بسمه تعالی

گزارش کار هشتم آزمایشگاه طراحی سیستم های دیجیتال

ALU اعداد مختلط

استاد: دکتر اجلالی

نویسندگان

مریم شیران ۴۰۰۱۰۹۴۴۶

مهدی بهرامیان ۴۰۱۱۷۱۵۹۳

مزدک تیموریان ۴۰۱۱۰۱۴۹۵

دانشگاه صنعتی شریف

تابستان ۱۴۰۳

فهرست مطالب

۱ آزمایش هشتم: ALU اعداد مختلط

۱-۱ مقدمه

در این آزمایش، هدف ما طراحی یک واحد محاسباتی ساده برای اعداد مختلط است که به ما امکان می‌دهد:

- دو عدد مختلط را با هم جمع یا از هم کم کنیم.

- دو عدد مختلط را در هم ضرب کنیم.

- دستورات از حافظه توسط یک واحد پایپ‌لاین خوانده شده و به صورت پایپ‌لاین اجرا شوند.

۱-۲ شرح آزمایش

۱ -۲ -۱ جمع کننده مختلط (complex_adder. v):

ماژول complex_adder یک واحد حسابی ساده را تعریف می‌کند که می‌تواند دو ورودی ۱۶ بیتی، a و b، را جمع یا تفریق کند.

اجزای مدار:

ورودی‌ها: a و b عملوندهای ۱۶ بیتی هستند. addnot_sub یک سیگنال کنترلی است که تعیین می‌کند آیا عملیات جمع (۰) یا تفریق (۱) انجام شود.

خروجی: c یک ثبات ۱۶ بیتی است که نتیجه عملیات در آن ذخیره می‌شود.

منطق ترکیبی: بلوک always_comb شامل منطقی است که هر زمان که هر یک از ورودی‌ها تغییر کنند اجرا می‌شود.

اگر addnot_sub برابر با ۱ باشد، عملیات انجام شده تفریق برای هر دو بخش ۸ بیتی بالایی و پایینی ورودی‌ها است.

اگر addnot_sub برابر با ۰ باشد، عملیات انجام شده جمع برای هر دو بخش ۸ بیتی بالایی و پایینی ورودی‌ها است. این ساختار کد تضمین می‌کند که عملیات به طور جداگانه بر روی ۸ بیت بالایی و پایینی ورودی‌ها انجام می‌شود،

کد وریلاگ:

سر تا سر کد، به منظور خوانایی و توضیح کامنت گذاری شده است.

```
module complex_adder (
   input [15:0] a,          // 16-bit input operand 'a'
   input [15:0] b,          // 16-bit input operand 'b'
   input addnot_sub,        // Control signal: '1' for subtraction, '0' for addition
   output reg [15:0] c      // 16-bit output result 'c'
);

   // Always block with combinational logic
   always_comb begin
      if (addnot_sub) begin
         // If addnot_sub is '1', perform subtraction
         c[15:8] = a[15:8] - b[15:8]; // Subtract upper 8 bits of 'b' from 'a'
         c[7:0]  = a[7:0]  - b[7:0];  // Subtract lower 8 bits of 'b' from 'a'
      end else begin
         // If addnot_sub is '0', perform addition
         c[15:8] = a[15:8] + b[15:8]; // Add upper 8 bits of 'b' to 'a'
         c[7:0]  = a[7:0]  + b[7:0];  // Add lower 8 bits of 'b' to 'a'
      end
   end

endmodule
```

۱ -۲ -۲ ضرب کننده :

کد Verilog زیر دو ماژول به نام‌های mult و complex_mult تعریف می‌کند. در اینجا خلاصه‌ای از آن آمده است:

۱. ماژول mult

هدف: انجام ضرب با علامت دو عدد N بیتی (a و b) با گسترش علامت، که خروجی M بیتی (c) تولید می‌کند.

پارامترها: N: عرض بیت‌های ورودی‌های a و b. M: عرض بیت‌های خروجی c، که معمولاً ۲ * N است.

عملکرد: این ماژول شامل یک تابع برای گسترش علامت b از N بیت به M بیت است. زمانی که سیگنال start فعال می‌شود، ماژول فرآیند ضرب را آغاز می‌کند. اعداد منفی را با گرفتن متمم دو آن‌ها مدیریت می‌کند. ضرب با استفاده از یک رویکرد تکراری (مشابه ضرب با شیفت و جمع) انجام می‌شود. سیگنال ready زمانی که ضرب کامل می‌شود فعال می‌گردد.

۲. ماژول complex_mult

هدف: محاسبه حاصل ضرب دو عدد مختلط ۱۶ بیتی (a و b) و تولید یک نتیجه مختلط ۱۶ بیتی (c).

عملکرد: اعداد مختلط ورودی به قسمت‌های حقیقی و موهومی تقسیم می‌شوند. چهار ضرب با استفاده از ماژول mult برای محاسبه قسمت‌های حقیقی و موهومی حاصل ضرب انجام می‌شود. خروجی نهایی c با ترکیب نتایج این ضرب‌ها شکل می‌گیرد. سیگنال ready نشان می‌دهد که تمامی چهار ضرب کامل شده‌اند، به این معنی که ضرب مختلط به پایان رسیده است.

به‌طور کلی ماژول mult یک بلوک ساختمانی برای ضرب با علامت عمومی است، و ماژول complex_mult از آن برای انجام ضرب اعداد مختلط استفاده می‌کند که شامل چندین ضرب و جمع/تفریق حقیقی می‌باشد.

سر تا سر کد، به منظور خوانایی و توضیح کامنت گذاری شده است.

```
// Module for performing multiplication with sign extension
module mult #(
  // Parameters for input size N and output size M
  parameter N = 8,
  parameter M = 2 * N
) (
  // Inputs
  input [N-1:0] a,        // First multiplicand (N bits)
```

```
    input [N-1:0] b,         // Second multiplicand (N bits)
    input clk,               // Clock signal
    input rst,               // Reset signal (active low)
    input start,             // Start signal for multiplication
    // Outputs
    output reg ready,        // Ready signal, indicates when multiplication is done
    output reg [M-1:0] c     // Output product (M bits)
);

    // Function to sign-extend input 'in' from N bits to M bits
    function [M-1:0] ext(input [N-1:0] in);
        ext[M-1:N-1] = {M - N + 1{in[N-1]}}; // Sign-extension for upper bits
        ext[N-2:0]   = in[N-2:0];          // Lower bits remain the same
    endfunction

    // Internal registers for holding intermediate values
    reg [N-1:0] aa;          // Register to hold the absolute value of 'a'
    reg [M-1:0] bb;          // Register to hold the extended and possibly negated
value of 'b'

    // Always block for sequential logic (multiplication process)
    always_ff @(posedge clk, negedge rst) begin
        if (!rst) begin
            // Reset condition: clear all registers and set 'ready' signal
            aa <= 0;
            bb <= 0;
            c <= 0;
```

```
        ready <= 1;
      end else begin
        if (start) begin
          // Start condition: initialize multiplication
          c <= 0;
          if (a[N-1]) begin
            // If 'a' is negative, take two's complement of 'a' and 'b'
            aa <= ~a + 1;
            bb <= ~ext(b) + 1;
          end else begin
            // If 'a' is positive, use 'a' and extended 'b' directly
            aa <= a;
            bb <= ext(b);
          end
          ready <= 0; // Clear 'ready' signal during computation
        end else begin
          // Multiplication process: add and shift based on LSB of 'aa'
          if (~|aa) ready <= 1;       // Set 'ready' when 'aa' is zero (done)
          if (aa[0]) c <= c + bb;     // Add 'bb' to 'c' if LSB of 'aa' is 1
          aa <= aa >> 1;              // Shift 'aa' right by 1
          bb <= bb << 1;              // Shift 'bb' left by 1
        end
      end
    end
endmodule
```

```
// Module for performing complex multiplication
module complex_mult (
   input [15:0] a,         // Complex input 'a' (8 bits for real, 8 bits for imaginary)
   input [15:0] b,         // Complex input 'b' (8 bits for real, 8 bits for imaginary)
   input clk,              // Clock signal
   input rst,              // Reset signal (active low)
   input start,            // Start signal for multiplication
   output ready,           // Ready signal, indicates when multiplication is done
   output [15:0] c         // Complex output 'c' (8 bits for real, 8 bits for
imaginary)
);

   // Split input 'a' into real and imaginary parts
   wire [7:0] ax = a[15:8]; // Real part of 'a'
   wire [7:0] bx = b[15:8]; // Real part of 'b'
   wire [7:0] ay = a[7:0];  // Imaginary part of 'a'
   wire [7:0] by = b[7:0];  // Imaginary part of 'b'

   // Wires for holding the results of intermediate multiplications
   wire [7:0] axbx, axby, aybx, ayby;
   wire rxx, rxy, ryx, ryy; // Ready signals for each multiplier

   // Calculate the final real and imaginary parts of the output 'c'
   assign c[15:8] = axbx - ayby; // Real part: ax * bx - ay * by
   assign c[7:0]  = axby + aybx; // Imaginary part: ax * by + ay * bx
   assign ready   = rxx && rxy && ryx && ryy; // Overall ready when all
multipliers are done
```

```
// Instantiate 4 multipliers for the complex multiplication
mult #(.M(8)) _axbx (  // Multiplier for ax * bx
    .a(ax),
    .b(bx),
    .start(start),
    .rst(rst),
    .clk(clk),
    .c(axbx),
    .ready(rxx)
);

mult #(.M(8)) _axby (  // Multiplier for ax * by
    .a(ax),
    .b(by),
    .start(start),
    .rst(rst),
    .clk(clk),
    .c(axby),
    .ready(rxy)
);

mult #(.M(8)) _aybx (  // Multiplier for ay * bx
    .a(ay),
    .b(bx),
    .start(start),
```

```
    .rst(rst),
    .clk(clk),
    .c(aybx),
    .ready(ryx)
  );

  mult #(.M(8)) _ayby (  // Multiplier for ay * by
    .a(ay),
    .b(by),
    .start(start),
    .rst(rst),
    .clk(clk),
    .c(ayby),
    .ready(ryy)
  );
endmodule
```

۱ -۲ -۳ Data memory :

توضیح اجزای کلیدی:

آرایه حافظه: آرایه data شامل ۳۲ مکان است که هر کدام داده‌های ۱۶ بیتی را ذخیره می‌کنند. این حافظه نمایانگر فضای ذخیره‌سازی اصلی است که عملیات خواندن و نوشتن در آن انجام می‌شود.

عملیات خواندن: دستورات assign خروجی‌های read_data۰ و read_data۱ را به‌طور مداوم با مقادیر ذخیره‌شده در حافظه در آدرس‌های مشخص‌شده توسط read_address۰ و read_address۱ هدایت می‌کنند.

عملیات نوشتن: بلوک always در لبه بالارونده سیگنال کلاک (clk) تحریک می‌شود. زمانی که سیگنال write فعال است (High)، حافظه در مکانی که توسط write_address مشخص شده است، با داده‌های write_data به‌روز می‌شود.

مقدارسازی اولیه: بلوک initial برای مقداردهی اولیه برخی از مکان‌های حافظه با اعداد مختلط خاص در طول شبیه‌سازی استفاده می‌شود. این مقادیر در سخت‌افزار سنتز شده وجود نخواهند داشت، اما برای شبیه‌سازی تست بنچ مفید هستند.

سر تا سر کد، به منظور خوانایی و توضیح کامنت گذاری شده است.

```
module dmem (
   input [4:0] read_address0,  // 5-bit input for the first read address (supports 32 memory locations)
   input [4:0] read_address1,  // 5-bit input for the second read address (supports 32 memory locations)
   input [4:0] write_address,  // 5-bit input for the write address (supports 32 memory locations)
   output [15:0] read_data0,   // 16-bit output for data read from the first address
   output [15:0] read_data1,   // 16-bit output for data read from the second address
   input write,                // Control signal for writing data; active when high
   input [15:0] write_data,    // 16-bit input data to be written to the specified write address
   input clk                   // Clock signal, used for synchronizing write operations
);

   // Declare a memory array of 32 locations, each 16 bits wide
   reg [15:0] data[31:0];
```

```
    // Assign the data at read_address0 to read_data0
    assign read_data0 = data[read_address0];

    // Assign the data at read_address1 to read_data1
    assign read_data1 = data[read_address1];

    // Initialize memory with predefined complex numbers (in real + imaginary
format)
    initial begin
        data[0] = {8'd1, 8'd2};  // Complex number: 1 + 2i
        data[1] = {8'd3, 8'd4};  // Complex number: 3 + 4i
        data[2] = {8'd0, 8'd1};  // Complex number: 0 + 1i (pure imaginary)
        data[3] = {8'd1, 8'd0};  // Complex number: 1 + 0i (pure real)
    end

    // On the rising edge of the clock, if the write signal is high,
    // update the memory at the specified write address with the provided
write_data
    always @(posedge clk) begin
        if (write)
            data[write_address] <= write_data;
    end
endmodule
```

۱ -۲ - ۴Instruction memory :

توضیح اجزای کلیدی:

آرایه حافظه (data): این یک آرایه با ۳۲ مکان است که هر مکان می‌تواند یک دستورالعمل ۱۶ بیتی را ذخیره کند. دستورالعمل‌ها به صورت دودویی نمایش داده می‌شوند.

سیگنال عملیات (op): بیشترین بیت (بیت ۱۵) هر دستورالعمل نوع عملیات را تعیین می‌کند (۱ برای ضرب، ۰ برای جمع). عملوندها (src۰, src۱, dst): بخش‌های ۵ بیتی هر دستورالعمل نشان‌دهنده عملوندهای منبع (src۰, src۱) و عملوند مقصد (dst) هستند.

بلوک مقداردهی اولیه: بلوک initial برای بارگذاری اولیه حافظه با دستورالعمل‌های خاص استفاده می‌شود. این دستورالعمل‌ها به صورت دودویی هستند و عملیات‌های اساسی مانند جمع و ضرب بر روی اعداد مختلط را نشان می‌دهند. توضیحات مربوطه عملیات‌های ریاضی مرتبط و نتایج مورد انتظار آن‌ها را به صورت دودویی توصیف می‌کنند.

فقط یک نکته باقی مانده است که باید به آن اشاره شود: واحد جمع/تفریق‌کننده قادر به انجام عملیات تفریق نیز هست. اما برای این عملیات، آپکد (Opcode) اختصاصی برای تفریق در نظر گرفته نشده است. دلیل این امر این است که میخواستیم طول کلمات همچنان ۱۶ بیت باقی بماند. در این طراحی، ۵ بیت برای مقصد، ۵ بیت برای خانه‌های ۰ و ۱، و ۱ بیت باقی می‌ماند. به این ترتیب، آپکد ۱ برای ضرب و آپکد ۰ برای جمع در نظر گرفته شده است.

سر تا سر کد، به منظور خوانایی و توضیح کامنت گذاری شده است.

```
module imem (
   input [4:0] address,     // 5-bit input address for accessing instructions (supports 32 memory locations)
   output op,               // Output signal for the operation type (1 for multiplication, 0 for addition)
   output [4:0] src0,       // 5-bit output for the first source operand
   output [4:0] src1,       // 5-bit output for the second source operand
   output [4:0] dst         // 5-bit output for the destination operand
);

   // Declare a memory array of 32 locations, each 16 bits wide, to store instructions
   reg [15:0] data[31:0];
```

```
    // Assign the operation type (most significant bit) of the instruction to `op`
    assign op   = data[address][15];

    // Assign bits [14:10] of the instruction to `dst`, representing the destination operand
    assign dst  = data[address][14:10];

    // Assign bits [9:5] of the instruction to `src1`, representing the second source operand
    assign src1 = data[address][9:5];

    // Assign bits [4:0] of the instruction to `src0`, representing the first source operand
    assign src0 = data[address][4:0];

    // Initialize the memory with predefined instructions
    initial begin
        // Instruction format: 1-bit operation type | 5-bit destination | 5-bit src1 | 5-bit src0

        data[0] = 16'b1_00100_00001_00000;  // Multiply: mem[4] = mem[0] * mem[1]
        // Explanation: (1 + 2i) * (3 + 4i) = -5 + 10i = 11111011 00001010

        data[1] = 16'b0_00105_00000_00000;  // Add: mem[5] = mem[0] + mem[0]
        // Explanation: (1 + 2i) + (1 + 2i) = 2 + 4i = 00000010 00000100
```

```
    data[2] = 16'b1_00110_00010_00001;  // Multiply: mem[6] = mem[2] * mem[1]
    // Explanation: (i) * (3 + 4i) = -4 + 3i = 11111100 00000011

    data[3] = 16'b0_00111_00010_00011;  // Add: mem[7] = mem[2] + mem[3]
    // Explanation: (i) + (1) = 1 + i = 00000001 00000001
  end
endmodule
```

۱ -۲ -۵ خط داده (pipeline.v):

نمای کلی ماژول:

این کد یک معماری خط لوله ساده را تعریف می‌کند که شامل مراحل مختلفی مانند واکشی دستورالعمل، دسترسی به حافظه داده، عملیات واحد حساب و منطق (ALU) و مرحله نوشتن خروجی است.

ثبات‌ها و سیم‌ها: ثبات‌ها برای ذخیره داده‌ها بین سیکل‌های کلاک استفاده می‌شوند، در حالی که سیم‌ها برای انتقال داده‌ها بین ماژول‌ها و منطق ترکیبی به کار می‌روند.

مراحل خط لوله: خط لوله شامل واکشی دستورالعمل (از طریق IMEM)، دسترسی به حافظه داده (از طریق DMEM)، عملیات حسابی (از طریق ALU) و در نهایت نوشتن نتیجه به ثبات‌ها (مرحله نوشتن خروجی) است.

ماژول‌های ALU و حافظه: واحد حساب و منطق (ALU) می‌تواند بسته به عملیات مشخص شده، جمع یا ضرب را انجام دهد. ماژول‌های حافظه (IMEM و DMEM) برای واکشی دستورالعمل‌ها و ذخیره/بازیابی داده‌ها استفاده می‌شوند.

سر تا سر کد، به منظور خوانایی و توضیح کامنت گذاری شده است.

```
module pipeline (
  input clk,  // Clock input for synchronizing the pipeline stages
```

```
    input rst   // Reset input for initializing or resetting the pipeline
);

    // Program Counter (PC) register to hold the address of the next instruction
    reg [4:0] PC;

    // Instruction memory (IMEM) interface wires
    wire imem_op;          // Operation type (e.g., add, multiply) fetched from IMEM
    wire [4:0] imem_src0;   // Source operand 0 address fetched from IMEM
    wire [4:0] imem_src1;   // Source operand 1 address fetched from IMEM
    wire [4:0] imem_dst;    // Destination register address fetched from IMEM

    // Data memory (DMEM) interface registers
    reg DMEM_valid;        // Validity of DMEM operation
    reg DMEM_op;           // Operation type (e.g., load/store)
    reg [4:0] DMEM_src0;    // Source operand 0 address for DMEM
    reg [4:0] DMEM_src1;    // Source operand 1 address for DMEM
    reg [4:0] DMEM_dst;     // Destination register address for DMEM

    // Data memory (DMEM) interface wires
    wire [15:0] dmem_data0; // Data read from DMEM at address DMEM_src0
    wire [15:0] dmem_data1; // Data read from DMEM at address DMEM_src1

    // ALU interface registers
    reg ALU_valid;         // Validity of ALU operation
    reg ALU_op;            // Operation type (e.g., add or multiply)
```

```
reg [4:0] ALU_dst;      // Destination register address for ALU
reg [15:0] ALU_data0;   // Operand 0 for ALU operation
reg [15:0] ALU_data1;   // Operand 1 for ALU operation
reg ALU_start;          // Signal to start the ALU operation

// ALU interface wires
wire [15:0] add_res;    // Result from the adder
wire [15:0] mul_res;    // Result from the multiplier
wire mul_ready;         // Multiplier ready signal
wire [15:0] alu_res = ALU_op ? mul_res : add_res;  // Select the ALU result based on the operation
wire alu_ready = ALU_valid ? (ALU_op ? mul_ready : 1) : 1; // ALU ready signal based on operation and validity

// Write-back stage registers
reg WB_write;           // Write-back enable signal
reg [4:0] WB_dst;       // Destination register address for write-back
reg [15:0] WB_data;     // Data to be written back to the register

// Program Counter (PC) update logic
always @(posedge clk, negedge rst) begin
    if (rst == 0)
        PC <= 0;  // Reset PC to 0
    else if (alu_ready)
        PC <= PC + 1;  // Increment PC if ALU is ready
end
```

```
  // Data Memory (DMEM) stage logic
  always @(posedge clk, negedge rst)
    if (rst == 0) begin
      DMEM_valid <= 0;  // Reset DMEM valid signal
    end else if (alu_ready) begin
      DMEM_valid <= 1;    // Set DMEM valid signal
      DMEM_op    <= imem_op;  // Pass operation type from IMEM to DMEM
      DMEM_src0  <= imem_src0; // Pass source operand 0 address from
IMEM to DMEM
      DMEM_src1  <= imem_src1; // Pass source operand 1 address from
IMEM to DMEM
      DMEM_dst   <= imem_dst;  // Pass destination address from IMEM to
DMEM
    end


  // ALU stage logic
  always @(posedge clk, negedge rst)
    if (rst == 0) begin
      ALU_valid <= 0;  // Reset ALU valid signal
    end else if (alu_ready) begin
      ALU_valid <= DMEM_valid; // Pass DMEM valid signal to ALU
      ALU_op    <= DMEM_op;    // Pass operation type from DMEM to ALU
      ALU_dst   <= DMEM_dst;   // Pass destination address from DMEM to
ALU
      ALU_data0 <= dmem_data0; // Pass operand 0 data from DMEM to ALU
      ALU_data1 <= dmem_data1; // Pass operand 1 data from DMEM to ALU
      ALU_start <= 1;         // Start ALU operation
```

```
    end else begin
        ALU_start <= 0;  // Stop ALU operation
    end

// Write-back (WB) stage logic
always_comb
    if (rst == 1 && alu_ready) begin
        WB_write = ALU_valid;  // Enable write-back if ALU operation is valid
        WB_data  = alu_res;    // Write ALU result to WB data
        WB_dst   = ALU_dst;    // Write ALU destination to WB destination
    end else begin
        WB_write = 0;          // Disable write-back
        WB_data  = 0;          // Clear WB data
        WB_dst   = 0;          // Clear WB destination
    end

// Instantiate instruction memory (IMEM)
imem _imem (
    .address(PC),       // Address input (Program Counter)
    .op(imem_op),       // Operation type output
    .src0(imem_src0),   // Source operand 0 output
    .src1(imem_src1),   // Source operand 1 output
    .dst(imem_dst)      // Destination register address output
);

// Instantiate data memory (DMEM)
```

```
dmem _dmem (
    .read_address0(DMEM_src0),   // Read address for operand 0
    .read_address1(DMEM_src1),   // Read address for operand 1
    .read_data0(dmem_data0),     // Data output for operand 0
    .read_data1(dmem_data1),     // Data output for operand 1

    .write_address(WB_dst),      // Write address input
    .write_data(WB_data),        // Write data input
    .write(WB_write),            // Write enable input

    .clk(clk)                    // Clock input
);

// Instantiate complex adder
complex_adder _adder (
    .a(ALU_data0),       // Operand 0 input
    .b(ALU_data1),       // Operand 1 input
    .c(add_res),         // Sum output
    .addnot_sub(0)       // Add operation (0 for add, 1 for subtract)
);

// Instantiate complex multiplier
complex_mult _mult (
    .a(ALU_data0),       // Operand 0 input
    .b(ALU_data1),       // Operand 1 input
    .c(mul_res),         // Product output
```

```
    .ready(mul_ready),   // Multiplier ready output
    .start(ALU_start),   // Multiplier start input
    .clk(clk),           // Clock input
    .rst(rst)            // Reset input
);

endmodule
```

۳-۱ کدنویسی ماژول های تست در وریلاگ

کد Verilog زیر شامل دو ماژول تست بنچ به نام‌های `mult_test` و `test` است که هر کدام برای تست اجزای سخت‌افزاری مختلف طراحی شده‌اند.

۱. ماژول `mult_test`:

هدف: این تست‌بنش برای بررسی عملکرد یک ماژول ضرب‌کننده ۱۶ بیتی علامت‌دار (`mult`) طراحی شده است.

اجزاء:

ورودی‌ها: دو ورودی ۱۶ بیتی علامت‌دار `a` و `b`، سیگنال ساعت `clk`، سیگنال بازنشانی `rst`، و سیگنال شروع `start`.

خروجی‌ها: یک خروجی ۳۲ بیتی علامت‌دار `c` (نتیجه ضرب) و سیگنال `ready` که نشان‌دهنده تکمیل عمل ضرب است.

عملکرد: این ماژول چندین تست را اجرا می‌کند:

دو عدد مثبت، یک عدد مثبت و یک عدد منفی، دو عدد منفی، و یک عدد منفی و یک عدد مثبت را ضرب می‌کند.

پس از هر ضرب، نتیجه زمانی نمایش داده می‌شود که سیگنال `ready` فعال شود.

تولید سیگنال ساعت: سیگنال ساعت هر واحد زمانی تغییر می‌کند تا عملیات متوالی ضرب‌کننده را تضمین کند.

۲. ماژول `test`:

هدف: این تست‌بنش برای تست ماژول خط لوله (`pipeline`) CPU استفاده می‌شود.

اجزاء:

ورودی‌ها: سیگنال ساعت (`clk`) و سیگنال بازنشانی (`rst`).

نظارت: یک دستور `monitor$` به‌طور مداوم محتوای حافظه داده (`dmem_`) را در CPU دنبال و نمایش می‌دهد.

عملکرد: این ماژول خط لوله CPU را با اعمال یک بازنشانی شبیه‌سازی می‌کند و سپس به خط لوله اجازه می‌دهد تا مراحل مختلفی مانند واکشی دستور، واکشی داده، اجرا و نوشتن نهایی را طی کند.

تولید سیگنال ساعت: مشابه ماژول `mult_test`، سیگنال ساعت هر واحد زمانی تغییر می‌کند تا عملکرد CPU را هدایت کند. به طور کلی، این کد برای تست کامل عملکرد یک ضرب‌کننده و یک خط لوله ساده CPU از طریق شبیه‌سازی سناریوهای مختلف و مشاهده خروجی‌ها طراحی شده است.

سر تا سر کد، به منظور خوانایی و توضیح کامنت گذاری شده است.

```
// Testbench module for the multiplier
module mult_test ();
  // Declare signed 16-bit registers 'a' and 'b' as inputs to the multiplier
  reg signed [15:0] a, b;

  // Declare signed 32-bit wire 'c' to hold the output of the multiplier
  wire signed [31:0] c;

  // Declare a wire 'ready' to indicate when the multiplication is complete
```

```
wire ready;

// Declare clock, reset, and start signals as registers
reg clk, rst, start;

// Instantiate the multiplier module and connect the ports
mult _mult (
   .a(a),       // Connect input 'a' to the multiplier's input 'a'
   .b(b),       // Connect input 'b' to the multiplier's input 'b'
   .clk(clk),   // Connect clock signal to the multiplier's clock input
   .rst(rst),   // Connect reset signal to the multiplier's reset input
   .start(start),// Connect start signal to the multiplier's start input
   .ready(ready),// Connect 'ready' signal to the multiplier's ready output
   .c(c)        // Connect the multiplier's output 'c' to wire 'c'
);

// Clock generation: toggle the clock signal every 1 unit of time
always #1 clk <= !clk;

// Initial block to define the test sequence
initial begin
   // Initialize reset, clock, and start signals
   rst   = 0;
   clk   = 0;
   start = 0;
```

```
// Apply reset signal after 2 units of time
#2 rst = 1;


// Test Case 1: Multiply 49 and 23
a = 49;      // Set input 'a' to 49
b = 23;      // Set input 'b' to 23
start = 1;   // Start the multiplication
#4 start = 0;// Deassert start signal after 4 time units
wait (ready) $display("%d %d => %d", a, b, c); // Wait for ready and display result


// Test Case 2: Multiply -49 and 23 (using two's complement for negative number)
a = ~a + 1;  // Set input 'a' to -49 (invert and add 1 for two's complement)
b = b;       // Keep 'b' as 23
start = 1;   // Start the multiplication
#4 start = 0;// Deassert start signal after 4 time units
wait (ready) $display("%d %d => %d", a, b, c); // Wait for ready and display result


// Test Case 3: Multiply -49 and -23
a = a;       // Keep 'a' as -49
b = ~b + 1;  // Set input 'b' to -23 (invert and add 1 for two's complement)
start = 1;   // Start the multiplication
#4 start = 0;// Deassert start signal after 4 time units
wait (ready) $display("%d %d => %d", a, b, c); // Wait for ready and display result
```

```
    // Test Case 4: Multiply 49 and -23
    a = ~a + 1;  // Set input 'a' to 49 (invert and add 1 for two's complement)
    b = b;       // Keep 'b' as -23
    start = 1;   // Start the multiplication
    #4 start = 0;// Deassert start signal after 4 time units
    wait (ready) $display("%d %d => %d", a, b, c); // Wait for ready and display result

    // Terminate the simulation
    $finish(0);
  end
endmodule

// Testbench module for the CPU pipeline
module test ();
  // Declare clock and reset signals as registers
  reg clk, rst;

  // Instantiate the pipeline CPU module and connect the ports
  pipeline _cpu (
    .clk(clk),  // Connect clock signal to the CPU's clock input
    .rst(rst)   // Connect reset signal to the CPU's reset input
  );

  // Clock generation: toggle the clock signal every 1 unit of time
  always #1 clk <= !clk;
```

```
// Initial block to define the test sequence
initial begin
    // Monitor specific signals and print them to the console every time they change
    // The following monitor has been commented out but would display detailed CPU states if used
    /*
    $monitor($time, " -- ", "PC: %d\n", _cpu.PC, "INST: %b%b%b%b\n", _cpu.imem_op,
            _cpu.imem_dst, _cpu.imem_src0, _cpu.imem_src1, "READ: %b,%b\n", _cpu.dmem_data0,
            _cpu.dmem_data1, "ALU: r/%d | %b\n", _cpu.alu_ready, _cpu.alu_res,
            "MEM : %b,%b,%b,%b,%b,%b,%b,%b", _cpu._dmem.data[0], _cpu._dmem.data[1],
            _cpu._dmem.data[2], _cpu._dmem.data[3], _cpu._dmem.data[4], _cpu._dmem.data[5],
            _cpu._dmem.data[6], _cpu._dmem.data[7]);
    */

    // Use $monitor to print memory contents of the CPU's data memory every time they change
    $monitor($time, " -- ", "MEM : %b,%b,%b,%b,%b,%b,%b,%b", _cpu._dmem.data[0],
            _cpu._dmem.data[1], _cpu._dmem.data[2], _cpu._dmem.data[3], _cpu._dmem.data[4],
            _cpu._dmem.data[5], _cpu._dmem.data[6], _cpu._dmem.data[7]);
```

```
    // Initialize clock and reset signals
    clk = 0;
    rst = 0;

    // Apply reset signal after 2 units of time
    #2 rst = 1;

    // Simulation steps for instruction fetch, data fetch, execution, and write-back
    #2;  // Instruction Fetch (IFETCH)
    #2;  // Data Fetch (DFETCH)
    #2;  // Execute (EXEC)
    #2;  // Write Back (WB)
  end
endmodule
```

نتایج زیر از اجرای ماژول های تست حاصل میشوند.

```
 0 -- MEM : 0000000100000010,0000001100000100,0000000000000001,0000000100000000,xxxxxxxxxxxxxxxx,xxxxxxxxxxxxxxxx,xxxxxxxxxxxxxxxx,xxxxxxxxxxxxxxxx
15 -- MEM : 0000000100000010,0000001100000100,0000000000000001,0000000100000000,1111101100001010,xxxxxxxxxxxxxxxx,xxxxxxxxxxxxxxxx,xxxxxxxxxxxxxxxx
17 -- MEM : 0000000100000010,0000001100000100,0000000000000001,0000000100000000,1111101100001010,0000001000000100,xxxxxxxxxxxxxxxx,xxxxxxxxxxxxxxxx
29 -- MEM : 0000000100000010,0000001100000100,0000000000000001,0000000100000000,1111101100001010,0000001000000100,1111110000000011,xxxxxxxxxxxxxxxx
31 -- MEM : 0000000100000010,0000001100000100,0000000000000001,0000000100000000,1111101100001010,0000001000000100,1111110000000011,0000000100000001
```

این نتایج با مقادیر داده شده ی دستورات تست زیر و مقادیر اولیه داده های تست زیر مطابقت دارند.

```
module imem (
    input [4:0] address,
    output op,
    output [4:0] src0,
    output [4:0] src1,
    output [4:0] dst
);
    reg [15:0] data[31:0];
    assign op   = data[address][15];
    assign dst  = data[address][14:10];
    assign src1 = data[address][9:5];
    assign src0 = data[address][4:0];
    initial begin
        // instructions
        data[0] = 16'b1_00100_00001_00000;  // mem[4] = mem[0] * mem[1] = (1 + 2i) * (3 + 4i) = -5 + 10i = 11111011 00001010
        data[1] = 16'b0_00101_00000_00000;  // mem[5] = mem[0] + mem[0] = (1 + 2i) + (1 + 2i) = 2 + 4i =   00000010 00000100
        data[2] = 16'b1_00110_00010_00001;  // mem[6] = mem[2] * mem[1] = (i) * (3 + 4i) = -4 + 3i =       11111100 00000011
        data[3] = 16'b0_00111_00010_00011;  // mem[7] = mem[2] + mem[3] = (i) + (1) = 1 + i =              00000001 00000001
    end
endmodule
```

```
module dmem (
    input [4:0] read_address0,
    input [4:0] read_address1,
    input [4:0] write_address,
    output [15:0] read_data0,
    output [15:0] read_data1,
    input write,
    input [15:0] write_data,
    input clk
);
    reg [15:0] data[31:0];
    assign read_data0 = data[read_address0];
    assign read_data1 = data[read_address1];
    initial begin
        // initial data
        data[0] = {8'd1, 8'd2};  // 1 + 2i
        data[1] = {8'd3, 8'd4};  // 3 + 4i
        data[2] = {8'd0, 8'd1};  // i
        data[3] = {8'd1, 8'd0};  // 1
    end
    always @(posedge clk) begin
        if (write) data[write_address] <= write_data;
    end
endmodule
```

۴-۱ منابع و مراجع

- وبسایت گیکزفورگیکز
- وبسایت یوتیوب
- وبسایت ویکی‌پدیا