Universidade de Brasília

Departamento de Ciência da Computação

Disciplina: Organização e Arquitetura de Computadores - Turma C

Professor: Ricardo Pezzuol Jacobi

Processador Risc-V Uniciclo

Autores:

Alexandre Abrahami Pinto da Cunha 18/0041169

David Fanchic Chatelard 18/0138863

Brasília

25 de maio de 2021

Sumário

1 Introdução

O presente estudo visa desenvolver uma versão do processador Risc-V Uniciclo em VHDL. Esse processador implementado deve ser capaz de executar um código gerado pelo RARS, utilizando o modelo de memória compacto. A plataforma de desenvolvimento utilizada é Altera e a ferramenta utilizada para o desenvolvimento do projeto foi o ModelSim-Altera. Dessa forma, o diagrama esquemático da arquitetura básica do RISC-V Uniciclo é apresentado abaixo:

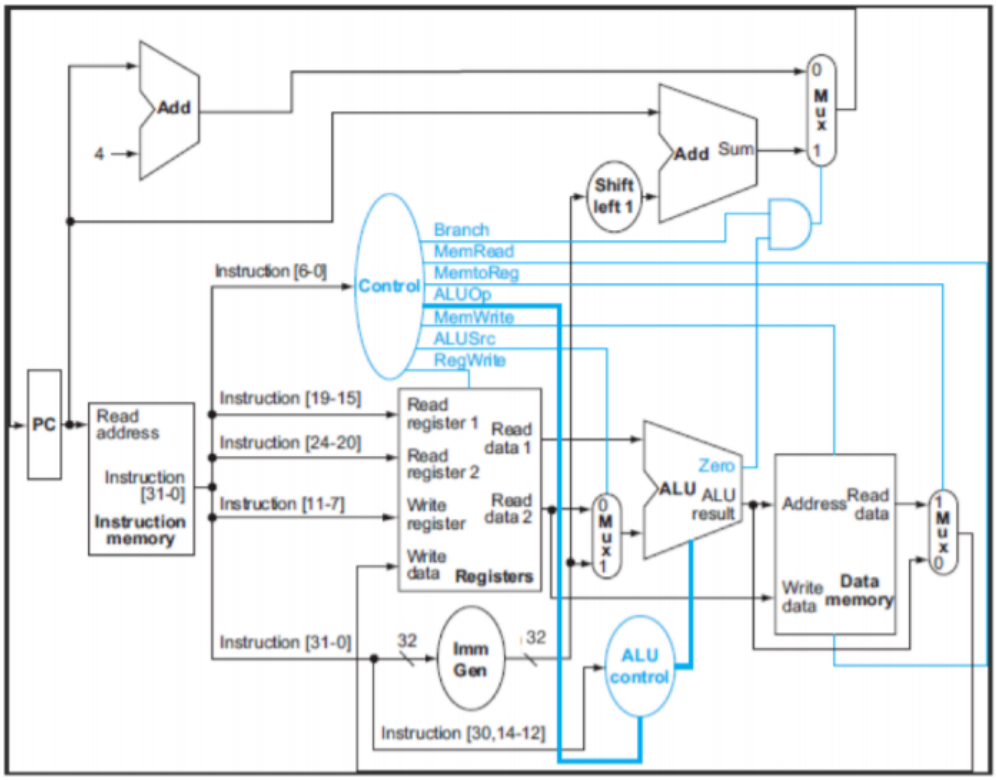

Figura 1 – RISC-V Uniciclo

Com base nesse diagrama, é possível perceber que ele é composto por diversos módulos, sendo que alguns deles já foram desenvolvidos/ implementados ao longo desta disciplina e apresentados em outros relatórios. Apesar disso, no roteiro deste trabalho cada um desses módulos está detalhado de forma clara e iremos abordar abaixo brevemente sobre eles somente para contextualizar o funcionamento do Risc-V Uniciclo:

- **PC:** É um registrador de 32 bits utilizado como o contador de programa do nosso processador, utilizando apenas o número de bits necessário a ser enviado à memória de instruções.
- **Memória de Instruções (MI):** É o local onde o código a ser executado é armazenado utilizando um espaço de endereçamento reduzido.
- **Banco de Registradores (BREG):** é um módulo onde se encontram todos os 32 registradores de 32 bits do processador do Risc-V Uniciclo. Ele possui três entradas de endereços, onde em duas ele permite a leitura de 2 registradores simultaneamente e na terceira seleciona um registrador para escrita de dados na transição de subida do relógio.

- **Unidade Lógico-Aritmética (ULA):** é um módulo que realiza todas as operações lógicas e ariméticas do processador, como somas, subtrações e deslocamento de bits, operando sobre dados de 32 bits. Todas as operações da ULA são apresentadas abaixo:

Operação	Significado	OpCode
ADD A, B	Z recebe a soma das entradas A, B	0000
SUB A, B	Z recebe A - B	0001
AND A, B	Z recebe a operação lógica A and B, bit a bit	0010
OR A, B	Z recebe a operação lógica A or B, bit a bit	0011
XOR A, B	Z recebe a operação lógica A xor B, bit a bit	0100
SLL A, B	Z recebe a entrada A deslocada B bits à esquerda	0101
SRL A, B	Z recebe a entrada A deslocada B bits à direita sem sinal	0110
SRA A, B	Z recebe a entrada A deslocada B bits à direita com sinal	0111
SLT A, B	Z = 1 se A < B, com sinal	1000
SLTU A, B	Z = 1 se A < B, sem sinal	1001
SGE A, B	Z = 1 se A ≥ B, com sinal	1010
SGEU A, B	Z = 1 se A ≥ B, sem sinal	1011
SEQ A, B	Z = 1 se A == B	1100
SNE A, B	Z = 1 se A != B	1101

Figura 2 – Instruções implementadas na ULA

  Além das instruções citadas acima, foram implementadas as seguintes: ADDi, ANDi, ORi, XORi, SLLi, SRLi, SRAi, SLTi, SLTUi, LUI, AUIPC, JAL, JALR, BEQ, BNE, BLT, BGE, LW e SW.

- **Memória de Dados (MD):** É o local onde os dados do programa são armazenados. É possível tanto escrever nela na subida do relógio, quando o sinal de controle EscreveMem estiver acionado, quanto ler as informações contidas nela. Por se tratar de um processador Uniciclo, não se faz a necessidade do sinal de leitura LerMem, normalmente utilizado para colocar na saída de dados o conteúdo da posição de memória endereçada.

- **Multiplexadores 2 para 1:** o processador utiliza 6 multiplexadores com 2 entradas de 32 bits e uma saída de 32 bits e um multiplexador com 2 entradas de 5 bits e uma saída de 5 bits.

- **Somadores:** o processador utiliza 2 somadores de 32 bits para operar com endereços(valores de PC).

- **Gerador de Imediatos:** é um módulo que recebe uma instrução, extraindo o opcode, o func3 e o func7, e gera um imediato extendido de 32 bits na saída a partir da instrução identificada.

O Gerador de Imediatos e a ULA já foram implementados e detalhados em 2 relatórios ao longo desta disciplina. Além disso, a MI e a MD já também foram desenvolvidas previamente, respectivamente, na construção da ROM e da RAM. A implementação de todos os módulos restantes é apresentada nesse relatório, onde ao final eles foram reunidos para a construção final do processador Risc-V Uniciclo.

2 Implementação

Dado os módulos apresentados na introdução deste relatório, falta a implementação do PC, do BREG, dos multiplexadores 2 para 1 e dos somadores para a construção do nosso processador Risc-V Uniciclo. O desenvolvimento deles e os códigos finais de cada um dos módulos são apresentados a seguir.

2.1 PC

O PC implementado possui como entradas o sinal de clock e um sinal de 32 bits e possui como saída outro sinal de 32 bits. Os 32 bits de entrada serão o endereço da próxima instrução a ser executada e ele será passado para a saída durante uma borda de subida do clock. O sinal de saída será enviado como o endereço para a memória de instruções. O resultado desenvolvido em código é apresentado abaixo:

```
library IEEE;
use IEEE.std_logic_1164.all;
use ieee.numeric_std.all;

entity pc is
    port (
           clk : in std_logic;
           input : in std_logic_vector(31 downto 0);
           output : out std_logic_vector(31 downto 0) := "00000000000000000000000000000000"
          );
end pc;

architecture arc_pc of pc is

begin

  process (clk) is
    begin
      if rising_edge(clk) then
        output <= input;
      end if;
  end process;
end arc_pc;
```

2.2 BREG

Foi implementado um banco de registradores que possui 32 registradores de 32 bits. Como entrada ele possui um sinal de 1 bit do clock, um 1 bit de WriteEnable, três de 5 bits de seleção de registradores (para R_{S1}, R_{S2} e R_D) e um de 32 bits que será o dado a ser escrito no registrador R_D. Além disso, esse banco de registradores possui como saídas os valores de 32 bits lidos dos registradores R_{S1} e R_{S2}. Os registradores R_{S1} e R_{S2} serão lidos sempre, porém o sinal WriteData só poderá ser escrito no R_D durante uma borda de subida do clock e com o sinal WriteEnable igual a '1'. O resultado desenvolvido em código é apresentado abaixo:

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity breg is
```

```
   port (
      clk : in std_logic;
      writeEnable : in std_logic;
     select_rs1 : in std_logic_vector(4 downto 0);
    select_rs2 : in std_logic_vector(4 downto 0);
     select_rd : in std_logic_vector(4 downto 0);
      write_data : in std_logic_vector(31 downto 0);
      output_rs1 : out std_logic_vector(31 downto 0);
    output_rs2 : out std_logic_vector(31 downto 0)
      );
end breg;

architecture arc_breg of breg is

type tipoRegistradores is array(0 to 31) of std_logic_vector(31 downto 0);
signal registradores : tipoRegistradores := (others => "00000000000000000000000000000000");


begin

  output_rs1 <= registradores(to_integer(unsigned(select_rs1)));
  output_rs2 <= registradores(to_integer(unsigned(select_rs2)));

   processo_regs:process(clk)
   begin
      if rising_edge(clk) then
         if writeEnable = '1' and select_rd /= "00000" then
               registradores(to_integer(unsigned(select_rd))) <= write_data;
         end if;
      end if;
   end process;
end arc_breg;
```

2.3 Multiplexadores 2 para 1

Os multiplexadores implementados possuem 2 sinais de dados de entrada, 1 sinal de seleção e 1 sinal de saída, porém um multiplexador possui entradas e saída de 32 bits e o outro possui entradas e saída de 5 bits. O resultado desenvolvido em código para o multiplexador com os valores de 32 bits é apresentado abaixo:

```
library IEEE;
use IEEE.std_logic_1164.all;
use ieee.numeric_std.all;

entity mux_2to1 is
    port (
          sel : in std_logic;
```

```
          A   : in  std_logic_vector(31 downto 0);
          B   : in  std_logic_vector(31 downto 0);
          X   : out std_logic_vector(31 downto 0)
         );
end mux_2to1;

architecture arc of mux_2to1 is
  begin
    X <= A when (sel = '0') else B;
end arc;
```

Já o resultado desenvolvido em código para o multiplexador com os valores de 5 bits é apresentado abaixo:

```
library IEEE;
use IEEE.std_logic_1164.all;
use ieee.numeric_std.all;

entity mux_2to1_5bits is
    port (
          sel : in std_logic;
          A   : in  std_logic_vector(4 downto 0);
          B   : in  std_logic_vector(4 downto 0);
          X   : out std_logic_vector(4 downto 0)
         );
end mux_2to1_5bits;

architecture arc of mux_2to1_5bits is
  begin
    X <= A when (sel = '0') else B;
end arc;
```

2.4 Somadores

O somador implementado possui duas entradas e uma saída de 32 bits. A saída recebe a soma dos valores da entrada. O resultado desenvolvido em código é apresentado abaixo:

```
library IEEE;
use IEEE.std_logic_1164.all;
use ieee.numeric_std.all;

entity adder_32_bit is
    port ( num1 : in std_logic_vector(31 downto 0);
           num2 : in std_logic_vector(31 downto 0);
           soma : out std_logic_vector(31 downto 0)
         );
end adder_32_bit;
```

```
architecture arc_add of adder_32_bit is
begin

    soma <= std_logic_vector(signed(num1) + signed(num2));

end arc_add;
```

3 Testes realizados

Os testes foram realizados com o arquivo de teste gerado pelo RARS disponibilizado pelo professor no Teams. A seguir, será testado todas as operações do processador Risc-V Uniciclo, onde mostraremos primeiramente o resultado esperado e em seguida os nossos resultados obtidos.

3.1 Testes com ORI, ANDI, LUI, LW e ADD

A primeira parte de nossos testes foi realizado com as operações ORI, ANDI, LUI, LW e ADD. Os resultados esperados eram:

```
.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
```

Figura 3 – Instruções executadas para a primeira parte dos testes realizados

Observando no ModelSim-Altera as formas de onda e os valores hexadecimais de cada variável, os nosso resultados obtidos são:

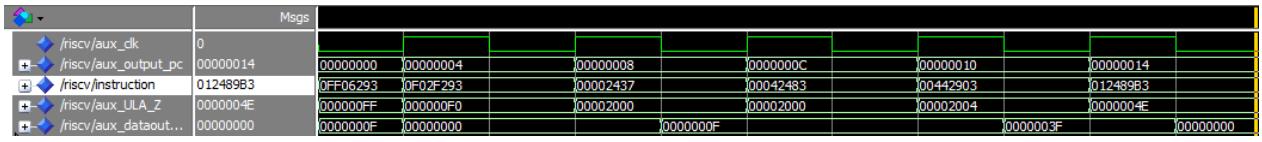

Figura 4 – Primeira parte dos resultados obtidos

3.2 Testes com SW, LW, ADDI, AND e OR

A segunda parte de nossos testes foi realizado com as operações SW, LW, ADDI, AND e OR. Os resultados esperados eram:

```
sw   s3, 8(s0)          # 00000018  01342423   00002008  0000004e
lw   a0, 8(s0)          # 0000001c  00842503   00002008  0000004e
addi s4, zero, 0x7F0    # 00000020  7f000a13   000007f0  00000000
addi s5, zero, 0x0FF    # 00000024  0ff00a93   000000ff  00000000
and  s6, s5, s4         # 00000028  014afb33   000000f0  00000000
or   s7, s5, s4         # 0000002c  014aebb3   000007ff  00000000
```

Figura 5 – Instruções executadas para a segunda parte dos testes realizados

Observando no ModelSim-Altera as formas de onda e os valores hexadecimais de cada variável, os nosso resultados obtidos são:

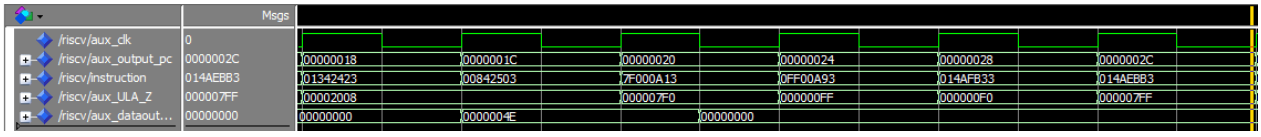

Figura 6 – Segunda parte dos resultados obtidos

3.3 Testes com XOR, SLLI, LUI, SRLI, SRAI e SLT

A terceira parte de nossos testes foi realizado com as operações XOR, SLLI, LUI, SRLI, SRAI e SLT. Os resultados esperados eram:

```
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
```

Figura 7 – Instruções executadas para a terceira parte dos testes realizados

Observando no ModelSim-Altera as formas de onda e os valores hexadecimais de cada variável, os nosso resultados obtidos são:

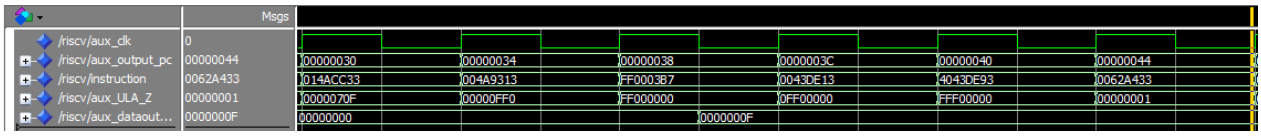

Figura 8 – Terceira parte dos resultados obtidos

3.4 Testes com SLT, SLTU, JAL, SUB e JALR

A quarta parte de nossos testes foi realizado com as operações SLT, SLTU, JAL, SUB e JALR. Os resultados esperados eram:

```
        slt  s1, t1, t0          # 00000048  005324b3   00000000  00000000
        sltu s3, zero, t0       # 0000004c  005039b3   00000001  00000000
        sltu s4, t0, zero       # 00000050  0002ba33   00000000  00000000

        jal  ra, testasub       # 00000054  008000ef   00000000  00000000 => 5c

        jal  x0, next           # 00000058  00c0006f   00000000  00000000 => 64
testasub:
        sub t3, t0, t1          # 0000005c  40628e33   fffffffe  00000000
        jalr x0, ra, 0          # 00000060  00008067   00000058  00000000 => 58
```

Figura 9 – Instruções executadas para a quarta parte dos testes realizados

Observando no ModelSim-Altera as formas de onda e os valores hexadecimais de cada variável, os nosso resultados obtidos são:

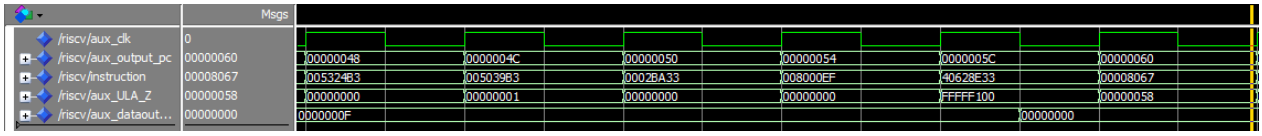

Figura 10 – Quarta parte dos resultados obtidos

3.5 Testes com JAL, SUB, JALR, ADDI e BEQ

A quinta parte de nossos testes foi realizado com as operações JAL, SUB, JALR, ADDI e BEQ. Os resultados esperados eram:

```
        jal  x0, next          # 00000058  00c0006f   00000000  00000000 => 64
testasub:
        sub t3, t0, t1          # 0000005c  40628e33   fffffffe  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
```

Figura 11 – Instruções executadas para a quinta parte dos testes realizados

Observando no ModelSim-Altera as formas de onda e os valores hexadecimais de cada variável, os nosso resultados obtidos são:

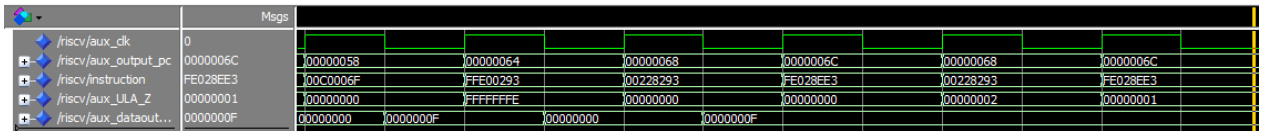

Figura 12 – Quinta parte dos resultados obtidos

3.6 Testes com ADDI e BNE

A sexta parte e última de nossos testes foi realizado com as operações ADDI e BNE. Os resultados esperados eram:

```
bnesim:
        addi t0, t0, -1          # 00000070  fff28293   0000000*  00000000 * t0 = 1, 0
        bne  t0, zero, bnesim    # 00000074  fe029ee3   00000000  00000000 => 70, 78
```

Figura 13 – Instruções executadas para a sexta parte dos testes realizados

Observando no ModelSim-Altera as formas de onda e os valores hexadecimais de cada variável, os nosso resultados obtidos são:

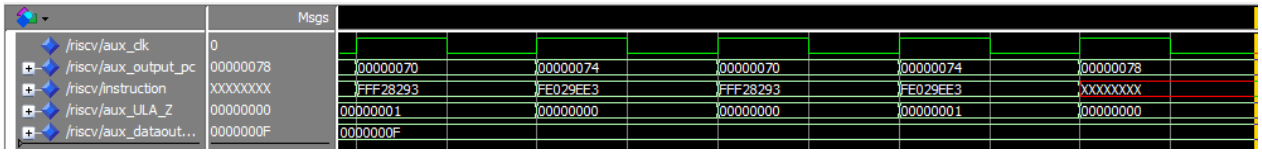

Figura 14 – Sexta parte dos resultados obtidos

É possível perceber que os resultados obtidos foram de acordo com os esperados. Apesar disso, é importante destacar que em alguns casos o sinal de saída da memória de dados (aux_dataout_ram) não estava igual ao resultado apresentado pelo RARS. Isso ocorreu visto que o multiplexador seleciona entre o sinal da saída da MD e da ULA, de tal forma que caso a instrução não acesse a memória ele irá selecionar a saída da ULA. Como todas as instruções que não são LW e SW não acessam a memória, ocorreu essa saída nelas, não afetando o resultado final do programa e não sendo um problema para o nosso processador.

4 Conclusão

Foi possível desenvolver uma versão do processador Risc-V Uniciclo em VHDL, executando ao final um código gerado pelo RARS disponibilizado pelo professor utilizando o modelo de memória compacto. Percebe-se que em todos os testes os resultados obtidos foram de acordo com o esperado, destacando-se o comentário realizado a respeito do sinal de saída da memória de dados, comprovando a funcionalidade dos códigos desenvolvidos para cada módulo restante do processador e o funcionamento de cada um deles trabalhando em conjunto.