به نام خدا

گزارشکار پروژه هم طراحی سخت افزار و نرم افزار

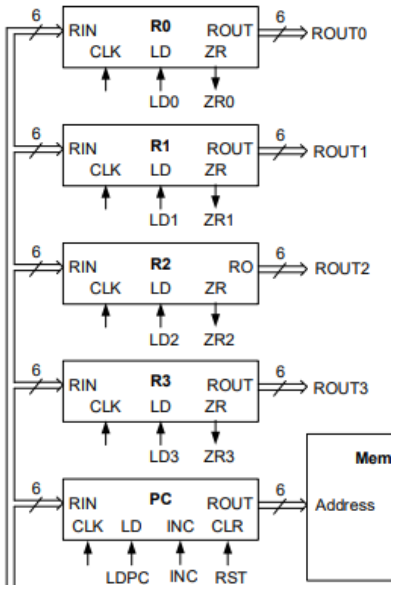
فرید افراخته – 990122680092

فهرست

بخش اول 1
پیاده سازي بخش اول 7
بخش دوم 8
پیاده سازي بخش دوم 14
بخش سوم 15
پیاده سازی بخش سوم 16
بخش امتیازی 16

بخش اول

براي پياده سازي اين مدار ابتدا به Register ها نياز داريم، تا هم بتوانيم FSM و هم ثبات هاي پردازنده را پياده سازي كنيم.



براي اين منظور ابتدا يك Type براي حالت هاي مختلف FSM تعريف كرده و دو سيگنال CurrentState و NextState را تعريف مي كنيم و سپس ساير رجيستر هاي پردازنده را نيز تعريف مي كنيم.

```
13  type State_t is (S0, HaltCheck ,S1, S2, S3, S4, S5, S6, S7);

42  --FSM
43  signal CurrentState, NextState : State_t;
44  -- Registers
45  signal R0,R1,R2,R3,IR,PC : std_logic_vector (5 downto 0);
46  signal R0Next,R1Next,R2Next,R3Next,IRNext,PCNext : std_logic_vector (5 downto 0);
```

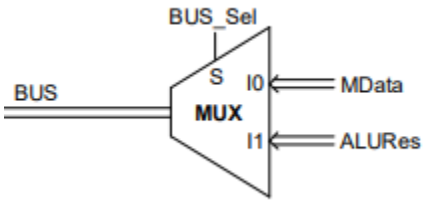
عملكرد Register ها را با نوشتن process حساس به لبه كلاك و ريست به صورت پياده سازي كرديم:

```
64      Registers: process(clk,reset)
65      begin
66          if reset='1' then
67              CurrentState <= s0;
68              IR <= (others => '0');
69              PC <= (others => '0');
70              R0 <= (others => '0');
71              R1 <= (others => '0');
72              R2 <= (others => '0');
73              R3 <= (others => '0');
74          elsif (rising_edge(clk)) then
75              CurrentState <= NextState;
76              IR <= IRNext;
77              PC <= PCNext;
78              R0 <= R0Next;
79              R1 <= R1Next;
80              R2 <= R2Next;
81              R3 <= R3Next;
82          end if;
83      end process;
```

سایر سیگنال هایي که در معماري پردازنده به صورت زیر تعریف کردیم:

```
47  -- Controls
48  signal MData, DataBUS, ALURes, IN1,IN2 : std_logic_vector(5 downto 0);
49  signal SelMux1,SelMux2 : std_logic_vector(1 downto 0);
50  signal ZR0,ZR1,ZR2,ZR3, BUSSel,LDPC,LDIR,INC,RST,CMD, LD0,LD1,LD2,LD3 : std_logic;
```

مطابق صورت پروژه مالتي پلکسر در معماري پردازنده:

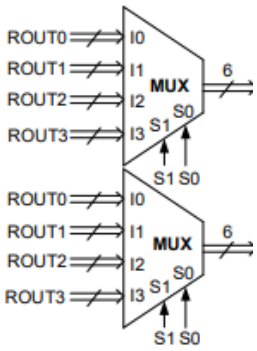


لیست حساسیت را مطابق تصویر بالا تعیین کردیم و عملکرد آن را به صورت زیر پیاده سازي نمودیم:

```
86      Mux0: process(MData, ALURes, BUSSel)
87      begin
88          case BUSSel is
89              when '0' =>
90                 DataBUS <= MData;
91              when '1' =>
92                 DataBUS <= ALURes;
93              when others =>
94                 DataBUS <= (others => '0');
95          end case;
96      end process;
```

همانطور که مشخص است MUX اگر مقدار 0 را بگیرد، MData را به خروجي مي فرستد و اگر مقدار 1 را بگیرد، ALURes را به خروجي مي فرستد.

مطابق صورت پروژه دو مالتي پلکسر دیگر در معماري پردازنده:

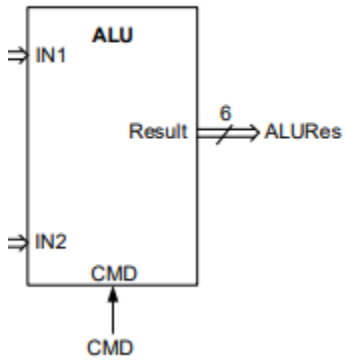


لیست حساسیت را مطابق تصویر بالا تعیین کردیم و عملکرد هر کدام را به صورت زیر پیاده سازي نمودیم:

```
98      Mux1: process(R0,R1,R2,R3,SelMux1)
99      begin
100         case SelMux1 is
101             when "00" =>
102                 IN1 <= R0;
103             when "01" =>
104                IN1 <= R1;
105             when "10" =>
106                IN1 <= R2;
107             when "11" =>
108                IN1 <= R3;
109             when others =>
110                IN1 <= (others => '0');
111         end case;
112     end process;
113
114     Mux2:process(R0,R1,R2,R3,SelMux2)
115     begin
116         case SelMux2 is
117             when "00" =>
118                IN2 <= R0;
119             when "01" =>
120                IN2 <= R1;
121             when "10" =>
122                IN2 <= R2;
123             when "11" =>
124                IN2 <= R3;
125             when others =>
126                IN2 <= (others => '0');
127         end case;
128     end process;
```

همانطور که مشخص است خروجي هاي MUX ها به ترتيب IN1 و IN2 است. خروجي اين مدارها به ورودي اول و دوم ALU متصل هستند.

مطابق صورت پروژه ALU در معماري پردازنده:

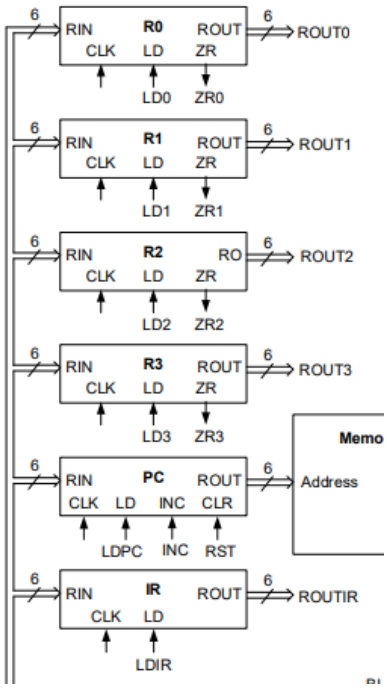


ليست حساسيت را مطابق تصوير بالا تعيين كرديم و عملكرد آن را به صورت زير پياده سازي نموديم:

```
145     ALU:process(IN1,IN2,CMD)
146     begin
147         if CMD='0' then
148             ALURes<=IN1+IN2;
149         elsif CMD='1' then
150             ALURes<=IN1-IN2;
151         end if;
152     end process;
```

به دليل اينكه دو دستور داريم، CMD را تك بيتي در نظر گرفتيم. به طوري كه اگر CMD صفر باشد، خروجي حاصل جمع دو مالتي پلكسر است؛ در غير اين صورت اگر CMD يك باشد، حاصل تفريق دو مالتي پلكسر خواهد بود.

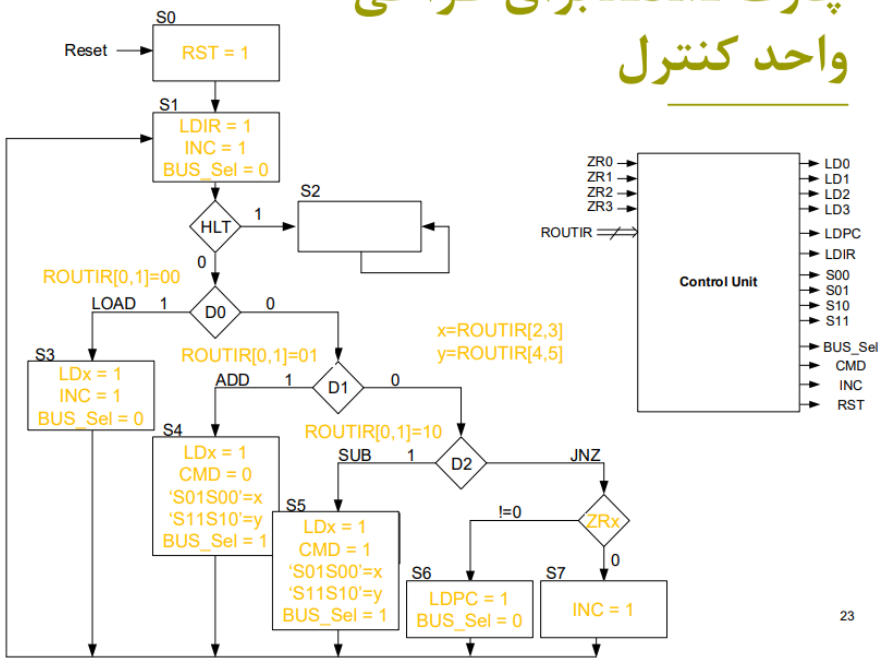
مطابق صورت پروژه خروجي هاي تركيبي مدار را پياده سازي مي كنيم:




```
131     MData <= Memory(to_integer(unsigned(PC)));
132     ZR0 <= '1' when R0="000000" else '0';
133     ZR1 <= '1' when R1="000000" else '0';
134     ZR2 <= '1' when R2="000000" else '0';
135     ZR3 <= '1' when R3="000000" else '0';
136     PCNext <= DataBUS when LDPC='1' else PC+1 when INC='1' else "000000" when RST='1' else PC;
137     IRNext <= DataBUS when LDIR='1' else IR;
138     R0Next <= DataBUS when LD0='1' else R0;
139     R1Next <= DataBUS when LD1='1' else R1;
140     R2Next <= DataBUS when LD2='1' else R2;
141     R3Next <= DataBUS when LD3='1' else R3;
```

از روي ASM Chart مطابق آنچه كه در ويديو تدريس گفته شد:

چارت ASM برای طراحی واحد کنترل



Process مربوط به FSM را به صورت Implicit طراحي مي كنيم:

```
154     process(IR, Z, CurrentState)
155     begin
156     -- Initialize signals
157     CMD <= '0';
158     INC <= '0';
159     RST <= '0';
160     LD0 <= '0';
161     LD1 <= '0';
162     LD2 <= '0';
163     LD3 <= '0';
164     LDPC <= '0';
165     LDIR <= '0';
166     SelMux1 <= "00";
167     SelMux2 <= "00";
168     BUSSel <= '0';
169
170     -- State transitions
171     case CurrentState is
172         when s0 =>
173             RST <= '1';
174             NextState <= s1;
175
176         when s1 =>
177             LDIR <= '1';
178             INC <= '1';
179             BUSSel <= '0';
180             NextState <= HaltCheck;
181
182         when HaltCheck =>
183             if IR = "111111" then
184                 NextState <= s2;
185             elsif IR(5 downto 4) = "00" then
186                 NextState <= s3;
187             elsif IR(5 downto 4) = "01" then
188                 NextState <= s4;
189             elsif IR(5 downto 4) = "10" then
190                 NextState <= s5;
191             elsif IR(5 downto 4) = "11" then
192                 if Z(index) = '0' then
193                     NextState <= s6;
194                 else
195                     NextState <= s7;
196                 end if;
197             end if;
```

```
199             when s2 =>
200                 NextState <= s2;
201
202             when s3 =>
203                 NextState <= s1;
204                 INC <= '1';
205                 BUSSel <= '0';
206                 case IR(3 downto 2) is
207                     when "00" => LD0 <= '1';
208                     when "01" => LD1 <= '1';
209                     when "10" => LD2 <= '1';
210                     when others => LD3 <= '1';
211                 end case;
```

```
213         when s4 =>
214             NextState <= s1;
215             CMD <= '0';
216             SelMux1 <= IR(3 downto 2);
217             SelMux2 <= IR(1 downto 0);
218             BUSSel <= '1';
219
220             case IR(3 downto 2) is
221                 when "00" => LD0 <= '1';
222                 when "01" => LD1 <= '1';
223                 when "10" => LD2 <= '1';
224                 when others => LD3 <= '1';
225             end case;
226
227         when s5 =>
228             NextState <= s1;
229             CMD <= '1';
230             SelMux1 <= IR(3 downto 2);
231             SelMux2 <= IR(1 downto 0);
232             BUSSel <= '1';
233
234             case IR(3 downto 2) is
235                 when "00" => LD0 <= '1';
236                 when "01" => LD1 <= '1';
237                 when "10" => LD2 <= '1';
238                 when others => LD3 <= '1';
239             end case;
```

```
241             when s6 =>
242                 NextState <= s1;
243                 LDPC <= '1';
244                 BUSSel <= '0';
245
246             when s7 =>
247                 INC <= '1';
248                 NextState <= s1;
249
250         end case;
251         end process;
252
253     end Processor;
```

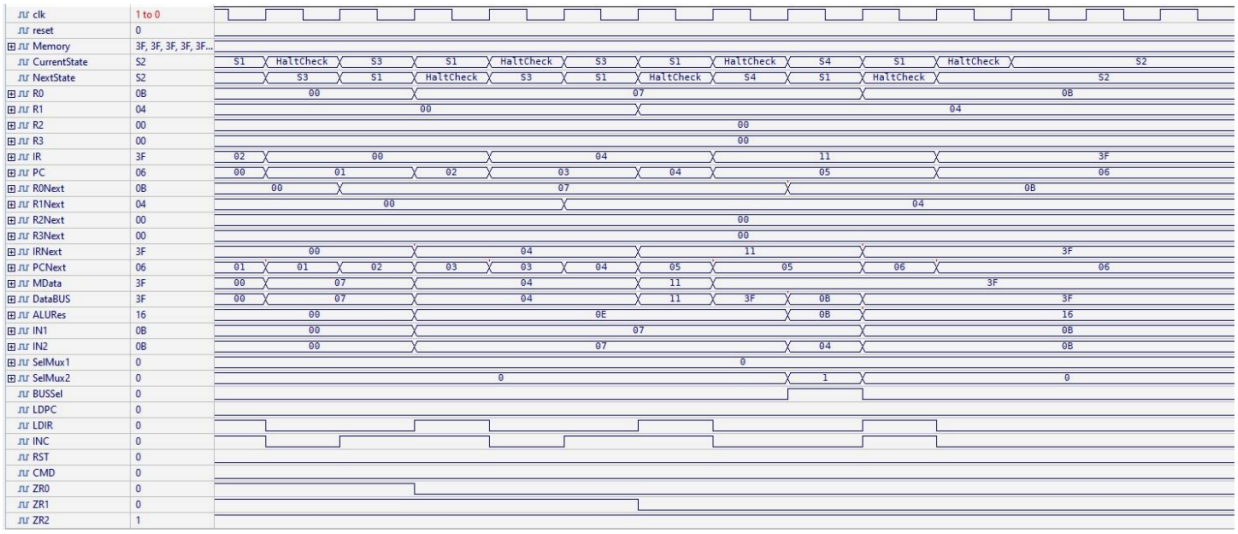
لازم است در نظر داشته باشيم براي ساده تر شدن كد حالت HaltCheck، به جاي استفاده از ZR0، ZR1، ZR2 و ZR3 يك آرايه Z ساخته شده است.

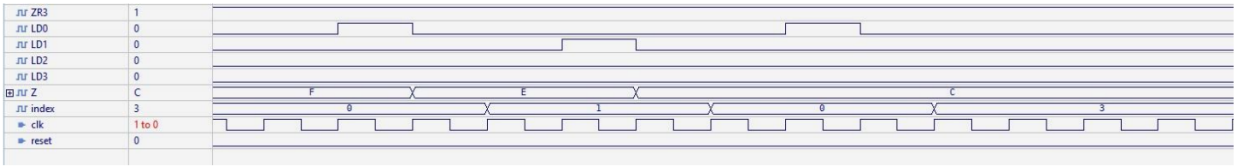
پياده سازي بخش اول

براي بررسي عملكرد بخش اول، كد خواسته شده در بخش اول را در Memory مي نويسيم:

```
12  type Memory_t is array (63 downto 0) of std_logic_vector(5 downto 0);
13  type State_t is (S0, HaltCheck ,S1, S2, S3, S4, S5, S6, S7);
14
15  signal Memory : Memory_t :=
16  (
17     -- PART 1:
18     0     => "000000", -- Load R0,
19     1     => "000111", -- 7
20     2     => "000100", -- Load R1,
21     3     => "000100", -- 4
22     4     => "010001", -- Add, R0, R1
23     others => "111111" -- Halt
24
25  );
```

توجه شود كه 111111 را حالت Halt در نظر گرفتيم.





همانطور که از نتیجه مشخص است، در آخر در Register 0 مقدار 0B ذخیره شد که معادل 11=7+4 است.

بخش دوم

برای این بخش از اسمبلری که برای بخش امتیازی آماده کردیم، کمک گرفتیم.

```
LOAD r0, 6
LOAD r1, 8
LOAD r2, 1
LOAD r3, 0
L0:add r3, r0
sub r1, r2
jnz r1, L0
```

در این قسمت کد را به اسمبلر می دهیم تا کد باینری را برای ما تولید کند:

با زبان پایتون:

متدها:

در این بخش کتابخانه مربوطه را import میکنیم:

```
sembler
2   from typing import Dict, List
3
```

در این بخش یک رشته فرعی را در یک رشته بدون حساسیت بزرگ و کوچک پیدا می کنیم:

```
3 usages
@staticmethod
def find_ci(data: str, to_search: str, pos: int = 0) -> bool:
    data = data.lower()
    return to_search in data[pos:]
```

در این بخش رشته را با جداکننده معین تقسیم می کند:

```
3 usages
@staticmethod
def split(input_string: str, delimiter: str) -> List[str]:
    return input_string.split(delimiter)
```

در این بخش یک عدد را به یک رشته باینری 6 بیتی تبدیل می کند که بین 0 و 63 گیر می کند:

```
2 usages
@staticmethod
def to_binary(num: int) -> str:
    num = max(0, min(num, 63))
    return format(num, '06b')
```

Opcode و Register :

```
opcodes: Dict[str, str] = {
    "load": "00",
    "add": "01",
    "sub": "10",
    "jnz": "11"
}

registers: Dict[str, str] = {
    "r0": "00",
    "r1": "01",
    "r2": "10",
    "r3": "11"
}

```

کلاس assembler

```
1 usage
class Assembler:
    def __init__(self, input_string: str):
        self.m_Input = input_string
        self.m_CurrentLabel = 1
        self.m_LabelAddress = {}
```

input_string: کد اسمبلی به عنوان یک رشته.

m_CurrentLabel: شماره برچسب فعلی را ردیابی می کند (در کد ارائه شده استفاده نمی شود).

m_LabelAddress: آدرس های برچسب را ذخیره می کند.

متد assemble: کد اسمبلی ورودی را به کد ماشین باینری تبدیل می کند.

```
40      def assemble(self) -> str:
41          self.calculate_label_addresses()
42          m_Output = ""
43          for line in self.m_Input.splitlines():
44              line = line.replace( _old: ',', _new: '')
45              instruction = Helpers.split(line, delimiter: ' ')
46              opcode = instruction[0]
47
48              if ':' in opcode:
49                  opcode = Helpers.split(opcode, delimiter: ':')[1]
50
51              m_Output += opcodes[opcode]
52              m_Output += registers[instruction[1]]
53
54              if Helpers.find_ci(opcode, to_search: "load"):
55                  m_Output += "00\n"
56                  m_Output += Helpers.to_binary(int(instruction[2]))
57              elif Helpers.find_ci(opcode, to_search: "jnz"):
58                  m_Output += "00\n"
59                  m_Output += Helpers.to_binary(self.m_LabelAddress[instruction[2]])
60              else:
61                  m_Output += registers[instruction[2]]
62
63              m_Output += "\n"
64
65          m_Output += "111111"
66          return m_Output
```

ورودی را خط به خط می خواند.

دستورالعمل را تجزیه و با استفاده از دیکشنری ها به باینری تبدیل می کند.

دستورالعمل های بار و jnz را به طور خاص با افزودن نمایش های دودویی مقادیر فوری یا آدرس های برچسب کنترل می کند.

یک ترمیناتور "111111" را در پایان اضافه می کند.

متد get_command_size:

```
1 usage
def get_command_size(self, command: str) -> int:
    if Helpers.find_ci(command, to_search: "load"):
        return 2
    else:
        return 1
```

این بخش اندازه یک دستور (یا 1 یا 2) را برمی‌گرداند.

متد calculate_label_addresses:

```
1 usage
def calculate_label_addresses(self):
    current_address = 0
    for line in self.m_Input.splitlines():
        if ':' in line:
            self.m_LabelAddress[Helpers.split(line, delimiter: ':')[0]] = current_address
        current_address += self.get_command_size(line)
```

این بخش آدرس ها را برای برچسب ها در کد اسمبلی محاسبه می کند.

بخش انتهایی یک فایل اسمبلی ورودی را می خواند و خروجی باینری مونتاژ شده را چاپ می کند.

```
81 if __name__ == "__main__":
82     input_file = 'input.txt'
83     with open(input_file, 'r') as file:
84         input_string = file.read()
85
86     assembler = Assembler(input_string)
87     print("output:\n" + assembler.assemble())
88
```

خروجی با توجه به ورودی ما:

```
C:\Users\Farid\AppData\Local\Programs\Python\Py
output:
000000
000110
000100
001000
000100
000001
001100
000000
011100
100110
110100
001000
111111

Process finished with exit code 0
```

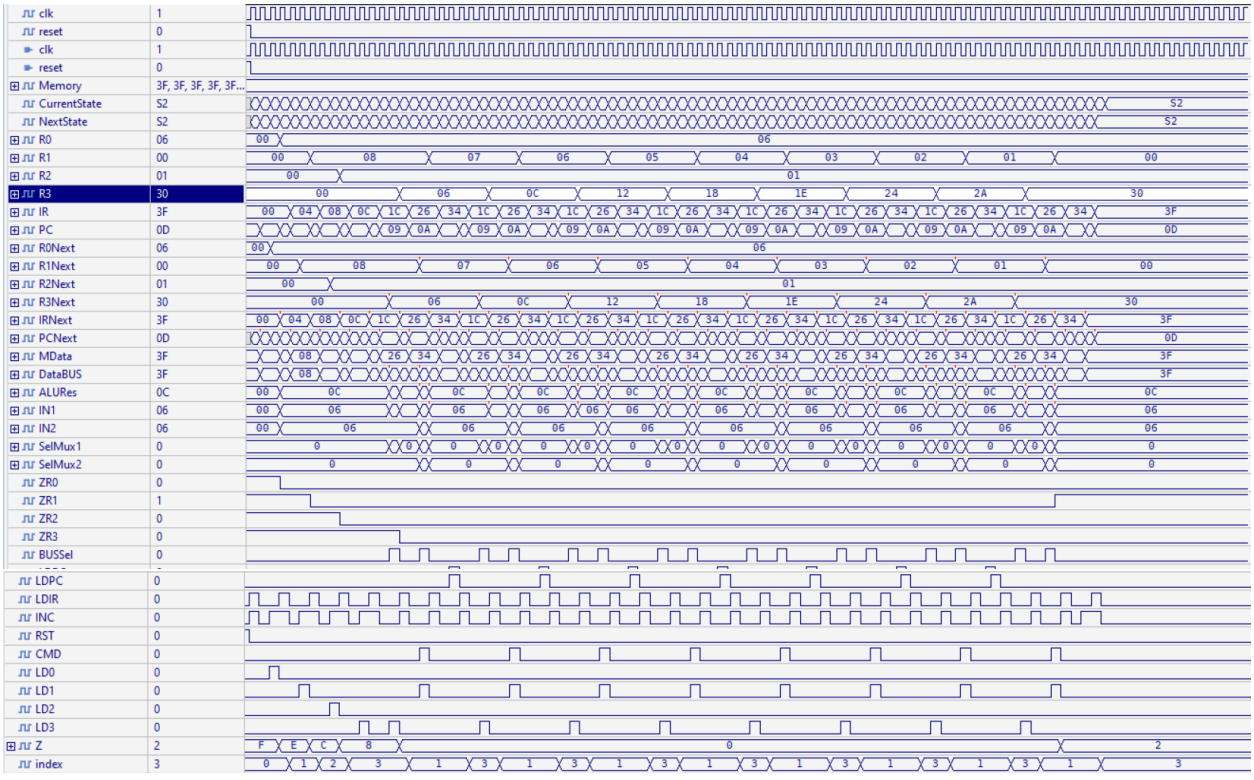
حال کد تولید شده توسط اسمبلر را در حافظه پردازنده می نویسیم:

```
signal Memory : Memory_t :=
(
   -- PART 1:
   --0     => "000000", -- Load R0,
   -- 1     => "000111", -- 7
   -- 2     => "000100", -- Load R1,
   -- 3     => "000100", -- 4
   --4     => "010001", -- Add, R0, R1
   --others => "111111" -- Halt

    --PART 2:
    0  => "000000",
    1  => "000110",
    2  => "000100",
    3  => "001000",
    4  => "001000",
    5  => "000001",
    6  => "001100",
    7  => "000000",
    8  => "011100",
    9  => "100110",
    10 => "110100",
    11 => "001000",
    others => "111111"

);
```

پیاده سازي بخش دوم



همانطور که مشخص است مقدار 30 هگزادسیمال و همان 48 دهدهي در رجیستر R3 ذخیره شده است.

بخش سوم

برای اضافه کردن دستور Mult لازم است ALU خود را ارتقا دهیم. برای این کار باید خط کنترلی ALU خود را از حالت تک بیتی به حالت دو بیتی افزایش دهیم. همچنین نیاز است ALURes را بزرگتر کنیم تا بتواند نتیجه حاصل ضرب را در خروجی نمایش دهد.

بنابراین ALU جدید ما به صورت زیر خواهد بود:

```
129     process(IN1, IN2, CMD)
130     begin
131     case CMD is
132         when "00" =>
133             ALURes <= "0000000" & (IN1 + IN2);
134         when "01" =>
135             ALURes <= "0000000" & (IN1 - IN2);
136         when "10" =>
137             ALURes <= IN1 * IN2;
138         when others =>
139             ALURes <= (others => '0');
140     end case;
141     end process;
```

در کد بالا کلا 3 حالت داریم، حالت others عملا هیچ گاه رخ نمی دهد. تنها زمانی ممکن است اجرا شود که خطایی رخ داده باشد. ما برای اینکه تمام حالت های case نوشته شود، آن را نیز اضافه کردیم و صرفا خروجی ALU را تمام صفر در نظر گرفتیم.

به خاطر اینکه قرار است دستور جدید اضافه شود، مجبوریم Opcode را از 2 بیت به 3 بیت افزایش دهیم. برای این منظور Opcode های قدیم همان باقی می مانند اما یک 0 به قبل از آن ها اضافه می شود، اما Opcode جدید یعنی Mult، 100 در نظر گرفته می شود. برای این منظور نیاز داریم تا سایز حافظه خود را از خانه ای 6 بیت، به خانه ای 7 بیت افزایش دهیم تا بتوانیم Opcode ها را در آن نگهداری کنیم.

حال نیاز داریم که یک حالت جدید برای FSM خود بنویسیم تا عمل ضرب انجام شود. یک حالت جدید به نام S8 درست می کنیم و در حالت Halt یا Not بررسی می کنیم. اگر IR، حاوی دستور ضرب یعنی 100 بود، به این حالت می رویم:

```
187             elsif IR(6 downto 4) = "100" then
188                 NextState <= s8;
189             else
190                 NextState <= s1;
191             end if;
```

سایر شرط های این قسمت را مانند همان قبلی می ماند. این تفاوت که یک بیت 0 به قبل از آن ها اضافه می شود.

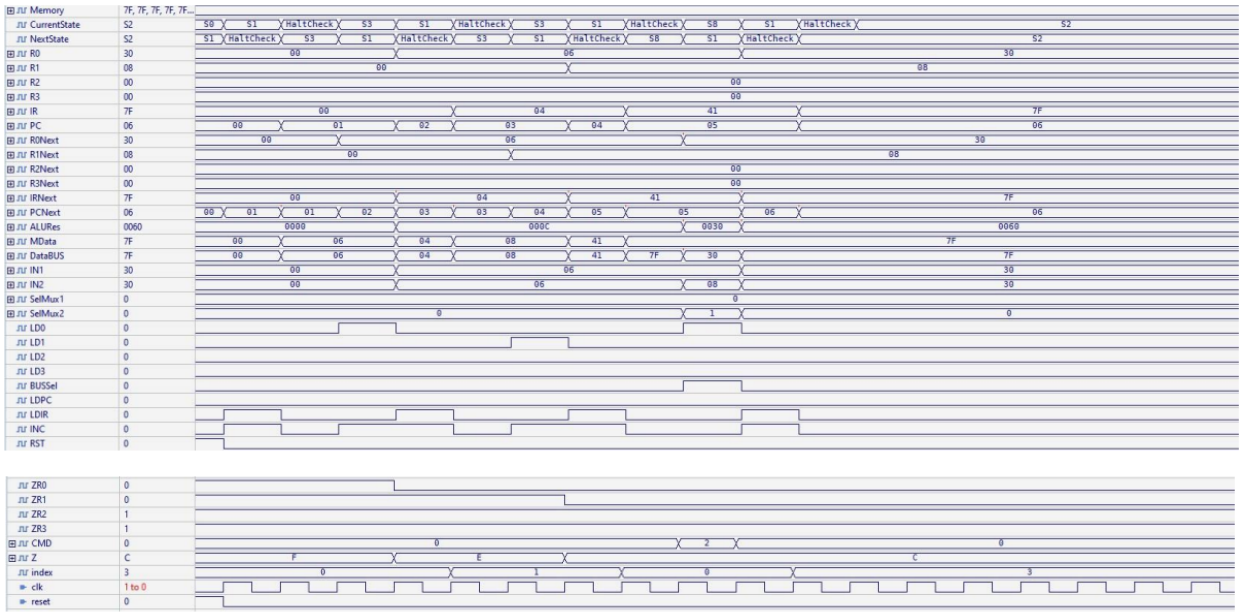
برای انجام ضرب نیز خواهیم داشت:

```
244         when s8 =>
245         NextState <= s1;
246         CMD <= "10";
247         BUSSel <= '1';
248         SelMux1 <= IR(3 downto 2);
249         SelMux2 <= IR(1 downto 0);
250
251         case IR(3 downto 2) is
252         when "00" => LD0 <= '1';
253         when "01" => LD1 <= '1';
254         when "10" => LD2 <= '1';
255         when others => LD3 <= '1';
256         end case;
```

در نهایت کد ضرب عدد 6 در 8 را در حافظه پردازنده خود می نویسیم:

```
signal Memory : Memory_t :=
(
    --PART 3:
    0 => "0000000", -- Load R0,
    1 => "0000110", -- 6
    2 => "0000100", -- Load R1,
    3 => "0001000", -- 8
    4 => "1000001", -- Mult, R0, R1
    others => "1111111" -- Halt
);
```

پیاده سازی بخش سوم



همانطور که مشخص است، در آخر در ثبات R0، ضرب دو عدد 6 و 8 ذخیره می شود. 30 هگزا دسیمال معادل همان 48 دهدهی است و مدار ما به درستی کار می کند.

بخش امتیازی

در بخش دوم کد Assembly نوشته شده نسبتا طولانی بود؛ از اسمبلری که نوشتیم، برای تولید کد باینری استفاده شد.

کد پایتون اسمبلر از Label پشتیبانی می کند و به صورت خودکار آدرس Label را حساب کرده و در دستور JNZ قرار می دهد و نیازی به Hardcode کردن آدرس وجود ندارد. همانطور که مشخص است در بخش دوم نیز از قابلیت اسمبلر استفاده کردیم.