para cada valor do *opcode*, ativando os sinais de controle como foram descritos anteriormente neste relatório. Como exemplo, veja a cláusula para quando o *opcode* indica que a operação é *sw:*

```
when iSType =>  -- sw
    ALUOp    <= "00";
    branch   <= '0';
    memRead  <= '0';
    regSrc   <= "00";
    memWrite <= '1';
    ALUSrcA  <= '0';
    ALUSrcB  <= '1';
    RegWrite <= '0';
    isJAL    <= '0';
    ALUtoPC  <= '0';
```

PC

A arquitetura do *program counter,* que possui *clk* e *datain* como entradas e *dataout* (endereço da instrução) como saída, foi implementada da seguinte maneira:

```
architecture arch_pc of pc is
    signal instr_address : std_logic_vector(31 downto 0) := (others => '0');
begin
    process(clk) is
    begin
        if rising_edge(clk) then
            instr_address <= datain;
        end if;
    end process;
    dataout <= instr_address;
end arch_pc;
```

Conexão dos componentes

A interconexão dos módulos está presente no arquivo *riscv_singlecycle.vhd*, esse arquivo representa o segundo diagrama apresentado neste relatório. Abaixo estão a conexões de entrada e saída do multiplexador 4x1 que define a entrada de dados para a escrita no banco de registradores:

```
    mux4x1_xregs_dataSrc: mux4x1 port map (
                                sel      => regSrc_ctrl,
                                datain_0 => z_alu,
                                datain_1 => dataout_adder_pc_imm,
                                datain_2 => dataout_adder_pc_4,
                                datain_3 => dataout_ram,
                                dataout  => dataout_mux_4x1);
```

Testes

Para verificar se a arquitetura foi implementada corretamente para todas as instruções, vamos executar um programa de teste e observar os valores de saída obtidos nos módulos.

Instruções ORI, ANDI, LUI, LW, ADD, SW, LW, ADDI

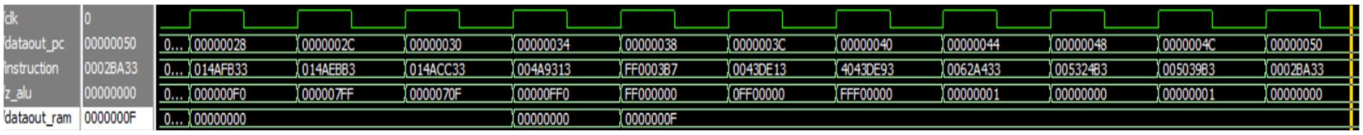
```
.text                                  # PC        MI        ULA       MD

    ori  t0, zero, 0xFF      # 00000000  0ff06293  000000ff  00000000
    andi t0, t0, 0xF0        # 00000004  0f02f293  000000f0  00000000
    lui  s0, 2               # 00000008  00002437  00002000  00000000
    lw   s1, 0(s0)           # 0000000c  00042483  00002000  0000000f
    lw   s2, 4(s0)           # 00000010  00442903  00002004  0000003f
    add  s3, s1, s2          # 00000014  012489b3  0000004e  00000000
    sw   s3, 8(s0)           # 00000018  01342423  00002008  0000004e
```

Observando as ondas no Modelsim, podemos verificar os 4 dados de saída para cada instrução:

Instruções AND, OR, XOR, SLLI, SRLI, SRAI, SLT, SLTU

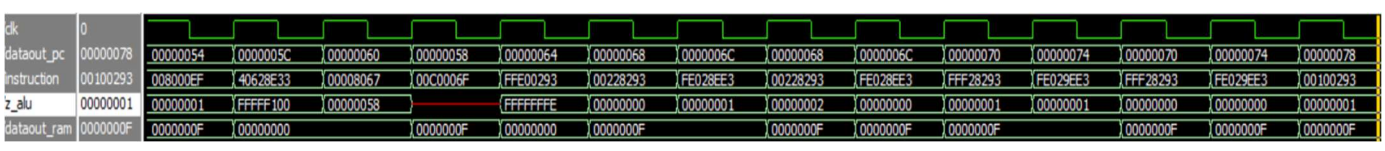
```
and  s6, s5, s4          # 00000028  014afb33  000000f0  00000000
or   s7, s5, s4          # 0000002c  014aebb3  000007ff  00000000
xor  s8, s5, s4          # 00000030  014acc33  0000070f  00000000
slli  t1, s5, 4          # 00000034  004a9313  00000ff0  00000000
lui  t2, 0xFF000         # 00000038  ff0003b7  ff000000  00000000
srli  t3, t2, 4          # 0000003c  0043de13  0ff00000  00000000
srai  t4, t2, 4          # 00000040  4043de93  fff00000  00000000
slt  s0, t0, t1          # 00000044  0062a433  00000001  00000000
slt  s1, t1, t0          # 00000048  005324b3  00000000  00000000
sltu s3, zero, t0        # 0000004c  005039b3  00000001  00000000
sltu s4, t0, zero        # 00000050  0002ba33  00000000  00000000
```



Instruções JAL, SUB, JALR, BEQ, BNE

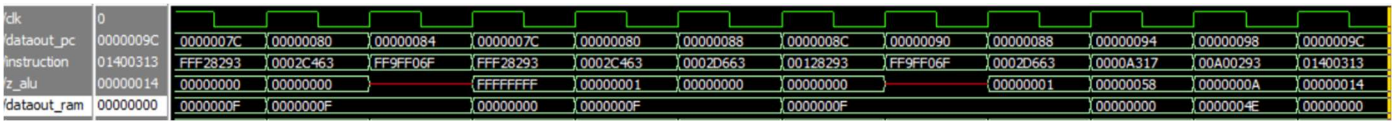
```
    jal  x0, next            # 00000058  00c0006f  00000000  00000000 => 64
testasub:
    sub t3, t0, t1           # 0000005c  40628e33  fffff100  00000000
    jalr x0, ra, 0           # 00000060  00008067  00000058  00000000 => 58
next:
    addi t0, zero, -2        # 00000064  ffe00293  fffffffe  00000000
beqsim:
    addi t0, t0, 2           # 00000068  00228293  0000000*  00000000 * t0 = 0, 2
    beq  t0, zero, beqsim    # 0000006c  fe028ee3  00000000  00000000 => 68, 70
bnesim:
    addi t0, t0, -1          # 00000070  fff28293  0000000*  00000000 * t0 = 1, 0
    bne  t0, zero, bnesim    # 00000074  fe029ee3  00000000  00000000 => 70, 78

    addi t0, zero, 1     # 00000078  00100293   00000001  xxxxxxxx
```



Instruções BLT, BGE, AUIPC

```
bltadd:
    addi t0, t0, -1          # 0000007c  fff28293   00000000  xxxxxxxx
    blt  t0, zero, blton     # 00000080  0002c463   xxxxxxxx  xxxxxxxx => 84, 88
    j    bltadd      # 00000084  ff9ff06f   xxxxxxxx  xxxxxxxx => 7c
blton:
    bge  t0, zero, end   # 00000088  0002d663   xxxxxxxx  xxxxxxxx => 8c, 94
    addi t0, t0, 1       # 0000008c  00128293   00000000  xxxxxxxx
    j blton          # 00000090  ff9ff06f   xxxxxxxx  xxxxxxxx => 88
end:
    auipc t1, 10         # 00000094  0000a317   00000000  xxxxxxxx
    addi  t0, x0, 10     # 00000098  00a00293   00000000  xxxxxxxx
    addi  t1, x0, 20     # 0000009c  01400313   00000000  xxxxxxxx
    add   t2, t0, t1         # 000000a0  006283b3   00000000  xxxxxxxx
```



Observando com cuidado o código (e seus comentários), vemos que o programa foi executado com sucesso. As instruções foram executadas na ordem correta respeitando as condições de salto, verificando, assim, o conteúdo dos registradores e a implementação dos saltos. Nota-se que alguns valores da saída da memória de dados (*dataout_ram*) estão diferentes do apresentado nos comentários do código, isso se deve ao fato da memória ser assíncrona para a leitura, mas é um detalhe de implementação que não afeta o funcionamento do programa no RISC-V Uniciclo.

Para executar o simulador no Modelsim, deve-se criar um projeto que contenha todos os arquivos de código .vhd do RISC-V Uniciclo (e o testbench *tb_riscv_singlecycle.vhd*) e os arquivos de texto utilizados para carregar as memórias de instruções e dados. O testbench *tb_riscv_singlecycle.vhd* é o ponto de partida para a simulação do programa armazenado na *rom*.