گزارش آزمایش هشتم

دانشگاه صنعتی شریف – **زمستان ۹۹**

آزمایشگاه طراحی سیستم‌های دیجیتال – دکتر اجلالی

نویسندگان:
سروش جهان‌زاد – ۹۸۱۰۰۳۸۹
علی حاتمی تاجیک – ۹۸۱۰۱۳۸۵

۱ مقدمه

در سند پیش رو مراحل انجام آزمایش هشتم درس آزمایشگاه طراحی سیستم‌های دیجیتال (ALU اعداد مختلط) شرح داده شده است. خروجی کار یک پردازنده بسیار ساده با روش پایپ‌لاین شده. از صبر و بردباری شما در مطالعه‌این مستند سپاسگزاریم.

با احترام

سروش جهان‌زاد، علی حاتمی تاجیک – **زمستان ۹۹**

۲ توضیحات کلی درباره روند کار

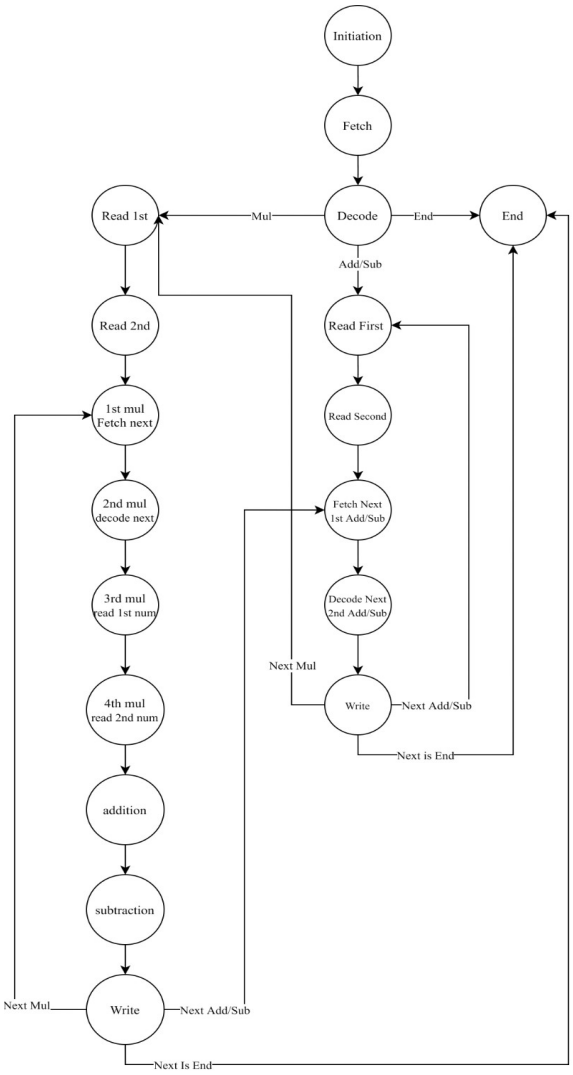
ماشین پیش‌رو

ماشینی که طراحی شده است یک ماشین حساب ساده برای اعداد مختلط است. هر عدد مختلط به این صورت در نظر گرفته شده است کـه ۸ بیت قسمت صحیح آن و ۸ بیت قسمت موهومی آن در پس هم و به صورت یک عدد ۱۶ بیتی ذخیره می‌شوند. اندازه هر کلمه نـیز ۱۶ بیت در نظر گرفته شده است. ماشین دارای ۴ عملیات کلی است. ضرب(۱۰)، جمع(۰۰)، تفریق(۰۱) و توقف (۱۱) (این عمل برای پایان دادن به رونـد انجام کار است). هر دستور مربوط به این ماشین ۱۶ بیت خواهد داشت. دو بیت به عنوان Opcode، یک بیت به عنوان ایندکس حافظه (اینکه کلمات عملگر مربوط به نیمه بالایی حافظه است یا نیمه پایینی آن)، چهاربیت برای عملگر اول، چهار بیت بـرای عملگـر دوم و پنج بیت بـرای محل ذخیره خروجی. اینکه چهاربیت وجود دارد به این دلیل است که نیمه حافظه تنها ۱۶ کلمه خواهد داشن و آدرس نسبی آن تنها دارای ۴ بیت است. مثالی از دستورات به صورت زیر آمده است:

add address1, address2 00 X AAAA BBBB CCCCC

در اینجا عمل جمع بین بین دو کلمه درون آدرس‌های XAAAA و XBBBB انجام خواهد شد و نتیجه آن درون آدرس CCCCC ریخته خواهد شد.

در ادامه شکلی از نحوه کار ماشین خواهید دید. ماشین در کل ۱۸ اسـتیت مختلـف دارد و عملیات‌هـای Fetch و Decode و بعضـا خوانـدن اعداد بعدی از حافظه در حین عملیات‌های منطقی در ALU انجام می‌شوند تا سریعا پس از تمام شدن یک دسـتور دسـتور بعـدی شـروع بـه انجام شدن کند و کمترین میزان بیکار بودن ALU را داشته باشیم (Pipeline). معماری پایپ‌لاین به این صورت پیاده سازه شده است.



همانطور که از شکل صفحه قبل مشخص است، در عملیات جمع که از شروع تا کامل شدن عملیات ۵ کلاک طول می‌کشد در کلاک‌هایی که حافظه درگیر نیست خواندن از آن صورت می‌گیرد و دستور دیکود نیز می‌شود تا برای عملیات بعد آماده باشد. در عملیات ضرب اما پا از آن نیز فراتر گذاشته شده و عدد اول و دوم هم خوانده می‌شوند تا بخشی از دستور بعدی نیز انجام شده باشد.

۳ پیاده‌سازی پیمانه‌ها

چند پیمانه اول پیمانه‌های ساده‌ای هستند که بدون توضیح اضافی از آنها عبور می‌کنیم.

جمع‌کننده / تفریق کننده

```
module addsub (
  mode, // zero for addition, one for subtraction
  first,// first operator
  second,//second operator
  result
);
  input wire mode;
  input wire signed [7:0] first, second;
  output wire signed [7:0] result;

  assign result = mode? first - second: first + second;
endmodule
```

ضرب کننده

```
module multiplier (
  first,  // first number
  second, // second number
  result  // result number => first * second
);
  input wire signed [7:0]first, second;
  output wire signed [7:0] result;

  assign result = first * second;
endmodule
```

ALU

```
module alu (
  first,  // first operator
  second, // second operator
  mul,    // is mul requested
  sub,    // is sub requested
  result  // 8 bit result
);
  input wire signed [7:0] first, second;
  input wire mul, sub;

  output wire signed [7:0] result;
  wire signed [7:0] mul_result;
  wire signed [7:0] addsub_result;

  multiplier mul_nit(
    .first(first),
    .second(second),
    .result(mul_result)
  );

  addsub addsub_unit(
    .mode(sub),
    .first(first),
```

```
    .second(second),
    .result(addsub_result)
  );
  assign result = mul? mul_result : addsub_result;
endmodule
```

حافظه

این واحد تنها برای تست در نظر گرفته شده است و قابل سنتز نخواهد بود. در این واحد یک initial استفاده شده است که در ابتدای شبیه‌سازی دستورات و کلمات داخل حافظه را می‌نویسد تا پردازنده بتواند با آن کار کند. لازم به ذکر است که پردازنده از خانه صفر حافظه شروع به خواندن می‌کند و أنقدری می‌خواند تا به دستور end برسد. به این دلیل در لبه پایین رونده فرایندها را انجام می‌دهد تا به تناقضی با پردازنده بر نخورد. در روبه‌رو هر مقداردهی حافظه چیزی که ان مقادیر نشان می‌دهد آمده است.

```
module memory (
  clk,
  readwriteN,
  address,
  data_in,
  data_out
);
  input wire [4:0] address;
  input wire clk;
  input wire [15:0] data_in;
  input wire readwriteN;
  output reg [15:0] data_out;

  reg signed [15:0] ram [0:31];

  always @(negedge clk) begin

    if (~readwriteN) begin
      ram [address] <= data_in;
    end else begin
      data_out <= ram [address];
    end
  end

  initial begin
    ram [5'b00000] <= 16'b00_1_0000_0001_10110;   // ram[22] = ram[16] + ram[17]
    ram [5'b00001] <= 16'b10_1_0000_0010_10111;   // ram[23] = ram[16] * ram[18]
    ram [5'b00010] <= 16'b10_1_0011_0100_11000;   // ram[24] = ram[19] * ram[20]
    ram [5'b00011] <= 16'b01_1_0101_0100_11001;   // ram[25] = ram[21] - ram[20]
    ram [5'b00100] <= 16'b11_0_0000_0000_00000;   // end

    ram [5'b10000] <= {8'd2,8'd4};      // ram[16] = 2 + 4i
    ram [5'b10001] <= {8'd8,8'd9};      // ram[17] = 8 + 9i
    ram [5'b10010] <= {-8'd3,8'd1};     // ram[18] = -3 + i
    ram [5'b10011] <= {-8'd1,-8'd1};    // ram[19] = -1 - i
    ram [5'b10100] <= {8'd7,8'd5};      // ram[20] = 7 + 5i
    ram [5'b10101] <= {8'd6,-8'd1};     // ram[21] = 6 - i
  end

endmodule
```

۴ پردازنده

در ادامه کد مربوط به پردازنده آمده است که طبق شکلی که پیشتر دیدیم طراحی شده است. کد به خوبی کامنت گذاری شده است و برای جلوگیری از شلوغی مستند از آوردن توضیحات بیشتر خودداری شده است:

واحد Pipeline

```
module pipeline (
  clk,            // incoming clock
  data_in,        // data read from Memory
  data_out,       // data to be written in Memory
  readwriteN,     // if 0 data will be written else data will be read
  address,        // the access address of memory
  result_of_alu,  // attributes for ALU
  first_alu,
  second_alu,
  mul,
  sub
);

input wire clk;
input wire [15:0] data_in;
output reg [15:0] data_out;
output reg readwriteN = 1;
output reg [4:0] address;

// for the ALU
input wire [7:0] result_of_alu;
output reg [7:0] first_alu, second_alu;
output reg mul, sub;

// Define parameters for increase the readability of the code
parameter INIT = 5'd0;
parameter FETCH1 = 5'd1;
parameter DECODE1 = 5'd2;
parameter ENDS = 5'd3;
parameter ASREAD1 = 5'd4;
parameter ASREAD2 = 5'd5;
parameter ASADD1 = 5'd6;
parameter ASADD2 = 5'd7;
parameter ASWRITE = 5'd8;
parameter MREAD1 = 5'd9;
parameter MREAD2 = 5'd10;
parameter MMUL1 = 5'd11;
parameter MMUL2 = 5'd12;
parameter MMUL3 = 5'd13;
parameter MMUL4 = 5'd14;
parameter MADD = 5'd15;
parameter MSUB = 5'd16;
parameter MWRITE = 5'd17;

// Current State of Machine
reg [4:0] current_state = 5'd0;
// Program Counter
reg [4:0] pc;
// Instruction Register
reg [15:0] ir;
```

```
// Instruction Parts
reg [4:0] first_address, second_address, result_address;
reg [1:0] opcode;

// Arithmetic Registers
reg [7:0]   first_re /* real part of first number */,
       first_im /* imaginary part of first number */,
       second_re,
       second_im,
       result_re,
       result_im;

// Temporary registers to keep multiplication and addition results
reg [7:0] temp [3:0];
reg [15:0] pre_num [1:0];


always @(posedge clk) begin
  case (current_state)
     INIT: begin     // initiate the module for the first time
       pc <= 5'b00001;
       readwriteN <= 1;
       address <= 5'b00000;
       current_state <= FETCH1;
     end

     FETCH1: begin   // fetch the first instruction
       ir <= data_in;
       current_state <= DECODE1;
     end

     DECODE1: begin  // decoding the first instruction
       // decode and partition the ir and free the ir for next instruction to be fetched
       opcode <= ir[15:14];
       first_address <= {ir[13],ir[12:9]};
       second_address <= {ir[13],ir[8:5]};
       result_address <= ir[4:0];
       mul <= ir[15];
       sub <= ir[14];
       // if Add or Sub was the instruction
       if (ir[15] == 0) begin
         current_state <= ASREAD1;       // next state is reading the first number
         address <= {ir[13],ir[12:9]};   // prepare address for reading first
       end

       // if End Instruction Reached
       else if (ir[15:14] == 2'b11) begin
         current_state <= ENDS;          // next state is ENDState
       end

       // if multiplication instruction were there
       else begin
         current_state <= MREAD1;        // prepare for reading first number
         address <= {ir[13],ir[12:9]};
       end
```

```
    end


    ENDS: begin
      // Nothing Happens
    end

    ASREAD1: begin  // Reading first number for addition
      first_re <= data_in[15:8];  // partition the number read into imaginary and real parts
      first_im <= data_in[7:0];
      address  <= second_address; // prepare to read the second number
      current_state <= ASREAD2;
    end

    ASREAD2: begin
      second_re <= data_in[15:8]; // partition the number read into imaginary and real parts
      second_im <= data_in[7:0];

      address <= pc;              // prepare address to fetch the next instruction
      pc <= pc + 1;               // add to program counter

      first_alu <= data_in[15:8];      // set the first parameter of the ALU to real part of first n
umber
      second_alu <= data_in[15:8];    // ... of second number

      mul <= opcode[1];           // set the mode of ALU by opcode
      sub <= opcode[0];

      current_state <= ASADD1;
    end

    ASADD1: begin
      result_re <= result_of_alu; // the result of add/sub is the real part of result

      first_alu <= first_im;
      second_alu <= second_im;

      ir <= data_in;              // fetching next instruction
      address <= result_address;  // save the address that we can store the result immediately after
last addition

      current_state <= ASADD2;
    end

    ASADD2: begin
      result_im <= result_of_alu; // the result of add/sub is the imaginary part of result

      opcode <= ir[15:14];        // decoding next instruction
      first_address <= {ir[13],ir[12:9]};
      second_address <= {ir[13],ir[8:5]};
      result_address <= ir[4:0];

      data_out <= {result_re, result_of_alu};  // prepare the result to be written
      readwriteN <= 1'b0;

      current_state <= ASWRITE;
```

```
end

ASWRITE: begin
  readwriteN <= 1'b1;

  // if Add or Sub was the instruction
  if (ir[15] == 0) begin
    current_state <= ASREAD1;       // next state is reading the first number
    address <= {ir[13],ir[12:9]};   // prepare address for reading first
  end

  // if End Instruction Reached
  else if (ir[15:14] == 2'b11) begin
    current_state <= ENDS;          // next state is ENDState
  end

  // if multiplication instruction were there
  else begin
    current_state <= MREAD1;        // prepare for reading first number
    address <= {ir[13],ir[12:9]};
  end
end

MREAD1: begin
  first_re <= data_in[15:8];  // partition the number read into imaginary and real parts
  first_im <= data_in[7:0];
  address  <= second_address; // prepare to read the second number
  current_state <= MREAD2;
end

MREAD2: begin
  second_re <= data_in[15:8]; // partition the number read into imaginary and real parts
  second_im <= data_in[7:0];

  address <= pc;              // prepare address to fetch the next instruction
  pc <= pc + 1;               // add to program counter

  first_alu <= first_re;      // set the first parameter of the ALU to real part of first number
  second_alu <= second_re;    // ... of second number

  mul <= 1'b1;           // set the mode of ALU to multiplication
  sub <= 1'b0;

  current_state <= MMUL1;
end

MMUL1: begin
  temp[0] <= result_of_alu;   // first multiplication (ac)

  first_alu <= first_im;      // prepare c and d for next multiplication
  second_alu <= second_im;

  ir <= data_in;              // fetch the next instruction

  // address of first number
  address <= {data_in[13],data_in[12:9]};
```

```
  current_state <= MMUL2;
end

MMUL2: begin
  temp[1] <= result_of_alu;   // second product (bd)

  first_alu <= first_re;      // prepare for next production
  second_alu <= second_im;

  pre_num[0] <= data_in;      // store the first number of next instruction
  address <= {ir[13],ir[8:5]};  // prepare for reading next second number

  current_state <= MMUL3;
end

MMUL3: begin
  temp[2] <= result_of_alu;   // third product (ad)

  first_alu <= first_im;      // prepare for 4th product
  second_alu <= second_re;

  pre_num[1] <= data_in;      // store the second number of next instruction
  address <= result_address;   // store the result address of the current instruction that
                 // we can decode the next instruction

  opcode <= ir[15:14];        // decoding next instruction
  result_address <= ir[4:0];//first and second address are trash because we retried the numbers

  current_state <= MMUL4;
end

MMUL4: begin
  temp[3] <= result_of_alu;   // 4th product (cb)

  first_alu <= temp[2];       // prepare for addition of third and 4th product (ad + cb)
  second_alu <= result_of_alu;

  mul <= 1'b0;                // set alu mode to addition
  sub <= 1'b0;

  current_state <= MADD;
end

MADD: begin
  result_im <= result_of_alu; // ad + cb is the imaginary part of the production

  first_alu <= temp[0];       // ac
  second_alu <= temp[1];      // bd

  sub <= 1'b1;                // set ALU mode to subtraction

  current_state <= MSUB;
end
```

```
    MSUB: begin
      result_re <= result_of_alu; // ac - bd is the real part of answer

      // prepare the writing data
      data_out <= {result_of_alu, result_im};
      readwriteN <= 1'b0;

      current_state <= MWRITE;
    end

    MWRITE: begin
      readwriteN <= 1'b1;         // change to read mode
      first_re <= pre_num[0][15:8];   // load pre-loaded numbers
      first_im <= pre_num[0][7:0];
      second_re <=  pre_num[1][15:8];
      second_im <=  pre_num[1][7:0];

      address <= pc;              // prepare address to fetch the next instruction
      pc <= pc + 1;               // add to program counter

      first_alu <= first_re;      // set the first parameter of the ALU to real part of first number
      second_alu <= second_re;    // ... of second number

      // if Add or Sub was the instruction
      if (ir[15] == 0) begin
        current_state <= ASADD1;       // next state is adding real parts because numbers are read

        mul <= opcode[1];           // set the mode of ALU by opcode
        sub <= opcode[0];
      end

      // if End Instruction Reached
      else if (ir[15:14] == 2'b11) begin
        current_state <= ENDS;          // next state is ENDState
      end

      // if multiplication instruction were there
      else begin
        current_state <= MMUL1;        // prepare for reading first number

        mul <= 1'b1;                   // set the mode of ALU to multiplication
        sub <= 1'b0;
      end
    end
    default: begin
      pc <= 5'b00001;
      readwriteN <= 1;
      address <= 5'b00000;
      current_state <= FETCH1;
    end
  endcase
end

endmodule
```

پردازنده اصلی

این پردازنده از یک ALU و یک پایپ‌لاین تشکیل شده است و ورودی و خروجی‌های آن مربوط به کارهای حافظه است:

```
module processor (

  clk,            // incoming clock
  data_in,        // data read from Memory
  data_out,       // data to be written in Memory
  readwriteN,     // if 0 data will be written else data will be read
  address         // the access address of memory
);

input wire clk;
input wire [15:0] data_in;
output wire [15:0] data_out;
output wire readwriteN;
output wire [4:0] address;

wire mul, sub;
wire [7:0] first, second, result;

pipeline pipe(
  .clk(clk),            // incoming clock
  .data_in(data_in),        // data read from Memory
  .data_out(data_out),       // data to be written in Memory
  .readwriteN(readwriteN),     // if 0 data will be written else data will be read
  .address(address),        // the access address of memory
  .result_of_alu(result),  // attributes for ALU
  .first_alu(first),
  .second_alu(second),
  .mul(mul),
  .sub(sub)
);

alu alu_unit(
  .first(first),  // first operator
  .second(second), // second operator
  .mul(mul),    // is mul requested
  .sub(sub),    // is sub requested
  .result(result)  // 8 bit result
);

endmodule
```

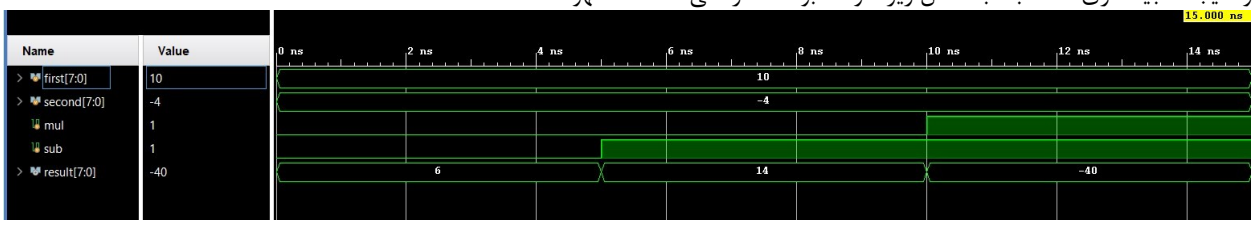
۵ تست مدار

تست ALU

با یک تست ساده ابتدا صحت کار ALU را می‌سنجیم:

```
`timescale 1ns / 1ps

module alu_tb();
  reg signed [7:0] first = 10, second = -4;
  reg mul = 0, sub = 0;
  wire signed [7:0] result;

  alu alu_unit(
    first,  // first operator
    second, // second operator
    mul,    // is mul requested
    sub,    // is sub requested
    result  // 8 bit result
  );

  initial begin
    #5 sub = 1;
    #5 mul = 1;
    #5 $finish;
  end
endmodule
```

و نتیجه شبیه‌سازی تست بالا به شکل زیر خواهد بود که درستی داده‌ها مشهود است:



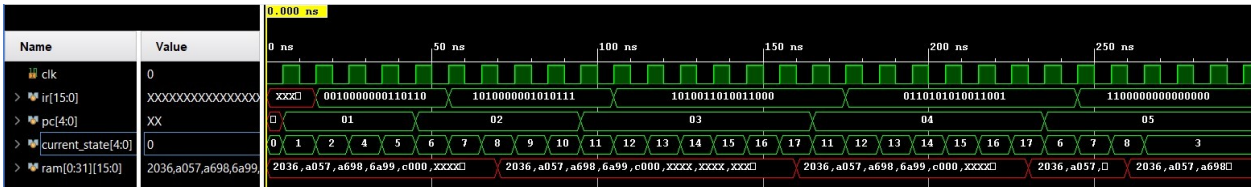
تست پردازنده

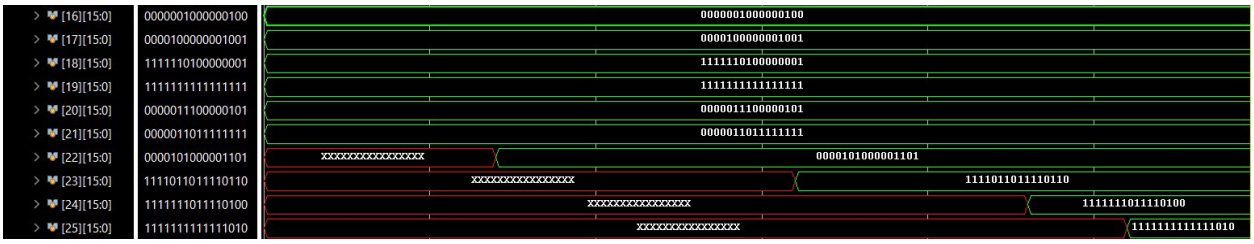
برای تست پردازنده کافیست حافظه بارگزاری شده را به یک واحد پردازنده متصل کنیم. باقی کارها را پردازنده انجام خواهد داد و کافیست تا صبر کنیم تا کار پردازنده به پایان برسد:

```
`timescale 1ns / 1ps

module pipe_tb();
  wire [4:0] address;
  reg clk = 0;
  wire [15:0] data_in;
  wire readwriteN;
  wire [15:0] data_out;

  initial begin
    forever #5 clk = ~clk;
  end
```

```
  memory memory_unit(
    .clk(clk),
    .readwriteN(readwriteN),
    .address(address),
    .data_in(data_in),
    .data_out(data_out)
  );

  processor processor_instant(
    .clk(clk),            // incomming clock
    .data_in(data_out),        // data read from Memory
    .data_out(data_in),       // data to be written in Memory
    .readwriteN(readwriteN),     // if 1 data will be writen else data will be read
    .address(address)         // the access address of memory
  );

  initial begin
    #2000 $finish;
  end
endmodule
```

و نتیجه کار پردازنده به شکل زیر خواهد بود. در تصویر اول نحوه خوانده شدن دستورات و استیت‌های پردازنده به نمایش در آمده است و در تصویر بعدی داده‌های موجود در RAM در قسمت ۱۶ تا ۲۵ آن (جایی که اعداد اولیه و جایی که اعداد نتیجه طبق مداردهی‌های حافظه) نمایش داده شده است:





برای راحتی تطابق نتایج با مقادیری که باید باشند محاسبات عددی و مقادیر باینری آن در پایین محاسبه شده است:

```
ram[22] = (2+4i) + (8+9i) = 10 + 13i = 00001010 + 00001101i
ram[23] = (2+4i) * (-3+i) = -10 - 10i = 11110110 + 11110110
ram[24] = (-1-i) * (7+5i) = -2 - 12i = 11111110  + 11110100i
ram[25] = (6-i ) - (7+5i) = -1 - 6i  = 11111111 + 1111010i
```