به نام خدا

آزمایشگاه طراحی سیستم‌های دیجیتال

گزارش کار دهم

پیاده‌سازی یک پردازنده ساده

دانشکده مهندسی کامپیوتر

دانشگاه صنعتی شریف

بهار ۱۴۰۱

استاد:

علیرضا اجلالی

دستیار آموزشی:

سحر رضاقلی

نویسندگان:

هیربد بهنام

۹۹۱۷۱۳۳۳

عرفان مجیبی

۹۹۱۰۵۷۰۷

علی نظری

۹۹۱۰۲۴۰۱

فهرست

مقدمه

در این آزمایش می‌خواهیم یک پردازنده طراحی کنیم که دارای معماری پشته‌ای است. یعنی کل داده‌های ما روی stack قرار می‌گیرند و سپس با هر دستور، یک یا دو داده بالای استک هستند که پردازش روی آن‌ها انجام می‌شود و نتیجه یا در حافظه و یا در همان استک گذاشته می‌شود. دستورهای حافظه هم در این پردازنده موجود است که یا از خانه بالایی استک به حافظه می‌برد و یا از خانه‌ای از حافظه، به خانه بالایی استک می‌آورد. پشته هم دارای ۸ خانه ۸ بیتی است. حافظه اصلی هم دارای ۲۵۶ خانه ۸ بیتی است. همانطور که گفته شده است هم ۸ خانه پایانی آن برای دستورات I/O هستند و نمی‌توان در آن‌ها داده‌ای قرار داد.

گزارش آزمایش

بخش ۱. طراحی پردازنده

بخش حافظه

برای این بخش، همانطور که گفته شد، ما دو حافظه جداگانه داریم که یکی حافظه مربوط به data است و دیگری حافظه مربوط به instruction که در ماژول‌های جداگانه‌ای طراحی این بخش‌ها انجام شده است که کدهای این بخش‌ها پایین‌تر قابل مشاهده است.

```
module DataMemory(
    input [7:0] d_address,
    inout [7:0] d_data,
    input write
);
    reg [7:0] d_storage [255:0];

    assign d_data = (~write && d_address < 'hF8) ? d_storage[d_address]
    : 8'hzz;

    always @(posedge write)
        d_storage[d_address] = d_data;

endmodule
```

```
module InstructionMemory (
    input [7:0] i_address,
    output reg [11:0] i_data
);
    reg [11:0] i_storage [255:0];

    always @(*)
        i_data = i_storage[i_address];
endmodule
```

نخست بخش instruction-memory را توضیح می‌دهیم و با توضیح ورودی‌ها و خروجی‌ها شروع می‌کنیم. ورودی i-address در واقع همان آدرس ۸ بیتی ما است که می‌خواهیم به محتویات دیتای موجود در این آدرس از حافظه دسترسی داشته باشیم.
خروجی i-data هم شامل داده ۱۲ بیتی است که در آن آدرس از حافظه وجود دارد و به این دلیل به شکل reg تعریف شده است چون می‌خواهیم در بدنه رفتاری به آن مقدار بدهیم.

سپس آرایه‌ای ۲۵۶ خانه‌ای که هر خانه آن به طول ۱۲ بیت است را می‌سازیم و این مورد، حافظه ما را تشکیل می‌دهد.
سپس در یک بلوک always با هر بار تغییر i-address مقدار درون آدرسی که خواسته شده است را درون خروجی ماژول می‌گذاریم.

حال ماژول data-memory را توضیح می‌دهیم. ورودی d-address که آدرس ۸ بیتی خانه‌ای از حافظه است که می‌خواهیم به آن دسترسی داشته باشیم.
مورد بعدی، پورت از نوع inout است با نام d-data که شامل داده ۸ بیتی است که از حافظه داده شده، خوانده شده است. و یا داده‌ای است که قرار است در حافظه گذاشته شود.
ورودی تک بیتی write هم برای این است که اگر یک باشد، یعنی می‌خواهیم مقداری در حافظه بریزیم و اگر صفر باشد، یعنی می‌خواهیم داده را از حافظه بخوانیم.
سپس یک آرایه ۲۵۶ خانه‌ای ساختیم که هر خانه آن دارای ۸ بیت است و خب این نشان‌دهنده حافظه داده است .
سپس با assign مقدار d-data را مشخص کردیم تا با خاصیت‌های وریلاگ، بتوانیم در هر لحظه که سمت راست عوض می‌شود، سمت چپ هم عوض شود و در نتیجه مقدار واقعی d-data را در هر لحظه داشته باشیم. مقدار این d-data به شکل زیر مشخص می‌شود:
اگر هم wire صفر باشد و هم d-address کمتر از F۸ باشد، یعنی بخواهیم از حافظه بخوانیم و مربوط به بخش I/O هم نباشد، در این صورت، مقدار خانه d-address را از آرایه d-storage در d-data قرار می‌دهیم و در غیر این صورت، d-data را قطع می‌کنیم و یا همان z می‌کنیم تا با توجه به inout بودن این d-data در سمت دیگر بتوان این سیم را به اصطلاح drive کرد و دیگر به مشکل conflict نخوریم.
در نهایت هم در یک بلوک always که به لبه بالارونده write حساس است، مقدار d-data را از خانه d-address مربوط به d-storage می‌خواهیم.

بخش I/O

```
module IO(
    input [7:0] d_address,
    inout [7:0] d_data,
    input write,
    input [7:0] X,
    output reg [7:0] Y
);

    assign d_data = (~write && d_address == 'hF8) ? X : 8'hzz;

    always @(posedge write)
        if (d_address == 8'hFF)
            Y = d_data;

endmodule
```

نخست به توضیح ورودی و خروجی‌های این ماژول می‌پردازیم:

- ورودی d-address که شامل آدرس memory-map شده I/O است که قرار است با آن کار کنیم.
- پورت inout با نام d-data هم داریم که شامل مقداری است که قرار است از حافظه خوانده شود و یا مقداری است که می‌خواهیم در حافظه نوشته شود.

- ورودی write هم مانند ماژول قبلی مشخص می‌کند که می‌خواهیم در حافظه بنویسیم یا از آن بخوانیم.
- ورودی X هم که مشخص شده است، همان ورودی‌ای است که از محیط بیرون و توسط عوامل خارجی به این واحد داده می‌شود.
- خروجی Y هم برای این است که بخواهیم خروجی‌ای را به واحد I/O بدهیم تا آن برای ما نمایش دهد و برای اینکه در بلاک می‌خواهیم از آن استفاده کنیم، پس جنس آن را به شکل reg تعریف می‌کنیم.

سپس، به کمک یک دستور توصیف جریان داده پس از تغییر ورودي هاي لازم همین دستور، تغییر زیر را اعمال می‌کند: اگر هم write صفر بود و هم d-address برابر از F۸ بود تنها در آن صورت، مقدار ورودي توسط کاربر را در پورت به نام d-data می‌ریزیم، وگرنه، از این سمت، d-data را قطع می‌کنیم تا با توجه به inout بودن d-data ،سمت دیگر بتواند این سیم را drive کند و خطاي Conflict پیش نیاید. سپس، در یک بلوک always که با لبه ي بالارونده ي سیگنال write فعال می‌شود، اگر آدرس داده شده، آدرس متصل به دستگاه خروجی بود، مقدار d-data را در خروجی Y می‌ریزیم تا توسط کاربر یا انسان، قابل رویت باشد.

بخش پردازنده

حال به بخش اصلی می‌رسیم که باید پردازنده را طراحی کنیم و قطعه کد آن به شکل زیر است:

```
`include "data_memory.v"
`include "instruction_memory.v"
`include "io.v"

module CPU (
    input clk,
    input reset,
    input [7:0] X,
    output [7:0] Y,
    output reg error
);

    reg [7:0] PC, nPC; // program counters
    wire [11:0] IR; // instruction register

    reg [2:0] SP, nSP; // stack pointer (points to empty cell)
    reg [7:0] push; // variable to hold the value that must be pushed
    reg signed [7:0] stack [7:0];

    reg Z; // zero flag (for +/-)
    reg S; // sign flag (for +/-)

    // Assign address wires
    wire [7:0] i_address = PC;
```

```
25      wire [7:0] d_address = IR[7:0];
26      // Only write on POP instruction
27      wire write = (IR[11:8] == 4'b0010);
28      wire [7:0] d_data = (IR[11:8] == 4'b0010) ? stack[SP-1] : 8'
    bzzzzzzzz;
29
30      InstructionMemory instruction_memory (i_address, IR);
31      DataMemory data_memory (d_address, d_data, write);
32      IO io (d_address, d_data, write, X, Y);
33
34      always @(*)
35      begin
36          if (IR[11:8] == 4'b0000) begin // PUSHC : push constant
37              nPC = PC+1;
38              nSP = SP+1;
39              push = IR[7:0];
40          end else if (IR[11:8] == 4'b0001) begin // PUSH : push from
    memory
41              nPC = PC+1;
42              nSP = SP+1;
43              push = d_data;
44          end else if (IR[11:8] == 4'b0010) begin // POP : pop to memory
45              nPC = PC+1;
46              nSP = SP-1;
47              push = 0;
48          end else if (IR[11:8] == 4'b0011) begin // JUMP : pop from stack
     to PC
49              nPC = stack[SP-1];
50              nSP = SP-1;
51              push = 0;
52          end else if (IR[11:8] == 4'b0100) begin // JZ : jump on zero
    flag
53              nPC = Z ? stack[SP-1] : PC+1;
54              nSP = Z ? SP-1 : SP;
55              push = 0;
56          end else if (IR[11:8] == 4'b0101) begin // JS : jump on sign
    flag
57              nPC = S ? stack[SP-1] : PC+1;
58              nSP = S ? SP-1 : SP;
59              push = 0;
60          end else if (IR[11:8] == 4'b0110) begin // ADD : add two top
    elements
61              nPC = PC+1;
62              nSP = SP-1;
```

```
63              push = stack[SP-2] + stack[SP-1];
64          end else if (IR[11:8] == 4'b0111) begin // SUB : add two top
    elements
65              nPC = PC+1;
66              nSP = SP-1;
67              push = stack[SP-2] - stack[SP-1];
68          end else begin
69              nPC = PC;
70              nSP = SP;
71              push = 0;
72              $display("INVALID INSTRUCTION %b", IR);
73          end
74      end
75
76      // At posedge clock go to next instruction
77      always @(posedge reset, posedge clk)
78      begin
79          // On reset clear registers
80          if (reset) begin
81              PC <= 0;
82              SP <= 0;
83              error <= 0;
84          end else begin // otherwise go to next instruction
85              PC <= nPC;
86              SP <= nSP;
87              // If ADD/SUB, lets set the flags
88              if (IR[11:8] == 4'b0111 || IR[11:8] == 4'b0110) begin
89                  stack[SP-2] <= push;
90                  Z = ~|push;
91                  S = push[7];
92                  // If two operands with same signs add up, or two
93                  // operands with opposie signs subtract, but result
94                  // has different sign with first one -> overflow
95                  if (stack[SP-2][7] == (stack[SP-1][7] ^ IR[8]) && stack[
    SP-2][7] != S)
96                      error = 1;
97              end
98              // Otherwise, just push the value
99              // on pop, it actually clears top of stack
100             else
101             begin
102                 stack[SP] <= push;
103                 // If negative number is pushed -> error
104                 if (IR[11:8] == 4'b0001 & push[7] == 1)
```

```
105                         error = 1;
106                 end
107             end
108         end
109 endmodule
```

نخست به توضیح ورودی‌ها و خروجی‌ها می‌پردازیم:

- ورودي clk
- ورودي reset که هر گاه ۱ شود، ریست انجام می‌شود.
- ورودي ۸ بیتی X که معادل bus ورودي به CPU است.
- خروجی ۸ بیتی Y که معادل bus خروجی از CPU است.
- خروجی تک بیتی error که وقتی یک می‌شود که خطاي overflow یا منفی بودن ورودي، رخ دهد؛ چون می‌خواهیم از بدنه ي رفتاري در آن مقدار بنویسیم، reg تعریف شده است.

سپس چند متغیر wire یا reg دیگر که در آینده به آن ها نیاز داریم را تعریف می‌کنیم و به توضیح آن ها در این بخش می‌پردازیم:

- رجیستر ۸ بیتی PC که به آدرس دستورالعمل فعلی اشاره دارد.
- رجیستر ۸ بیتی nPC که به آدرس دستورالعمل بعدي اشاره دارد.
- رجیستر ۱۲ بیتی IR که دستورالعمل فعلی را در خود نگه می‌دارد.
- رجیستر ۳ بیتی SP که اندازه‌ی آن اجازه می‌دهد تا ۲ به توان ۳ ،یعنی ۸ خانه، در پشته داشته باشیم که آدرس خانه‌ای از پشته که خالی است را نگه می‌دارد. اگر کل خانه ها پر باشد هم صفر را درون خود نگه می‌دارد
- رجیستر ۳ بیتی nSP که آدرسی که SP باید به آن در مرحله ي بعد تغییر کند را نگه می‌دارد.
- رجیستر ۸ بیتی push که شامل مقداري خواهد بود که باید در stack مان push شود.
- آرایه ي ۸ تایی از رجیسترهاي ۸ بیتی علامت دار به اسم stack براي نگهداري خانه هاي پشته.
- رجیستر Z که معادل zero flag است.
- رجیستر S که معادل sign flag است.
- سیم ۸ بیتی i-address که شامل آدرس دستورالعمل است و به شکل پیوسته، مقدارش توسط PC تعیین می‌شود.
- سیم ۸ بیتی d-address که شامل آدرس داده است و به شکل پیوسته، مقدارش توسط هشت بیت کم ارزش دستورالعمل تعیین می‌شود.
- سیم write که به ورودي واحد d-memory و نیز io-handler داده می‌شود و مقدار آن تنها وقتی یک است که opcode برابر ۰۰۱۰ به معناي Pop باشد، زیرا فقط در دستور Pop است که کار نوشتن در حافظه و یا I/O را انجام می‌دهیم.

- سیم d-data که به ورودي واحد d-memory و نیز io-handler داده می‌شود که اگر آپکد ۰۰۱۰ به معنای pop باشد، مقداري که باید Pop شود یعنی خانه ي SP-1 ام استک در آن نوشته شد، وگرنه قطع است و از سمت دیگر، مقدار به دست آمده و خوانده شده از حافظه، در آن نوشته می‌شود؛ قطع کردیم تا خطاي Conflict پیش نیاید.

سپس، نمونه گیري و portmap واحدهاي حافظه ي دستورالعمل، حافظه ي داده و نیز IO-handler انجام گرفته است. سپس، در یک بلاک always که با هر تغییر در مقادیري که می‌خواند ، فراخوانی می‌شود و مربوط به به دست آوردن nPC و nSP از روي PC و SP است. حال حالت های مختلف آپکد را توضیح می‌دهیم:
اگر opcode برابر با ۰۰۰۰ بود یعنی pushc داریم:

- اول PC را برای مرحله بعد، یکی جلو می‌بریم.
- SP را یکی زیاد می‌کنیم زیرا یک داده دارد Push می‌شود و باید به نوعی، یک خانه به بالاي پشته افزوده شود.
- مقدار ۸ بیت کم ارزش دستورالعمل که شامل عدد ثابتی است که باید push شود را در push می‌ریزیم تا بعدا push شود.

حال اگر opcode برابر با ۰۰۰۱ بود یعنی با push رو به رو هستیم:

- pc را براي مرحله ي بعد، یکی به جلو می‌بریم.
- SP را یکی زیاد می‌کنیم زیرا یک داده دارد Push می‌شود و باید به نوعی، یک خانه به بالاي پشته افزوده شود.
- مقدار d-data که از حافظه خوانده شده است را در push می‌ریزیم تا بعدا push شود

حال اگر opcode برابر با ۰۰۱۰ بود یعنی با pop رو به رو هستیم:

- pc را براي مرحله ي بعد، یکی به جلو می‌بریم.
- SP را یکی کم می‌کنیم زیرا یک داده دارد pop می‌شود و باید به نوعی، یک خانه از بالاي پشته کم شود.
- مقدار صفر را در push می‌ریزیم

اگر opcode برابر با ۰۰۱۱ بود یعنی با jump رو به رو هستیم:

- pc را براي مرحله ي بعد، یکی به جلو می‌بریم.
- SP را یکی کم می‌کنیم زیرا یک داده دارد pop می‌شود و باید به نوعی، یک خانه از بالاي پشته کم شود.
- مقدار صفر را در push می‌ریزیم

اگر opcode برابر با ۰۱۰۰ باشد یعنی با JZ رو به رو هستیم:

- اگر Z یک بود، PC را برابر خانه ي بالایی پشته می‌کنیم وگرنه به طور پیشفرض، اجراي برنامه را از سر گرفته و PC را یکی زیاد می‌کنیم.

- اگر Z یک بود، SP را یکی کم می‌کنیم وگرنه چون نیازي به Pop نیست، مجددا SP را در SP می‌ریزیم.
- مقدار صفر را در push می‌ریزیم.

اگر opcode برابر با ۰۱۰۱ باشد یعنی با JS طرف هستیم:

- اگر S یک بود، PC را برابر خانه ي بالایی پشته می‌کنیم وگرنه به طور پیشفرض، اجراي برنامه را از سر گرفته و PC را یکی زیاد می‌کنیم.
- اگر S یک بود، SP را یکی کم می‌کنیم ، وگرنه چون نیازي به Pop نیست، مجددا SP را در SP می‌ریزیم.
- مقدار صفر را در push می‌ریزیم.

اگر opcode برابر با ۰۱۱۰ بود یعنی با ADD طرف هستیم:

- pc را براي مرحله ي بعد، یکی به جلو می‌بریم.
- SP را یکی کم می‌کنیم زیرا یک داده دارد pop می‌شود و باید به نوعی، یک خانه از بالاي پشته کم شود. در واقع ۲ تا کم می‌شود و یکی اضافه می‌شود و در نهایت، همان یکی کم می‌شود.
- مقدار جمع خانه ي بالایی و خانه ی پایینش در پشته را در push می‌ریزیم تا بعدا push شود.

اگر opcode برابر با ۰۱۱۱ باشد یعنی با sub طرف هستیم:

- pc را براي مرحله ي بعد، یکی به جلو می‌بریم.
- SP را یکی کم می‌کنیم زیرا یک داده دارد pop می‌شود و باید به نوعی، یک خانه از بالاي پشته کم شود. در واقع ۲ تا کم می‌شود و یکی اضافه می‌شود و در نهایت، همان یکی کم می‌شود.
- مقدار تفریق خانه ي بالایی و خانه ی پایینش در پشته را در push می‌ریزیم تا بعدا push شود.

اگر هم opcode هیچ کدام از موارد بالا نباشد، هیچ کاری انجام نمی‌دهیم.
سپس داخل یک بلاک always که به لبه بالارونده clock و reset حساس است، می‌رسیم:
اگر ریست یک بود، هم PC و هم SP و هم error را صفر می‌کنیم.
وگرنه یعنی با لبه ي بالارونده ي کلاك به داخل این بلاك always آمده‌ایم و در نتیجه باید به دستور بعدي برویم:

- ابتدا PC را برابر nPC و SP را برابر nSP می‌کنیم در واقع، مقادیر PC و SP اي که براي حالت بعدي در نظر گرفته شده بودند را در خود PC و SP می‌ریزیم
- سپس، اگر آپکد ۰۱۱۰ و یا ۰۱۱۱ بود یعنی باید مقادیر فلگ هایمان را هم در نظر داشته باشیم که ممکن است عوض شوند.
- ابتدا مقداري که باید در استکمان push می‌شد و پس از محاسبه، آن را در push ریخته بودیم را در خانه ي SP-2 ام استک می‌ریزیم. دلیل استفاده از SP-2 این است که در حقیقت استک پوینتر به یک خانه بالاتر از بالاترین خانه ي استک اشاره می‌کند و باید SP-1 می‌نوشتیم، اما چون انتساب nSP به شکل نان بلاکینگ صورت گرفته است، در اصل مقدار SP هنگام اجراي این خط، یکی بیشتر از مقداري ست که باید باشد.

- سپس Reduction OR بیت هاي push را در پرچم Z می‌ریزیم؛ در نتیجه، اگر حتی یک بیت push یک باشد Z یک خواهد شد وگرنه صفر است
- سپس بیت ۷ ام push را در پرچم S می‌ریزیم.
- حال، در صورت رخ دادن سرریز، error را یک می‌کنیم. سرریز وقتی رخ می‌دهد که دو عبارت هم علامت با هم جمع و یا دو عبارت ناهم علامت از یکدیگر کم شوند که این دو حالت در بیت شماره ي ۸ دستورالعمل متفاوت اند و به کمک XOR عبارت مربوطه تشکیل داده شده است ولی حاصل محاسبه با اپراند اول ناهم علامت باشند.
- اگر دستور جمع یا تفریق نباشد هم برای push چند مورد را باید در نظر داشته باشیم.
- آنچه در push بوده است را push می‌کنیم.
- حال اگر عددي که قرار است push شود منفی باشد خطا می‌دهیم و error را یک می‌کنیم.

بخش ۲. تست و شبیه‌سازی مدار

بررسی عملکرد مدار

کد این بخش به شکل زیر است:

```
`include "cpu.v"
module CPU_Test;
    reg clk;
    reg reset;
    reg signed [7:0] X;
    wire signed [7:0] Y;
    wire error;

    initial clk = 1;
    always #5 clk = ~clk;

    CPU cpu (clk, reset, X, Y, error);

    // put instructions in instruction memory
    initial begin
        cpu.instruction_memory.i_storage[00] = 12'h_1_F8; // PUSH X
        cpu.instruction_memory.i_storage[01] = 12'h_0_17; // PUSHC 23
        cpu.instruction_memory.i_storage[02] = 12'h_6_00; // ADD
        cpu.instruction_memory.i_storage[03] = 12'h_2_AA; // POP TMP
        cpu.instruction_memory.i_storage[04] = 12'h_1_AA; // PUSH TMP
        cpu.instruction_memory.i_storage[05] = 12'h_1_AA; // PUSH TMP
        cpu.instruction_memory.i_storage[06] = 12'h_6_00; // ADD
        cpu.instruction_memory.i_storage[07] = 12'h_0_0C; // PUSHC 12
        cpu.instruction_memory.i_storage[08] = 12'h_7_00; // SUB
        cpu.instruction_memory.i_storage[09] = 12'h_2_FF; // POP Y
        cpu.instruction_memory.i_storage[10] = 12'h_0_0A; // PUSHC 10
        cpu.instruction_memory.i_storage[11] = 12'h_3_0A; // JUMP 10
    end

    initial begin
        X = $random;
        reset = 1;
        #10;
        reset = 0;
        #150;
        $finish;
    end

```

```
39      always @(Y)
40          $display("for input X = %4d -> output Y = %4d (error = %b)", X,
        Y, error);
41
42  endmodule
```

در تست بنچ ابتدا ورودي هاي مدار از نوع رجيستر و خروجی هاي مدار از نوع wire تعريف شده اند. سپس، در يک بلاک initial ابتدا مقدار کلاک را يک می‌کنيم و سپس در يک بلاک always هر ۵ نانوثانيه يک بار، مقدار کلاک را not مقدار قبلی اش می‌کنيم تا بدين وسيله، دوره ي تناوب کلاک ۱۰ نانوثانيه شود. سپس، نمونه گيري از CPU اصلی و portmap نيز انجام شده است. حال در يک بلاک initial ،دستورات متعددي را در حافظه مان در ابتداي کار جهت اجرا شدن توسط پردازنده درج می‌کنيم. اين برنامه که نوشته شده است، به شکل زير است:

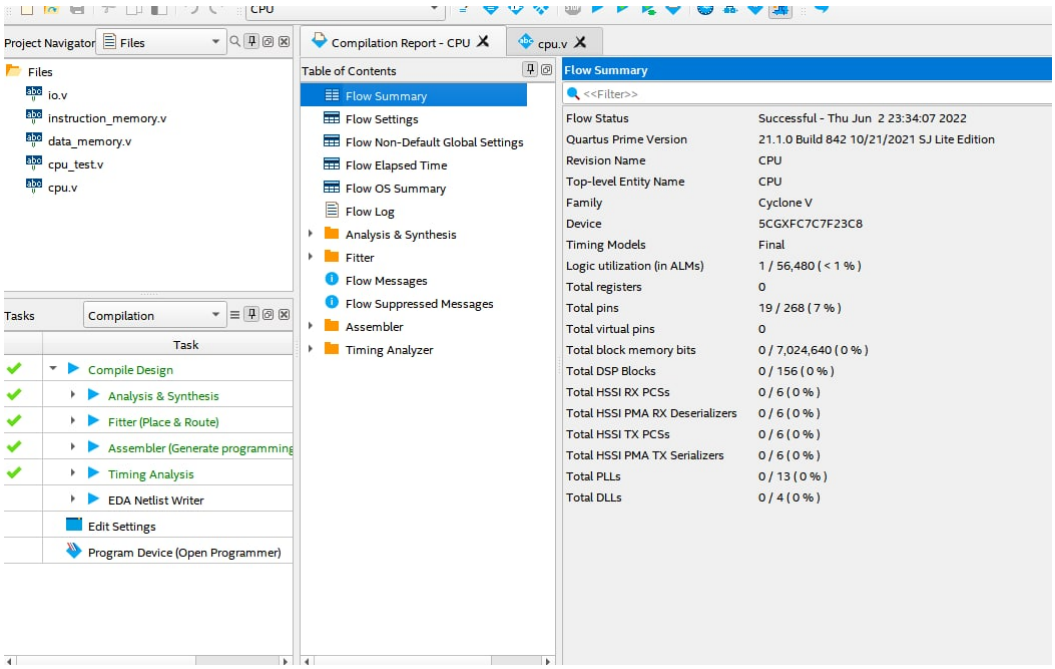
ابتدا X را از I/O ورودي، در پشته مان قرار می‌دهيم، سپس عدد ۲۳ را در بالاي آن push می‌کنيم و سپس با دستور add ، دو مقدار درون پشته برداشته شده و مقدار X + 23 به جاي آن قرار می‌گيرد. اين مقدار را در يکی از خانه هاي حافظه ي داده مان pop می‌کنيم. اين مقدار پاپ شده را مثلا TMP می‌ناميم.

سپس ۲ بار TMP را در استک پوش می‌کنيم و سپس آن دو را جمع می‌زنيم تا برداشته شوند. با اين کار مقدار 2x + 46 را داريم.

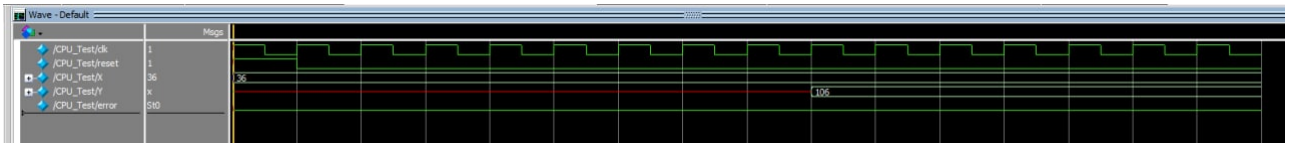
سپس عدد ثابت ۱۲ را بالای آن پوش می‌کنيم. و دستور تفريق را می‌دهيم تا بين اين دو مقدار انتهايی استک، عمليات تفريق صورت گيرد. حال درون پشته، مقدار 2x + 46 - 12 را داريم. اين همان چيزی است که صورت آزمايش از ما می‌خواهد. سپس حاصل به دست آمده را هم در خروجی I/O ميا‌د pop ميشه تا توسط کاربر قابل ديدن باشد.

در نهايت، عدد ثابت ۱۰ را به پشته مان push کرده و دستور jump را فراخوانی می‌کنيم تا اين عدد را pop کند و در PC قرار دهد تا دوباره به دستور قبلی برگرديم و همين جا مدام loop بزنيم و به دستورات بعدي که unknown اند، همينطور به پيش نرويم.

سپس در يک بلاک initial يک عدد تصادفی توليد می‌کنيم و reset را انجام می‌دهيم تا همه چيز پاک شود و در نهايت مدتی صبر می‌کنيم تا خروجی را مشاهده کنيم. عدد ۱۵۰ هم برای اين گذاشتيم تا مطمئن باشيم که کار دستور قبلی تمام شده. آن بخش display هم که برای اين است که واضح تر بتوان خروجی و داشتن يا نداشتن ارور را مشاهده کرد. با مشاهده کنسول و همينطور Waveform که پايين تر گذاشته می‌شود، صحت عملکرد مدار تاييد می‌شود. همچنين با ابزار کوآرتوس، سنتز پذير بودن مدار را هم بررسی کرديم و با موفقيت انجام شد.



شکل ۱: کامپایل موفقیت آمیز



شکل ۲: شکل موج

نتیجه‌گیری

پس در این آزمایش توانستیم یک پردازنده بسازیم که دارای تعدادی دستورالعمل است و با تعریف حافظه داده و حافظه دستور و بخش I/O به شکل جداگانه، توانستیم ارتباط همه بخش‌ها را ایجاد کنیم و در نهایت پردازنده‌ای ساختیم.