# 数字电路与数字系统实验

**EX11:字符输入界面**

191220029 傅小龙

周一5-6节班

[1830970417@qq.com](mailto:1830970417@qq.com)

2020年11月29日

# **目录**

1. [实验内容](#_实验内容)……………………………………………………………………………………3
   1. [实验要求](#_1.1实验要求)………………………………………………………………………………3
   2. [实验工具](#_1.2实验工具)………………………………………………………………………………3
2. [实验过程](#_实验过程)……………………………………………………………………………………4

2.1 [模型概述](#_2.2.1模型概述)………………………………………………………………………………4

2.2 [数字抽象](#_2.2.2数字抽象)………………………………………………………………………………4

2.3 [建立模型](#_2.2.3建立模型)………………………………………………………………………………4

2.3.1 vga\_char模块………………………………………………………………………4

2.3.2 vga\_ctrl模块………………………………………………………………………12

2.3.3 keyboard模块………………………………………………………………………12

2.3.4 kbdecoder模块……………………………………………………………………13

2.4 [分析/综合](#_2.4分析/综合)…………………………………………………………………………… 15

2.5 [分配引脚](#_2.6分配引脚)………………………………………………………………………………16

2.6 [全编译](#_2.7全编译)…………………………………………………………………………………17

三、[实验总结](#_实验总结)……………………………………………………………………………………18

四、附……………………………………………………………………………………………19

#### 实验内容

##### 1.1实验要求

实现一个可以用键盘输入，并在 VGA 显示器上回显的交互界面。界面实

现要求可以参考 DOS 字符界面，Window 命令行或 Linux 的字符终端。

基本要求

·支持所有小写英文字母和数字输入，以及不用 Shift 即可输入的符号。

·一直按压某个键时，重复输出该字符。

·输入至行尾后自动换行，输入回车也换行。

可选扩展要求

·可以显示光标，建议可以用显示闪烁的竖线或横线作为光标。

·支持 BackSpace 键删除光标前的字符。

·BackSpace 删除至本行开始后，再按 BackSpace 可以删除回车键，光标停 留在上一行末尾的非空字符后。

·支持自动滚屏，即输入到最后一行后回车出现新空白行，并且所有已输入

的行自动上移一行。

·支持 Shift 键以及大小写字符输入。

·支持方向键移动光标。

·在行首显示命令提示符。

感兴趣的同学还可以考虑如何实现彩色字符、绘制 ASCII 艺术图或实现类

似 Matrix 开头的字符雨效果。

##### 1.2实验工具

软件环境：

设计、编译、仿真：Quartus Prime Version 17.1.0 Build 590 10/25/2017 SJ Lite Edition

DE10\_Standard\_SystemBuilder

硬件环境： DE-10 Standard开发平台

FPGA芯片： Cyclone V 5CSXFC6D6F31C6

#### 实验过程

##### 2.1模型概述

在前面实验中已经实现的键盘、VGA控制器模块的基础上，使用ROM存储器存储字模点阵，使用RAM作为显存存储键盘已输入对应字符的点阵. 可以用mif文件对显存初始化以显示ASCII字符图.

本次实验中实现的功能包括1.1节中的所有基础要求以及可选扩展要求中的：①可以显示光标，建议可以用显示闪烁的竖线或横线作为光标; ②支持 BackSpace 键删除光标前的字符; ③BackSpace 删除至本行开始后，再按BackSpace 可以删除回车键，光标停留在上一行末尾的非空字符后; ④支持 Shift 键以及大小写字符输入; ⑤支持方向键移动光标。

##### 2.2数字抽象

下图给出了各模块间的关系及其作用, 其中exp11ch为顶层文件名.

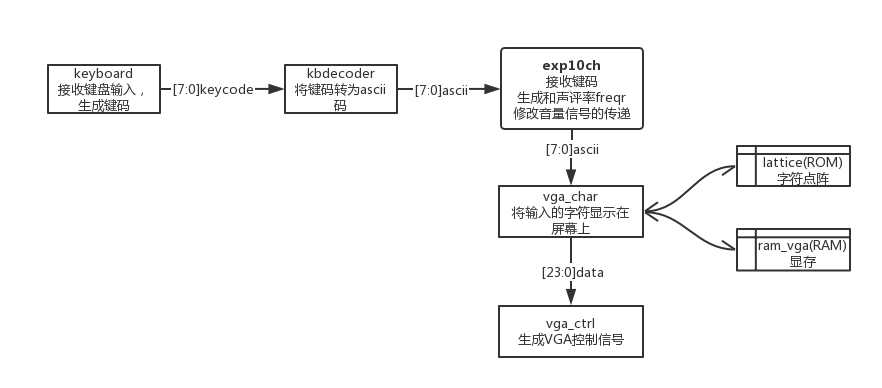


图2-2-1 模块关系示意图

##### 2.3建立模型

###### 2.3.1 vga\_char模块

vga\_char模块负责从只读存储器lattice中读取键盘输入对应的字符点阵并将之写入显存RAM以及光标闪烁色块的生成.

I)存储器的设置

由于每个字符的点阵大小为16x9像素点，共有128个字符，存放点阵的只读存储器lattice的规模设置为4096x9bits, 单口读, 使用IP核生成，如图2-3-1所示. 对于分辨率为640x480像素的屏幕，可以显示30行70列的字符，每个单元存储ASCII码，故显存RAM的规模为2100x8bits, 使用双口读写，读写采用不同的时钟，使用IP核生成. 如图2-3-2(a)(b)所示.

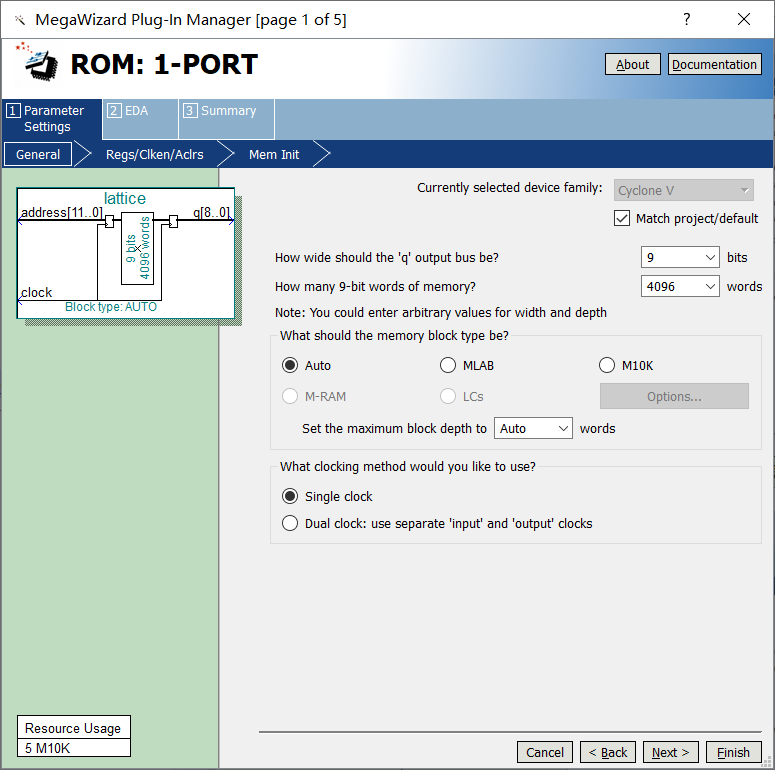


图2-3-1 lattice的设置

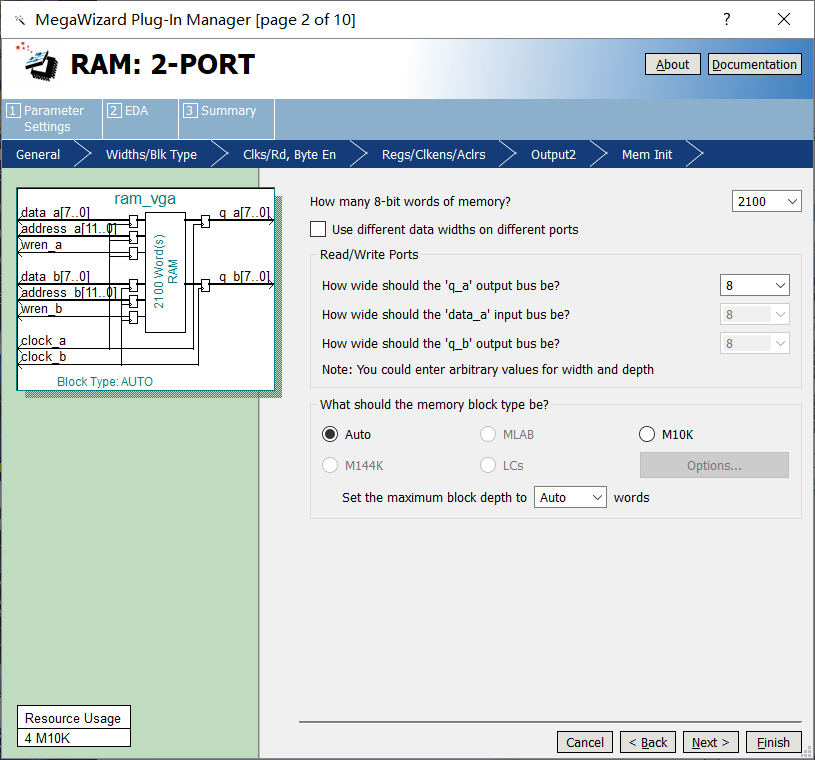


图2-3-2（a） ram\_vga的设置

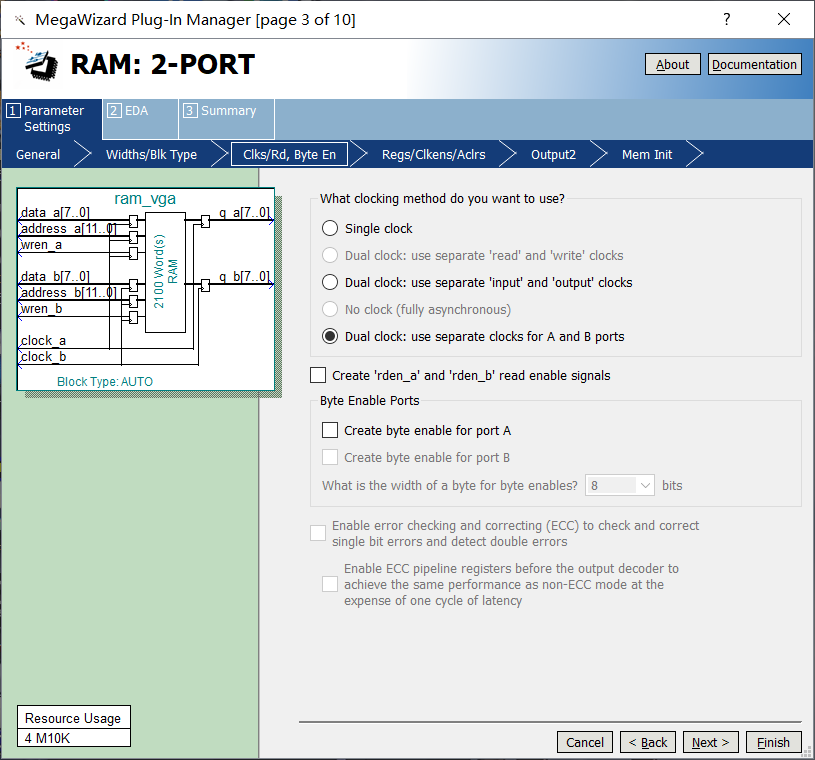


图2-3-2（b） ram\_vga的设置

II)将键盘输入写入显存

使用频率较低的时钟（40Hz）来处理由顶层文件传入的ascii码. 通过reg型变量[3:0]cnt来对该部分经过的40Hz低频时钟周期数进行计数, reg型变量[11:0]index用于记录当前输入的位置（对应屏幕上显示的第index个位置的ascii字符）, reg型变量[11:0]ram\_index[29:0]用于记录每一行（共30行）最后一个有效字符的位置, 以便在删除字符时能够使index能够回到正确的位置.

有按键按下时cnt每经过一个40Hz时钟周期将会加一, 达到4’d15时（0.375s）认为该键长按，将会持续对显存写入被按下的键码对应的ascii码. 若输入的ascii码为0则为无效操作，cnt置0. 对于功能型按键的输入不应该写入显存, 比如说Caps\_Lock键、Backspace键等, 故在遇到此类输入时需要将ram\_vga的写使能wen置0, 遇到有实际显示意义的字符再将wen置1. 考虑到器件可能存在的时序上的延迟, ram\_vga的写使能设置采用阻塞式赋值.

由上所述，ram\_vga的写地址为index, 写时钟为应为40Hz的时钟信号, 写使能为wen. 普通字符的输入需要将index加1，并设置写使能为1.

1 //add one char

2 wen = 1;

3 index <= index + 1;

若当前行输入已至行尾，则需要记录当前行的有效字符结束位置对应的index.

1 if((index + 1) % 70 == 0)begin

2 //current line is to be full

3 ram\_index[(index - 1) / 70] <= index;

4 end

若输入回车，则需要将index移动至下一行的起始位置并记录旧行的有效字符的结束位置.

1 if(ascii == 8'h0d)begin//hit enter

2 backspace\_state <= 0;

3 //store this line's ending position

4 ram\_index[index/70] <= index + 1;

5 //move cursur to next line

6 index <= index + 70 - (index % 70);

7 enter\_state <= 0;

8 end

若输入退格, 则需要将index减1以起到删除字符的效果, 当index退到行首时，需要将将index回退至ram\_index中存储的上一行行尾位置对应的index值.

1 if (ascii == 8'h08) begin//hit backspace

2 if(backspace\_state == 0)begin

3 index <= index - 1;

4 backspace\_state <= 1'b1;

5 end

6 else begin //backspace\_state == 1'b0

7 if(index % 70 == 0)begin

8 //return to last line's end

9 index <= preindex;

10 end

11 else begin

12 //delete one char

13 index <= index - 1;

14 end

15 end

16 end

其中, preindex为ram\_index中对应当前index上一行的行末位置.其取值如下：

1 always @(posedge clk\_keyboard)begin

2 if(index < 12'd70)begin

3 preindex <= 0;

4 end

5 else begin

6 preindex <= ram\_index[(index/70)-1];

7 end

8 end

若输入方向键，则将写使能置0并按相应方向修改index的值. ←键将index减1, →键将index加1, ↓键将index加70并记录旧行的行尾位置. ↑键将index减去70. 为保证index的值始终在屏幕显示区域内, 即index的值应满足 index < 2100. 故当index大于2100时将index置0, 故当输入到当前页面末尾后继续输入, 光标将跳转置页面起始位置覆盖之前写的位置.

综合上面所述, 给出将ascii码写入显存的对应Verilog语言实现如下:

always @(posedge clk\_keyboard) begin

if(ascii != 0)

begin

if(cnt == 0 ) //short hit

begin

if(ascii == 8'h0d)begin//hit enter

backspace\_state <= 0;

//store this line's ending position

ram\_index[index/70] <= index + 1;

//move cursur to next line

index <= index + 70 - (index % 70);

//index <= index + 70 - (index % 70) + 1;

enter\_state <= 0;

end

else if (ascii == 8'h08) begin//hit backspace

if(backspace\_state == 0)begin

index <= index - 1;

backspace\_state <= 1'b1;

end

else begin //backspace\_state == 1'b0

if(index % 70 == 0)begin

//return to last line's end

index <= preindex;

end

else begin

//delete one char

index <= index - 1;

end

end

end

else begin//hit key

if(backspace\_state == 1 || enter\_state == 0)

index <= index; //deleting or changing line avoid writing ram

else begin

if((index + 1) % 70 == 0)begin

//current line is to be full

ram\_index[(index - 1) / 70] <= index;

end begin

if(ascii == 8'h38) begin

//move cursur to last line

index <= index - 70;

wen = 0;

end

else if(ascii == 8'h32) begin

//move cursur to next line

//store this line's ending position

ram\_index[index / 70] <= index + 1;

//backspace\_state <= 1;

index <= index + 70;

wen = 0;

end

else if(ascii == 8'h34) begin

//left move cursur

index <= index - 1;

wen = 0;

end

else if(ascii == 8'h36) begin

//right move cursur

index <= index + 1;

wen = 0;

end

else if(ascii == 8'h14)begin

//skip caps\_lock

wen = 0;

index <= index;

end

else begin

//add one char

wen = 1;

index <= index + 1;

end

end

end

backspace\_state <= 0;

enter\_state <= 1;

end

cnt <= cnt + 1;

end

else if(cnt == 4'd15)begin //long press

if(ascii == 8'h0d) //enter

index <= index + 70 - (index % 70) - 1;

else if (ascii == 8'h08)begin //backspace

if(index % 70 == 0)begin

//this line has nothing left, ret to last line

index <= preindex;

end

else begin

//delete one char

index <= index - 1;

end

end

else begin

if((index + 1) % 70 == 0)begin //end of line

//store this line's ending position

ram\_index[(index-1)/70] <= index;

end

if(ascii == 8'h38) begin

//move cursur to last line

index <= index - 70;

wen = 0;

end

else if(ascii == 8'h32) begin

//move cursur to next line

//store this line's position

ram\_index[index/70] <= index + 1;

//backspace\_state <= 1;

index <= index + 70;

wen = 0;

end

else if(ascii == 8'h34) begin

//left move cursur

index <= index - 1;

wen = 0;

end

else if(ascii == 8'h36) begin

//right move cursur

index <= index + 1;

wen = 0;

end

else if(ascii == 8'h14)begin

//skip caps\_lock

wen = 0;

index <= index;

end

else begin

//add one char

wen = 1;

index <= index + 1;

end

end

end

else begin //waiting if long press

index <= index;

cnt <= cnt + 1;

end

ascii\_data <= ascii;

end

else begin //reset cnt

index <= index;

cnt <= 0;

end

if(index >= 2100) //return to the start of the page if reaching the end

index <= 0;

end

|  |  |
| --- | --- |
| 1 |  |
| 2 |  |
| 3 |  |
| 4 |  |
| 5 |  |
| 6  if(ascii == 8'h0d)begin//hit enter  wen = 0;  backspace\_state <= 0;  //store this line's ending position  ram\_index[index/70] <= index + 1;  //move cursur to next line  index <= index + 70 - (index % 70);  enter\_state <= 0;  end  else if (ascii == 8'h08) begin//hit backspace  wen = 0;  if(index % 70 == 0)begin  //return to last line's end  index <= preindex;  end  else begin  //delete one char  index <= index - 1;  end  end  end  else begin//hit key  if((index + 1) % 70 == 0)begin  //current line is to be full  ram\_index[(index - 1) / 70] <= index;  end begin  if(ascii == 8'h38) begin  //move cursur to last line  index <= index - 70;  wen = 0;  end  else if(ascii == 8'h32) begin  //move cursur to next line  //store this line's ending position  ram\_index[index / 70] <= index + 1;  index <= index + 70;  wen = 0;  end  else if(ascii == 8'h34) begin  //left move cursur  index <= index - 1;  wen = 0;  end  else if(ascii == 8'h36) begin  //right move cursur  index <= index + 1;  wen = 0;  end  else if(ascii == 8'h14)begin  //skip caps\_lock  wen = 0;  index <= index;  end  else begin  //add one char  wen = 1;  index <= index + 1;  end  end  end  backspace\_state <= 0;  enter\_state <= 1;  end  cnt <= cnt + 1;  end  else if(cnt == 4'd15)begin //long press  if(ascii == 8'h0d) //enter  index <= index + 70 - (index % 70) - 1;  else if (ascii == 8'h08)begin //backspace  if(index % 70 == 0)begin  //this line has nothing left, ret to last line  index <= preindex;  end  else begin  //delete one char  index <= index - 1;  end  end  else begin  if((index + 1) % 70 == 0)begin //end of line  //store this line's ending position  ram\_index[(index-1)/70] <= index;  end  if(ascii == 8'h38) begin  //move cursur to last line  index <= index - 70;  wen = 0;  end  else if(ascii == 8'h32) begin  //move cursur to next line  //store this line's position  ram\_index[index/70] <= index + 1;  //backspace\_state <= 1;  index <= index + 70;  wen = 0;  end  else if(ascii == 8'h34) begin  //left move cursur  index <= index - 1;  wen = 0;  end  else if(ascii == 8'h36) begin  //right move cursur  index <= index + 1;  wen = 0;  end  else if(ascii == 8'h14)begin  //skip caps\_lock  wen = 0;  index <= index;  end  else begin  //add one char  wen = 1;  index <= index + 1;  end  end  end  else begin //waiting if long press  index <= index;  cnt <= cnt + 1;  end  ascii\_data <= ascii;  end    else begin //reset cnt  index <= index;  cnt <= 0;  end  if(index >= 2100) //return to the start of the page if reaching the end  index <= 0;  end |  |
| 7 |  |
| 8 |  |
| 9 |  |
| 10 |  |
| 11 |  |
| 12 |  |
| 13 |  |
| 14 |  |
| 15 |  |
| 16 |  |
| 17 |  |
| 18 |  |
| 19 |  |
| 20 |  |
| 21 |  |
| 22 |  |
| 23 |  |
| 24 |  |
| 25 |  |
| 26 |  |
| 27 |  |
| 28 |  |
| 29 |  |
| 30 |  |
| 31 |  |
| 32 |  |
| 33 |  |
| 34 |  |
| 35 |  |
| 36 |  |
| 37 |  |
| 40 |  |
| 41 |  |
| 42 |  |
| 43 |  |
| 44 |  |
| 45 |  |
| 46 |  |
| 47 |  |
| 48 |  |
| 49 |  |
| 50 |  |
| 51 |  |
| 52 | //right move cursur  index <= index + 1;  wen = 0;  end  else if(ascii == 8'h14)begin  //skip caps\_lock  wen = 0;  index <= index;  end  else begin  //add one char  wen = 1;  index <= index + 1;  end  end  end  backspace\_state <= 0;  enter\_state <= 1;  end  cnt <= cnt + 1;  end  else if(cnt == 4'd15)begin //long press  if(ascii == 8'h0d) //enter  index <= index + 70 - (index % 70) - 1;  else if (ascii == 8'h08)begin //backspace  if(index % 70 == 0)begin  //this line has nothing left, ret to last line  index <= preindex;  end  else begin  //delete one char  index <= index - 1;  end  end  else begin  if((index + 1) % 70 == 0)begin //end of line  //store this line's ending position  ram\_index[(index-1)/70] <= index;  end  if(ascii == 8'h38) begin  //move cursur to last line  index <= index - 70;  wen = 0;  end  else if(ascii == 8'h32) begin  //move cursur to next line  //store this line's position  ram\_index[index/70] <= index + 1;  //backspace\_state <= 1;  index <= index + 70;  wen = 0;  end  else if(ascii == 8'h34) begin  //left move cursur  index <= index - 1;  wen = 0;  end  else if(ascii == 8'h36) begin  //right move cursur  index <= index + 1;  wen = 0;  end  else if(ascii == 8'h14)begin  //skip caps\_lock  wen = 0;  index <= index;  end  else begin  //add one char  wen = 1;  index <= index + 1;  end  end  end  else begin //waiting if long press  index <= index;  cnt <= cnt + 1;  end  ascii\_data <= ascii;  end    else begin //reset cnt  index <= index;  cnt <= 0;  end  if(index >= 2100) //return to the start of the page if reaching the end  index <= 0;  end |
| 53 |  |
| 54 |  |
| 55 |  |
| 56 |  |
| 57 |  |
| 58 |  |
| 59 |  |
| 60 |  |
| 61 |  |
| 62 |  |
| 63 |  |
| 64 |  |
| 65 |  |
| 66 |  |
| 67 |  |
| 68 |  |
| 69 |  |
| 70 |  |
| 71 |  |
| 72 |  |
| 73 |  |
| 74 |  |
| 75 |  |
| 76 |  |
| 77 |  |
| 78 |  |
| 79 |  |
| 80 |  |
| 81 |  |
| 82 |  |
| 83 |  |
| 84 |  |
| 85 |  |
| 86 |  |
| 87 |  |
| 88 |  |
| 89 |  |
| 90 |  |
| 91 |  |
| 92 |  |
| 93 |  |
| 94 |  |
| 95 |  |
| 96  else if(ascii == 8'h32) begin  //move cursur to next line  //store this line's position  ram\_index[index/70] <= index + 1;  index <= index + 70;  wen = 0;  end  else if(ascii == 8'h34) begin  //left move cursur  index <= index - 1;  wen = 0;  end  else if(ascii == 8'h36) begin  //right move cursur  index <= index + 1;  wen = 0;  end  else if(ascii == 8'h14)begin  //skip caps\_lock  wen = 0;  index <= index;  end  else begin  //add one char  wen = 1;  index <= index + 1;  end  end  end  else begin //waiting if long press  index <= index;  cnt <= cnt + 1;  end  ascii\_data <= ascii;  end  else begin //reset cnt  index <= index;  cnt <= 0;  end  if(index >= 2100) //return to the start of the page if reaching the end  index <= 0;  end |  |
| 97 |  |
| 98 |  |
| 99 |  |
| 100 |  |
| 101 |  |
| 102 |  |
| 103 |  |
| 104 |  |
| 105 |  |
| 106 |  |
| 107 |  |
| 108 |  |
| 109 |  |
| 110 |  |
| 111 |  |
| 112 |  |
| 113 |  |
| 114 |  |
| 115 |  |
| 116 |  |
| 117 |  |
| 118 |  |
| 119 |  |
| 120 |  |
| 121 |  |
| 122 |  |
| 123 |  |
| 124 |  |
| 125 |  |
| 126 |  |
| 127 |  |
| 128 |  |
| 129 |  |
| 130 |  |
| 131 |  |
| 132 |  |
| 133 |  |
| 134 |  |
| 135 |  |
| 136 |  |
| 137 |  |
| 138 |  |

III)从显存中读取已输入的字符并将之转化为点阵以及光标的显示

根据vga\_ctrl模块提供的当前扫描点坐标([9:0]h\_addr, [9:0]v\_addr)确定正在扫描的字符位置[11:0]block\_addr. 由前面所述的字符点阵大小可知这三者的关系满足

block\_addr <= (v\_addr >> 4) \* 70 + ((h\_addr) / 9);

[11:0]block\_addr即为ram\_vga的读地址. ram\_vga的读输出由[7:0]ram\_vga\_ret接收，[7:0]ram\_vga\_ret即为当前扫描位置的字符的ascii码. 由该ascii码可算出该字符对应点阵在lattice中的（读）地址[11:0]address应为

address <= (ram\_vga\_ret << 4) + (v\_addr % 16);

lattice的读输出由[8:0]font接收, font即为该字符在当前扫描行上的数据. 对应到扫描点上的数据应为font[(h\_addr + 2) % 9]. ‘+2’是对扫描位置的修正(否则将会造成最后一个字符的显示不完整). 若该扫描点对应的点阵数据为0, 则传给vga\_ctrl的RGB参数[23:0]data为24'h000000(黑色), 否则为全1(白色).

光标的显示位置在当前输入位置index的下一格. 这里采用的显示区域为

v\_addr > (index / 70) \* 16 && v\_addr < (1 + index / 70) \* 16 && h\_addr > (index % 70 + 1) \* 9 + 5 && h\_addr < (index % 70 + 2

需要注意的是若光标的显示区域过宽可能会遮挡住已经输入的字符. 光标的闪烁由一个低频时钟控制(5Hz).

综合上面所述, 显存读取部分以及光标的显示的Verilog实现如下:

always @(posedge clk\_vga)begin

block\_addr <= (v\_addr >> 4) \* 70 + ((h\_addr) / 9);

address <= (ram\_vga\_ret << 4) + (v\_addr % 16);

//show the cursor

//cursor is at the end of current line

if(v\_addr > (index / 70) \* 16 && v\_addr < (1 + index / 70) \* 16

&& h\_addr > (index % 70 + 1) \* 9 + 5 && h\_addr < (index % 70 + 2) \* 9)

if(clk\_cursur) data <= 24'hffffff;

else data <= 24'h000000;

//show the font

else if(font[(h\_addr + 2) % 9] == 1'b1)

data <= 24'hffffff;

else

data <= 24'h000000;

//current line: (index - 1)/70

//cursur pos: (index - 1) % 70

end

|  |  |
| --- | --- |
| 1 |  |
| 2 |  |
| 3 |  |
| 4 |  |
| 5 |  |
| 6 |  |
| 7 |  |
| 8 |  |
| 9 |  |
| 10 |  |
| 11 |  |
| 12 |  |
| 13 |  |
| 14 |  |
| 15 |  |
| 16 |  |
| 17 |  |
| 18 |  |
| 19 |  |
| 20 |  |
| 21 |  |
| 22 |  |

以上提到的任意频率的时钟为显示器实验中所使用的clkgen模块, 作用是将50MHz时钟信号转化为设置的频率. 其Verilog实现如下：

|  |  |
| --- | --- |
| 1 | module clkgen(  input clkin,  input rst,  input clken,  output reg clkout  );  parameter clk\_freq = 1000;  parameter countlimit = 50000000/2/clk\_freq; // 自 动 计 算 计 数 次 数  reg[31:0] clkcount;  always @ (posedge clkin)  if(rst) begin  clkcount=0;  clkout=1'b0;  end  else begin  if(clken)  begin  clkcount = clkcount + 1;  if(clkcount >= countlimit) begin  clkcount = 32'd0;  clkout = ~clkout;  end  else  clkout=clkout;  end  else begin  clkcount=clkcount;  clkout=clkout;  end  end  endmodule |
| 2 |  |
| 3 |  |
| 4 |  |
| 5 |  |
| 6 |  |
| 7 |  |
| 8 |  |
| 9 |  |
| 10 |  |
| 11 |  |
| 12 |  |
| 13 |  |
| 14 |  |
| 15 |  |
| 16 |  |
| 17 |  |
| 18 |  |
| 19 |  |
| 20 |  |
| 21 |  |
| 22 |  |
| 23 |  |
| 24 |  |
| 25 |  |
| 26 |  |
| 27 |  |
| 28 |  |
| 29 |  |
| 30 |  |
| 31 |  |
| 32 |  |

###### 2.3.2 vga\_ctrl模块

vga\_ctrl模块采用了ex09 显示器实验中对VGA控制器的参考实现, 故这里不再赘述. 主要功能是将RGB数据转化为VGA控制信号输出至显示器.

###### 2.3.3 keyboard模块

keyborad模块采用了ex08 PS2键盘实验中给出的参考实现, 故这里不再赘述. 主要功能是接收PS2键盘输入并将键码输出.

###### 2.3.4 kbdecoder模块

kbdecoder模块中，相应键码的ASCII码用类似ROM的形式存放，与键码是一一对应关系. 再根据shift, capslock键的状态对ASCII码的输出进行赋值.

下面是kbdecoder模块的相关代码：

module kbdecoder(datain, dataout, shift\_state, caps\_state, ctrl\_state);

input [7:0]datain;

input shift\_state; //shift键是否被按下

input caps\_state; //caps键是否被按下

input ctrl\_state; //ctrl键是否被按下

output reg [7:0]dataout;

reg [7:0] asc [255:0];

reg [7:0] ascii\_shift [255:0];

reg [7:0] ascii\_caps [255:0];

reg [7:0] temp;

initial

begin

$readmemh(".\\init\\ascii\_init.txt", asc, 0, 255);

$readmemh(".\\init\\ascii\_init\_shift.txt", ascii\_shift, 0, 255);

$readmemh(".\\init\\ascii\_init\_caps.txt", ascii\_caps, 0, 255);

end

always @(\*)

begin

if(shift\_state && !caps\_state) begin

dataout <= ascii\_shift[datain];

end

else if(caps\_state && !shift\_state) begin

dataout <= ascii\_caps[datain];

end

else dataout <= asc[datain];

end

endmodule

|  |  |
| --- | --- |
| 1 |  |
| 2 |  |
| 3 |  |
| 4 |  |
| 5 |  |
| 6 |  |
| 7 |  |
| 8 |  |
| 9 |  |
| 10 |  |
| 11 |  |
| 12 |  |
| 13 |  |
| 14 |  |
| 15 | begin  $readmemh(".\\init\\ascii\_init.txt", asc, 0, 255);  $readmemh(".\\init\\ascii\_init\_shift.txt", ascii\_shift, 0, 255);  $readmemh(".\\init\\ascii\_init\_caps.txt", ascii\_caps, 0, 255);    end    always @(\*)  begin  if(shift\_state && !caps\_state) begin  dataout <= ascii\_shift[datain];  end  else if(caps\_state && !shift\_state) begin  dataout <= ascii\_caps[datain];  end  else dataout <= asc[datain];  end  endmodule |
| 16 |  |
| 17 |  |
| 18 |  |
| 19 |  |
| 20 |  |
| 21 |  |
| 22 |  |
| 23 |  |
| 24 |  |
| 25 |  |
| 26 |  |
| 27 |  |
| 28 |  |
| 29 |  |
| 30 |  |
| 31 |  |
| 32 |  |
| 33 |  |
| 34 |  |
| 35 |  |

初始化文件详见项目文件夹的init/文件夹中的.txt文件.

对于shift, capslock按键的状态设计和next\_data\_n信号的设置在顶层文件中给出如下实现：

|  |  |
| --- | --- |
| 1 | always@(posedge CLOCK\_50) begin  if(ready == 1 && next\_data\_n == 1)begin  temp <= keycode;  next\_data\_n <= 0;  if(keycode == 8'hf0) begin//realse  release\_flag <= 1;  cnt <= cnt + 1;  end  else if(keycode == 8'h12 || keycode == 8'h59) begin//shift  if(release\_flag) begin  shift\_state <= 0;  e\_out <= 0;  release\_flag <= 0;  end  else begin  shift\_state <= 1;  e\_out <= 1;  end  end    ctrl\_state <= 0;  e\_out <= 0;  release\_flag <= 0;  end  else begin  ctrl\_state <= 1;  e\_out <= 1;  end  end  else begin  if(release\_flag) begin  e\_out <= 0;  release\_flag <= 0;  if(keycode == 8'h58) caps\_state <= ~caps\_state;  else;  end  else e\_out <= 1;  end  end  else begin  next\_data\_n <= 1;  end  end |
| 2 |  |
| 3 |  |
| 4 |  |
| 5 |  |
| 6 |  |
| 7 |  |
| 8 |  |
| 9 |  |
| 10 |  |
| 11 |  |
| 12 |  |
| 13 |  |
| 14 |  |
| 15 |  |
| 16 |  |
| 17 |  |
| 18 |  |
| 19 |  |
| 20 |  |
| 21  else if(keycode == 8'h14) begin //ctrl  if(release\_flag) begin  ctrl\_state <= 0;  e\_out <= 0;  release\_flag <= 0;  end  else begin  ctrl\_state <= 1;  e\_out <= 1;  end  end  else begin  if(release\_flag) begin  e\_out <= 0;  release\_flag <= 0;  if(keycode == 8'h58) caps\_state <= ~caps\_state;  else;  end  else e\_out <= 1;  end  end  else begin  next\_data\_n <= 1;  end  end |  |
| 22 |  |
| 23 |  |
| 24 |  |
| 25 |  |
| 26 |  |
| 27 |  |
| 28 |  |
| 29 |  |
| 30 |  |
| 31 |  |
| 32 |  |
| 33 |  |
| 34 |  |
| 35 |  |
| 36 |  |
| 37 |  |
| 38 |  |
| 39 |  |
| 40 |  |
| 41 |  |
| 42 |  |
| 43 |  |
| 44 |  |

需要注意的是keyboard模块给出的键盘码输出并不一定是有效的，只有在ready和next\_data\_n信号都为1时才有效（Line2）.获取有效的键码输出后，需要将next\_data\_n信号置0以准备接受下一个有效信号.

对于shift, ctrl按键状态的设置：若release\_flag为零，则说明是按键按下，将对应的标志信号置1，否则置0并设置数码管的使能信号为0(Line 9-31).

对于caps\_lock按键状态的设置：caps\_lock键按下、松开后caps\_lock按键的状态改变一次（Line 33-38）.

##### 2.4分析/综合

分析/综合实验成功，如下图所示：

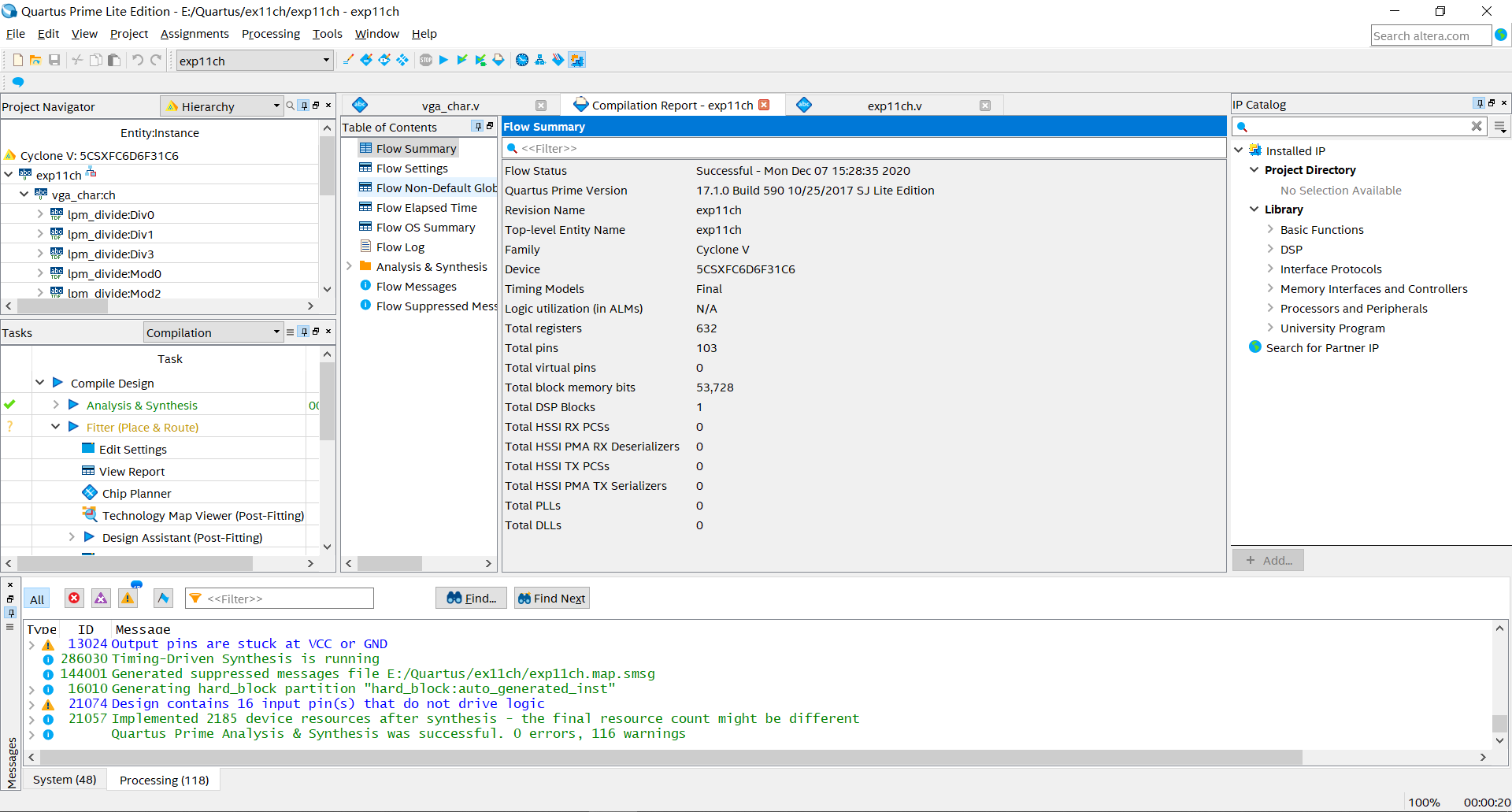


图2-4-1：分析/综合成功

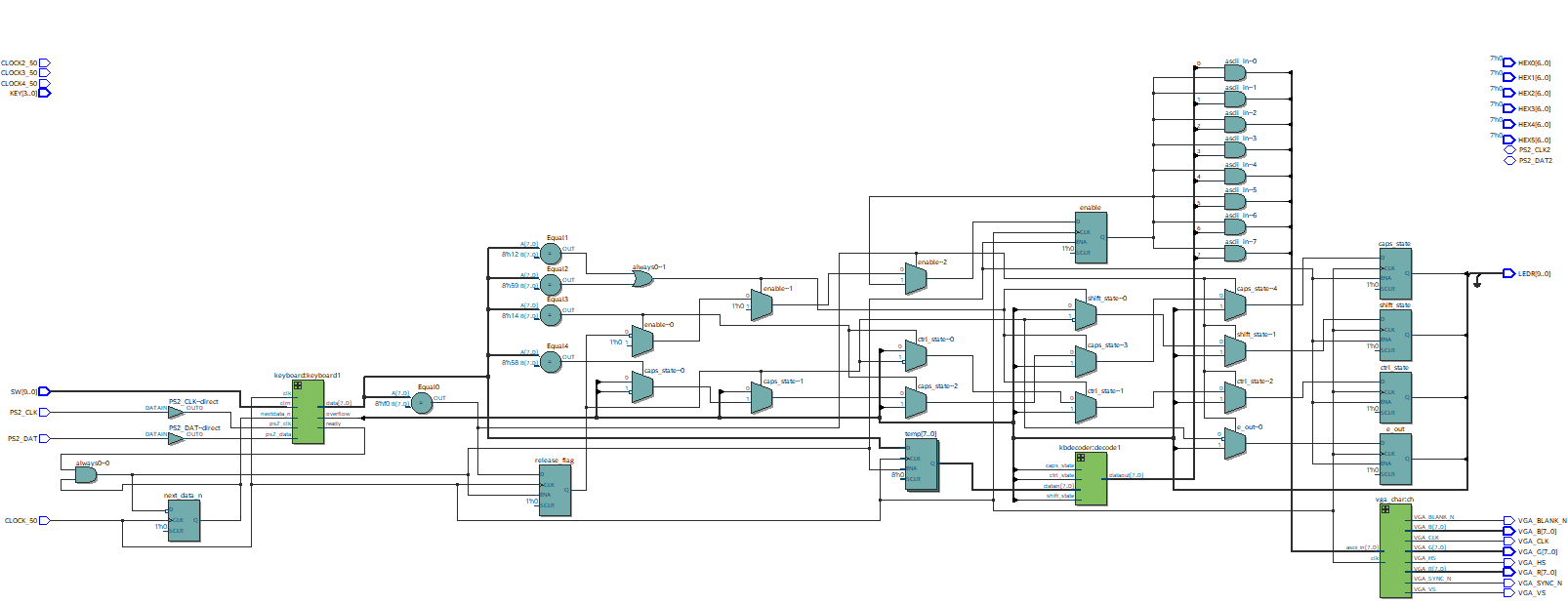
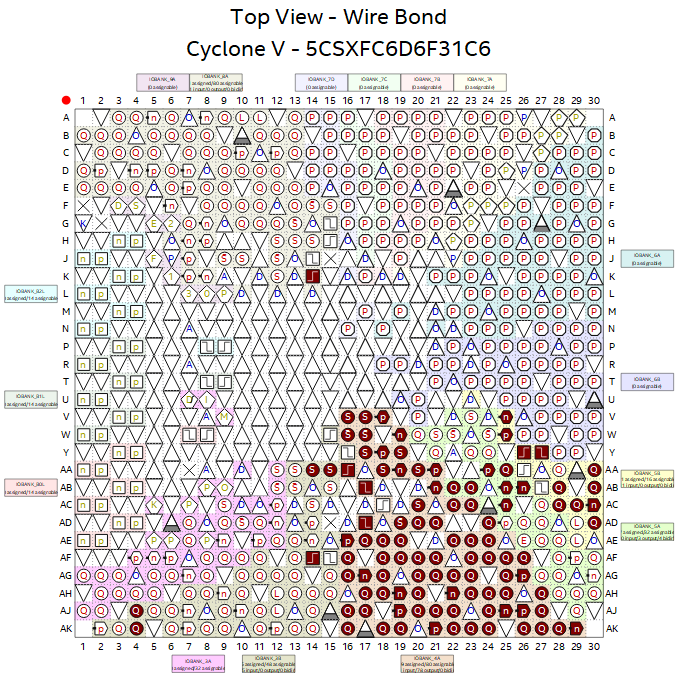
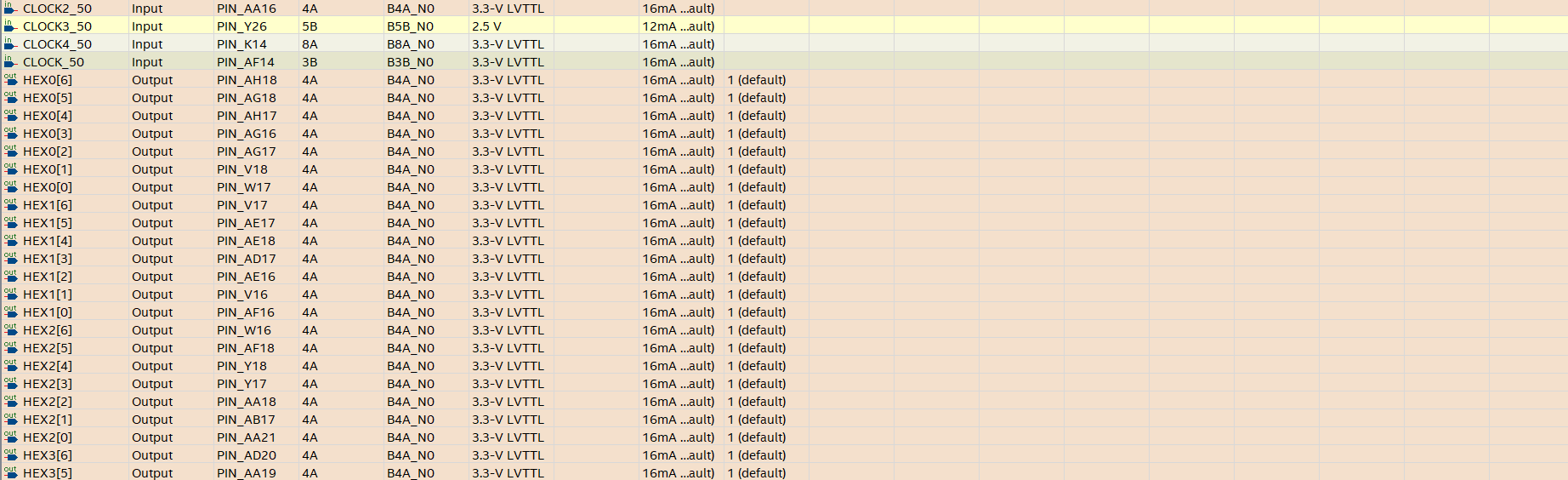


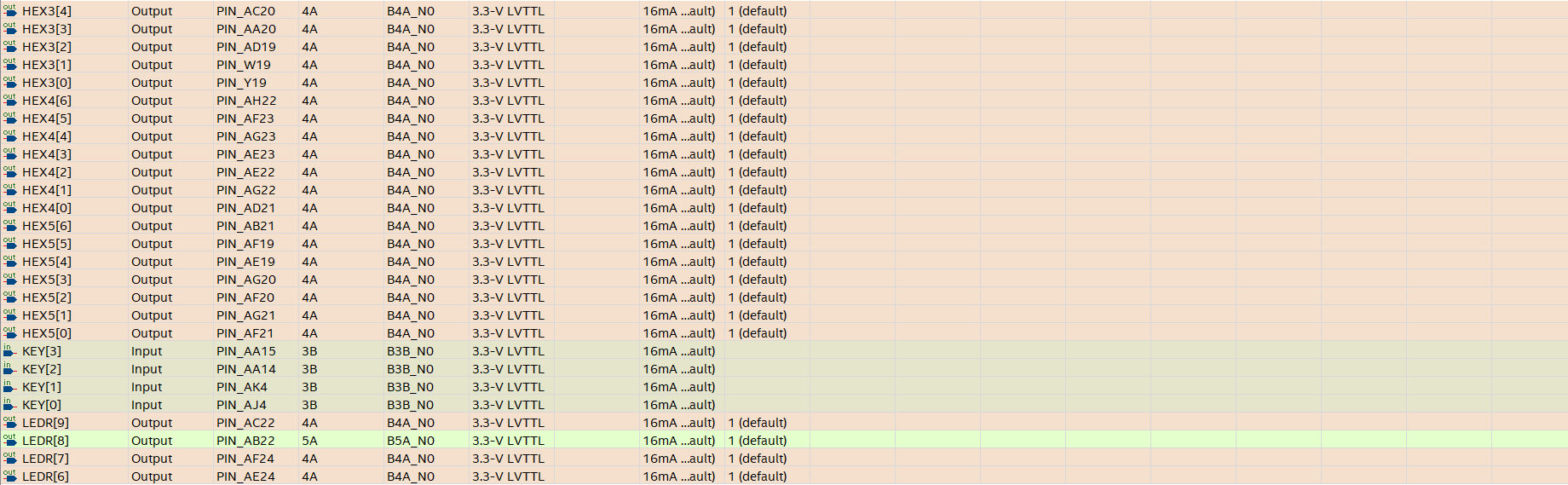
图 2-4-2:RTL视图

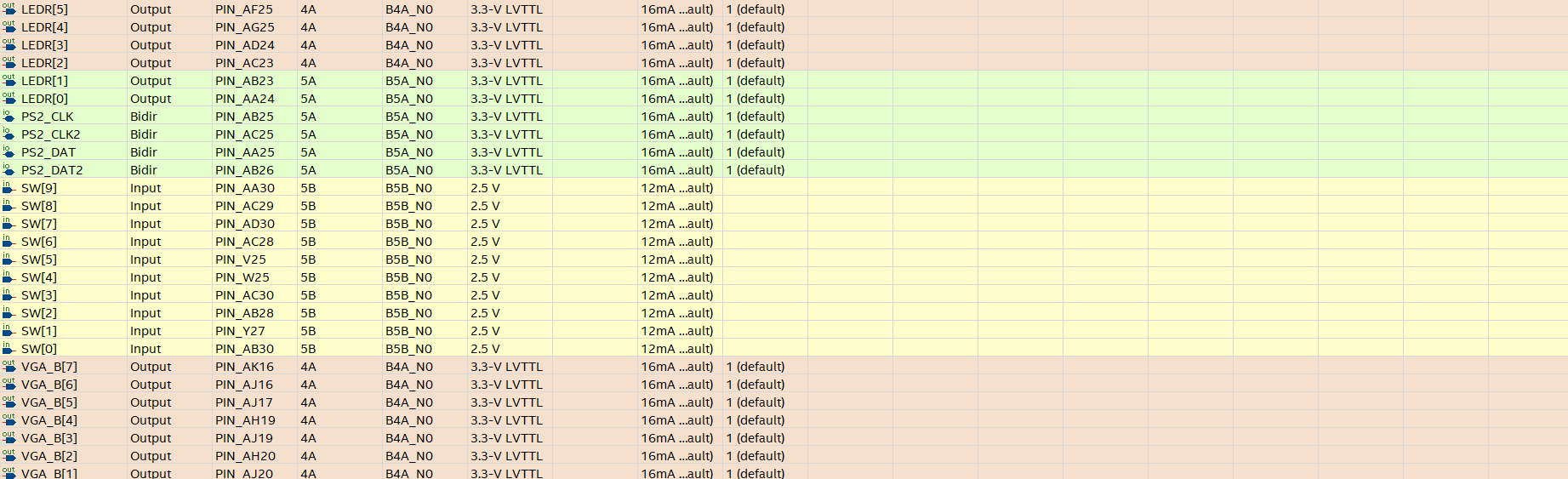
##### 2.5分配引脚

引脚分配使用DE10\_Standard\_SystemBuilder生成。









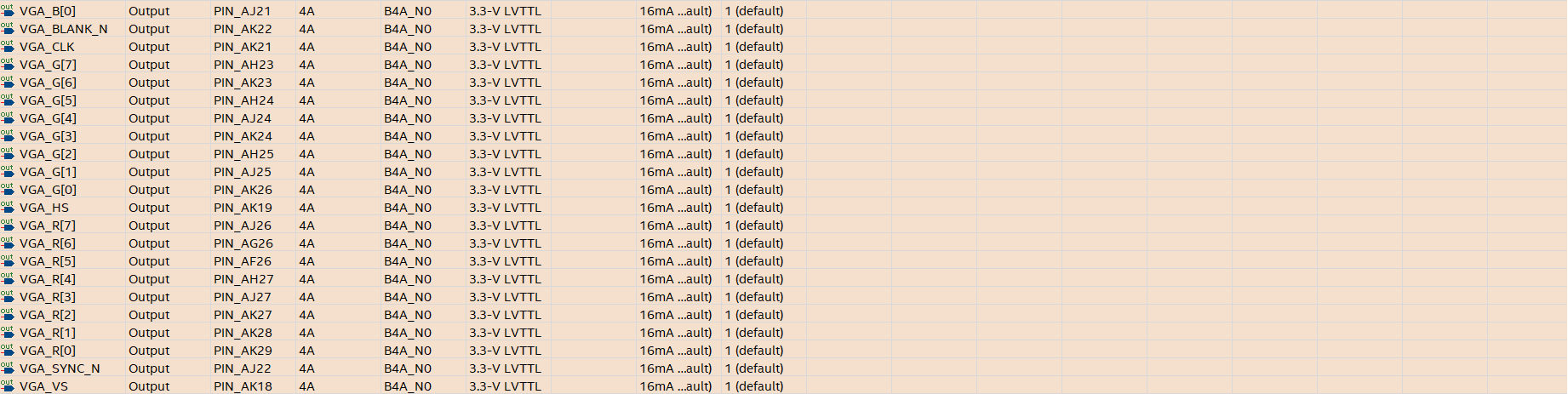


图2-5引脚分配图

##### 2.6全编译

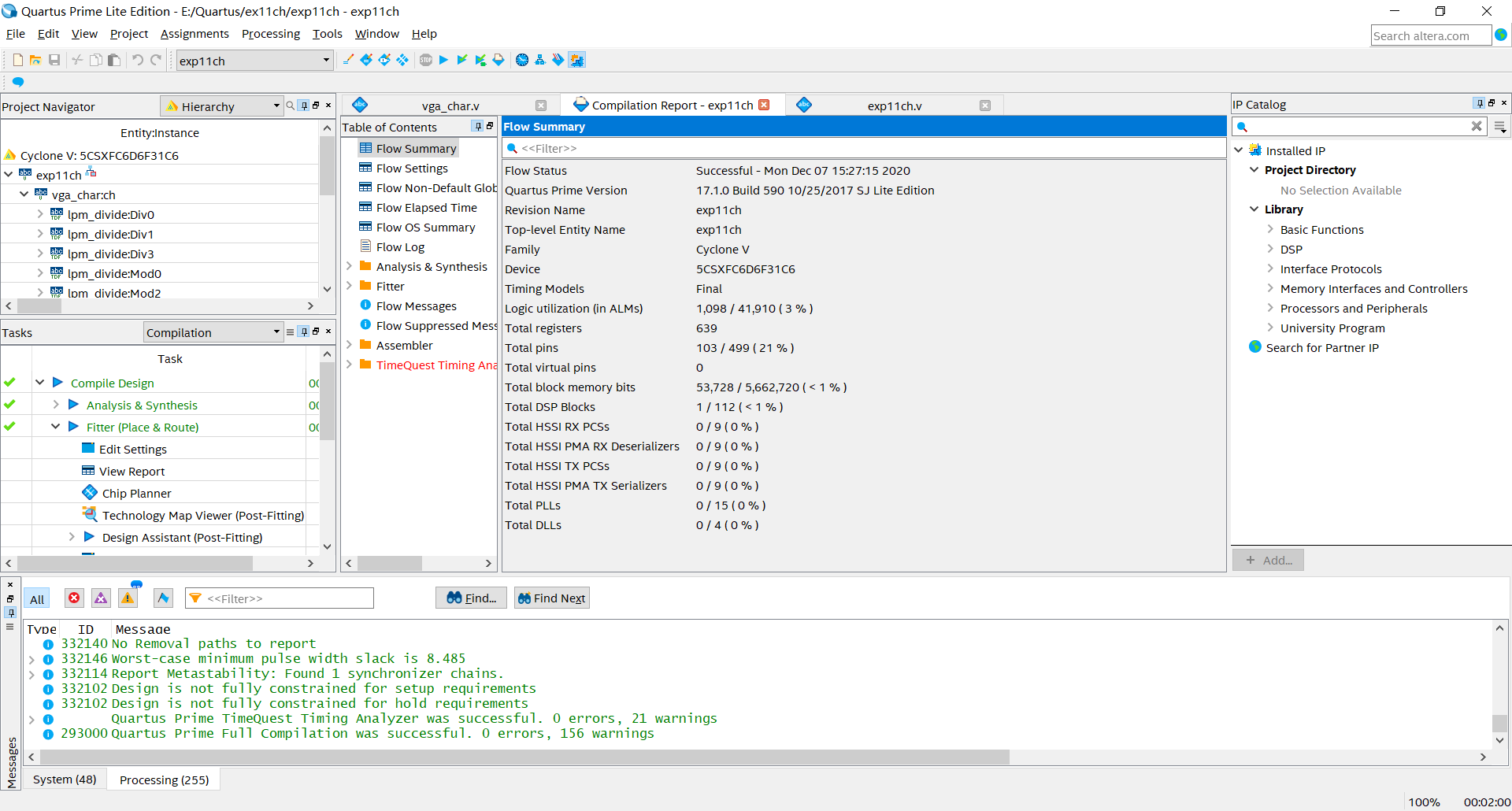


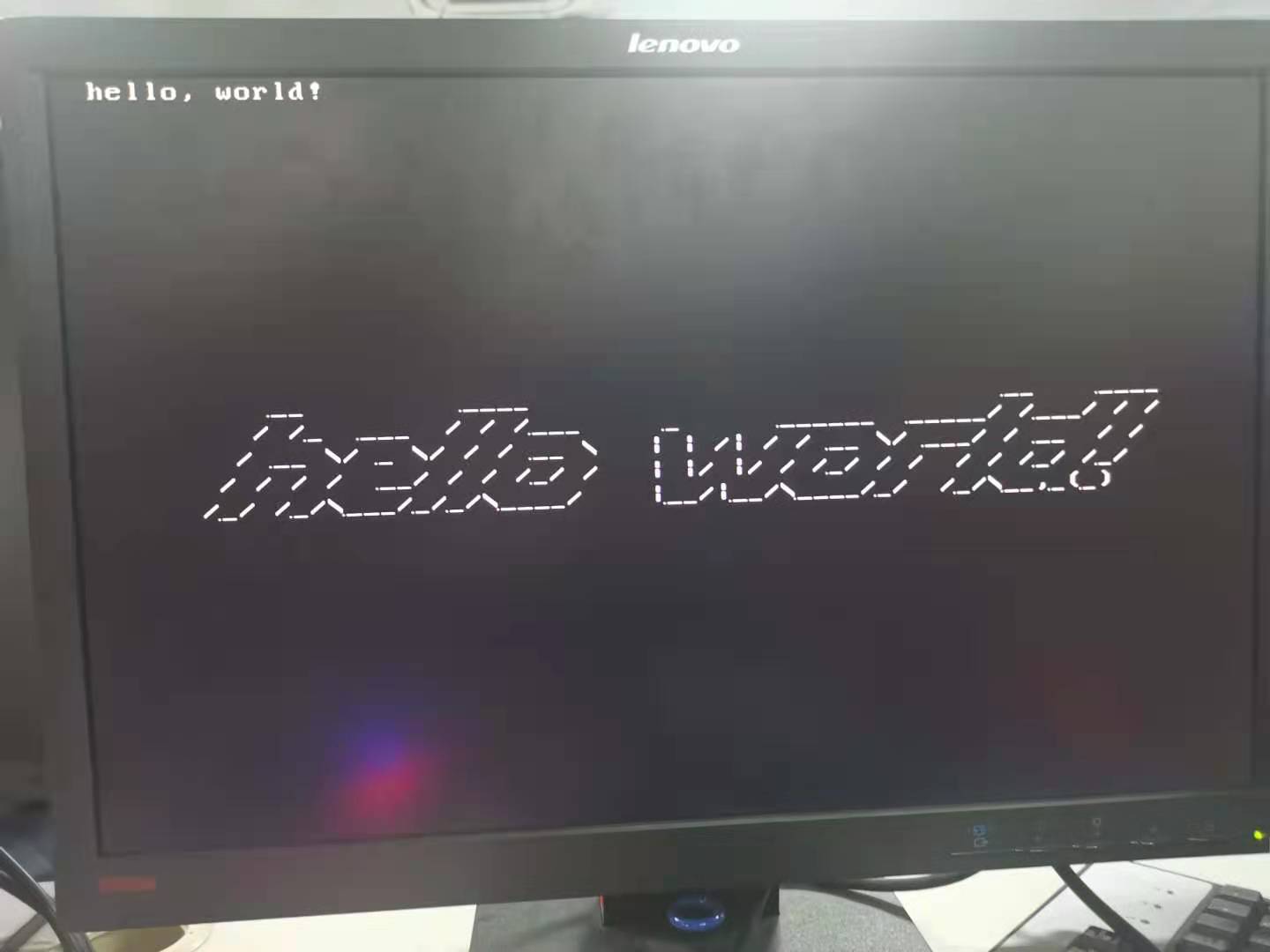
图2-6全编译成功

#### 实验总结

本次实验设计了字符串输入界面. 结合之前实验中的ROM、RAM存储器, PS2键盘，VGA显示器，通过图2-2-1的方式与本实验中要设计的交互模块vga\_char构成一个具有输入、显示字符功能的交互机器. 本次实验中需要注意的是对显存的读写的时序逻辑应分别与其对应的显示、字符输入功能的时序相一致. 显存的读写使能端的控制也应当与其相应的功能相符. 本次实验的实现中美中不足的是光标移动是不受当前输入的影响的, 即无法做到像文本编辑器一样限制光标移动的位置.

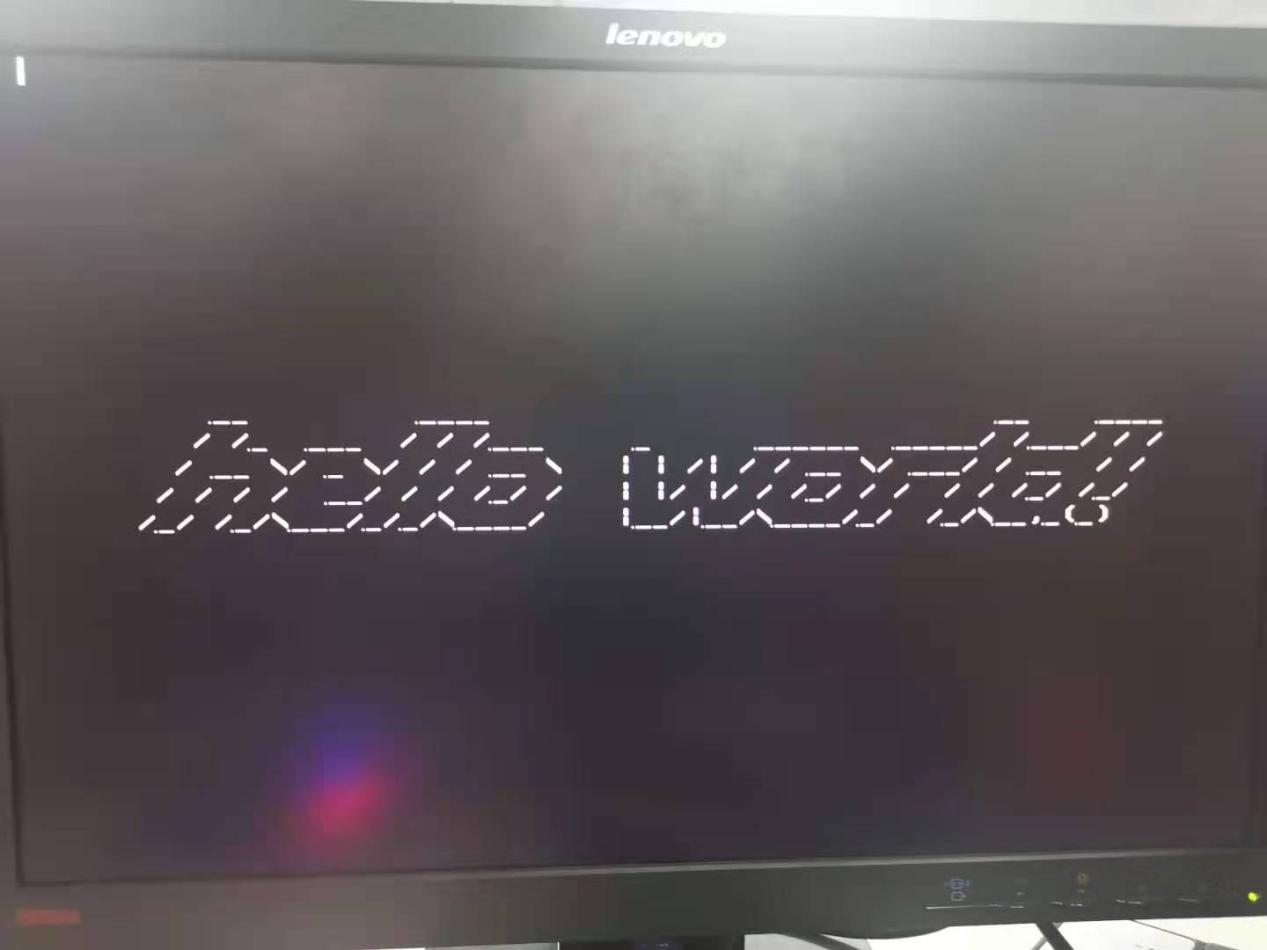
#### 附

以下给出的是本实验的的图片：



附图1 效果图1

屏幕中央的艺术字”hello world!”字样由显存的mif初始化文件实现. 该mif文件在提交的项目文件夹中(./helloworld.mif).屏幕最上方的”hello, world!”字样由键盘输入. 由于拍摄原因附图1中并没有拍到光标. 故以附图2作为补充.



附图1 效果图2

由于长按效果不便以照片形式展现，这里不再赘述.