CURSO: ENGENHARIA ELETROTÉCNICA E DE COMPUTADORES

DISCIPLINA: ARQUITETURA DE COMPUTADORES

TRABALHO DE LABORATÓRIO II

CONCEÇÃO E ANÁLISE DE UM PROCESSADOR

Trabalho Realizado por: Xavier Abreu Dias Nº 87136

Diogo Martins Alves Nº 86980

Data: 23/04/2017

Objetivo:

Compreender a metodologia usada na síntese, implementação e programação de um processador básico, constituído pelos 5 estágios convencionais de um MIPS: Instruction Fetch (IF), Instruction Decode (ID), Execute (EX), Memory (MEM) e Write-Back (WB).

Descrição breve do processador:

Este processador é um processador de 32 bits constituído por 5 estágios:

- IF (Instruction Fetch) – Leitura da instrução indicada pelo PC (Program Counter) da memória de instruções;
- ID (Instruction Decode) – Descodificação da instrução e leitura dos operandos;
- EX (Execute) – Execução da instrução, i.e., cálculo do resultado (Unidade Funcional) ou alteração do fluxo de instruções (teste de condição e realização de um salto na Unidade de Controlo de Salto);
- MEM (Memory) – Leitura ou escrita de um valor na memória de dados;
- WB (Write-Back) – Escrita do resultado da instrução na Unidade de Armazenamento (Register File).

4.1.

a)

No estágio de ID, o bloco InstructionDecoder recebe o sinal Instruction(31:0) do componente InstructionFetch e gera todos os sinais de dados e de controlo.

Dentro do InstructionDecoder, existe uma memória (decode_memory) cujo objetivo é descodificar o sinal correspondente ao Opcode (Instruction(31:26)).

```
-- Assign memory outputs
PL    <= mem_out(30);
dDA   <= mem_out(21 downto 18);
dAA   <= mem_out(29 downto 26);
dBA   <= mem_out(25 downto 22);
MASel <= mem_out(10 downto 9);
MBSel <= mem_out( 8 downto 7);
FS    <= mem_out(17 downto 14);
KNSSel<= mem_out(13 downto 11);
MMA   <= mem_out( 6 downto 5);
MMB   <= mem_out( 4 downto 3);
MW    <= mem_out( 2);
MDSel <= mem_out( 1 downto 0);
```

As saídas do decode_memory e as respetivas funções estão apresentadas na tabela seguinte:

Saída	Função
PL	Indica se a instrução a ser executada é de controlo (PL=1) ou de dados (PL=0).
FS(3:0)	Seleciona a operação a ser executada pelo bloco Execute
MMA(1:0)	Seleciona um de quatro sinais para endreçar a memória (bloco Memory)
MMB(1:0)	Seleciona um de quatro sinais a ser escrito na memória (bloco Memory)

MW	Ativa a escrita na memória (bloco Memory)
MDSel(1)	Seleciona o sinal que irá corresponder à saida DA - DR (= Instruction(25:22)) quando MDSel(1) ='0' ou dDA (gerado a partir do opcode pelo bloco decode_memory) quando MDSel(1) = '1'. A saída DA terá a função de selecionar o registo de destino no RegisterFile
MDSel(0)	correponde à saída MD do InstructionDecode
MASel(1)	Seleciona o sinal que irá corresponder à saida AA - SA (= Instruction(21:18)) quando MASel(1) ='0' ou dAA (gerado a partir do opcode pelo bloco decode_memory) quando MASel(1) = '1'. A saída AA terá a função de selecionar o registo de onde será retirado o operando A no RegisterFile.
MASel(0)	Correponde à saída MA, que escolhe entre o operando A (quando MASel(0) = '0') e o sinal da constante KNS (quando MASel(0) = '1') para entrar na FunctionalUnit do bloco Execute
MBSel(1)	Seleciona o sinal que irá corresponder à saida BA - SB (= Instruction(17:14)) quando MBSel(1) ='0' ou dBA (gerado a partir do opcode pelo bloco decode_memory) quando MBSel(1) = '1'. A saída BA terá a função de selecionar o registo de onde será retirado o operando B no RegisterFile.
MBSel(0)	Correponde à saída MB, que escolhe entre o operando B (quando MBSel(0) = '0') e o sinal da constante KNS (quando MBSel(0) = '1') para entrar na FunctionalUnit do bloco Execute
KNSSel	Decide qual será o valor do sinal KNS conforme o seguinte código:

```
-- Constant value (KNS) is always extended to 32 bits, depending on KNSSel
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;
```

b)

OpCode	Mnem	PL	dAA	dBA	dDA	FS	KNSSel	MASel	MBSel	MMA	MMB	MW	MDSEL
000010	SUB	0	X	X	X	0011	X	00	00	X	X	0	00
000011	SUBI	0	X	X	X	0011	001	00	X1	X	X	0	00
000000	ADD	0	X	X	X	0000	X	00	00	X	X	0	00
010000	ROL	0	X	X	X	1110	X	XX	00	X	X	0	00
010111	B	1	X	X	0000	0011	011	00	00	X	X	0	10
010111	B.EQ	1	X	X	0000	0011	011	00	00	X	X	0	10
010111	B.NEQ	1	X	X	0000	0011	011	00	00	X	X	0	10

Justificações:

As três ultimas operações (B, B.EQ e B.NEQ) partilham o mesmo Opcode, logo, todos os sinais de controlo gerados pelo decode_memory têm necessariamente de ser iguais para os três.

O sinal PL é zero nas cinco primeiras operações, pois são instruções de dados e o PC apenas deve ser incrementado e é 1 nas três ultimas operações, que são operações de salto. Os sinais dAA e dBB são insignificantes, pois estes nunca serão tidos em conta, uma vez que em todas as as operações os sinais MASel(1) e MBSel(1) irão estar a 0 (excepto na operação ROL, que não necessita de operando A e a operação SUBI, cujo operando B corresponde a uma constante). O sinal dDA também é irrelevante nas cinco primeiras operações porque MDSel(1) é zero, mas nas operações de salto tem importância porque não queremos que o resultado da operação seja escrito num registo, logo forçamos a escrita no registo 0, cujo valor não pode ser alterado. Nas cinco primeiras operações, FS corresponde à operação efetuada e nas operações de salto é 0011 (subtração) porque desta forma podemos saber se um operando é maior, igual ou menor que outro. Os valores de KNSSel foram atribuídos consultado a tabela acima. MASel(0) e MBSel(0) são sempre 0 pois as saídas A e B do RegisterFile são sempre escolhidas como operandos para a FuncionalUnit, excepto no ROL que não precisa de operando A e do SUBI que utiliza o sinal KNS como operando A. Os valores de MMA e MMB são insignificantes e o valor de MW está sempre a 0 porque em nenhuma operação é efetuada a escrita na memória. O sinal MDSel é sempre "00", porque o resultado é sempre guardado no registo de destino DR, excepto nas operações de salto, em que o resultado não é guardado.

4.2.

Para construirmos a lógica necessária à implementação da unidade de controlo de salto, começámos por analisar o schematic e chegámos à conclusão de que no bloco InstructionFetch existe um multiplexer cuja entrada de seleção é a saída **PC_Load** da UCS e a saída deste tomava os valores de **PCValue** quando PC_Load = '1' ou PC + 1 quando PC_Load = '0'. Como isto acontece, só temos que nos preocupar em construir a lógica para o sinal **PC_Load**, em função dos sinais BC, PL e das flags, garantindo que este toma o valor '1' quando é suposto existir salto e o valor '0' quando isto não acontece.

Como BC(3) é sempre '0' quando existe salto e PL é sempre '1', chegámos à conclusão que o sinal PC_Load seria dado pela seguinte expressão:

$$\mathbf{PC\_Load} = \overline{\mathbf{BC(3)}} \,.\, \mathbf{PL} \,.\, \mathbf{BCout}$$

em que o sinal BCout toma os valores indicados na tabela seguinte, em função de BC(2:0):

Tipo de salto	PL	BC	BCout[(1)]	PC_Load
Não há salto	0	X	X	0
B	1	000X	1	1
B.EQ / BI.EQ	1	0010	Z	1
B.NE / BI.NE	1	0011	$\overline{\mathrm{Z}}$	1
B.GT / BI.GT	1	0100	$\overline{\mathrm{N}} \,.\, \overline{\mathrm{Z}} = \mathrm{P}$	1
B.GE / BI.GE	1	0101	$\overline{\mathrm{N}}$	1
B.LT / BI.LT	1	0110	N	1
B.LE / BI.LE	1	0111	$\mathrm{N} + \mathrm{Z} = \overline{\mathrm{P}}$	1

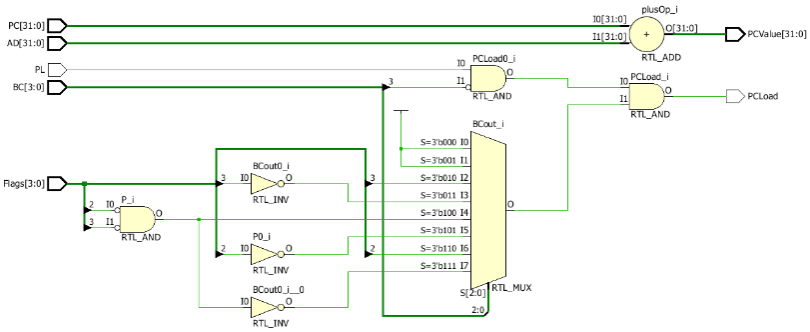
(1) Não existe salto se BCout ≠ '1'

Uma vez que a operação aritmética analisada nas operações de salto é A – B, então foi bastante fácil chegar à conclusão de quais seriam as flags (ou complementos) a serem analisadas em cada situação: por exemplo, para o salto B.GE, existe salto se o operando A for maior ou igual que o operando B, isto é, se A-B for um número negativo, logo a flag a analisar será N.

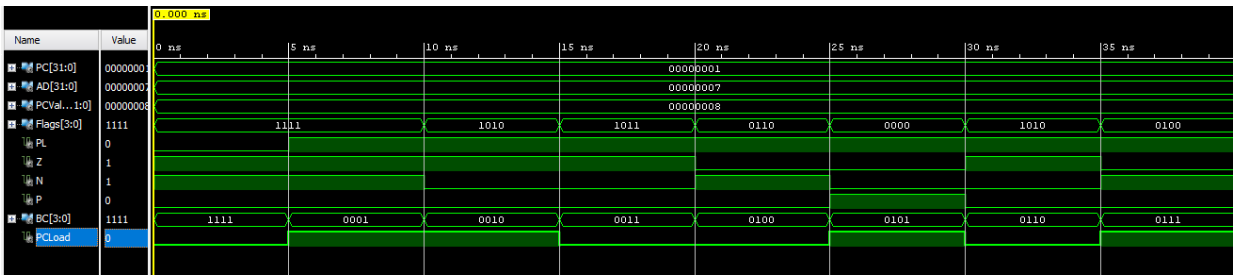
De seguida apresentam-se as alterações feitas ao branchcontrol.vhd e o respetivo schematic:

```
with BC(2 downto 0) select
BCout<= '1' when "000" | "001",
        Z when "010",
        not(Z) when "011",
        P when "100",
        not(N) when "101",
        N when "110",
        not(P) when "111",
        'X' when others;

PCLoad<= PL and BCout and not(BC(3));
PCValue<= PC + AD;
```



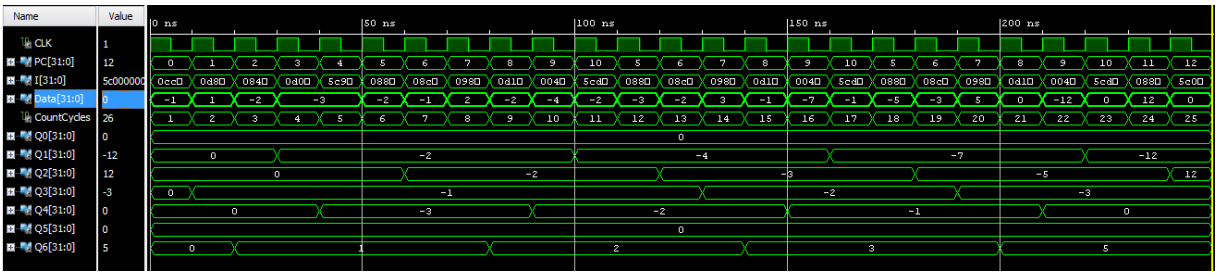
Na imagem seguinte encontram-se oito simulações deste circuito, com as flags escolhidas aleatoriamente: na primeira, como PL=’1’, PCLoad será ‘0’; na segunda BC = “0001” é um salto incondicional, logo PCLoad será ‘1’; na terceira BC = “0010” é um B.EQ e Z = 1, logo existirá salto; na quarta BC = “0011” é um B.NE e Z = 1, logo não existe salto; na quinta BC = “0100” é um B.GT e P = 0, logo não existe salto; na sexta BC = “0101” é um B.GE e N = 0, logo existe salto; na sétima BC = “0110” é um B.LT e N = 0, logo não existe salto e na oitava BC = “0111” é um B.LE e P = 0, logo existe salto.



Concluímos então que o circuito foi bem concebido, uma vez que a simulação deu os resultados esperados.

4.3.

A simulação obtida foi a seguinte:



Os resultados da simulação foram confirmados e são iguais aos resultados esperados, o que significa que o circuito executou as operações corretamente.

O número de ciclos necessários à correta execução do troço de código indicado até à execução (inclusive) da instrução "B #0" é 25.

Ciclo	PC	Instrução	Operação
1	0	SUBI R3,R0,#1	R3 = R0 – 1 = 0 – 1 = -1
2	1	SUBI R6,R0,#-1	R6 = R0 – (-1) = 0 + 1 = 1
3	2	SUB R1,R3,R6	R1 = R3 – R6 = -1 – 1 = -2
4	3	SUBI R4,R0,#3	R4 = R0 – 3 = 0 – 3 = -3
5	4	B.EQ R4,R0,#7	PC ← PC+7 se R4=R0; R4=-3≠0=R0, logo PC ← PC+1
6	5	SUB R2,R3,R6	R2 = R3 – R6 = -1 – 1= -2
7	6	SUB R3,R0,R6	R3 = R0 – R6 = 0 – 1 = -1
8	7	SUB R6,R0,R2	R6 = R0 – R2 = 0 – (-2) = 2
9	8	SUBI R4,R4,#-1	R4 = R4 – (-1) = -3 – (-1) = -2
10	9	ADD R1,R1,R2	R1 = R1 + R2 = -2 + (-2) = -4
11	10	B.NE R4,R0,#-5	PC ← PC-5 se R4≠R0; R4=-2≠0=R0, logo PC ← PC-5
12	5	SUB R2,R3,R6	R2 = R3 – R6 = -1 – 2= -3
13	6	SUB R3,R0,R6	R3 = R0 – R6 = 0 – 2 = -2
14	7	SUB R6,R0,R2	R6 = R0 – R2 = 0 – (-3) = 3
15	8	SUBI R4,R4,#-1	R4 = R4 – (-1) = -2 – (-1) = -1
16	9	ADD R1,R1,R2	R1 = R1 + R2 = -4 + (-3) = -7
17	10	B.NE R4,R0,#-5	PC ← PC-5 se R4≠R0; R4=-1≠0=R0, logo PC ← PC-5
18	5	SUB R2,R3,R6	R2 = R3 – R6 = -2 – 3 = -5
19	6	SUB R3,R0,R6	R3 = R0 – R6 = 0 – 3 = -3
20	7	SUB R6,R0,R2	R6 = R0 – R2 = 0 – (-5) = 5
21	8	SUBI R4,R4,#-1	R4 = R4 – (-1) = -1 – (-1) = 0
22	9	ADD R1,R1,R2	R1 = R1 + R2 = -7 + (-5) = -12
23	10	B.NE R4,R0,#-5	PC ← PC-5 se R4≠R0; R4=0=R0, logo PC ← PC+1
24	11	SUB R2,R0,R1	R2 = R0 – R1 = 0 – (-12) = 12
25	12	B #0	PC ← PC + 0

5.1.

A frequência máxima de um circuito é calculada da seguinte forma:

$f_{max} = \frac{1}{T_{min}}$, em que T_{min} é igual ao tempo de propagação mínimo do circuito, tendo em conta que todas as operações devem funcionar corretamente. Deste modo:

T_{min} = máx{ IF(PC →Adder→MUX→Write on PC Register) ;

IF(Read PC from register→Memory→Instruction) + ID(Instruction→Decoder→RF→A,B) + EX(A,B→UF→Data) + WB(Data→Write to register file) ;

IF(Read PC from register→Memory→Instruction) + ID(Instruction→Decoder→RF→A,B) + EX(A,B→UF→Data) + MEM(A,B,D→Read from Memory) + WB(Data→Write to register file) ;

IF(Read PC from register→Memory→Instruction) + ID(Instruction→Decoder→RF→A,B) + EX(A,B→UF→Data) + MEM(A,B,D→Write to Memory) ;

IF(Read PC from register→Memory→Instruction) + ID(Instruction→Decoder→RF→A,B) +

EX(A,B→UF→Branch Control→PCLoadEnable,PCLoadValue→MUX→Write on PC Register) }

= máx{15; 35 + 30 + 30 + 15; 35 + 30 + 30 + 40 + 15; 35 + 30 + 30 + 30; 35 + 30 + 40} = 150ns

Logo, $f_{máx} = 1/(150*10^{-9}) = 6{,}6$ MHz

5.2.

Neste caso, o T_{min} será igual ao tempo de propagação do estágio com maior tempo de propagação mais os tempos de setup e propagação dos flip-flops dos registos pipeline:

$T_{min} = T_{Propagação}(FF) + T_{Propagação}(\text{Estágio mais demorado}) + T_{Setup}(FF) = T_{Propagação}(FF) + T_{Propagação}(\text{MEM(A,B,D→Read from Memory)}) + T_{Setup}(FF) = 1 + 40 + 1 = 42$ ns

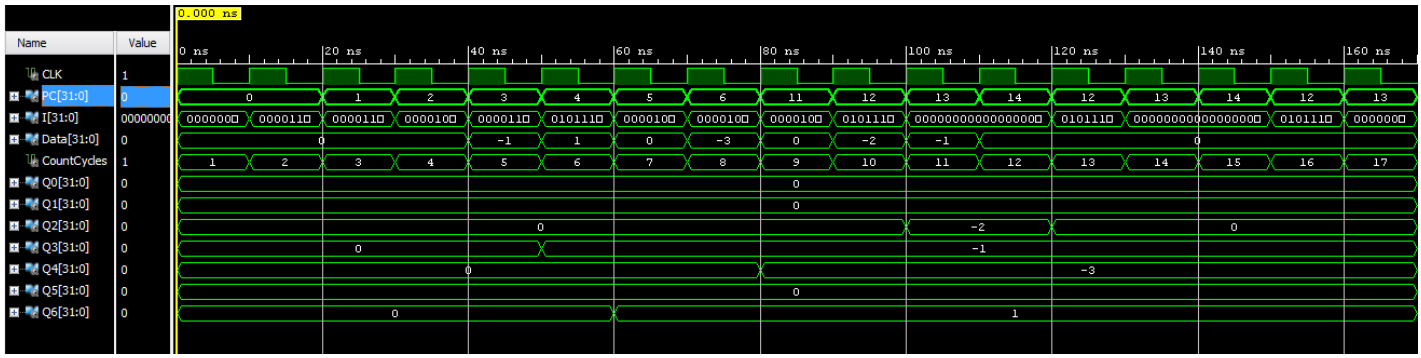
Logo, $f_{máx} = 1/(42*10^{-9}) = 23{,}8$ MHz

O aumento teórico de desempenho é então dado por:

$Speedup = T_{Ciclo\ Único} / T_{Pipelined} = 150/42 = 3{,}6$

5.3.1.

Na imagem seguinte encontra-se a simulação do troço de código da alínea 4.3 mas desta vez executado na arquitetura pipeline:



No seguinte esquema encontram-se os erros de execução que detetámos:

0. SUBI R3,R0,#1
1. SUBI R6,R0,#-1
2. SUB R1,R3,R6 — **Data Hazard (1), Data Hazard (2)**
3. SUBI R4,R0,#3
4. B.EQ R4,R0,#7 — **Data Hazard (3), Control Hazard (1)**
5. SUB R2,R3,R6
6. SUB R3,R0,R6
7. SUB R6,R0,R2 — **Data Hazard (4)**
8. SUBI R4,R4,#-1
9. ADD R1,R1,R2
10. B.NE R4,R0,#-5 — **Data Hazard (5), Control Hazard (2)**
11. SUB R2,R0,R1
12. B #0

IF	ID	EX	MEM	WB	Valor disponível
	IF	ID	EX	MEM	WB
		IF	ID	EX	MEM
			IF	ID	EX
				IF	ID

Como existem 5 estágios nesta arquitetura em *pipeline*, IF, ID, EX, MEM e WB e o valor de escrita nos registos em cada instrução só vai estar disponível a seguir ao último estágio, *Write Back* e os registos são lidos no segundo estágio, *Instruction Decode*, são necessários 3 ciclos de relógio de intervalo entre uma instrução que escreve num registo e uma instrução que lê esse mesmo registo, como mostra a tabela acima. Portanto os erros na execução provêm da falta deste requerimento.

Ao analisar as instruções do troço de código, reparámos na existência de cinco *Data Hazards* (*DH*) e de dois *Control Hazards* (*CH*). Na instrução 2, o valor de R3 que será utilizado

para realizar a operação não é o mesmo que foi escrito na instrução 0, há um *DH.* Ainda na instrução 2, vai precisar-se também do valor de R6 calculado na anterior, portanto temos mais um *DH*. Na instrução 4 será necessário ler o valor de R4 da instrução anterior, portanto estamos na presença de outro *DH*. Ainda na instrução 4, se houver salto, que é avaliado no estágio *EX*, haverá já duas instruções a serem executadas, neste caso as instruções 5 e 6, o que não é pretendido. Estamos então na presença de um *CH*. Caso não haja salto, encontramos mais dois *DH*: na instrução 7, vai ser lido o registo R2 cujo valor teria sido escrito na instrução 5 e na instrução 10 irá ser lido o registo R4, cujo valor foi escrito na instrução 8. Por fim, como existe um salto condicional na instrução 10, estamos perante outro *CH.*

5.3.2.

Para garantirmos a correta execução do troço de código, decidimos resolver o problema por software, adicionando instruções NOP e trocando a ordem de instruções de modo a que não existissem mais conflitos de dados nem de controlo.

```
0. SUBI  R4,R0,#3        11. SUB   R6,R0,R2
1. SUBI  R3,R0,#1        12. ADD   R1,R1,R2
2. SUBI  R6,R0,#-1       13. B.NE  R4,R0,#-6
3. NOP                   14. NOP
4. B.EQ  R4,R0,#12       15. NOP
5. NOP                   16. SUB   R2,R0,R1
6. SUB   R1,R3,R6        17. B     #0
7. SUB   R2,R3,R6        18. NOP
8. SUBI  R4,R4,#-1       19. NOP
9. SUB   R3,R0,R6
10. NOP
```

Quanto aos conflitos de controlo, uma vez que o processador pipeline apenas sabe se existe ou não salto quando as instruções de salto chegam ao estágio EXE (que corresponde ao 3º estágio), então as duas instruções subsequentes à instrução de salto irão ser executadas de qualquer forma. Para evitar que sejam executadas instruções que não são suposto serem executadas, a solução é inserir instruções NOP imediatamente a seguir a cada instrução de salto ou mover instruções que seriam executadas antes do salto para uma das duas posições a seguir ao salto. Deste modo, para resolver o conflito de dados (1), adicionámos o NOP 5 e movemos a instrução SUB R1,R3,R6 para depois do salto e para resolver o conflito de dados (2) adicionámos os NOP 14 e 15. Por fim, resolvemos também adicionar dois NOP ao fim do código, pois, na prática, existiriam outras instruções que não deveriam ser executadas caso o último salto seja tomado.

Quanto aos conflitos de dados, uma vez que o processador pipeline tem 5 estágios e os dados são escritos nos registos no 5º estágio (Write Back) e lidos no 2º estágio (Instruction Decode), para que não existam conflitos de dados, entre uma instrução que escreva num registo e uma instrução que leia esse mesmo registo têm que existir pelo menos 3 instruções de intervalo.

Por esta razão, os Data Hazards (1) e (2), ficaram resolvidos quando movemos a instrução SUB R1,R3,R6; para resolvermos o conflito de dados (3) movemos a instrução SUBI

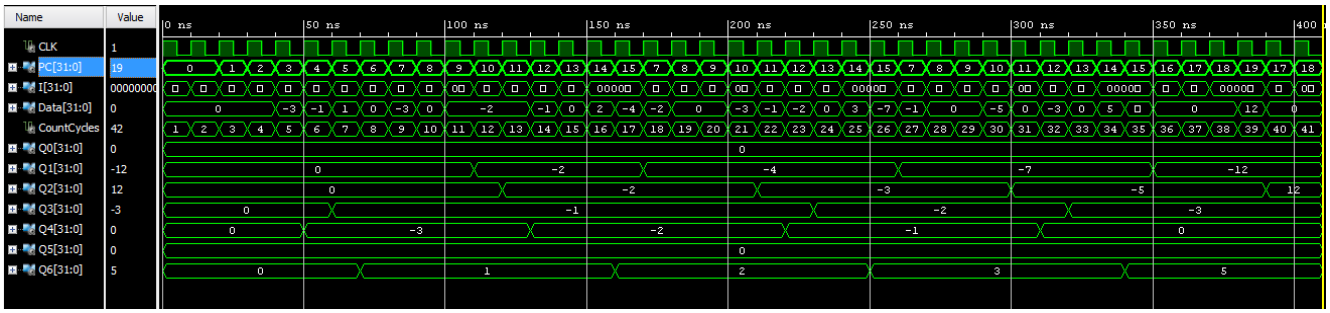
R4,R0,#3 para o início do programa; para o **(5)**, recuámos a instrução SUBI R4,R4,#-1 duas posições e, por isso, para resolver o **(4)** foi apenas necessário introduzir o NOP 10.

Por fim, bastou-nos apenas ajustar os offsets dos saltos para que as instruções de salto vão para os sítios corretos.

Deste modo, corrigimos todos os conflitos e, ao analisar o código verificamos que não são criados nenhuns outros conflitos.

5.3.3.

Ao alterarmos o conteúdo da memória para o troço de código modificado como mostrado na questão 5.3.2, obtemos a seguinte simulação:



Analisando a simulação, verificámos que a instrução B #0 (PC = 17) entra no pipeline no 37º ciclo, logo, como o pipeline tem 5 estágios, só irá sair no 41º ciclo. Por esta razão, o número total de ciclos de relógio que o programa demora a ser executado na arquitetura pipeline é 41. Para determinarmos o aumento real de desempenho basta aplicarmos a seguinte fórmula (C – número de ciclos para processar a sequencia de operações):

$$\text{Speedup real} = \frac{C_{\text{Ciclo Único}} \times (T_{CLK})_{\text{Ciclo Único}}}{C_{\text{Pipeline}} \times (T_{CLK})_{\text{Pipeline}}} = \frac{25 \times 150}{41 \times 42} = 2{,}18$$

Conclusões:

Comparando os valores do Speedup teórico (3,6) e do Speedup real (2,18) verificamos que, como seria de esperar, o speedup real é menor que o teórico, uma vez que o speedup teórico admite que o número de instruções a executar na arquitetura de ciclo único e na arquitetura pipeline é igual. Ora, na prática, isto não é verdade, uma vez que foi necessário adicionar instruções NOP para resolver os conflitos de dados e de controlo, o que explica a razão de o speedup real ser mais baixo que o teórico