INSTITUTO FEDERAL DO ESPÍRITO SANTO

ENGENHARIA DE CONTROLE E AUTOMAÇÃO

MATHEUS MESQUITA GOMES MOURA

PEDRO DO AMARAL GONÇALVES

IMPLEMENTAÇÃO EM HARDWARE DE INSTRUÇÃO DE PROCESSADOR ARM MONOCÍCLO

SERRA - ES

2023/2

SUMÁRIO

1 ESQUEMÁTICO DO PROCESSADOR BASE E DESENVOLVIDO

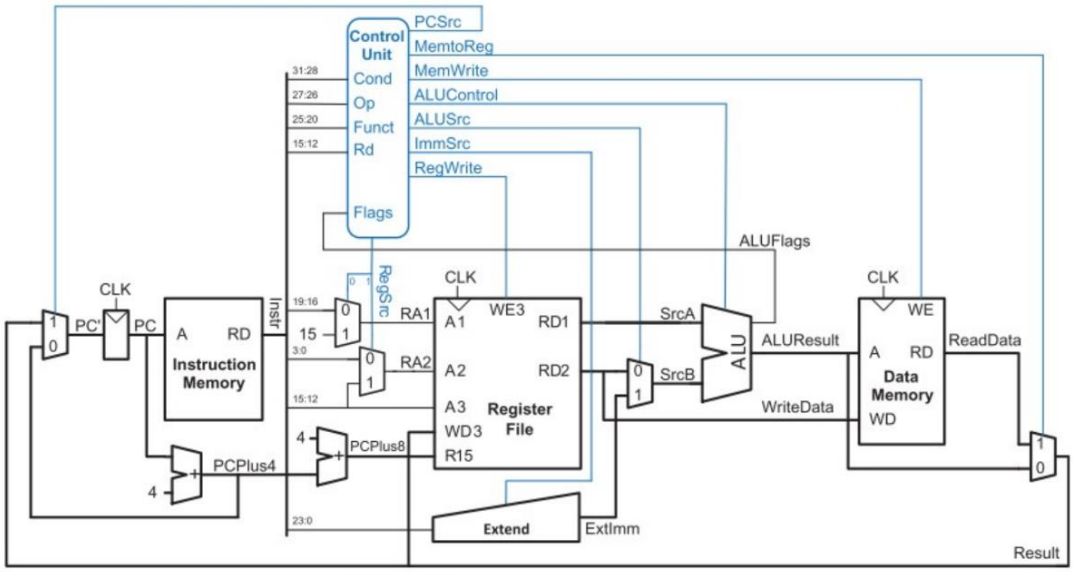
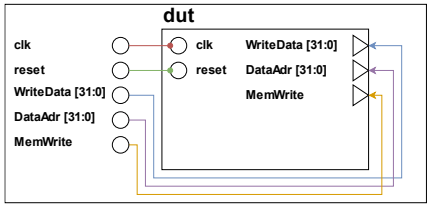
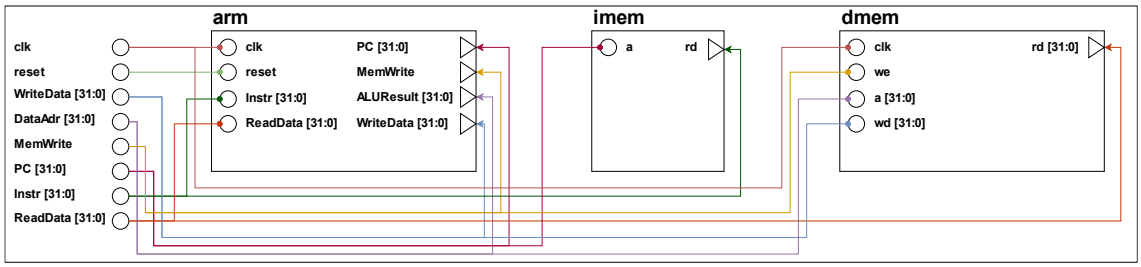


Figure 7.13 Complete single-cycle processor

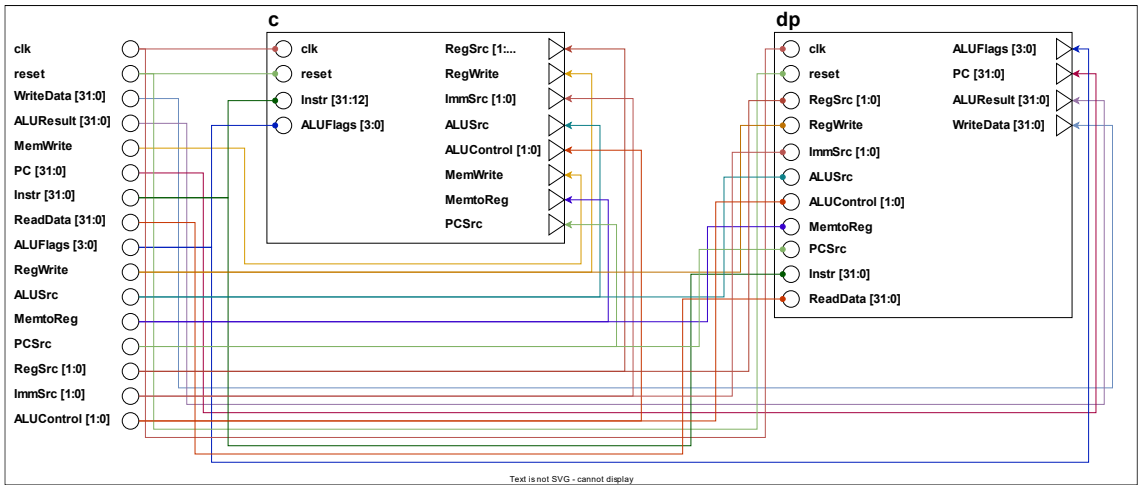
testbench



dut



arm



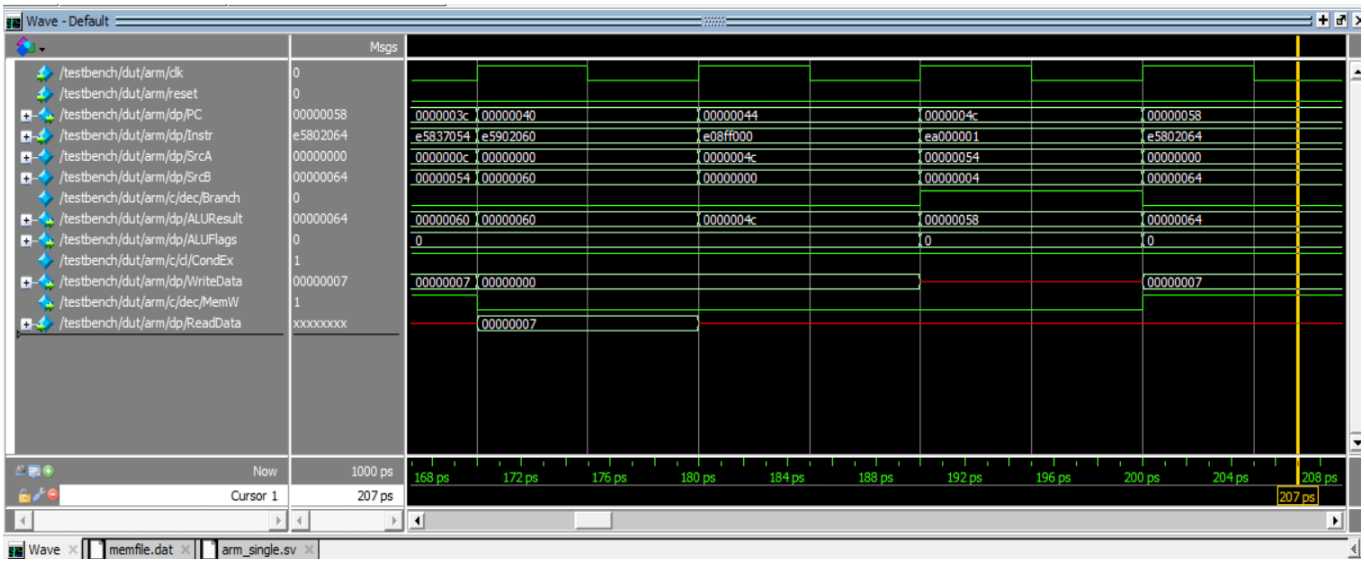
2 SIMULAÇÃO DO PROCESSADOR

2.1 TABELA DE IMPLEMENTAÇÃO

TABELA DE EXECUÇÃO DO ARQUIVO MEMFILE.DAT

Ciclo	Reset	PC	Instrução (Montagem / Máquina)	SrcA	SrcB	Branch	AluResult	Flags3:0	CondEx	WriteData	MemWrite	ReadData
1	1	00000000	E04F000F	00000008	00000008	0	00000000	6	1	00000008	0	xxxxxxxx
2	0	00000004	E2802005	00000000	00000005	0	00000005	0	1	xxxxxxxx	0	xxxxxxxx
3	0	00000008	E280300C	00000000	0000000c	0	0000000c	0	1	xxxxxxxx	0	xxxxxxxx
4	0	0000000c	E2437009	0000000c	00000009	0	00000003	2	1	xxxxxxxx	0	xxxxxxxx
5	0	00000010	E1874002	00000003	00000005	0	00000007	0	1	00000005	0	xxxxxxxx
6	0	00000014	E0035004	0000000c	00000007	0	00000004	0	1	00000007	0	xxxxxxxx
7	0	00000018	E0855004	00000004	00000007	0	0000000b	0	1	00000007	0	xxxxxxxx
8	0	0000001c	E0558007	0000000b	00000003	0	00000008	2	1	00000003	0	xxxxxxxx
9	0	00000020	0A00000C	00000028	00000030	1	00000058	0	0	xxxxxxxx	0	xxxxxxxx
10	0	00000024	E0538004	0000000c	00000007	0	00000005	2	1	00000007	0	xxxxxxxx
11	0	00000028	AA000000	00000030	00000000	1	00000030	0	1	00000000	0	xxxxxxxx
12	0	00000030	E0578002	00000003	00000005	0	fffffffe	8	1	00000005	0	xxxxxxxx
13	0	00000034	B2857001	0000000b	00000001	0	0000000c	0	1	xxxxxxxx	0	xxxxxxxx
14	0	00000038	E0477002	0000000c	00000005	0	00000007	2	1	00000005	0	xxxxxxxx
15	0	0000003c	E5837054	0000000c	00000054	0	00000060	0	1	00000007	1	xxxxxxxx
16	0	00000040	E5902060	00000000	00000060	0	00000060	0	1	00000000	0	00000007
17	0	00000044	E08FF000	0000004c	00000000	0	0000004c	0	1	00000000	0	xxxxxxxx
18	0	0000004c	EA000001	00000054	00000004	1	00000058	0	1	xxxxxxxx	0	xxxxxxxx
19	0	00000058	E5802064	00000000	00000064	0	00000064	0	1	00000007	1	xxxxxxxx

2.2 WAVE DE SIMULAÇÃO



3 IMPLEMENTAÇÕES

3.1 INSTRUÇÃO MOV

Utilizado para mover dados de uma localização de memória para outra, copiando o valor de um operando de origem para um operando de destino, podendo ser usado para transferir dados entre registradores, endereços de memória ou constantes.

Adição do MovF (Flag responsável por definir que a instrução é um MOV) e da instrução MOV (Definida no ALUOp) no decoder.

```
4'b1101:    begin
                        ALUControl = 3'bx; // MOV
NoWrite = 1'b0;
                        MovF = 1'b1;
                end
```

Definição do sinal de saída (MovFlag) através do condlogic presente no controller.

```
assign MovFlag    = MovF  & CondEx;
```

Realização de um mux para uma nova saída (MovORAluResult). Caso MovFlag esteja ativa, SrcB irá direto para o imediato. Também foi necessário realizar a substituição do mux resmux, visto que MovORAluResult é a nova variável responsável por enviar o resultado da ULA ou do SrcB.

```
mux2 #(32)  movmux(ALUResult, SrcB, MovFlag, MovORAluResult);
```

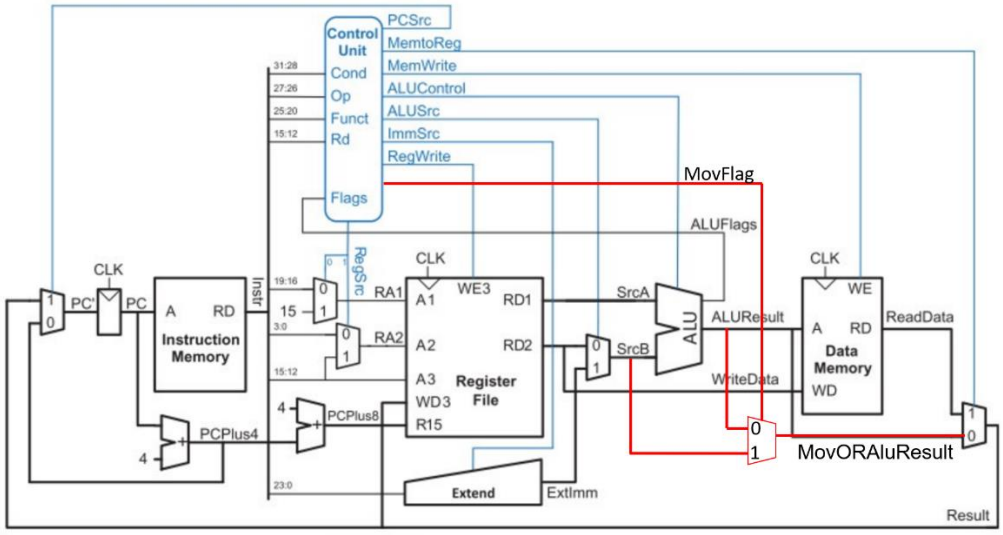


Figure 7.13 Complete single-cycle processor

3.2 INSTRUÇÃO CMP

A instrução CMP é responsável por subtrair o valor do segundo operando do primeiro operando, atualizando as flags de status do processador (como a flag de zero, sinal, carry e overflow) sem armazenar o resultado da subtração em nenhum lugar.

Adição do NoWrite (Flag responsável por não permitir a escrita dos resultados da ULA) e da instrução CMP (Definida no ALUOp) no decoder.

```
4'b1010:  begin
  ALUControl = 3'b001; // CMP
  NoWrite = 1'b1;
  MovF = 1'b0;
end
```

Após isso, essa variável foi implementada do condlogic, sendo responsável por alterar o valor de RegWrite (Sinal que define a escrita no Registrador).

```
assign RegWrite  = RegW  & CondEx & ~NoWrite;
```

Nesse sentido, ao realizar a lógica and com CondEx e RegW estando negada, NoWrite força a saída 0, impossibilitando a escrita no registrador.

3.3 INSTRUÇÃO TST

Verifica se a operação lógica AND entre os bits dos dois operandos resulta em zero ou não, afetando as flags do processador.

Definição do NoWrite (Flag responsável por não permitir a escrita dos resultados da ULA) como 1 e adição da instrução TST (Definida no ALUOp como sendo um AND) no decoder.

```
4'b1000:  begin
  ALUControl = 3'b010; //TST
  NoWrite = 1'b1;
  MovF = 1'b0;
end
```

3.4 INSTRUÇÃO EOR

Executa a operação lógica XOR (OU Exclusivo) entre dois operandos e armazena o resultado no registrador destino.

Expansão do ALUControl para mais um bit e adição da operação XOR na mesma.

```
always_comb
  casex (ALUControl[2:0])
    3'b00?: Result = sum;        //ADD
    3'b010: Result = a & b;    //AND
    3'b011: Result = a | b;    //OR
    3'b100: Result = a ^ b;    //XOR
  endcase
```

Adição da instrução EOR (Definida no ALUOp) no decoder.

```
4'b0001:  begin
  ALUControl = 3'b100; //EOR
  NoWrite = 1'b0;
  MovF = 1'b0;
end
```

3.5 INSTRUÇÃO LDRB

Responsável por carregar um byte (8 bits) de uma memória ou endereço específico para um registrador (Valores de 0x00 até 0Xff).

No código base já há a função LDR. Sendo assim, para a implementação do LDRB foi adicionado uma nova variável de saída do decoder (MaskLDR). Além disso, foi necessária uma expansão da quantidade de bits em controls para o desenvolvimento.

```
always_comb
      case(Op)

        2'b00: if (Funct[5])  controls = 13'b0000101001000; // Data processing immediate

               else           controls = 13'b0000001001000; // Data processing register

        2'b01: if (Funct[0])
              if (Funct[2]) controls = 13'b0001111000100; //LDRB

              else        controls = 13'b0001111000000; // LDR
```

A saída MaskLDR recebe o valor definido em controls, sendo passada para o datapath. Após isso, é realizado uma máscara no ReadData, no qual a saída (ReadData2) recebe a própria leitura do endereço de memória caso MaskLDR seja 0. Se MaskLDR for 1, são enviados apenas os últimos 8 bits do endereço e é forçado o valor 0 nos outros 24 bits mais significativos.

```
//Mascara LDRB
always_comb
  case(MaskLDR)
    1'b0: ReadData2 = ReadData;
    1'b1: ReadData2 = {24'b0, ReadData[7:0]}; //LDRB
  endcase
```

3.6 INSTRUÇÃO STRB

Utilizada para armazenar um byte (8 bits) de um registrador específico em uma localização de memória.

Seguindo o mesmo princípio da função LDRB, foi adicionado uma nova variável de saída no decoder (MaskSTR) e novamente uma expansão da quantidade de bits em controls foi necessária.

```
if(Funct[2]) controls = 13'b1001110100010; //SRTB

else
controls = 13'b1001110100000; // STR
```

A saída MaskSTR recebe o valor definido em controls, sendo passada para o datapath. Após isso, é realizado uma máscara no WriteData, no qual a saída recebe o próprio valor do registrador caso MaskSTR seja 0 (Valor armazenado em WriteData2 retornado através do regfile). Se MaskSTR for 1, WriteData recebe apenas os últimos 8 bits armazenados no registrador e é forçado o valor 0 nos outros 24 bits mais significativos. Com isso, salvando apenas os últimos 8 bits no endereço de memória.

```
//Mascara SRTB
always_comb
  case(MaskSTR)
    1'b0: WriteData = WriteData2;
    1'b1: WriteData = {24'b0, WriteData2[7:0]}; //STRB
  endcase
```

3.7 INSTRUÇÃO BL

Utilizada para realizar chamadas de sub-rotina (funções ou procedimentos) em linguagem de montagem.

Seguindo o mesmo princípio das funções STRB e LDRB, foi adicionado uma nova variável de saída no decoder (BLFlag) e novamente uma expansão da quantidade de bits em controls foi necessária.

```
2'b10: if (Funct[4]) controls = 13'b0110101010001; //BL

       else
       controls = 13'b0110100010000; // B

default:              controls = 12'bx; // Unimplemented
```

Se BLFlag for 1, RegWriteBL (Sinal novo criado para controle do BL) recebe 1.

```
assign RegWriteBL = (BLFlag) ? 1'b1 : RegWrite;
```

No datapath há a criação de um novo mux para controle da entrada em A3. Se BranchLinkado (Saída gerada a partir do controller) for 0, registrador em A3 recebe o valor do registrador destino. Caso BranchLinkado seja 1, é forçado o registrador do Linker Register (Registrador 14) para a escrita de PC + 4.

```
mux2 #(4)    ra3mux(Instr[15:12], 4'b1110, BranchLinkado, RA3);
```

Em regfile, RA3 é passado juntamente com PCPlus4 e BrankLinkado.

```
regfile     rf(clk, PCPlus4, BranchLinkado, RegWrite, RA1, RA2, RA3,
                Result, PCPlus8,
                SrcA, WriteData2);
```

Um novo controle é criado e uma nová variável de controle é criada (WriteDataPC). Nesse sentido, se BrankLinkado for 0, WriteDataPC recebe o valor de Result (wd3). Caso contrário, receberá o valor de PCPlus4 (PC + 4).

```
always_comb
  case(BL)
    1'b0: WriteDataPC = wd3;
    1'b1: WriteDataPC = PC4;
  endcase
```

Por fim, WriteDataPC é passado para registrador destino caso RegWrite esteja ativo.

```
always_ff @(posedge clk)
  if (we3) rf[wa3] <= WriteDataPC;
```

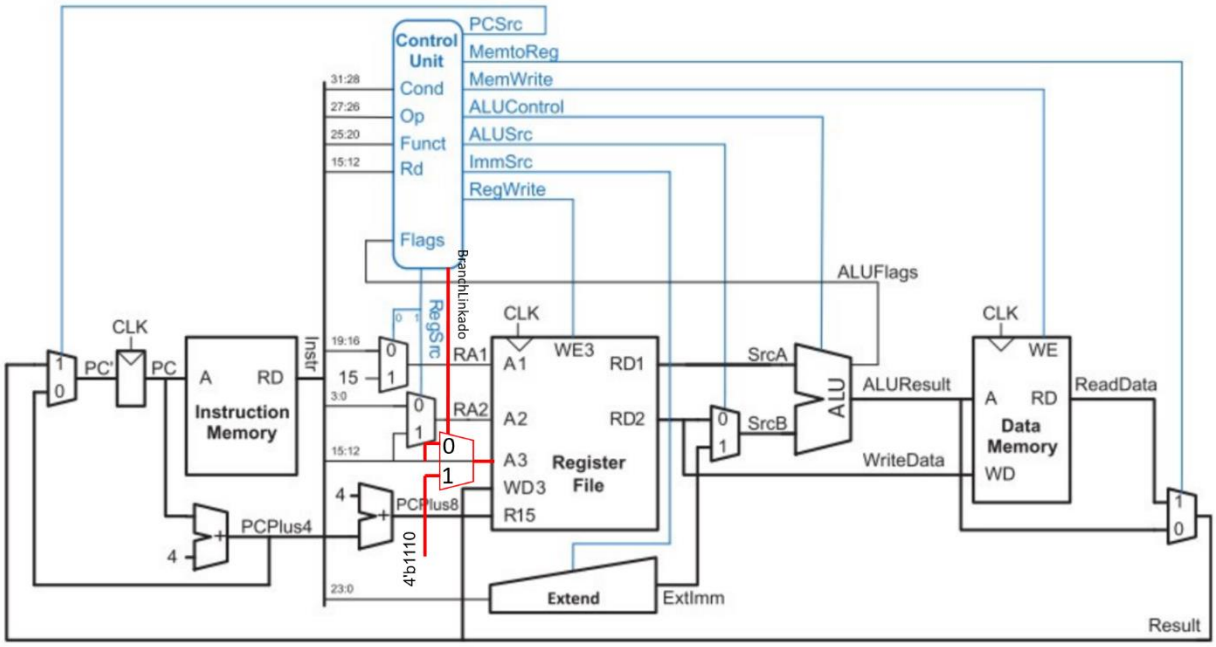


Figure 7.13 Complete single-cycle processor

4 REPOSITÓRIO GIT

https://github.com/Larama47/Trabalho_Arq/tree/main