LAB 2

Diogo Moura- 86976

António Fragoso-79116

Semana 1(Ciclo Único)

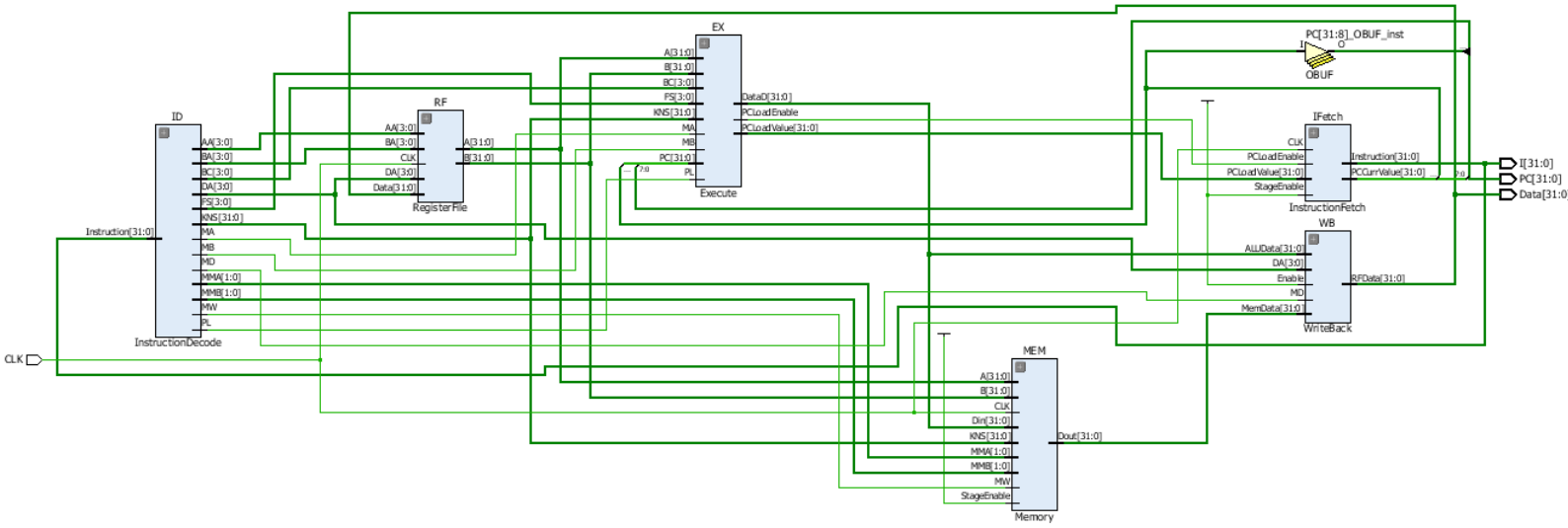

Fig1: Esquema do Processador de Ciclo Único

4.1

a)

O *opcode* decide qual a operação a realizar sobre os dados pelo processador.

O sinal PL faz o *enable* do *Load Program Counter,* pelo que será apenas utilizado nas condições de salto.

A entrada dDA está ligada ao endereço da memória e ao registo de destino , isto é o registo a ser escrito. Dependendo da situação escolhe o endereço de memória utilizado ou o registo que irá ser escrito.

Os sinais AA e BB são sinais de entrada dos MUXs A e B do *RegisterFile*, respetivamente, sendo que escolhem um dos registos para ter a sua saída redirecionada para as saídas, respetivamente A e B do *RegisterFile*.

O sinal FS escolhe a operação a ser realizada na Unidade Funcional, de acordo com a Tabela 2.

Tipo	Operação (mnemónica)	Entradas	
		Operandos	FSUF
Aritmética	ADD	A , B	0000
	ADD+	A , B	0001
	SUB-	A , B	0010
	SUB	A , B	0011
Lógica	AND	A , B	0100
	NAND	A , B	0101
	OR	A , B	0110
	NOR	A , B	0111
	XOR	A , B	1000
	XNOR	A , B	1001
Deslocamento	SHL	B	1010
	SHR	B	1011
	SHLA	B	1100
	SHRA	B	1101
	ROL	B	1110
	ROR	B	1111

Tabela1: Operações realizadas na Unidade Funcional, de acordo com o sinal FS

O bit mais significativo dos sinais MASel(1), MBSel(1) e MDSel(1) escolhe se os registos que irão ser escritos/lidos a partir do Register File são escolhidos pelo sinais SA, SB, ou DR da *Instruction Memory* ou pelo sinais AA, BA ou DA, que estão definidos no *Instruction Decoder*.

```
with MASel(1) select
   AA <= SA  when '0',      -- escolha do dado de endereço do registo  A do register file
        dAA when others;

with MBSel(1) select
   BA <= SB  when '0',      -- escolha do  dado do endereço registo  do registo B do register file
        dBA when others;

with MDSel(1) select
   DA <= DR  when '0',          -- registo que é escrito no Register File
        dDA when others;

MA <= MASel(0);      -- escolher entre a entrada A ou KNS da ALu
MB <= MBSel(0);      -- escolher entre a entrada B ou KNS da ALU
MD <= MDSel(0);      --escolher se se escreve o conteúdo da memória (a 1) ou o da ALU(a 0)
```

Fig2: Função de MASEL, MBSEL, MDSEL

Os sinais MMA e MMB escolhem respetivamente, o endereço e os dados que irão ser escritos na memória.

```
with MMA select
   Address <= KNS when "00",
            A   when "01",
            B   when "10",
            Din when others;   --escolher o endereço dos dados a serem escritos na memória.


with MMB select
   MemDataIn <= KNS when "00",
            A   when "01",
            B   when "10",
            Din when others;    --escolher os dados a serem escritos na memória.
```

Fig3: Função de MMA e MMB

O sinal MW permite o enable da escrita na memória. Apenas é utilizado em operações de escrita na mesma

O sinal KNSSel permite a manipulação da constante KNS ,a qual irá sempre ser estendida para 32 bits de acordo com este sinal. Esta manipulação revela-se de particular interesse, tendo em conta que , para cada operação, existe uma constante diferente a ser utilizada.

```
with KNSSel select
   KNS  <= (31 downto 18=>'0') & Instruction(17 downto 0)          when "000",
         (31 downto 18=>Instruction(17)) & Instruction(17 downto 0) when "001",
         (31 downto 18=>Instruction(25)) & Instruction(25 downto 22) & Instruction(13 downto
0) when "010",
         (31 downto 14=>Instruction(13)) & Instruction(13 downto 0) when "011",
         (31 downto 16=>'0')          & Instruction(15 downto 0) when "100",
         (31 downto 16=>'1')          & Instruction(15 downto 0) when "101",
         Instruction(15 downto 0)      & (31 downto 16=>'0')     when "110",
         Instruction(15 downto 0)      & (31 downto 16=>'1')     when others;
```

Fig4: Função de KNSSel

b)

OPCode	Mnemónica	PL	dAA	dBA	dDA	FS	KNSSel	MASEl	MBSEL	MMA	MMB	MW	MDSEL
000000	ADD	0	XXXX	XXXX	XXXX	0000	XXX	00	00	XX	XX	0	00
000001	ADDI	0	XXXX	XXXX	XXXX	0000	001	00	X1	XX	XX	0	00
000100	AND	0	XXXX	XXXX	XXXX	0100	XXX	00	00	XX	XX	0	00
001100	XOR	0	XXXX	XXXX	XXXX	1000	XXX	00	00	XX	XX	0	00
010100	LD	0	XXXX	XXXX	XXXX	0000	XXX	00	00	11	XX	0	01
010011	SHRA	0	XXXX	XXXX	XXXX	1101	XXX	00	00	XX	XX	0	00
011000	BI.NE	1	XXXX	XXXX	0000	0011	011	00	01	XX	XX	0	01
010111	B	1	XXXX	XXXX	0000	0011	011	00	00	XX	XX	0	01

Tabela 2: Memória de Descodificação

4.2)

Para efetuar as alterações pedidas, utilizámos um MUX que seleciona, de acordo com o sinal, *BranchControl* , o *Load* do *ProgramCounter*. Deste modo, só ocorrerá o salto quando , para o *BranchControl* designado na instrução se verificar as condições de *flags* correspondentes. Deste modo, o *BranchControl* seleciona as condições pretendidas para o salto.

```
with BC( 3 downto 0) select
   PCLoad<= PL when "0000" | "0001",
            PL and Z when "0010",
            PL and (not Z) when "0011",
            PL and P when "0100",
            PL and (P or Z) when "0101",
            PL and (not Z and N) when "0110",
            PL and (Z or N) when "0111",
            '0' when others;
PCValue<= PC+AD;
```

Fig5: Código VHDL

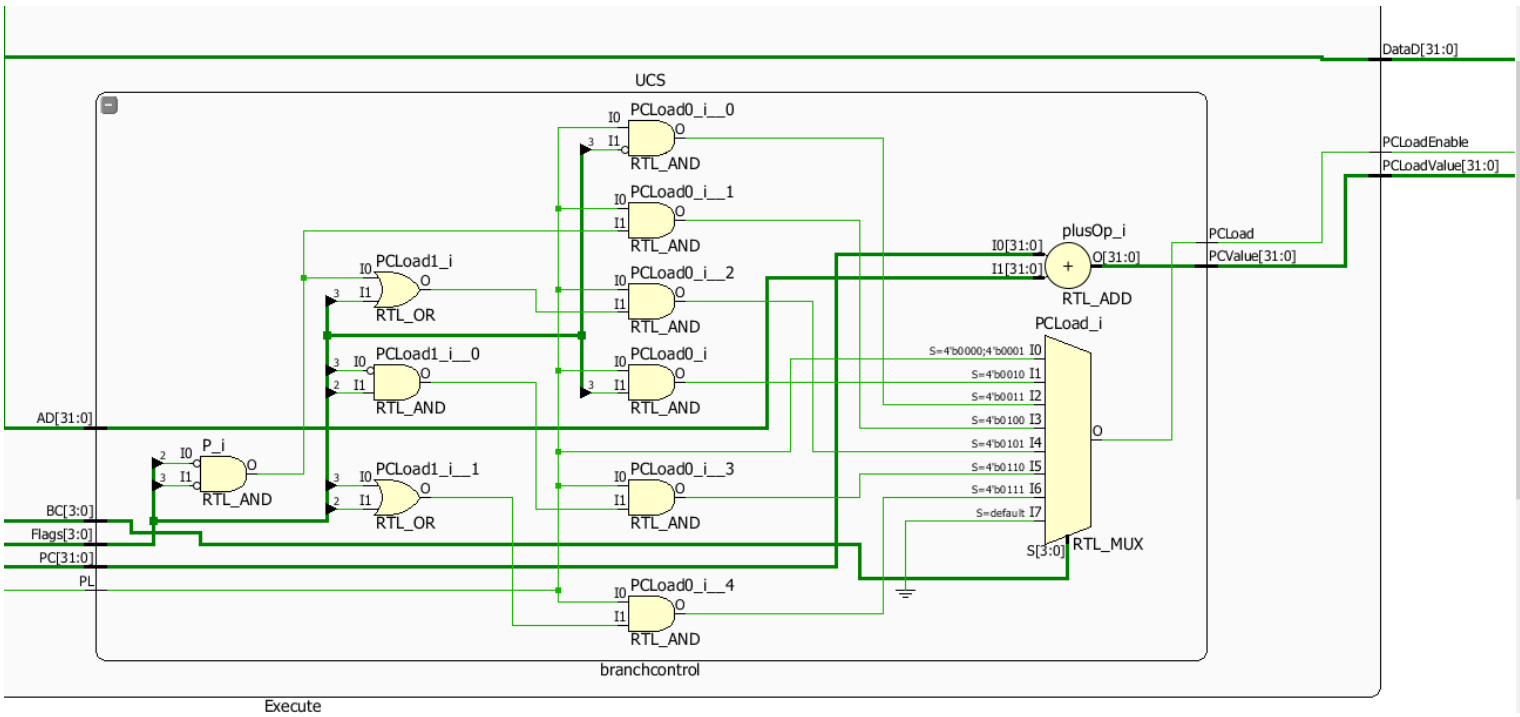

Fig6: Esquema do BranchControl

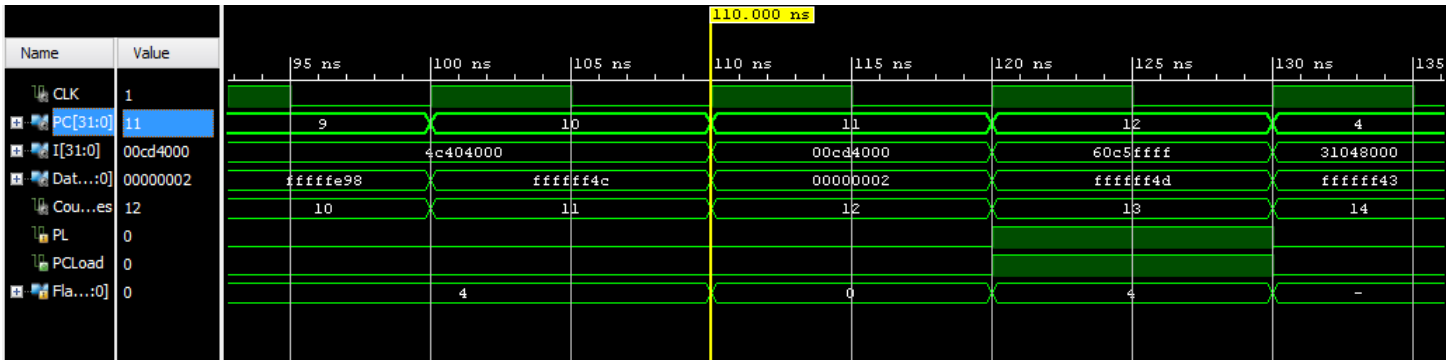

Fig7: Teste ao BranchControl

Analisando os valores do PC, PL e PCLoad, observamos que os saltos são executados corretamente e após a instrução 12(de salto) para o valor inicial do Loop(4),

4.3 Teste das Instruções

Para calcular o número de ciclos de relógio é necessário ter em conta o número de SHRA executados por cada LOOP (4) e o número de SHRA necessários para que o valor guardado no registo 1 se altere para #-1, que será 12, dado que o número inicial guardado no R1(#-2876) tem representação binária com 18 bits de 11 1111 0100 1100 0011. Assim, o número de LOOPs executados será 12/4=3.

O número de ciclos de relógio necessários à correta execução do troço de código é:

N.total= N. Instruções Fora do LOOP+3* Instruções Dentro do LOOP+1=4+3*9+1=32ciclos

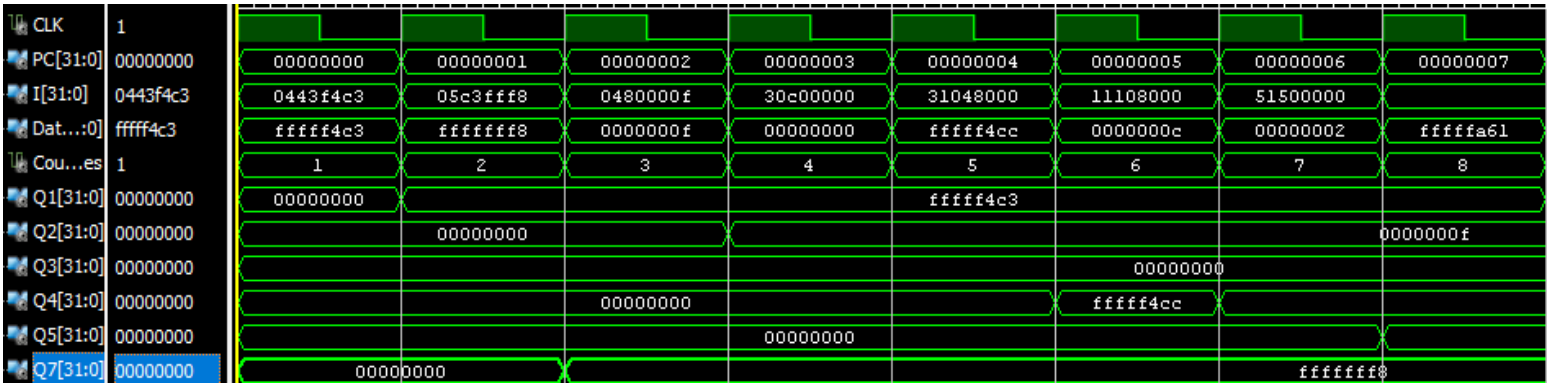

Fig8: Início da simulação

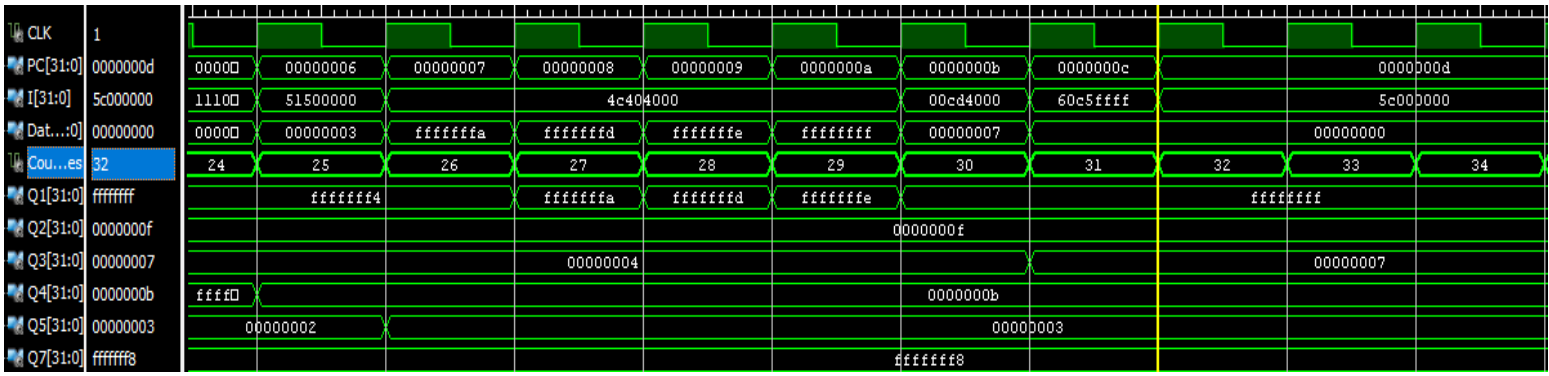

Fig9: Fim da simulação

A simulação é iniciada conforme o esperado sendo que o valor guardado no registo 1 é -2876, isto é, FFFFF4C3h em hexadecimal, no primeiro ciclo de relógio.

O registo 1 *Arithmetic Shift Rights sucessivos* até que o valor guardado no mesmo é -1 e o Loop chega ao fim, assim como o programa, logo de seguida, a seguir à instrução B #0.

Ao fim de 31 ciclos de relógio, os valores guardados nos registos permanecem inalterados e são, respetivamente, para os registos 1, 2, 3, 4, 5 e 7 FFFFFFFFh, 0000000Fh, 00000007h, 0000000Bh, 00000003h e FFFFFFF8h, conforme o previsto. Se somarmos a este número (31) o ciclo do Branch final, obteremos o valor previsto de ciclos necessários (32).

Semana 2(Pipeline)

5.1 Cálculo teórico do desempenho do processador de ciclo único

Para determinar o tempo crítico e a frequência de relógio é necessário ter em conta os tempos de propagação nos seguintes Estágios:

- IF (Read PC from register◊Memory◊Instruction): 35ns
- IF (PC ◊Adder◊MUX◊Write on PC Register): 15ns
- ID (Instruction◊Decoder◊RF◊A,B): 30ns
- EX (A,B◊UF◊Data): 30ns
- EX (A,B◊UF◊Branch Control◊PCLoadEnable,PCLoadValue◊MUX◊Write on PC Register): 40ns
- MEM (A,B,D◊Write to Memory): 30ns
- MEM (A,B,D◊Read from Memory): 40ns
- WB (Data◊Write to register file): 15ns

Tempo crítico :

MAX

{

- T.propagação(PC ->Adder->MUX->Write on PC Register)=T.IF=15ns
- T. propação= T.IF +T.ID+T.EX(A,B->UF->Branch Control->PCLoadEnable->PCLoadValue->MUX ->Write on PC Register)=35+30+40=105ns
- T.propagação= T.IF +T.ID+T.EX(A,B->UF->Data)+TMEM(A,B,D->Write to Memory)

  =35+30+30+30=125ns
- T.propagação= T.IF+T.ID+T.EX(A,B->UF->Data)+T.MEM(A,B,D->Read from Memory)+T.WB((Data->Write to register file)=35+30+30+40+15=150ns

}=150 ns

Notas:

Não existe nenhuma instrução que escreva numa memória e no registo simultaneamente, pelo que não é necessário considerar a hipótese em que o tempo de propagação soma o tempo TMEM(de escrita na memória) e , de seguida, o tempo de WB.

Não é preciso considerar no tempo de propagação a soma do tempo TMEM(de escrita na memória) somado com o tempo WB (de escrita no registo), dado que não existe nenhuma instrução que realiza as duas ações simultaneamente.

5.2 Frequência de Relógio de um Processador Pipeline

Cálculo da Frequência

F. Máx=1/T.crítico=1/150ns=6,6667GHz

Cáculo da Frequência em Pipeline

Para os Registos->TSetup=TPropagação=1ns

T.crítico =T.MEM (A,B,D->Read from Memory)+T.Propagação+T.Setup=40+1+1=42ns

Frequência =1/T.crítico=1/42ns=23,8095GHz

$$Speedup = \frac{T_{Ciclo\ Único}}{T_{Pipelined}}$$

=150/42=3,571

5.3 Execução de um troço de código num processador pipelined

1.

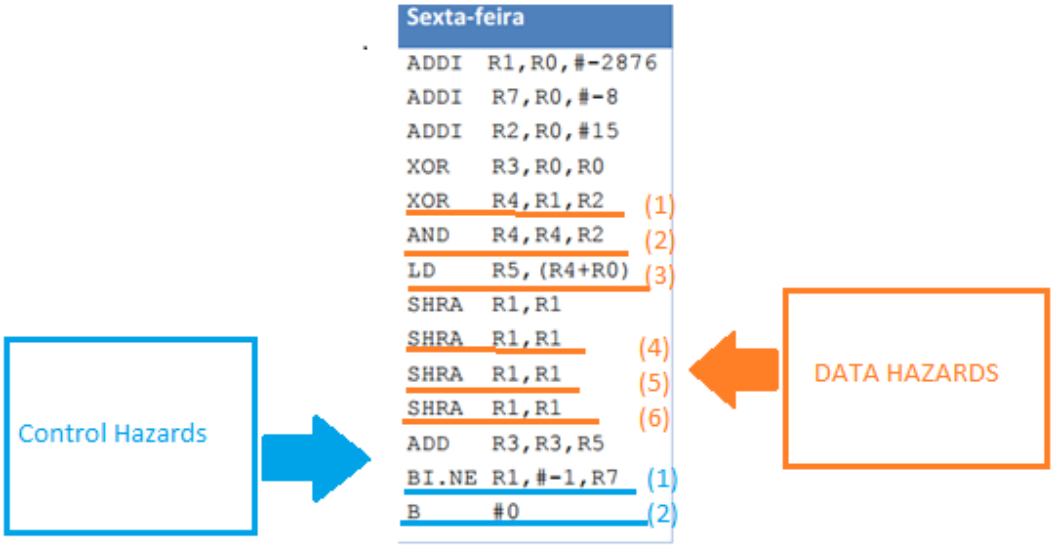

Fig10: Identificação e numeração dos Conflitos de Controlo e Dados

CONFLITOS DE DADOS (1 a 6)

Os conflitos de Dados existentes devem-se ao fato de o "Operand Fetch" nas operações assinaladas não poder ser executado corretamente dado que a operação anterior ainda não executou o "Write Back".

Esquema:

Instrução1 : IF ID EXEC MEM WB

Instrução2 : IF ID EXEC MEM WB

Os conflitos de dados resolvem-se com a Introdução de 3 instruções NOPS

Esquema:

Instrução1 : IF ID EXEC MEM WB

Instrução2 NOP : IF ID EXEC MEM WB

Instrução NOP : IF ID EXEC MEM WB

Instrução NOP : IF ID EXEC MEM WB

Instrução2 : IF ID EXEC MEM WB

No momento do "Operand Fetch" da instrução seguinte os registos já terão de estar escritos na instrução anterior se os dados forem utilizados.

CONFLITOS DE CONTROLO (2)

Esquema:

Instrução BI.NE : IF ID EXEC MEM WB

Instrução NOP : IF ID EXEC MEM WB

Instrução NOP: IF ID EXEC MEM WB

Instrução B #0 : IF ID EXEC MEM WB

No momento do IF da segunda condição de Salto ainda não foi concluída a instrução de salto anterior, uma vez que o endereço da instrução seguinte só é obtido após o EXEC da instrução BI.NE. É necessário aguardar

CONFLITOS DE CONTROLO (1)

Esquema:

Instrução SHRA R1, R1 : IF ID EXEC MEM WB

Instrução ADD R3, R5 : IF ID EXEC MEM WB

Instrução NOP: IF ID EXEC MEM WB

Instrução NOP: IF ID EXEC MEM WB

Instrução B.NE R1, #-1 : IF ID EXEC MEM WB

No momento do ID da instrução BI.NE ainda não foi concluída a escrita WB no registo R1 da instrução SHRA. Seria necessário introduzir dois NOPs, para que a instrução de salto fosse executada de acordo com o valor do R1 adequado e dependente do SHRA .

2.

Para resolver os conflitos de dados e de controlo, a ordem das instruções foi trocada entre si e foram introduzidos NOPS entre instruções.

```
-2 semana (Pipeline)--alterada a ordem das instruções para minimizar o número de instruções NOP
gnal storage: storage_type := (
        --    OPCODE &   DR   &   SA   &   SB   &         KNS         -- ASSEMBLY CODE
       0 => "000001" & "0001" & "0000" & "1111" & "11010011000011", --        ADDI  R1,R0,#-2876
       1 => "000001" & "0111" & "0000" & "1111" & "11111111100111", --        ADDI  R7,R0,#-25
       2 => "000001" & "0010" & "0000" & "0000" & "00000000001111", --        ADDI  R2,R0,#15
       3=> x"00000000",--NOP
       4=> x"00000000",--NOP
       5 => "001100" & "0011" & "0000" & "0000" & "00000000000000", --        XOR   R3,R0,R0
       6 => "001100" & "0100" & "0001" & "0010" & "00000000000000", -- LOOP:  XOR   R4,R1,R2
       7=> x"00000000",--NOP
       8=> x"00000000",--NOP
       9 => "010011" & "0001" & "0000" & "0001" & "00000000000000", --        SHRA  R1,R1
       10 => "000100" & "0100" & "0100" & "0010" & "00000000000000", --        AND   R4,R4,R2
       11=> x"00000000",--NOP
       12=> x"00000000",--NOP
       13 => "010011" & "0001" & "0000" & "0001" & "00000000000000", --        SHRA  R1,R1
      14 => "010100" & "0101" & "0100" & "0000" & "00000000000000", --        LD    R5,(R4+R0)
       15=> x"00000000",--NOP
       16=> x"00000000",--NOP
       17 => "010011" & "0001" & "0000" & "0001" & "00000000000000", --        SHRA  R1,R1
       18=> x"00000000",--NOP
       19=> x"00000000",--NOP
       20 => "000000" & "0011" & "0011" & "0101" & "00000000000000", --        ADD   R3,R3,R5
      21=> "010011" & "0001" & "0000" & "0001" & "00000000000000", --        SHRA  R1,R1
      22=> x"00000000",--NOP
      23=> x"00000000",--NOP
      24=> x"00000000",--NOP
      25 => "011000" & "0011" & "0001" & "0111" & "11111111111111", --        BI.NE R1,#-1,R7
      26=> x"00000000",--NOP
      27=> x"00000000", --NOP
      28 => "010111" & "0000" & "0000" & "0000" & "00000000000000", -- END:   B     #0
  others => x"00000000" -- NOP
   );
```

Fig11: Código VHDL das alterações efetuadas

3.

Ciclo unico = 32 ciclos; Ciclo = 160ns; T_unico = 160x32 = 5120ns

Pipelined = N.ciclos.fora.Loop+ N.Ciclos.Loop *3+ Instrução.B#0+ (N.andares-1)

=6+3*22+1+4 ciclos= 77; Ciclo = 42ns;

T_pipeline = 77x42 = 3234ns

Speedup_real = T_unico / T_pipeline = 1.58

Conclusão

Concluímos que uma arquitetura Pipeline traz grandes benifícios, e que o *speedup* vai aumentando com o número de instruções efetuadas. Ainda assim, este tipo de arquitetura poderá ser menos rápida que a de Ciclo Único se for executado um número de instruções bastante reduzido e interdependente, que exija introdução de NOPs.