UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
DISCIPLINA: SISTEMAS DIGITAIS
Prof. Fernanda Lima Kastensmidt

2022-2

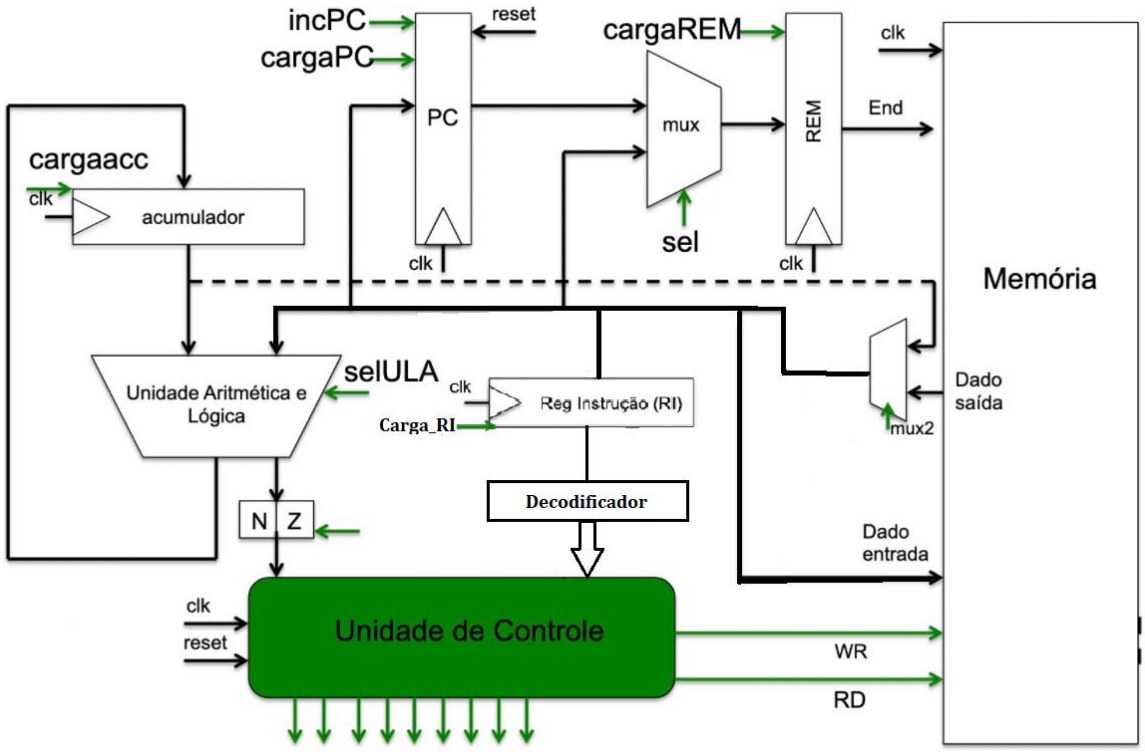
Trabalho 1 – Sistemas Digitais – Individual
Aluno: Pedro Henrique Casarotto Rigon (00325358)

Projeto do Processador Ahmes em VHDL

Descrição do Ahmes:

Este relatório tem como objetivo fornecer uma descrição detalhada da implementação do Processador Ahmes que é descrito na linguagem de descrição de hardware VHDL (VHSIC Hardware Description Language) e consiste em três componentes principais:

- **A memória**: é descrita como um componente genérico que é utilizado para armazenar dados. Ele possui portas de entrada e saída para a leitura e escrita de dados, respectivamente.
- **O datapath:** descreve a estrutura de dados da CPU e inclui todas as operações que são executadas pelo CPU. Ele inclui as portas para controlar as operações que são executadas e a troca de dados.
- **A unidade de controle**: descreve a lógica de controle da CPU e inclui todas as decisões sobre as operações que devem ser executadas.



Descrição do Datapath:

O datapath é o caminho de dados do processador e é composto por vários componentes, incluindo registradores, unidades de controle e multiplexadores, que trabalham juntos para realizar as operações de processamento. O datapath do processador Ahmes é composto por:

- Registrador acumulador: Armazena o resultado de operações da ULA.
- Registrador de programa contador: Armazena o endereço da próxima instrução a ser executada.
- Registrador de instrução: Armazena a instrução atual.
- Registrador de Flags: Armazena as flags geradas pela ULA.
- Registrador de endereço remoto: Armazena o endereço de memória remoto.
- Unidade lógico-aritmética (ULA): Realiza as operações aritméticas e lógicas.
- Multiplexador 1: Seleciona a entrada para a memória.
- Multiplexador 2: Seleciona a entrada para a ULA, Reg_RI e Reg_REM.

Processamento:

O processamento no datapath do processador Ahmes é controlado por sinais de entrada e por seus registradores internos.

- Carga_acumulador: Se este sinal estiver ativo, a saída da ULA será armazenada no registrador acumulador.
- Inc_PC: Se este sinal estiver ativo, o contador de programa será incrementado em um.
- Carga_PC: Se este sinal estiver ativo, o contador de programa será carregado com a entrada de endereço.
- Reset: Se este sinal estiver ativo, todos os registradores serão reiniciados.
- Carga_rem: Se este sinal estiver ativo, o registrador de endereço remoto será carregado com a entrada de endereço.
- Sel_mux: Seleciona a entrada para o multiplexador 1.
- Sel_mux2: Seleciona a entrada para o multiplexador 2.
- Carga_RI: Se este sinal estiver ativo, o registrador de instrução será carregado com a entrada de dado.
- Sel_ULA: Seleciona a operação a ser realizada pela ULA.

Qual componente FPGA escolheste para a síntese? Spartan3E

Quantos registradores tem o datapath do RAMSES? 5

Quantas operações diferentes tem a ULA? 10

A área do DATAPTH em # LUTs: 116 e #ffps: 37

Descrição da Unidade de Controle:

A entidade "unidade_de_controle" é definida com vários elementos de entrada e saída. Os elementos de entrada incluem o clock, o reset, várias instruções, flags de condições, entre outros. Já os portos de saída incluem informações sobre a carga de flags, o selecionador da ULA, o selecionador do MUX2, a carga de ACC, o selecionador do MUX1, a incrementação do PC, a carga do PC, a carga do REM, a escrita na memória e a instrução HLT.

A UC é implementada como uma FSM com 8 estados. Cada estado representa uma etapa do ciclo de clock da UC e é determinado pela combinação de entrada de instruções e flags de condições. A constante é utilizada para definir o valor de saída para o selecionador da ULA de acordo com a instrução de entrada. O processo principal da UC consiste em ler a entrada atual e decidir qual será o próximo estado e as saídas. No estado inicial, S0, a UC aguarda a recepção do reset para inicializar. Em seguida, a UC entra em um estado determinado pelo tipo de instrução recebida e pela condição das flags.

tempo	STA	LDA	ADD	OR	AND	NOT
t0	sel=0, carga REM	sel=0, carga REM	sel=0, carga REM	sel=0, carga REM	sel=0, carga REM	sel=0, carga REM
t1	Read, incrementa PC	Read, incrementa PC	Read, incrementa PC	Read, incrementa PC	Read, incrementa PC	Read, incrementa PC
t2	carga RI	carga RI	carga RI	carga RI	carga RI	carga RI
t3	sel=0, carga REM	sel=0, carga REM	sel=0, carga REM	sel=0, carga REM	sel=0, carga REM	UAL(NOT), carga AC, carga NZ, goto t0
t4	Read, incrementa PC	Read, incrementa PC	Read, incrementa PC	Read, incrementa PC	Read, incrementa PC	
t5	sel=1, carga REM	sel=1, carga REM	sel=1, carga REM	sel=1, carga REM	sel=1, carga REM	
t6	carga RDM	Read	Read	Read	Read	
t7	Write, goto t0	UAL(Y), carga AC, carga NZ, goto t0	UAL(ADD), carga AC, carga NZ, goto t0	UAL(OR), carga AC, carga NZ, goto t0	UAL(AND, carga AC, carga NZ, goto t0	

tempo	JMP	JN, N=1	JN, N=0	JZ, Z=1	JZ, Z=0	NOP	HLT
t0	sel=0, carga REM	sel=0, carga REM	sel=0, carga REM	sel=0, carga REM	sel=0, carga REM	sel=0, carga REM	sel=0, carga REM
t1	Read, incrementa PC	Read, incrementa PC	Read, incrementa PC	Read, incrementa PC	Read, incrementa PC	Read, incrementa PC	Read, incrementa PC
t2	carga RI	carga RI	carga RI	carga RI	carga RI	carga RI	carga RI
t3	sel=0, carga REM	sel=0, carga REM	incrementa PC, goto t0	sel=0, carga REM	incrementa PC, goto t0	goto t0	Halt
t4	Read	Read		Read			
t5	carga PC, goto t0	carga PC, goto t0		carga PC, goto t0			
t6							
t7							

Código	Instrução	Significado
0000 xxxx	NOP	nenhuma operação
0001 xxxx	STA end	MEM(end) ← AC
0010 xxxx	LDA end	AC ← MEM(end)
0011 xxxx	ADD end	AC ← AC + MEM(end)
0100 xxxx	OR end	AC ← AC **OR** MEM(end) *("ou" bit-a-bit)*
0101 xxxx	AND end	AC ← AC **AND** MEM(end) *("e" bit-a-bit)*
0110 xxxx	NOT	AC ← **NOT** AC (complemento de 1)
0111 xxxx	SUB end	AC ← AC - MEM(end)
1000 xxxx	JMP end	PC ← end (desvio incondicional)
1001 00xx	JN end	IF N=1 THEN PC ← end
1001 01xx	JP end	IF N=0 THEN PC ← end
1001 10xx	JV end	IF V=1 THEN PC ← end
1001 11xx	JNV end	IF V=0 THEN PC ← end

Código	Instrução	Significado
1010 00xx	JZ end	IF Z=1 THEN PC ← end
1010 01xx	JNZ end	IF Z=0 THEN PC ← end
1011 00xx	JC end	IF C=1 THEN PC ← end
1011 01xx	JNC end	IF Z=0 THEN PC ← end
1011 10xx	JB end	IF B=1 THEN PC ← end
1011 11xx	JNB end	IF Z=0 THEN PC ← end
1110 xx00	SHR	C ← AC(0); AC(i-1) ← AC(i); AC(7) ← 0
1110 xx01	SHL	C ← AC(7); AC(i) ← AC(i-1); AC(0) ← 0
1110 xx10	ROR	C ← AC(0); AC(i-1) ← AC(i); AC(7) ← C
1110 xx11	ROL	C ← AC(7); AC(i) ← AC(i-1); AC(0) ← C
1111 xxxx	HLT	término da execução (halt)

Completar a tabela a seguir com as instruções AHMES que não tem no NEANDER

Tempo	SHR	SHL	ROR	ROL	SUB
T0	sel_MUX1=0, Carga_REM	sel_MUX1=0, carga_REM	sel_MUX1=0, carga_REM	sel_MUX1=0, carga_REM	sel_MUX1=0, carga_REM
T1	Read, Inc_PC	Read, inc_PC	Read, inc_PC	Read, inc_PC	Read, inc_PC
T2	carga_RI	carga_RI	carga_RI	carga_RI	carga_RI
T3	ULA(SHR), carga_ACC, carga_FLAG, Goto t0	ULA(SHR), carga_ACC, carga_FLAG, Goto t0	ULA(SHR), carga_ACC, carga_FLAG, Goto t0	ULA(SHR), carga_ACC, carga_FLAG, Goto t0	sel_MUX1=0, carga_REM
T4					Read, inc_PC
T5					sel_MUX1=1, carga_REM
T6					Read
T7					ULA(SUB), carga_ACC, carga_FLAG, Goto t0

E as instruções novas de Desvio

Tempo	JP (N=0)	JP (N=1)	JV (V=1)	JV (V=0)	JNV (V=0)	JNV (V=1)
T0	sel_MUX1=0, Carga_REM	sel_MUX1=0, Carga_REM	sel_MUX1=0, Carga_REM	sel_MUX1=0, Carga_REM	sel_MUX1=0, Carga_REM	sel_MUX1=0, Carga_REM
T1	Read, Inc_PC	Read, Inc_PC	Read, Inc_PC	Read, Inc_PC	Read, Inc_PC	Read, Inc_PC
T2	carga_RI	carga_RI	carga_RI	carga_RI	carga_RI	carga_RI
T3	sel_MUX1=0, carga_REM	inc_PC, goto t0	sel_MUX1=0, carga_REM	inc_PC, goto t0	sel_MUX1=0, carga_REM	inc_PC, goto t0
T4	Read		Read		Read	
T5	carga_PC, Goto t0		carga_PC, Goto t0		carga_PC, Goto t0	
T6						
T7						

Tempo	JC (C=1)	JC (C=0)	JNC (C=0)	JNC (C=1)	JB (B=1)	JB (B=0)
T0	sel_MUX1=0, Carga_REM	sel_MUX1=0, Carga_REM	sel_MUX1=0, Carga_REM	sel_MUX1=0, Carga_REM	sel_MUX1=0, Carga_REM	sel_MUX1=0, Carga_REM
T1	Read, Inc_PC	Read, Inc_PC	Read, Inc_PC	Read, Inc_PC	Read, Inc_PC	Read, Inc_PC
T2	carga_RI	carga_RI	carga_RI	carga_RI	carga_RI	carga_RI
T3	sel_MUX1=0, carga_REM	inc_PC, goto t0	sel_MUX1=0, carga_REM	inc_PC, goto t0	sel_MUX1=0, carga_REM	inc_PC, goto t0
T4	Read		Read		Read	
T5	carga_PC, Goto t0		carga_PC, Goto t0		carga_PC, Goto t0	
T6						
T7						

Tempo	JNB (B=0)	JNB (B=1)
T0	sel_MUX1=0, Carga_REM	sel_MUX1=0, Carga_REM
T1	Read, Inc_PC	Read, Inc_PC
T2	carga_RI	carga_RI
T3	sel_MUX1=0, carga_REM	inc_PC, goto t0
T4	Read	
T5	carga_PC, Goto t0	
T6		
T7		

Descrição do Decodificador:

O decodificador de instruções é responsável por identificar a instrução a ser executada a partir da saída do Registrador de Instrução (RI). Ele decodifica a saída do RI, sinalizando com o valor 1 a flag da instrução correta. As entradas são a saída do RI, enquanto as saídas são as flags de instruções, como NOP, STA, LDA, ADD,

OR, AND, NOT, SUB, JMP, JN, JP, JV, JNV, JZ, JNZ, JC, JNC, JB, JNB, SHR, SHL, ROR, ROL e HLT.

A lógica do decodificador é implementada com um processo que observa a saída do RI. Todas as flags de instruções são inicializadas com o valor 0 e, em seguida, o processo verifica a saída do RI para identificar a instrução correta. Se a saída do RI corresponder a uma das opções previamente definidas, a flag da instrução correspondente é sinalizada com o valor 1.

Tabela de Instruções:

Código em binário	Instrução
00000000	NOP
00010000	STA
00100000	LDA
00110000	ADD
01000000	OR
01010000	AND
01100000	NOT
01110000	SUB
10000000	JMP
10010000	JN
10010100	JP
10011000	JV
10011100	JNV
10100000	JZ
10100100	JNZ
10101000	JC
10101100	JNC
10110000	JB
10110100	JNB
11000000	SHR
11010000	SHL
11100000	ROR
11110000	ROL
11111111	HLT

VHDL completo do Ahmes:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
entity ahmes is
    Port ( clk : in  STD_LOGIC;
           reset : in  STD_LOGIC;
           flags : out STD_LOGIC_VECTOR (4 downto 0);
           dout : out  STD_LOGIC_VECTOR (7 downto 0);
           hlt : out STD_LOGIC);
end ahmes;

architecture Behavioral of ahmes is

COMPONENT memoria
  PORT (
    clka : IN STD_LOGIC;
    wea : IN STD_LOGIC_VECTOR(0 DOWNTO 0);
    addra : IN STD_LOGIC_VECTOR(7 DOWNTO 0);
    dina : IN STD_LOGIC_VECTOR(7 DOWNTO 0);
    douta : OUT STD_LOGIC_VECTOR(7 DOWNTO 0)
  );
END COMPONENT;


component datapath_ahmes
    Port ( clk : in  STD_LOGIC;
           carga_acc : in  STD_LOGIC;
           inc_pc : in  STD_LOGIC;
           carga_pc : in  STD_LOGIC;
           reset : in  STD_LOGIC;
           carga_rem : in  STD_LOGIC;
           sel_mux : in  STD_LOGIC;
           sel_mux2 : in  STD_LOGIC;
           carga_ri : in  STD_LOGIC;
           sel_ula : in  STD_LOGIC_VECTOR (3 downto 0);
           carga_flag : in  STD_LOGIC;
           endereco : out  STD_LOGIC_VECTOR (7 downto 0);
           dado_entrada : out  STD_LOGIC_VECTOR (7 downto 0);
           dado_saida : in  STD_LOGIC_VECTOR (7 downto 0);
           reg_ri : out STD_LOGIC_VECTOR(7 downto 0);
           flag_ula_control : out STD_LOGIC_VECTOR(4 downto 0));
end component;
```

```
component unidade_de_controle
    Port ( clk : in STD_LOGIC;
           reset : in STD_LOGIC;
           inst_NOP : in  STD_LOGIC;
           inst_STA : in  STD_LOGIC;
           inst_LDA : in  STD_LOGIC;
           inst_ADD : in  STD_LOGIC;
           inst_OR : in  STD_LOGIC;
           inst_AND : in  STD_LOGIC;
           inst_NOT : in  STD_LOGIC;
           inst_SUB : in  STD_LOGIC;
           inst_JMP : in  STD_LOGIC;
           inst_JN : in  STD_LOGIC;
           inst_JP : in  STD_LOGIC;
           inst_JV : in  STD_LOGIC;
           inst_JNV : in  STD_LOGIC;
           inst_JZ : in  STD_LOGIC;
           inst_JNZ : in  STD_LOGIC;
           inst_JC : in  STD_LOGIC;
           inst_JNC : in  STD_LOGIC;
           inst_JB : in  STD_LOGIC;
           inst_JNB : in  STD_LOGIC;
           inst_SHR : in  STD_LOGIC;
           inst_SHL : in  STD_LOGIC;
           inst_ROR : in  STD_LOGIC;
           inst_ROL : in  STD_LOGIC;
           inst_HLT : in  STD_LOGIC;
           flag_N : in STD_LOGIC;
           flag_Z : in STD_LOGIC;
           flag_V : in STD_LOGIC;
           flag_B : in STD_LOGIC;
           flag_C : in STD_LOGIC;
           carga_FLAG : out  STD_LOGIC;
           carga_RI : out  STD_LOGIC;
           sel_ULA : out  STD_LOGIC_VECTOR (3 downto 0);
           sel_MUX2 : out  STD_LOGIC;
           carga_ACC : out  STD_LOGIC;
           sel_MUX1 : out  STD_LOGIC;
           inc_PC : out  STD_LOGIC;
           carga_PC : out  STD_LOGIC;
           carga_REM : out  STD_LOGIC;
           hlt : out STD_LOGIC;
           write_mem : out  STD_LOGIC);
end component;
```

```
component decodificador_inst
   port (saida_RI : in STD_LOGIC_VECTOR (7 downto 0);
         inst_NOP : out  STD_LOGIC;
         inst_STA : out  STD_LOGIC;
         inst_LDA : out  STD_LOGIC;
         inst_ADD : out  STD_LOGIC;
         inst_OR : out  STD_LOGIC;
         inst_AND : out  STD_LOGIC;
         inst_NOT : out  STD_LOGIC;
         inst_SUB : out  STD_LOGIC;
         inst_JMP : out  STD_LOGIC;
         inst_JN : out  STD_LOGIC;
         inst_JP : out  STD_LOGIC;
         inst_JV : out  STD_LOGIC;
         inst_JNV : out  STD_LOGIC;
         inst_JZ : out  STD_LOGIC;
         inst_JNZ : out  STD_LOGIC;
         inst_JC : out  STD_LOGIC;
         inst_JNC : out  STD_LOGIC;
         inst_JB : out  STD_LOGIC;
         inst_JNB : out  STD_LOGIC;
         inst_SHR : out  STD_LOGIC;
         inst_SHL : out  STD_LOGIC;
         inst_ROR : out  STD_LOGIC;
         inst_ROL : out  STD_LOGIC;
         inst_HLT : out  STD_LOGIC);
end component;


--signals
signal carga_PC : std_logic;
signal carga_REM : std_logic;
signal carga_ACC : std_logic;
signal carga_FLAG : std_logic;
signal carga_RI : std_logic;

signal inst_NOP : STD_LOGIC;
signal inst_STA : STD_LOGIC;
signal inst_LDA : STD_LOGIC;
signal inst_ADD : STD_LOGIC;
signal inst_OR : STD_LOGIC;
signal inst_AND : STD_LOGIC;
signal inst_NOT : STD_LOGIC;
signal inst_SUB : STD_LOGIC;
signal inst_JMP : STD_LOGIC;
```

```
signal inst_JN : STD_LOGIC;
signal inst_JP : STD_LOGIC;
signal inst_JV : STD_LOGIC;
signal inst_JNV : STD_LOGIC;
signal inst_JZ : STD_LOGIC;
signal inst_JNZ : STD_LOGIC;
signal inst_JC : STD_LOGIC;
signal inst_JNC : STD_LOGIC;
signal inst_JB : STD_LOGIC;
signal inst_JNB : STD_LOGIC;
signal inst_SHR : STD_LOGIC;
signal inst_SHL : STD_LOGIC;
signal inst_ROR : STD_LOGIC;
signal inst_ROL : STD_LOGIC;
signal inst_HLT : STD_LOGIC;

signal inc_PC : std_logic;
signal sel_mux : std_logic;
signal sel_mux2 : std_logic;
signal sel_ULA : std_logic_vector (3 downto 0);


signal input_MUX2 : STD_LOGIC_VECTOR (7 downto 0);  --vai endereco ou é da
memória
signal dado_entrada : STD_LOGIC_VECTOR (7 downto 0); --vai endereco ou é da
memória
signal output_RI : STD_LOGIC_VECTOR(7 downto 0);
signal output_REM : STD_LOGIC_VECTOR (7 downto 0); --vai endereco ou é da
memória
signal flag_ula_control : STD_LOGIC_VECTOR(4 downto 0);

signal iWrite : std_logic_vector (0 downto 0);

begin


datapapath: datapath_ahmes
    port map ( clk => clk,
           carga_acc => carga_ACC,
           inc_pc => inc_PC,
           carga_pc => carga_PC,
           reset => reset,
           carga_rem => carga_REM,
           sel_mux => sel_mux,
           sel_mux2 => sel_mux2,
```

```
            carga_ri => carga_RI,
            sel_ula => sel_ULA,
            carga_flag => carga_FLAG,
            endereco => output_REM,
            dado_entrada => dado_entrada,
            dado_saida => input_MUX2,
            reg_ri => output_RI,
            flag_ula_control => flag_ula_control);

decod_RI: decodificador_inst
   port map (saida_RI => output_RI,
         inst_NOP => inst_NOP,
         inst_STA => inst_STA,
         inst_LDA => inst_LDA,
         inst_ADD => inst_ADD,
         inst_OR => inst_OR,
         inst_AND => inst_AND,
         inst_NOT => inst_NOT,
         inst_SUB => inst_SUB,
         inst_JMP => inst_JMP,
         inst_JN => inst_JN,
         inst_JP => inst_JP,
         inst_JV => inst_JV,
         inst_JNV => inst_JNV,
         inst_JZ => inst_JZ,
         inst_JNZ => inst_JNZ,
         inst_JC => inst_JC,
         inst_JNC => inst_JNC,
         inst_JB => inst_JB,
         inst_JNB => inst_JNB,
         inst_SHR => inst_SHR,
         inst_SHL => inst_SHL,
         inst_ROR => inst_ROR,
         inst_ROL => inst_ROL,
         inst_HLT => inst_HLT);

control_unity: unidade_de_controle
    port map ( clk => clk,
            reset => reset,
            inst_NOP => inst_NOP,
            inst_STA => inst_STA,
            inst_LDA => inst_LDA,
            inst_ADD => inst_ADD,
            inst_OR => inst_OR,
            inst_AND => inst_AND,
```

```
            inst_NOT => inst_NOT,
            inst_SUB => inst_SUB,
            inst_JMP => inst_JMP,
            inst_JN => inst_JN,
            inst_JP => inst_JP,
            inst_JV => inst_JV,
            inst_JNV => inst_JNV,
            inst_JZ => inst_JZ,
            inst_JNZ => inst_JNZ,
            inst_JC => inst_JC,
            inst_JNC => inst_JNC,
            inst_JB => inst_JB,
            inst_JNB => inst_JNB,
            inst_SHR => inst_SHR,
            inst_SHL => inst_SHL,
            inst_ROR => inst_ROR,
            inst_ROL => inst_ROL,
            inst_HLT => inst_HLT,
            flag_N => flag_ula_control(1), -- flag_ula_control (1) negativo
            flag_Z => flag_ula_control(0),-- flag_ula_control (0) zero
            flag_V => flag_ula_control(2), -- flag_ula_control (2) overflow
            flag_B => flag_ula_control(3), -- flag_ula_control (3) borrow
            flag_C => flag_ula_control(4), -- flag_ula_control (4) carry
            carga_FLAG => carga_FLAG,
            carga_RI => carga_RI,
            sel_ULA => sel_ULA,

            sel_MUX2 => sel_mux2,
            carga_ACC => carga_ACC,
            sel_MUX1 => sel_mux,
            inc_PC => inc_PC,
            carga_PC => carga_PC,
            carga_REM => carga_REM,
            hlt => hlt,
            write_mem => iWrite(0));

memory_ahmes : memoria
  PORT MAP (
    clka => clk,
    wea => iWrite,
    addra => output_REM,
    dina => dado_entrada,
    douta => input_MUX2
  );
```

```
flags <= flag_ula_control;
dout <= dado_entrada;

end Behavioral;
```

VHDL Completo do DataPath:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
entity datapath_ahmes is
    Port ( clk : in  STD_LOGIC;
           carga_acc : in  STD_LOGIC;
           inc_pc : in  STD_LOGIC;
           carga_pc : in  STD_LOGIC;
           reset : in  STD_LOGIC;
           carga_rem : in  STD_LOGIC;
           sel_mux : in  STD_LOGIC;
           sel_mux2 : in  STD_LOGIC;
           carga_ri : in  STD_LOGIC;
           sel_ula : in  STD_LOGIC_VECTOR (3 downto 0);
           carga_flag : in  STD_LOGIC;
           endereco : out  STD_LOGIC_VECTOR (7 downto 0);
           dado_entrada : out  STD_LOGIC_VECTOR (7 downto 0);
           dado_saida : in  STD_LOGIC_VECTOR (7 downto 0);
           reg_ri : out STD_LOGIC_VECTOR(7 downto 0);
           flag_ula_control : out STD_LOGIC_VECTOR(4 downto 0));

end datapath_ahmes;

architecture Behavioral of datapath_ahmes is

--Saida ULA é entrada do acumulador
signal ula_output : std_logic_vector(7 downto 0);

--Saida do acumulador é entrada da ULA e do MUX2
signal acc_output : std_logic_vector(7 downto 0);

--saida do MUX2 e entrada do RDM
signal mux2_output : std_logic_vector(7 downto 0);

--saida do MUX e entrada do RDM
signal mux_output : std_logic_vector(7 downto 0);

--saida do RI e entrada do UC que é o reg_ri
```

```
signal ri_output : std_logic_vector(7 downto 0);

--saida do PC e entrada do MUX
signal pc_output : std_logic_vector(7 downto 0);

--saida do rem depois passa para end
signal rem_output : std_logic_vector(7 downto 0);

--saida ULA controle flag
signal flag_output : std_logic_vector(4 downto 0);

--saida registrador flag
signal regflag_output : std_logic_vector(4 downto 0);

begin

reg_acumulador: process(clk, reset)
begin
   if reset = '1' then
      acc_output <= "00000000";
   elsif clk'event and clk='1' then
      if carga_acc = '1' then
         acc_output <= ula_output;
      else
         acc_output <= acc_output;
      end if;
   end if;
end process;

reg_pc: process(clk, reset)
begin
   if reset='1' then
      pc_output <= "00000000";
   elsif clk'event and clk='1' then
      if carga_pc = '1' then
         pc_output <= mux2_output;
      elsif inc_pc = '1' then
         pc_output <= std_logic_vector(unsigned(pc_output) + 1);
      else
         pc_output <= pc_output;
      end if;
   end if;
end process;

reg_instrucao: process(clk, reset)
```

```
begin
   if reset='1' then
      ri_output <= "00000000";
   elsif clk'event and clk='1' then
      if carga_ri = '1' then
         ri_output <= mux2_output;
      else
         ri_output <= ri_output;
      end if;
   end if;
end process;
reg_ri <= ri_output;

--mux_1:
mux_output <= pc_output when sel_mux = '0' else
            mux2_output;


reg_rem: process(clk, reset)
begin
   if reset='1' then
      rem_output <= "00000000";
   elsif clk'event and clk='1' then
      if carga_rem = '1' then
         rem_output <= mux_output;
      else
         rem_output <= rem_output;
      end if;
   end if;
end process;
endereco <= rem_output;


--mux_2
mux2_output <= acc_output when sel_mux2 = '1' else
            dado_saida;

dado_entrada <= mux2_output;

ula_imp : process(mux2_output, acc_output, sel_ula, flag_output, ula_output)
variable ula_operation : std_logic_vector(8 downto 0);
variable ula_output_var : std_logic_vector(7 downto 0);
begin
   case sel_ula is
```

```
      when "0000" =>
         ula_operation := std_logic_vector(signed('0'&acc_output) +
signed('0'&mux2_output));
         ula_output_var := ula_operation(7 downto 0);
         flag_output(4) <= ula_operation(8);
         flag_output(2) <= ula_operation(7) xor ula_operation(8);
         flag_output(3) <= '0';

      when "0001" =>
         ula_operation := std_logic_vector(signed('0'&acc_output) -
signed('0'&mux2_output));
         ula_output_var := ula_operation(7 downto 0);
         flag_output(2) <= ula_operation(8);
         flag_output(3) <= (ula_operation(7) xnor ula_operation(8));
         flag_output(4) <= '0';

      when "0010" =>
         ula_output_var := (mux2_output OR acc_output);
         flag_output(3) <= '0';
         flag_output(4) <= '0';
         flag_output(2) <= '0';
         ula_operation := "000000000";

      when "0011" =>
         ula_output_var := (mux2_output AND acc_output);
         flag_output(3) <= '0';
         flag_output(4) <= '0';
         flag_output(2) <= '0';
         ula_operation := "000000000";

      when "0100" =>
         ula_output_var := (NOT acc_output);
         flag_output(3) <= '0';
         flag_output(4) <= '0';
         flag_output(2) <= '0';
         ula_operation := "000000000";

      when "0101" =>
         ula_output_var(7 downto 1) := acc_output(6 downto 0); --SHL
         ula_output_var(0) := '0';
         flag_output(4) <= acc_output(7);
         flag_output(3) <= '0';
         flag_output(2) <= '0';
         ula_operation := "000000000";
```

```
when "0110" =>
   ula_output_var(6 downto 0) := acc_output(7 downto 1);
   ula_output_var(7) := '0'; --SHR
   flag_output(4) <= acc_output(0);
   flag_output(3) <= '0';
   flag_output(2) <= '0';
   ula_operation := "000000000";

when "0111" =>
   --ROR
   ula_output_var(7) := acc_output(0);
   ula_output_var(6 downto 0) := acc_output(7 downto 1);
   flag_output(4) <= acc_output(0);
   flag_output(3) <= '0';
   flag_output(2) <= '0';
   ula_operation := "000000000";

when "1001" =>
   --y
   ula_output_var := mux2_output;
   flag_output(4) <= '0';
   flag_output(3) <= '0';
   flag_output(2) <= '0';
   ula_operation := "000000000";

when "1111" =>
    -- ROL
   ula_operation(0) := acc_output(7);
   ula_operation(7 downto 1) := acc_output(6 downto 0);
   ula_output_var := ula_operation(7 downto 0);
   flag_output(4) <= acc_output(7);
   flag_output(3) <= '0';
   flag_output(2) <= '0';
   ula_operation(8) := '0';

when others =>
   ula_operation := "000000000";
   ula_output_var := "00000000";
   flag_output <= "00000";
end case;

if (ula_output(7) = '1') then
      flag_output(1) <= '1'; --flag do negativo
else
      flag_output(1) <= '0';
```

```
      end if;
      if (ula_output = "00000000") then
         flag_output(0) <= '1'; --flag do zero
      else
         flag_output(0) <= '0';
      end if;

      ula_output <= ula_output_var;
end process;

reg_flag: process(clk)
begin
   if clk'event and clk='1' then
      if carga_flag = '1' then
         regflag_output <= flag_output;
      else
         regflag_output <= regflag_output;
      end if;
   else
      regflag_output <= regflag_output;
   end if;
end process;
flag_ula_control <= regflag_output;

end Behavioral;
```

VHDL da Unidade de Controle Completo:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity unidade_de_controle is
    Port ( clk : in STD_LOGIC;
           reset : in STD_LOGIC;
           inst_NOP : in  STD_LOGIC;
           inst_STA : in  STD_LOGIC;
           inst_LDA : in  STD_LOGIC;
           inst_ADD : in  STD_LOGIC;
           inst_OR : in  STD_LOGIC;
           inst_AND : in  STD_LOGIC;
           inst_NOT : in  STD_LOGIC;
           inst_SUB : in  STD_LOGIC;
           inst_JMP : in  STD_LOGIC;
           inst_JN : in  STD_LOGIC;
           inst_JP : in  STD_LOGIC;
```

```
            inst_JV : in  STD_LOGIC;
            inst_JNV : in  STD_LOGIC;
            inst_JZ : in  STD_LOGIC;
            inst_JNZ : in  STD_LOGIC;
            inst_JC : in  STD_LOGIC;
            inst_JNC : in  STD_LOGIC;
            inst_JB : in  STD_LOGIC;
            inst_JNB : in  STD_LOGIC;
            inst_SHR : in  STD_LOGIC;
            inst_SHL : in  STD_LOGIC;
            inst_ROR : in  STD_LOGIC;
            inst_ROL : in  STD_LOGIC;
            inst_HLT : in  STD_LOGIC;
            flag_N : in STD_LOGIC;
            flag_Z : in STD_LOGIC;
            flag_V : in STD_LOGIC;
            flag_B : in STD_LOGIC;
            flag_C : in STD_LOGIC;
            carga_FLAG : out  STD_LOGIC;
            carga_RI : out  STD_LOGIC;
            sel_ULA : out  STD_LOGIC_VECTOR (3 downto 0);
            sel_MUX2 : out  STD_LOGIC;
            carga_ACC : out  STD_LOGIC;
            sel_MUX1 : out  STD_LOGIC;
            inc_PC : out  STD_LOGIC;
            carga_PC : out  STD_LOGIC;
            carga_REM : out  STD_LOGIC;
            hlt : out STD_LOGIC;
            write_mem : out  STD_LOGIC);
end unidade_de_controle;

architecture Behavioral of unidade_de_controle is

type tipoestado is (S0, S1, S2, S3, S4 , S5, S6, S7);
signal estado, prox_estado : tipoestado;

constant ADD_inst : STD_LOGIC_VECTOR(3 downto 0) := "0000";
constant SUB_inst : STD_LOGIC_VECTOR(3 downto 0) := "0001";
constant OR_inst  : STD_LOGIC_VECTOR(3 downto 0) := "0010";
constant AND_inst : STD_LOGIC_VECTOR(3 downto 0) := "0011";
constant NOT_inst : STD_LOGIC_VECTOR(3 downto 0) := "0100";
constant SHL_inst : STD_LOGIC_VECTOR(3 downto 0) := "0101";
constant SHR_inst : STD_LOGIC_VECTOR(3 downto 0) := "0110";
constant ROR_inst : STD_LOGIC_VECTOR(3 downto 0) := "0111";
constant ROL_inst : STD_LOGIC_VECTOR(3 downto 0) := "1111";
```

```
constant Y_inst   : STD_LOGIC_VECTOR(3 downto 0) := "1001";


begin

process(clk, reset)
begin
   if reset = '1' then
      estado <= s0;
   elsif rising_edge(clk) then
      estado <= prox_estado;
   else
      estado <= estado;
   end if;
end process;


process(estado, inst_NOP, inst_STA, inst_LDA, inst_ADD, inst_OR, inst_AND,
inst_NOT, inst_SUB, inst_JMP, inst_JN, inst_JP, inst_JV, inst_JNV, inst_JZ,
inst_JNZ,
inst_JC, inst_JNC, inst_JB, inst_JNB, inst_SHR, inst_SHL, inst_ROR,
inst_ROL, inst_HLT,
flag_N, flag_Z, flag_B, flag_C, flag_V)
begin


   --declarar aqui
   prox_estado <= S0;
   hlt <= '0';
   carga_FLAG <= '0';
   carga_RI <= '0';
   sel_ULA <= "0000";
   sel_MUX2 <= '0';
   carga_ACC <= '0';
   sel_MUX1 <= '0';
   inc_PC <= '0';
   carga_PC <= '0';
   carga_REM <= '0';
   write_mem <= '0';

   Case estado is

   when S0 =>

   sel_MUX1 <= '0';
```

```
carga_REM <= '1';
prox_estado <= S1;

when S1 =>

inc_PC <= '1';
prox_estado <= S2;

when S2 =>

carga_RI <= '1';
prox_estado <= S3;

when S3 =>

if (inst_NOT = '1') then
   sel_ULA <= NOT_inst;
   carga_ACC <= '1';
   carga_FLAG <= '1';
   prox_estado <= S0;

elsif (inst_JN = '1') then
   if flag_N = '0' then
      inc_PC <= '1';
      prox_estado <= S0;
   else
      sel_MUX1 <= '0';
      carga_REM <= '1';
      prox_estado <= S4;
   end if;

elsif (inst_JZ = '1') then
   if flag_Z = '0' then
      inc_PC <= '1';
      prox_estado <= S0;
   else
      sel_MUX1 <= '0';
      carga_REM <= '1';
      prox_estado <= S4;
   end if;

elsif (inst_NOP = '1') then
   prox_estado <= S0;

elsif (inst_HLT = '1') then
```

```
      hlt <= '1';
      inc_PC <= '0';
      prox_estado <= S3;

   elsif (inst_SHR = '1') then
      sel_ULA <= SHR_inst;
      carga_ACC <= '1';
      carga_FLAG <= '1';
      prox_estado <= S0;

   elsif (inst_SHL = '1') then
      sel_ULA <= SHL_inst;
      carga_ACC <= '1';
      carga_FLAG <= '1';
      prox_estado <= S0;

   elsif (inst_ROR = '1') then
      sel_ULA <= ROR_inst;
      carga_ACC <= '1';
      carga_FLAG <= '1';
      prox_estado <= S0;

   elsif (inst_ROL = '1') then
      sel_ULA <= ROL_inst;
      carga_ACC <= '1';
      carga_FLAG <= '1';
      prox_estado <= S0;

   elsif (inst_JP = '1') then
      if flag_N = '1' then
         inc_PC <= '1';
         prox_estado <= S0;
      else
         sel_MUX1 <= '0';
         carga_REM <= '1';
         prox_estado <= S4;
      end if;

   elsif (inst_JV = '1') then
      if flag_V = '0' then
         inc_PC <= '1';
         prox_estado <= S0;
      else
         sel_MUX1 <= '0';
         carga_REM <= '1';
```

```
      prox_estado <= S4;
   end if;

elsif (inst_JNV = '1') then
   if flag_V = '1' then
      inc_PC <= '1';
      prox_estado <= S0;
   else
      sel_MUX1 <= '0';
      carga_REM <= '1';
      prox_estado <= S4;
   end if;

elsif (inst_JC = '1') then
   if flag_C = '0' then
      inc_PC <= '1';
      prox_estado <= S0;
   else
      sel_MUX1 <= '1'; ---
      carga_REM <= '1';
      prox_estado <= S4;
   end if;

elsif (inst_JNC = '1') then
   if flag_C = '1' then
      inc_PC <= '1';
      prox_estado <= S0;
   else
      sel_MUX1 <= '0';
      carga_REM <= '1';
      prox_estado <= S4;
   end if;

elsif (inst_JB = '1') then
   if flag_B = '0' then
      inc_PC <= '1';
      prox_estado <= S0;
   else
      sel_MUX1 <= '0';
      carga_REM <= '1';
      prox_estado <= S4;
   end if;

elsif (inst_JNB = '1') then
   if flag_B = '1' then
```

```
      inc_PC <= '1';
      prox_estado <= S0;
   else
      sel_MUX1 <= '0';
      carga_REM <= '1';
      prox_estado <= S4;
   end if;

elsif (inst_JNZ = '1') then
   if flag_Z = '1' then
      inc_PC <= '1';
      prox_estado <= S0;
   else
      sel_MUX1 <= '0';
      carga_REM <= '1';
      prox_estado <= S4;
   end if;

else
   sel_MUX1 <= '0';
   carga_REM <= '1';
   prox_estado <= S4;
end if;


when S4 =>

   if (inst_STA = '1') then
      inc_PC <= '1';
      prox_estado <= S5;
   elsif (inst_LDA = '1') then
      inc_PC <= '1';
      prox_estado <= S5;
   elsif (inst_ADD = '1') then
      inc_PC <= '1';
      prox_estado <= S5;
   elsif (inst_OR = '1') then
      inc_PC <= '1';
      prox_estado <= S5;
   elsif (inst_AND = '1') then
      inc_PC <= '1';
      prox_estado <= S5;
   elsif (inst_SUB = '1') then
      inc_PC <= '1';
      prox_estado <= S5;
```

```
      else
         prox_estado <= S5;
      end if;

   when S5 =>
      if (inst_STA = '1') then
         sel_MUX1 <= '1';
         carga_REM <= '1';
         prox_estado <= S6;
      elsif (inst_LDA = '1') then
         sel_MUX1 <= '1';
         carga_REM <= '1';
         prox_estado <= S6;
      elsif (inst_ADD = '1') then
         sel_MUX1 <= '1';
         carga_REM <= '1';
         prox_estado <= S6;
      elsif (inst_OR = '1') then
         sel_MUX1 <= '1';
         carga_REM <= '1';
         prox_estado <= S6;
      elsif (inst_AND = '1') then
         sel_MUX1 <= '1';
         carga_REM <= '1';
         prox_estado <= S6;
      elsif (inst_SUB = '1') then
         sel_MUX1 <= '1';
         carga_REM <= '1';
         prox_estado <= S6;
      elsif (inst_JMP = '1') then
         carga_PC <= '1';
         prox_estado <= S0;
      else
         carga_PC <= '1';
         prox_estado <= S0;
      end if;

   when S6 =>

         prox_estado <= S7;

   when S7 =>
      if (inst_STA = '1') then
         sel_MUX2 <= '1';
         write_mem <= '1';
```

```
            prox_estado <= S0;
        elsif (inst_LDA = '1') then
            sel_ULA <= Y_inst;
            carga_ACC <= '1';
            carga_FLAG <= '1';
            prox_estado <= S0;
        elsif (inst_ADD = '1') then
            sel_ULA <= ADD_inst;
            carga_ACC <= '1';
            carga_FLAG <= '1';
            prox_estado <= S0;
        elsif (inst_OR = '1') then
            sel_ULA <= OR_inst;
            carga_ACC <= '1';
            carga_FLAG <= '1';
            prox_estado <= S0;
        elsif (inst_AND = '1') then
            sel_ULA <= AND_inst;
            carga_ACC <= '1';
            carga_FLAG <= '1';
            prox_estado <= S0;
        elsif (inst_SUB = '1') then
            sel_ULA <= SUB_inst;
            carga_ACC <= '1';
            carga_FLAG <= '1';
            prox_estado <= S0;
        else
            prox_estado <= S0;
        end if;

    when others =>
        prox_estado <= S0;
    end case;

end process;

end Behavioral;
```

VHDL Completo do decodificador:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity decodificador_inst is
    port (saida_RI : in STD_LOGIC_VECTOR (7 downto 0);
```

```
          inst_NOP : out  STD_LOGIC;
          inst_STA : out  STD_LOGIC;
          inst_LDA : out  STD_LOGIC;
          inst_ADD : out  STD_LOGIC;
          inst_OR : out  STD_LOGIC;
          inst_AND : out  STD_LOGIC;
          inst_NOT : out  STD_LOGIC;
          inst_SUB : out  STD_LOGIC;
          inst_JMP : out  STD_LOGIC;
          inst_JN : out  STD_LOGIC;
          inst_JP : out  STD_LOGIC;
          inst_JV : out  STD_LOGIC;
          inst_JNV : out  STD_LOGIC;
          inst_JZ : out  STD_LOGIC;
          inst_JNZ : out  STD_LOGIC;
          inst_JC : out  STD_LOGIC;
          inst_JNC : out  STD_LOGIC;
          inst_JB : out  STD_LOGIC;
          inst_JNB : out  STD_LOGIC;
          inst_SHR : out  STD_LOGIC;
          inst_SHL : out  STD_LOGIC;
          inst_ROR : out  STD_LOGIC;
          inst_ROL : out  STD_LOGIC;
          inst_HLT : out  STD_LOGIC);
end decodificador_inst;

architecture Behavioral of decodificador_inst is
begin

process(saida_RI)
begin
   inst_NOP <= '0';
   inst_STA <= '0';
   inst_LDA <= '0';
   inst_ADD <= '0';
   inst_OR <= '0';
   inst_AND <= '0';
   inst_NOT <= '0';
   inst_SUB <= '0';
   inst_JMP <= '0';
   inst_JN <= '0';
   inst_JP <= '0';
   inst_JV <= '0';
   inst_JNV <= '0';
   inst_JZ <= '0';
```

```
inst_JNZ <= '0';
inst_JC <= '0';
inst_JNC <= '0';
inst_JB <= '0';
inst_JNB <= '0';
inst_SHR <= '0';
inst_SHL <= '0';
inst_ROR <= '0';
inst_ROL <= '0';
inst_HLT <= '0';
case saida_RI is
   when "00000000" =>
      inst_NOP <= '1';
   when "00010000" =>
      inst_STA <= '1';
   when "00100000" =>
      inst_LDA <= '1';
   when "00110000" =>
      inst_ADD <= '1';
   when "01000000" =>
      inst_OR <= '1';
   when "01010000" =>
      inst_AND <= '1';
   when "01100000" =>
      inst_NOT <= '1';
   when "01110000" =>
      inst_SUB <= '1';
   when "10000000" =>
      inst_JMP <= '1';
   when "10010000" =>
      inst_JN <= '1';
   when "10010100" =>
      inst_JP <= '1';
   when "10011000" =>
      inst_JV <= '1';
   when "10011100" =>
      inst_JNV <= '1';
   when "10100000" =>
      inst_JZ <= '1';
   when "10100100" =>
      inst_JNZ <= '1';
   when "10110000" =>
      inst_JC <= '1';
   when "10110100" =>
      inst_JNC <= '1';
```

```
      when "10111000" =>
         inst_JB <= '1';
      when "10111100" =>
         inst_JNB <= '1';
      when "11100000" =>
         inst_SHR <= '1';
      when "11100001" =>
         inst_SHL <= '1';
      when "11100010" =>
         inst_ROR <= '1';
      when "11100011" =>
         inst_ROL <= '1';
      when "11110000" =>
         inst_HLT <= '1';
      when others =>
         inst_HLT <= '1';
   end case;
end process;

end Behavioral;
```

TestBench VHDL Completo:

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY tb_ahmes IS
END tb_ahmes;

ARCHITECTURE behavior OF tb_ahmes IS

    -- Component Declaration for the Unit Under Test (UUT)

    COMPONENT ahmes
    PORT(
         clk : IN  std_logic;
         reset : IN  std_logic;
         flags : OUT  std_logic_vector(4 downto 0);
         dout : OUT  std_logic_vector(7 downto 0);
         hlt : OUT  std_logic
        );
    END COMPONENT;


   --Inputs
   signal clk : std_logic := '0';
```

```
   signal reset : std_logic := '0';

   --Outputs
   signal flags : std_logic_vector(4 downto 0);
   signal dout : std_logic_vector(7 downto 0);
   signal hlt : std_logic;

   -- Clock period definitions
   constant clk_period : time := 10 ns;

BEGIN
   -- Instantiate the Unit Under Test (UUT)
   uut: ahmes PORT MAP (
          clk => clk,
          reset => reset,
          flags => flags,
          dout => dout,
          hlt => hlt
        );

   -- Clock process definitions
   clk_process :process
   begin
      clk <= '0';
      wait for clk_period/2;
      clk <= '1';
      wait for clk_period/2;
   end process;

   -- Stimulus process
   stim_proc: process
   begin
      -- hold reset state for 100 ns.
      wait for 100 ns;
      reset <= '1';
      wait for clk_period*2;
      reset <= '0';
      wait for clk_period*10;

      wait;
   end process;

END;
```

Operação de multiplicação: (4 x 2 = 8)

NOP
LDA 29 (LDA no operando 1 == 2)
STA 33 (salva no endereço 33 que vai ser o contador)
LDA 35 (zera o valor do endereço 30 que contém a resposta final)
STA 30
Desvio_1: LDA 30 (LDA no valor que estará o resultado final)
ADD 28 (Adiciona o valor do operando 2 == 4)
STA 30 (salva no edenreço do resultado final)
LDA 33 (dá LDA no contador)
SUB 34 (decrementa em porque o endereço 34 contém a constante 1)
STA 33 (contador decrementado é salvo)
JZ 25 (testa se o contador é zero, se for vai para o endereço do HLT)
JMP 9 (desvio_1 == 9)
HLT
0
0
4 (addr 28)
2 (addr 29)
0
0
0
0 (addr 33)
1 (constante 1 para decrementar o contador)
0

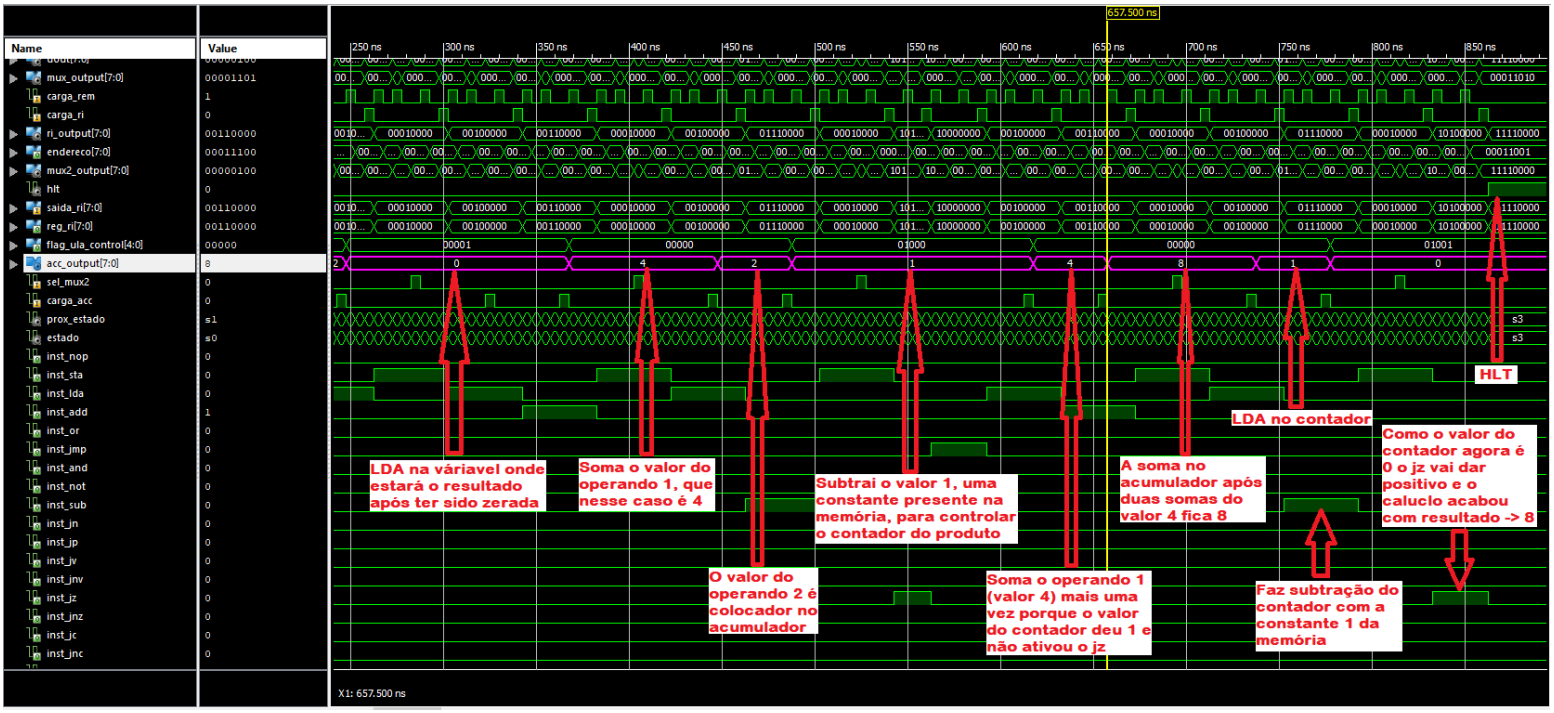
Assembly do arquivo ".coe":

MEMORY_INITIALIZATION_RADIX= 10;
MEMORY_INITIALIZATION_VECTOR= 0, 32, 29, 16, 33, 32, 35, 16, 30, 32, 30, 48, 28, 16, 30, 32, 33, 112, 34, 16, 33, 160, 25, 128, 9, 240, 0, 0, 4, 2, 0, 0, 0, 0, 1, 0;

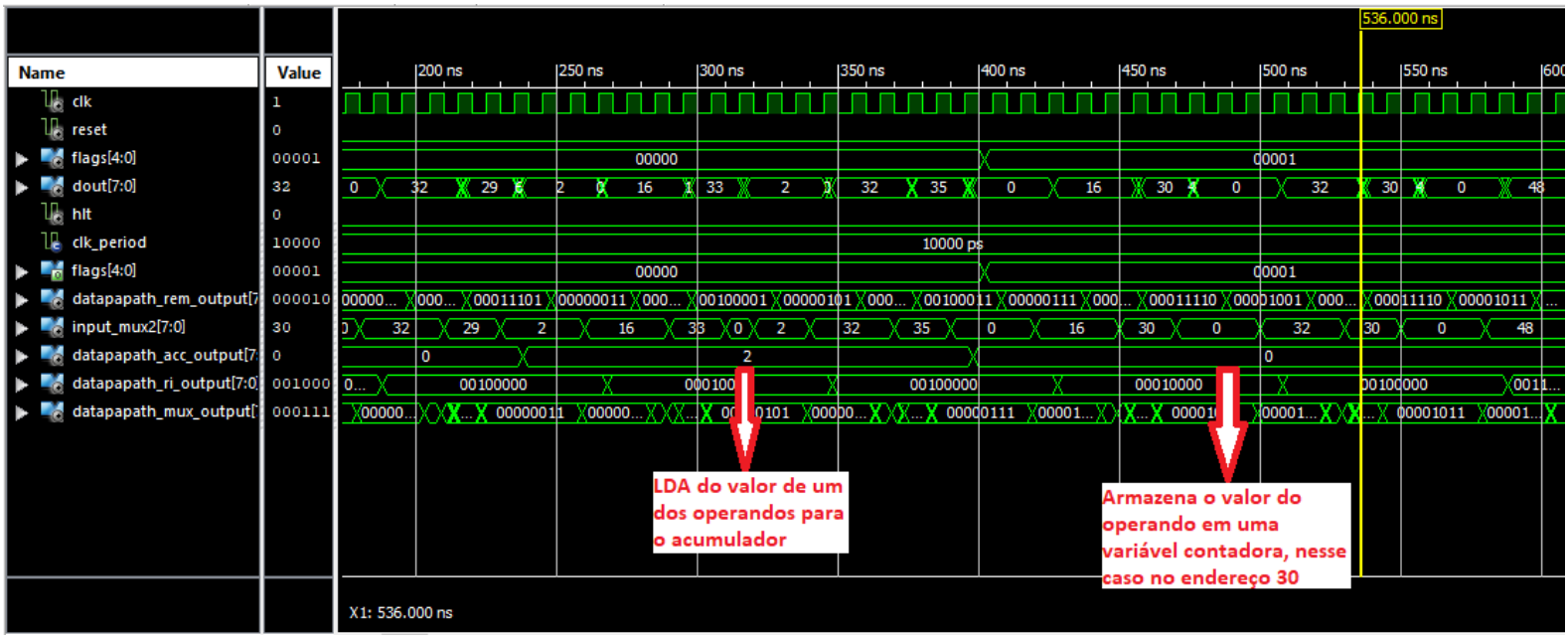
- Valor vermelho: Operando 1
- Valor azul: Operando 2
- Valor Roxo: Constante subtratora

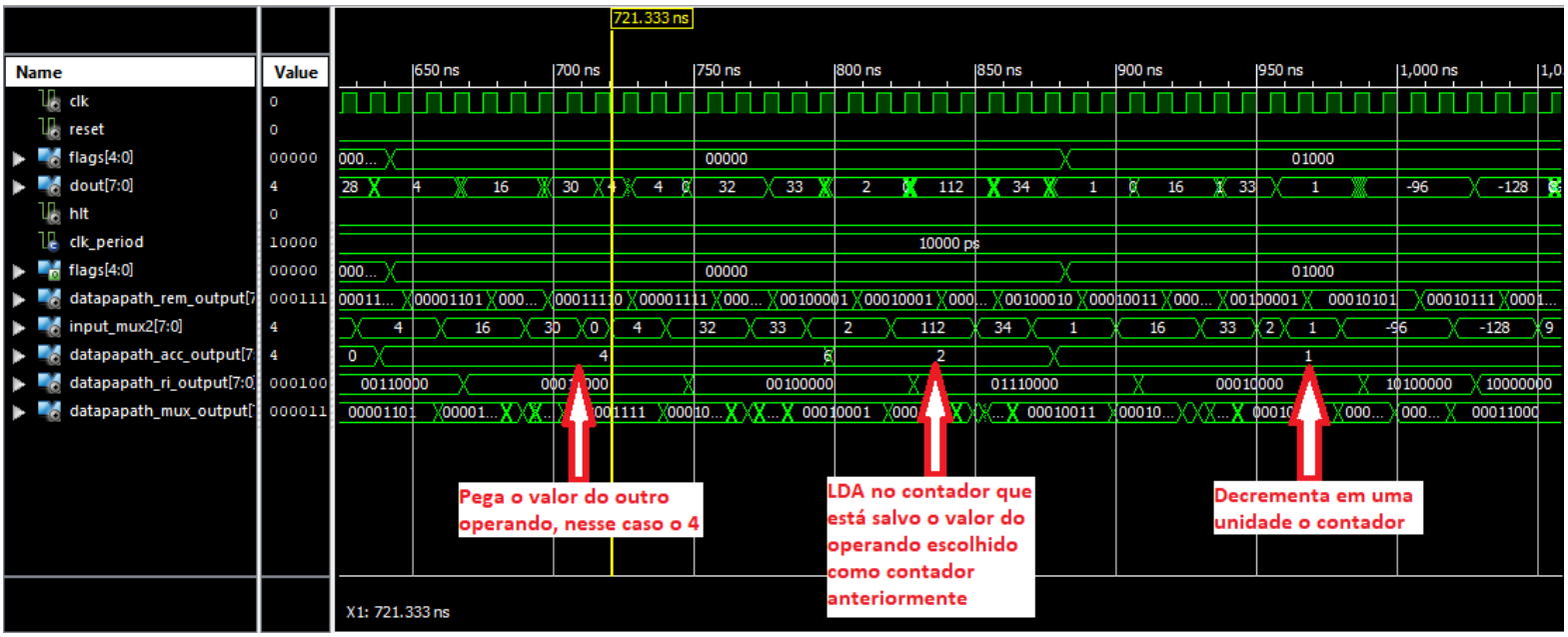
Simulações sem e com atraso com detalhes e flechas mostrando início meio e final do programa e resultados:

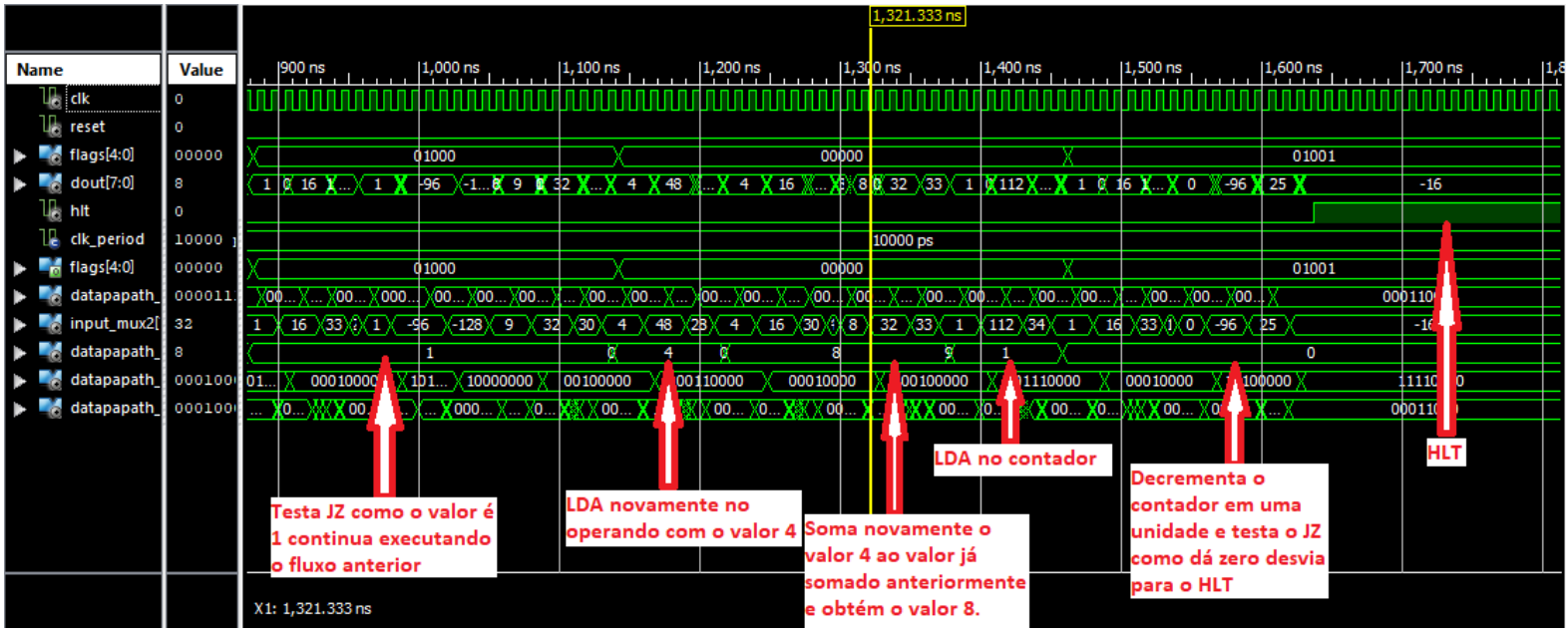
Sem Atraso:



Com Atraso:




721.333 ns
Name
Value
clk
reset
flags[4:0]
dout[7:0]
hlt
clk_period
flags[4:0]
datapapath_rem_output[7
input_mux2[7:0]
datapapath_acc_output[7
datapapath_ri_output[7:0]
datapapath_mux_output[
650 ns
700 ns
750 ns
800 ns
850 ns
900 ns
950 ns
1,000 ns
10000 ps
Pega o valor do outro operando, nesse caso o 4
LDA no contador que está salvo o valor do operando escolhido como contador anteriormente
Decrementa em uma unidade o contador
X1: 721.333 ns


1,321.333 ns
Name
Value
clk
reset
flags[4:0]
dout[7:0]
hlt
clk_period
flags[4:0]
datapapath_
input_mux2[
datapapath_
datapapath_
datapapath_
900 ns
1,000 ns
1,100 ns
1,200 ns
1,300 ns
1,400 ns
1,500 ns
1,600 ns
1,700 ns
10000 ps
Testa JZ como o valor é 1 continua executando o fluxo anterior
LDA novamente no operando com o valor 4
Soma novamente o valor 4 ao valor já somado anteriormente e obtém o valor 8.
LDA no contador
Decrementa o contador em uma unidade e testa o JZ como dá zero desvia para o HLT
HLT
X1: 1,321.333 ns

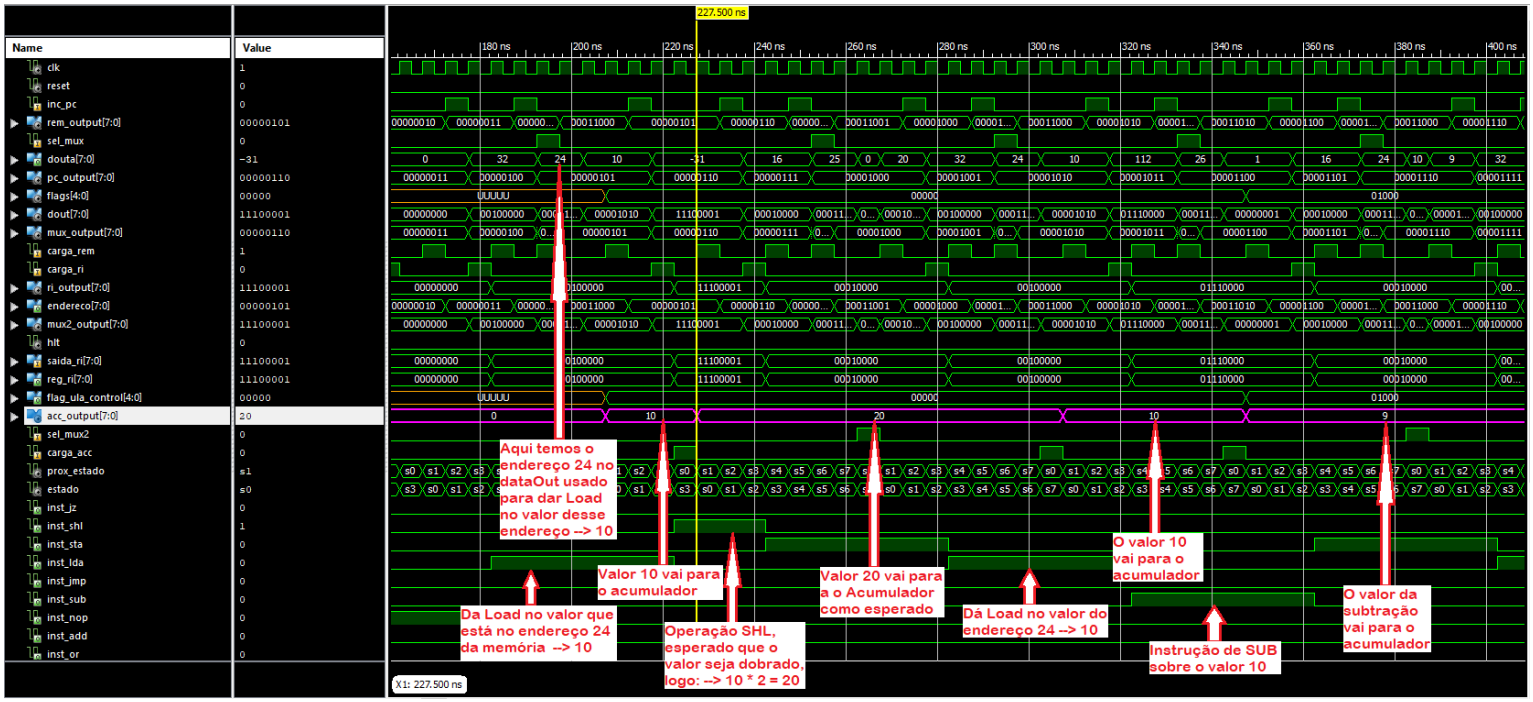
Operação de SHL e Decrementador com desvio (JZ) e (JMP):

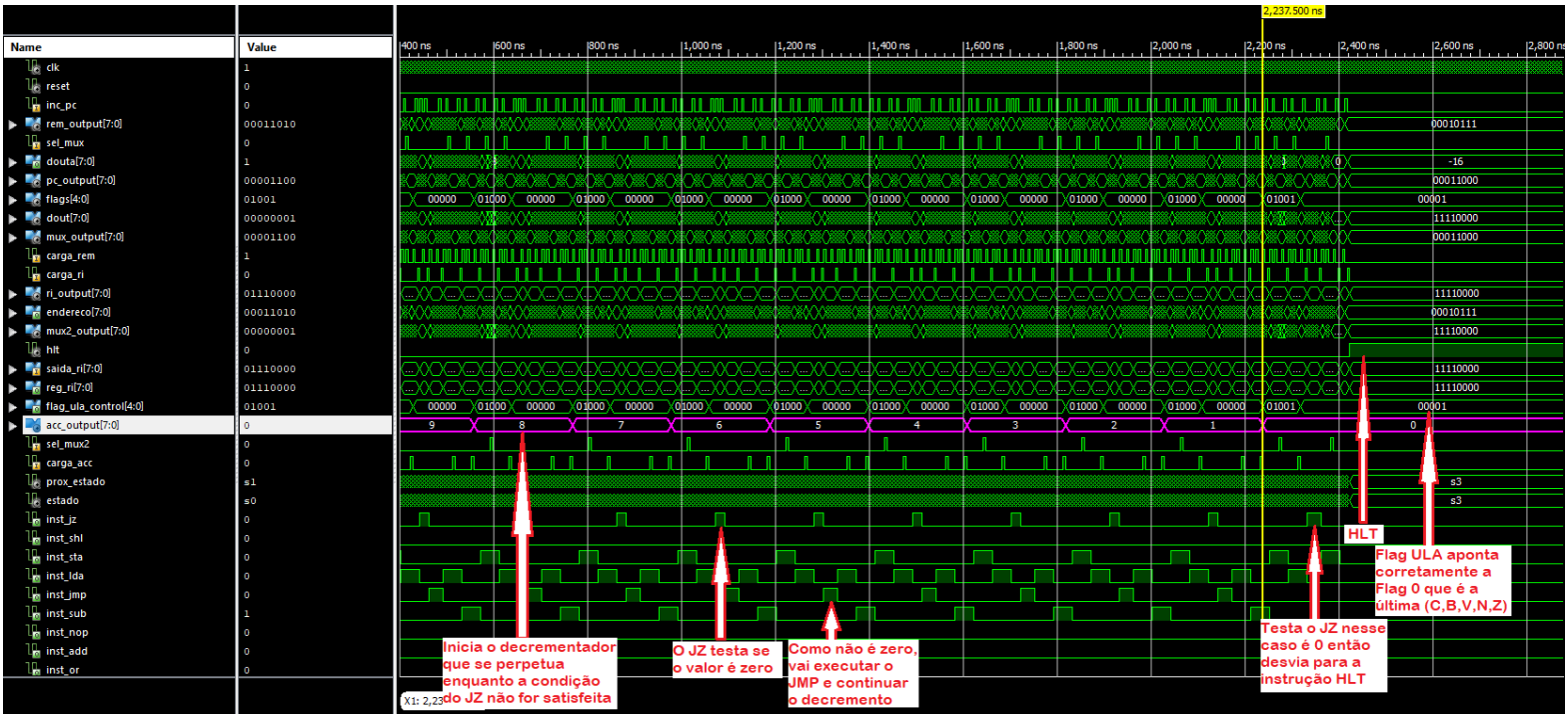
NOP
NOP
NOP
LDA 24 (LDA no endereço 24 que contém o valor 10)
SHL (SHL em complemento de 2 gera uma multiplicação por 2)
STA 25 (salva no endereço 25 a resposta do SHL)
Desvia_1: LDA 24 (Dá LDA no endereço 24 → 10)
SUB 26 (Decrementa em uma unidade esse valor porque no endereço 26 há → 1)
STA 24 (salva no endereço 24 o resultado do decremento)
LDA 24 (dá LDA nesse valor salvo)
JZ 20 (Se o valor é zero vai para o endereço 20) → addr_20
JMP 8 (senão vai para o endereço de reinicio do fluxo) → (Desvia_1)
Addr_20: STA 24
NOP
HLT
10 (addr 24)
0
1 (constante subtratora)

Assembly do arquivo ".coe":

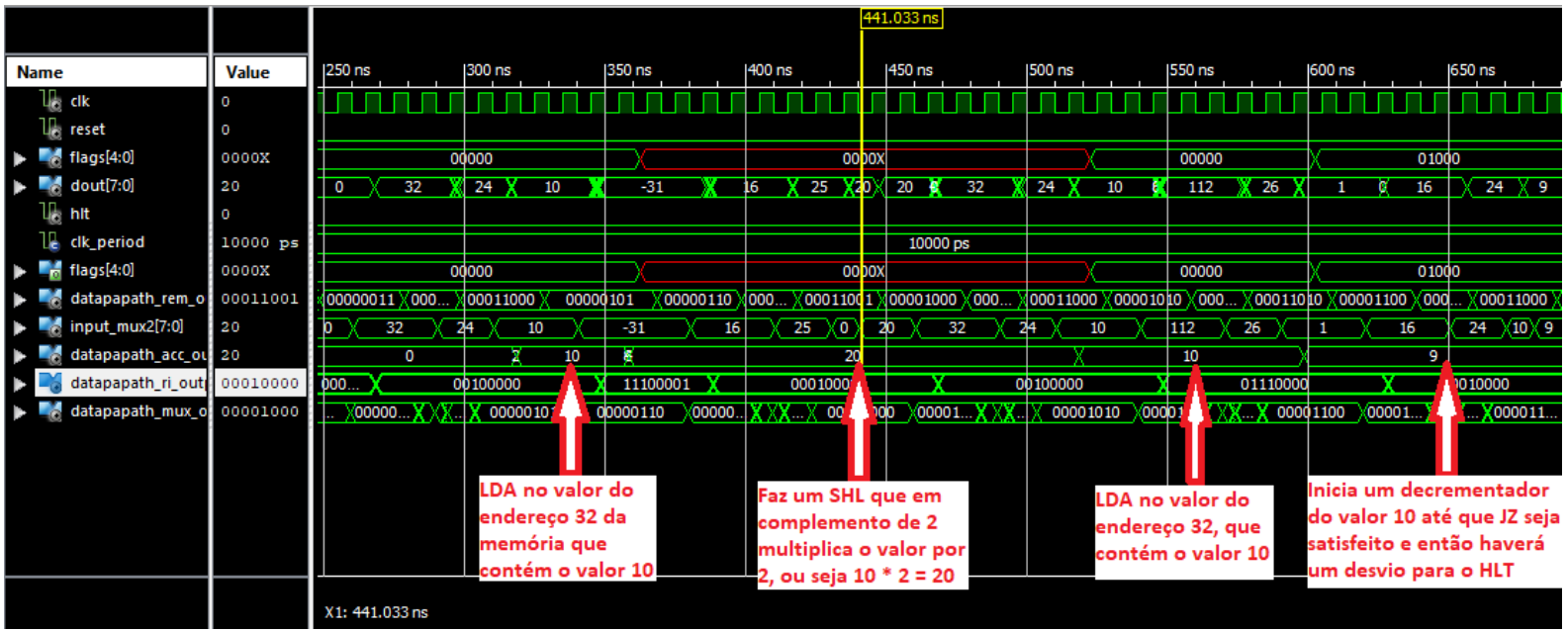
MEMORY_INITIALIZATION_RADIX= 10;
MEMORY_INITIALIZATION_VECTOR= 0, 0, 0, 32, 24, 225, 16, 25, 32, 24, 112, 26, 16, 24, 32, 24, 160, 20, 128, 8, 16, 24, 0, 240, 10, 0, 1;

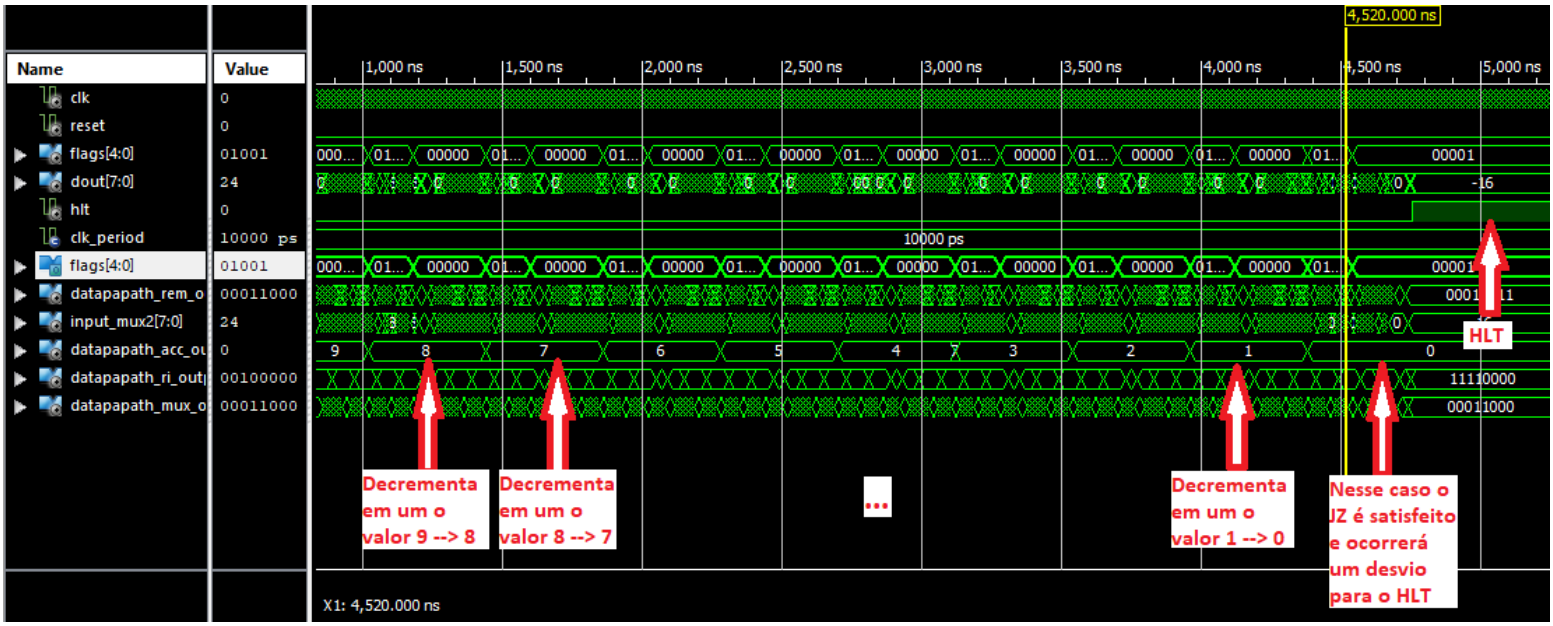
Sem Atraso:




Name
Value
clk
reset
inc_pc
rem_output[7:0]
sel_mux
douta[7:0]
pc_output[7:0]
flags[4:0]
dout[7:0]
mux_output[7:0]
carga_rem
carga_ri
ri_output[7:0]
endereco[7:0]
mux2_output[7:0]
hlt
saida_ri[7:0]
reg_ri[7:0]
flag_ula_control[4:0]
acc_output[7:0]
sel_mux2
carga_acc
prox_estado
estado
inst_jz
inst_shl
inst_sta
inst_lda
inst_jmp
inst_sub
inst_nop
inst_add
inst_or
2,237.500 ns
Inicia o decrementador que se perpetua enquanto a condição do JZ não for satisfeita
O JZ testa se o valor é zero
Como não é zero, vai executar o JMP e continuar o decremento
Testa o JZ nesse caso é 0 então desvia para a instrução HLT
HLT
Flag ULA aponta corretamente a Flag 0 que é a última (C,B,V,N,Z)

Com Atraso:


Name
Value
clk
reset
flags[4:0]
dout[7:0]
hlt
clk_period
flags[4:0]
datapapath_rem_o
input_mux2[7:0]
datapapath_acc_ou
datapapath_ri_out
datapapath_mux_o
441.033 ns
10000 ps
LDA no valor do endereço 32 da memória que contém o valor 10
Faz um SHL que em complemento de 2 multiplica o valor por 2, ou seja 10 * 2 = 20
LDA no valor do endereço 32, que contém o valor 10
Inicia um decrementador do valor 10 até que JZ seja satisfeito e então haverá um desvio para o HLT
X1: 441.033 ns



Dados de área, tempo de execução em ciclos de relógio e tempo em segundos deve ser apresentado dado um determinado clock usado:

Programa	Número de Instruções Executadas	Tempo de execução em # de ciclos de relógio (c.c.)	Tempo de execução em Segundos (Neander operando a 50 MHz)
Multiplicação por somas sucessivas	19	148	1480 ns
Programa com SUB e SHL	64	454	4540 ns