UNIVERSIDADE FEDERAL DE SÃO CARLOS - CAMPUS SOROCABA

LABORATÓRIO DE ARQUITETURA DE COMPUTADORES

PROJETO FINAL

(Microprocessador - MIPS R2000)

Professora Dra. Yeda Regina Venturini

GRUPO: **1** TURMA: **A**

Antônio Jorge Medeiros **RA:** 620521
Daniel Souza Bertoldi **RA**: 620548
Guilherme Quintal Gonçalves **RA:** 620386
Lucas Buzzo **RA:** 620408

SUMÁRIO

1. Introdução

O projeto do microprocessador consiste em implementar um MIPS simplificado, utilizando-se da linguagem VHDL. As instruções implementadas para o microprocessador são: Add, Addi, Sub, Lw, Sw, Beq, Bne, Jal, J, Slt, And, Or, Jr, Sll e Srl.

MIPS R2000 é um microprocessador de arquitetura RISC criado em 1986 com 5 estágios pipeline e 450.000 transistores. Este foi o primeiro microprocessador RISC comercial. O conceito básico era aumentar muito o desempenho com o uso de pipelines profundos para as instruções, uma técnica que, embora boa, viria a revelar-se difícil de ser produzida. O MIPS possui instruções de tamanho fixo de 32 bits e 32 registradores de propósito geral de 32 bits cada um. O registrador 0 sempre possui valor zero. O tamanho da palavra de memória é de 32 bits.

2. Descrição do Projeto

Neste documento está contido a documentação Projeto Final da disciplina Laboratório de Arquitetura de Computadores, onde será mostrado os MIFs e as Waves dos seguintes formatos de instruções: J_format (J e Jal), R_Format (Jr, Sll e Srl) e I_Format (Lw, Sw, Addi e Bne).

Para efetuar os testes das instruções citadas acima, iremos executar um código em Assembly que procura o maior e menor elemento dentro de um vetor.

3. Instruções

3.1. J_Format

- **Instrução: Jump**

A instrução Jump deve fazer um salto incondicional para qualquer região da memória de instruções.

Para fazer o teste da instrução, criamos um código que realiza diversos pulos entre vários endereços de memória, como mostrado abaixo, retornando para o endereço de memória inicial, criando um loop.

A imagem a seguir representa o **Jump.mif**

```
-- Place MIPS Instructions here
-- Note: memory addresses are in words and not bytes
-- i.e. next location is +1 and not +4
    00: 08000004;   -- j 04          <- pula para a instrucao 04
    01: 08000003;   -- j 03          <- pula para a instrucao 03
    02: 00000000;   -- nop           <- nop
    03: 08000005;   -- j 05          <- pula para a instrucao 05
    04: 08000001;   -- j 01          <- pula para a instrucao 01
    05: 08000000;   -- j 00          <- volta ao inicio do programa
    [06..FF]: 00000000; -- nop (sll r0,r0,0)
```

A imagem a seguir representa as Waves do **J.**

Como podemos observar, o jump está funcionando perfeitamente. Como todas as instruções (exceto a instrução no endereço 02, que contém um nop) são de Jump, a partir da primeira instrução o Jump sobe para nível lógico alto e assim permanece durante toda a execução do programa. A instrução contida no endereço 00 faz um jump para o endereço 04, que faz um jump para o endereço 01, que pula para o endereço 03, que pula para a

instrução 05, que finalmente, faz jump para a instrução 00, concluindo o loop. Em nenhum momento o endereço 02 é lido.

- ## Instrução: Jal

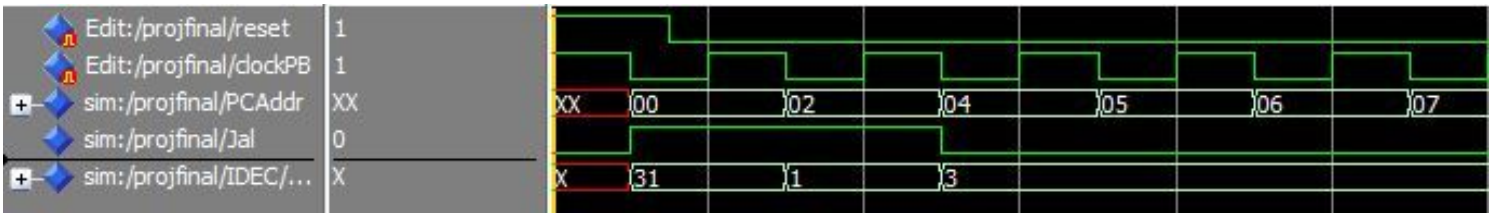
O comando Jal deve desviar a execução de PC, o Registrador Contador de Programa, para uma posição onde uma sub-rotina se encontra.

Para a realização do teste do Jal, criamos um breve programa que faz um jump-and-link duas vezes seguidas, portanto o valor contido no registrador 31 deve atualizar o endereço contido nele duas vezes.

A imagem a seguir, representa o **Jal.mif**

```
-- Place MIPS Instructions here
-- Note: memory addresses are in words and not bytes
-- i.e. next location is +1 and not +4
    00: 0C000002;   -- jal 02   <- pula para a instrucao 02 e salva a instrucao "8C020000" no registrador 32.
    01: 8C020000;   -- lw $2,0 ;memory(00)=55
    02: 0C000004;   -- jal 04   <- pula para a instrucao 04 e salva a instrucao "8C030001" no registrador 32.
    03: 8C030001;   -- lw $3,1 ;memory(01)=AA
    [04..FF]: 00000000; -- nop (sll r0,r0,0)
```

A imagem a seguir representa as Waves do **Jal.**



A execução da instrução está correta. Como podemos ver, no endereço 00 há um jal para o endereço 02, o PC recebe o endereço 02, e o registrador 31 recebe o valor da próxima instrução depois do jal antes do pulo, ou seja, o endereço 01. A instrução do endereço 02, faz um jal para o endereço 04, e o registrador 31 recebe o endereço 03.

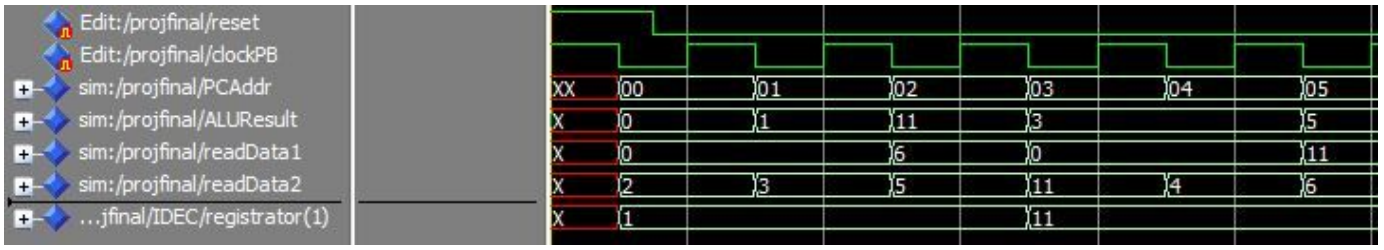
3.2. R_Format

- **Instrução: Add**

A instrução Add consiste em somar o conteúdo em readData1 (rs) com o conteúdo em readData2 (rt), e armazenar no registrador destino. Para testar esta função, utilizamos o seguinte código em Assembly:

```
00: 8C020000;   -- lw $2,0 ;memory(00)=6
01: 8C030001;   -- lw $3,1 ;memory(01)=5
02: 00430820;   -- add $1,$2,$3
03: AC010003;   -- sw $1,3 ;memory(03)=11
04: 8C040003;   -- lw $4,3 ;memory(03)=11
05: 1022FFFF;   -- beq $1,$2,-4
06: 1021FFF9;   -- beq $1,$1,-28
-- Use NOPS for default instruction memory values
[07..FF]: 00000000; -- nop (sll r0,r0,0)
```

A imagem a seguir representa as Waves do **Add.**



Como podemos ver a partir do endereço 02, ALUResult recebe o resultado do readData1, que é 6, com a soma do readData2, que é 5, ou seja, ALUResult fica com valor 11. Ao chegar no endereço 03, o registrador 1 é atualizado com o resultado da soma.

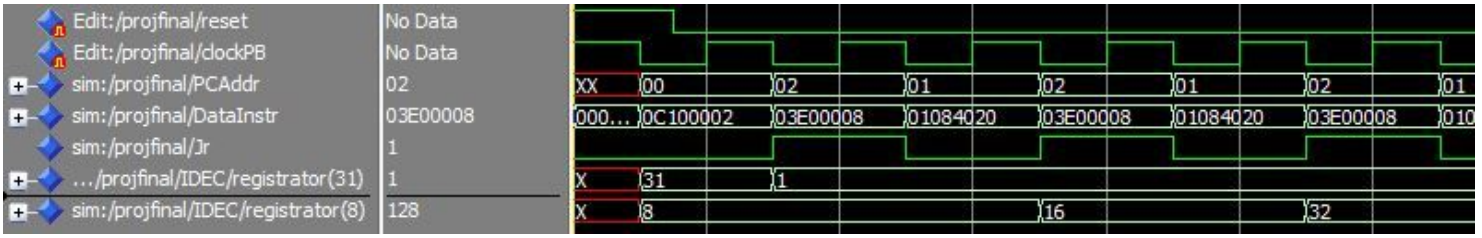
- **Instrução: Jr**

A operação do tipo Jr é necessária quando desejamos retomar a execução do program.mif para o endereço posterior, contido no registrador $ra (registrador 31), após ocorrer um Jal. Para testar a execução desta instrução fazemos primeiro um Jal para o endereço 02, que por sua vez faz um Jr para o endereço contido no registrador $ra, que faz uma soma e retorna ao endereço 02, criando um laço.

A imagem a seguir representa o **Jr.mif**

```
-- MIPS Instruction Memory Initialization File
Depth = 256;
Width = 32;
Address_radix = HEX;
Data_radix = HEX;
Content
Begin
-- Place MIPS Instructions here
-- Note: memory addresses are in words and not bytes
-- i.e. next location is +1 and not +4
        00: 0C100002;   -- jal 02
    01: 01084020;   -- add $t0, $t0, $t0
    02: 03E00008;   -- jr $ra
        03: 08100000;   -- j 00
    [04..FF] : 00000000;
End;
```

A figura a seguir representa as Waves da instrução **Jr**



A primeira instrução do código faz um Jal para a o endereço 02, consequentemente salvando o endereço 01 no registrador 31, podemos ver que o endereço armazenado no registrador 31 foi atualizado, já que após a execução da instrução ele passa a ter valor 1. O endereço 02 pula para o endereço armazenado no registrador 31, ou seja, ele irá pular para o endereço 01, que soma o valor contido nele. Podemos ver que tudo está funcionando corretamente já que o valor lógico da instrução Jr sobe, e o endereço do PC passa a ser 01, o valor armazenado no registrador 31. O nível lógico do Jr desce, e $to recebe o resultado da soma, o PC é incrementado, lê a instrução no endereço 02, fechando o loop.

● Instrução: SII

A operação SII faz um determinado deslocamento à esquerda dos bits de um registrador de entrada, colocando o resultado no registrador destino. Para essa instrução, criamos um laço que irá multiplicar o valor contido no registrador $t0 por 4, equivalente a deslocar dois bits à esquerda.

A imagem a seguir, representa o **SII.mif**

```
-- MIPS Instruction Memory Initialization File
Depth = 256;
Width = 32;
Address_radix = HEX;
Data_radix = HEX;
Content
Begin
-- Place MIPS Instructions here
-- Note: memory addresses are in words and not bytes
-- i.e. next location is +1 and not +4
        00: 00084080;   -- sll $t0, $to, 2
    01: 00084080;   -- sll $t0, $t0, 2
    02: 00084080;   -- sll $t0, $t0, 2
    03: 08100000;   -- j Loop
    [04..FF] : 00000000;
End;
```

Figura 3 - SII.mif

A figura a seguir, representa as Waves da instrução **SII:**

Não é difícil notar que o valor contido em $t0, o valor 8, está sendo multiplicado por 4 durante os endereços 00, 01 e 02, e, ao chegar no endereço 03, é feito um jump para o início do programa, o nível lógico do SI passa a ser baixo, e o PC retorna ao ínicio, continuando as multiplicações.

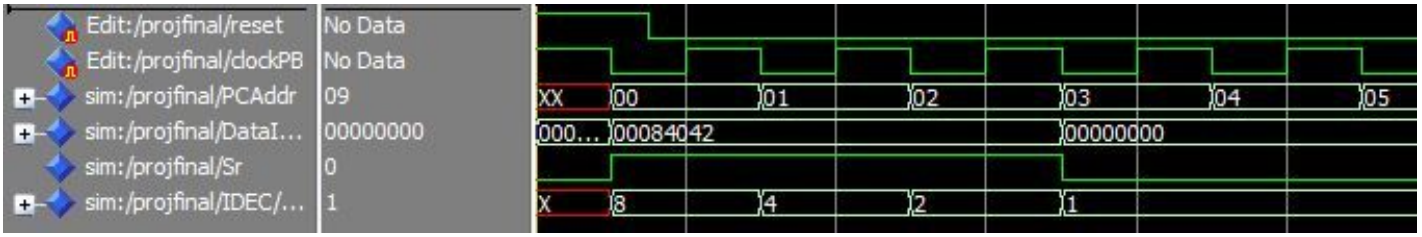
● Instrução: Srl

A instrução Srl faz um determinado deslocamento à direita dos bits de um registrador de entrada, colocando a resposta no registrador destino. A lógica do código é simples: o registrador $t0 contém o valor 8, portanto, vamos fazer um deslocamento à direita de 1 bit, ou seja, dividiremos ele por 2. Essa divisão ocorrerá nos endereços 00, 01 e 02, finalizando o programa em seguida.

A imagem a seguir, representa o **Srl.mif**

```
-- MIPS Instruction Memory Initialization File
Depth = 256;
Width = 32;
Address_radix = HEX;
Data_radix = HEX;
Content
Begin
-- Place MIPS Instructions here
-- Note: memory addresses are in words and not bytes
-- i.e. next location is +1 and not +4
        00: 00084042;   -- srl $t0, $t0, 1
    01: 00084042;   -- srl $t0, $t0, 1
        02: 00084042;   -- srl $t0, $t0, 1
    [03..FF] : 00000000;
End;
```

A figura a seguir, representa as Waves da instrução **Srl:**



Novamente, não é difícil notar que a divisão está sendo feita corretamente. A partir do endereço 00, o valor lógico de Sr passa a ser alto, a divisão por 2 é feita, $t0 passa a ter o valor 4, o mesmo ocorre nos endereços 01 e 02, apenas mudando o valor armazenado em $t0, que ao fim do programa fica com o valor 1, e o programa é finalizado a partir do endereço 03.

3.3. I_Format

- **Instrução: Load Word**

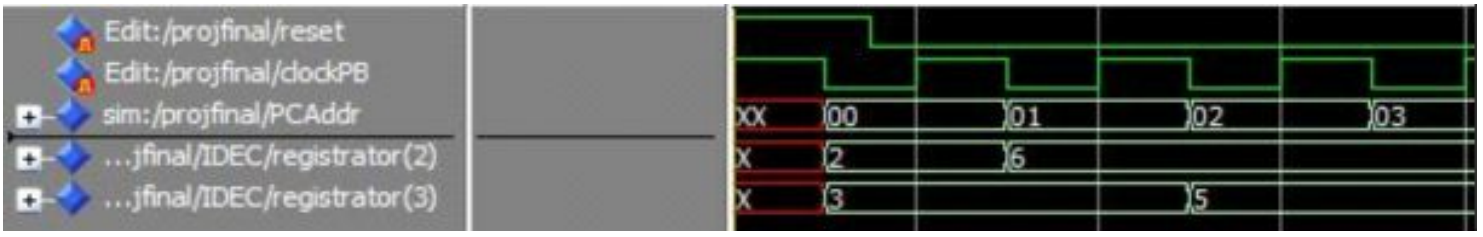
A instrução Load Word deve pegar um valor da memória e armazenar no registrador destino. Os valores de memória usados foram:

```
00: 00000006;  -- Tamanho do vetor (6)
01: 00000005;  -- Valor 1 (5)
02: 00000002;  -- Valor 2 (2)
03: 00000014;  -- Valor 3 (20)
04: FFFFFFFB;  -- Valor 4 (-5)
05: FFFFFFFE;  -- Valor 5 (-2)
06: FFFFFFEC;  -- Valor 6 (-20)
[07..FF]: 00000000;
```

Para demonstrar o funcionamento do Load Word, usamos o seguinte código:

```
00: 8C020000;   -- lw $2,0 ;memory(00)=6
01: 8C030001;   -- lw $3,1 ;memory(01)=5
02: 00430820;   -- add $1,$2,$3
03: AC010003;   -- sw $1,3 ;memory(03)=11
04: 8C040003;   -- lw $4,3 ;memory(03)=11
05: 1022FFFF;   -- beq $1,$2,-4
06: 1021FFF9;   -- beq $1,$1,-28
-- Use NOPS for default instruction memory values
[07..FF]: 00000000; -- nop (sll r0,r0,0)
```

A figura a seguir, representa as Waves da instrução **Lw:**



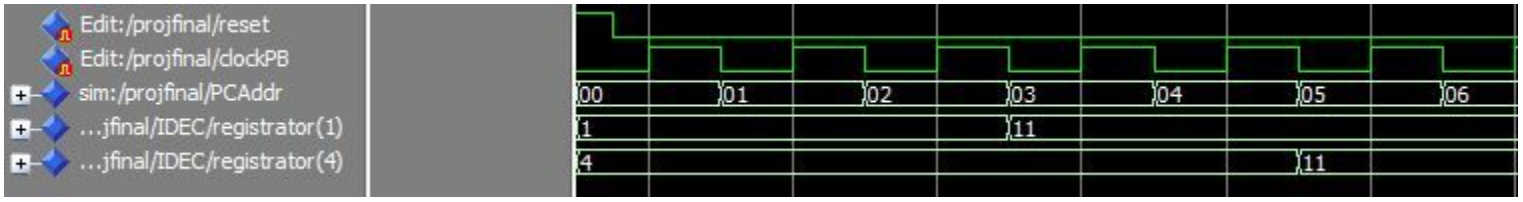
O funcionamento da instrução está correto, já que no endereço 01, o valor do registrador 2 é atualizado com o primeiro valor da memória mostrada acima, que é 6, e no endereço 02, o valor do registrador 3 é atualizado com o segundo valor da memória, que é 5.

● Instrução: Store Word

A instrução Store Word deve armazenar na memória o valor contido no registrador que foi passado à instrução. Usamos os mesmos valores na memória da instrução Load Word acima, assim como o código em Assembly:

```
00: 8C020000;   -- lw $2,0 ;memory(00)=6
01: 8C030001;   -- lw $3,1 ;memory(01)=5
02: 00430820;   -- add $1,$2,$3
03: AC010003;   -- sw $1,3 ;memory(03)=11
04: 8C040003;   -- lw $4,3 ;memory(03)=11
05: 1022FFFF;   -- beq $1,$2,-4
06: 1021FFF9;   -- beq $1,$1,-28
-- Use NOPS for default instruction memory values
[07..FF]: 00000000; -- nop (sll r0,r0,0)
```

A figura a seguir, representa as Waves da instrução **Sw**.



Após $1 receber o resultado da soma no endereço 02, no endereço 03 é feito um store word de $1 na posição 03 da memória, passando a ter valor 11. No endereço 04, é feito um load word do endereço 03 da memória no registrador $4, e podemos perceber na wave que $4 recebe o valor 11, portanto o store word está funcionando corretamente.

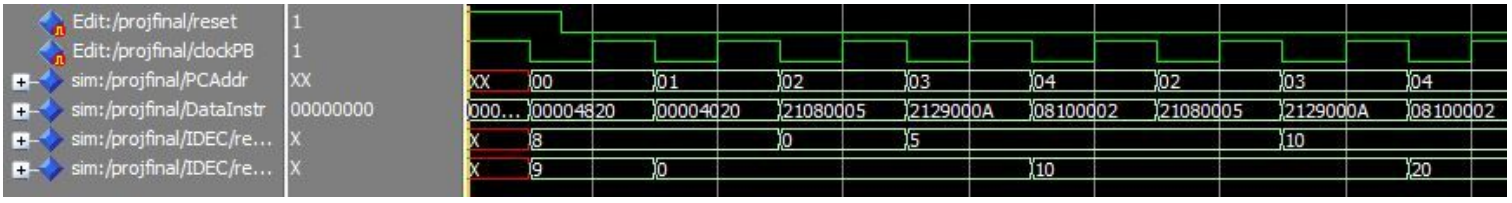
● Instrução: Addi

A instrução Addi permite calcular a soma de um registrador de entrada com um valor imediato, presente na instrução, colocando a resposta em um registrador destino.

A imagem a seguir, representa o **Addi.mif**

```
00: 00004820;   -- add t1, 0, 0
01: 00004020;   -- add t0, 0, 0
02: 21080005;   -- addi $t0, $t0, 5
03: 2129000A;   -- addi $t1, $t1, 10
04: 08100002;   -- j 02
[05..FF] : 00000000;
```

A imagem a seguir representa as Waves do **Addi.**



As instruções nos endereços 00 e 01 inicializam os registradores $9 e $8 como 0, e então, a partir do endereço 02, o registrador $8 será somado seu valor + 5, e o registrador $9 será somado seu valor + A (10 em decimal). Pela wave dá para verificar que cada registrador está recebendo seu valor correspondente. A instrução contida no endereço 04 é um jump que retorna ao endereço 02, fechando o loop.

- ## Instrução: Beq

A instrução beq faz um pulo condicional para outro endereço de memória, se os valores contidos em dois registradores forem iguais. Para testá-lo, usamos o seguinte programa:

```
00: 8C020000;   -- lw $2,0 ;memory(00)=6
01: 8C030001;   -- lw $3,1 ;memory(01)=5
02: 00430820;   -- add $1,$2,$3
03: AC010003;   -- sw $1,3 ;memory(03)=11
04: 8C040003;   -- lw $4,3 ;memory(03)=11
05: 1022FFFF;   -- beq $1,$2,-4
06: 1021FFF9;   -- beq $1,$1,-28
-- Use NOPS for default instruction memory values
[07..FF]: 00000000; -- nop (sll r0,r0,0)
```

A figura a seguir, representa as Waves do **Beq**.

Zero é a variável responsável por averiguar se os valores em ambos registradores são iguais, ela recebe 1 se os valores forem iguais, ou 0 se não forem. Como podemos ver no endereço 05, os valores nos registradores $1 e $2 são diferentes, portanto Zero permanece nível lógico baixo, e o pulo não é realizado. Já na próxima instrução, como os registradores comparados são os mesmos, Zero fica nível lógico alto e o pulo é feito para 7 instruções atrás, retornando ao início e fechando o loop.

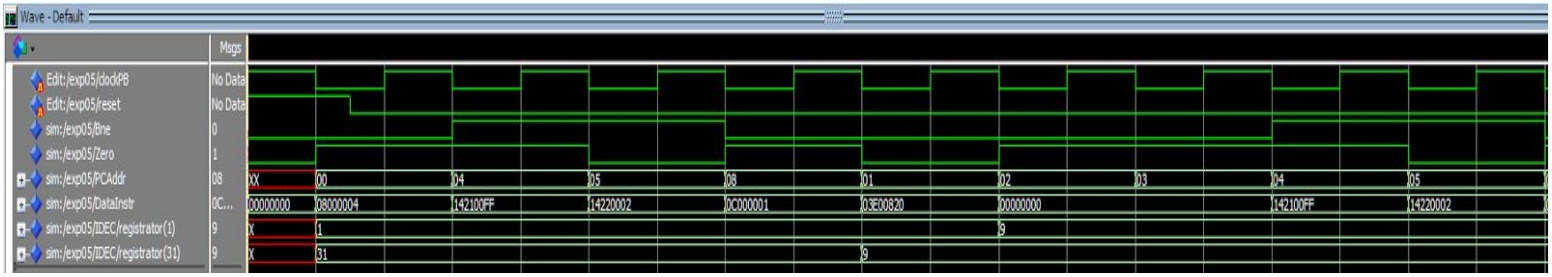
● Instrução: Bne

A operação do tipo Bne compara o conteúdo de 2 registradores, passados como parâmetro e, se o conteúdo dos dois não forem iguais, ocorre um desvio condicional para alguma região da memória de instruções. Finalmente, para demonstrarmos o funcionamento correto desta instrução, criamos um código que age desta forma: carregamos o que está na posição 00 da memória em $v0, depois carregamos da memória o que estiver na posição 01 e salvamos em $v1. Logo após, conferimos se $at é igual a $at, levando em conta a lógica do Bne, não deve ser feito o pulo pois os registradores comparados são os mesmos, e portanto possuem o mesmo valor. No endereço 03, o valor contido em $at e $v1 são diferentes, portanto deve ser feito um pulo para o início do programa, criando um loop.

A imagem a seguir, representa o **Bne.mif**

```
00:     08000004; -- j 04
01:     03E00820; -- add $31, $31, $zero
02:     00000000; -- nop
03:     00000000; -- nop
04:     142100FF; -- bne $1, $1, PC_Next + 0xFF
05:     14220002; -- bne $1, $2, PC_Next + 0x02
06:     00000000; -- nop
07:     00000000; -- nop
08:     0C000001; -- jal 01
[09..FF]: 00000000; -- nop
```

A imagem a seguir representa as Waves do **Bne.**



A primeira instrução faz um jump para o endereço 04, que faz um bne com dois registradores idênticos. Nessa hora podemos ver que, por estar comparando o mesmo registrador, o Zero fica nível lógico alto, indicando que são iguais, e o pulo não é realizado, já que a condição (Bne = ‘1’ AND Zero = ‘0’) não é satisfeita. No endereço seguinte, 05, é realizado uma comparação entre os registradores $1 e $2, como o conteúdo de ambos são diferentes, Zero fica nível lógico baixo e o jump para duas instruções a frente é feito, chegando ao endereço 08.

4. Demonstração com Vetor

Criamos um programa que irá ler os valores de um vetor e:

1 - Armazenar o maior elemento
2 - Armazenar o menor elemento
3 - Armazenar a soma de todos os valores do vetor

```
00: 8C0F0000;  -- lw $t7, 0($zero)
01: 11E00011;  -- beq $t7, $zero, Fim
02: 8C0E0001;  -- lw $t6, 4($zero)
03: 000E3820;  -- add $a3, $zero, $t6
04: 000E3020;  -- add $a2, $zero, $t6
05: 000E2820;  -- add $a1, $zero, $t6
06: 200D0001;  -- addi $t5, $zero, 4
    -- Loop:
07: 11AF000A;  -- beq $t5, $t7, Fim
08: 8DAE0001;  -- lw $t6, 4($t5)
09: 00EE402A;  -- slt $t0, $a3, $t6
0A: 11000001;  -- beq $t0, $zero, MaiorOk
0B: 000E3820;  -- add $a3, $zero, $t6
    -- NãoÉMaior:
0C: 01C6402A;  -- slt $t0, $t6, $a2
0D: 11000001;  -- beq $t0, $zero, MenorOk
0E: 000E3020;  -- add $a2, $zero, $t6
    -- NãoÉMenor:
0F: 00AE2820;  -- add $a1, $a1, $t6
10: 21AD0001;  -- addi $t5, $t5, 4
11: 08100007;  -- j Loop
    -- Fim:
12: 00000000;  -- nop
[13..FF] : 00000000;
```

Para um melhor compreendimento, do código, a imagem a seguir representa sucintamente o intuito de cada instrução:

```
00: # Carrega o tamanho do vetor
01: # Branch caso o vetor esteja vazio
02: # Carrega o primeiro item do vetor
03: # Inicializa o maior valor
04: # Inicializa o menor valor
05: # Inicializa a soma total
06: # Inicializa o item atual
 -- Loop:
07: # Branch caso seja o fim do vetor
08: # Carrega o valor do item
09: # Compara o maior valor
0A: # Verifica se atualiza o maior valor
0B: # Atualiza maior valor
 -- NãoÉMaior:
0C: # Compara o menor valor
0D: # Verifica se atualiza o menor valor
0E: # Atualiza menor valor
 -- NãoÉMenor:
0F: # Atualiza a soma total
10: # Incrementa o indice
11: # Volta para o início do loop
 -- Fim:
12: # Fim do programa
[13..FF] : # nop;
```

Os registradores utilizados para armazenamento do maior valor, menor valor e soma total foram:

- $a3 ($7): Maior valor
- $a2 ($6): Menor valor
- $a1 ($5): Soma total

Também usamos registradores para demarcar o fim do vetor (seu tamanho), o índice do vetor, o valor contido no índice do vetor, e um registrador para fazer comparações:

- $t7 ($15): Fim do vetor
- $t6 ($14): Valor do item atual
- $t5 ($13): Índice do vetor
- $t0 ($8): Comparação

Valores contidos no vetor:

```
00: 00000006;  -- Tamanho do vetor (6)
01: 00000005;  -- Valor 1 (5)
02: 00000002;  -- Valor 2 (2)
03: 00000014;  -- Valor 3 (20)
04: FFFFFFFB;  -- Valor 4 (-5)
05: FFFFFFFE;  -- Valor 5 (-2)
06: FFFFFFEC;  -- Valor 6 (-20)
[07..FF]: 00000000;
```

Explicação do código:

Inicio:

```
00: 8C0F0000;  -- lw $t7, 0($zero)
01: 11E00011;  -- beq $t7, $zero, Fim
02: 8C0E0001;  -- lw $t6, 4($zero)
03: 000E3820;  -- add $a3, $zero, $t6
04: 000E3020;  -- add $a2, $zero, $t6
05: 000E2820;  -- add $a1, $zero, $t6
06: 200D0001;  -- addi $t5, $zero, 4
```

00: Fazemos o registrador $t7 ($15) receber o tamanho do vetor

01: Caso o tamanho do vetor seja 0, significa que o vetor está vazio portanto pulamos direto para o Fim do programa.

02: É carregado no registrador $t6 ($14) o primeiro valor do vetor, portanto ele receberá o próximo valor a partir do primeiro endereço da memória (4($zero)).

03 ~ 05: Inicializamos os registradores de maior valor (registrador $a3($7)), menor valor (registrador $a2($6)) e soma total (registrador $a1($5)) com o primeiro valor do vetor.

06: O registrador $t5 ($13), que armazena o índice, passa a ter o valor do índice da primeira posição do vetor, ou seja, índice 1.

Loop:

```
    -- Loop:
07: 11AF000A;  -- beq $t5, $t7, Fim
08: 8DAE0001;  -- lw $t6, 4($t5)
09: 00EE402A;  -- slt $t0, $a3, $t6
0A: 11000001;  -- beq $t0, $zero, NãoÉMaior
0B: 000E3820;  -- add $a3, $zero, $t6
    -- NãoÉMaior:
0C: 01C6402A;  -- slt $t0, $t6, $a2
0D: 11000001;  -- beq $t0, $zero, NãoÉMenor
0E: 000E3020;  -- add $a2, $zero, $t6
    -- NãoÉMenor:
0F: 00AE2820;  -- add $a1, $a1, $t6
10: 21AD0001;  -- addi $t5, $t5, 4
11: 08100007;  -- j Loop
```

07: Verificamos se o índice é igual ao tamanho do vetor comparando os registradores $t5 e $t7, se sim, nosso vetor chegou ao fim, portanto pulamos para o Fim, finalizando o programa.

08: Carregamos em $t6 o valor da próxima posição da memória de acordo com o valor do índice, ou seja, se nosso índice é 1, iremos carregar em $t6 o valor contido no endereço 2 da memória.

09: Checamos se o valor contido no registrador $a3, que armazena o maior valor, é menor que o valor carregado em $t6, se sim, o registrador $t0 irá receber 1, se $a3 não for menor que $t6, então $t0 irá receber 0.

0A: Se $t0 recebeu 0, quer dizer que o valor em $t6 não é maior que $a3, portanto não precisamos atualizar o valor em $a3, e pulamos para NãoÉMaior.

0B: Caso $t0 tenha recebido 1, o valor em $a3 é menor que $t6, portanto atualizamos o valor de $a3 com o valor contido em $t6.

0C: Checamos se o valor em $t6 é menor que o valor em $a2, que armazena o menor valor, e $t0 recebe 1 se esta condição for verdadeira, ou 0 caso seja falsa.

0D: Se $t0 recebeu 0, significa que $t6 não é menor que $a2, portanto pulamos para NãoÉMenor, se $t0 recebeu 1, o programa continua para o endereço 0E.

0E: Atualizamos o valor de $a2 com o valor em $t6, lembrando que o programa só irá chegar nesta instrução se $t0 recebeu 1 na instrução anterior.

0F: O registrador responsável pela soma total, $a1($5), soma o seu valor atual com o valor em $t6.

10: Atualizamos o índice em mais uma posição.

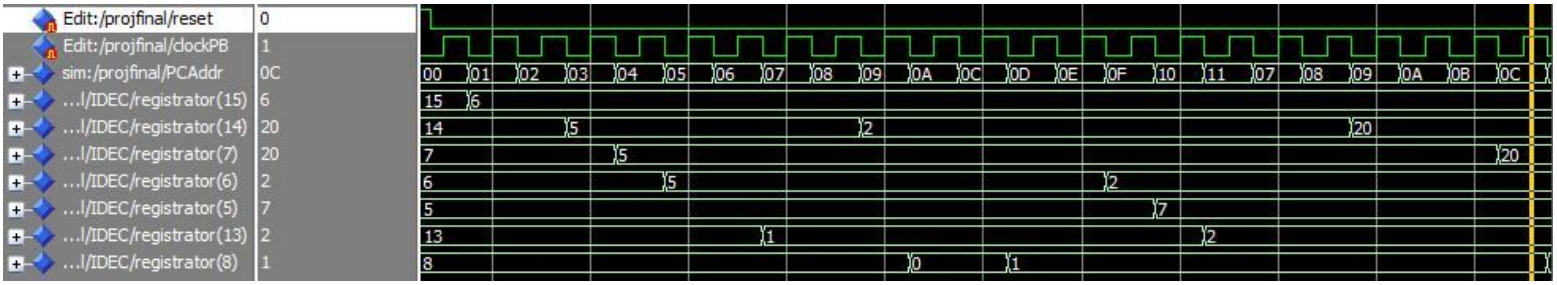
11: Fazemos um jump para a instrução 07, fechando o loop.

Fim

```
    -- Fim:
12: 00000000;  -- nop
[13..FF] : 00000000;
```

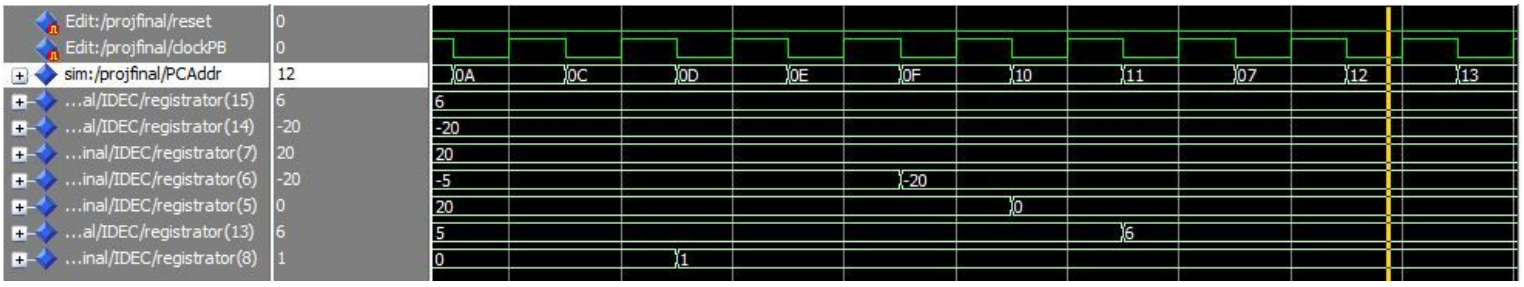
12 ~ FF: O programa recebe nop's, finalizando o programa.

A figura a seguir, representa a Wave inicial:



Podemos ver entre os endereços 00 e 05 que o registrador $t7 recebe o tamanho do vetor, $t6 recebe o primeiro valor do vetor, que é copiado para os registradores $a3, $a2 e $a1. Quando o valor atual é 20, no endereço 09, o registrador $a3, que antes tinha valor 5, é atualizado, como podemos ver no endereço 0C, onde a barra amarela está.

A figura a seguir representa a Wave final:



Podemos ver que o programa funciona corretamente, já que quando o programa chega ao fim, indicado pela barra amarela na wave acima, os valores dos registradores principais são:

- $a3 ($7): Maior valor = 20
- $a2 ($6): Menor valor = -20
- $a1 ($5): Soma total = 0

Detalhes de Implementação

- **Implementações gerais:**

Para cada instrução implementada, foram criados sinais de saída na entidade Control com os nomes de cada instrução, que recebem o seu devido valor lógico de acordo com o Opcode e Function_opcode. As condições para cada instrução são:

```
-- Code to generate control signals using opcode bits
R_format    <= '1'   WHEN  Opcode = "000000" ELSE '0';
SW          <= '1'   WHEN  Opcode = "101011" ELSE '0';
LW          <= '1'   WHEN  Opcode = "100011" ELSE '0';
Beq         <= '1'   WHEN  Opcode = "000100" ELSE '0';
Bne         <= '1'   WHEN  Opcode = "000101" ELSE '0';
Jump        <= '1'   WHEN  Opcode = "000010" ELSE '0';
Jal         <= '1'   WHEN  Opcode = "000011" ELSE '0';
Jr          <= '1'   WHEN R_format = '1' AND Function_opcode = "001000" ELSE '0';
Sl          <= '1'   WHEN R_format = '1' AND Function_opcode = "000000" ELSE '0';
Sr          <= '1'   WHEN R_format = '1' AND Function_opcode = "000010" ELSE '0';
ADDi        <= '1'   WHEN Opcode = "001000" ELSE '0';
```

Jump: recebe 1 se o Opcode = 00010, caso contrário recebe 0.
Jal: recebe 1 se o Opcode = 000011, caso contrário recebe 0.
Jr: recebe 1 se o Opcode = 000000 e Function_opcode = 001000, caso contrário recebe 0.
Sll: recebe 1 se Opcode = 000000 e Function_opcode = 000000, caso contrário recebe 0.
Srl: recebe 1 se Opcode = 000000 e Function_opcode = 000010, caso contrário reccebe 0.
Addi: recebe 1 se Opcode = 001000, caso contrário recebe 0.

Na top level (ProjFinal.vhd), foram criadas sinais internos com o mesmo nome das instruções para podermos interligar os sinais das componentes.

```
SIGNAL Bne          : STD_LOGIC;
SIGNAL Jump         : STD_LOGIC;
SIGNAL Jal          : STD_LOGIC;
SIGNAL Jr           : STD_LOGIC;
SIGNAL Sl           : STD_LOGIC;
SIGNAL Sr           : STD_LOGIC;
SIGNAL PC_inc       : STD_LOGIC_VECTOR(7 DOWNTO 0);
SIGNAL ALUOp        : STD_LOGIC_VECTOR(1 DOWNTO 0);
SIGNAL Reg31        : STD_LOGIC_VECTOR(7 DOWNTO 0);
```

4.1. J_Format

```
Next_PC <=  "00000000" WHEN reset = '1' ELSE

            --Deve-se somar o endereço do pulo caso seja Beq ou Bne
            ADDResult  WHEN (Beq = '1' AND Zero = '1') OR (Bne = '1' AND Zero = '0') ELSE

            --Deve-se receber o endereco requisitado caso seja uma instrucao de pulo incondicional
            J_Address WHEN Jump = '1' OR Jal = '1' ELSE

            --Deve-se receber o endereço armazenado no registrador 31 quando a instrução for Jr
            Reg31    WHEN Jr = '1' ELSE

            --Caso nao seja nenhuma instrucao de pulo, continua normalmente
            PC_inc;
```

- Instrução: Jump

Para a instrução Jump, criamos um vetor de 8 posições chamado J_Address na entidade Ifetch, e na top level fizemos ele receber os 8 bits menos significativos da instrução contida no DataInstr, que, caso a instrução seja de pulo, são esses 8 bits que armazenarão o endereço a ser pulado.
Quando Jump ou Jal estiver ativo alto, o Next_PC irá receber o J_Address, ou seja, ele irá receber o endereço do pulo.

- Instrução: Jal

Funciona igual ao jump em relação a como é realizado o pulo, descrito acima. A diferença é que no process da entidade Idecode, caso Jal esteja ativo alto, fazemos o registrador 31 receber o endereço da próxima instrução antes do pulo, para isso criamos um vetor chamado L_Address, que recebe o valor de PC_inc na top level.

```
PROCESS
   BEGIN
      WAIT UNTIL clock'EVENT AND clock = '1';
      IF reset = '1' THEN
         -- Inicializa os registradores com seu numero
         FOR i IN 0 TO 31 LOOP
            registrator(i) <= CONV_STD_LOGIC_VECTOR( i, 32 ); --i e um inteiro, regist
portanto precisamos converter
         END LOOP;
      --O registrador 32, $ra, armazena o endereço da próxima instrução antes do pulo
      ELSIF Jal = '1' THEN
         registrator(31) <= X"000000"&L_Address;
      ELSIF RegWrite = '1' AND write_reg_ID /= X"00" THEN
         -- Escreve no registrador indicado pela instrucao
         registrator(CONV_INTEGER(write_reg_ID)) <= write_data;
      END IF;
   END PROCESS;
```

Concatenamos 24 bits 0's (X"000000") com o L_Address já que o registrador tem 32 bits, e o L_Address tem apenas 8 bits, logo 24+8 = 32 bits.

4.2. R_Format:

- ### Instrução: Jr

Para o Jr, criamos dois vetores no Idecode. Um interno, de 32 bits, chamado RegEndereco, que recebe o conteúdo do registrador 31, e outro de saída, de 8 bits, chamado Reg31 que recebe apenas os 8 bits menos significativos de RegEndereco, ou seja, o endereço da memória.

```
--Reg31 recebe o endereco contido no registrador 31
RegEndereco <= registrator(31);
Reg31       <= RegEndereco(7 DOWNTO 0);
```

Dessa forma, no IFetch, fazemos o Next_PC receber o Reg31 caso Jr esteja em nível lógico alto.

```
Next_PC <=  "00000000" WHEN reset = '1' ELSE

            --Deve-se somar o endereço do pulo caso seja Beq ou Bne
            ADDResult  WHEN (Beq = '1' AND Zero = '1') OR (Bne = '1' AND Zero = '0') ELSE

            --Deve-se receber o endereco requisitado caso seja uma instrucao de pulo incondicional
            J_Address WHEN Jump = '1' OR Jal = '1' ELSE

            --Deve-se receber o endereço armazenado no registrador 31 quando a instrução for Jr
            Reg31    WHEN Jr = '1' ELSE

            --Caso nao seja nenhuma instrucao de pulo, continua normalmente
            PC_inc;
```

- ### Instrução: Sll

Com o SLL, tivemos que criar um sinal de entrada, Shamt, que contém os bits de 10 a 6 do DataInstr, ou seja, contém o deslocamento desejado. Além disso, tivemos que acrescentar mais um bit na ALU_ctl, que terá o controle de qual instrução deverá ser realizada. A lógica dos 3 bits menos significativos não mudam.

```
ALU_ctl( 0 ) <= ( Function_opcode( 0 ) OR Function_opcode( 3 ) ) AND ALUOp(1 );
ALU_ctl( 1 ) <= ( NOT Function_opcode( 2 ) ) OR (NOT ALUOp( 1 ) );
ALU_ctl( 2 ) <= ( Function_opcode( 1 ) AND ALUOp( 1 )) OR ALUOp( 0 );
ALU_ctl( 3 ) <= Sl OR Sr;
```

O bit mais significativo da ALU_ctl só será nível lógico alto caso SL ou SR sejam nível lógico alto.

```
CASE ALU_ctl IS
                -- ALU performs ALUresult = A_input AND B_input
   WHEN "0000"    => ALU_output_mux    <= Ainput AND Binput;
                -- ALU performs ALUresult = A_input OR B_input
   WHEN "0001"    => ALU_output_mux    <= Ainput OR Binput;
                -- ALU performs ALUresult = A_input + B_input
   WHEN "0010"    => ALU_output_mux    <= Ainput + Binput;
                -- ALU performs ALUresult = A_input -B_input
   WHEN "0110"    => ALU_output_mux    <= Ainput - Binput;
                -- ALU performs SLT
   WHEN "0111"    => ALU_output_mux    <= Ainput - Binput;
                -- ALU performs Srl
   WHEN "1110"    => ALU_output_mux    <= SHR(Binput, Shamt);
                -- ALU performs Sll
   WHEN "1010"    => ALU_output_mux    <= SHL(Binput, Shamt);
   WHEN OTHERS => ALU_output_mux    <= X"00000000" ;
```

Como podemos ver, caso os bits da ALU_ctl sejam 1010, será utilizado uma função chamada SHL, que recebe como parâmetros os bits a serem deslocados, que no caso estão armazenados em Binput (que recebe o segundo endereço ou o extensor de sinal), e quantos bits serão deslocados, que estão armazenados em Shamt, faz o cálculo, e armazena em ALU_output_mux. O resultado é então escrito no registrador, no endereço indicado por read_Rt_ID.

- Instrução: Srl

Seu funcionamento é idêntico ao SLL, as únicas diferenças são que a sequência de bits necessária na ALU_ctl para que sua operação seja feita deve ser 1110, e a função chamada é a SHR, que utiliza os mesmos parâmetros da função SHL, porém ele desloca os bits à direita.

4.3. I_Format

- Instrução: Addi

O Addi é bem simples, basicamente o que fizemos foi criar um sinal interno dentro da entidade Control que terá valor lógico alto quando o Opcode satisfazer a condição do Addi, e então, atualizamos as condições do ALUSrc e RegWrite para também levarem em conta o Addi, escrevendo no registrador o valor digitado.

```
RegDst      <=  R_format;
RegWrite    <=  R_format OR LW OR ADDi;
ALUSrc      <=  LW OR SW OR ADDi;
MemToReg    <=  LW;
MemRead     <=  LW;
MemWrite    <=  SW;
```

- Instrução: Bne

Para o Bne, as principais diferenças foram: na entidade Control, adicionamos a condição “OR Bne” na atribuição da ALUOp(0), que será usada na entidade Execute.

```
ALUOp( 1 )  <=  R_format;
ALUOp( 0 )  <=  Beq OR Bne; -- Beq deve ser 1 quando a instrução for BEQ (BNE tbm)
```

E, no IFetch, criamos a condição do Next_PC receber o ADDResult (que calcula o pulo) caso o valor do Bne seja ativo lógico alto e Zero (responsável por comparar os dados de dois registradores) seja 0.
Ou seja, Next_PC receberá ADDResult se a instrução for do tipo BNE, e o conteúdo dos dois registradores comparados sejam diferentes.

```
Next_PC <=  "00000000" WHEN reset = '1' ELSE

            --Deve-se somar o endereço do pulo caso seja Beq ou Bne
            ADDResult  WHEN (Beq = '1' AND Zero = '1') OR (Bne = '1' AND Zero = '0') ELSE

            --Deve-se receber o endereco requisitado caso seja uma instrucao de pulo incondicional
            J_Address WHEN Jump = '1' OR Jal = '1' ELSE

            --Deve-se receber o endereço armazenado no registrador 31 quando a instrução for Jr
            Reg31    WHEN Jr = '1' ELSE

            --Caso nao seja nenhuma instrucao de pulo, continua normalmente
            PC_inc;
```

5. Reconstrução do Projeto

Para reconstrução do projeto são necessários os seguintes arquivos:

- Arquivos de implementação (.VHD): LCD_Display, Ifetch, Idecode, Execute, DMEMORY e Control;
- Arquivo de Memória de Dados (DMEMORY.mif) e Arquivo de Memória de Instruções (program.mif);
- Arquivo pin planner (.csv) para a placa.

Com os devidos arquivos, basta seguir os passos listados abaixo:

1. Crie uma pasta e guarde nela todos os arquivos fornecidos;
2. Renomeie o arquivo de memória de dados da instrução que deseja testar para programT.mif
3. Abra o Quartus II;
4. Já na tela inicial do software aperte Ctrl + O ou vá em File > New > New Quartus II Project;
5. Clique em “next” na janela de Introduction;
6. Nesta página escolha o diretório onde o projeto deve ficar salvo. Dê o nome a ele de acordo com o da Top Level (ProjetoFinal) e clique em next;
7. Clique em Add All ou selecione todos os VHDs listados anteriormente de forma manual apertando em “...”;

8. No campo Family escolha Cyclone II; no campo Pin Count escolha 672; no campo Speed Grade coloque 6. Em Avaliable Devices aparecerão três opções: clique na primeira (EP2C35F672C6) e dê “Finish”.
9. Vá agora em Assignments > Import Assigments e escolha o arquivo ProjetoFinal.csv clicando no “...”. Dê “Ok”;
10. Adicione os arquivos de memória do projeto clicando em Project > Add/Remove Files In Project. Na janela que se abriu, clique em “...”, selecione os arquivos .mif das memórias, dê ADD > Aplly > OK;
11. Por fim vá até Assignments > Device > Device and Pin Options... > Unused Pins e troque a opção de Reserve All Unused Pins para “As input tri- staded”;
12. Compile o projeto.