پیاده‌سازی پردازنده MIPS

اهداف

۱- یادگیری مفاهیم اصلی معماری کامپیوتر
۲- یادگیری مفاهیم خط لوله در پردازنده
۳- تاثیرات اجزای مختلف پردازنده در کارایی آن و نحوه افزایش آن
۴- یادگیری طراحی سخت افزار و کدنویسی هافمن
۵- نحوه کدنویسی Verilog با قابلیت سنتز
۶- نحوه عیب‌یابی و تست مدارهای سخت افزاری طراحی شده

توضیحات کلی

۱- در این آزمایش باید یک پردازنده MIPS ساده که دارای ۱۸ دستور العمل اصلی است، پیاده‌سازی گردد.
۲- معماری اصلی این پردازنده را طراحی و کد Verilog آن را (با توضیحات کامل) به طور سنتزشدنی نوشته شود.
۳- ابتدا کد را با استفاده از ModelSim شبیه سازی و نتایج آن را در آزمایشگاه نشان دهید. سپس کد را با استفاده از Quartus II سنتز کنید (نتایج سنتز باید در گزارش کار بیاید) و سپس برد را برنامه‌ریزی کنید.
۴- برای هر قسمت از این پردازنده (هر ماژول) باید یک ماژول تست نوشته و آن را شبیه‌سازی و تست نمایید.
۵- پس از طراحی تمامی ماژول‌های پردازنده، ماژول‌ها را به یکدیگر متصل نمایید و کل پردازنده را شبیه‌سازی و تست نمایید.
۶- برای تست نهایی پردازنده یک ماژول سطح بالا (Testbench) طراحی کنید که کد دودویی یک عملیات (مانند حاصلضرب دو عدد) را داخل Program ROM قرار دهید و آن را خط به خط اجرا نمایید. تا در نهایت جواب نهایی حاصل شود. برای تبدیل کد اسمبلی به کد ماشین می توانید از برنامه‌ای که به شما داده می شود استفاده کنید.
۷- همچنین برای تست باید یک کلاک دستی به سیستم اضافه کنید، به طوری که با هر بار فشردن KEY[0] یک کلاک زده می شود، که از این کلاک برای تست استفاده می شود. بدین شکل که اگر SW[0] صفر باشد پردازنده با کلاک برد (حالت عادی) و اگر SW[0] یک باشد با کلاک دستی (حالت تست) کار می کند. هنگامی که در حالت تست قرار می گیرد باید دستوری که داخل هر پایپ از پردازنده قرار دارد را برروی 7-Segmentهای متناظر (۵ 7-Seg به ترتیب) نشان دهید.

دستور کار

پردازنده‌ای که در این آزمایش طراحی و پیاده سازی می گردد، یک پردازنده MIPS ساده شده است که دارای ۱۸ دستور العمل اصلی است. این پردازنده قابلیت انجام علمیات های ریاضی (ADD, ADDI, SUB,SUBI)، عملیات‌های منطقی (AND, OR, NOR, XOR)، عملیات های شیفت (SLA, SLL, SRA, SRL)، عملیات خواندن و نوشتن در حافظه (LD, ST)، عملیات پرش شرطی (BEZ, BNE) و پرش غیرشرطی (JMP) را دارد. لیست عملیات ها به همراه جزئیات آنها در جدول ۱ آورده شده است.

R-type Instructions		Description	Bits				
			31:26	25:21	20:16	15:11	11:00
			OP Code	RS1	RS2	RD	--
0	NOP	No Operation	000000	rs1 (0)	rs2 (0)	rd (0)	00000000000
1	ADD	Addition	000001	rs1	rs2	rd	00000000000
3	SUB	Subtraction	000011	rs1	rs2	rd	00000000000
5	AND	And	000101	rs1	rs2	rd	00000000000
6	OR	Or	000110	rs1	rs2	rd	00000000000
7	NOR	Nor	000111	rs1	rs2	rd	00000000000
8	XOR	Xor	001000	rs1	rs2	rd	00000000000
9	SLA	Shift left arithmetic	001001	rs1	rs2	rd	00000000000
10	SLL	Shift left logical	001010	rs1	rs2	rd	00000000000
11	SRA	Shift right arithmetic	001011	rs1	rs2	rd	00000000000
12	SRL	Shift right logical	001100	rs1	rs2	rd	00000000000
I-type Instructions			Bits				
			31:26	25:21	20:16	15:00	
32	ADDI	Add Immediate	100000	rs1	rd	immediate	
33	SUBI	Sub Immediate	100001	rs1	rd	immediate	
36	LD	Load	100100	rs1	rd	offset	
37	ST	Store	100101	rs1	rd(rs)	offset	
40	BEZ	Branch Equal Zero	101000	rs1	00000	offset	
41	BNE	Branch Not Equal	101001	rs1	rd(rs)	offset	
42	JMP	Jump	101010	00000	00000	offset	

جدول ۱- لیست دستورهای پردازنده

نکته: دستورات ADDI/SUBI/LD/ST/BEZ/BNE/JMP که دارای مقدار immediate یا offset هستند، مقدار آن بین -2^{16} تا $+2^{16}-1$ هستند.

$(-2^{16} <= \text{Immediate (Offset)} <= +2^{16}-1)$

مشخصات پردازنده:

۱- پهنای خط داده: ۳۲ بیت

۲- تعداد مراحل خط لوله: ۵ مرحله‌ای

۳- تعداد دستورات: ۱۸ دستور، به علاوه دستور صفر که NOP است.

۴- میزان تاخیر انشعاب: ۲ مرحله

۵- ۳۲ ثبات همه منظوره (ثبات صفر خاص است، براساس معماری MIPS همواره مقدار آن صفر خواهد بود)

۶- آدرس‌دهی برحسب بایت و فضای آدرس دستورات (Instructions) و داده (Data) تفکیک شده می‌باشد.
(آدرس ۰ تا ۱۰۲۳ به Instruction Memory اختصاص دارد و آدرس ۱۰۲۴ به بعد به Data Memory تعلق دارد.)

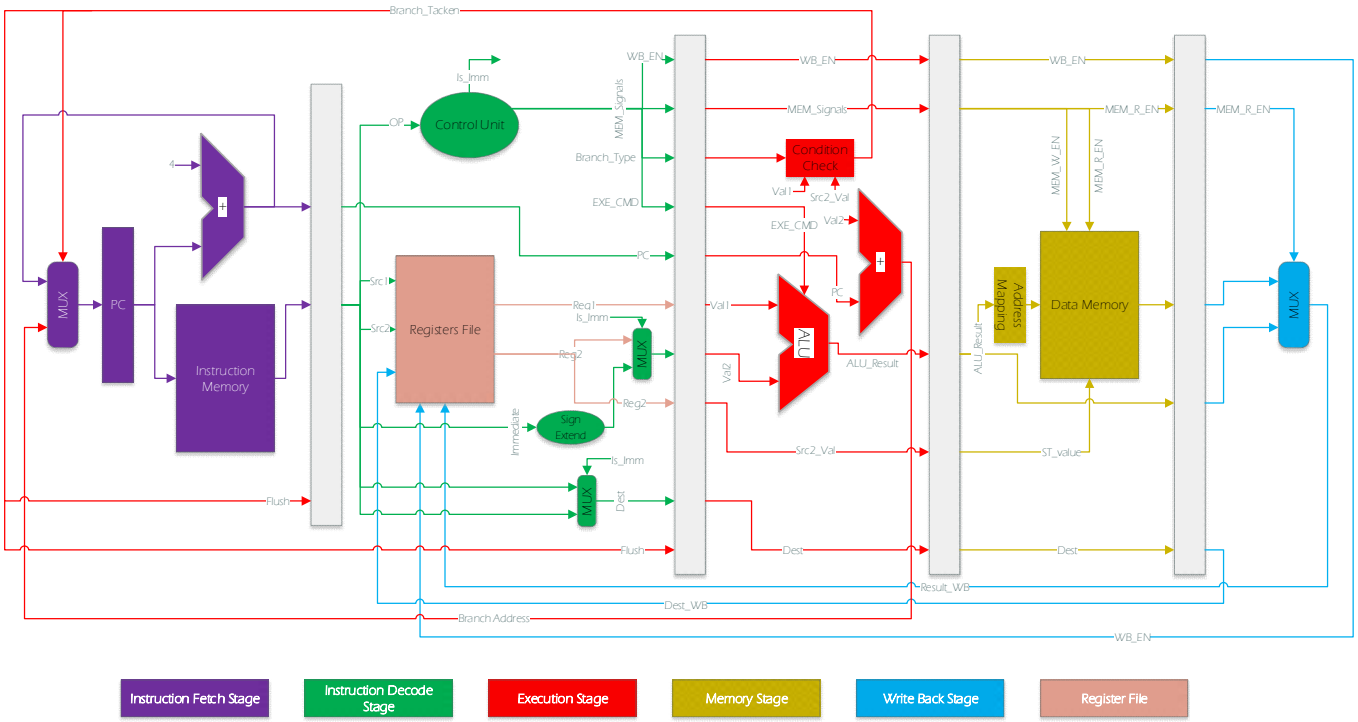
۷- تمامی پرش‌ها از نوع محلی تعریف شده است و پس از پرش مقدار رجیستر شمارنده دستور به شکل زیر خواهد بود.

$$PC=PC+ (Offset<<2) +4$$

۸- قابلیت تشخیص و جلوگیری هازاد داده‌ای (Hazard Detection Unit) و واحد ارسال به جلو (Forwarding Unit) ندارد.

معماری پردازنده

در شکل ۱ معماری کلی پردازنده در سطح RTL ترسیم شده است.



شکل ۱– معماری کلی پردازنده MIPS ساده شده

- **ایجاد تمامی Pipe-line به صورت کامل**

برای ایجاد خط لوله به ازای تمامی مراحل تمامی مراحل پردازنده و رجیسترهای پشت هر مرحله یک ماژول با ورودی خروجی‌های زیر ایجاد کنید. مقادیر PC را به تمامی مراحل پایپ ارسال نمایید این کار در اشکال زدایی کد بسیار مفید خواهد بود.

مرحله واکشی و رجیستر پس از آن

```
module IF_Stage
    (
        input clk,
        input rst,
        input Br_taken,
        input [31:0] Br_Addr,
        output [31:0] PC,
        output [31:0] Instruction
    );
```

```
module IF_Stage_reg
    (
    input clk,
    input rst,
    input flush,
    input [31:0] PC_in,
    input [31:0] Instruction_in,
    output reg [31:0] PC,
    output reg [31:0] Instruction
    );
```

مرحله دیکد و رجیستر پس از آن

```
module ID_Stage
    (
        input clk,
        input rst,
        //From IF
        input[31:0] Instruction,
        //From WB Stage
        input WB_Write_Enable,
        input[4:0] WB_Dest,
        input[31:0] WB_Data,
        //to IF stage registers
        output IF_flush,
        //to stage registers
        output[4:0] Dest,
        output[31:0] Reg2,
        output[31:0] Val2,
        output[31:0] Val1,
        output Br_taken,
        output[3:0] EXE_CMD,
        //MEM_Signals
        output MEM_R_EN,
        output MEM_W_EN,
        //Write Back Enable
        output WB_EN,
    );
```

```
module ID_Stage_reg
    (
        input clk,
        input rst,
        //from EXE Stage
        input Flush,
        //to stage registers
        input[4:0] Dest_in,
        input[31:0] Reg2_in,
        input[31:0] Val2_in,
        input[31:0] Val1_in,
        input[31:0] PC_in,
        input Br_taken_in,
        input[3:0] EXE_CMD_in,
        input MEM_R_EN_in,
        input MEM_W_EN_in,
        input WB_EN_in,
        //to stage registers
        output   reg[4:0] Dest,
        output   reg[31:0] Reg2,
        output   reg[31:0] Val2,
        output   reg[31:0] Val1,
        output   reg[31:0] PC_out,
        output   reg Br_taken,
        output   reg [3:0] EXE_CMD,
        output   reg MEM_R_EN,
        output   reg MEM_W_EN,
        output   reg WB_EN
    );
```

مرحله اجرا و رجیستر پس از آن

```
module EXE_stage
    (
        input clk,
        input [3:0] EXE_CMD,
        input [31:0]val1,
        input [31:0]val2,
        input [31:0]val_src2,
        input [31:0] PC,
        input [1:0] Br_type,

        output [31:0] ALU_result,
        output [31:0] Br_Addr,
        output Br_tacken
    );
```

```
module EXE_stage_reg
    (
        input clk,
        input rst,
        input WB_en_in,
        //MEM_Signals
        input MEM_R_EN_in,
        input MEM_W_EN_in,

        input [31:0] PC_in,
        input [31:0] ALU_result_in,
        input [31:0]ST_val_in,
        input [31:0] Dest_in,

        output   reg WB_en,
        //MEM_Signals
        output   reg MEM_R_EN,
        output   reg MEM_W_EN,
        output   reg[31:0]PC,
        output   reg[31:0]ALU_result,
        output   reg[31:0] ST_val,
        output   reg[4:0] Dest
    );
```

مرحله حافظه و رجیستر پس از آن

```
module MEM_stage
    (
        input clk,
        //MEM_Signals
        input MEM_R_EN_in,
        input MEM_W_EN_in,

        input [31:0] ALU_result_in,
        input [31:0]ST_val,

        //MEM_Signals
        output[31:0] Mem_read_value
    );
```

```
module MEM_stage_reg
    (
        input clk,
        input rst,
        input WB_en_in,
        //MEM_Signals
        input MEM_R_EN_in,
        //memory Address
        input [31:0]ALU_result_in,

        input [31:0] Mem_read_value_in,
        input [4:0] Dest_in,

        output reg WB_en,
        //MEM_Signals
        output reg MEM_R_EN,
        //memory Address
        output reg [31:0] ALU_result,

        output reg [31:0]Mem_read_value,
        output reg [4:0]Dest
    );
```

مرحله بازنویسی

```
module WB_stage
    (
        input clk,
        input WB_en_in,
        //MEM_Signals
        input MEM_R_EN,
        //memory Address
        input [31:0] ALU_result,

        input [31:0] Mem_read_value,
        input [4:0] Dest_in,

        output WB_en,
        output [31:0] Write_value,
        output [4:0] Dest
    );
```

- **مرحله واکشی**

در مرحله واکشی دستورالعمل به یک ثبات برای نگه داری شماره برنامه (PC) نیاز است. همانطور که در شکل ۱ دیده می شود، این ثبات با توجه به نوع دستور، با PC+4 یا آدرس پرش(Branch Address) جایگزین می‌شود. همچنین از یک حافظه دستور العمل ناهمگام (Instruction Memory) برای نگه‌داری دستورالعمل‌ها استفاده می‌شود.

```
module IF_Stage
    (
        input clk,
        input rst,
        input Br_taken,
        input [15:0] Br_Addr,
        output [31:0] PC,
        output [31:0] Instruction
    );
```

```
module IF_Stage_reg
    (
    input clk,
    input rst,
    input flush,
    input [31:0] PC_in,
    input [31:0] Instruction_in,
    output reg [31:0] PC,
    output reg [31:0] Instruction
    );
```

- **مرحله کدگشایی (دیکد)**

در مرحله کدگشایی می‌بایست دستور به صورت کامل دیکد گردد، سیگنال‌های کنترلی ایجاد و مقادیر رجیستر خوانده شود. برای پیاده سازی مرحله کد گشایی انجام مراحل زیر الزامیست

۱- ایجاد مجموعه ثبات‌های عمومی

یک آرایه ۳۲ تایی با ثبات‌های ۳۲ بیتی، که دارای یک پورت نوشتن همگام با لبه پایین رونده و دو پورت خواندن ناهمگام است.
نکته: در این پردازنده ثبات شماره صفر همواره مقدار 0 را در خود نگهداری می‌کند.
لیست پورتها مجموعه ثبات‌ها در زیر نشان داده شده است.

2-
```
module Registers_file
    (
        input clk,
        input rst,
        input [4:0] src1,
        input [4:0] src2,
        input [4:0] dest,
        input [31:0] Write_Val,
        input Write_EN,
        output [31:0] reg1,
        output [31:0] reg2
    );
```

۳- تکمیل مرحله کدگشایی

در این مرحله دستور به صورت کامل کدگشایی می‌گردد به گونه‌ای که دیگر در هیچ مرحله‌ای به Op-code نیازی نخواهد بود. از قسمت‌های اصلی این بخش پیاده‌سازی Control Unit به منظور ایجاد تمامی سیگنال‌های کنترلی پردازنده است. در مرحله کد گشایی همچنین کارهایی مانند تعیین سیگنال پرش، تعیین ورودی اول و دوم ALU، خواندن از رجیستر یا ارسال داده Immediate و تعیین آدرس رجیستر مقصد می‌بایست انجام گردد. پورت‌های ورودی مرحله کدگشایی و رجیسترهای پس از آن به شکل زیر است.

```
module ID_Stage
    (
        input clk,
        input rst,
        //From IF
        input[31:0] Instruction,
        //to IF stage registers
        output IF_flush,
        //to stage registers
        output[4:0] Dest,
        output[31:0] Reg2,
        output[31:0] Val2,
        output[31:0] Val1,
        output Br_taken,
        output[3:0] EXE_CMD,
        //MEM_Signals
        output MEM_R_EN,
        output MEM_W_EN,
        //Write Back Enable
        output WB_EN,
    );
```

```
module ID_Stage_reg
    (
        input clk,
        input rst,
        //from EXE Stage
        input Flush,
        //to stage registers
        input[4:0] Dest_in,
        input[31:0] Reg2_in,
        input[31:0] Val2_in,
        input[31:0] Val1_in,
        input[31:0] PC_in,
        input Br_taken_in,
        input[3:0] EXE_CMD_in,
        input MEM_R_EN_in,
        input MEM_W_EN_in,
        input WB_EN_in,
        //to stage registers
        output   reg[4:0] Dest,
        output   reg[31:0] Reg2,
        output   reg[31:0] Val2,
        output   reg[31:0] Val1,
        output   reg[31:0] PC_out,
        output   reg Br_taken,
        output   reg [3:0] EXE_CMD,
        output   reg MEM_R_EN,
        output   reg MEM_W_EN,
        output   reg WB_EN
    );
```

- **مرحله اجرا**

واحد اجرا شامل پورت های ورودی و خروجی زیر است.

```
module EXE_stage
    (
        input clk,
        input [3:0] EXE_CMD,
        input [31:0]val1,
        input [31:0]val2,
        input [31:0]val_src2,
        input [31:0] PC,
        input [1:0] Br_type,

        output [31:0] ALU_result,
        output [31:0] Br_Addr,
        output Br_tacken
    );
```

در پردازنده‌های مختلف مرحله اجرا شامل واحدهایی همچون واحد حساب و منطق (ALU)، FMA، X87، Cryptography module و... است. در پردازنده مورد نظر در این آزمایش مرحله اجرا شامل ALU و اجرای دستور پرش خواهد بود. ALU دارای دو ورودی داده، یک خروجی داده و یک ورودی چهار بیتی است که توسط Control Unit تولید شده و تعیین کننده عملیات ALU است. این ورودی کنترلی در جدول ۲ مشخص شده است.

Op-code	Instruction	ALU Command	Operation
0	NOP	XXXX	Not matter
1	ADD	0000	result = in1 + in2
3	SUB	0010	result = in1 - in2
5	AND	0100	result = in1 And in2
6	OR	0101	result = in1 Or in2
7	NOR	0110	result = in1 Nor in2
8	XOR	0111	result = in1 Xor in2
9	SLA	1000	result = in1 << in2
10	SLL	1000	result = in1 << in2
11	SRA	1001	result = in1 >>> in2
12	SRL	1010	result = in1 >> in2
32	ADDI	0000	result = in1 + in2
33	SUBI	0010	result = in1 - in2
36	LD	0000	result = in1 + in2
37	ST	0000	result = in1 + in2
40	BEZ	XXXX	Not matter
41	BNE	XXXX	Not matter
42	JMP	XXXX	Not matter

جدول ۲- ریز دستورهای واحد حساب و منطق

لیست پورت های ماژول ALU نیز در شکل زیر نشان داده شده است.

```
1   module ALU(
2       input [31:0] in1,
3       input [31:0] in2,
4       input [3:0] cmd,
5       output [31:0] result,
6       );
```

- **مرحله حافظه**

در مرحله حافظه داده‌ها از یک حافظه RAM شبیه سازی شده با سیگنال‌های MEM_R_EN و MEM_W_EN به ترتیب خوانده و در آن نوشته می‌شود. این سیگنال‌ها در مرحله گدگشایی توسط Control unit تولید و همراه با دستور در پایپ به جلو حرکت ارسال می شود. حافظه داده از آدرس ۱۰۲۴ شروع می‌شود و آدرس دهی براساس بایت خواهد بود. در هر مرحله خواندن از حافظه ۳۲ بیت داده خوانده یا نوشته می‌شود و دسترسی به تک بایت امکانپذیر نیست.

- ❖ خواندن و نوشتن فقط از آدرس‌های مضرب ۴ (به دلیل ۳۲ بیتی بودن معماری) انجام می‌شود. به طور مثال: در ازای خواندن از آدرس‌های ۱۰۲۴، ۱۰۲۵، ۱۰۲۶ و ۱۰۲۷ نتایج یکسانی خوانده می‌شود یعنی ۴ بایت از آدرس ۱۰۲۴.
- ❖ حجم حافظه را ۲۵۶ بایت در نظر بگیرید.

- **مرحله بازنشانی**

در این مرحله با سیگنال WB_EN داده ارسالی از مرحله حافظه یا اجرا در ثبات مقصد از ثبات‌های عمومی نوشته خواهد. سیگنال WB_EN توسط واحد کنترل همراه با دستور به جلو ارسال می‌گردد. همچنین به کمک سیگنال MEM_R_EN نیز نوع دستور(حافظه‌ای یا محاسباتی) تشخیص داده می‌شود و مقدار خوانده شده از حافظه یا مقدار محسابه شده از ALU در ثبات مقصد نوشته می‌شود.

- **اجرای برنامه محک**

در این مرحله باید برنامه محک در Instruction Memory قرار گیرد و نتایج اجرا به همراه تعداد سیکل‌های اجرا ثبت شود. به علت نداشتن واحد تشخیص هازاد داده‌ای (Hazard Detection Unit) می‌بایست در قسمت‌هایی از کد که هازارد داده‌ای وجود دارد دستور NOP به تعداد کافی اضافه گردد. همچنین توجه داشته باشید که پس از اجرای دستور ۴۸ پرش به دستور ۳۴ انجام می شود پس در صورت اضافه نمودن دستورات NOP به برنامه محک آدرس پرش را به درستی جایگزاری نمایید. به طور مثال برای پرش به دستور ۳۴ آدرس 15- (15− = (1 + 48) − 34)را به عنوان آدرس در دستور پرش قرار می‌دهیم. پس از اجرای دستور ۵۰ نیز در صورت برقراری شرط پرش می بایست ادامه اجرا می‌بایست از دستور ۳۳ اجرا شود.

نکات:

- دستور SRA شیفت به راست محاسباتی می باشند و می‌بایست علامت در آن حفظ شود (بیت علامت وارد می‌شود).
- دستور SRL عملوند اول را به اندازه عملوند دوم را شیفت به راست می دهد و بیت صفر وارد می شود.
- دستور SL عملوند اول را به اندازه عملوند دوم به چپ شیفت می دهد، بیت صفر وارد می شود.
- سیگنال ریست (rst) کل ثبات ها (Register File, PC, Instruction Register, Pipeline Registers) را صفر می کند.

پیش گزارش

- معماری پردازنده، نحوه کار خط لوله و عملکرد دستورها را به طور کامل یاد بگیرد.
- قبل از حضور در کلاس باید کد Verilog ماژول‌های گفته شده، نوشته شود و شبیه سازی گردد.
- هسته اصلی کد
- **رفع هازارد داده‌ای از برنامه محک:** بخش‌های دارای هازارد داده‌ای در برنامه محک را مشخص کنید و با جابجایی دستورات یا اضافه کردن NOP هازارد داده‌ای را رفع نمایید.

گزارش کار

- در ابتدای گزارش کار باید مدار طراحی شده در سطح عملکردی توضیح داده شود، سپس معماری آن در سطح RTL را با توضیحات کامل نوشته شود.
- در قسمت بعد کد Verilog معادل با RTL طراحی شده توضیح داده شود و نتایج شبیه سازی برای نشان دادن درستی کد آورده شود (به ازای هر دستور یک نتیجه به همراه تصویری از SignalTapII ارائه شود).
- پس از آن نتایج سنتز آورده شود و مدار RTL استخراج شده از Quartus II با مدار RTL طراحی شده در قسمت اول مقایسه شود و تفاوت ها را توضیح دهید.
- نتایج برنامه ریزی روی برد را توضیح دهید.
- تصویر گزارش کامپایل (Compilation Report)
- جدولی حاوی موارد زیر را گزارش نمایید:
  - تعداد کل المان‌های منطقی استفاده شده در پروژه (Total Logic Elements)
  - تعداد المان‌های منطقی استفاده شده در مدارات ترتیبی (Total Combinational functions)

- تعداد المان‌های منطقی استفاده شده توسط رجیسترها (Dedicated Logic registers)
- زمان اجرای برنامه: زمان اجرای برنامه برابر با تعداد کلاک‌هایی است که PC برای اولین بار به دستور "JMP -1" می‌رسد.
- میزان CPI (تعداد کلاک‌های اجرای برنامه بر دستور العمل).

➢ در قسمت آخر گزارش کار باید مشکلاتی که هنگام کدنویسی داشته‌اید، همچنین خطاهای زمان کامپایل و سنتز نوشته شود و راهکارهایی که این مشکلات و خطاها را برطرف نموده‌اید را بیان کنید.

موفق باشید

نصیحت‌کن

پیوست : برنامه محک

کد ماشین به همراه اسمبلی:

```
1.  32'b100000_00000_00001_00000_11000001010;//-- Addi r1 ,r0 ,1546 //r1=1546
2.  32'b000001_00000_00001_00010_00000000000;//-- Add r2 ,r0 ,r1//r2=1546
3.  32'b000011_00000_00001_00011_00000000000;//-- sub r3 ,r0 ,r1//r3=-1546
4.  32'b000101_00010_00011_0010000000000000; //--and r4,r2,r3 //r4=2
5.  32'b100001_00011_00101_0001101000110100; //--subi r5,r3,//r5=-8254
6.  32'b000110_00011_00100_0010100000000000; //--or r5,r3,r4 //r5=-1546
7.  32'b000111_00101_00000_0011000000000000; //--nor r6,r5,r0//r6=1545
8.  32'b000111_00100_00000_0101100000000000; //--nor r11,r4,r0//r11=-3
9.  32'b000011_00101_00101_0010100000000000; //--sub r5,r5,r5//r5=0
10. 32'b100000_00000_00001_0000010000000000; //--addi r1,r0,1024 //r1=1024
11. 32'b100101_00001_00010_0000000000000000;//-- st r2 ,r1 ,0 //
12. 32'b100100_00001_00101_00000_00000000000;//-- ld r5 ,r1 ,0 //r5=1546
13. 32'b101000_00101_00000_00000_00000000001;//-- Bez r5 ,1//not taken
14. 32'b001000_00101_00001_00111_00000000000;//-- xor r7 ,r5 ,r1 //r7=522
15. 32'b001000_00101_00001_00000_00000000000;//-- xor r0 ,r5 ,r1 //r0=0
16. 32'b001001_00011_00100_00111_00000000000;//-- sla r7 ,r3 ,r4//r7=-6184
17. 32'b100101_00001_00111_00000_00000010100;//-- st r7 ,r1 ,20
18. 32'b001010_00011_00100_01000_00000000000;//-- sll r8 ,r3 ,r4  //r8=-6184
19. 32'b001011_00011_00100_01001_00000000000;//-- sra r9 ,r3 ,r4 //r9=1073741437
20. 32'b001100_00011_00100_01010_00000000000;//-- srl r10 ,r3 ,r4//r10=-384
21. 32'b100101_00001_00011_00000_00000000100;//-- st r3 ,r1 ,4
22. 32'b100101_00001_00100_00000_00000001000;//-- st r4 ,r1 ,8
23. 32'b100101_00001_00101_00000_00000001100;//-- st r5 ,r1 ,12
24. 32'b100101_00001_00110_00000_00000010000;//-- st r6 ,r1 ,16
25. 32'b100100_00001_01011_00000_00000000100;//-- ld r11 ,r1 ,4//r11=-1456
26. 32'b100101_00001_01011_00000_00000011000;//-- st r11 ,r1 ,24
27. 32'b100101_00001_01001_00000_00000011100;//-- st r9 ,r1 ,28
28. 32'b100101_00001_01010_00000_00000100000;//-- st r10 ,r1 ,32
29. 32'b100101_00001_01000_00000_00000100100;//-- st r8 ,r1 ,36
30. 32'b100000_00000_00001_00000_00000000011;//-- Addi r1 ,r0 ,3 //r1=3
31. 32'b100000_00000_00100_00000_10000000000;//-- Addi r4 ,r0 ,1024   //r4=1024
32. 32'b100000_00000_00010_00000_00000000000;//-- Addi r2 ,r0 ,0 //r2=0
33. 32'b100000_00000_00011_00000_00000000001;//-- Addi r3 ,r0 ,1 //r3=1
34. 32'b100000_00000_01001_00000_00000000010;//-- Addi r9 ,r0 ,2 //r9=2
35. 32'b001010_00011_01001_01000_00000000000;//-- sll r8 ,r3 ,r9 //r8=r3*4
36. 32'b000001_00100_01000_01000_00000000000;//-- Add r8 ,r4 ,r8 //r8=1024+r3*4
37. 32'b100100_01000_00101_00000_00000000000;//-- ld r5 ,r8 ,0 //
38. 32'b100100_01000_00110_11111_11111111100;//-- ld r6 ,r8 ,-4 //
39. 32'b000011_00101_00110_01001_00000000000;//-- sub r9 ,r5 ,r6
```

```
40.  32'b100000_00000_01010_10000_00000000000;//-- Addi r10 ,r0 ,0x8000
41.  32'b100000_00000_01011_00000_00000010000;//-- Addi r11 ,r0 ,16 //2
42.  32'b001010_01010_01011_01010_00000000000;//-- sll r10 ,r1 ,r11 //2
43.  32'b000101_01001_01010_01001_00000000000;//-- And r9 ,r9 ,r10 // if(r5>r6) r9=0 else r9=-2147483648
44.  32'b101000_01001_00000_00000_00000000010;//-- Bez r9 ,2
45.  32'b100101_01000_00101_11111_11111111100;//-- st r5 ,r8 ,-4
46.  32'b100101_01000_00110_00000_00000000000;//-- st r6 ,r8 ,0
47.  32'b100000_00011_00011_00000_00000000001;//-- Addi r3 ,r3 ,1 //2
48.  32'b101001_00001_00011_11111_11111110001;//-- BNE r1 ,r3 ,-15
49.  32'b100000_00010_00010_00000_00000000001;//-- Addi r2 ,r2 ,1 //2
50.  32'b101001_00001_00010_11111_11111101110;//-- BNE r1 ,r2 ,-18
51.  32'b100000_00000_00001_00000_10000000000;//-- Addi r1 ,r0 ,1024 //r1=1024
52.  32'b100100_00001_00010_00000_00000000000;//-- ld ,r2 ,r1 ,0 //r2=-1546
53.  32'b100100_00001_00011_00000_00000000100;//-- ld ,r3 ,r1 ,4 //r3=2
54.  32'b100100_00001_00100_00000_00000001000;//-- ld ,r4 ,r1 ,8 //r4=1546
55.  32'b100100_00001_00100_00000_01000001000;//-- ld ,r4 ,r1 ,520 // after SRAM r4=random number
56.  32'b100100_00001_00100_00000_10000001000;//-- ld ,r4 ,r1 ,1023 // after SRAM r4=random number
57.  32'b100100_00001_00101_00000_00000001100;//-- ld ,r5 ,r1 ,12 // r5=1546
58.  32'b100100_00001_00110_00000_00000010000;//-- ld ,r6 ,r1 ,16 //r6=1545
59.  32'b100100_00001_00111_00000_00000010100;//-- ld ,r7 ,r1 ,20 //r7=-6184
60.  32'b100100_00001_01000_00000_00000011000;//-- ld ,r8 ,r1 ,24 //r8=-1546
61.  32'b100100_00001_01001_00000_00000011100;//-- ld ,r9 ,r1 ,28 //r9=1073741437
62.  32'b100100_00001_01010_00000_00000100000;//-- ld ,r10,r1 ,32 //r10=-387
63.  32'b100100_00001_01011_00000_00000100100;//-- ld ,r11,r1 ,36 //r11=-6184
64.  32'b101010_00000_00000_11111_11111111111;//-- JMP -1*/
```