中国科学技术大学计算机学院

《数字电路实验》报告

实验题目：综合实验

学生姓名：傅申

学生学号：PB20000051

完成日期：2021 年 12 月 30 日

计算机实验教学中心制

2020 年 09 月

【实验题目】

综合实验

【实验目的】

- 熟练掌握前面实验中的所有知识点
- 熟悉几种常用通信接口的工作原理及使用
- 独立完成具有一定规模的功能电路设计

【实验环境】

- Windows PC
- Microsoft Visual Studio Code
- Xilinx Design Tools Vivado HL Design Edition 2019.1
- FPGAOL

【实验练习】

在 FPGAOL 平台上实现了一个简单的 LC-3 片上系统, 并且能正确运行示例程序, 但是没有通过串口进行交互式操作.

LC-3 简介

LC-3 (Little Computer 3) 是一个图灵完备的 16 位计算机, 拥有 $2^{16} \times 16$ bits 的内存, 8 个 16 位寄存器和若干个系统寄存器 (如 PC, IR, Condition Codes, etc.). 它拥有包含 15 条指令的指令集, 每条指令的操作码为指令的高 4 位. 具体的指令集如图 1 和表 1, 其中带有上标 $^+$ 的指令会修改条件码寄存器.

修改与简化

与真正的 LC-3 不同, 实现的片上系统针对特殊情况进行了适当的简化, 具体如下:

1. LC-3 的内存过于巨大, 如果使用 IP 核进行生成, 将会耗费很多事件, 因此这里将内存大小减小为 $2^{12} \times 8$ bits. 并且取原地址线的低 12 位作为内存的地址线.
2. 由于没有用到串口, LC-3 中除了 HALT 以外的系统函数都不会起作用, 因此 TRAP 指令被指定为是 HALT, 并且 RTI 指令并不会被调用.
3. LC-3 中有 60 余个状态的有限状态机被化简为只有 25 个状态的有限状态机 (如图), 并且 LC-3 的数据通路被化简 (如图 2).

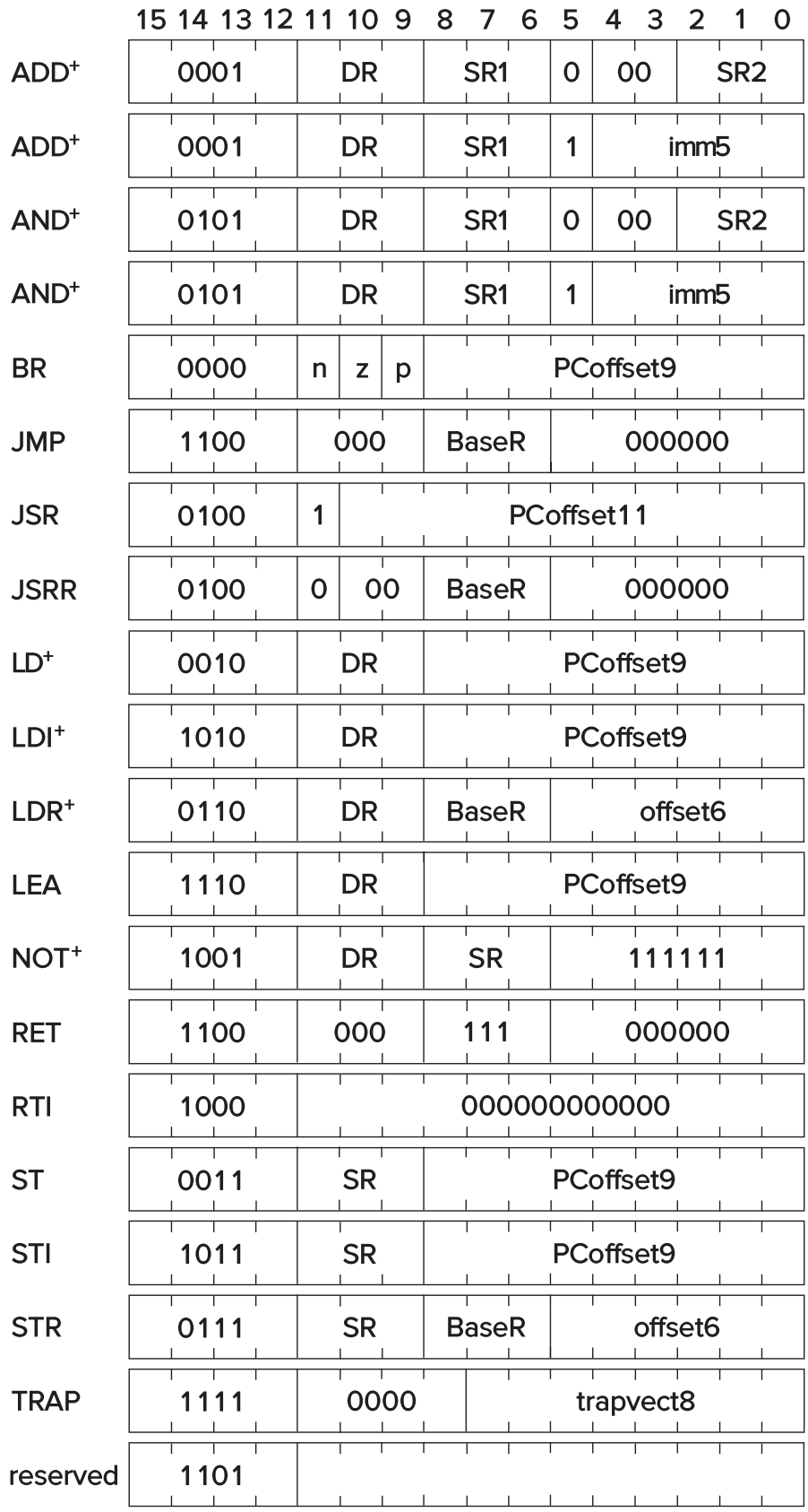

15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
ADD+ 0001 DR SR1 0 00 SR2
ADD+ 0001 DR SR1 1 imm5
AND+ 0101 DR SR1 0 00 SR2
AND+ 0101 DR SR1 1 imm5
BR 0000 n z p PCoffset9
JMP 1100 000 BaseR 000000
JSR 0100 1 PCoffset11
JSRR 0100 0 00 BaseR 000000
LD+ 0010 DR PCoffset9
LDI+ 1010 DR PCoffset9
LDR+ 0110 DR BaseR offset6
LEA 1110 DR PCoffset9
NOT+ 1001 DR SR 111111
RET 1100 000 111 000000
RTI 1000 000000000000
ST 0011 SR PCoffset9
STI 1011 SR PCoffset9
STR 0111 SR BaseR offset6
TRAP 1111 0000 trapvect8
reserved 1101

图 1: LC-3 的指令集

表 1: LC-3 的指令集

指令名	指令全称	操作码	操作
ADD$^+$	Addition	0001	`DR = SR1 + SR2` 或 `DR = SR1 + imm5`
AND$^+$	Bit-Wise Logical AND	0101	`DR = SR1 & SR2` 或 `DR = SR1 & imm5`
BR	Conditional Branch	0000	`if (nzp & cond_code)` `PC = PC + PCoffset9`
JMP	Jump	1100	`PC = BaseR`

JSR	Jump to Subroutine	0100	`Temp = RC`
			`PC = BaseR`
			或 `PC = PC + offset11`
			`R7 = Temp`

LD$^+$	Load	0010	`DR = mem[PC + PCoffset9]`
LDI$^+$	Load Indirect	1010	`DR = mem[mem[PC + PCoffset9]]`
LDR$^+$	Load Base+Offset	0110	`DR = mem[BaseR + offset6]`
LEA	Load Effective Address	1110	`DR = PC + PCoffset9`
NOT$^+$	Bit-Wise Logical NOT	1001	`DR = SR1`
RTI	Return from Trap or Interrupt	1000	`PC, PSR = top of system stack`
ST	Store	0011	`mem[PC + PCoffset9] = SR`
STI	Store Indirect	1011	`mem[mem[PC + PCoffset9]] = SR`
STR	Store Base+Offset	0111	`mem[BaseR + offset6] = SR`
TRAP	System Call	0000	Calls a system subroutine

数据通路

LC-3 的数据通路遵循冯 · 诺依曼结构, 即包括内存, 处理单元, 控制单元和 I/O, 在本次实验中, 处理单元和控制单元被集成到有限状态机中, 而 I/O 则是 FPGAOL 上的按钮与七位数码管. 如下图 2.

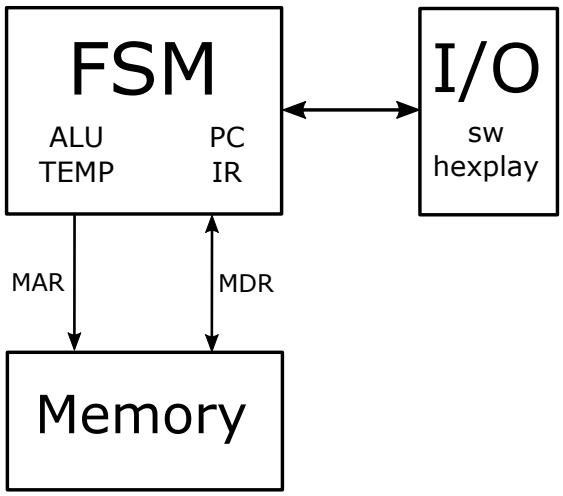


图 2: 设计的数据通路

其中 Memory 由 IP 核实现, 是一个 Single Port RAM, 位深为 4096, 位宽为 16 bits. 内存的初始值即为示例程序, 由下面的 python 程序调用汇编器导入到 coe 文件中.

代码 1: 初始化内存的 Python 程序

```python
import os
os.system("assembler.exe -s -f path\to\asm -o path\to\outputfile")
code = []
with open("path\to\outputfile", "r") as fin:
  code = fin.readlines()
vector = ''
for ins in code:
  vector = vector + ins[:-1] + ' '
vector = vector[:-1] + ';'
with open("path\to\coe", "w+") as fout:
  fout.write("memory_initialization_radix=16;\n")
  fout.write("memory_initialization_vector=%s\n" % vector)
```

例如示例代码如下:

代码 2: 示例汇编代码

```
        .ORIG x3000
        ADD     R0, R0, #0
        BRz     Stop
        BRp     Pos
        NOT     R0, R0 ; Negate R0
```

```
6           ADD     R0, R0, #1
7           NOT     R1, R1 ; Nagate R1
8           ADD     R1, R1, #1
9   Pos     ADD     R2, R2, #1
10  Loop    AND     R3, R0, R2
11          BRnp    BitOne
12          ADD     R1, R1, R1
13          ADD     R2, R2, R2
14          BRnzp   Loop
15  BitOne  ADD     R7, R7, R1
16          ADD     R1, R1, R1
17          ADD     R2, R2, R2
18          ADD     R4, R0, #-1 ; Remove the lowest
19          AND     R0, R4, R0  ; 1 in R0
20          BRnp    Loop
21  Stop    HALT
22          .END
```

导入的 coe 文件如下

代码 3: coe 文件

```
1  memory_initialization_radix=16;
2  memory_initialization_vector=1020 0411 0204 903F 1021 927F 1261
       14A1 5602 0A03 1241 1482 0FFB 1FC1 1241 1482 183F 5100 0BF5
       F025;
```

有限状态机

在本次实验中, 有限状态机被简化成只有 25 个状态, 具体的状态转移图如图 . 其中最下面的 fetch_0' 状态与最上面的 fetch_0 状态是同一个状态.

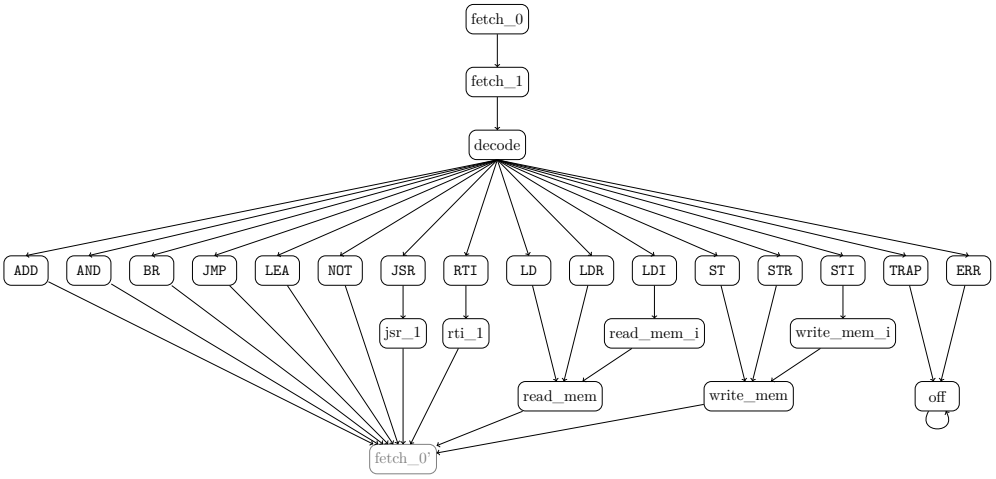


图 3: 简化后的状态转移图

对于在执行过程 (除去 fetch 和 decode) 中只有一个状态的指令, 如 `ADD`, `AND`, 其在执行过程的操作与表 1 中无异. 而其他状态的操作如下表 2 所示.

表 2: 部分状态的操作

状态	操作
fetch_0	MAR <- PC, PC <- PC + 1, 刷新寄存器缓冲
fetch_1	IR <- MDR, 刷新 PC 缓冲
decode	根据 IR 高 4 位确定下一状态
`RTI`	MAR <- R6, R6 <- R6 + 2
rti_i	PC <- MDR
`JSR`	temp <- PC, PC <- PC + PCoffset11 或 BaseR
jsr_i	R7 <- temp
`LD`	MAR <- PC + PCoffset9
`LDR`	MAR <- BaseR + offset6
`LDI`	MAR <- PC + PCoffset9
read_mem_i	MDR <- MDR
read_mem	DR <- MDR
`ST`	MDR <- DR, MAR <- PC + PCoffset9, mem_we <- 1
`STR`	MDR <- DR, MAR <- BaseR + offset6, mem_we <- 1
`STI`	MAR <- PC + PCoffset9, mem_we <- 0
write_mem_i	MAR <- MDR, MDR <- DR, mem_we <- 1
write_mem	mem_we <- 0

实际上 MDR 由 MDR_in 和 MDR_out 组成, 上面的 MDR 只是一个统称.

其他设计细节

内存

上面提到了在本次实验中使用 IP 核生成了一个 4096 × 16 bits 的 Single Port RAM, IP 核的设计界面如图 4. 而内存的实际接线情况见代码 4.

代码 4: 内存的实际接线情况

```
reg  [15:0] MAR, MDR_in;
reg  mem_we;
wire [15:0] MDR_out;
wire [11:0] MAR12;
initial mem_we <= 1'b0;
assign MAR12 = MAR[11:0];
memory mem(.a(MAR12),
           .d(MDR_in),
           .spo(MDR_out),
           .we(mem_we),
           .clk(clk_100mhz));
```

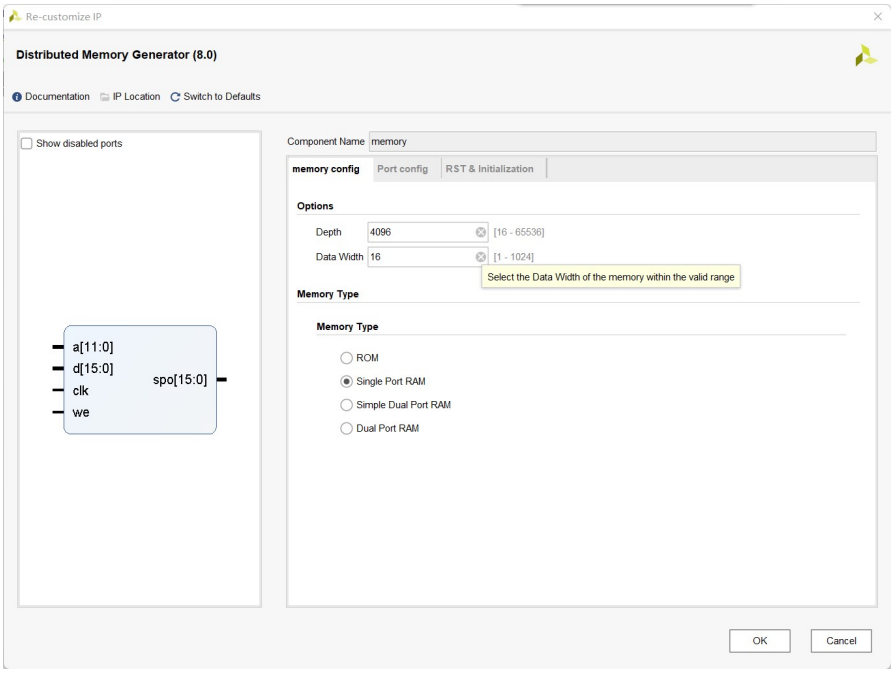


图 4: IP 核设计界面

状态机编码

状态机总共 25 个状态, 状态编码共 5 位, 其中指令的编码为 0 + 操作码, 其他状态的编码则为 1xxxx. 这样, 在 decode 阶段, 只需要将 `{1'b0,IR[15:12]}` 赋值给下一阶段即可. 具体的状态机细节可见详细代码 7.

条件码 Conditional Code

条件码用于指示指令的执行结果的正负性, 在本次实验中仅与寄存器类型的 `result` 有关, 在会改变条件码的指令周期的最后一个状态, 执行结果会被赋值给 `result` 寄存器, 然后通过以下的组合逻辑电路 (代码 5) 赋值给 `cc` 寄存器.

代码 5: 条件码的组合逻辑

```
reg  [15:0] result;
wire [2:0]  cc;
assign cc = (result == 16'h0000) ? 3'b010 :
             ((result[15] == 1'b0) ? 3'b001 : 3'b100); // setCC
```

寄存器缓冲

为了避免在时序逻辑电路的赋值中可能的自赋值冲突的情况 (比如 `ADD R0, R0, R0` 指令以及 PC 加上偏移的情况), 在本次实验中设计了缓冲, 包括寄存器缓冲 `R_buffer` 和 PC 缓冲 `PC_buffer`. 其中 `R_buffer` 在 fetch_0 状态刷新 (`R_buffer <- R`), 而 `PC_buffer` 在 fetch_1 状态刷新 (`PC <- PC_buffer`).

I/O

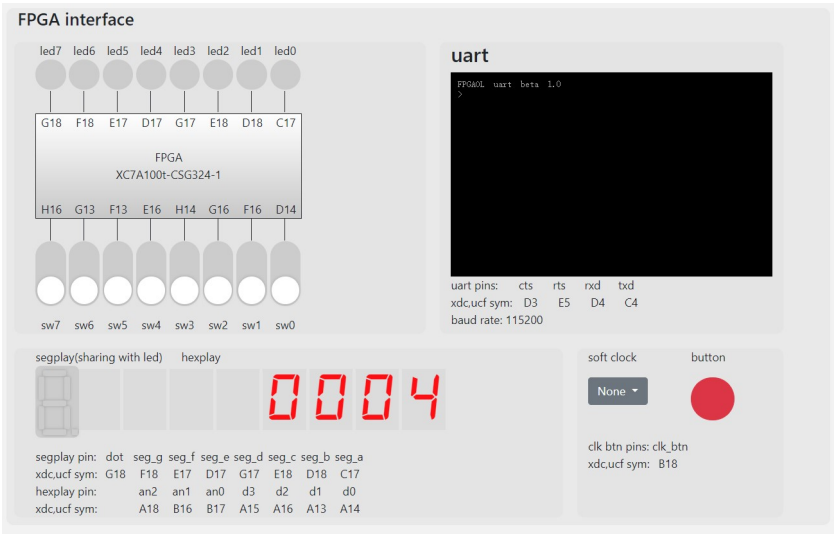
本次实验用到了四个 sw 开关, 一个 btn 按钮作为输入, 四个七位数码管作为输出. 其中 sw 控制数码管显示的值, 而 btn 则控制着程序的启动. 在比特流文件烧入至 FPGAOL 平台后, LC-3 并不会马上启动, 而是需要按下 btn 后才能开始运行示例程序. 而 sw 的值与七位数码管显示的值的关系如下代码 6.

代码 6: sw 与 hexplay 的关系

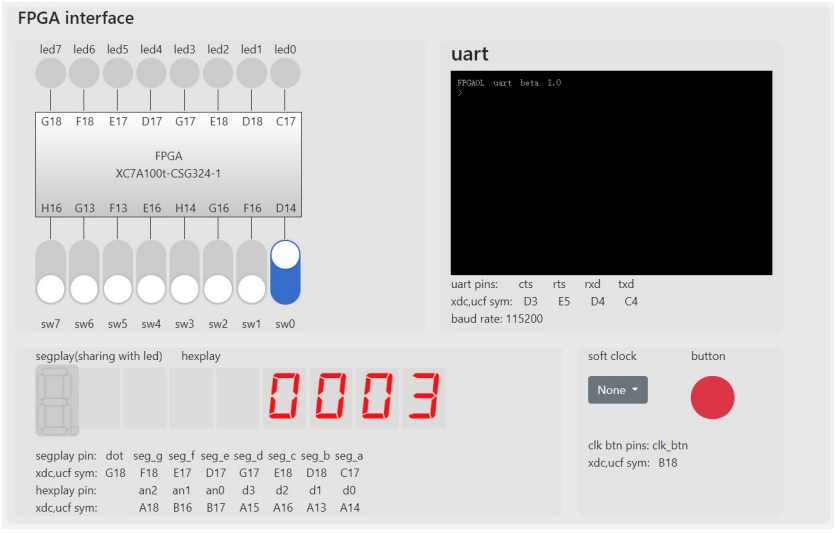
```
always @(posedge clk_100mhz) begin
case(sw)
    4'h0: data <= R[0];
    4'h1: data <= R[1];
    4'h2: data <= R[2];
    4'h3: data <= R[3];
    4'h4: data <= R[4];
    4'h5: data <= R[5];
    4'h6: data <= R[6];
    4'h7: data <= R[7];
    4'h8: data <= {7'h00, cc[2], 3'h0, cc[1], 3'h0, cc[0]};
    4'h9: data <= PC;
    4'hA: data <= IR;
    4'hB: data <= MAR;
    4'hC: data <= MDR_out;
    default: data <= 16'h0000;
endcase
end
```

运行示例程序

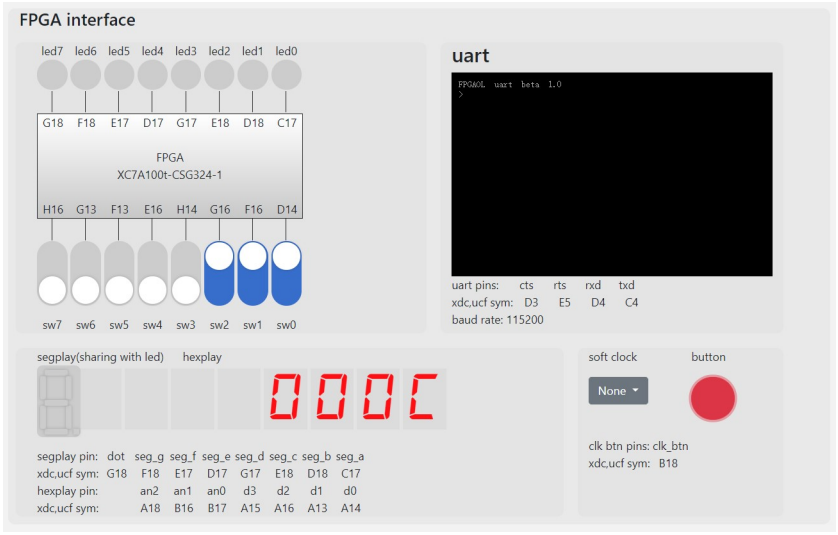
前面提到我们将代码 2 的程序导入到了内存中, 该程序的功能实际上是 `R7 <- R0 * R1`, 将寄存器初值设为 `{3, 4, 0, 0, 0, 0, 0, 0}`, 生成比特流并烧写至 FPGAOL 平台, 系统运行前后的输出如图 5 所示. 可以看到在 5(c) 中, 程序运行完后 R7 的值为 `x000C`, 符合预期.



(a) 运行前的 R0



(b) 运行前的 R1



(c) 运行后的 R7

图 5: 运行前后的寄存器状态

源代码

不考虑模拟文件, 整个项目总共有两个文件, 分别是 LC-3 的模块文件与约束文件. 模块文件如代码 7, 约束文件如代码 8.

代码 7: 模块文件源代码

```
// LC-3.srcs/sources_1/new/main.v
module main(
    input clk_100mhz, btn,
    input [3:0] sw,
    output reg [1:0] hexplay_an,
    output reg [3:0] hexplay_data
    );
// Control LC-3 clock
reg run;
initial run <= 1'b0;
always @(posedge clk_100mhz) if (btn) run <= 1'b1;
// MEMORY
reg [15:0] MAR, MDR_in;
wire [15:0] MDR_out;
wire [11:0] MAR12;
reg mem_we;
initial mem_we <= 1'b0;
assign MAR12 = MAR[11:0];
memory mem(.a(MAR12),
            .d(MDR_in),
            .spo(MDR_out),
            .we(mem_we),
            .clk(clk_100mhz));
// Finite State Machine
    // control unit
    reg [15:0] PC, IR, PC_buffer;
    initial begin
        PC <= 16'h0000;
        IR <= 16'h0000;
        PC_buffer <= 16'h0000;
    end
    // states
        reg [4:0] curr_state, next_state;

        // instruction code, the state is 0+opcode

        // Add: DR = SR1 + SR2 or DR = SR1 + imm5
        parameter ADD  = 5'b00001;
```

```
39          // And: DR = SR1 & SR2 or DR = SR1 & imm5
40          parameter AND  = 5'b00101;
41          // Branch: if CC(condition code) meet the
42          // requirement the change PC
43          parameter BR   = 5'b00000;
44          // Jump: PC = BaseR
45          parameter JMP  = 5'b01100;
46          // Jump to SubRoutine: Call a subroutine
47          // temp = PC
48          // then PC = BaseR or PC += PCoffset11
49          // then R[7] = temp
50          parameter JSR  = 5'b00100;
51          // Load: DR = mem[PC + PCoffset9]
52          parameter LD   = 5'b00010;
53          // Load Indirect:
54          // DR = mem[mem[PC + PCoffset9]]
55          parameter LDI  = 5'b01010;
56          // Load Base+offset:
57          // DR = mem[BaseR + offset6]
58          parameter LDR  = 5'b00110;
59          // Load Effective Address:
60          // DR = PC + PCoffset9
61          parameter LEA  = 5'b01110;
62          // Not: DR = ~ SR
63          parameter NOT  = 5'b01001;
64          // Return from Trap or Interrupt
65          // (Not effctive since we won't use TRAP
66          // except TRAP x25(HALT))
67          parameter RTI  = 5'b01000;
68          // Store: mem[PC + PCoffset9] = SR(DR here)
69          parameter ST   = 5'b00011;
70          // Store Indirect:
71          // mem[mem[PC + PCoffset9]] = SR(DR here)
72          parameter STI  = 5'b01011;
73          // Store Base+offset:
74          // mem[BaseR + offset6] = SR(DR here)
75          parameter STR  = 5'b00111;
76          // System call:
77          // treated as HALT, stop the system
78          parameter TRAP = 5'b01111;
79          // Error Instrution
80          parameter ERR  = 5'b01101;
```

```
81
82          // instruction cycle phrases and indirect state
83
84          // System is off
85          parameter off         = 5'h10;
86          // Fetch the instructions
87          // IR = mem[PC], PC = PC + 1
88          parameter fetch_0     = 5'h11;
89          parameter fetch_1     = 5'h12;
90          // Decode IR to the corresponding state
91          parameter decode      = 5'h13;
92          // Access the memory
93          parameter read_mem    = 5'h14;
94          parameter write_mem   = 5'h15;
95          parameter read_mem_i  = 5'h16;
96          parameter write_mem_i = 5'h17;
97          // transfering state
98          parameter jsr_1       = 5'h18;
99          parameter rti_1       = 5'h19;
100
101         initial curr_state <= fetch_0;
102     // FSM Part 1
103     always @(posedge clk_100mhz) begin
104         case (curr_state)
105             fetch_0:     next_state <= fetch_1;
106             fetch_1:     next_state <= decode;
107             decode:      next_state <= {1'b0, IR[15:12]};
108             ADD:         next_state <= fetch_0;
109             AND:         next_state <= fetch_0;
110             BR:          next_state <= fetch_0;
111             JMP:         next_state <= fetch_0;
112             JSR:         next_state <= jsr_1;
113             LD:          next_state <= read_mem;
114             LDI:         next_state <= read_mem_i;
115             LDR:         next_state <= read_mem;
116             LEA:         next_state <= fetch_0;
117             NOT:         next_state <= fetch_0;
118             RTI:         next_state <= rti_1;
119             ST:          next_state <= write_mem;
120             STI:         next_state <= write_mem_i;
121             STR:         next_state <= write_mem;
122             TRAP:        next_state <= off;
```

```
123             ERR:         next_state <= off;
124             read_mem:    next_state <= fetch_0;
125             write_mem:   next_state <= fetch_0;
126             read_mem_i:  next_state <= read_mem;
127             write_mem_i: next_state <= write_mem;
128             jsr_1:       next_state <= fetch_0;
129             rti_1:       next_state <= fetch_0;
130             off:         next_state <= off;
131             default:     next_state <= off;
132         endcase
133     end
134     // FSM Part 2 (Processing Unit)
135     reg  [15:0] R_buffer [7:0], R [7:0], result, temp;
136     wire [2:0]  cc, dr, sr1, sr2, baseR;
137     wire [15:0] imm5, PCoffset9, PCoffset11, offset6; // the
        numbers in the end of instruction after sign-extension
138     assign cc         = (result == 16'h0000) ? 3'b010 : ((result[
        15] == 1'b0) ? 3'b001 : 3'b100); // setCC
139     assign dr         = IR[11:9];
140     assign sr1        = IR[8:6];
141     assign sr2        = IR[2:0];
142     assign baseR      = IR[8:6];
143     assign imm5       = {{11{IR[4]}}, IR[4:0]};
144     assign PCoffset9  = {{7{IR[8]}} , IR[8:0]};
145     assign PCoffset11 = {{5{IR[10]}}, IR[10:0]};
146     assign offset6    = {{10{IR[5]}}, IR[5:0]};
147     initial begin
148         R[0]   <= 16'h0004;
149         R[1]   <= 16'h0003;
150         R[2]   <= 16'h0000;
151         R[3]   <= 16'h0000;
152         R[4]   <= 16'h0000;
153         R[5]   <= 16'h0000;
154         R[6]   <= 16'h0000;
155         R[7]   <= 16'h0000;
156         result <= 16'h0000;
157         temp   <= 16'h0000;
158     end
159     always @(posedge clk_100mhz) begin
160         if (run) begin
161             case (curr_state)
162                 fetch_0:begin
```

```
163                         // Access the memory
164                         MAR <= PC_buffer;
165                         PC  <= PC_buffer + 16'h0001;
166                         // Refresh register buffer
167                         R_buffer[0] <= R[0];
168                         R_buffer[1] <= R[1];
169                         R_buffer[2] <= R[2];
170                         R_buffer[3] <= R[3];
171                         R_buffer[4] <= R[4];
172                         R_buffer[5] <= R[5];
173                         R_buffer[6] <= R[6];
174                         R_buffer[7] <= R[7];
175                         curr_state <= next_state;
176                     end
177                 fetch_1:begin
178                         // Load the instruction
179                         IR   <= MDR_out;
180                         // Refresh PC buffer
181                         temp <= PC;
182                         PC_buffer <= PC;
183                         curr_state <= next_state;
184                     end
185                 ADD: begin // Add: DR = SR1 + SR2 or DR = SR1 +
                        imm5
186                       if (IR[5]) begin
187                           R[dr]  <= R_buffer[sr1] + imm5;
188                           result <= R_buffer[sr1] + imm5;
189                       end else begin
190                           R[dr]  <= R_buffer[sr1] + R_buffer[
                                sr2];
191                           result <= R_buffer[sr1] + R_buffer[
                                sr2];
192                       end
193                       curr_state <= next_state;
194                   end
195                 AND: begin // And: DR = SR1 & SR2 or DR = SR1 &
                        imm5
196                       if (IR[5]) begin
197                           R[dr]  <= R_buffer[sr1] & imm5;
198                           result <= R_buffer[sr1] & imm5;
199                       end else begin
200                           R[dr]  <= R_buffer[sr1] & R_buffer[
```

```
                        sr2];
201                   result <= R_buffer[sr1] & R_buffer[
                        sr2];
202               end
203               curr_state <= next_state;
204           end
205       BR:  begin // Branch: if CC(condition code) meet
              the
206               // requirement the change PC
207               if (cc & IR[11:9]) PC_buffer <= PC +
                    PCoffset9;
208               curr_state <= next_state;
209           end
210       JMP: begin // Jump: PC = BaseR
211               PC_buffer <= R_buffer[baseR];
212               curr_state <= next_state;
213           end
214       JSR: begin // Jump to SubRoutine: Call a
              subroutine
215               // temp = PC
216               // then PC = BaseR or PC += PCoffset11
217               // then R[7] = temp
218               temp <= PC;
219               if (IR[11]) PC_buffer <= PC + PCoffset11
                    ;
220               else        PC_buffer <= R_buffer[baseR]
                    ;
221               curr_state <= next_state;
222           end
223       LD:  begin // Load: DR = mem[PC + PCoffset9]
224               MAR <= PC + PCoffset9;
225               curr_state <= next_state;
226           end
227       LDI: begin // Load Indirect:
228               // DR = mem[mem[PC + PCoffset9]]
229               MAR <= PC + PCoffset9;
230               curr_state <= next_state;
231           end
232       LDR: begin // Load Base+offset:
233               // DR = mem[BaseR + offset6]
234               MAR <= R_buffer[baseR] + offset6;
235               curr_state <= next_state;
```

```
236                     end
237                 LEA: begin // Load Effective Address:
238                           // DR = PC + PCoffset9
239                         R[dr] <= PC + PCoffset9;
240                         curr_state <= next_state;
241                     end
242                 NOT: begin // Not: DR = ~ SR
243                         R[dr]  <= ~R_buffer[sr1];
244                         result <= ~R_buffer[sr1];
245                         curr_state <= next_state;
246                     end
247                 RTI: begin // Return from Trap or Interrupt
248                           // (Not effctive since we won't use
                              TRAP
249                           // except TRAP x25(HALT))
250                         MAR  <= R_buffer[6];
251                         R[6] <= R_buffer[6] + 16'h0002;
252                         curr_state <= next_state;
253                     end
254                 ST:  begin // Store: mem[PC + PCoffset9] = SR(DR
                        here)
255                           MAR    <= PC + PCoffset9;
256                           MDR_in <= R_buffer[dr];
257                           mem_we <= 1'b1;
258                           curr_state <= next_state;
259                     end
260                 STI: begin // Store Indirect:
261                           // mem[mem[PC + PCoffset9]] = SR(DR
                              here)
262                         MAR    <= PC + PCoffset9;
263                         mem_we <= 1'b0;
264                         curr_state <= write_mem_i;
265                     end
266                 STR: begin // Store Base+offset:
267                           // mem[BaseR + offset6] = SR(DR here)
268                         MAR    <= R_buffer[baseR] + offset6;
269                         MDR_in <= R_buffer[dr];
270                         mem_we <= 1'b1;
271                         curr_state <= next_state;
272                     end
273                 read_mem:   begin
274                                 R[dr]  <= MDR_out;
```

```
275                                 result <= MDR_out;
276                                 curr_state <= next_state;
277                             end
278                 write_mem:  begin
279                                 mem_we <= 1'b0;
280                                 curr_state <= next_state;
281                             end
282                 read_mem_i: begin
283                                 MAR <= MDR_out;
284                                 curr_state <= next_state;
285                             end
286                 write_mem_i:begin
287                                 MAR    <= MDR_out;
288                                 MDR_in <= R_buffer[dr];
289                                 mem_we <= 1'b1;
290                                 curr_state <= next_state;
291                             end
292                 jsr_1:    begin
293                                   R[7] <= temp;
294                                   curr_state <= next_state;
295                           end
296                 rti_1:    begin
297                                   PC_buffer  <= MDR_out;
298                                   curr_state <= next_state;
299                           end
300                 off:      begin
301                                   mem_we <= 1'b0;
302                                   curr_state <= next_state;
303                           end
304                 default: curr_state <= next_state;
305             endcase
306         end
307     end
308     // FSM Part 3 (Hexplay)
309     reg [18:0] pluse_cnt;
310     reg [16:0] data;
311     wire pluse_200hz;
312     assign pluse_200hz = pluse_cnt == 1'b1;
313     always @(posedge clk_100mhz) begin
314         if (pluse_cnt >= 19'h7A120) pluse_cnt <= 19'h00000;
315         else                        pluse_cnt <= pluse_cnt + 19'
            h0001;
```

```
316     end
317     always @(posedge clk_100mhz) begin
318         if (pluse_200hz) hexplay_an <= hexplay_an + 2'b01;
319     end
320     always @(posedge clk_100mhz) begin
321         case(sw)
322             4'h0: data <= R[0];
323             4'h1: data <= R[1];
324             4'h2: data <= R[2];
325             4'h3: data <= R[3];
326             4'h4: data <= R[4];
327             4'h5: data <= R[5];
328             4'h6: data <= R[6];
329             4'h7: data <= R[7];
330             4'h8: data <= {7'h00, cc[2], 3'h0, cc[1], 3'h0, cc[0]
                    };
331             4'h9: data <= PC;
332             4'hA: data <= IR;
333             4'hB: data <= MAR;
334             4'hC: data <= MDR_out;
335             default: data <= 16'h0000;
336         endcase
337     end
338     always @(posedge clk_100mhz) begin
339         case(hexplay_an)
340             2'b00:   hexplay_data <= data[3:0];
341             2'b01:   hexplay_data <= data[7:4];
342             2'b10:   hexplay_data <= data[11:8];
343             2'b11:   hexplay_data <= data[15:12];
344             default: hexplay_data <= 4'h0;
345         endcase
346     end
347 endmodule
```

代码 8: 约束文件

```
1  ## LC-3.srcs/constrs_1/new/fpga.xdc
2
3  ## Clock signal
4  set_property -dict { PACKAGE_PIN E3    IOSTANDARD LVCMOS33 } [get_ports { clk_100mhz }];
5
6  ## FPGAOL SWITCH
7
8  set_property -dict { PACKAGE_PIN D14   IOSTANDARD LVCMOS33 } [get_ports { sw[0] }];
9  set_property -dict { PACKAGE_PIN F16   IOSTANDARD LVCMOS33 } [get_ports { sw[1] }];
10 set_property -dict { PACKAGE_PIN G16   IOSTANDARD LVCMOS33 } [get_ports { sw[2] }];
```

```
11 set_property -dict { PACKAGE_PIN H14   IOSTANDARD LVCMOS33 } [get_ports { sw[3] }];
12
13 ## FPGAOL HEXPLAY
14
15 set_property -dict { PACKAGE_PIN A14   IOSTANDARD LVCMOS33 } [get_ports { hexplay_data[0]
       }];
16 set_property -dict { PACKAGE_PIN A13   IOSTANDARD LVCMOS33 } [get_ports { hexplay_data[1]
       }];
17 set_property -dict { PACKAGE_PIN A16   IOSTANDARD LVCMOS33 } [get_ports { hexplay_data[2]
       }];
18 set_property -dict { PACKAGE_PIN A15   IOSTANDARD LVCMOS33 } [get_ports { hexplay_data[3]
       }];
19 set_property -dict { PACKAGE_PIN B17   IOSTANDARD LVCMOS33 } [get_ports { hexplay_an[0] }];
20 set_property -dict { PACKAGE_PIN B16   IOSTANDARD LVCMOS33 } [get_ports { hexplay_an[1] }];
21
22 ## FPGAOL BUTTON & SOFT_CLOCK
23
24 set_property -dict { PACKAGE_PIN B18   IOSTANDARD LVCMOS33 } [get_ports { btn }];
```

【总结与思考】

收获 了解了如何设计一个较为复杂的系统.

难易程度 中等

任务量 较多

最后在此感谢老师和助教的付出, 希望这门课能越办越好!