

بسم الله الرحمن الرحيم



آزمایشگاه معماری کامپیوتر
گزارش کار آزمایش هفتم: کنترل توسط برنامه ذخیره شده در حافظه

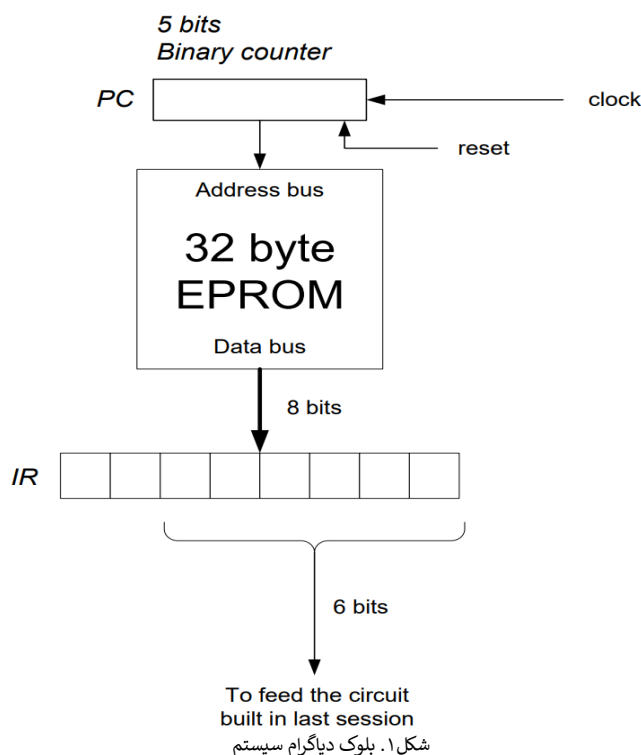
دانشکده مهندسی کامپیوتر
دانشگاه صنعتی شریف

تابستان 1403

رادین چراغی ۴۰۱۱۰۵۸۱۵
مبین پورعابدینی ۴۰۱۱۱۰۵۵۶
آرین نوری ۴۰۱۱۰۶۶۶۳

مقدمه

در این آزمایش می‌خواهیم تا به مداری که در آزمایش قبل طراحی کردیم یک ویژگی اضافه کنیم و آن اضافه کردن حافظه برای دستورات است، به این صورت که برای هر دستوری یک کد تعیین کرده و داخل حافظه‌ی EPROM قرار می‌دهیم و با آغاز برنامه دستورات خط به خط تا انتها پیش می‌روند. این فرمان‌ها توسط یک Program counter آدرس‌دهی می‌شوند و پس از واکنشی از حافظه اجرا خواهند شد. در شکل زیر بلوک دیاگرام سیستم را مشاهده می‌کنید. که در آن دستورها 8 بیتی هستند و ما با دو بیت سمت چپ کاری نداریم.



در ادامه پس از طراحی بخش حافظه و شمارنده‌ی برنامه، قطعه‌کدی را برای اجرای برنامه‌ی فیبوناچی درون حافظه قرار داده و آن را اجرا می‌کنیم. پس به ترتیب ابتدا آزمایش را **شرح** می‌کنیم و **طراحی** را آغاز می‌کنیم. سپس طراحی را مورد **آزمایش** قرار داده و در نهایت قطعه‌ی کد مربوط به **فیبوناچی** را در آن اجرا می‌کنیم.

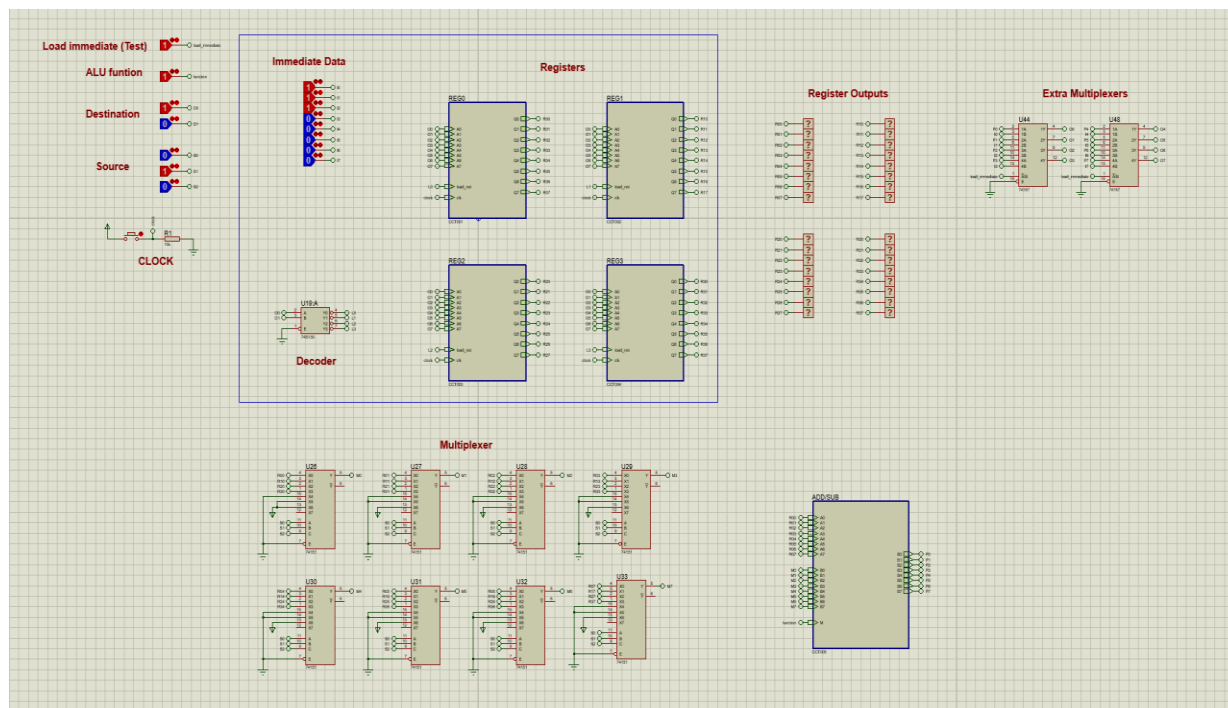
شرح و طراحی آزمایش در پروتئوس

حالا طراحی را شروع می‌کنیم، ابتدا باید نحوه کار مدار آزمایش قبل را مرور کنیم، در آزمایش قبل ما چهار رجیستر هشت بیتی داشتیم، به همراه یک واحد جمع/تفریق‌کننده و در نتیجه یک مالتی پلکسر برای انتخاب مقصد خروجی این واحد. همچنین یکی از ورودی‌های واحد محاسباتی همواره رجیستر صفر مدار بود و ورودی دیگر آن نیز هشت حالت داشت که به ترتیب به این صورت بودند: رجیستر صفر، رجیستر یک، رجیستر دو، رجیستر سه، 0، 1 و منفی 1. (ورودی هشتم نیز رزرو برای آزمایش‌های بعدی)

به طور خلاصه مدار از چهار رجیستر، هشت واحد مالتی پلکسر هشت به یک برای انتخاب ورودی واحد محاسباتی، یک واحد محاسباتی با قابلیت جمع و تفریق، ورودی و خروجی و ساعت، و در نهایت یک سری قطعات کمکی اضافه برای Load Immediate تشکیل شده بود که این قطعات تنها برای تست راحت‌تر مدار به آن اضافه شده بودند و در این آزمایش برای ما اهمیتی ندارند. همچنین می‌دانیم که فرمت دستورها به این شکل است:

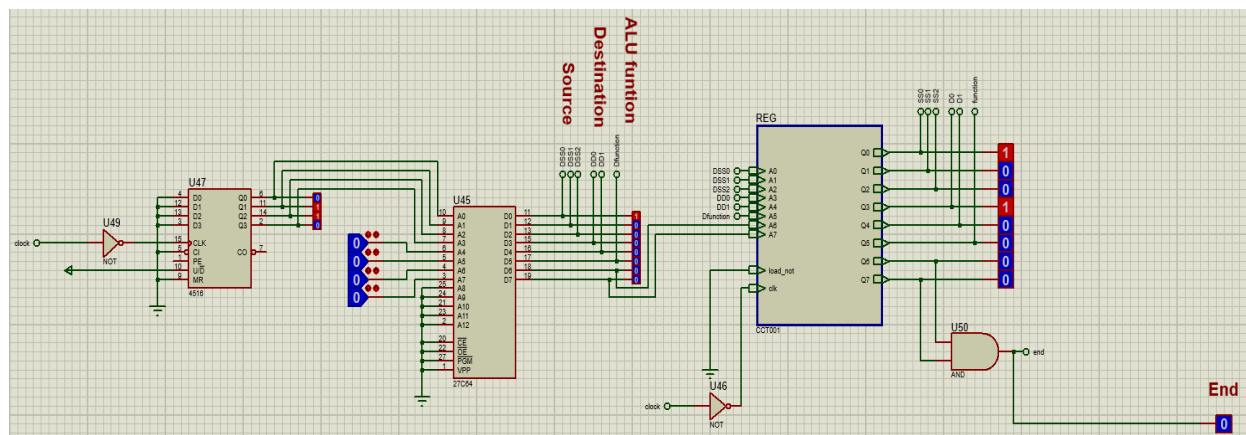
00 M DD SSS, M = Adder/Subtractor Selector, DD = Destination Selector, SSS = ALU Source Selector

در نهایت ساخت مدار را از روی مدار قبلی شروع می‌کنیم،



شکل ۲. مدار نهایی آزمایش قبلی

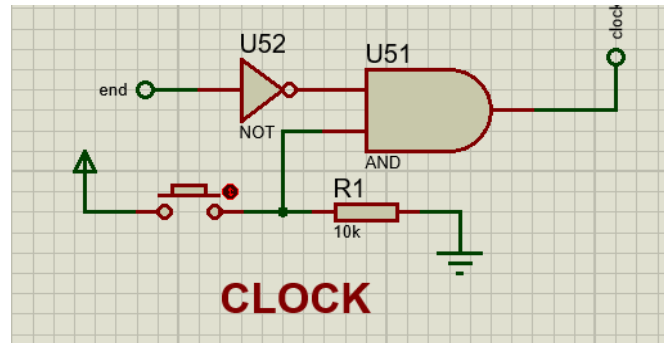
باید یک شمارنده برای شمارش دستورات قرار دهیم و در ادامه حاصل شمارش به قطعه‌ی EPROM آماده بدهیم، برای راحتی مدار و وضوح بیشتر یک رجیستر هشت بیتی جلوی خروجی EPROM گذاشته تا ترتیب دستورات را پشت سر یکدیگر ببینیم. همچنین برای سیگنال ساعت آن باید NOT ساعت اصلی مدار را به آن متصل کنیم. با این کار موقع لبه‌ی بالارونده، دستوری که در خروجی رجیستر دستورات قرار دارد اجرا شده و در لبه‌ی پایین‌رونده، دستور بعدی وارد رجیستر دستور می‌شود. پس نحوه‌ی پیاده‌سازی به این صورت می‌شود:



شکل ۳. چیدمان EPROM در مدار

گیت EPROM، همان قطعه‌ی 27C64 در شکل بالا است که از شمارنده ورودی می‌گیرد و به رجیستر می‌دهد. در انتهای مدار نیز می‌بینید که ترکیب AND دو بیت انتهایی به عنوان بیت end در نظر گرفته شده. این یعنی هرگاه بخواهیم که به EPROM اعلام کنیم مدار به پایان رسیده کافیست دستوری قرار دهیم که دو بیت آخر (که همان بیت‌های اضافی در دستورات بودند) در آن یک باشند.

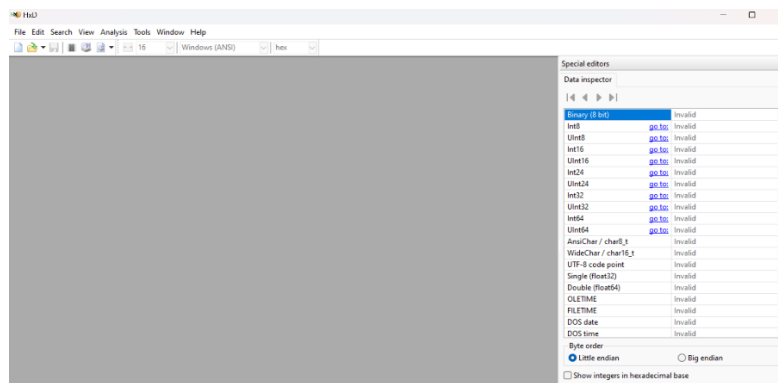
برای مثال دستور پایانی به این صورت می‌تواند باشد: 1100 0000. در ادامه نیز این بیت اعلام پایان را به ساعت متصل می‌کنیم، به طوری که دیگر ساعت زدن را بی‌اثر کند.



شکل ۴. بی‌اثر کردن ساعت پس از پایان دستورات

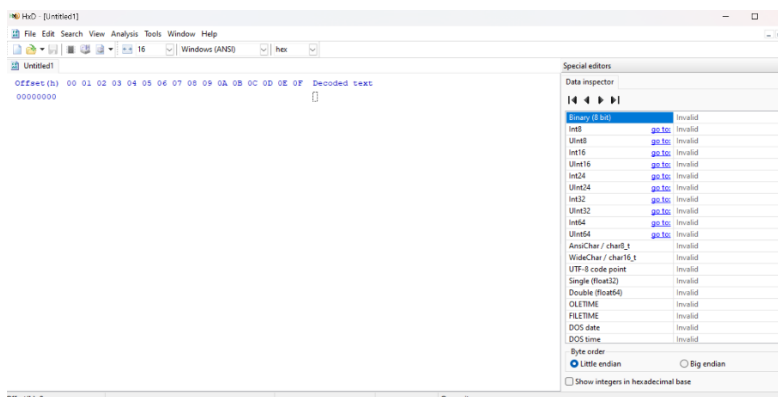
آزمایش مدار طراحی شده در پروتئوس

با به پایان رسیدن تغییرات مدار کافی است که آن را تست کنیم. برای تست مدار و ایجاد قابلیت نوشتن دستورات در EPROM، باید برنامه‌ای داشته باشیم که بتواند به ما در طراحی دستورات به طور HEX کمک کند. از برنامه‌ی HxD کمک می‌گیریم. پس از نصب و باز کردن آن با چنین صفحه‌ای روبه‌رو می‌شویم:



شکل ۵. برنامه‌ی HxD

از طریق منوی File، یک فایل جدید می‌سازیم:



شکل ۶. ساخت فایل جدید

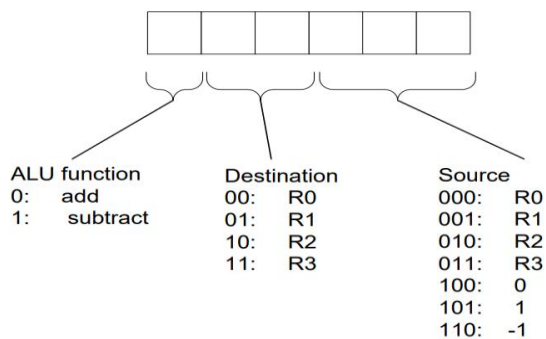
سپس می‌توانیم در این فایل دستورات را وارد کنیم. فرض کنید برای مثال هدف اجرای برنامه‌ی زیر باشد:

```
SUB R0,
ADD R0, 1
ADD R0, 1
MOV R1, R0
ADD R0, 1
ADD R0, 1
MOVADD R2, 1
SUB R0, R2
ADD R0, R1
```

از آنجایی که همواره یکی از ورودی‌های ALU، R0 است که کد بالا در واقع عملیات زیر را انجام می‌دهد:

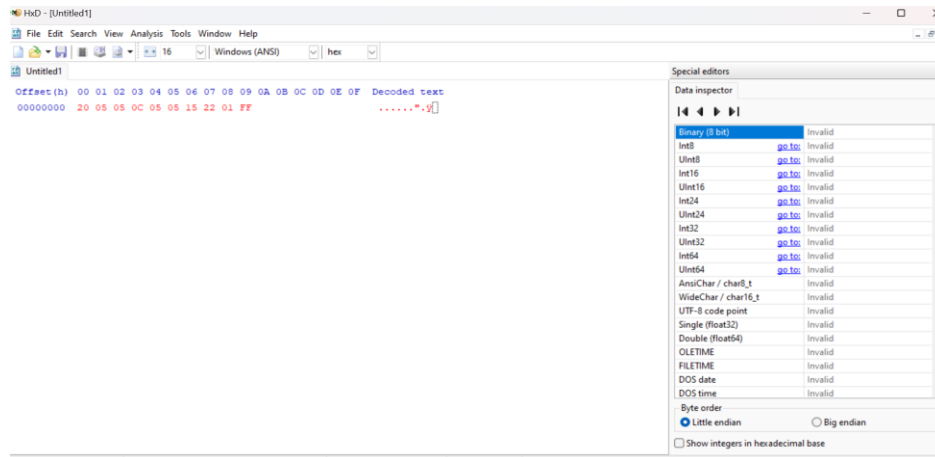
```
R0 -= R0;      //R0 = 0
R0++;          //R0 = 1
R0++;          //R0 = 2
R1 = R0;       //R1 = 2
R0++;          //R0 = 3
R0++;          //R0 = 4
R2 = R0 + 1;   //R2 = 5
R0 = R0 - R2;  //R0 = 4 - 5 = -1
R0 = R0 + R1;  //R0 = -1 + 2 = 1
```

طبق قالبی که ابتدا نیز به آن اشاره کردیم دستور بالا را کد می‌کنیم:



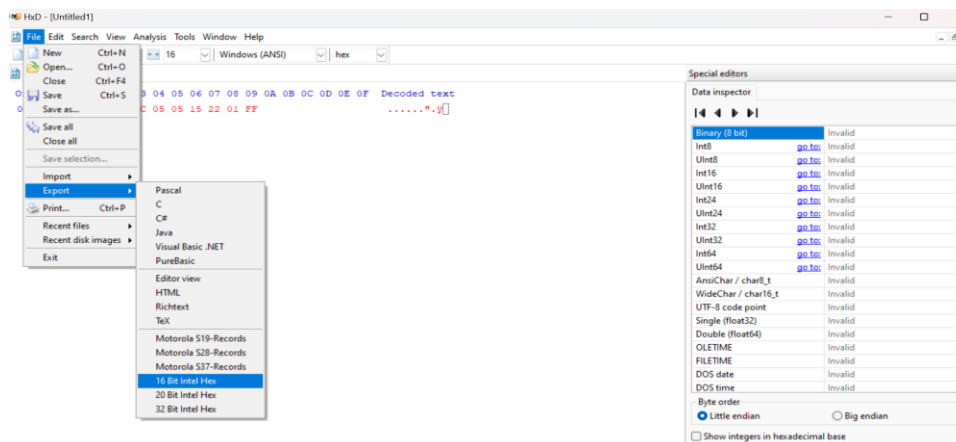
شکل ۷. قالب دستورات ورودی به EPROM

R0 -= R0;	//R0 = 0	CODE: 00 1 00 000	HEX: 20
R0++;	//R0 = 1	CODE: 00 0 00 101	HEX: 05
R0++;	//R0 = 2	CODE: 00 0 00 101	HEX: 05
R1 = R0;	//R1 = 2	CODE: 00 0 01 100	HEX: 0C
R0++;	//R0 = 3	CODE: 00 0 00 101	HEX: 05
R0++;	//R0 = 4	CODE: 00 0 00 101	HEX: 05
R2 = R0 + 1;	//R2 = 5	CODE: 00 0 10 101	HEX: 15
R0 = R0 - R2;	//R0 = 4 - 5 = -1	CODE: 00 1 00 010	HEX: 22
R0 = R0 + R1;	//R0 = -1 + 2 = 1	CODE: 00 0 00 001	HEX: 01
	//FINISH	CODE: 11 X XX XXX	HEX: FF



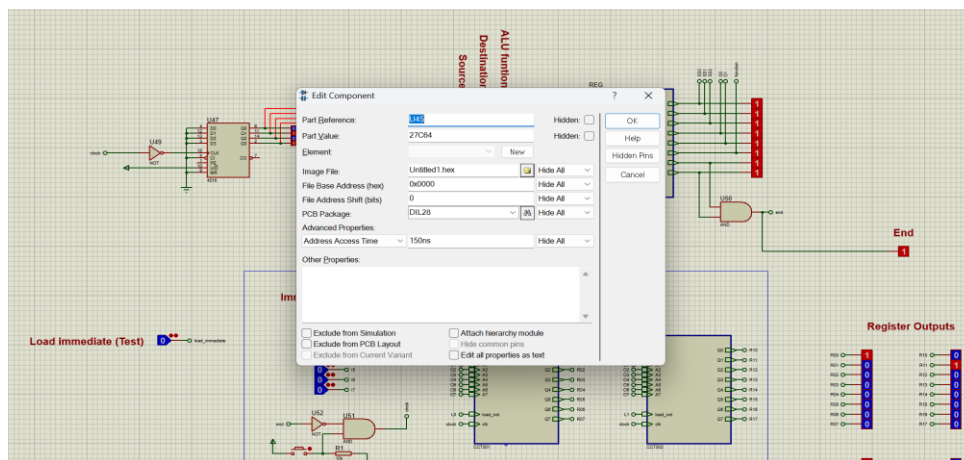
شکل ۸. نوشتن دستورات در برنامه

از طریق منوی FILE، و سپس منوی EXPORT، بر روی 16 Bit Intel Hex کلیک می‌کنیم تا فایل ساخته شود:



شکل ۹. ساختن فایل دستورات

در ادامه وارد پروتئوس شده و کد را بر روی EPROM قرار می‌دهیم، به این صورت که بر روی Edit Properties کلیک کرده و در بخش Image File، فایلی که ساخته‌ایم را قرار می‌دهیم:



شکل ۱۰. اتصال کد نوشته شده به EPROM

بعد از این که کد به درستی متصل شد کافیست آن را تست کنیم تا ببینیم همان خواسته‌ی ما انجام می‌شود یا خیر.

The screenshot shows a Proteus simulation of a 4-bit ALU and Register File. The ALU function is set to 'Destination Source'. The Register File contains four registers (REG0, REG1, REG2, REG3) with outputs displayed on the right. The ALU output is 0.

The screenshot shows a digital logic circuit simulation for a 4-bit ALU. The circuit includes an ALU function block, a Register File (REG), and an AND gate (U50). The ALU function block has inputs for Destination, Source, and ALU function. The Register File has four registers (REG0, REG1, REG2, REG3) and an AND gate (U50) for the ALU function. The circuit is controlled by a clock signal and a load immediate (Test) signal. The output of the ALU function is shown as a 4-bit bus.

ALU function

Destination

Source

REG

U50

clock

load_immediate

End

0

Immediate Data

Load immediate (Test)

Registers

Register Outputs

CLOCK

Decoder

REG0

REG1

REG2

REG3

U50A

U50

U51

U52

U53

U54

U55

U56

U57

U58

U59

U60

U61

U62

U63

U64

U65

U66

U67

U68

U69

U70

U71

U72

U73

U74

U75

U76

U77

U78

U79

U80

U81

U82

U83

U84

U85

U86

U87

U88

U89

U90

U91

U92

U93

U94

U95

U96

U97

U98

U99

U100

U101

U102

U103

U104

U105

U106

U107

U108

U109

U110

U111

U112

U113

U114

U115

U116

U117

U118

U119

U120

U121

U122

U123

U124

U125

U126

U127

U128

U129

U130

U131

U132

U133

U134

U135

U136

U137

U138

U139

U140

U141

U142

U143

U144

U145

U146

U147

U148

U149

U150

U151

U152

U153

U154

U155

U156

U157

U158

U159

U160

U161

U162

U163

U164

U165

U166

U167

U168

U169

U170

U171

U172

U173

U174

U175

U176

U177

U178

U179

U180

U181

U182

U183

U184

U185

U186

U187

U188

U189

U190

U191

U192

U193

U194

U195

U196

U197

U198

U199

U200

U201

U202

U203

U204

U205

U206

U207

U208

U209

U210

U211

U212

U213

U214

U215

U216

U217

U218

U219

U220

U221

U222

U223

U224

U225

U226

U227

U228

U229

U230

U231

U232

U233

U234

U235

U236

U237

U238

U239

U240

U241

U242

U243

U244

U245

U246

U247

U248

U249

U250

U251

U252

U253

U254

U255

U256

U257

U258

U259

U260

U261

U262

U263

U264

U265

U266

U267

U268

U269

U270

U271

U272

U273

U274

U275

U276

U277

U278

U279

U280

U281

U282

U283

U284

U285

U286

U287

U288

U289

U290

U291

U292

U293

U294

U295

U296

U297

U298

U299

U300

U301

U302

U303

U304

U305

U306

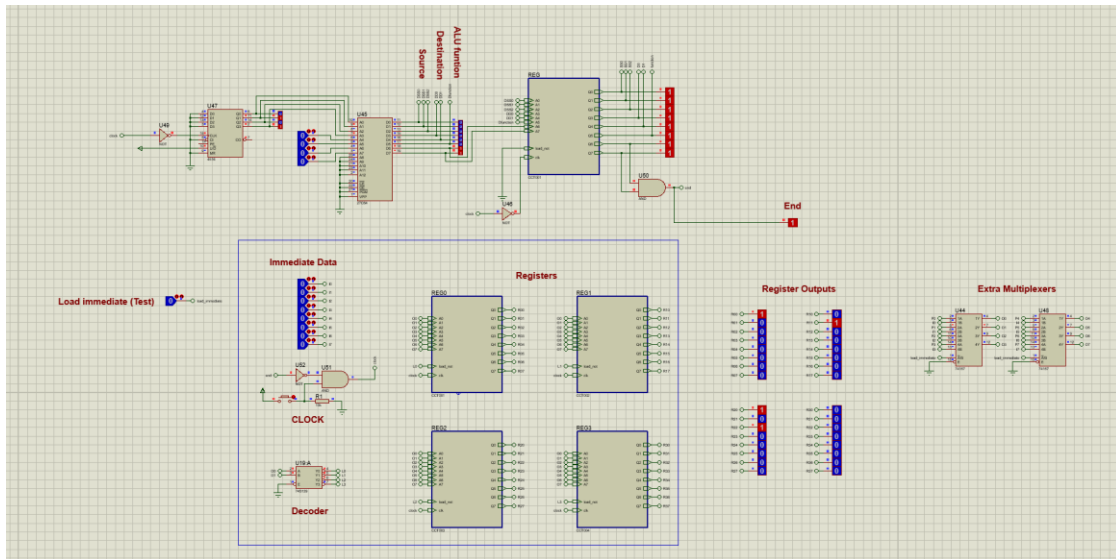
U307

U308

U309

U310

7



شکل ۱۳. در انتهای کد، مطابق انتظار مقدار یک درون رجیستر صفرم ریخته شده و بیت end روشن شده

در نتیجه قطعه کد امتحانی به درستی کار کرد و حالا به بخش پایانی آزمایش یعنی نوشتن کد فیبوناچی می‌پردازیم.

برنامه‌ی فیبوناچی

در این بخش، تابع فیبوناچی با ورودی‌های ابتدایی صفر و یک را کدنویسی خواهیم کرد:

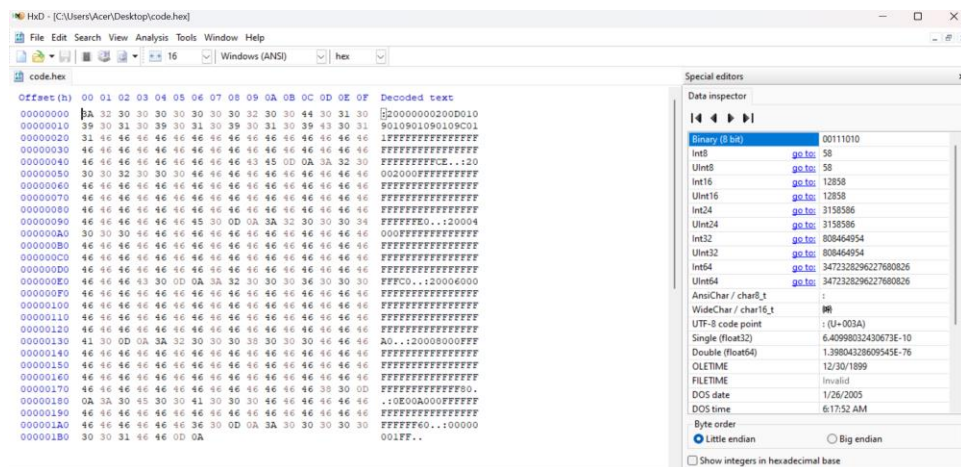
$$F(n) := \begin{cases} 0 & \text{if } n = 0; \\ 1 & \text{if } n = 1; \\ F(n-1) + F(n-2) & \text{if } n > 1. \end{cases}$$

شکل ۱۴. تابع فیبوناچی

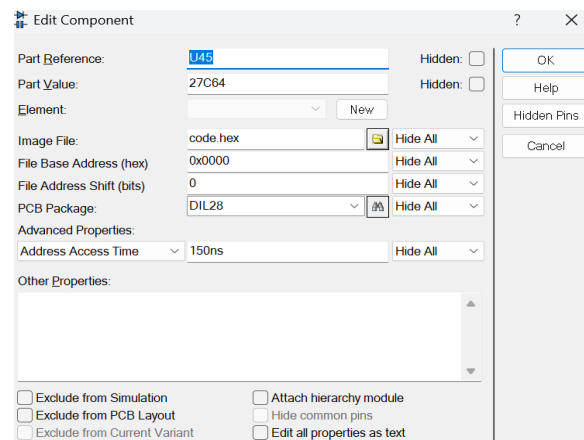
جدول کد مورد انتظار ما به این شکل خواهد بود:

ADDRESS	CODE	INSTRUCTION	COMMENT
0000	00100000: 20	SUB R0, R0	R0 = 0
0001	00001101: 0D	ADD R1, 1	R1 = 1
0010	00000001: 01	ADD R0, R1	R0 = 1
0011	00001001: 09	ADD R1, R0	R1 = 2
0100	00000001: 01	ADD R0, R1	R0 = 3
0101	00001001: 09	ADD R1, R0	R1 = 5
0110	00000001: 01	ADD R0, R1	R0 = 8
0111	00001001: 09	ADD R1, R0	R1 = 13
1000	00000001: 01	ADD R0, R1	R0 = 21
1001	00001001: 09	ADD R1, R0	R1 = 34
1010	11000000: C0	EOF	END

جدول ۱. کد برنامه‌ی فیبوناچی



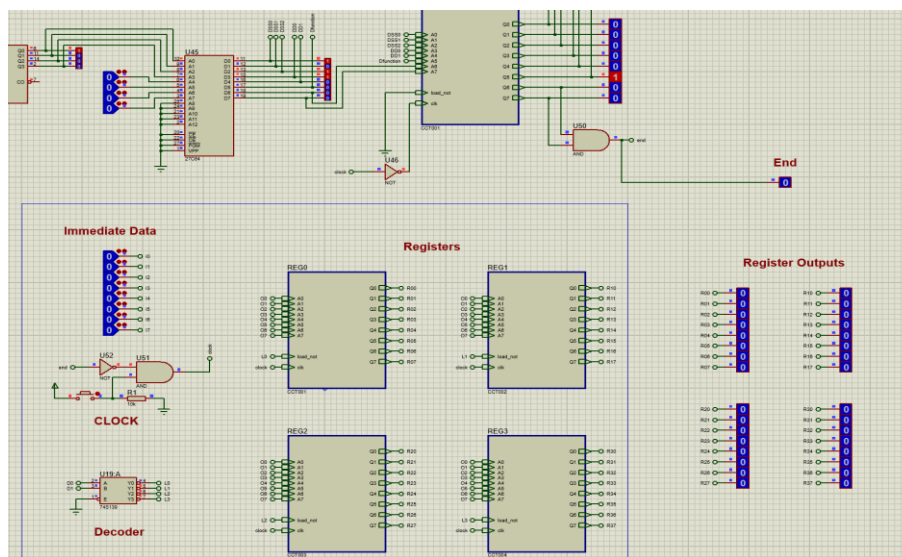
شکل ۱۵. تصویر درون برنامه پس از ذخیره سازی در کامپیوتر



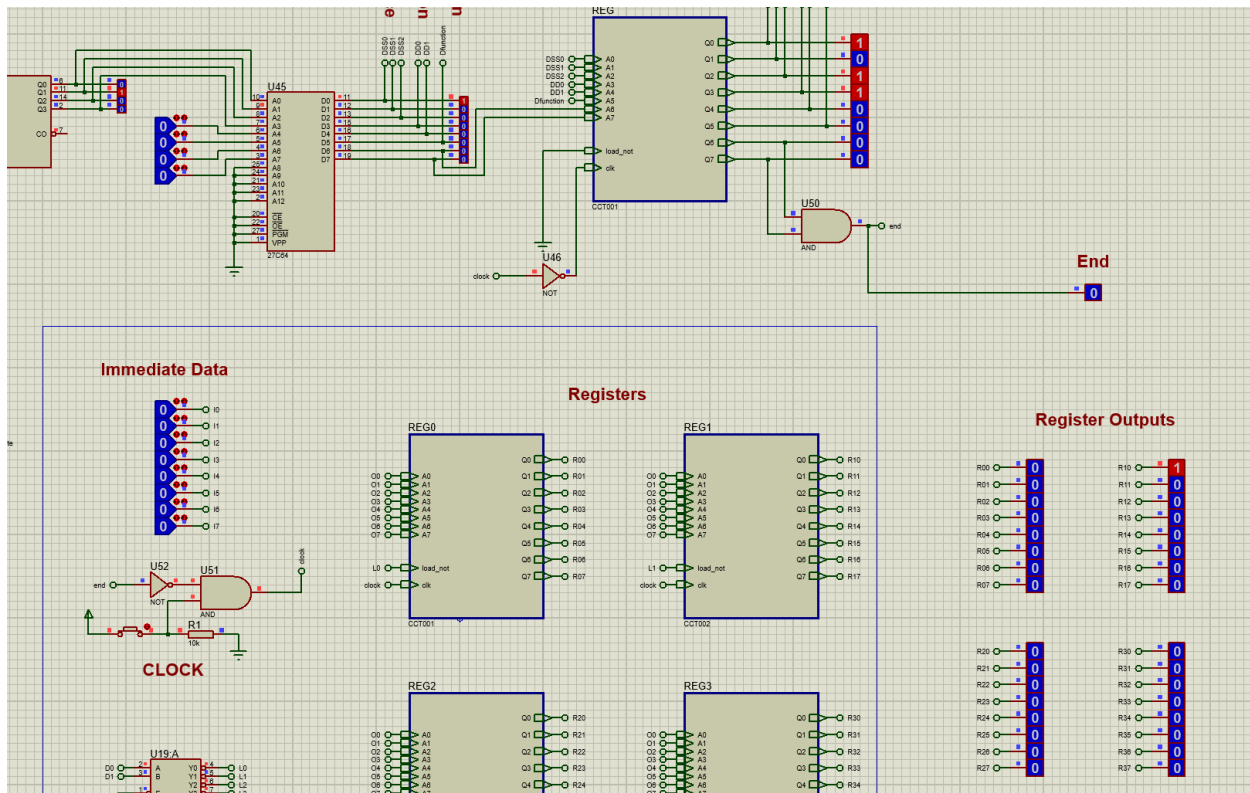
شکل ۱۶. اتصال به EPROM

در ادامه به تصاویر تست بخش پایانی از مدار می‌رسیم.

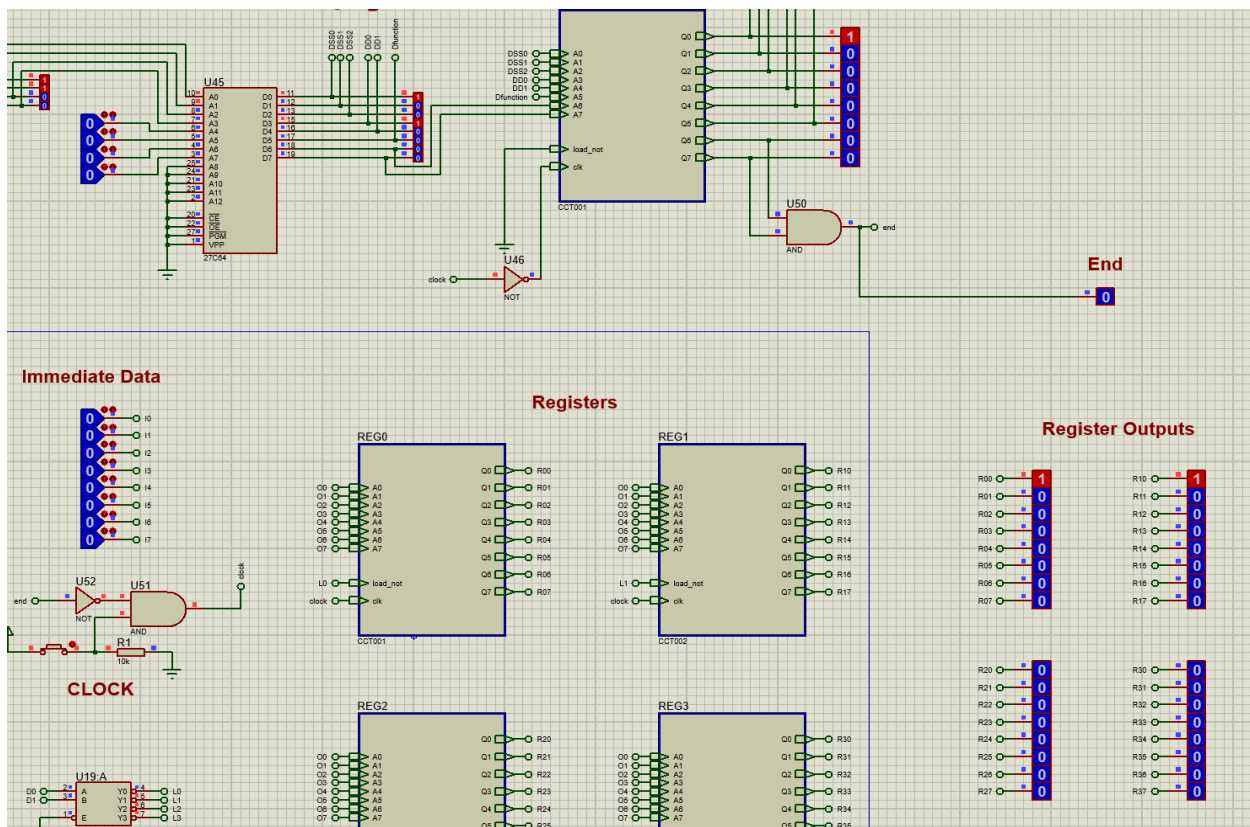
تصاویر تست برنامه‌ی فیبوناچی



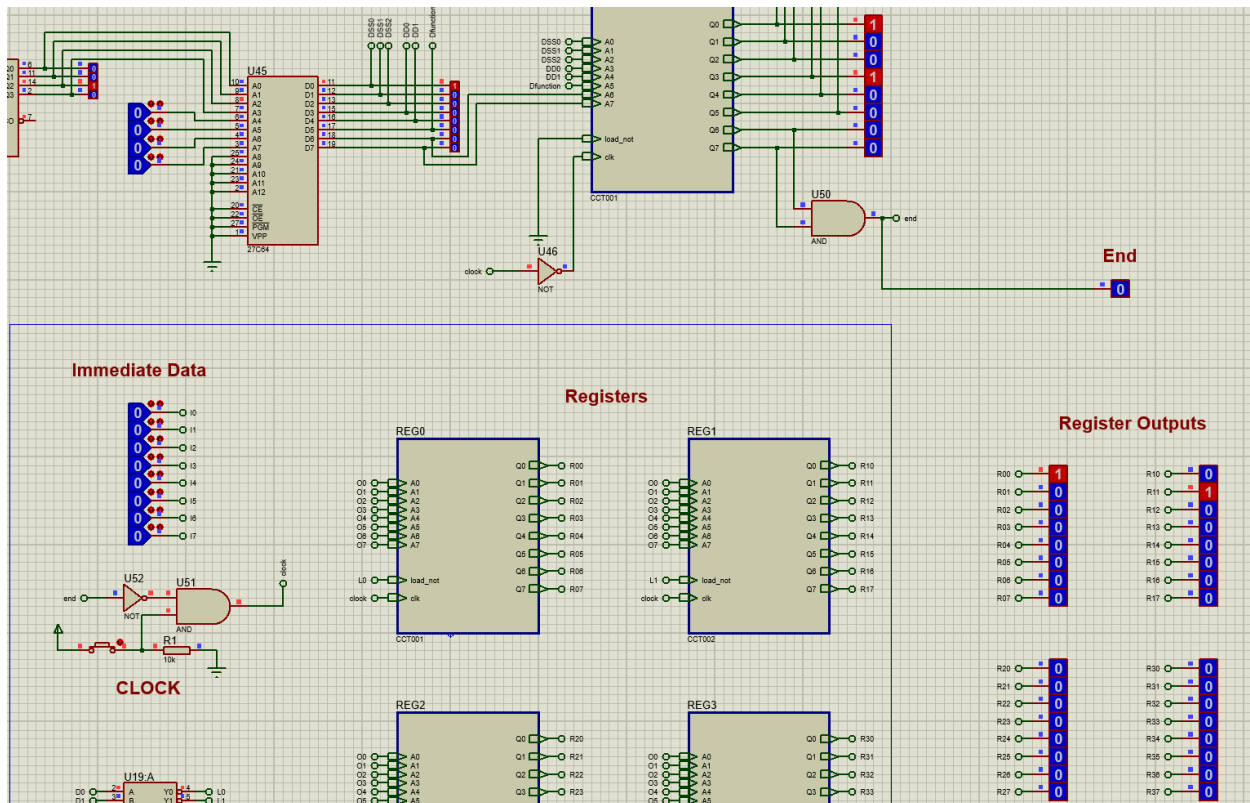
شکل ۱۷. دستور اول: دادن صفر به رجیستر صفرم



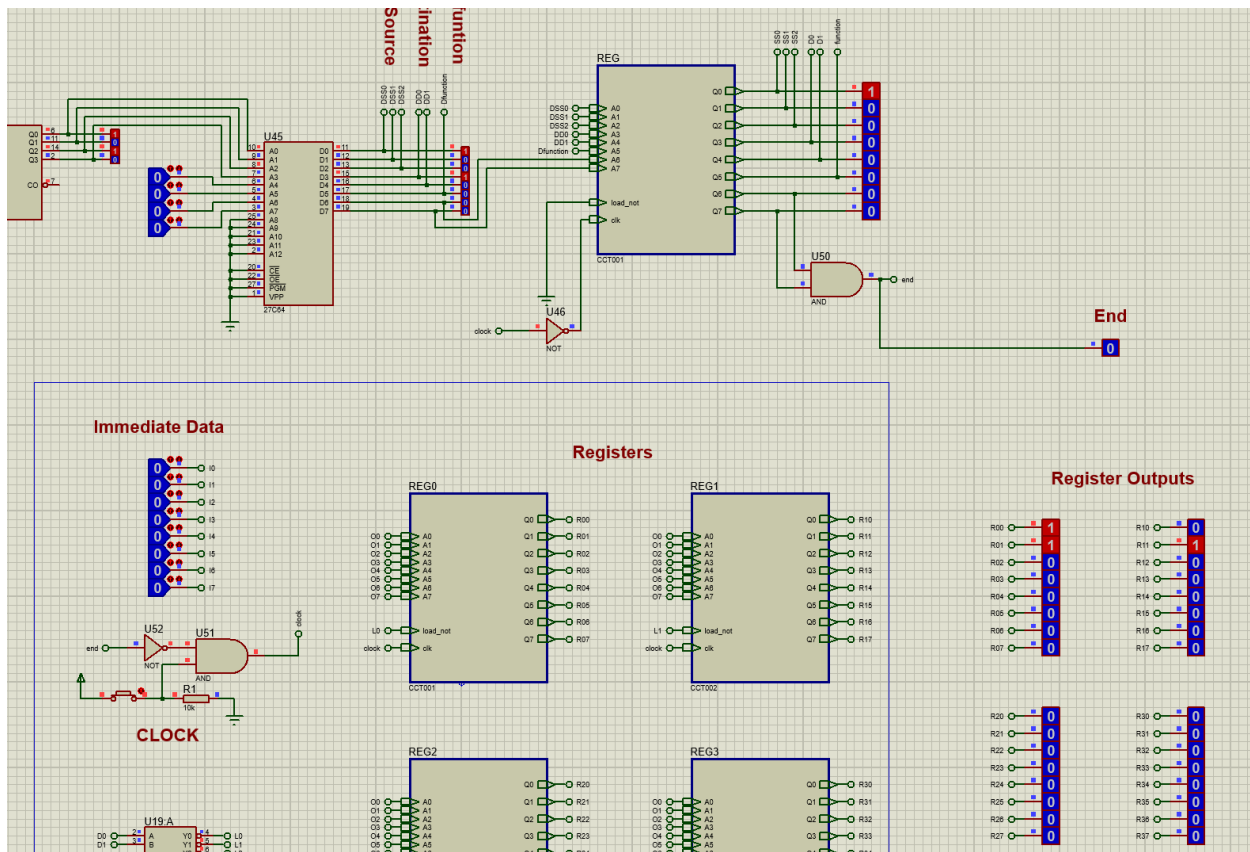
شکل ۱۸. دستور دوم: دادن یک به رجیستر یکم



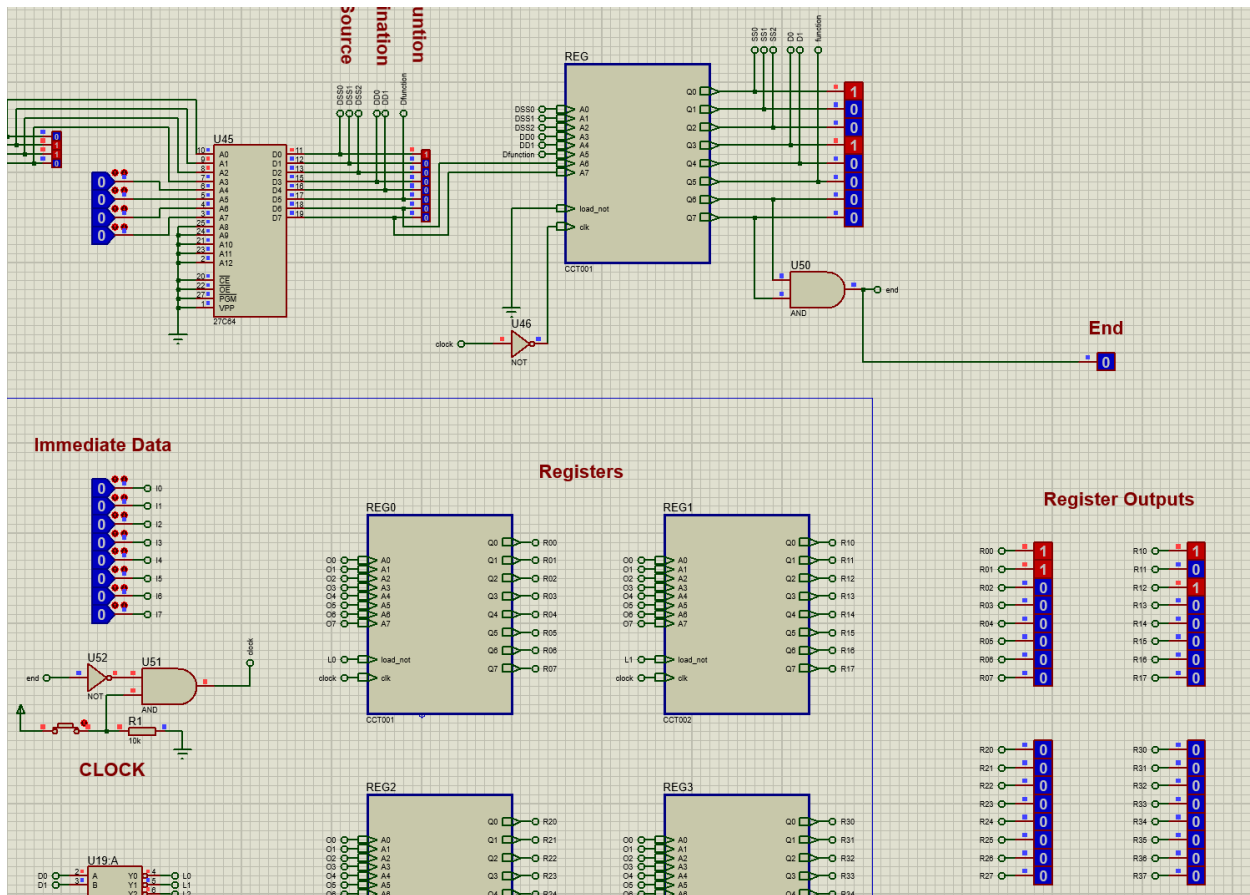
شکل ۱۹. دستور سوم: حاصل جمع دو رجیستر در رجیستر صفرم



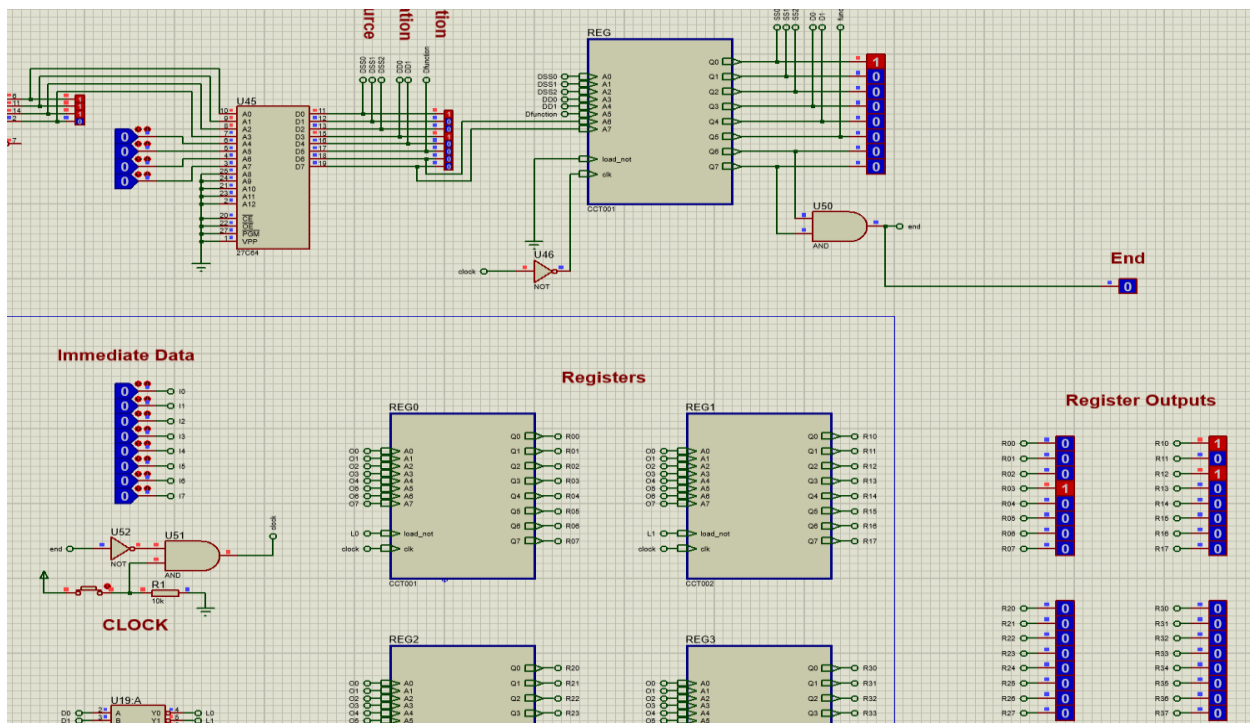
شکل ۲۰. حاصل جمع برابر ۲ است و وارد رجیستر یکم می‌شود



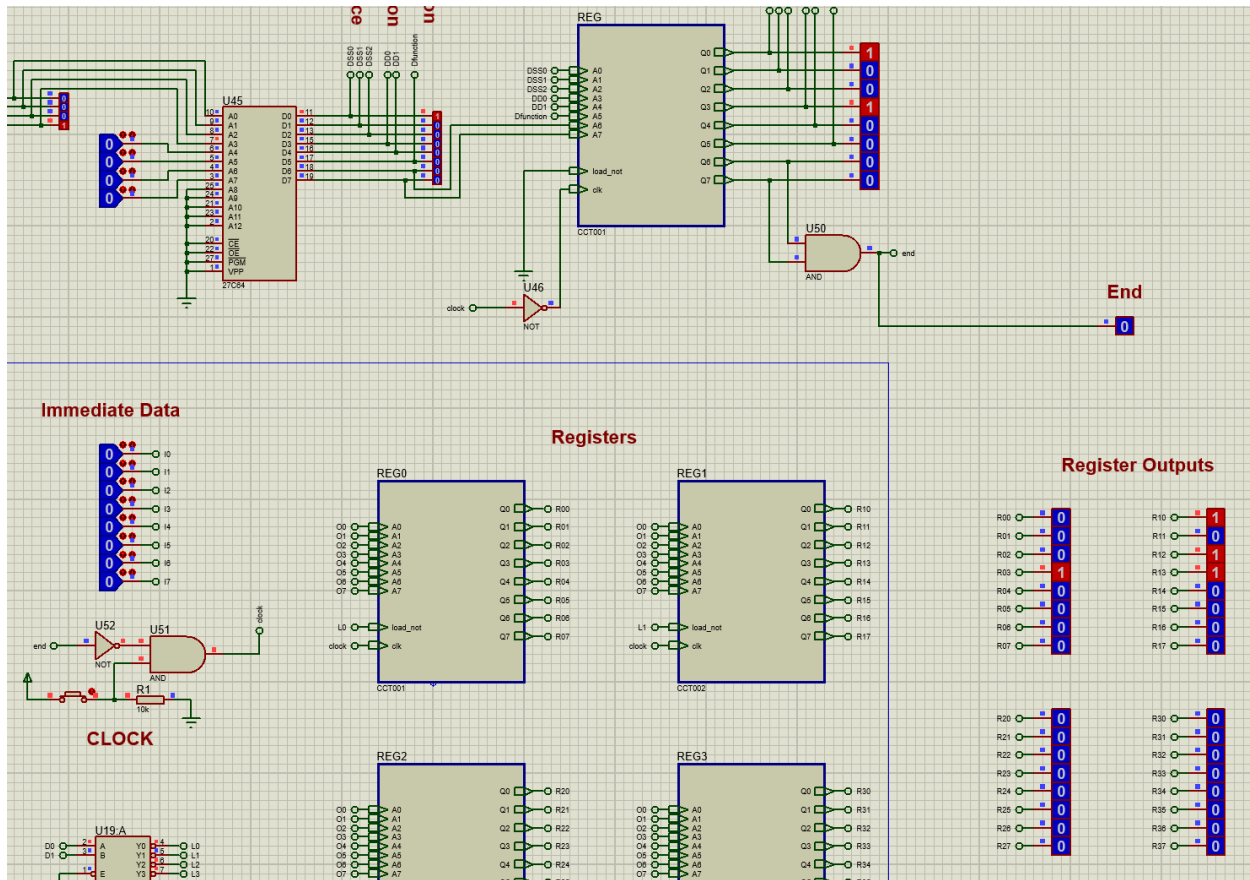
شکل ۲۱. حاصل جمع برابر ۳ است و وارد رجیستر صفرم می‌شود



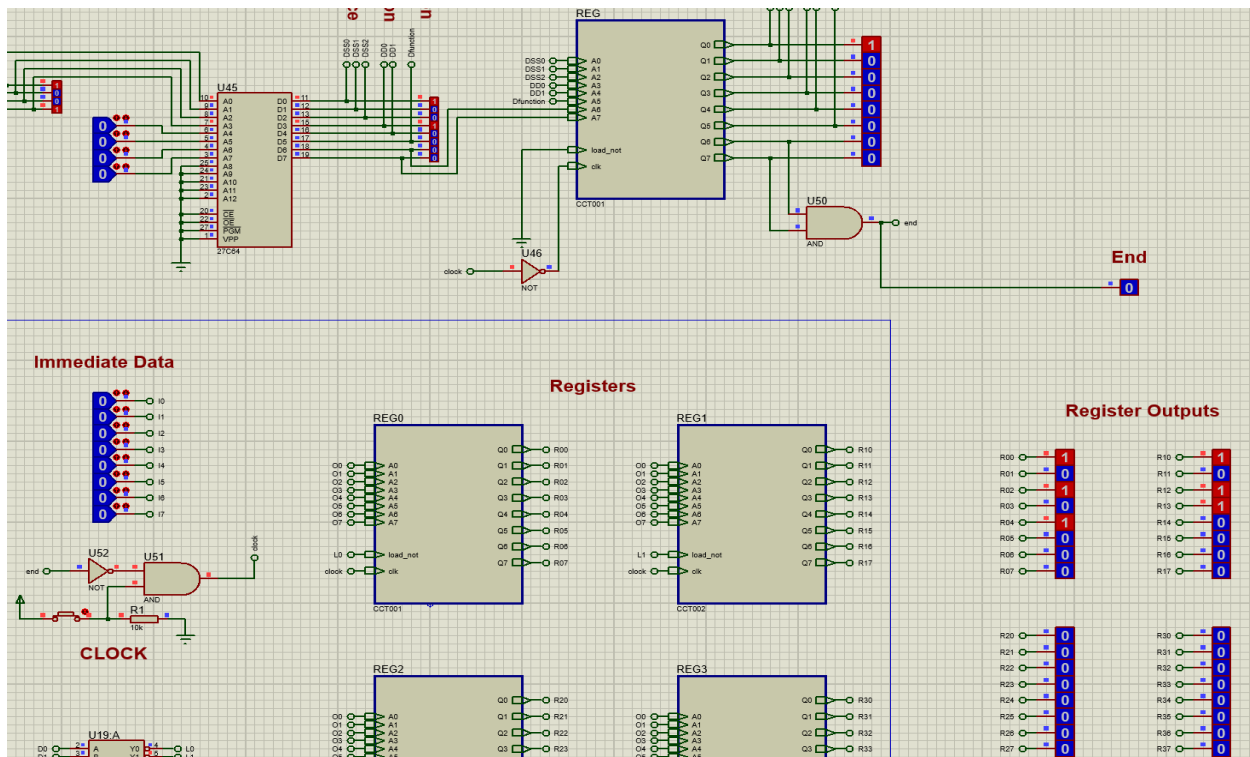
شکل ۲۲. حاصل جمع برابر ۵ است و وارد رجیستر یکم می‌شود



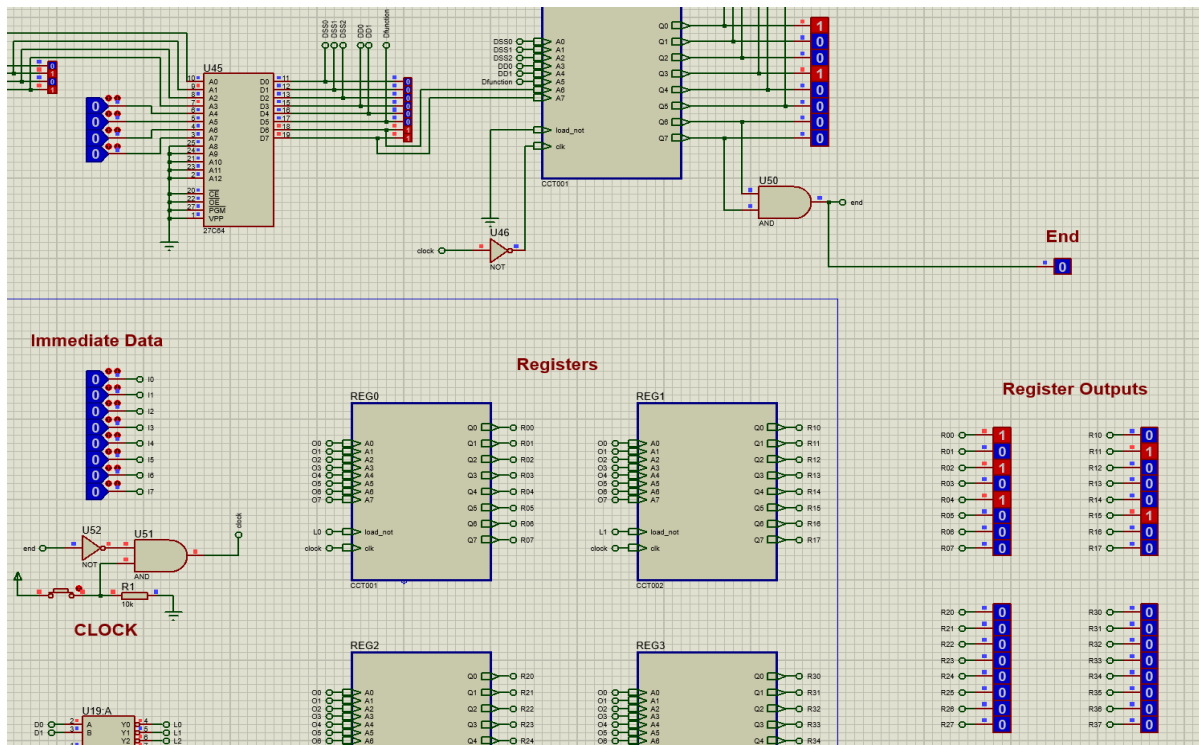
شکل ۲۳. حاصل جمع برابر ۸ است و وارد رجیستر صفرم می‌شود



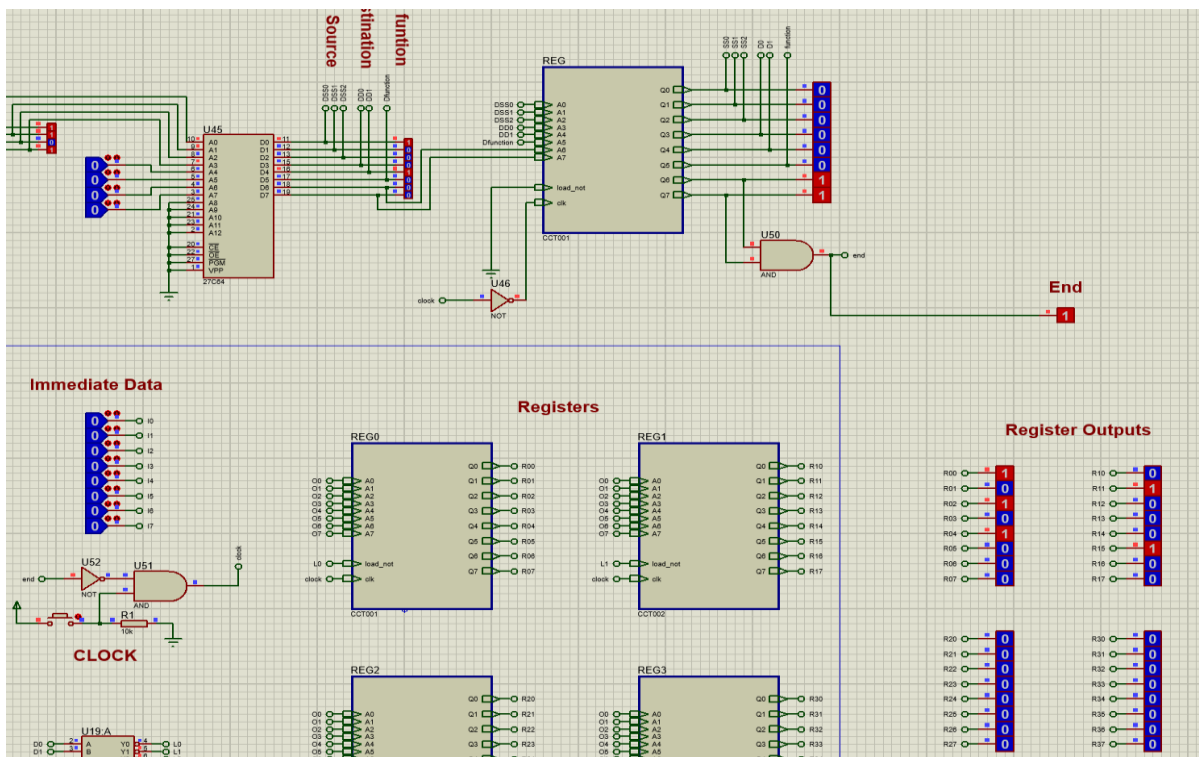
شکل ۲۴. حاصل جمع برابر ۱۳ است و وارد رجیستر یکم می‌شود



شکل ۲۵. حاصل جمع برابر ۲۱ است و وارد رجیستر صفرم می‌شود



شکل ۲۶. حاصل جمع برابر ۳۴ است و وارد رجیستر یکم می‌شود



شکل ۲۷. انتهای برنامه و روشن شدن بیت end

در نتیجه تمامی دستورات به شکل مرتب و درست، از حافظه اجرا شده و به نتیجه‌ی مطلوب رسیدیم.

پایان گزارش کار آزمایش هفتم