Arquitetura MIPS

MIPS (*Microprocessor Without Interlocked Pipeline Stages* ou Microprocessador sem Estágios de Pipeline Intertravados) é um arquitetura de microprocessador RISC (*Reduced Instructions Set Computer* ou Computador de Conjunto de Instruções reduzidos).

Arquiteturas geralmente possuem diversas *microarquiteturas*.
Foi utilizado como referencial o capítulo 7 do livro Digital Design and Computer Architecture (David Money Harris & Sarah L. Harris).

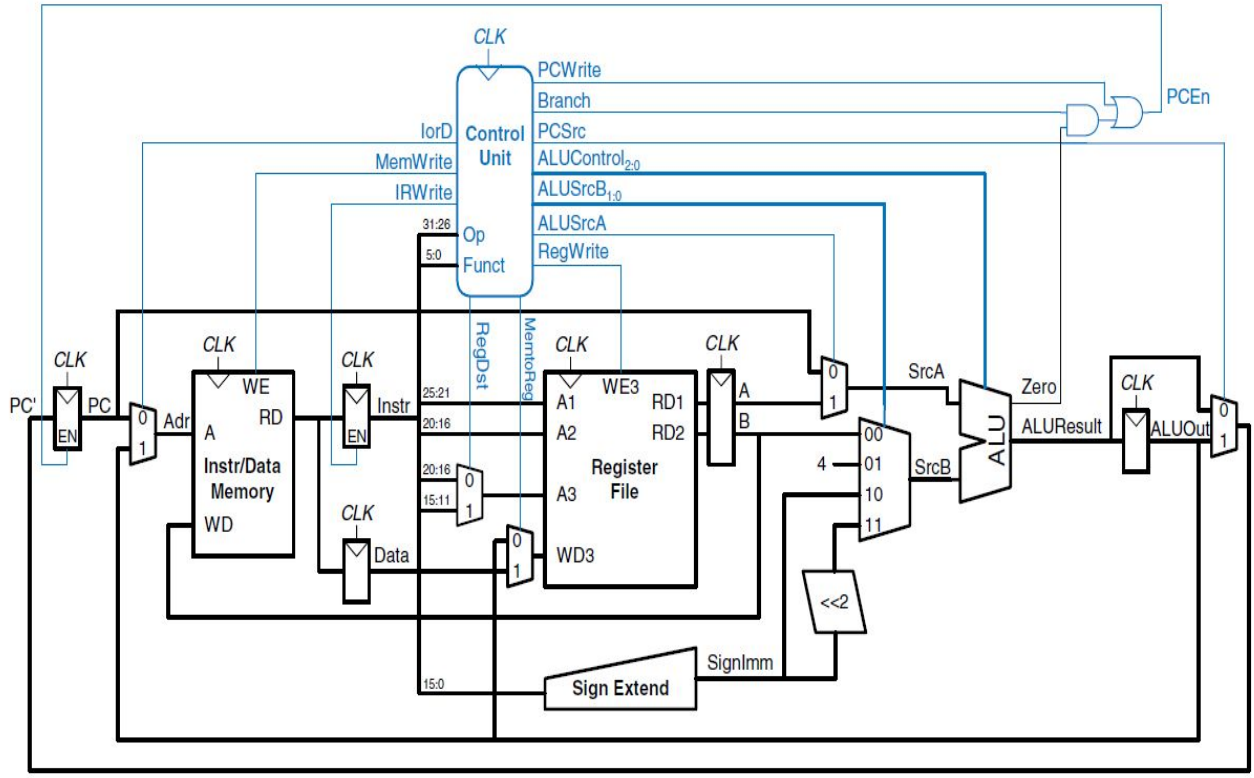
Microarquitetura

Uma *Microarquitetura* trata-se da conexão entre a lógica e a arquitetura, definindo como devem estar arranjados todos os registradores, ULAs, Máquinas de estado, memórias, além de outros blocos lógicos que fazem parte do processador.

Arquiteturas diferentes possuem abordagens diferentes, com balanço entre desempenho, custo de produção, complexibilidade, etc.

É abordada portanto, a Microarquitetura *Multiciclo* do MIPS, no qual as operações do processador são realizadas em série, com vários ciclos curtos de *clock* (relógio), assim, instruções mais curtas são realizadas em menos ciclos. Nessa abordagem, um bloco pode ser usado várias vezes, tendo em mente a natureza cíclica do processador, reduzindo o espaço físico necessário.

A seguir temos o diagrama completo do processador *MIPS* Multiciclo sendo em azul, a Unidade de Controle, em preto, o Caminho de dados (datapath):



Datapath

Datapath ou Caminho de Dados é o local onde estão as unidades funcionais do processador, como ULAs e é responsável pelo processamento dos dados.

Dentre os recursos do *Datapath* destacamos:

- Armazena as instruções do programa (registrador de instrução);
- Armazena o endereço das instruções a serem lidas na memória, um contador de programa (PC);
- Somador de incremento do PC;

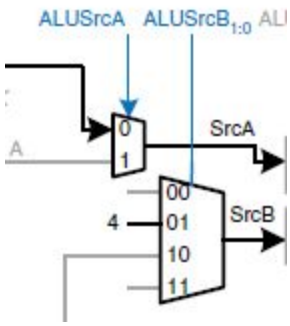
Control Unit

Control Unit ou Unidade de Controle é a parte do MIPS responsável por controlar todos os blocos do Caminho de dados, controlando então como será realizado o fluxo dos dados no processador. Sendo de extrema importância o funcionamento correto desta unidade para o processamento do MIPS.

Blocos do Datapath

Em relação aos blocos do *Datapath* serão abordados a seguir as implementações feitas na linguagem C, juntamente com os devidos arquivos gerados pelo GENPAT demonstrando os testes e simulações feitas.

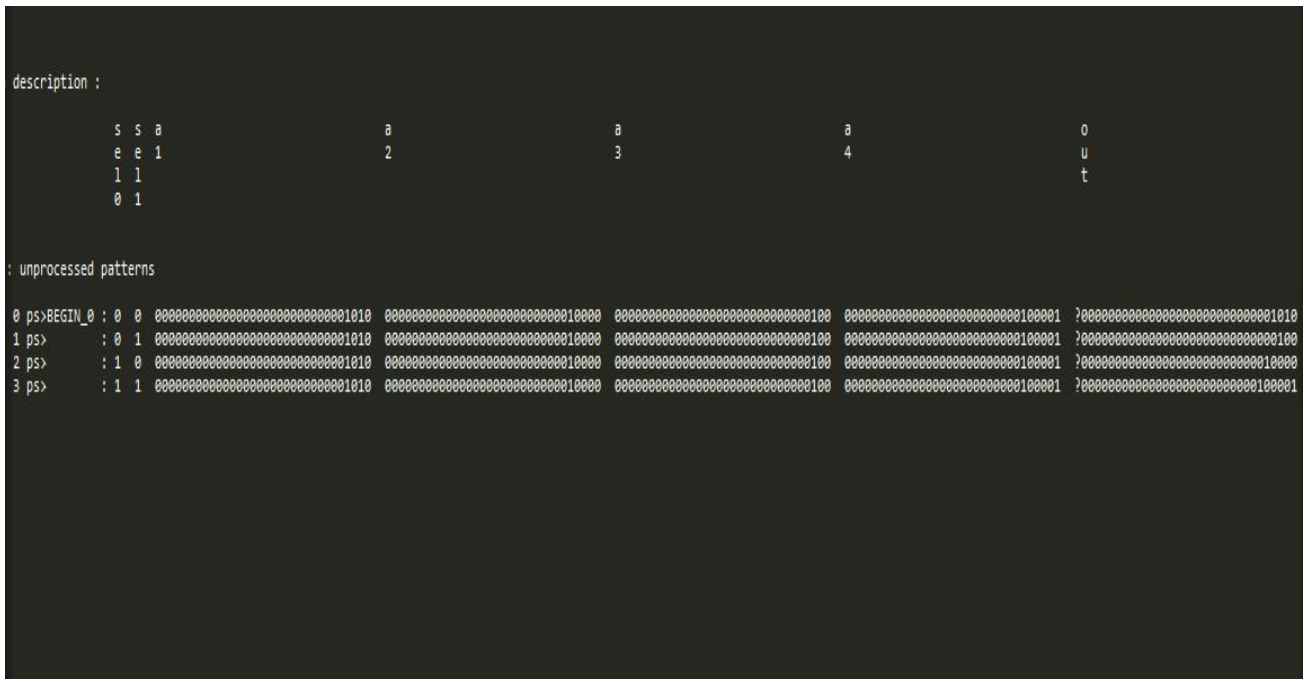
Multiplexadores:



Foram utilizados dois tipos de multiplexadores: Mux de 2 entradas para 1 saída, e outro Mux com 4 entradas e 1 saída. Segue abaixo a implementação que se localiza no arquivo 'Mux.h'.

```
1  int Mux_2_1(int a_0, int a_1, int s)
2  {
3    if(s == 0)
4      return a_0;
5    else return a_1;
6  }
7
8  int Mux_4_1(int a_0, int a_1, int a_2, int a_3, int *S)
9  {
10   int y_0, y_1;
11   y_0 = Mux_2_1(a_0, a_1, S[0]);
12   y_1 = Mux_2_1(a_2, a_3, S[0]);
13   return Mux_2_1(y_0, y_1, S[1]);
14 }
```

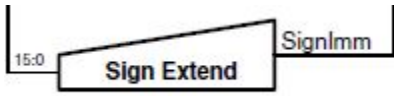
O arquivo ‘mux_test.c’ realiza a inserção de entradas como ferramentas de teste, resultando no arquivo ‘mux_test.pat’ com a seguinte saída:



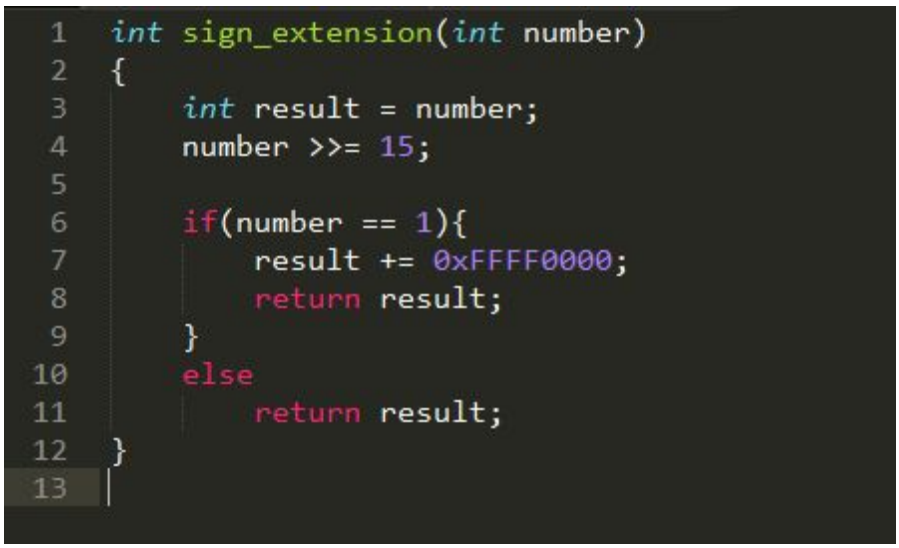
Para mais detalhes e melhor visualização, consultar o arquivo ‘mux_test.pat’.

Sign Extend:

Sign Extend (Extensor de Sinal) trata-se de um bloco utilizado no MIPS em instruções do tipo I, ou seja, load word e store word. Sua função é estender o sinal de entrada de 16 bits para um sinal de saída composto por 32bits.



No arquivo ‘Sign_Extended.h’ temos sua implementação em forma de função:



```
1  int sign_extension(int number)
2  {
3      int result = number;
4      number >>= 15;
5
6      if(number == 1){
7          result += 0xFFFF0000;
8          return result;
9      }
10     else
11         return result;
12 }
13
```

O arquivo ‘Sign_Extend.c’ realiza a chamada do header passando o parâmetro necessário ao chamar a função ‘sign_extension()’, como demonstrado a seguir:

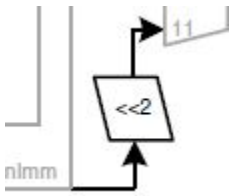
```
25
26  int main()
27  {
28      DEF_GENPAT("Sign_Extend");
29      int i;
30      int size_16 = 65535;
31
32      DECLAR("in",":2","B",IN,"15 downton 0","");
33      DECLAR("Signlmm",":2","B",OUT,"31 downton 0","");
34      DECLAR("vdd",":2","B",IN,"","");
35      DECLAR("vss",":2","B",IN,"","");
36
37      AFFECT("0","vdd","0b1");
38      AFFECT("0","vss","0b0");
39
40      for(i=0; i <= size_16; i++){
41
42          AFFECT(inttostr(cur_vect),"in",inttostr(i));
43          LABEL("sinal");
44          R = sign_extension(i);
45          AFFECT(inttostr(cur_vect),"Signlmm",inttoHstr(R));
46          cur_vect++;
47
48      }
49
50      printf("\nFIM\n");
51      SAV_GENPAT();
52  }
53
```

Com isso, o arquivo ‘sign_extended.pat’ é gerado e resultando em:

```
-- Pattern description :

--                          i               s                           v  v
--                          n               i                           d  s
--                                          g                           d  s
--                                          n
--                                          l
--                                          m
--                                          m


-- Beware : unprocessed patterns

<          0 ps>sinal_0     : 0000000000000000  ?00000000000000000000000000000000  1  0  ;
<          1 ps>sinal_1     : 0000000000000001  ?00000000000000000000000000000001  1  0  ;
<          2 ps>sinal_2     : 0000000000000010  ?00000000000000000000000000000010  1  0  ;
<          3 ps>sinal_3     : 0000000000000011  ?00000000000000000000000000000011  1  0  ;
<          4 ps>sinal_4     : 0000000000000100  ?00000000000000000000000000000100  1  0  ;
<          5 ps>sinal_5     : 0000000000000101  ?00000000000000000000000000000101  1  0  ;
<          6 ps>sinal_6     : 0000000000000110  ?00000000000000000000000000000110  1  0  ;
<          7 ps>sinal_7     : 0000000000000111  ?00000000000000000000000000000111  1  0  ;
<          8 ps>sinal_8     : 0000000000001000  ?00000000000000000000000000001000  1  0  ;
<          9 ps>sinal_9     : 0000000000001001  ?00000000000000000000000000001001  1  0  ;
<         10 ps>sinal_10    : 0000000000001010  ?00000000000000000000000000001010  1  0  ;
<         11 ps>sinal_11    : 0000000000001011  ?00000000000000000000000000001011  1  0  ;
<         12 ps>sinal_12    : 0000000000001100  ?00000000000000000000000000001100  1  0  ;
<         13 ps>sinal_13    : 0000000000001101  ?00000000000000000000000000001101  1  0  ;
<         14 ps>sinal_14    : 0000000000001110  ?00000000000000000000000000001110  1  0  ;
<         15 ps>sinal_15    : 0000000000001111  ?00000000000000000000000000001111  1  0  ;
<         16 ps>sinal_16    : 0000000000010000  ?00000000000000000000000000010000  1  0  ;
<         17 ps>sinal_17    : 0000000000010001  ?00000000000000000000000000010001  1  0  ;
<         18 ps>sinal_18    : 0000000000010010  ?00000000000000000000000000010010  1  0  ;
```

Shifter

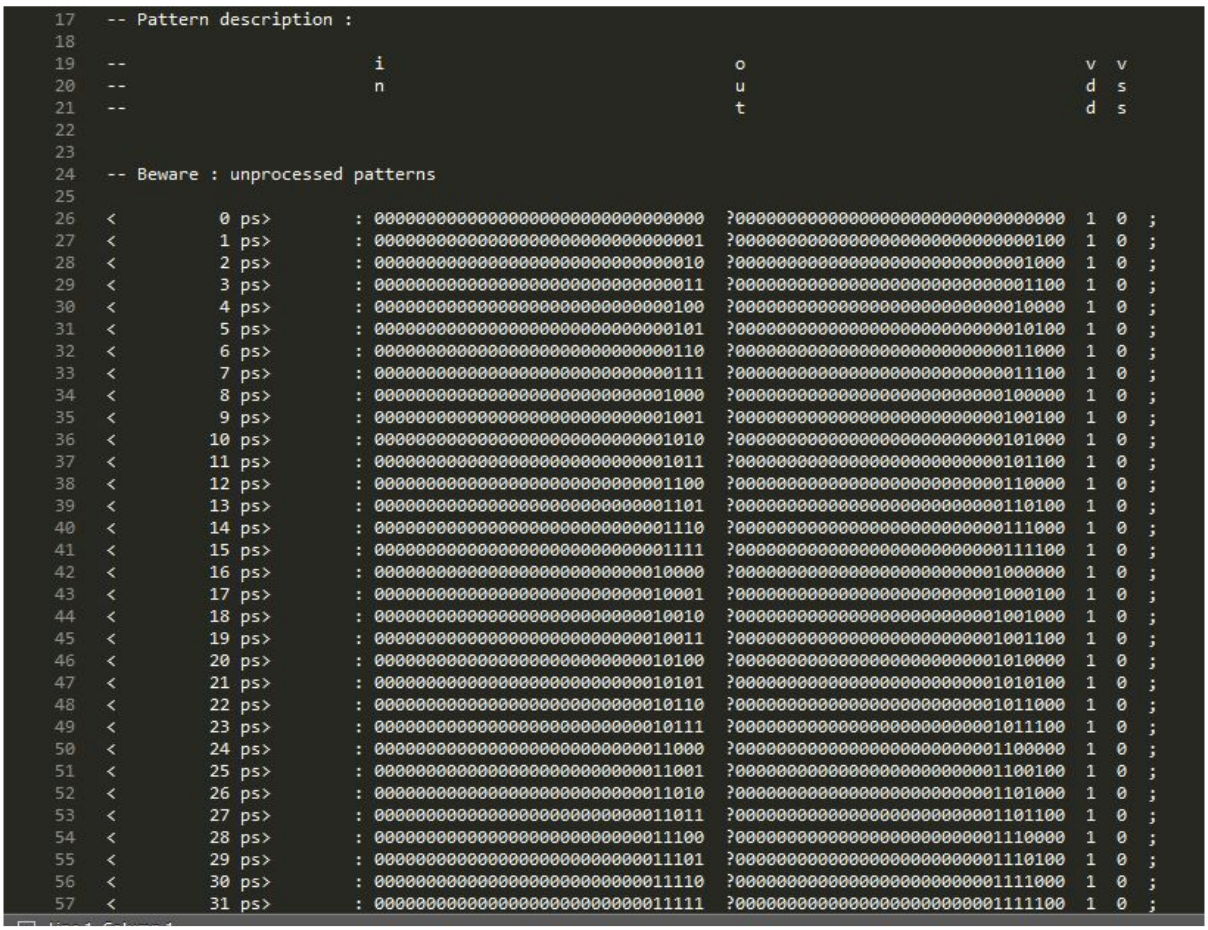
Bloco utilizado na instrução *beq*. Este bloco é responsável por realizar um deslocamento bit-a-bit, no caso um deslocamento de 2 bits da entrada para a esquerda, o que equivale a uma multiplicação por 4.



Sua implementação é dada por:

```c
/*Função Deslocador*/
int shift2(int s){

    return s << 2;
}

```

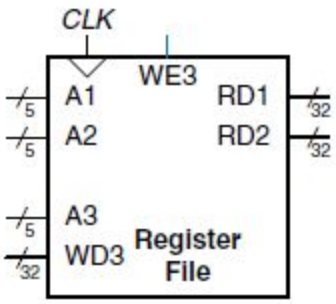
Resultando no seguinte arquivo .pat:



```
-- Pattern description :

--                        i                                o                                v  v
--                        n                                u                                d  s
--                                                         t                                d  s


-- Beware : unprocessed patterns

<          0 ps>        : 00000000000000000000000000000000  ?00000000000000000000000000000000  1  0  ;
<          1 ps>        : 00000000000000000000000000000001  ?00000000000000000000000000000100  1  0  ;
<          2 ps>        : 00000000000000000000000000000010  ?00000000000000000000000000001000  1  0  ;
<          3 ps>        : 00000000000000000000000000000011  ?00000000000000000000000000001100  1  0  ;
<          4 ps>        : 00000000000000000000000000000100  ?00000000000000000000000000010000  1  0  ;
<          5 ps>        : 00000000000000000000000000000101  ?00000000000000000000000000010100  1  0  ;
<          6 ps>        : 00000000000000000000000000000110  ?00000000000000000000000000011000  1  0  ;
<          7 ps>        : 00000000000000000000000000000111  ?00000000000000000000000000011100  1  0  ;
<          8 ps>        : 00000000000000000000000000001000  ?00000000000000000000000000100000  1  0  ;
<          9 ps>        : 00000000000000000000000000001001  ?00000000000000000000000000100100  1  0  ;
<         10 ps>        : 00000000000000000000000000001010  ?00000000000000000000000000101000  1  0  ;
<         11 ps>        : 00000000000000000000000000001011  ?00000000000000000000000000101100  1  0  ;
<         12 ps>        : 00000000000000000000000000001100  ?00000000000000000000000000110000  1  0  ;
<         13 ps>        : 00000000000000000000000000001101  ?00000000000000000000000000110100  1  0  ;
<         14 ps>        : 00000000000000000000000000001110  ?00000000000000000000000000111000  1  0  ;
<         15 ps>        : 00000000000000000000000000001111  ?00000000000000000000000000111100  1  0  ;
<         16 ps>        : 00000000000000000000000000010000  ?00000000000000000000000001000000  1  0  ;
<         17 ps>        : 00000000000000000000000000010001  ?00000000000000000000000001000100  1  0  ;
<         18 ps>        : 00000000000000000000000000010010  ?00000000000000000000000001001000  1  0  ;
<         19 ps>        : 00000000000000000000000000010011  ?00000000000000000000000001001100  1  0  ;
<         20 ps>        : 00000000000000000000000000010100  ?00000000000000000000000001010000  1  0  ;
<         21 ps>        : 00000000000000000000000000010101  ?00000000000000000000000001010100  1  0  ;
<         22 ps>        : 00000000000000000000000000010110  ?00000000000000000000000001011000  1  0  ;
<         23 ps>        : 00000000000000000000000000010111  ?00000000000000000000000001011100  1  0  ;
<         24 ps>        : 00000000000000000000000000011000  ?00000000000000000000000001100000  1  0  ;
<         25 ps>        : 00000000000000000000000000011001  ?00000000000000000000000001100100  1  0  ;
<         26 ps>        : 00000000000000000000000000011010  ?00000000000000000000000001101000  1  0  ;
<         27 ps>        : 00000000000000000000000000011011  ?00000000000000000000000001101100  1  0  ;
<         28 ps>        : 00000000000000000000000000011100  ?00000000000000000000000001110000  1  0  ;
<         29 ps>        : 00000000000000000000000000011101  ?00000000000000000000000001110100  1  0  ;
<         30 ps>        : 00000000000000000000000000011110  ?00000000000000000000000001111000  1  0  ;
<         31 ps>        : 00000000000000000000000000011111  ?00000000000000000000000001111100  1  0  ;
```

Registradores:

O registrador de arquivos trata-se de um banco de registradores tem o objetivo de guardar informações dentro do MIPS sendo constituído por 32 registradores. A entrada de endereçamento é composta por 5 bits, como representado abaixo:



No arquivo ‘Reg.h’ temos a implementação de dois registradores utilizados.

```c
int e=0xFFFFFFFF;
int Registrador0(int clock, int clockA, int reset, int enable, int dado){
    int l0;
    if (clock==1 && clockA == 0){
        if (reset == 1){
            l0 = 0x00000000;
        }
        else if (enable == 1){
            l0 = dado;

        }
    }
    return l0;
}
int Registrador(int clock, int clockA, int reset, int enable, int dado){

    if (clock==1 && clockA == 0){
        if (reset == 1){
            e = 0x00000000;
        }
        else if (enable == 1){
            e = dado;

        }
    }
    return e;
}
```

No arquivo ‘Register_File.c’ encontra-se a função ‘TestarRegistrador()’ que será utilizada na função principal do mesmo para realizar os devidos testes.

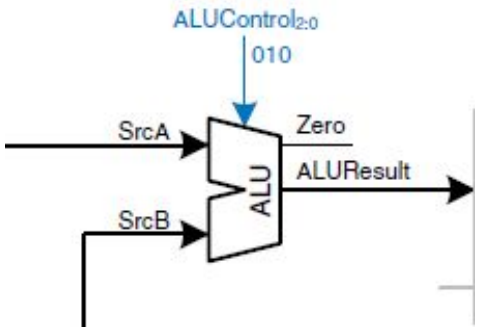
```
void TestarRegistrador()
{
    int Reg_a, Reg_b;
    AFFECT (inttostr(cur_vect), "clk", inttostr(clk) );
    AFFECT (inttostr(cur_vect), "rst", inttostr(rst) );
    AFFECT (inttostr(cur_vect), "WD3", inttoHstr(D) );
    AFFECT (inttostr(cur_vect), "WE3", inttostr(En) );


    /*verifica se o registrador selecionado foi o ReG_0*/
    if(c[0] == 1){ /*atribui zero nas entradas do mux*/
        for(n = 0; n < 32; n++)
            a[n] = 0x00000000;

        Reg_a = Mux_32_1(ENTRADA);
        Reg_b = Mux_32_1(ENTRADA1);

        /*Chama os registradores de saida e joga o valor lá*/
        AFFECT (inttostr(cur_vect), "Reg_A", inttoHstr(Registrador0(clk,clka,rst,En,Reg_a)) );
        AFFECT (inttostr(cur_vect), "Reg_B", inttoHstr(Registrador0(clk,clka,rst,En,Reg_b)) );
    }
    else{
        /*joga o dado em todos os 32 registradores e apenas o que mais estiver habilitado pra
         gravar é que fixa o valor*/
        for(n = 0; n < 32; n++)
            a[n] = Registrador(clk,clka,rst,En,D);

        Reg_a = Mux_32_1(ENTRADA);
        Reg_b = Mux_32_1(ENTRADA1);

        AFFECT(inttostr(cur_vect), "Reg_A", inttoHstr(Registrador(clk,clka,rst,En,Reg_a)));
        AFFECT(inttostr(cur_vect), "Reg_B", inttoHstr(Registrador(clk,clka,rst,En,Reg_b)));
    }
    clka = clk;
    clk = !clk;
    cur_vect++;
}
```

Como resultado temos o arquivo ‘register_file.pat’ gerado utilizando a ferramenta GENPAT.

```
-- Pattern description :

--                                a     a     a     c  r  w                        w  r
r                              v  v
--                                1     2     3     l  s  d                        e  e
e                              d  s
--                                                   k  t  3                        3  g
g                              d  s
--                                                                                     _
_
--                                                                                     a
b


-- Beware : unprocessed patterns

<         0 ps>Seleciona_Reg_0_0    : 00000   00000   00000   0  1  11111111111111111111111111111111  0  ?00000000000000000000000000000000
?00000000000000000000000000000000  1  0  ;
<         1 ps>EN_0_1               : 00000   00000   00000   1  1  11111111111111111111111111111111  0  ?00000000000000000000000000000000
?00000000000000000000000000000000  1  0  ;
<         2 ps>D_0_2                : 00000   00000   00000   0  0  00000000000000000000000000000000  0  ?00000000000000000000000000000000
?00000000000000000000000000000000  1  0  ;
<         3 ps>                     : 00000   00000   00000   1  0  00000000000000000000000000000000  0  ?00000000000000000000000000000000
?00000000000000000000000000000000  1  0  ;
<         4 ps>EN_1_4               : 00000   00000   00000   0  0  11111111111111111111111111111111  1  ?00000000000000000000000000000000
?00000000000000000000000000000000  1  0  ;
<         5 ps>                     : 00000   00000   00000   1  0  11111111111111111111111111111111  1  ?00000000000000000000000000000000
?00000000000000000000000000000000  1  0  ;
<         6 ps>D_0_6                : 00000   00000   00000   0  0  00000000000000000000000000000000  1  ?00000000000000000000000000000000
?00000000000000000000000000000000  1  0  ;
<         7 ps>                     : 00000   00000   00000   1  0  00000000000000000000000000000000  1  ?00000000000000000000000000000000
?00000000000000000000000000000000  1  0  ;
<         8 ps>Seleciona_Reg_1_8    : 00001   00001   00001   0  1  11111111111111111111111111111111  0  ?11111111111111111111111111111111
?11111111111111111111111111111111  1  0  ;
<         9 ps>EN_0_9               : 00001   00001   00001   1  1  11111111111111111111111111111111  0  ?00000000000000000000000000000000
?00000000000000000000000000000000  1  0  ;
<        10 ps>D_0_10               : 00001   00001   00001   0  0  00000000000000000000000000000000  0  ?00000000000000000000000000000000
?00000000000000000000000000000000  1  0  ;
<        11 ps>                     : 00001   00001   00001   1  0  00000000000000000000000000000000  0  ?00000000000000000000000000000000
?00000000000000000000000000000000  1  0  ;
```

ULA:

A Unidade Lógica Aritmética recebe dois operandos, SrcA e SrcB. SrcA vem do registrador de arquivo e SrcB do sinal estendido imediato. Ela pode realizar muitas operações que serão especificadas pelo sinal ALUControl de 3 bits. Gera o sinal ALUResult de 32 bits e uma flag Zero que indica quando o ALUResult é igual a zero. Segue a visualização:



A implementação da ULA encontra-se no header ‘ULA.h’ mostrada a seguir:

```
int AddSub(int op){
    if(op == 6 || op == 7) return 1;
    else return 0;
}

int SomaSub(int a, int b,int addsub, int op){
    long int AluResult;
    switch (op){
        case 0: return a & b;
        case 1: return a | b;
        case 2:
        case 6:
            if (addsub) b = ~b;
            AluResult = (a + b + addsub) & 0xFFFFFFFF;
            return AluResult;
        case 4: return a & ~b;
        case 5: return a | ~b;
        case 7: return a < b;
        default:
            break;
    }
}

int Zero(int resultado){
    return !resultado;
}

int Cout(unsigned int a, unsigned int b, unsigned int addsub, unsigned int resultado){
    if(!addsub)
        if(((a <= 0x7FFFFFFF) && (b <= 0x7FFFFFFF) && (resultado > 0x7FFFFFFF)) ||
            ((a > 0x7FFFFFFF) && (b > 0x7FFFFFFF) && (resultado <= 0x7FFFFFFF)))
            return 1;
        else
            return 0;
    else
        if(((a <= 0x7FFFFFFF) && (b > 0x7FFFFFFF) && (resultado > 0x7FFFFFFF)) ||
            ((a > 0x7FFFFFFF) && (b <= 0x7FFFFFFF) && (resultado <= 0x7FFFFFFF)))
            return 1;
    else
            return 0;
}

```

O arquivo ‘ULA_32.c’ utiliza da função ‘TestarUla()’ para fazer os devidos testes:

```
28
29  void TestarUla(unsigned int a, unsigned int b, int op){
30              printf ("Entrada: SrcA = %u, SrcB = %u, op = %d \n", a, b, op);
31              AFFECT (inttostr(cur_vect), "SrcA", inttoHstr(a) );
32              AFFECT (inttostr(cur_vect), "SrcB", inttoHstr(b) );
33              AFFECT (inttostr(cur_vect), "AluResult", inttoHstr(SomaSub(a,b,AddSub(op),op)) );
34              AFFECT (inttostr(cur_vect), "OVFL", inttostr(Cout(a,b,AddSub(op),SomaSub(a,b,AddSub(op),op))) );
35              AFFECT (inttostr(cur_vect), "Zero", inttostr(Zero(SomaSub(a,b,AddSub(op),op)))  );
36              cur_vect++;
37  }
38  int main(){
```

Resultando assim no arquivo ‘ula_32.pat’, mostrado a seguir:

```
21  -- Pattern description :
22
23  --                              o    s        s        a         o   z  v  v
24  --                              p    r        r        l         v   e  d  s
25  --                                   c        c        u         f   r  d  s
26  --                                   a        b        r         l   o
27  --                                                     e
28  --                                                     s
29  --                                                     u
30  --                                                     l
31  --                                                     t
32
33
34  -- Beware : unprocessed patterns
35
36  <          0 ps>ADD_0           : 010  00000000  00000000  ?00000000  ?0  ?1  1  0  ;
37  <          1 ps>                : 010  00000000  7fffffff  ?7fffffff  ?0  ?0  1  0  ;
38  <          2 ps>                : 010  7fffffff  00000000  ?7fffffff  ?0  ?0  1  0  ;
39  <          3 ps>                : 010  7ffffffe  00000001  ?7fffffff  ?0  ?0  1  0  ;
40  <          4 ps>                : 010  7fffffff  7fffffff  ?fffffffe  ?1  ?0  1  0  ;
41  <          5 ps>                : 010  8fffffff  00000001  ?90000000  ?0  ?0  1  0  ;
42  <          6 ps>                : 010  8fffffff  8fffffff  ?1ffffffe  ?1  ?0  1  0  ;
43  <          7 ps>                : 010  ffffffff  7fffffff  ?7ffffffe  ?0  ?0  1  0  ;
44  <          8 ps>SUB_8           : 110  00000000  00000000  ?00000000  ?0  ?1  1  0  ;
45  <          9 ps>                : 110  00000000  7fffffff  ?80000001  ?0  ?0  1  0  ;
46  <         10 ps>                : 110  7fffffff  00000000  ?7fffffff  ?0  ?0  1  0  ;
47  <         11 ps>                : 110  7ffffffe  00000001  ?7ffffffd  ?0  ?0  1  0  ;
48  <         12 ps>                : 110  7fffffff  7fffffff  ?00000000  ?0  ?1  1  0  ;
49  <         13 ps>                : 110  8fffffff  00000001  ?8ffffffe  ?0  ?0  1  0  ;
```

Instruction and Data Memory:

Este componente não representa um bloco presente na Microarquitetura Multiciclo, uma vez que a memória não faz parte do processador, e trata-se de um componente de acesso externo. Contudo,

como a leitura e escrita é uma etapa de fundamental importância para o MIPS, foi implementada uma função que realiza este processo.

A leitura e escrita é realizada em um único arquivo denominado "memoria.txt", representando assim uma memória compartilhada.

No arquivo ‘instr_data_mem.h’ temos a implementação da função ‘instr_data_Memory()’ que de acordo com os parâmetros usados realiza a leitura ou escrita no arquivo .txt.

```
30  void instr_data_Memory(int clock, int clockA, int A, int MemWrite, int WD) {
31      FILE *instrucoes;
32
33      if (MemWrite == 0) {
34          instrucoes= fopen("./memoria.txt", "r");
35          if(instrucoes){
36              int linha = 0, j=1,cod1=0,k;
37              int i, funct, opcode, pot, rst= 1;
38              char cod[32];
39              printf("ARQUIVO ABERTO \n");
40
41              while(!feof(instrucoes)){
42                  fscanf(instrucoes, "%s", cod);
43                  if (linha == A) {
44                      printf("%d%c instrucao: %s\n", linha, '@', cod);
45                      funct = 0;
46                      opcode = 0;
47                      pot = 1;
48
49                      for (i=5; i>=0; i--) {
50                          opcode += pot*(cod[i] - 48) ;
51                          funct += pot*(cod[26+i] - 48) ;
52                          pot = pot*2;
53                      }
54
55                      for(k=31;k>=0;k--) {
56                          cod1 += j*(cod[k]-48);
57                          j=j*2;
58                      }
59
60                      printf("Opcode: %d, Funct: %d, Cod1: %d\n\n", opcode, funct, cod1);
```

```
61                      //MaquinaDeEstados(rst, opcode, funct,cod1);
62
63                      AFFECT(inttostr(linha), "clk",  inttostr(clock));
64                      AFFECT(inttostr(linha), "A",  inttostr(A));
65                      AFFECT(inttostr(linha), "MemWrite",  inttostr(MemWrite));
66                      AFFECT(inttostr(linha), "WD",  inttostr(WD));
67                      AFFECT(inttostr(linha), "Cod",  inttoHstr(cod1));
68                      AFFECT(inttostr(linha), "Funct",  inttostr(funct));
69                      AFFECT(inttostr(linha), "Opcode",  inttostr(opcode));
70                      LABEL("Leitura");
71                      j=1;
72                      cod1=0;
73                      rst = 0;
74
75                      break;
76                  }
77                  linha++;
78
79              }
80              fclose(instrucoes);
81
82          }
83          else{
84              printf("ERRO AO ABRIR ARQUIVO \n");
85          }
86      }
87      else {
88          if(clock == 1 && clockA == 0){
89              LABEL("Escrita");
90
91              FILE *temp;
92
93              instrucoes = fopen("./memoria.txt", "r+");
```

```
94              temp = fopen("./~memoria.txt", "w");
95
96              if(instrucoes && temp){
97
98                  char aux[255];
99                  int cont = 0;
100
101                 while(fgets(aux, sizeof(aux), instrucoes)){
102                     if(cont == A - 1) {
103                         int ik;
104                         int aux[32];
105
106                         for(ik = 0; ik < 32; ++ik){
107                             aux[ik] = Pow((WD & 0b01), ik);
108                             WD >>= 1;
109                         }
110
111                         for(ik = 31; ik >= 0; --ik)
112                             fprintf(temp, "%i", aux[ik]);
113
114                         fprintf(temp, "\n");
115
116                     } else
117                         fprintf(temp, "%s", aux);
118
119                     cont++;
120                 }
121                 fclose(temp);
122                 fclose(instrucoes);
123
124                 char ch;
125
126                 temp = fopen("./~memoria.txt", "r");
```

```
                instrucoes = fopen("./memoria.txt", "w");

                while(1){
                    ch = fgetc(temp);

                    if (ch == EOF)
                        break;
                    else
                        putc(ch, instrucoes);
                }

                fclose(instrucoes);
                fclose(temp);
            }

            else
                printf("ERRO AO ABRIR ARQUIVO \n");
        }
    }
}
```

No arquivo da linguagem C ‘instr_data_mem.c’ temos a implementação dos devidos testes feitos para leitura na memória:

```
int main(){
    DEF_GENPAT("inst_data_mem");


    DECLAR ("clk",          ":2", "B", IN, "", "" );
    DECLAR ("A",          ":2", "B", IN, "3 downto 0", "" );
    DECLAR ("MemWrite",         ":2", "B", IN, "", "" );
    DECLAR ("WD",         ":2", "B", IN, "", "" );
    DECLAR ("Cod",          ":2", "B", OUT, "31 downto 0", "" );
    DECLAR ("Funct",          ":2", "B", OUT, "5 downto 0", "" );
    DECLAR ("Opcode",         ":2", "B", OUT, "5 downto 0", "" );

    DECLAR ("vdd", ":2", "B", IN, "", "" );
    DECLAR ("vss", ":2", "B", IN, "", "" );

    AFFECT ("0", "clk", "0b0");
    AFFECT ("0", "vdd", "0b1");
    AFFECT ("0", "vss", "0b0");

    int a;
    int clock = 0;
    int clockA = 0;
    int FSMMemWrite = 0;
    int wd = 0;

    for(a = 0; a < 8; a++) {
        instr_data_Memory(clock, clockA, a, FSMMemWrite, wd);
    }

    clock = 1;
    clockA = 0;
    FSMMemWrite = 1;
    wd = 0b00010001000100010001000100010001;

    for(a = 10; a < 20; a++)
        instr_data_Memory(clock, clockA, a, FSMMemWrite, wd);

    SAV_GENPAT();
}
```

Como resultado temos o arquivo ‘instr_data_mem.pat’ gerado, demonstrando a quebra correta da instrução:

```
22  -- Pattern description :
23
24  --                          c  a     m  w   c                                f       o       v  v
25  --                          l        e  d   o                                u       p       d  s
26  --                          k        m      d                                n       c       d  s
27  --                                   w                                       c       o
28  --                                   r                                       t       d
29  --                                   i                                               e
30  --                                   t
31  --                                   e
32
33
34  -- Beware : unprocessed patterns
35
36  <          0 ps>Escrita_0 : 1  0000  0  0  ?00000000000000000000000000100000  ?100000  ?000000  1  0  ;
37  <          1 ps>Leitura_1 : 0  0001  0  0  ?00000000000000000000000000100010  ?100010  ?000000  1  0  ;
38  <          2 ps>Leitura_2 : 0  0010  0  0  ?00000000000000000000000000100100  ?100100  ?000000  1  0  ;
39  <          3 ps>Leitura_3 : 0  0011  0  0  ?00000000000000000000000000100101  ?100101  ?000000  1  0  ;
40  <          4 ps>Leitura_4 : 0  0100  0  0  ?00000000000000000000000000101010  ?101010  ?000000  1  0  ;
41  <          5 ps>Leitura_5 : 0  0101  0  0  ?00010000000000000000000000100000  ?100000  ?000100  1  0  ;
42  <          6 ps>Leitura_6 : 0  0110  0  0  ?00100000000000000000000000100000  ?100000  ?001000  1  0  ;
43  <          7 ps>Leitura_7 : 0  0111  0  0  ?00001000000000000000000000100000  ?100000  ?000010  1  0  ;
44
45  end;
```