

同济大学计算机系  
计算机系统实验报告



题目: 龙芯 MIPS 指令集 CPU 核 LS132R 改造与性  
能验证

学 号 2050525  
姓 名 陈开煦  
专 业 计算机科学与技术  
授课老师 秦国锋老师

# 目录

<b>1 实验环境与实验内容</b>	<b>3</b>
<b>2 实验过程与实验方法</b>	<b>3</b>
2.1 数码管实验 . . . . .	3
2.2 Flash 读取实验 . . . . .	4
2.3 将龙芯移植到 NEXY4 开发板上（AXI 通信实验） . . . . .	8
2.4 汇编版点亮 LED 实验 . . . . .	10
2.5 C 语言版点亮 LED 灯实验 . . . . .	12
2.6 C 语言版时钟实验 . . . . .	13
<b>3 程序修改说明</b>	<b>13</b>
<b>4 约束文件修改说明</b>	<b>14</b>
<b>5 仿真分析</b>	<b>14</b>
<b>6 性能验证模型及算法程序</b>	<b>14</b>
<b>7 性能验证程序下板测试过程与实现</b>	<b>15</b>
<b>8 CPU 的性能指标定性分析（分别用 C 语言和 MIPS 指令编写性能验证程序，C 语言利用 gcc 编译器编译生成目标程序，MIPS 指令汇编程序 Mars 编译生成目标程序，测试并比较分析两种方式 CPU 的定点运算性能及差异，单位：MIPS）</b>	<b>23</b>
<b>9 实验体会</b>	<b>25</b>

# 1 实验环境与实验内容

实验环境:

- 操作系统: Windows 10 64bit
- 仿真工具: Vivado 2016.3 simulator
- 综合工具: Vivado 2016.3 / vivado 2018.3
- 下板工具: Vivado 2016.3
- 开发板: DIGILENT Nexys4 DDR<sup>TM</sup> FPGA Board

**实验内容:** 这次实验的实验内容是改造龙芯 CPU, 我们需要将龙芯 CPU LS132R 内核移植到我们的 Nexy4 开发板上。具体从思路上来说, 我们需要将 c 语言/mips 汇编语言的程序代码以二进制的形式烧录到 flash 中, 然后再烧录 FPGA 程序(verilog 语言编写, 在 vivado 上得到的.bit)。当我们按下开发板上的复位键的时候, LS132R 会先进入 flash 上取指令, 完成初始化过程, 然后进入 main 函数进行 C 语言程序。

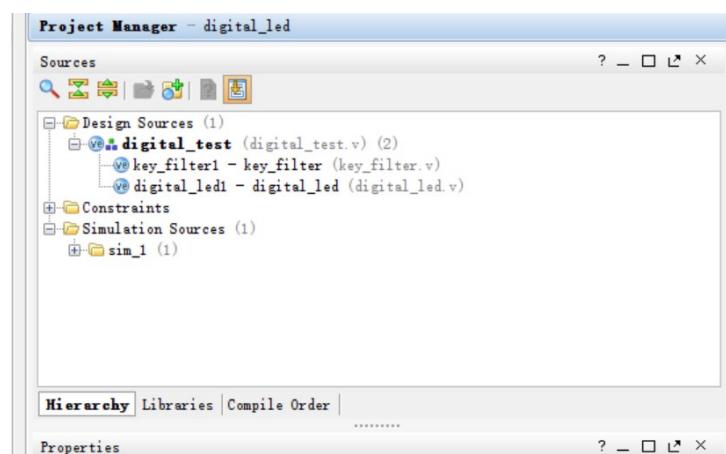
为了使整个实验过程更加顺利可靠, 我们先进行了数个测试实验, 然后向 Nexy4 开发板移植了龙芯 CPU。

## 2 实验过程与实验方法

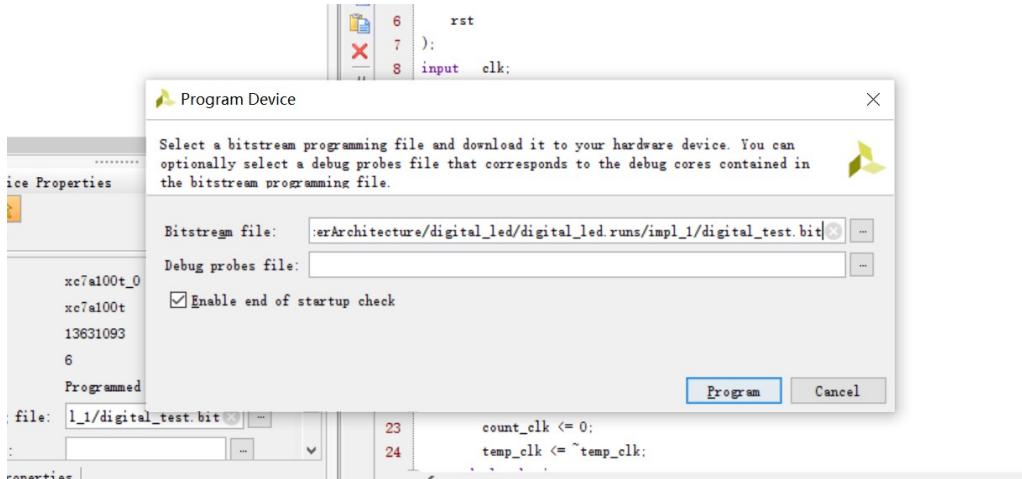
### 2.1 数码管实验

这个实验主要复习了通过 Vivado 建立工程, 添加源文件, 添加约束文件以及综合, 下板的全过程, 在下面我们会给出核心步骤的截图和最终的结果。

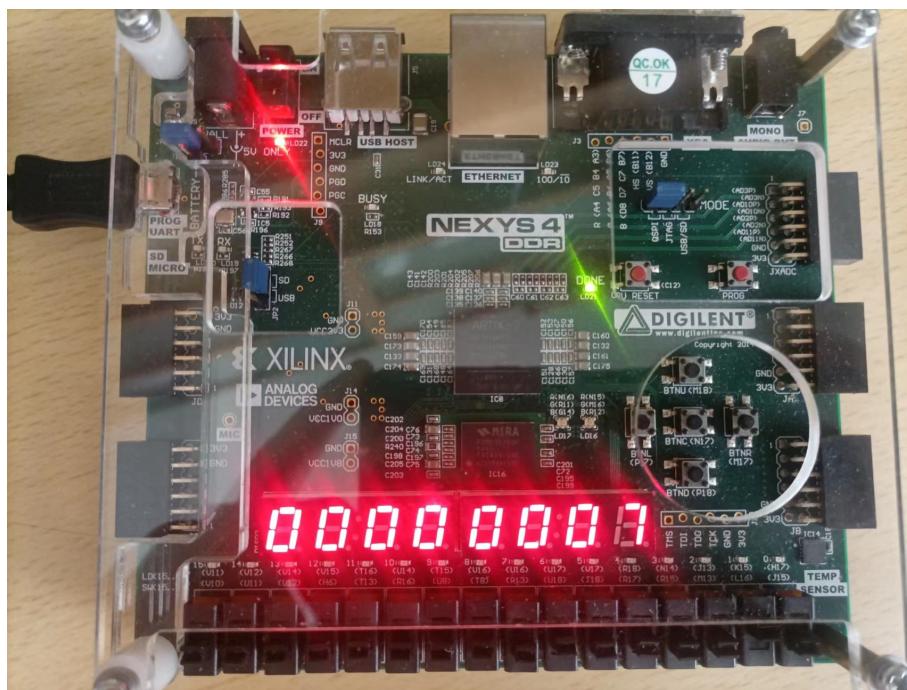
- 首先我们新建工程, 命名为 *digital\_led*.
- 然后我们添加源文件, 这里使用的是“数码管测试”文件夹下的"digital\_led.v","key\_filter.v" 和"digital\_test.v". 添加完源文件后整个工程的结构如下:



- 接下来我们进行综合，点击“synthesis”即可，在综合之后我们需要添加管脚，这里可以手动添加也可以直接导入一个约束文件，我们选择前者。
- 接下来我们点击“run implementation”，在该步结束之后生成 bit 流文件。
- 在得到 bit 流文件之后我们打开“hardware manager”，将 bit 流写入开发板即可。



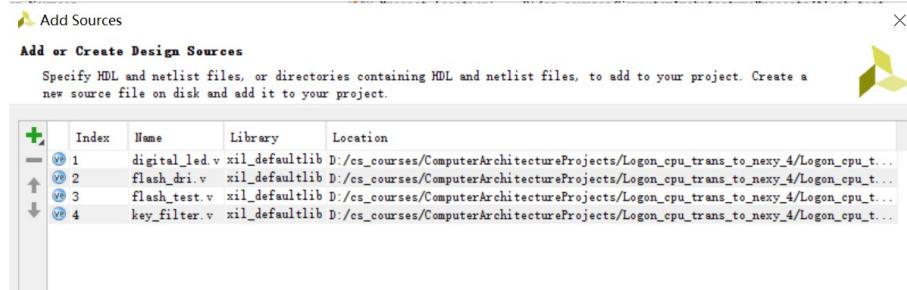
在开发板上可以看到亮起来的数码管。



## 2.2 Flash 读取实验

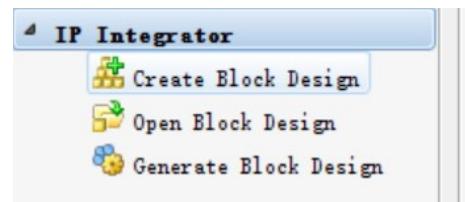
通过学习开发教程可知，为了龙芯 CPU 核可以在 Nexy4 开发板上工作，我们需要先向 flash 中烧入已知数据的二进制文件。为了跑通这一步，我们在这个实验中向 Nexy4 开发板中烧入一些二进制数据文件，然后写一个 verilog 程序对相应位置的数据进行读操作并将其显示在数码管上。接下来我们说明其步骤，重点说明一些之前未接触过的操作。

- 首先，我们建立工程，添加源文件，这里添加的源文件来自于"Flash 测试" 文件夹，工程的源文件如下图所示：

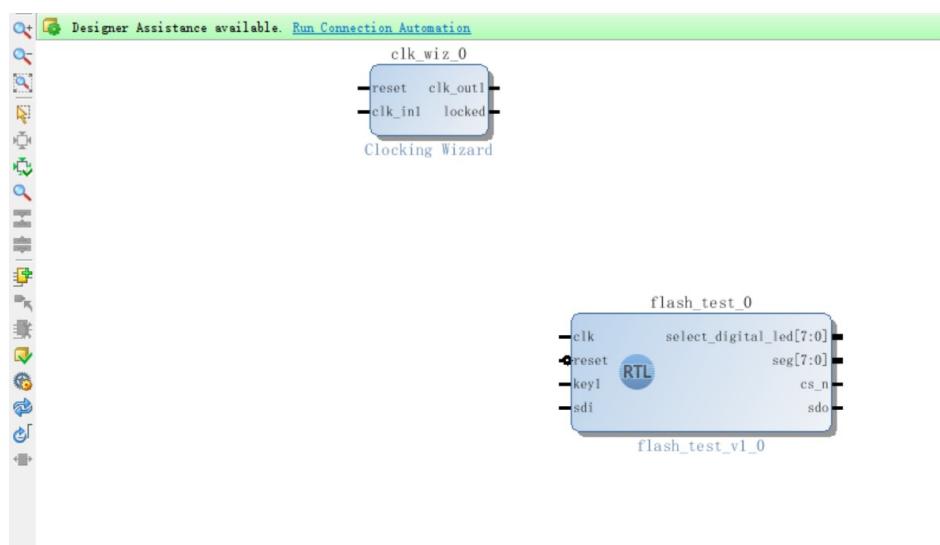


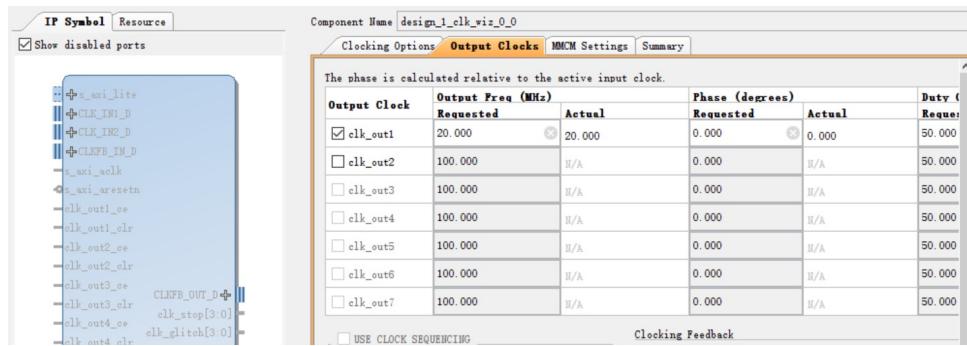
- 由于 Flash 的读取速度较慢，无法跟上 Nexy4 开发板 100Mhz 的速度，因此需要对时钟进行分频。这里我们使用在 block design 中添加 ip 核的方式来完成分频。

首先添加 Block design，然后在 block design 中添加 module，如下图所示。

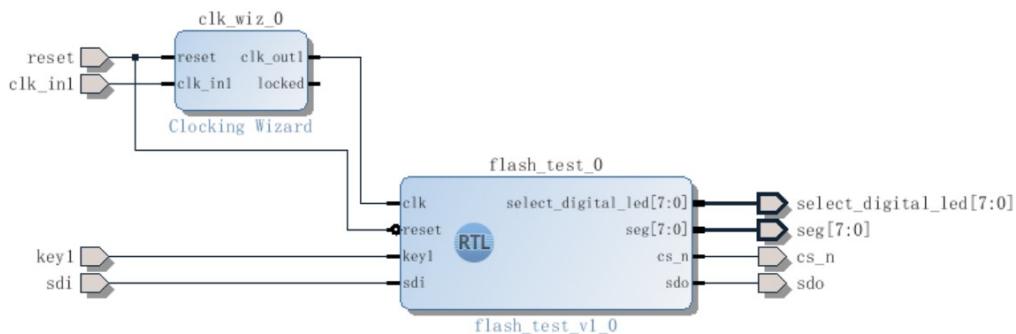


- 接下来我们添加一个名为 clocking wizard 的 IP 核，并修改其频率为 20Mhz，如下图所示

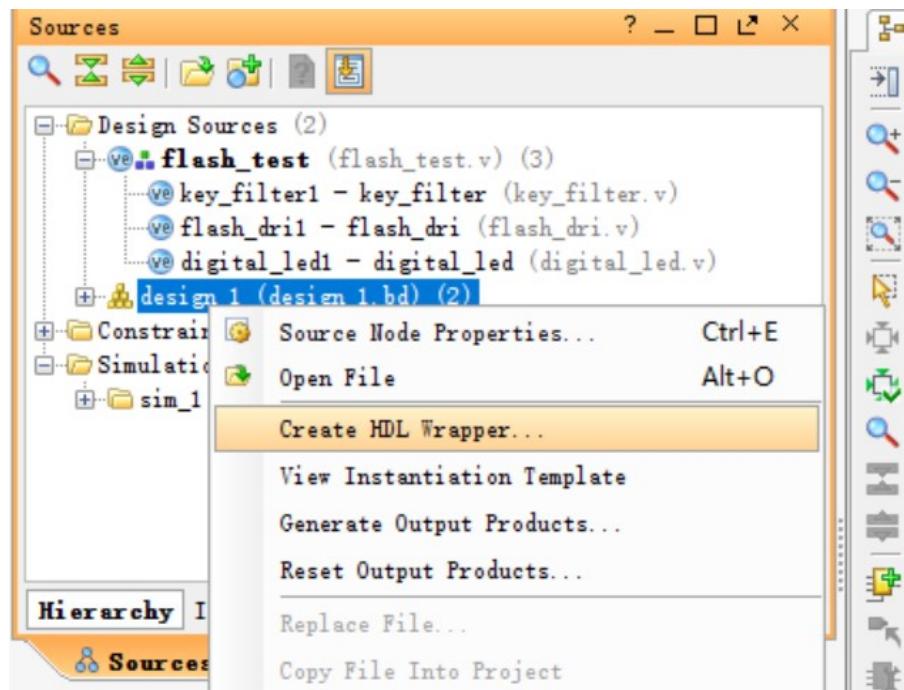




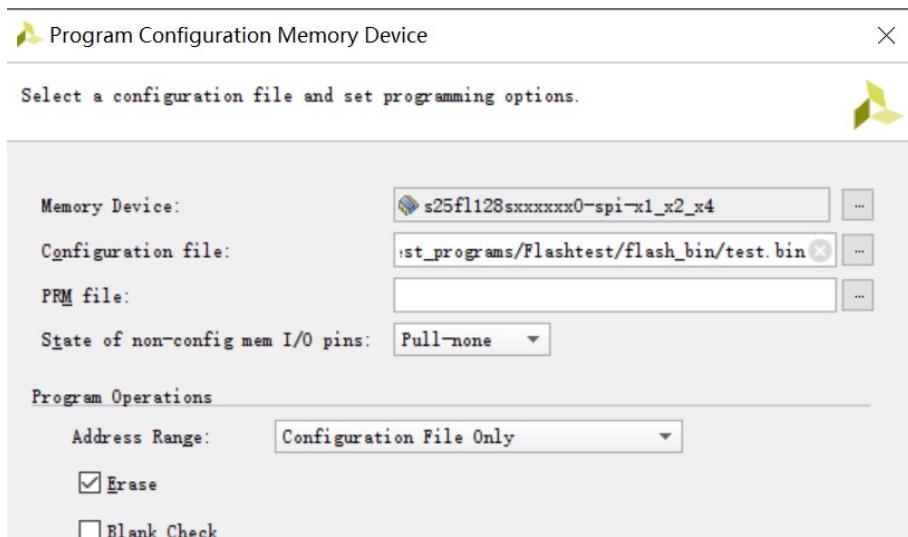
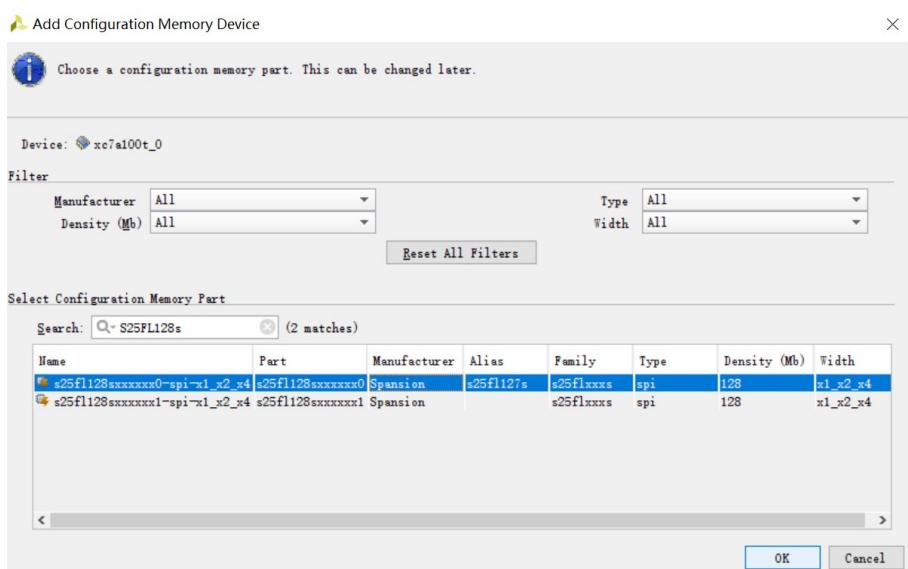
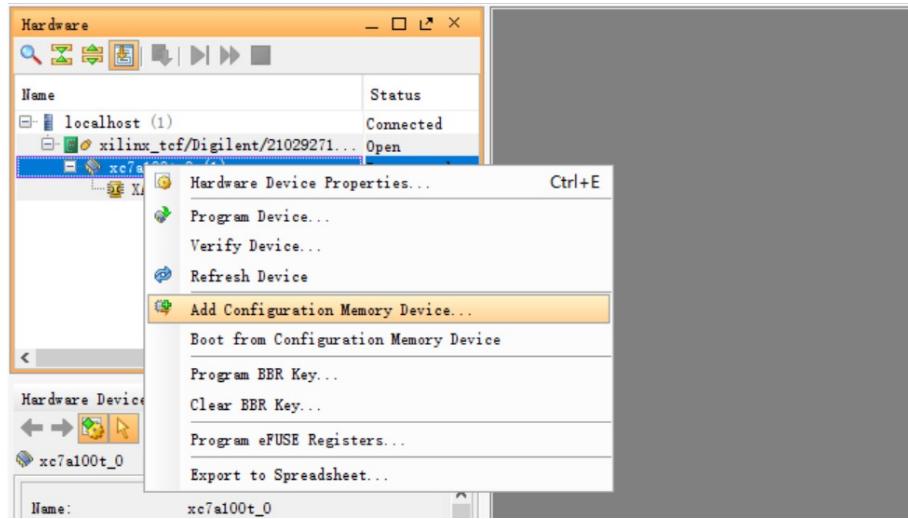
- 在这之后我们将模块之间连接起来，如下图所示。



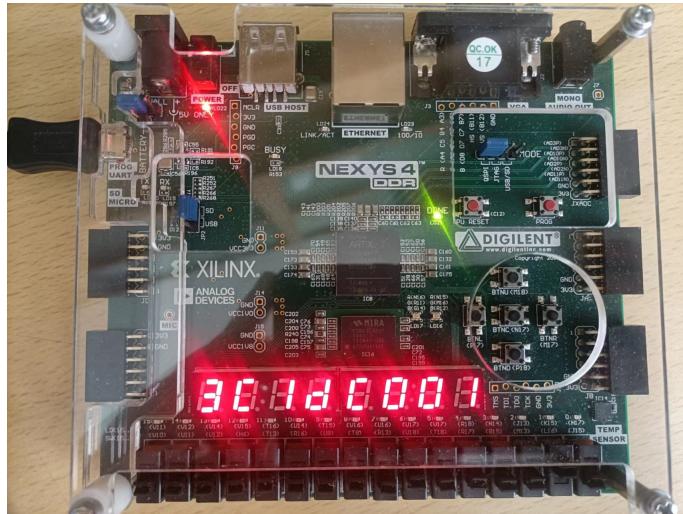
- 接下来我们通过 Vivado 提供的 HDL Wrapper 将设计图转换为 Verilog 文件，如下图所示。



- 接下来我们向板卡上烧入文件，需要注意的是需要先烧入数据文件再烧入 verilog 程序，如下列三张图所示。



- 最后我们将 verilog 程序生成的 Bit 文件写入 NEXY4 板卡，通过按按钮，程序会从 flash 中不停地以 4 字节为单位读取数据并显示在数码管上，如下所示。



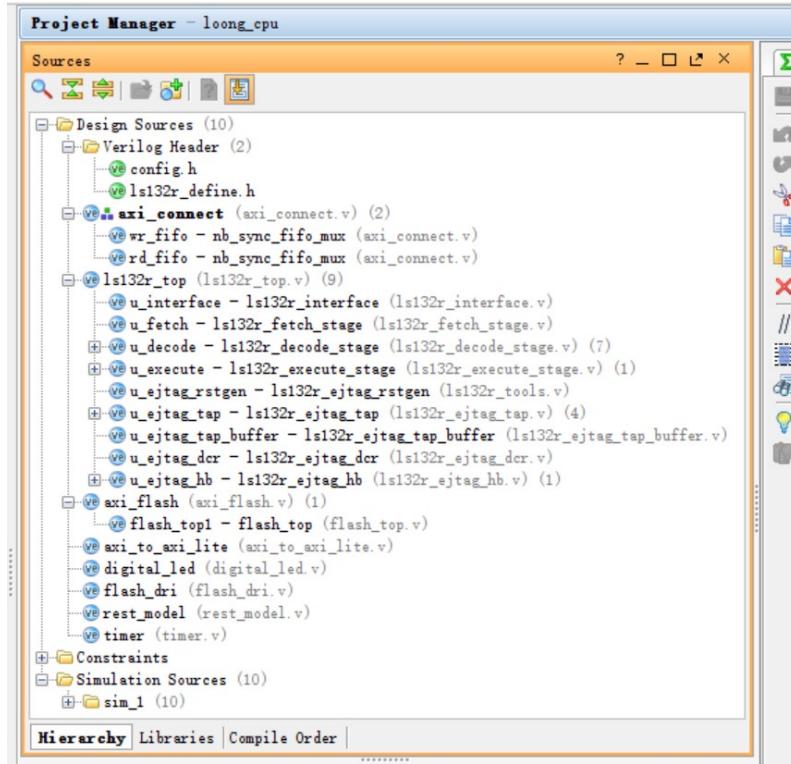
可以看到某次读取的数据为 3C1DC001，接下来我们查找 bin 文件，发现确实存在该数值（第一行最右侧），说明我们的程序可以从 flash 中正确读取数据。

	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f	
00000000h:	00	00	00	00	00	00	00	00	00	00	00	3C	1D	C0	01	;	
00000010h:	27	BD	80	00	3C	08	BF	C0	25	08	14	28	00	00	58	20	;
00000020h:	00	00	00	00	00	00	00	00	00	00	00	0B	F0	00	12	;	
00000030h:	00	00	00	00	00	00	00	00	01	0B	70	20	8D	CD	00	00	;
00000040h:	AD	8D	00	00	21	6B	00	04	3C	09	C0	00	25	29	00	00	;
00000050h:	3C	0A	C0	00	25	4A	00	68	01	2B	60	20	00	00	00	00	;
00000060h:	00	00	00	00	15	8A	FF	F4	00	00	00	00	00	00	00	00	;
00000070h:	3C	09	C0	00	25	29	00	68	00	00	58	20	00	00	00	00	;
00000080h:	00	00	00	00	00	00	00	00	0B	F0	00	28	00	00	00	00	;
00000090h:	00	00	00	00	00	00	68	20	AD	8D	00	00	21	6B	00	04	;
000000a0h:	3C	0A	C0	00	25	4A	00	90	01	2B	60	20	00	00	00	00	;
000000b0h:	00	00	00	00	15	8A	FF	F7	00	00	00	00	00	00	00	00	;
000000c0h:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	;

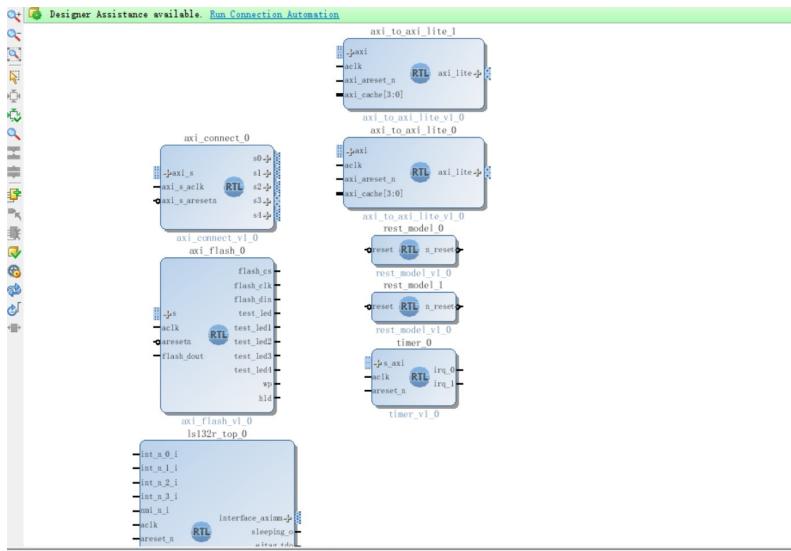
### 2.3 将龙芯移植到 NEXY4 开发板上（AXI 通信实验）

在这一个实验中我们完成的实现 AXI 通信协议，在这个过程中我们一起完成了将龙芯 CPU 核移植到 NEXY4 开发板的过程。步骤如下。

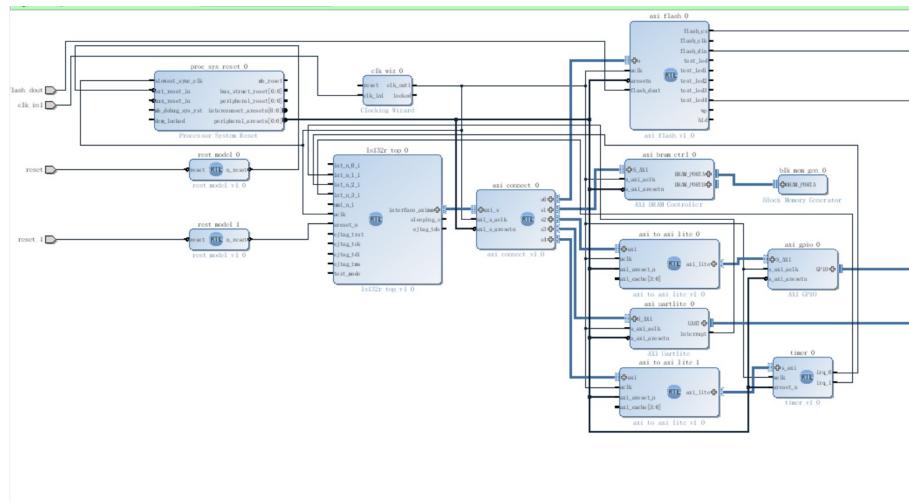
- 首先我们新建一个项目，导入参考工程里面的文件，整个工程结构如下页所示。



- 接下来我们建立一个 Block design，并将所有的模块导入进去，如下所示：



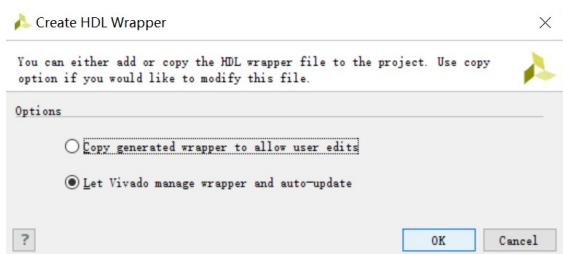
- 接着我们通过添加 IP 核的方式向 block design 中添加一些外设，如 gpio 和串口等并将它们连起来。



- 接下来我们进行地址分配，如下图所示

Cell	Slave Interface	Base Name	Offset Address	Range	High Address
axi_connect_0	s0 (32 address bits : 4G)				
	-> axi_flash_0	s	reg0	0x1FC0_0000	64K ~ 0x1FC0_FFFF
	-> axi_bram_ctrl_0	S_AXI	Mem0	0xC000_0000	32K ~ 0xC000_FFFF
	-> s1 (32 address bits : 4G)				
	-> axi_to_axi_lite_0	axi	reg0	0xD000_0000	64K ~ 0xD000_FFFF
	-> s2 (32 address bits : 4G)				
	-> axi_to_axi_lite_0	S_AXI	Reg	0xD060_0000	64K ~ 0xD060_FFFF
	-> s3 (32 address bits : 4G)				
	-> axi_to_axi_lite_0	axi	reg0	0xD070_0000	64K ~ 0xD070_FFFF
	-> s4 (32 address bits : 4G)				
	-> axi_to_axi_lite_1	S_AXI	Reg	0xD070_0000	64K ~ 0xD070_FFFF
ls132r_top_0	interface_aximm (32 address bits : 4G)				
	-> axi_connect_0	axi_s	reg0	0x0000_0000	4G ~ 0xFFFF_FFFF
	-> axi_to_axi_lite_0				
	-> axi_to_axi_lite_0	S_AXI	Reg	0xD000_0000	64K ~ 0xD000_FFFF
	-> axi_to_axi_lite_1				
	-> axi_to_axi_lite_1	axi	reg0	0xD070_0000	64K ~ 0xD070_FFFF
	-> timer_0	s_axi	reg0	0xD070_0000	64K ~ 0xD070_FFFF

- 最后我们通过 vivado 的 HDL wrapper 生成 verilog 文件并将 wrapper 设置为顶层文件，如下。



- 接下来的步骤我们就很熟悉了，即进行 synthesis, implementation，生成 Bit 流，最终得到的 ls132r\_soc\_wrapper.bit 即为龙芯 CPU 的 bit 流文件。

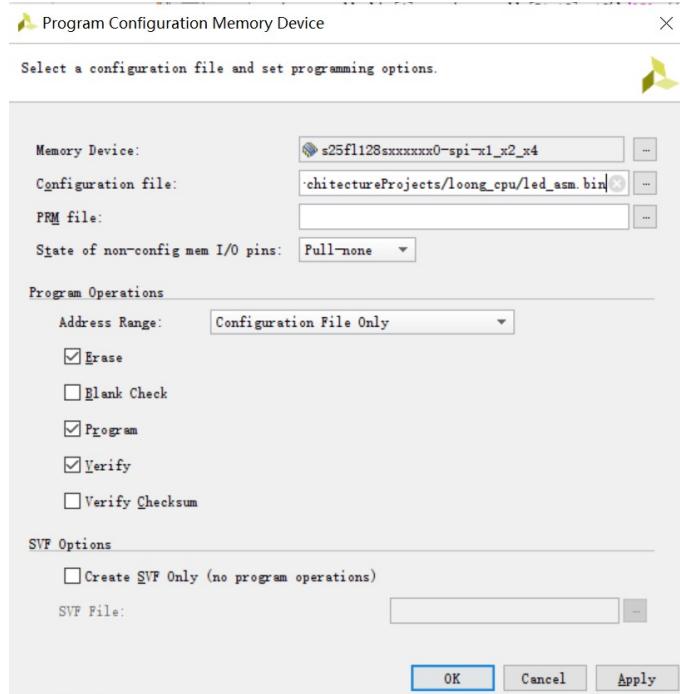
## 2.4 汇编版点亮 LED 实验

在将龙芯 LS132RCPU 核移植到开发板之后，我们进行使用汇编程序汇编得到的二进制文件作为烧入 Flash 中供龙芯 CPU 核读取的文件进行实验，在这一实验中我们使用的汇编程序为文件夹"汇编版点亮 LED 灯"中的"led\_asm.s"。

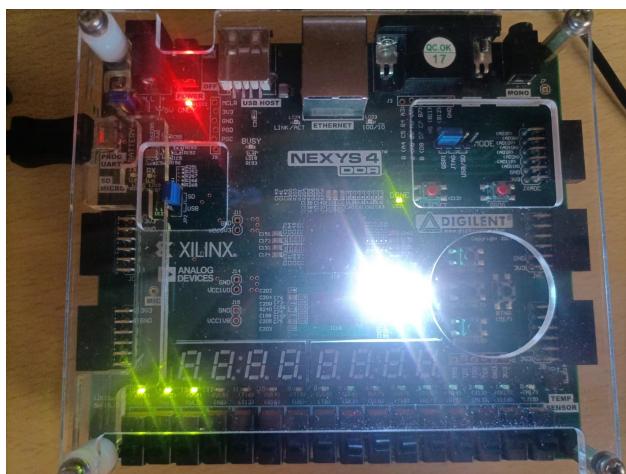
接下来我们将其拷贝到"toolchain build" 目录下，然后将其汇编为二进制文件，在 cmd 窗口输入的命令如下：

```
1 mips-mti-elf-as.exe -32 -mips32 led_asm.s -o led_asm.o  
2 mips-mti-elf-ld.exe -T mytest.1d led_asm.o -o led_asm.om  
3 mips-mti-elf-objcopy.exe -O binary led_asm.om led_asm.bin  
4 mips-mti-elf-objdump.exe -D led_asm.om > led_asm.asm
```

由于该汇编程度汇编得到的二进制文件已经于文件夹中给出，我们在这里便不再过多赘述，接下来将该二进制文件"led\_asm.bin" 烧入到 Flash 中，然后将上一节得到的"ls132r\_soc\_wrapper.bit" 作为 FPGA 程序写入板卡，即完成了本实验的主要步骤。



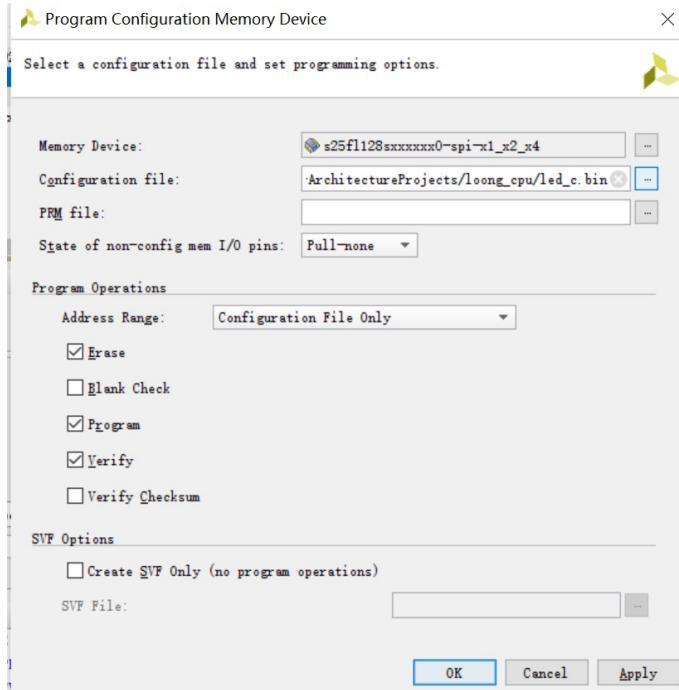
为了启动 CPU 使我们可以在数码管上看到结果，我们需要先长按 P17 按钮，再长按 M18 按钮对总线和 CPU 分别进行重置，随后我们便可以看到如下图所示的结果，可以看到 LED 灯顺利亮起，说明结果无误。



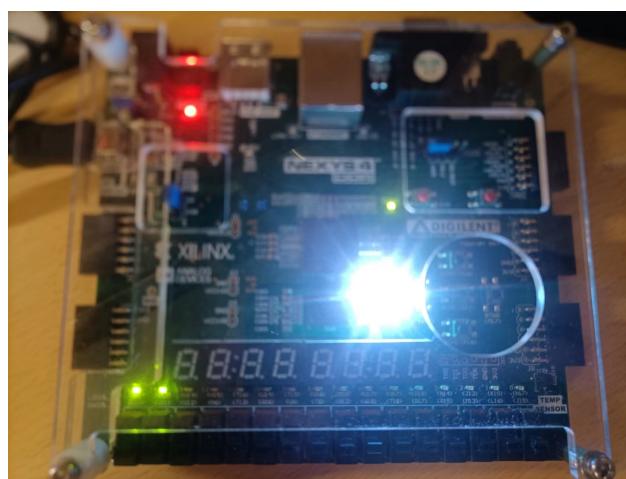
## 2.5 C 语言版点亮 LED 灯实验

在使用汇编语言得到得二进制文件作为烧入 FLASH 的数据文件后，我们使用 C 语言编译，链接，汇编得到的二进制文件作为数据文件来完成点亮 LED 灯的实验，该 C 语言文件已经在"C 语言版时钟实验/main.c" 中给出。

接下来我们将"main.c" 拷贝到"toolchain/build/src" 目录下，然后再进入 build 目录下将其编译为二进制文件"lec\_c.bin"。在这之后，我们将其烧入 Flash，并将上一节得到的"ls132r\_soc\_wrapper.bit" 作为 FPGA 程序写入板卡。



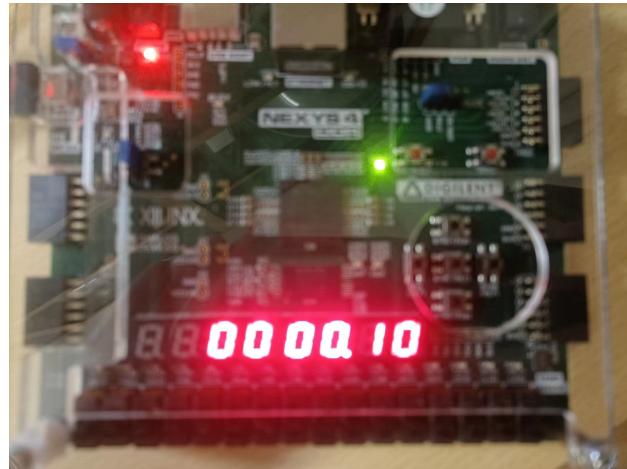
为了在开发板数码管上看到结果，我们先后长按 P17 和 M18 启动总线和 CPU，得到的结果如下，可以看到 LED 灯顺利亮起，说明结果无误：



## 2.6 C 语言版时钟实验

在这个实验中，我们使用 C 语言编写的时钟程序来验证 CPU 的中断功能是否正确，时钟的 C 语言程序在目录中的" 测试程序/C 语言版时钟实验" 中。和上一节一样，我们将该 C 语言程序经过编译，链接，汇编等数个过程得到二进制文件，并将二进制文件烧入 Flash，在这之后将上一节得到的"ls132r\_soc\_wrapper.bit" 作为 FPGA 程序写入板卡。

为了在开发板数码管上看到结果，我们先后长按 P17 和 M18 启动总线和 CPU，得到的结果如下，可以看到数码管上的数字随时间匀速跃升，说明结果无误：



## 3 程序修改说明

在上面的几个实验中我们使用的是龙芯 LS132R 的 CPU 源代码，因此基本没有需要修改程序的地方，唯一的一处修改出现在我们进行仿真的过程中。由于龙芯 CPU 的源代码是为了下板而写的，因此在 *flash\_top.v* 文件中，读取数据的方式是从 Flash 中进行读取，而在进行仿真的时候，我们需要直接从数据文件 *.data* 中读取，需要修改的两处如下所示：

```
1 //这些需要注释掉
2 reg [31:0] memory[0:2000];
3 initial
4 begin
5     $readmemh("D:/cs_courses/ComputerArchitectureProjects/loong_cpu
6         /cal_c_sim.data",memory);
7 end
```

```
1 // 这些需要注释掉
2 rdata[31:24]<=temp_rdata[7:0];
3 rdata[23:16]<=temp_rdata[15:8];
4 rdata[15:8]<=temp_rdata[23:16];
5 rdata[7:0]<=temp_rdata[31:24];
6 if (raddr >= 32'd0 && raddr <= 32'd8) begin
7     rdata <= 32'h0;
```

```

8  end else begin
9      rdata <= memory[(raddr>>2)-3];
10 end

```

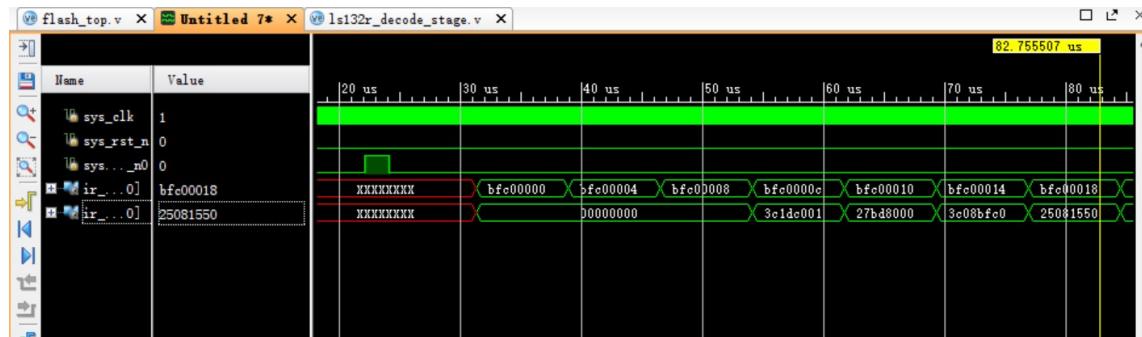
## 4 约束文件修改说明

在龙芯 LS132R 的 CPU 核中，所有的电路会在开始运行前被重置，通过两个管脚来实现，这两个管脚对应开发板上的 P17 和 M18，这两个管脚以及数码管等管脚的配置都已经在龙芯 CPU 参考工程中给出，我未做修改。

## 5 仿真分析

为了验证 AXI 通信是否正常，我们还进行了前仿真的操作，在前仿真中，我们将 *flash\_top.v* 中从 flash 中读取数据改为从文件中读取数据，只要可以从仿真波形中观察到读回来的指令，便说明 AXI 通信是正确的。

为此，我们需要先修改 *flash\_top.v*，使之可以从文件中读取 16 进制的指令，接下来我们将 *test\_tb* 设置为顶层，接下来启动前仿真即可，前仿真的波形图如下。



从波形图中可以看出，CPU 从地址为 0xbfc00000 中读取指令，并读出了 3c1dc001 这条指令，这与我们的数据文件中的 16 进制数相符。

## 6 性能验证模型及算法程序

我们进行性能验证的模型如下页所示：

可以看到，该性能测试模型中创建了四个数组，在一个循环中迭代计算数组中每个元素的值。其中 d 数组第 i 个位置的值依赖于 c 数组，b 数组，a 数组和 d 数组自身 i-1 个位置的值，因此我们在下板验证时，只需要验证 d 数组的最后一个元素，即 d[49] 的数值无误，便可以证明程序和下板过程的正确性。

通过编写简单的 C 语言程序，我们可以计算得到 d[49] 的值为 -41894，即二进制下的 FFFF5C5A。这里需要额外说明的是，尽管我们的测试模型中并未出现负号，看似全部是正整数，但由于我

```

a[m],b[m],c[m],d[m];
a[0]=0.0;
b[0]=1.0;
a[i]=a[i-1]+i;
b[i]=b[i-1]+3i;
c[i]=  

  { a[i],      0≤i≤9  

    a[i]+b[i], 10≤i≤29  

    (a[i]*b[i])<<1, 30≤i≤49  

  }  

d[i]=  

  { b[i]+c[i] , 0≤i≤9  

    a[i]*c[i], 10≤i≤29  

    c[i]*b[i]/(d[i-1] >>1) , 30≤i≤49
  }

```

们无论是在汇编语言中还是 C 语言中，全部使用的是 32 位的整型，而在 d 数组的计算中会存在溢出的情况，因此会出现负数。

## 7 性能验证程序下板测试过程与实现

性能测试模型的下班验证总共分为如下几个过程，首先，我们编写性能测试模型的汇编代码和 C 语言代码，如下所示。

```

1 //C 语言文件
2 int main(void)
3 {
4     set_gpio_tri(0xffff0000 false);
5     int a[50],b[50],c[50],d[50],i;
6     i=1;
7     a[0]=0;
8     b[0]=1;
9     while(i<50)
10    {
11        a[i]=a[i-1]+i;
12        b[i]=b[i-1]+3*i;
13        i++;
14    }
15    i=0;
16    while(i<50)
17    {
18        if(0<=i&&i<=9)

```

```

19     {
20         c[ i ]=a[ i ];
21         d[ i ]=b[ i ]+c[ i ];
22     }
23     else if(10<=i&&i <=29)
24     {
25         c[ i ]=a[ i ]+b[ i ];
26         d[ i ]=a[ i ]*c[ i ];
27     }
28     else
29     {
30         c[ i ]=( a[ i ]*b[ i ]) <<1;
31         d[ i ]=c[ i ]*b[ i ]/( d[ i -1]>>1 );
32
33     }
34     i++;
35 }
36 int minus=0;
37 if(d[49]<0)
38 {
39     minus=10;
40     d[49]=-d[49];
41 }
42 while(1)
43 {
44     digital_led(0,d[49] %10);
45     digital_led(1,(d[49]/10) %10);
46     digital_led(2,(d[49]/100) %10);
47     digital_led(3,(d[49]/1000) %10);
48     digital_led(4,(d[49]/10000) %10);
49     digital_led(5,(d[49]/100000) %10);
50     digital_led(6,(d[49]/1000000) %10);
51     digital_led(7,minus);
52 }
53 return 0;
54 }
```

```

1 # 汇编代码
2 .text
3 .align 2
4 .globl main
5 .set nomips16
6 .set nomicromips
```

```

7   main:
8     nop
9     lui $3,    0xd000
10    ori $3,$3,0x0004
11    lui $2,0
12    sw  $2,0x0($3)
13    addi $8,$0,1
14    lui  $2,0xC000
15    ori   $2,0x0000
16    addi $3,$0,0
17    sw   $3,0($2)
18    addi $3,$0,1
19    sw   $3,200($2
20 inita_b:
21    mul   $9, $8,3
22    lw    $10,0($2)
23    add   $10,$10,$8
24    sw    $10,4($2)
25    lw    $11,200($2)
26    add   $11,$11,$9
27    sw    $11,204($2)
28    addiu $8,$8,1
29    addiu $2,$2,4
30    slti  $3,$8,50
31    bnez  $3,inita_b
32    nop
33    lui   $2,0xC000
34    ori   $2,0x0000
35    addiu $8,$0,0
36 calc_d1:
37    lw    $9,0($2)
38    add   $11,$9,$0
39    sw    $11,400($2)
40    lw    $10,200($2)
41    add   $12,$10,$11
42    sw    $12,600($2)
43    addiu $8,$8,1
44    addiu $2,$2,4
45    slti  $3,$8,10
46    bnez  $3,calc_d1
47    nop
48

```

```

49 calc_d2 :
50     lw      $9,0($2)
51     lw      $10,200($2)
52     add   $11,$9,$10
53     sw      $11,400($2)
54     mul   $12,$9,$11
55     sw      $12,600($2)
56     addiu $8,$8,1
57     addiu $2,$2,4
58     slti  $3,$8,30
59     bnez  $3,calc_d2
60     nop
61 calc_d3 :
62     lw      $9,0($2)
63     lw      $10,200($2)
64     lw      $12,596($2)
65     mul   $11,$9,$10
66     sll   $11,$11,1
67     sw      $11,400($2)
68     mul   $11,$11,$10
69     sra   $12,$12,1
70     div   $12,$11,$12
71     sw      $12,600($2)
72     addiu $8,$8,1
73     addiu $2,$2,4
74     slti  $3,$8,50
75     bnez  $3,calc_d3
76     nop
77 loop:
78     lw      $24,596($2)
79     addiu $8,$0,0
80     andi  $15,$24,0x000f
81     jal    digital_led
82     addiu $8,$0,1
83     sra   $24,$24,4
84     andi  $15,$24,0x000f
85     jal    digital_led
86     addiu $8,$0,2
87     sra   $24,$24,4
88     andi  $15,$24,0x000f
89     jal    digital_led
90     addiu $$8,$0,3

```

```

91    sra    $24,$24,4
92    andi   $15,$24,0x000f
93    jal    digital_led
94    addiu $8,$0,4
95    sra    $24,$24,4
96
97    andi   $15, $24, 0x000f
98    jal    digital_led
99    addiu $8, $0, 5
100   sra    $24, $24, 4
101   andi   $15, $24, 0x000f
102   jal    digital_led
103   addiu $8, $0, 6
104   sra    $24, $24, 4
105   andi   $15, $24, 0x000f
106   jal    digital_led
107   addiu $8, $0, 7
108   sra    $24, $24, 4
109   andi   $15, $24, 0x000f
110   jal    digital_led
111   j     loop
112
113 digital_led:
114   beq   $15, 0, num0
115   beq   $15, 1, num1
116   beq   $15, 2, num2
117   beq   $15, 3, num3
118   beq   $15, 4, num4
119   beq   $15, 5, num5
120   beq   $15, 6, num6
121   beq   $15, 7, num7
122   beq   $15, 8, num8
123   beq   $15, 9, num9
124   beq   $15, 10,numA
125   beq   $15, 11,numB
126   beq   $15, 12,numC
127   beq   $15, 13,numD
128   beq   $15, 14, numE
129   beq   $15, 15, numF
130 id:
131   beq   $8, 0, id0
132   beq   $8, 1, id1

```

```

133    beq $8, 2, id2
134    beq $8, 3, id3
135    beq $8, 4, id4
136    beq $8, 5, id5
137    beq $8, 6, id6
138    beq $8, 7, id7

139
140 data:
141     lui $10, 0x0000
142     or  $10, $10, $9

143
144     lui      $3,0xd000
145     ori      $3,$3,0x0000
146     sw       $10,0($3)
147     jr      $ra

148
149 num0:
150     addiu $9,$0,0x03
151     j      id

152
153 num1:
154     addiu $9,$0,0x9f
155     j      id

156
157 num2:
158     addiu $9,$0,0x25
159     j      id

160
161 num3:
162     addiu$9,$0,0x0d
163     j      id

164
165 num4:
166     addiu$9,$0,0x99
167     j      id

168
169 num5:
170     addiu $9,$0,0x49
171     j      id

172
173 num6:
174     addiu $9,$0,0x41

```

```

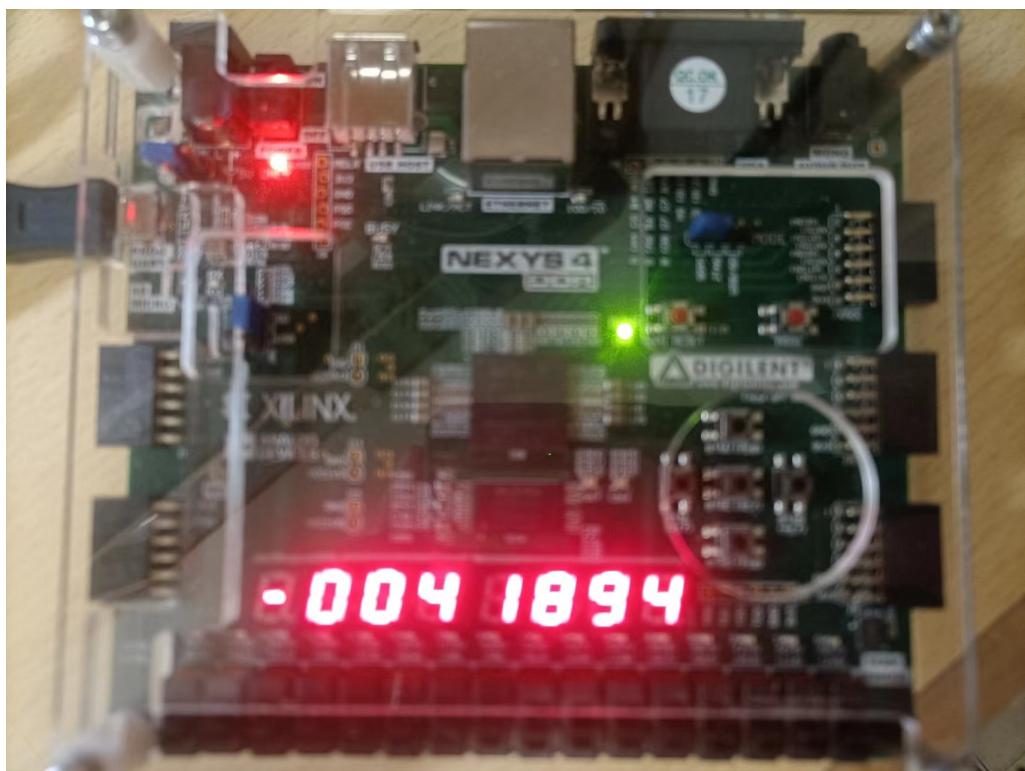
175      j      id
176
177 num7:
178     addiu $9,$0,0x1f
179      j      id
180 num8:
181     addiu $9,$0x01
182      j      id
183 num9:
184     addiu $9,$0,0x09
185      j      id
186 numA:
187     addiu $9,$0,0x11
188      j      id
189 numB:
190     addiu $9,$0,0xc1
191      j      id
192
193 numC:
194     addiu$9,$0,0x63
195      j      id
196 numD:
197     addiu $9,$0,0x85
198      j      id
199 numE:
200     addiu $9,$0,0x61
201      j      id
202 numF:
203     addiu $9,$0,0x71
204      j      id
205
206 id0:
207     ori   $9,$9,0xfe00
208      j      data
209 id1:
210     ori   $9,$9,0xfd00
211      j      data
212 id2:
213     ori   $9,$9,0xfb00
214      j      data
215 id3:
216     ori   $9,$9,0xf700

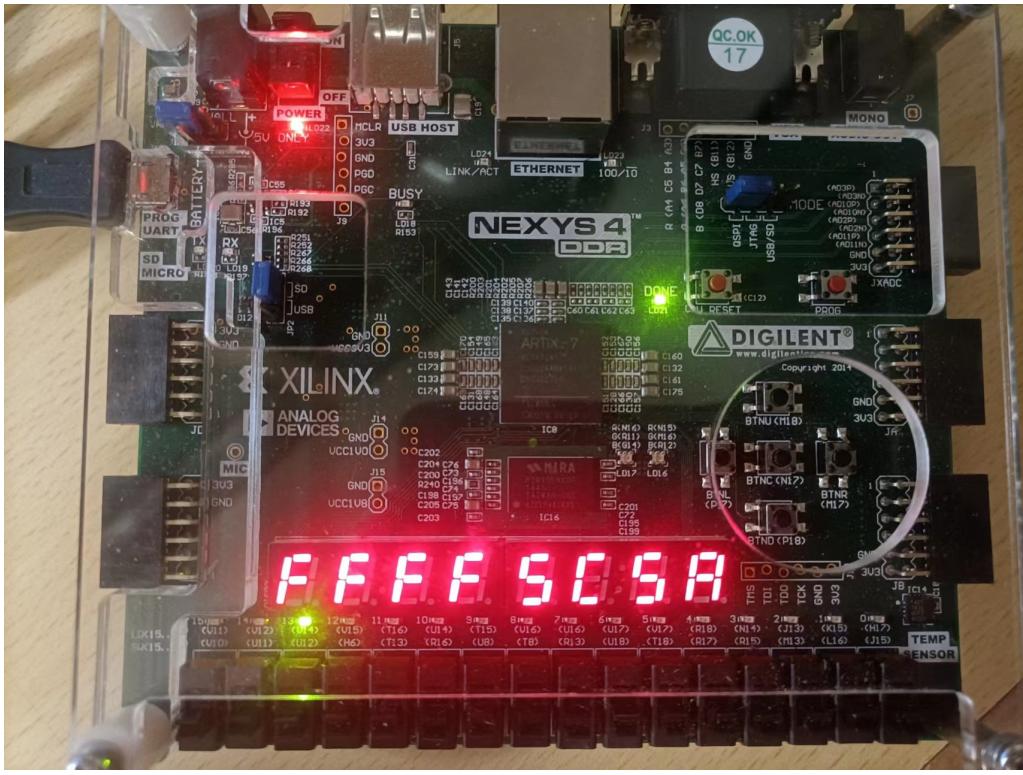
```

```
217      j      data
218 id4 :
219      ori    $9,$9,0xef00
220      j      data
221 id5 :
222      ori    $9,$9,0xdf00
223      j      data
224 id6 :
225      ori    $9,$9,0xbf00
226      j      data
227 id7 :
228      ori    $9,$9,0x7f00
229      j      data
```

接下来，我们将汇编代码汇编为二进制文件，将 C 语言编译，链接，汇编为二进制文件。分别将这两个文件作为数据文件烧入 Flash，再向 FPGA 板卡烧入龙芯 Ls132RCPU 生成的 bit 流文件。

通过先后长按 P17 和 M18 并启动总线和 CPU 并进行等待后，我们便可以在数码管上看到 d[49] 的结果，如下图所示。需要说明的是，C 语言编写的性能测试模型在数码管上会显示的是十进制数字-41894，而 Mips32 汇编语言编写的性能测试模型在数码管上会显示的是 16 进制数 FFFF5C5A，这二者在数值上是一致的。

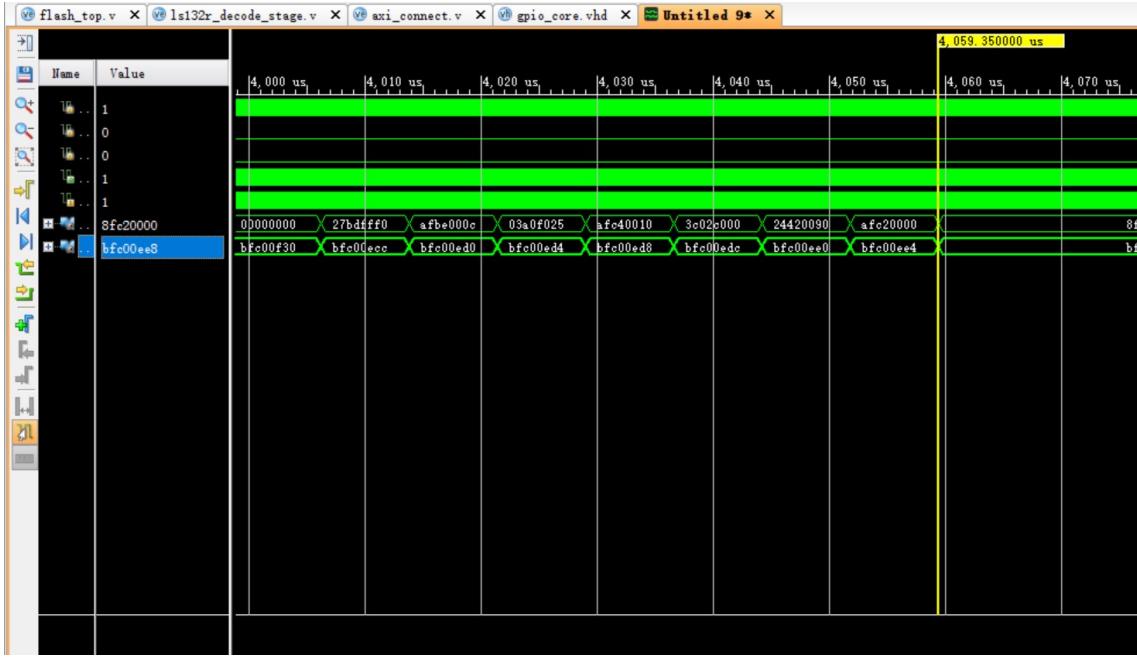




## 8 CPU 的性能指标定性分析(分别用 C 语言和 MIPS 指令编写性能验证程序, C 语言利用 gcc 编译器编译生成目标程序, MIPS 指令汇编程序 Mars 编译生成目标程序, 测试并比较分析两种方式 CPU 的定点运算性能及差异, 单位: MIPS)

在这一部分, 我们使用前仿真的方法来估测使用 C 语言指令编写的性能验证程序和使用汇编编写的性能验证程序在龙芯 LS132R 上执行分别需要多少个周期, 然后根据板卡的实际频率 (20MHz) 计算这两种方式得到的性能验证程序在龙芯 LS132R 上运行的 Mips 数值。

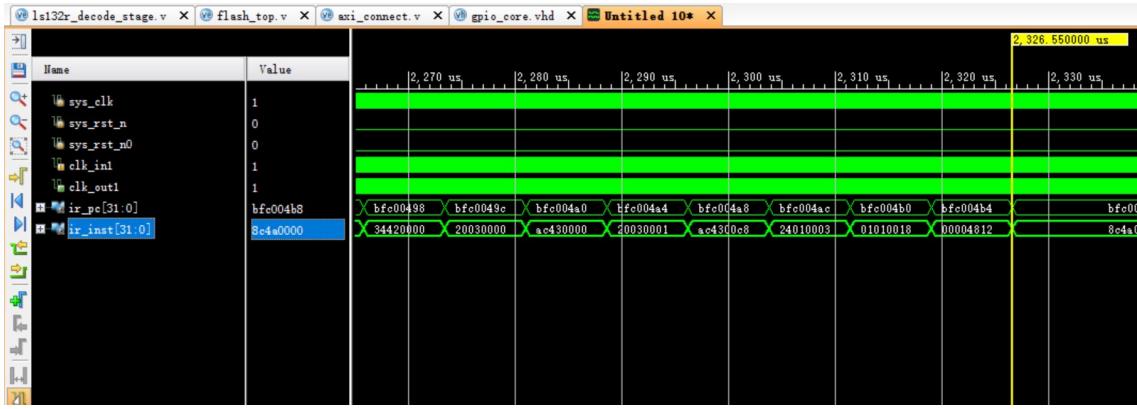
首先, 我们使用 C 语言生成的二进制文件, 将其转换为 16 进制的指令文件供 CPU 读取, 通过在仿真器中查看波形我们得到, 验证模型从启动到运行结束共需要 4028us, 共执行 504 条指令, 波形图如下所示。



由于仿真中时钟频率为  $\frac{1}{100ns} = 10MHz$ , 而实际我们的下板频率为 20Mhz, 因此需要的时间需要乘以 0.5 的系数, 因此下板所需时间约为  $4028 \times 0.5 = 2014\mu s = 2014 \times 10^{-6}s$ , 从而可以计算得到在 C 语言编写的性能验证模型中, 吞吐率为:

$$TP = \frac{504}{2014 \times 10^{-6}} = 0.25025MIPS$$

接下来我们来使用 Mips 编写的性能验证模型来计算 CPU 的性能, 通过在仿真器中查看波形图我们可以得到, 验证模型从启动到运行结束共需要 2295.2us, 共执行 302 条指令, 波形图如下。



由于仿真中时钟频率为  $\frac{1}{100ns} = 10MHz$ , 而实际我们的下板频率为 20Mhz, 因此需要的时间需要乘以 0.5 的系数, 因此下板所需时间约为  $2295.2 \times 0.5 = 1147.6\mu s = 1147.6 \times 10^{-6}s$ , 从而可以计算得到在汇编语言编写的性能验证模型中, 吞吐率为:

$$TP = \frac{302}{1147.6 \times 10^{-6}} = 0.26316MIPS$$

可以看出, 汇编实现的性能验证程序和 C 语言实现的性能验证程序在龙芯 LS132R 上的吞吐率相近, 但汇编的实现略快, 我认为这是因为汇编可以直接访问地址空间中的数据, 比如

本验证程序中我可以通过汇编直接访问 axi\_bram，而 C 语言需要通过栈帧的中转，不仅如此，C 语言还要额外初始化栈空间、中断处理等内容，因此慢一点。

## 9 实验体会

在本次实验中，我们首先完成了数码管实验和 Flash 读取实验，接下来，我们将龙芯 LS132R CPU 内核移植到了 Next4 开发板上，并完成了汇编版点亮 LED,C 语言版点亮 LED 和 C 语言时钟实验。最后，我们分别使用 C 语言和汇编语言实现了性能验证模型，并定量测试了用 C 语言和汇编语言实现的性能验证模型在龙芯 CPU 上的吞吐率。

通过此次实验，我有如下体会和收获。

- 尽管 LS132R 是龙芯系列 CPU 中较为简单的一款，但通过移植龙芯 LS132R CPU 内核，我对于一个 CPU 的复杂性有了更深的认识，意识到了自己在系统结构这方面的功力仍有所欠缺，仍需多加努力。
- 通过本次实验，我学习到了通过 block design 添加模块和 IP 核，并通过 HDL wrapper 生成 verilog 文件的方法。
- 通过本次实验，我学习到了汇编语言的汇编与反汇编，以及 C 语言编译，链接，汇编的全过程，对我计算机科学的“内功”帮助很大。