Universidade de Brasília
Departamento de Ciências da Computação
Professor: Marcus Vínicius Lamar
Disciplina: Organização e Arquitetura de Computadores

Laboratório 5
– CPU MIPS PIPELINE –

Adarley Luiz Grando Filho	11/0007344
Guilherme Silva Lemos	13/0142468
Nur Corezzi	15/0143290
Matheus Abrantes Cerqueira	13/0144291
Paulo da Cunha Passos	10/0118577
Rafael Lima	10/0131093

Brasília
2016

Conteúdo

1 Simulação de testeWAVEFORM

Exportando o arquivo testeWAVEFORM.s para o Quartus II por meio da ”MIF Exporter”foi possível fazer a simulação em forma de onda, onde o correto funcionamento do Pipeline pode ser observado pela figuras 1 a 4. Veja que as instruções em hexadecimal, o valor do registrador PC são condizentes com o esperado teórico gerado pelo Mars. Porém algo peculiar do Pipeline pode ser observado na instrução 0x8d090000 com PC igual a 0x00400008, onde o valor de $t1 passa a valer 5. Veja que a atribuição de $t1 acontece somente 4 ciclos de clock após a instrução aparecer no campo OwIstr. Isso confirma o Pipeline, pois a atribuição ocorre somente na quinta etapa do pipeline da instrução lw. Da mesma forma o próximo valor de $t1 será atualizado 4 ciclos de clock após a insrução addi (OwInstr = 0x2129ffff e PC = 0x00400028).

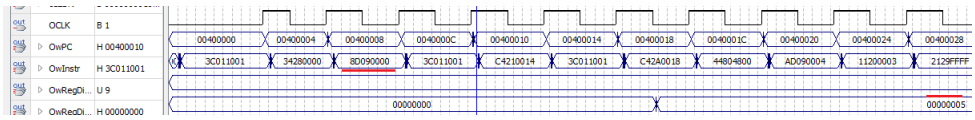

Figura 1: testeWaveform.s com detalhe para a transição de $t1 de 0 para 5 no Quartus II

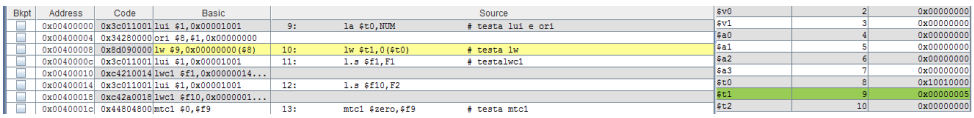

Figura 2: testeWaveform.s com detalhe para a transição de $t1 de 0 para 5 no MARS

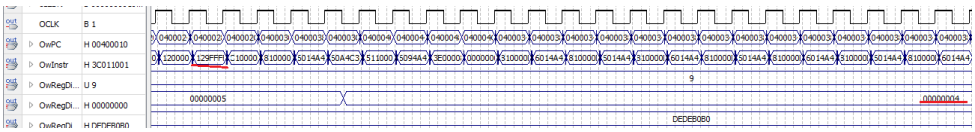

Figura 3: testeWaveform.s com detalhe para a transição de $t1 de 5 para 4 no Quartus II

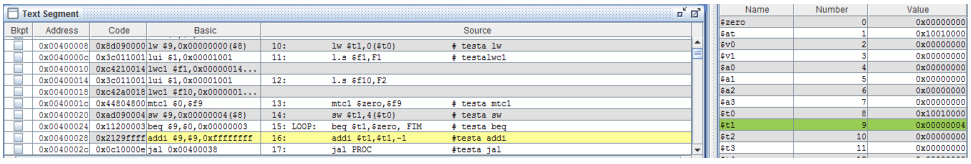

Figura 4: testeWaveform.s com detalhe para a transição de $t1 de 5 para 4 no MARS

2 Caminho de Dados MIPS Pipeline

O caminho de dados gerado pelo Quartus II é apresentado na figura 6.

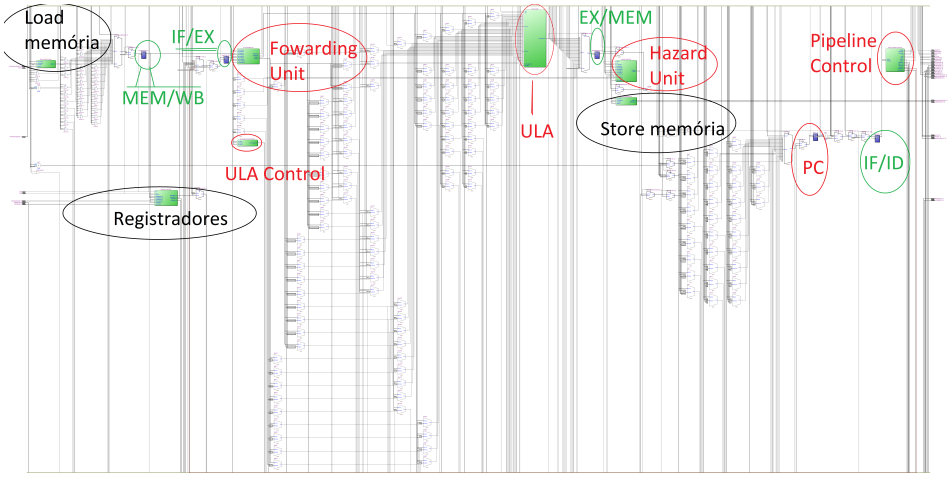

Figura 5: Caminho de dados do processador no Quartus II

se comparado com o caminho de dados apresentado em aula, da figura **??**

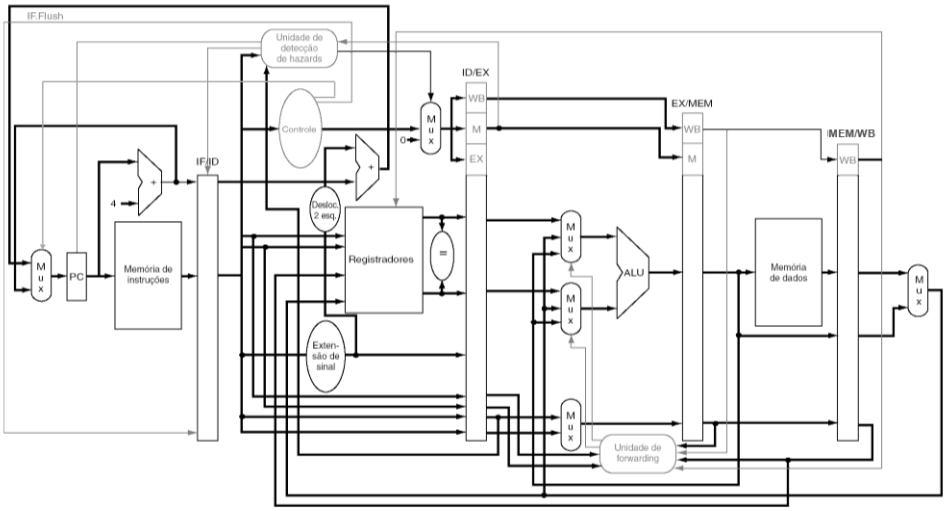

Figura 6: Caminho de dados do processador Pipeline apresentado em sala

Pode-se observar que o caminho de dados do Quartus apresenta as mesmas unidades básicas que o esperado visto em aula, como estão destacados na figura **??**, como os registradores intermediários IF/EX, MEM/WB, EX/MEM e IF/ID.

	OrigPC	Jump	nBranch	Jr
LW	0	000	0	0
SW	0	000	0	0
BEQ	0	001	0	0
Tipo R	0	000	0	0
J	1	010	0	0

Tabela 1: Tabela de controle para estágio Decodificação da instrução

	RegDst	OrigALU	OpALU
LW	00	01	00
SW	00	01	00
BEQ	00	00	01
TIPO R	01	00	10
J	00	00	00

Tabela 2: Tabela de controle para estágio execução/cálculo de endereço

	SavePC	LeMem	EscreveMem	Branch	LoadType	WriteType
LW	0	1	0	0	LW	00
SW	0	0	1	0	000	SW
BEQ	0	0	0	1	000	00
Tipo R	0	0	0	0	000	00
J	0	0	0	0	000	00

Tabela 3: Tabela de controle para estágio acesso à memória

	EscreveReg	MemparaReg
LW	1	
SW	0	
BEQ	0	
Tipo R	1	
J	0	

Tabela 4: Tabela de controle para estágio escrita do resultado

As tabelas 1 a 4 compõem a tabela verdade do controle do processador Pipeline, elas foram divididas de acordo com o estágio do pipeline. Veja que existe a adição de alguns sinais como nBranch para função *bne* e em outros o número de bits são maiores, como por exemplo o OrigPC, isso reflete a maior complexidade do processador implementado no Quartus.

Outro aspecto importante da comparação é a divisão da memória de dados, que foi dividida em dois blocos : bloco de leitura e bloco de escrita.

3 Funcionamento da Unidade de Hazard e Forward

O código do bloco de *Foward* pode ser visto a seguir:

```
always @(*) begin

    oFwdA = ((iMEM_RegWrite) && (iMEM_NumRd != 5'b0) && (iMEM_NumRd == iEX_NumRs)) ? 2'b10
          :((iWB_RegWrite) && (iWB_NumRd != 5'b0) && (iMEM_NumRd != iEX_NumRs) &&
(iWB_NumRd == iEX_NumRs)) ? 2'b01
          : 2'b00;

    oFwdB = ((iMEM_RegWrite) && (iMEM_NumRd != 5'b0) && (iMEM_NumRd == iEX_NumRt)) ? 2'b10
          : ((iWB_RegWrite) && (iWB_NumRd != 5'b0) && (iMEM_NumRd != iEX_NumRt) &&
(iWB_NumRd == iEX_NumRt)) ? 2'b01
          : 2'b00;

    oFwdBranchRs = ( iMEM_RegWrite && (iID_NumRs != 5'b0) && (iMEM_NumRd == iID_NumRs)) ? 1'b1 : 1'b0;

    oFwdBranchRt = ( iMEM_RegWrite && (iID_NumRt != 5'b0) && (iMEM_NumRd == iID_NumRt)) ? 1'b1 : 1'b0;

    end

endmodule
```

A instrução analisada (n) está na etapa de execução(EX), compara-se os registradores RS e RT da instrução atual com os registradores RD da instrução anterior , n-1, ou com a instrução que está na etapa de escrita do resultado (WB), ou instrução n-2.

Os resultados de *fowarding* alteram a entrada da ULA, mudando a seleção de multiplexadores (sinais oFwdA e oFwdB) onde o sinal 00 não altera a entrada da ULA, 10 seleciona como entrada da ULA o resultado calculado na etapa anterior que está no registrador EX/MEM. A figura 7 representa essa ocorrencia de *fowarding* onde o multiplexador altera a entrada A da ULA por meio do sinal oFwdA.

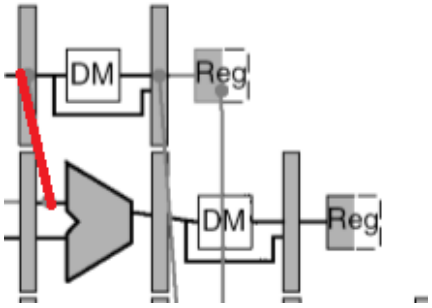

Figura 7: *Fowarding* de estágios MEM para EX

Se o sinal de seleção for igual a 01 o *fowarding* acontece da instrução que está no estágio WB para a instrução no estágio EX. A figura 8 apresenta esse tipo de seleção, porém quando a origem B da ULA é modificada.

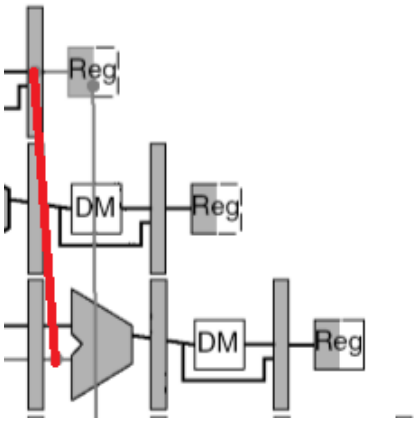

Figura 8: *Fowarding* do estágio WB para EX

Para ambos os casos descritos acima o sinal de *fowarding* é acionado somente se os registradores analisados forem diferentes do registrador *$zero*. Além disso o sinal *oFwdBranchRs* verifica se o registrador atualizado (Rd) no estágio MEM é igual ao registrador operando (Rs ou Rt) no estágio ID, assim englobando operações de branch, onde é verificado no estágio ID, assim como a figura 9.

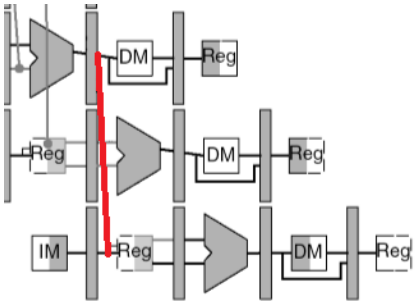

Figura 9: *Foward* em operações de branch como BEQ e BNE

Já o bloco de Hazard possui três sinais que são mostrados a seguir:

```
    wire wEX_RegHazard  = (iEX_RegDst == iID_NumRs) || (iEX_RegDst == iID_NumRt);
    wire wMEM_RegHazard = (iMEM_RegDst == iID_NumRs) || (iMEM_RegDst == iID_NumRt);

    wire wEX_Hazard  = (iEX_MemRead || iBranch) && iEX_RegWrite && (iEX_RegDst != 5'b0) &&
wEX_RegHazard;
    wire wMEM_Hazard = iBranch && iMEM_MemRead && iMEM_RegWrite && (iMEM_RegDst != 5'b0) &&
wMEM_RegHazard;

    assign oHazard = (wEX_Hazard || wMEM_Hazard) ? 1'b1 : 1'b0;

    assign oForwardJr = ((iCJr) && (iEX_RegDst == iID_NumRs)) ? 1'b1 : 1'b0;

    assign oForwardPC4 = ((iCJr) && (iMEM_RegDst == 5'd31)) ? 1'b1 : 1'b0;
```

O sinal *wEX_RegHazard* verifica se os registrados rs ou rt foram atualizados na instrução anterior, e o sinal *wMEM_RegHazrd* verifica se mesmos registradores foram atualizados duas instruções atrás.

Com esses sinais, verifica-se se a instrução atual é um branch, se 2 instruções atrás (estágio MEM) ocorre leitura de memória e se ocorre um hazard do tipo *wMEM_RegHazard*, caso sejam todos verdadeiros o sinal *wMEM_Hazard* é verdadeiro

Se a etapa anterior ocorreu leitura da memória ou a atual instrução um branch e o sinal de *wEX_Hazard* estiver ativo ocorre um sinal de *EX_Hazard*.

Ao fazer OR de *EX_Hazard* com *MEM_Hazard* cobre-se casos como:

```
    ...
    lw \$t1,0(\$sp)
    _
    beq \$t1,\$t2,label
    ...

    ou

    ..
    lw \$t1,0(\$sp)
    add \$t2,\$t1,\$t3
    ..

    ou
    ..
    addi \$t1,\$zero,0
    beq \$t1,\$t2, label
    ..
```

Além disso os seguinte os sinais de *oFowardJr* e *oFowardPC4* cobrem os seguintes casos:

```
    ..
    add \$r,\$a0,\$a1
    jr \$r
    ..
    ..
    add \$ra,\$s0,\$s1
    _
    jr \$ra
    ..
```

Isso ocorre pois compara-se se a instrução é um *jr* e se o registrador atualizado na instrução anterior é utilizado instrução atual, caso ambos sejam verdadeiros *oFowardJr* será ativada. Em contrapartida verifica-se se o registrador atualizado duas instruções acima é $\$ra$, caso seja verdade *oFowardPC4* será ativado.

4 Teste Instruções MIPS Pipeline

Foi criado um arquivo teste.s com as instruções presentes no Pipeline para verificar o correto funcionamento das mesmas. O arquivo foi exportado pela ferramenta *MIF Exporter* e simulado em forma de onda no Quartus II para comparar os resultados da forma de onda com os resultados obtidos no Mars. Algumas dessas comparações são mostradas nas figuras 10 a 25.

4.1 Teste ADD

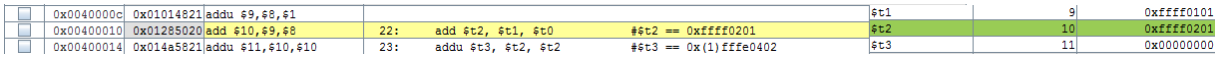

Figura 10: Execução do arquivo teste no Mars, instrução add e resultado

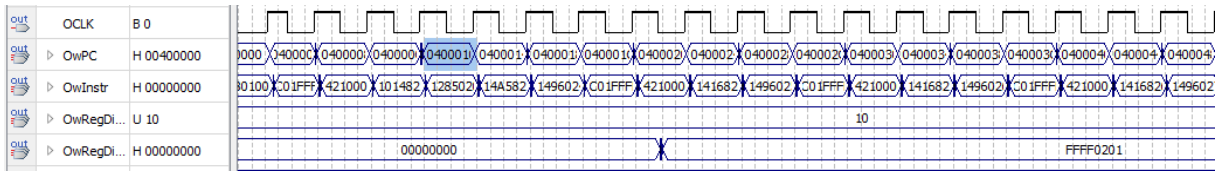

Figura 11: Forma de onda para teste da instrução add

4.2 Teste SUB

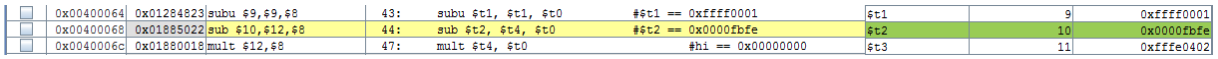

Figura 12: Parte do arquivo no Mars com instrução sub e respectivo resultado

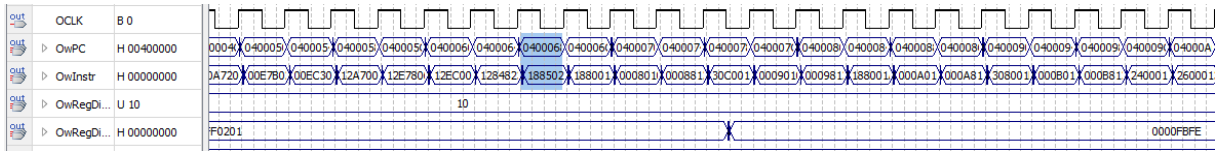

Figura 13: Parte da forma de onda para com execução da instrução sub

4.3 Teste OR

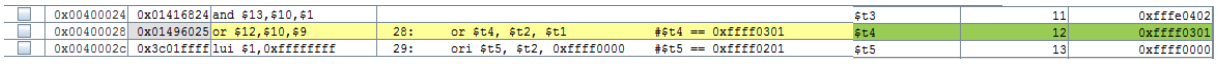

Figura 14: Parte do arquivo no Mars com instrução or e respectivo resultado

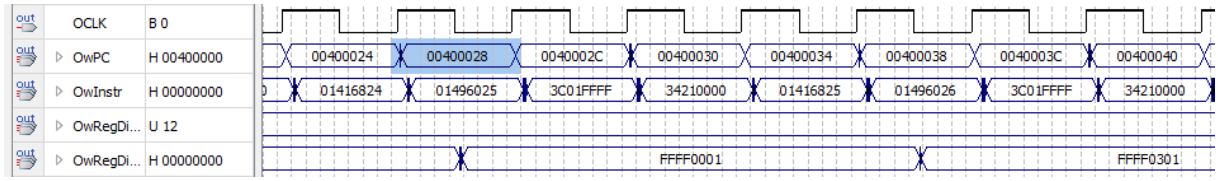

Figura 15: Segmento da forma de onda com execução da função or

4.4 Teste SLL

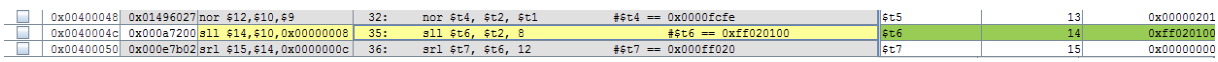

Figura 16: Instrução sll no Mars para referência

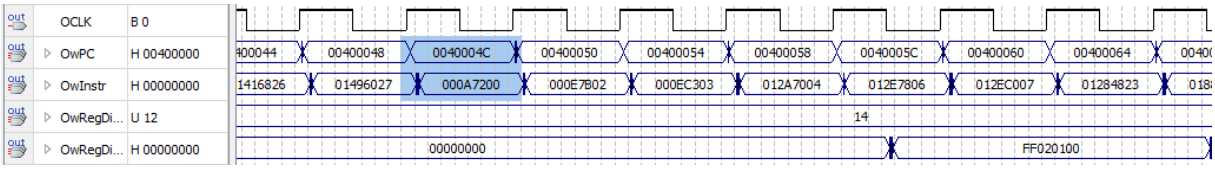

Figura 17: Forma de onda com execução da instrução sll

4.5 Teste AND

```
0x00400014  0x014a5821  addu $11,$10,$10      23:  addu $t3, $t2, $t2      #$t3 == 0x(1)fffe0402  $t3  11  0xfffe0402
0x00400018  0x01496024  and $12,$10,$9        26:  and $t4, $t2, $t1       #$t4 == 0xffff0001     $t4  12  0xffff0001
0x0040001c  0x3c01ffff  lui $1,0xffffffff     27:  andi $t5, $t2, 0xffff0000  #$t5 == 0xffff0000  $t5  13  0x00000000
```

Figura 18: Segmento do arquivo teste com instrução and para referência

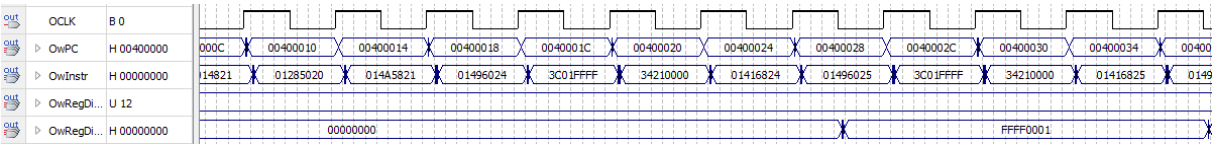

Figura 19: Forma de onda com execução da instrução and para comparação

4.6 Teste ADDI

```
0x00400000  0x20080100  addi $8,$0,0x00000100   20:  addi $t0, $zero, 0x00000100 #$t0 == 0x00000100  $a2  6  0x00000000
0x00400004  0x3c01ffff  lui $1,0xffffffff       21:  addiu $t1, $t0, 0xffff0001  #$t1 == 0xffff0101  $a3  7  0x00000000
0x00400008  0x34210001  ori $1,$1,0x00000001                                                         $t0  8  0x00000100
0x0040000c  0x01014821  addu $9,$8,$1                                                                $t1  9  0x00000000
0x00400010  0x01285020  add $10,$9,$8           22:  add $t2, $t1, $t0       #$t2 == 0xffff0201      $t2  10 0x00000000
```

Figura 20: Segmento do arquivo teste com instrução addi para referência

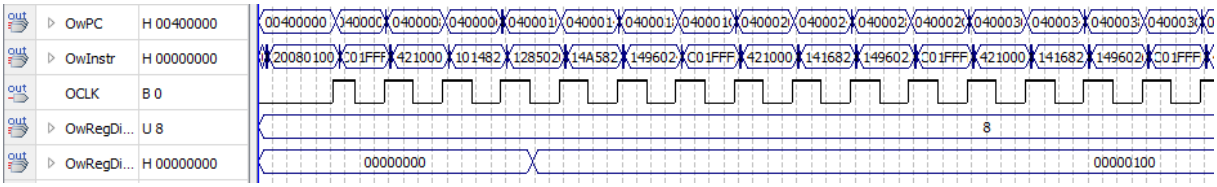

Figura 21: Forma de onda com execução da instrução addi para comparação

4.7 Teste ADDIU

```
0x00400004  0x3c01ffff  lui $1,0xffffffff       21:  addiu $t1, $t0, 0xffff0001  #$t1 == 0xffff0101  $t0  8  0x00000100
0x00400008  0x34210001  ori $1,$1,0x00000001                                                         $t1  9  0xffff0101
0x0040000c  0x01014821  addu $9,$8,$1                                                                $t2  10 0x00000000
```

Figura 22: Parte do código com instrução addiu e seu resultado

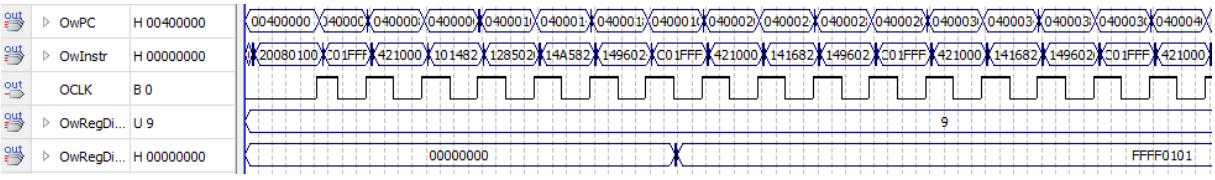

Figura 23: Forma de onda para comparação da instrução addiu

4.8 Teste de SW e LW

Para teste das instuções sw e lw o arquivo teste.s atribui o endereço 0x10010080 no registrador $\$t9$ e em seguida a função sw salva 0xFFFEF4F2 que será lida na instrução lw e salva no registrador $\$vo$.

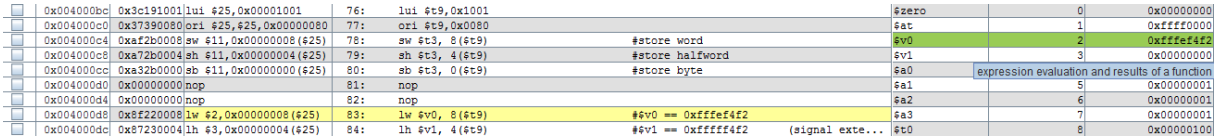

Figura 24: Execução no Mars das instruções Sw e Lw para referência

A figura 25 executa o arquivo teste.s em forma de onda verificando o registrador $\$vo$. O correto funcionamento das instruções foi verificado uma vez que o registrador $\$vo$ teve seu valor alterado após a instrução lw, mais precisamente 4 ciclos de clock após a decodificação da instrução.

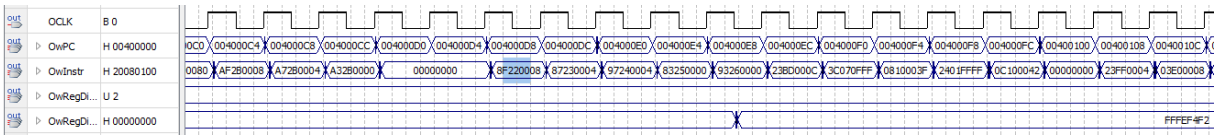

Figura 25: Forma de onda com instruções SW e LW para comparação.

5 Teste Plotter Pipeline

O arquivo com desenhos gráficos utilizado no outros processadores não pode ser utilizado no Pipeline uma vez que instruções de ponto flutuante eram utilizadas, dessa forma teve-se que alterar as funções de ponto flututuante por instruções inteiras. Além disso foram inseridas 2 instruções *nop* após cada instrução do tipo jump, como j, jr e jal.

Após alterações no código para utiliza-lo no pipeline o correto funcionamento pode ser observado.[1]

6 Novas Instruções

Para implementar as operações MADD, MADDU, MSUB e MSUB, realizou-se modificações na ULA, no controle da ALU, nos Parâmetros do processador e no Controle do Pipeline. E não houve a necessidade de alterar a unidade de alterar o caminho de dados e as demais unidades funcionais.

6.0.1 Modificações no Controle da ALU

Para a modificação do controle da ALU seguiu-se a seguinte lógica, quando a ALUop == 2'b11 e o Opcode == OPMACC, procura-se qual a operação será realizada por meio de seu *Funct* e passa-se o parâmetro da operação para o sinal de controle da ALU. As modificações no arquivo ALUControl.v foram feitas conforme apresentado abaixo:

```
always @(iFunct, iOpcode, iALUOp, iRt)
begin
    case (iALUOp)
        2'b00:
        ...
        2'b11:
            case (iOpcode)
            ...
            // 2/2016 FUNMADD e FUNSLL possuem mesmos valores portanto avaliaremos aqui
```

[1] https://www.youtube.com/watch?v=gLnXIQuTqHw

```
                OPMACC:
                    begin
                        case(iFunct)
                            FUNMADD:
                                oControlSignal = OPMADD;
                            FUNMADDU:
                                oControlSignal = OPMADDU;
                            FUNMSUB:
                                oControlSignal = OPMSUB;
                            FUNMSUBU:
                                oControlSignal = OPMSUBU;
                             default:
                                oControlSignal  = 5'b00000;
                        endcase
                    end
                ...
```

6.0.2 Modificações na ALU

Na ALU, foram incluídos os comandos OPMADD, OPMADDU, OPMSUB e OPMSUBU, que serão executados quando a ALU receber algum deles por meio do sinal de controle da ALU:

```
...
always @(posedge iCLK)
  begin
    if (iRST)
    begin
        {HI,LO}     <= 64'b0;
    end
    else
        case (iControlSignal)
        ...
            OPMADD:
                    {HI,LO} <= {HI,LO} + (iA * iB);

            OPMADDU:
                    {HI,LO} <= {HI,LO} + ($unsigned(iA) *$unsigned(iB));

            OPMSUB:
                    {HI,LO} <= {HI,LO} - (iA * iB);

            OPMSUBU:
                    {HI,LO} <= {HI,LO} - ($unsigned(iA) * $unsigned(iB));
           ...
           endcase
         ...
```

6.1 Modificações no Controle do Pipeline

Para implementar as operações MADD, MSUB, MADDU e MSUBU alterou-se a estrutura do Controle do Pipeline da seguinte forma:

Listing 1: Control_PIPEM.v

```
        ...
                OPMACC:
                        begin
                                oRegDst     = 2'b11;  // seleciona o \$0 (pra nada)
                                oOrigALU    = 2'b00;  // seleciona o resultado do fowardB
                                oSavePC     = 1'b0;   // seleciona o resultado da ALU
                                oEscreveReg = 1'b0;   // desativa EscreveReg
                                oLeMem      = 1'b0;   // desativa LeMem
                                oEscreveMem = 1'b0;   // desativa EscreveMem
                                oOrigPC     = 3'b000; // seleciona PC+4
                                oOpALU      = 2'b11;  // operacao da ALU
                                oJump       = 1'b0;   // desativa jump
                                oBranch     = 1'b0;   // desativa branch
                                onBranch    = 1'b0;   // desativa BNE
                                oJr         = 1'b0;   // desativa o Jr
                                oLoadType   = 3'b000; // load word/ignore
                                oWriteType  = 2'b00;  // write word/ignore
                        end
        ...
```

- RegDst: Terá valor 11, mas que de fato não nos interessa uma vez que não será efetuada escrita no banco de registradores, pois apenas acumulamos valor do resultado da operação em registradores internos a ALU também conhecidos como HI e LO;

- OrigALU: Recebe 00, que representa a seleção do resultado que vem do fowardB

- OpALU: Terá valor 11, que é o código que levará a seleção das novas operações implementadas no interior da ULA

Os demais sinais de controle seguem conforme o código a cima.

6.1.1 Modificações em Parâmetros

Nos parâmetros do processador acrescentou-se o seguinte código:

Listing 2: Parametros.v

```
...
OPMADD      = 5'b11000,                    //24                    2016/2
    OPMADDU         = 5'b11001,                        //25                    2016/2
    OPMSUB      = 5'b11010,                //26                    2016/2
    OPMSUBU     = 5'b11011,                //27                    2016/2
    ...
    FUNMADD     = 6'h00,                           // 2/2016       AluOp <= 11 para diferenciar FUN igu
    FUNMADDU        = 6'h01,                               // 2/2016
    FUNMSUB         = 6'h04,                               // 2/2016
    FUNMSUBU        = 6'h05,                               // 2/2016
    ...
MACC            = 6'd61;
```

6.1.2 Modificações no Caminho de Dados e Hazards

Não houve a necessidade de alterar o caminho de dados, pois apenas com as modificações nas unidades funcionais já apresentadas, foi possível viabilizar as operações MADD, MSUB, MADDU e MSUBU. Também não houve a necessidade de alterar as unidades HazardUnitM e ForwardUnitM, pois apesar de haver a possibilidade de hazard de dados, já há implementação do tratamento desses hazard por meio de forwarA e forwardB na unidade funcional ForwardUnitM. E ainda como as demais instruções que necessitam de dados provenientes de HI e LO leem tais registradores somente na terceira etapa, não haverá hazard de dados, pois as operações implementadas também leem e escrevem nesses registradores somente na terceira etapa do processamento (etapa de calculo na ALU EX).

6.1.3 Funcionalidade das Operações

Para verificar a funcionalidade das instruções foram codificados testes no mars da forma abaixo, apenas substituindo as instruções madd por maddu, msub e msubu:

```
.text
    addi $t0, $zero, -1
    addi $t2, $zero, 1

    madd $t0, $t2
    madd $t0, $t2
    mfhi $t1
    mflo $t1

    add $t1, $0, $0
    mthi $0
    mtlo $0
    maddu $t0, $t2
    maddu $t0, $t2
    mfhi $t1
    mflo $t1

    add $t1, $0, $0
    mthi $0
    mtlo $0
    msub $t0, $t2
    msub $t0, $t2
    mfhi $t1
```

```
mflo $t1

add $t1, $0, $0
mthi $0
mtlo $0
msubu $t0, $t2
msubu $t0, $t2
mfhi $t1
mflo $t1

add $t1, $0, $0
```

Desta forma iremos sempre operar -1 e 1 para podermos notar as diferenças entre as operações com e sem sinal, e reproduzir a operação pelo menos 2 vezes para mostrar que realmente estamos acumulando o resultado em HI e LO. A seguir obtivemos os seguintes dados

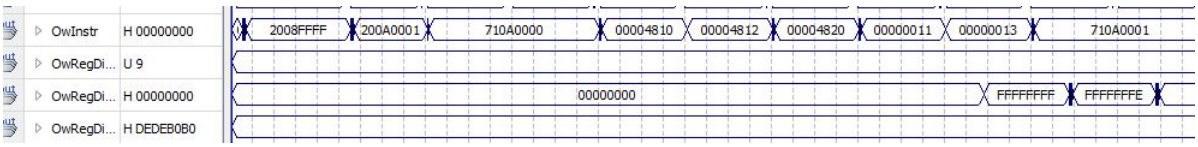

Figura 26: Simulação em forma de onda MADD

Aqui iremos operar (-1*1) e adicionar 2 vezes em HI e LO portanto como podemos notar os resultados esperados seriam LO = -2 = FFFF_FFFE e HI = -1 = FFFF_FFFF, para completar o valor de 64 bits com sinal.

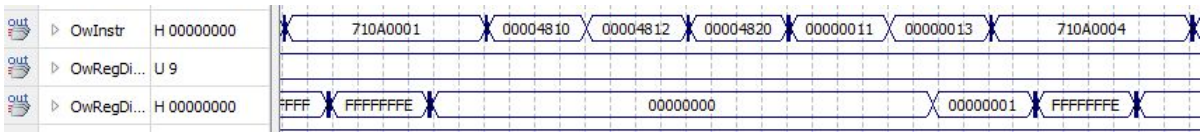

Figura 27: Simulação em forma de onda MADDU

Aqui iremos operar (-1*1) e adicionar 2 vezes em HI e LO porém o resultado em LO não será propagado para HI pois iremos considerar uma soma sem sinal logo como podemos notar os resultados esperados seriam LO = -2 = FFFF_FFFE e HI = 1 = 0000_0001, resultante da soma de FFFF_FFFF com FFFF_FFFF.

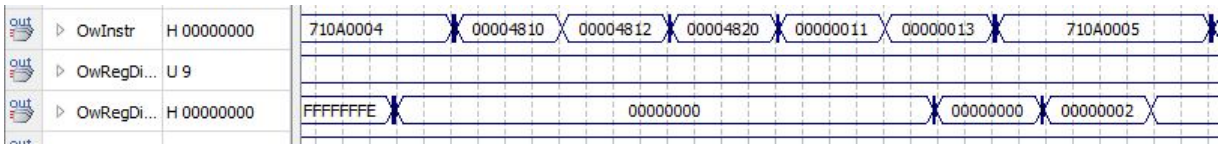

Figura 28: Simulação em forma de onda MSUB

Aqui iremos operar (-1*1) e subtrair 2 vezes em HI e LO portanto como podemos notar os resultados esperados seriam LO = 2 = 0000_0002 e HI = 0 = 0000_0000, pois a ao subtrair HI,LO de -1 estaremos executando uma soma.

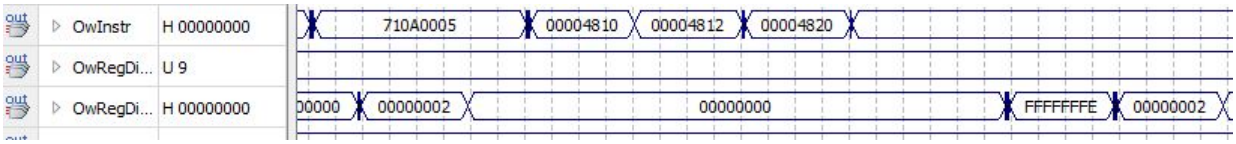

Figura 29: Simulação em forma de onda MSUBU

Aqui iremos operar (-1*1) e subtrair 2 vezes em HI e LO porém desconsiderando, portanto como podemos notar os resultados esperados seriam LO = 2 = 0000_0002 e HI = -2 = FFFF_-FFFE.

7 Lock Interno e Externo

8 IrDa

Para implementação do IrDA primeiramente foi necessário estabelecer o protocolo de comunicação a ser utilizado para envio e recebimento de dados. O protocolo escolhido por simplicidade e por ser um padrão comumente utilizado foi o NEC. Este padrão utiliza 32 bits de dados para envio com algumas convenções para inicio e encerramento de leitura. Abaixo temos o padrão exemplificado:

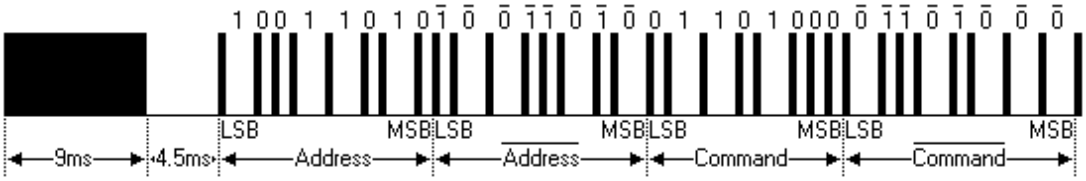

Figura 30: NEC Protocol

8.1 Receber Dados

Para a implementação do recebimento de dados foi necessário a utilização de uma máquina de estados com os estados anteriormente informados, utilizando como base os tempos convencionados logo acima, aqui faremos uma breve analise do código uma vez que o código fonte esta em anexo. Em verilog teremos:

```
parameter IDLE                  = 2'b00;
parameter GUIDANCE              = 2'b01;
parameter DATAREAD              = 2'b10;
```

Os quais explicitam os estados a serem utilizados identificando-os por 2 bits que serão armazenados em um registrador de estados. Em seguida especificamos alguns limites para leitura dos dados:

```
parameter IDLE_HIGH_DUR         =  262143;      //262143*0.02us = 5.24ms
parameter GUIDE_LOW_DUR         =  230000;      //230000*0.02us = 4.60ms
parameter GUIDE_HIGH_DUR        =  210000;      //210000*0.02us = 4.20ms
parameter DATA_HIGH_DUR         =  41000;       //41500 *0.02us = 0.83ms
parameter BIT_AVAILABLE_DUR     =  20000;       //20000 *0.02us = 0.4ms
```

Estes parâmetros especificam limites máximos para determinadas situações, onde teremos, portanto, para melhor compreensão:

- IDLE_HIGH_DUR: Valor limite para mudança de estado DATAREAD para IDLE, caso o sinal tenha parado de ser recebido, podemos garantir que passaremos novamente para o estado IDLE caso tenhamos alguma falha no recebimento.

- GUIDE_LOW_DUR: Valor limite que permanecerá em estado idle ao iniciar leitura da cadeia de dados (Identificação do pulso de 9ms em low).

- GUIDE_HIGH_DUR : Valor limite que permanecerá em estado GUIDANCE ao identificar HIGH de 4.5ms após sinal de 9ms.

- DATA_HIGH_DUR: Duração do sinal de bit 1.

- BIT_AVAILABLE_DUR: Duração mínima para se identificar um bit seja 0 ou 1.

Em seguida estabelecemos o fluxo da máquina de estados que de forma simplificada ficará como identificado logo abaixo:

```
always @(posedge iCLK or negedge iRST_n)
          if (!iRST_n)
             state <= IDLE;
          else
                        case (state)
                           IDLE      : if (idle_count > GUIDE_LOW_DUR)
                                          state <= GUIDANCE;
                           GUIDANCE : if (state_count > GUIDE_HIGH_DUR)
                                          state <= DATAREAD;
                           DATAREAD : if ((data_count >= IDLE_HIGH_DUR) || (bitcount >= 33))
                                        state <= IDLE;
                default   : state <= IDLE;
                        endcase
```

Portanto, sempre que estivermos em estado IDLE caso seja identificado um sinal em baixo(sinal de 9ms em low), iniciaremos uma contagem que será armazenada em idle_count, após passar o limite anteriormente estabelecido de GUIDE_LOW_DUR permitimos então a passagem para o estado GUIDANCE.

Em GUIDANCE iniciaremos a contagem em state_count que ao passar do limite GUIDE_-HIGH_DUR de 4.5ms passará a fazer a leitura efetivamente do dado no estado de DATAREAD.

Em DATAREAD iniciaremos uma nova contagem em data_count que caso não receba nenhuma nova informação, contará até após limite de IDLE_HIGH_DUR o que fará com que retorne para IDLE porém sem um dado válido. Caso sejam encontrados todos os 32 bits de dados retornará para IDLE com os dados lidos.

Para melhor explicitação da leitura de dados iremos avaliar também o seu fluxo iniciando pela contagem do índice que nos informará em que bit nos encontramos na leitura de dados:

```
always @(posedge iCLK or negedge iRST_n)
    if (!iRST_n)
       bitcount <= 6'b0;
          else if (state == DATAREAD)
                begin
                        if (data_count == BIT_AVAILABLE_DUR)
                                        bitcount <= bitcount + 1'b1;
                end
          else
             bitcount <= 6'b0;
```

Portanto, sempre que estivermos em DATAREAD caso a contagem de bit atinja um valor mínimo de BIT_AVAILABLE_DUR, iremos acrescentar ao nosso índice. Em seguida vamos verificar se a duração do sinal em HIGH equivale a um bit 1 ou 0, caso seja um bit 0 não iremos modificar em nada nosso registrador de leitura (identificado por data) uma vez que este já foi inicializado com valores em 0 e o nosso índice já foi acrescido para a avaliação do próximo bit, logo é necessário apenas avaliar se teremos a duração de um bit HIGH que será avaliado da seguinte forma:

```
always @(posedge iCLK or negedge iRST_n)
          if (!iRST_n)
             data <= 0;
                else if (state == DATAREAD)
                begin
                        if (data_count >= DATA_HIGH_DUR)
                           data[bitcount-1'b1] <= 1'b1;
                end
                else
                        data <= 0;
```

Vale ressaltar que data_count somente inicia sua contagem ao receber um bit em HIGH, logo quando temos um gap em DATAREAD o valor de data_count é reiniciado, para que possamos novamente observar se a contagem corresponde a um bit 0 ou 1.

Em seguida após a leitura de todos os bits é necessário apenas que seja feita uma verificação do padrão utilizado antes de jogar o valor de data para o módulo que o requisitou, para isso foi acrescentado o seguinte:

```
always @(posedge iCLK or negedge iRST_n)
          if (!iRST_n)
             data_ready <= 1'b0;
    else if (bitcount == 32)
                begin
                        if (data[31:24] == ~data[23:16])
                        begin
                                data_buf <= data;
                                data_ready <= 1'b1;
                        end
                        else
                                data_ready <= 1'b0 ;
                end
                else
                   data_ready <= 1'b0 ;
```

Ou seja, caso os bits do comando sejam complementares teremos um dado válido e podemos informá-lo para quem requisitou o módulo de leitura, jogando data_buff para o valor de output do módulo.

8.2 Enviar Dados

Para o envio de dados foi utilizado apenas uma lógica inversa do receptor de dados para isso foram utilizados os mesmos estados apenas com o acréscimo de um estado END que por praticidade informará o final da cadeia de dados.

```
                                parameter IDLE              = 2'b00;
                                parameter GUIDANCE          = 2'b01;
                                parameter DATAREAD          = 2'b10;
                                parameter END               = 2'b11;
```

Para os valores de duração de dados e validação teremos os seguintes:

```
    parameter GUIDE_LOW                 = 450000;       // 9ms low voltage
    parameter GUIDE_HIGH                = 225000;       // 4.5ms high voltage
    parameter DATA_GAP                  = 30000;        // 0.6ms gap signal
    parameter DATA_HIGH                 = 83000;        // 1.66ms high voltage
    parameter DATA_LOW                  = 26000;        // 0.52ms high voltage
    parameter END_HIGH                  = 30000;        // 0.6ms high voltage
```

Uma vez estabelecido o protocolo os valores acima são auto explicativos, estes irão determinar exatamente a duração do envio calculados para um clock especifico de 50Mhz.

Para o fluxo da máquina de estados teremos condições muito parecidas com as de recepção explicitando-as logo abaixo:

```
always @(posedge iCLK or negedge iRST_n)
          if (!iRST_n)
                        begin
                                state <= IDLE;
                                data <= 32'b0;
                                txd_busy <= 1'b0;
                        end
          else
                        begin

                                if(iTXD_READY && !txd_busy)
                                        begin
                                                data <= iDATA;
                                                txd_busy <= 1'b1;
                                        end

                                case (state)
                                        IDLE      : if (idle_count > GUIDE_LOW)
                                                state <= GUIDANCE;
                                        GUIDANCE : if (guide_count > GUIDE_HIGH)
                                                    state <= DATASEND;
                                        DATASEND : if (bitcount == 32 && data_txd)/
                                                      begin
                                                              txd_busy <= 1'b0;
                                                              state <= END;
                                                      end
                                        END              : if(end_count > END_HIGH)
                                                                        state <= IDLE;
                                  default   : state <= IDLE; //default
                                endcase
                        end
```

Estando em IDLE esperamos então a recepção de um sinal HIGH em iTXD_READY e caso não estejamos fazendo nenhuma transmissão de dados iremos inicializar nosso buffer data e jogar txd_busy para HIGH para assim mostrar que a transmissão já esta sendo feita.

Assim que txd_busy estiver em HIGH idle_count irá iniciar uma contagem de 9ms e informara por meio do output oIRDA_TXD um sinal LOW para ser transmitido.

Em seguida entraremos em estado GUIDANCE que irá da mesma forma fazer uma contagem de 4.5ms informando HIGH para oIRDA_TXD.

Terminada a contagem entraremos em estado DATA_SEND, aqui a cadeia de dados será percorrida por um indice que identificará se o bit sendo enviado é 1 ou 0 setado o limite do contador para o valor correto, para melhor compreensão iremos analisar um fragmento do código:

```
if(!data_txd)   //caso estejamos em um gap
        begin
                if(data_count > DATA_GAP)
                        begin
                                data_txd <= 1'b1;//jogamos para high transmissao de bit
                                data_count <= 0;
                        end
        end

else if(data_txd) //caso estejamos enviando um bit
        begin
                if(data[bitcount])                                          //caso seja bit high
                        begin
                                if(data_count > DATA_HIGH)      //esperamos por timer do high
                                        begin
                                                data_txd <= 1'b0;
                                                bitcount <= bitcount + 1'b1;
                                                data_count <= 0;
                                        end
                        end
                else if(!data[bitcount])                                                                    //cas
                        begin
                                if(data_count > DATA_LOW)       //esperamos por timer do low
                                        begin
                                                data_txd <= 1'b0;
                                                bitcount <= bitcount + 1'b1;
                                                data_count <= 0;
                                        end
                        end
        end
```

Em DATA_SEND, data_txd é utilizado para identificar se estamos em um gap ou não alternando sempre seu valor, ora enviamos gap ora enviamos bit de dado reiniciando sempre a contagem de data_count. Aqui para atualizar o indice do bit sendo enviado acrescentamos diretamente ao identificar o bit. Assim ao chegar a contagem de 32 bits, contaremos mais um gap que ira jogar data_txd para 1, esta situação será diretamente avaliada na máquina de estados para passar para estado END (bitcount == 32 && data_txd).

Para envio efetivamente dos sinais foi sintetizado um circuito a parte capaz de avaliar o estado em que o envio se encontra e suas necessidades de envio:

```
    always @(posedge iCLK or negedge iRST_n)
        if (!iRST_n)
                send_high <= 1'b1;
        else
                case (state)
                        IDLE     : if (idle_txd)
                                send_high = 1'b0;
                        GUIDANCE : if (guide_txd)
                                send_high = 1'b1;
                        DATASEND : if (data_txd)
                                send_high = 1'b1;
                         else if(!data_txd)
                                send_high = 1'b0;
                        END              : if(end_txd)
                                send_high = 1'b1;
                         default  : state <= IDLE; //default
                endcase
```

Podemos observar que em cada estado temos um sinal de controle que é capaz de informar para oIRDA_TXD um sinal HIGH ou LOW dependendo da sua necessidade, para IDLE por exemplo é necessário apenas enviar um sinal em LOW mas para DATA_SEND como teremos gaps e bits de data poderemos enviar sinais em HIGH ou em LOW dependendo da necessidade.

8.3 Comunicação entre duas DE2-70

Para a comunicação entre placas é valido compreender melhor a implementação das interfaces do transmissor e do receptor, e como suas instancias se comunicam com o processador mips. Primeiramente para o receptor foi implementada a seguinte interface:

```
IR_RECEIVE IrDAreceiver
(
    .iCLK(iCLK_50),
    .iRST_n(Reset),
    .iIRDA(oIRDA_TXD),
    .oDATA_READY(data_ready),
    .oDATA(hex_data)
);

    always @(*)
    if(wReadEnable)
        if(wAddress == IrDA_READ_ADDRESS)
           wReadData = hex_data;
                        else if(wAddress == IrDA_CONTROL_ADDRESS)
                            wReadData = {30'b0,txd_busy,data_ready};
         else
                            wReadData = 32'hzzzzzzzz;
    else
                        wReadData = 32'hzzzzzzzz;
```

Teremos portanto que enviar para o modulo receptor um sinal de clock de 50Mhz, um sinal de reset e os sinais recebidos pelo receptor. Como saídas este informará se o dado esta pronto, caso tenha sido codificado de forma correta, e o próprio dado decodificado.

Para recepção de códigos foi criado um endereço em parâmetros nomeado IrDA_READ_-ADDRESS que ao tentar ser acessado jogara para o barramento de dados wReadData o valor lido hex_data. Para que seja possível uma comunicação mais segura foi criado também um endereço em parâmetros para o sinal de controle chamado IrDA_CONTROL_ADDRESS, este será responsável por informar se o dado já esta pronto ou não para ser lido.

Em seguida para o transmissor também foi necessário a identificação de um endereço para a escrita do dado chamado de IrDA_WRITE_ADDRESS como mostrado a seguir:

```
IR_TRANSMITTER IrDAtransmitter
(
    .iCLK(iCLK_50),                     //clk 50MHz
    .iRST_n(Reset),                     //reset
    .iDATA(txd_data),                   //data input
    .iTXD_READY(txd_start),             //start bit
    .oTXD_BUSY(txd_busy),               //transmitter sending
    .oIRDA_TXD(oIRDA_TXD)               //transmitter coded signal
);

always @(posedge iCLK)
    if(wWriteEnable)
        if (wAddress == IrDA_WRITE_ADDRESS)
            txd_data <= wWriteData;
        else if (wAddress == IrDA_CONTROL_ADDRESS)
            txd_start <= wWriteData[0];
```

O módulo transmissor exige portando que se envie uma cadeia de bits txd_data de 32 bits e um sinal de start para início de envio. Como saída este fornece um sinal de busy enquanto ainda está fazendo envio e o sinal para o circuito integrado do infravermelho oIRDA_TXD.

Para que funcione corretamente é necessario anteriormente carregar em IrDA_WRITE_AD-DRESS os dados necessários para envio para que logo em seguida seja setado o bit de envio txd_start. Também é importante que o bit de start seja logo em seguida jogado para LOW para que a cadeia de bits não seja enviada mais de uma vez.

8.3.1 Teste via formas de onda

Para o teste de envio e recepção, o valor de recepção da interface do receptor, foi setado para o valor de envio do transmissor ou seja, o que foi decodificado pelo receptor foi o valor codificado pelo transmissor. Para que fosse possível a simulação no quartus os intervalos anteriormente estipulados foram divididos por 1000 ou seja não mais estamos em *ms* e sim em *us* .O valor escolhido para envio foi o de 0x00FF_FFFF para que os bits respeitem o protocolo de comunicação. Para setar este valor vale observar as seguintes formas de onda:

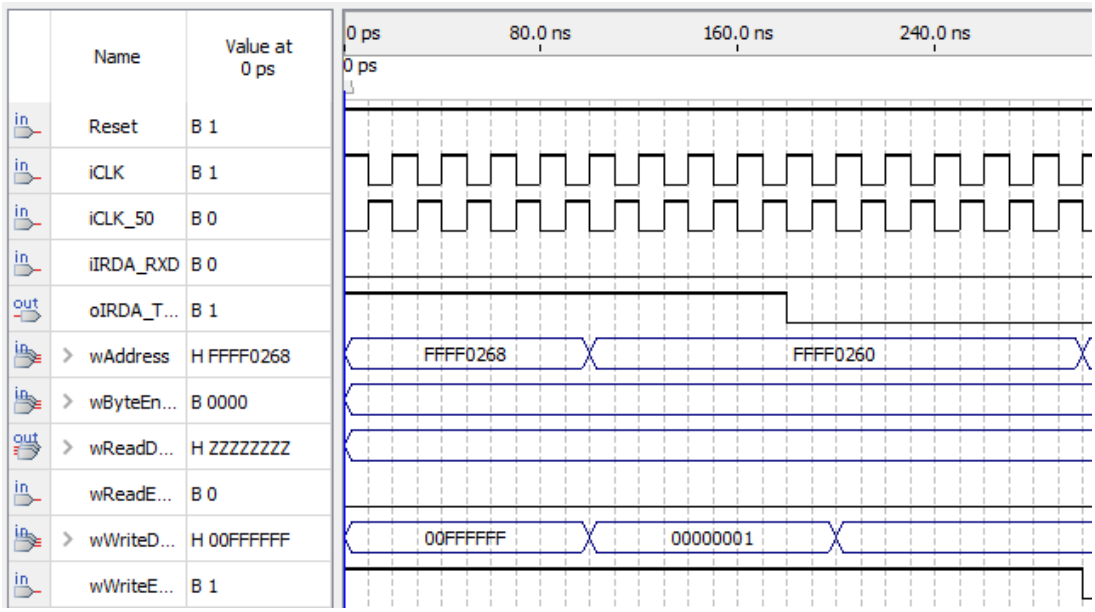

Figura 31: Valores setados para envio

Podemos observar portanto que acima foi necessario informar o endereço para envio de dados IrDA_WRITE_ADDRESS = 0xFFFF_0268 para armazenar o dado 0x00FF_FFFF, logo em seguida foi informado o endereço de controle IrDA_CONTROL_ADDRESS = 0xFFFF_0260 para setar txd_start para 1 e abaixá-lo novamente em seguida para que não seja enviado mais de uma vez o dado. A seguir temos a forma de onda obtida na saída oIRDA_TXD:

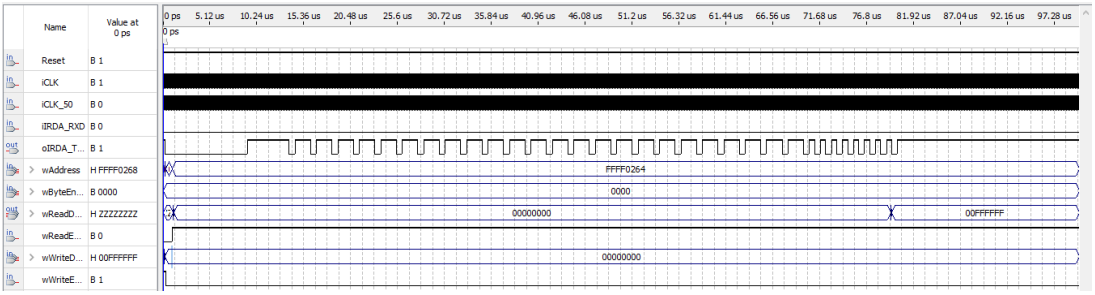

Figura 32: Forma de onda da codificação da cadeia de dados

É possivel observar portanto a similaridade da forma de onda produzida por oIRDA_TXD com a figura do protocolo NEC, aqui podemos observar o estado IDLE em HIGH anteriormente ao inicio do envio de dados, 9us em HIGH e 4.5us em LOW de GUIDANCE e o envio de dados DATASEND com 1.66us para HIGH e 0.52us para LOW, finalizando em END de onde retornamos para IDLE.

Para a recepção o endereço setado foi de IrDA_READ_ADDRESS = 0xFFFF_0264, onde podemos observar logo ao final da transmissão do dado onde teremos o data_ready subindo para HIGH um pouco antes da finalização e o valor esperado que havia sido enviado de 0x00FF_FFFF.

Obs: Apesar de funcionais , o transmissor e o receptor, a integração com o processador mips não foi possibilitada por motivos ainda desconhecidos, porém sabe-se que os padrões funcionam pois como observado nas formas de onda acima produzidas tanto a leitura quanto o envio de dados se comportam como esperado.

8.3.2 Testes criados para processador mips

Para os teste no processador mips foram criados dois programas diferentes, um somente para recepção de sinais e outro somente para envio de sinais. Para ambos foram gerados os .mif para testes nas placas:

```
.data
IrDA_READ_ADDRESS:        .word     0xFFFF0264
IrDA_WRITE_ADDRESS:       .word     0xFFFF0268
IrDA_CONTROL_ADDRESS:     .word     0xFFFF0260
DATA_SEND:                          .word     0xAA55FFFF

.text
        lw $a0, IrDA_CONTROL_ADDRESS
        lw $a1, IrDA_WRITE_ADDRESS
        lw $a2, IrDA_READ_ADDRESS
        lw $a3, DATA_SEND
        j test.irda.receiver
    #j test.irda.transsmiter

test.irda.receiver:
read.control:    lw $t0, 0($a0)
                 beq $t0, $zero, read.control    # esperando Data_ready ir para 31'b0000_0001
                 lw $s0, 0($a2)                  # descarregamos o que foi lido
read.control.1: lw $t0, 0($a0)
                 bne $t0, $zero, read.control.1  # esperamos Data_ready voltar para 0 para proxima leitura
                 add $t1, $zero, $s0
                 j test.irda.receiver

test.irda.transsmiter:
                 sw $a3, 0($a1)                  # descarregamos bits a serem enviados
                 addi $t0, $zero, 1
                 sw $t0, 0($a0)                  # jogamos controle para 31'b0000_0001 txd_start = 1
                 sw $zero, 0($a0)                # abaixamos txd_start para enviar apenas uma vez o dado
read.control2:  lw $t0, 0($a0)
                 bne $t0, $zero, read.control2   # esperamos txd_busy voltar para 0 para proximo envio
                 j test.irda.transsmiter
```

Referências