Contents

1 Informacion general

1.1 Integrantes

- Lautaro Bachmann
- Fabricio Longhi
- Valentino Mensio

1.2 Ejercicios resueltos

- Ejercicio 1
- Ejercicio 2

2 Ejercicio 1

Durante el primer ejercicio se agrego las funcionalidades de ADDI y SUBBI, todas las modificaciones de ambas instrucciones fueron realizadas en los mismos modulos.

2.1 ADDI y SUBI

Para realizar las nuevas implementaciones se tuvieron que modificar las siguientes entidades:

- **aludec.sv:** Se agregaron nuevos condicionales para que ahora se tenga en cuenta las nuevas funcionalidades ADDI y SUBI.

```
 else if((aluop == 2'b11)  & (funct[10:1] == 10'b1001000100)) alucontrol = 4'b0010; //ADI
 else if ((aluop == 2'b10) & (funct[10:1] == 10'b1101000100)) alucontrol = 4'b0110; // SU
```

- **maindec.sv:** Durante este modulo se agrego al decodificador las funcionalidades de ADDI y SUB. Dado el opcode de la nueva instruccion, devuelve las señales correspondietes.

```
// ADDI
11'b1001000100?:begin
    Reg2Loc = 1'b0;
    ALUSrc = 1'b1;
    MemtoReg = 1'b0;
    RegWrite = 1'b1;
    MemRead = 1'b0;
    MemWrite = 1'b0;
    Branch = 1'b0;
    ALUOp = 2'b10;
            end

// SUBI
11'b1101000100?: begin
    Reg2Loc = 1'b0;
    ALUSrc = 1'b1;
    MemtoReg = 1'b0;
    RegWrite = 1'b1;
    MemRead = 1'b0;
    MemWrite = 1'b0;
    Branch = 1'b0;
    ALUOp = 2'b10;
            end
```

- **signext.sv:** En este modulo se añade dentro del casez los opcode de las instrucciones nuevas, el cual determina si es necesario realizar una extension de signo.

```
11'b1001000100?: // ADDI
    y = {{52{a[21]}}, a[21:10]};
        11'b1101000100?: // SUBI
    y = {{52{a[21]}}, a[21:10]};
        default: y = 0;

11'b1001000100?: // ADDI
    y = {{52{a[21]}}, a[21:10]};
        default: y = 0;
```

2.2 Verificaciones

2.2.1 SUBI

En el siguiente fragmento de codigo estamos testeando la nueva instruccion SUBI.

```
.text
.org 0x0000

SUB X29, X15, #15       // X15 - 15
NOP
NOP
STUR X29, [X0, #208]       // MEM 26: 0x0000000000000000


SUB X30, X15, #10         // X15 - 10
NOP
NOP
STUR X30, [X0, #216]       // MEM 27: 0x0000000000000005


SUB X30, X15, #30         // X15 - 30
NOP
NOP
STUR X30, [X0, #224]       // MEM 28: 0xFFFFFFFFFFFFFFF1


finloop: CBZ XZR, finloop
```

2.2.2 ADDI

En el siguiente fragmento de codigo estamos testeando la nueva instruccion ADDI.

```
.text
.org 0x0000
STUR X1, [X0, #0] // MEM 0:0x1
STUR X2, [X0, #8] // MEM 1:0x2
STUR X3, [X16, #0] // MEM 2:0x3
ADD X3, X4, X5
NOP
NOP
STUR X3, [X0, #24] // MEM 3:0x9
SUB X3, X4, X5
NOP
NOP
STUR X3, [X0, #32] // MEM 4:0xFFFFFFFFFFFFFFFF
SUB X4, XZR, X10
NOP
NOP
STUR X4, [X0, #40] // MEM 5:0xFFFFFFFFFFFFFFF6
ADD X4, X3, X4
NOP
NOP
STUR X4, [X0, #48] // MEM 6:0xFFFFFFFFFFFFFFF5
SUB X5, X1, X3
NOP
NOP
STUR X5, [X0, #56] // MEM 7:0x2
AND X5, X10, XZR
NOP
NOP
STUR X5, [X0, #64] // MEM 8:0x0
AND X5, X10, X3
NOP
NOP
STUR X5, [X0, #72] // MEM 9:0xA
AND X20, X20, X20
NOP
NOP
STUR X20, [X0, #80] // MEM 10:0x14
ORR X6, X11, XZR
NOP
NOP
```

```
STUR X6, [X0, #88] // MEM 11:0xB
ORR X6, X11, X3
NOP
NOP
STUR X6, [X0, #96] // MEM 12:0xFFFFFFFFFFFFFFFF
LDUR X12, [X0, #0]
NOP
NOP
NOP
ADD X7, X12, XZR
NOP
NOP
STUR X7, [X0, #104] // MEM 13:0x1
STUR X12, [X0, #112] // MEM 14:0x1
ADD XZR, X13, X14
STUR XZR, [X0, #120] // MEM 15:0x0
CBZ X0, L1
NOP
NOP
NOP
STUR X21, [X0, #128] // MEM 16:0x0(si falla CBZ =21)
L1: STUR X21, [X0, #136] // MEM 17:0x15
ADD X2, XZR, X1
NOP
NOP
L2: SUB X2, X2, X1
ADD X24, XZR, X1
NOP
NOP
STUR X24, [X0, #144] // MEM 18:0x1 y MEM 19=0x1
ADD X0, X0, X8
CBZ X2, L2
NOP
NOP
STUR X30, [X0, #144] // MEM 20:0x1E
ADD X30, X30, X30
SUB X21, XZR, X21
NOP
ADD X30, X30, X20
NOP
NOP
LDUR X25, [X30, #-8]
NOP
```

```
ADD X30, X30, X30
NOP
NOP
ADD X30, X30, X16
NOP
NOP
STUR X25, [X30, #-8] // MEM 21:0xA (= MEM 9)
ADD X0, XZR, XZR
NOP
NOP
ADD X25, X15, #15 // ADDI test case 1
NOP
NOP
STUR X25, [X0, #176] // MEM 22: 0x1E
ADD X26, X15, #20 // ADDI test case 2
NOP
NOP
STUR X26, [X0, #184] // MEM 23: 0x23
ADD X27, X15, #5 // ADDI test case 3
NOP
NOP
STUR X27, [X0, #192] // MEM 24: 0x14
ADD X28, X15, #30 // ADDI test case 4
NOP
NOP
STUR X28, [X0, #200] // MEM 25: 0x2D

finloop: CBZ XZR, finloop
```

---

3 Ejercicio 2

En este ejercicio, se añadieron al microprocesador con pipeline las instrucciones de salto condicional **B.cond** y las instrucciones aritméticas que configuran banderas **ADDS, SUBS, ADDIS y SUBIS**. Estos cambios permiten al microprocesador evaluar y ejecutar saltos condicionales utilizando los estados de las banderas **Zero (Z), Negative (N), Carry (C) y Overflow (V)**.

3.1 Modificaciones realizadas

Las modificaciones se realizaron en varios módulos del microprocesador para implementar las nuevas funcionalidades requeridas:

- **ALU**:
  - Se implementó la resta utilizando el complemento a dos del operando `b` y sumándolo con `a` para asegurar una generación correcta de la bandera **Carry**.
  - Se agregaron las cuatro banderas Z, N, C, y V para representar los resultados de las operaciones.
  - Las operaciones ADDS, SUBS, ADDIS y SUBIS generan un aluControl con un 1 en el bit más significativo, activando la señal `write_flags` cuando se ejecutan estas instrucciones.

```
module alu #(parameter N = 64) (
    input logic [63:0] a, b,
    input logic [3:0] ALUControl,
    output logic [63:0] result,
    output logic zero,
    output logic negative,
    output logic carry,
    output logic overflow,
    output logic write_flags
);
    logic [63:0] comp_2;
    logic [64:0] carry_aux;

    always_comb begin
        comp_2 = ~b + 1;
        carry_aux = 0;
        casez(ALUControl)
            4'b0000: result = a & b;
            4'b0001: result = a | b;
            4'b?010: begin
                carry_aux = a + b;
                result = carry_aux[63:0];
            end
            4'b?110: result = a + comp_2;
            4'b0111: result = b;
            default: result = '1;
        endcase;

        // TODO: check
```

```
        zero = result == '0 ? '1 : '0;
        carry = carry_aux[64]; // Este metodo me lo dijo el Gonza
        overflow = (a[63] == b[63]) && (a[63] != result[63]);
        negative = result[63];
        write_flags = ALUControl[3];
    end
endmodule
```

- **Execute**:
  - Se añadió un registro de cuatro bits (`CPSR_flags`) en el cual se almacenan las banderas Z, N, C, y V. Este registro se actualiza únicamente cuando se ejecutan instrucciones que configuran banderas.
  - Se agrego el modulo bcondcheck, el cual se encarga de determinar si se cumplen o no las condiciones para un salto condicional.
    - Se incluyeron las 14 condiciones de salto de LEGv8 para B.cond.

```
module execute #(parameter N = 64) (
        input logic AluSrc,
        input logic [3:0] AluControl,
        input logic [N-1:0] PC_E, signImm_E, readData1_E, readData2_E,
        input logic [4:0] qIF_ID,
        output logic [N-1:0] PCBranch_E, aluResult_E, writeData_E,
        output logic zero_E,
        output logic doBranch_E
    );

    logic [N-1:0] MUX_out;
    logic [N-1:0] sl2_out;
    logic [3:0] CPSR_flags;
    logic negative, carry, overflow, write_flags;

    mux2 #(N) MUX(readData2_E, signImm_E, AluSrc, MUX_out);
    sl2 #(N) sl2(signImm_E, sl2_out);
    adder #(N) adder(PC_E, sl2_out, PCBranch_E);
    alu #(N) alu_(readData1_E, MUX_out, AluControl, aluResult_E, zero_E,
        negative, carry, overflow, write_flags);
    bCondCheck bcond(qIF_ID, CPSR_flags, doBranch_E);


    assign writeData_E = readData2_E;

    always_comb begin
        CPSR_flags = 4'b0000;
```

```
        if (write_flags) begin // LINE 26
            CPSR_flags[3] = zero_E;   // Z flag
            CPSR_flags[2] = negative; // N flag
            CPSR_flags[1] = carry;    // C flag
            CPSR_flags[0] = overflow;  // V flag
        end
    end
endmodule
```

- **bCondCheck**:

```
module bCondCheck (

    input logic [4:0] cond,
    input logic [3:0] flags,
    output logic doBranch
);
    logic Z, N, C, V;
    always_comb begin
        {Z, N, C, V} = flags; // Line 8
        doBranch = 1'b0;
        case(cond[3:0])
            4'b0000: doBranch = Z == 1; // EQ
            4'b0001: doBranch = Z == 0; // NE
            4'b0010: doBranch = C == 1; // HS
            4'b0011: doBranch = C == 0; // LO
            4'b0100: doBranch = N == 1; // MI
            4'b0101: doBranch = N == 0; // PL
            4'b0110: doBranch = V == 1; // VS
            4'b0111: doBranch = V == 0; // VC
            4'b1000: doBranch = (Z==0 && C==1); // HI
            4'b1001: doBranch = ~(Z==0 && C==1); // LS
            4'b1010: doBranch = N == V; // GE
            4'b1011: doBranch = N != V; // LT
            4'b1100: doBranch = (Z == 0 && N == V); // GT
            4'b1101: doBranch = ~(Z == 0 && N == V); // LE
            default: doBranch = 1'b0;
        endcase
    end

endmodule
```

- **Datapath**:

- Se agregó la señal de control `condBranch` para manejar los saltos condicionales.
- Se agregó la señal `doBranch` que sale desde el execute y va hacia Memory.
  - Esta señal se activa cuando se cumplen las condiciones del salto estipuladas por B.cond.

3.2 ADDS, SUBS, ADDIS, SUBIS y B.cond

- **aludec.sv:** Se han añadido casos en el aludec para incluir las nuevas instrucciones ADDS, SUBS, ADDIS y SUBIS.

```
else if((aluop == 2'b10)  & (funct == 11'b101_0101_1000)) alucontrol = 4'b1010; // ADDS
else if ((aluop == 2'b10) & (funct == 11'b111_0101_1000)) alucontrol = 4'b1110; // SUBS
else if((aluop == 2'b10)  & (funct[10:1] == 10'b1111000100)) alucontrol = 4'b1110; // SU
else if((aluop == 2'b10)  & (funct[10:1] == 10'b1011000100)) alucontrol = 4'b1010; // AD
```

- **maindec.sv:** Se implementaron las señales para las nuevas instrucciones
- **signext.sv:** Añadimos ADDIS, SUBIS y B.cond al case del signextend.

3.3 Verificaciónes

3.3.1 ADDIS

```
.text
.org 0x0000
ADDS XZR, X2, #3
NOP
NOP
B.PL PL_pass
NOP
NOP
STUR X15, [X0, #0] // MEM 0:0x0F -> WRONG
CBZ XZR, finloop
NOP
NOP
NOP
PL_pass: STUR X30, [X0, #0] // MEM 0:0x1E -> CORRECT
NOP
NOP
VS_loop: ADDS X30, X0, #4095
NOP
```

```
NOP
B.VS VS_pass
NOP
NOP
CBZ XZR, VS_loop
VS_pass: STUR X7, [X0, #8] // MEM 1: 0x7
finloop: CBZ XZR, finloop
```

3.3.2 SUBS_TEST

```
.text
.org 0x0000

SUBS X1, X10, X5
B.PL PL_pass_1
NOP
NOP
STUR X1, [X0, #0]
CBZ XZR, finloop
PL_pass_1: STUR X1, [X0, #0]


SUBS X2, X5, X10
B.MI MI_pass_2
NOP
NOP
STUR X15, [X0, #8]
CBZ XZR, finloop
MI_pass_2: STUR X2, [X0, #8]

SUBS X3, X10, X10
B.EQ EQ_pass_3
NOP
NOP
STUR X15, [X0, #16]
CBZ XZR, finloop
EQ_pass_3: STUR X3, [X0, #16]

finloop: CBZ XZR, finloop
```

3.3.3 SUBIS_TEST

```
.text
.org 0x0000

SUBS X1, X10, #5
B.PL PL_pass_1
NOP
NOP
STUR X1, [X0, #0]
CBZ XZR, finloop
PL_pass_1: STUR X1, [X0, #0]


SUBS X2, X5, #10
B.MI MI_pass_2
NOP
NOP
STUR X15, [X0, #8]
CBZ XZR, finloop
MI_pass_2: STUR X2, [X0, #8]

SUBS X3, X10, #10
B.EQ EQ_pass_3
NOP
NOP
STUR X15, [X0, #16]
CBZ XZR, finloop
EQ_pass_3: STUR X3, [X0, #16]

finloop: CBZ XZR, finloop
```

3.3.4 defaultPipelineTest

- Este es el codigo provisto en el enunciado del laboratorio

```
.text
.org 0x0000
STUR X1, [X0, #0] // MEM 0:0x1
STUR X2, [X0, #8] // MEM 1:0x2
STUR X3, [X16, #0] // MEM 2:0x3
ADD X3, X4, X5
```

```
NOP
NOP
STUR X3, [X0, #24] // MEM 3:0x9
SUB X3, X4, X5
NOP
NOP
STUR X3, [X0, #32] // MEM 4:0xFFFFFFFFFFFFFFFF
SUB X4, XZR, X10
NOP
NOP
STUR X4, [X0, #40] // MEM 5:0xFFFFFFFFFFFFFFF6
ADD X4, X3, X4
NOP
NOP
STUR X4, [X0, #48] // MEM 6:0xFFFFFFFFFFFFFFF5
SUB X5, X1, X3
NOP
NOP
STUR X5, [X0, #56] // MEM 7:0x2
AND X5, X10, XZR
NOP
NOP
STUR X5, [X0, #64] // MEM 8:0x0
AND X5, X10, X3
NOP
NOP
STUR X5, [X0, #72] // MEM 9:0xA
AND X20, X20, X20
NOP
NOP
STUR X20, [X0, #80] // MEM 10:0x14
ORR X6, X11, XZR
NOP
NOP
STUR X6, [X0, #88] // MEM 11:0xB
ORR X6, X11, X3
NOP
NOP
STUR X6, [X0, #96] // MEM 12:0xFFFFFFFFFFFFFFFF
LDUR X12, [X0, #0]
NOP
NOP
NOP
```

```
ADD X7, X12, XZR
NOP
NOP
STUR X7, [X0, #104] // MEM 13:0x1
STUR X12, [X0, #112] // MEM 14:0x1
ADD XZR, X13, X14
STUR XZR, [X0, #120] // MEM 15:0x0
CBZ X0, L1
NOP
NOP
NOP
STUR X21, [X0, #128] // MEM 16:0x0(si falla CBZ =21)
L1: STUR X21, [X0, #136] // MEM 17:0x15
ADD X2, XZR, X1
NOP
NOP
L2: SUB X2, X2, X1
ADD X24, XZR, X1
NOP
NOP
STUR X24, [X0, #144] // MEM 18:0x1 y MEM 19=0x1
ADD X0, X0, X8
CBZ X2, L2
NOP
NOP
STUR X30, [X0, #144] // MEM 20:0x1E
ADD X30, X30, X30
SUB X21, XZR, X21
NOP
ADD X30, X30, X20
NOP
NOP
LDUR X25, [X30, #-8]
NOP
ADD X30, X30, X30
NOP
NOP
ADD X30, X30, X16
NOP
NOP
STUR X25, [X30, #-8] // MEM 21:0xA (= MEM 9)
finloop: CBZ XZR, finloop
```

3.3.5 bcond_all

- Este codigo testea B.cond en general

```
.text
.org 0x0000
ADDS XZR, X2, X3
NOP
NOP
B.PL PL_pass
NOP
NOP
STUR X15, [X0, #0] // MEM 0:0x0F -> WRONG
CBZ XZR, PL_end
NOP
NOP
NOP
PL_pass: STUR X30, [X0, #0] // MEM 0:0x1E -> CORRECT
PL_end:
ADD X30, X0, #4095
NOP
NOP
VS_loop: ADDS X30, X30, X30
NOP
NOP
B.VS VS_pass
NOP
NOP
CBZ XZR, VS_loop
VS_pass: STUR X7, [X0, #8] // MEM 1: 0x7
SUBS X1, X10, X5
B.PL PL_pass_1
NOP
NOP
STUR X15, [X0, #16]
CBZ XZR, PL_end2
NOP
NOP
PL_pass_1: STUR X1, [X0, #16] // MEM 2: 0x05
PL_end2:
SUBS X2, X5, X10
B.MI MI_pass_2
NOP
NOP
```

```
STUR X15, [X0, #24]
CBZ XZR, MI_end
NOP
NOP
MI_pass_2: STUR X2, [X0, #24] // MEM 3: 0xFFFFFFFFFFFFFFFB
MI_end:
SUBS X3, X10, X10
B.EQ EQ_pass_3
NOP
NOP
STUR X15, [X0, #32]
CBZ XZR, EQ_end
NOP
NOP
EQ_pass_3: STUR X3, [X0, #32] // MEM 4: 0x00
EQ_end:
// Test for B.NE (Not Equal)
SUBS X4, X1, X1 // X4 = 0
B.NE NE_pass_4
NOP
NOP
STUR X15, [X0, #40] // Should not execute
CBZ XZR, NE_end
NE_pass_4: STUR X4, [X0, #40] // MEM 5: 0x00
NE_end:
// Test for B.GT (Greater Than)
SUBS X5, X1, X10 // X5 = 5 - 10 = -5
B.GT GT_pass_5
NOP
NOP
STUR X15, [X0, #48] // Should not execute
CBZ XZR, GT_end
GT_pass_5: STUR X5, [X0, #48] // MEM 6: 0xFFFFFFFFFFFFFFFB
GT_end:
// Test for B.LT (Less Than)
SUBS X6, X10, X1 // X6 = 10 - 5 = 5
B.LT LT_pass_6
NOP
NOP
STUR X15, [X0, #56] // Should not execute
CBZ XZR, LT_end
LT_pass_6: STUR X6, [X0, #56] // MEM 7: 0x05
LT_end:
```

```
// Test for B.GE (Greater Than or Equal)
SUBS X7, X1, X1 // X7 = 5 - 5 = 0
B.GE GE_pass_7
NOP
NOP
STUR X15, [X0, #64] // Should not execute
CBZ XZR, GE_end
GE_pass_7: STUR X7, [X0, #64] // MEM 8: 0x00
GE_end:
// Test for B.LE (Less Than or Equal)
SUBS X8, X5, X10 // X8 = 10 - 5 = 5
B.LE LE_pass_8
NOP
NOP
STUR X15, [X0, #72] // Should not execute
CBZ XZR, finloop
LE_pass_8: STUR X10, [X0, #72] // MEM 9: 0x0A
finloop: CBZ XZR, finloop
```

3.3.6 bcond_addition

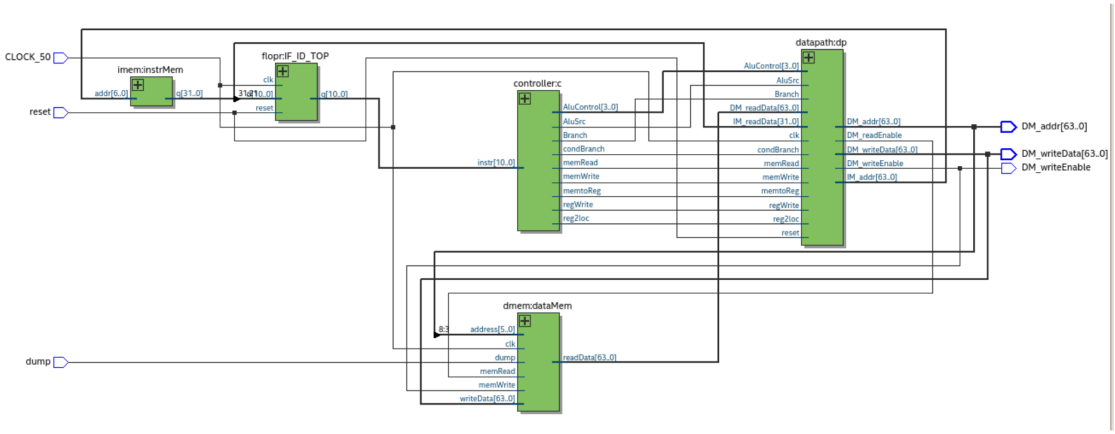
- Este codigo testea ADDS y las condiciones de B.cond que tengan que ver con la adicion

```
.text
.org 0x0000
ADDS XZR, X2, X3
NOP
NOP
B.PL PL_pass
NOP
NOP
STUR X15, [X0, #0] // MEM 0:0x0F -> WRONG
CBZ XZR, finloop
NOP
NOP
NOP
PL_pass: STUR X30, [X0, #0] // MEM 0:0x1E -> CORRECT
ADD X30, X0, #4095
NOP
NOP
VS_loop: ADDS X30, X30, X30
```

```
NOP
NOP
B.VS VS_pass
NOP
NOP
CBZ XZR, VS_loop
VS_pass: STUR X7, [X0, #8] // MEM 1: 0x7
finloop: CBZ XZR, finloop
```

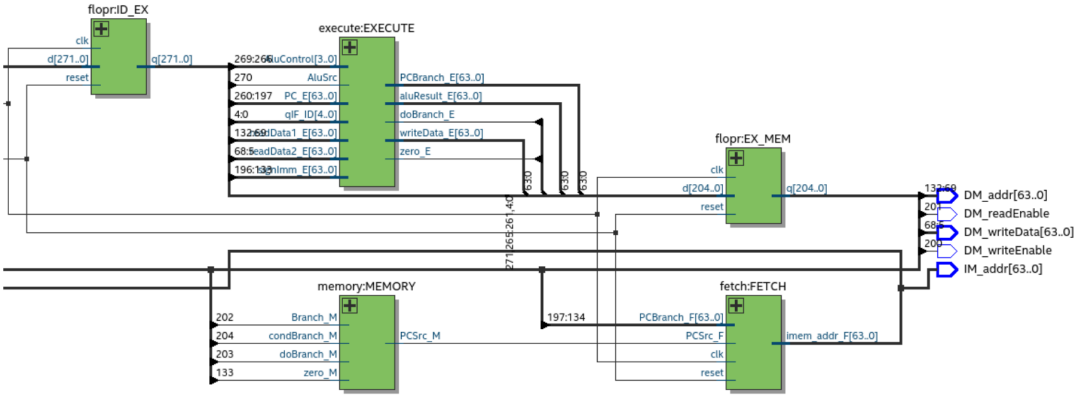
3.4 Diagramas

3.4.1 Procesador completo



3.4.2 Porcion del datapath modificada

(Hubo otras leves modificaciones, pero aqui se encuentran las más relevantes)


flopr:ID_EX
clk
d[271..0]
reset
q[271..0]
execute:EXECUTE
269:266
AluControl[3..0]
270
AluSrc
260:197
PC_E[63..0]
4:0
qIF_ID[4..0]
132:69
rdData1_E[63..0]
68:5
rdData2_E[63..0]
196:133
Imm_E[63..0]
PCBranch_E[63..0]
aluResult_E[63..0]
doBranch_E
writeData_E[63..0]
zero_E
flopr:EX_MEM
clk
d[204..0]
reset
q[204..0]
DM_addr[63..0]
DM_readEnable
DM_writeData[63..0]
DM_writeEnable
IM_addr[63..0]
memory:MEMORY
202
Branch_M
204
condBranch_M
203
doBranch_M
133
zero_M
PCSrc_M
fetch:FETCH
197:134
PCBranch_F[63..0]
PCSrc_F
clk
reset
imem_addr_F[63..0]