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 2152118

姓　　名 史君宝

专　　业 计算机科学与技术

授课老师 秦国峰

一、实验环境与实验内容

1. 实验环境：

操作系统：Windows11 专业中文版

软件环境：vivado 2018 版本（未使用之前下载的 2016 版）

开发板：Nexys 4 DDR Artix-7 FPGA 开发板

2. 实验内容：

本次实验我们需要完成龙芯 LS132R-CPU 的移植任务。在移植过程中还需要对程序进行改造，并进行性能验证。

根据秦老师所发布的实验教程可以知道，整个移植任务一共分为 6 个实验。分别是数码管实验、Flash 读取实验、AXI 通信实验、汇编点亮 LED 实验、C 语言点亮 LED 实验和 C 语言时钟实验。

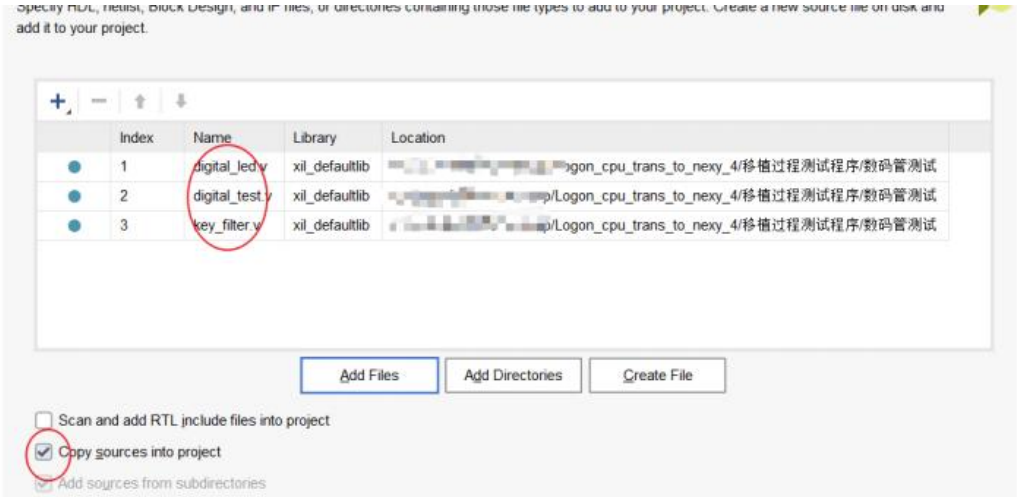
具体的实验步骤在指导书上都已经给出，我们按照教程来做就好了。

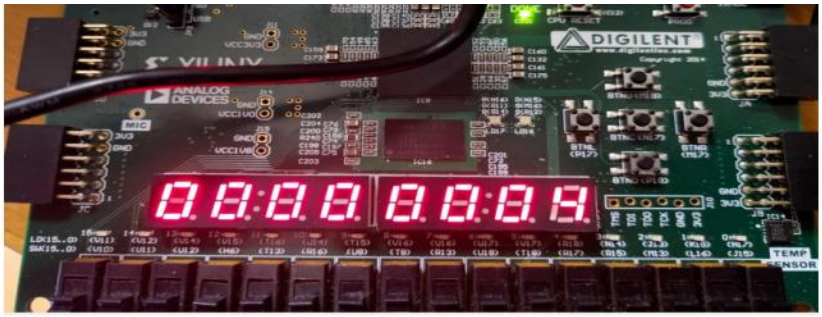
二、实验过程与方法

具体实验过程和方法如下：

（1）数码管实验

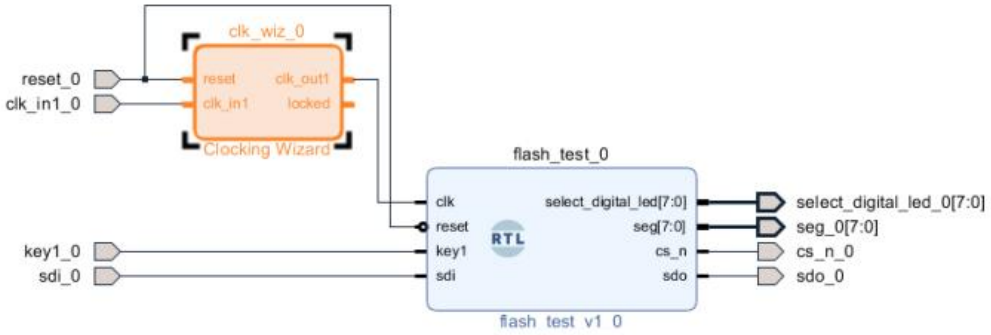
我们根据指导教程的详细步骤进行操作，建立相关的 LED 项目，并完成 bit 流文件的生成，进行实际下板操作。





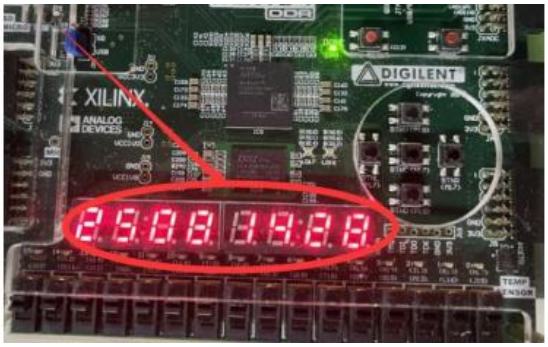
（2）Flash 读取实验

在本实验中，首先向 flash 中烧入已知数据的 bin 文件，FPGA 程序会读取 Flash 中的数据，显示在数码管上就可以知道读取数据是否成功。实际的实验过程中我们会先进行仿真观察波形，观察波形正常之后，最后再进行下板验证。我们按照步骤进行实验，在引入分频模块后，可以构建以下的 Block Design：



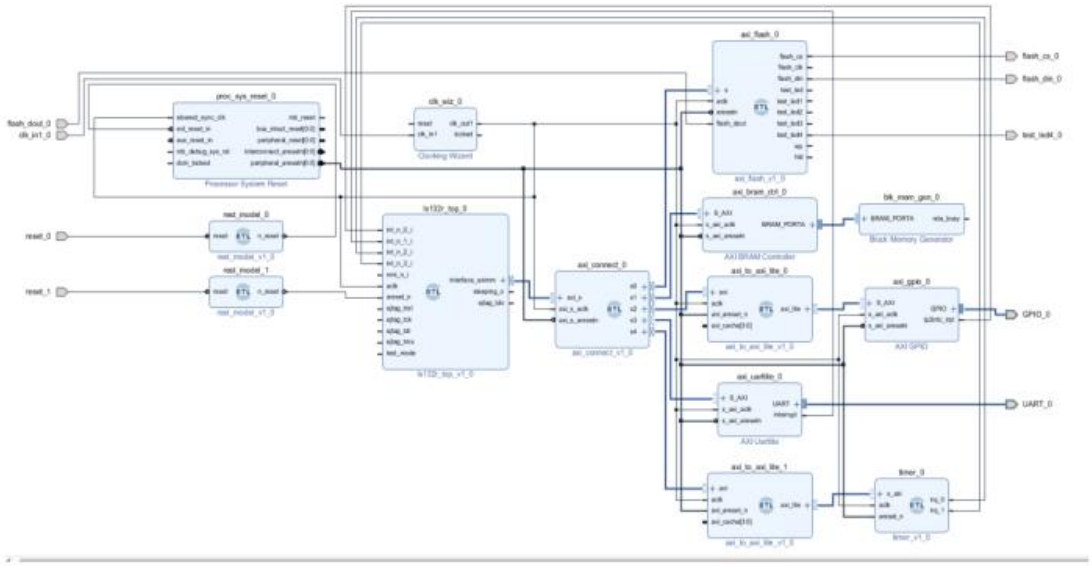
在分频模块中我们设定分频率为 20MHz，并配置 Xdc 约束文件，最后我们连接开发板并添加 Flash 芯片，完成 bin 文件烧录与 bit 文件导入。

按动按键后，每次会自动从 Flash 芯片中取 4 个字节的数据（flash 中一个地址存储 1 个字节）。数码管上会成功显示烧入到 flash 中的数据。具体的下板测试结果如下：



（3）AXI 通信实验

我们依旧按照教程指导进行实验，首先是添加项目文件，我们可以从示例项目中就将相应的项目工程文件移植到对应位置。添加完成所有的文件后，需要设置一个顶层文件，这里我们将 block design 设置为顶层文件，block design 的作用大致来看是帮助生成一个 verilog 模块对应的 Block Designr 如下图：



之后我们还需要进行地址的分配：

Diagram | Address Editor

Cell	Slave Int...	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
s1 (32 address bits : 4G)					
axi_bram_ctrl_0	S_AXI	Mem0	0xC000_0000	32K	0xC000_7FFF
s2 (32 address bits : 4G)					
axi_to_axi_lite_0	axi	reg0	0xD000_0000	64K	0xD000_FFFF
s3 (32 address bits : 4G)					
axi_uartlite_0	S_AXI	Reg	0xD060_0000	64K	0xD060_FFFF
s4 (32 address bits : 4G)					
axi_to_axi_lite_1	axi	reg0	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_lite (32 address bits : 4G)					
axi_gpio_0	S_AXI	Reg	0xD000_0000	64K	0xD000_FFFF
axi_to_axi_lite_1					
axi_lite (32 address bits : 4G)					
timer_0	s_axi	reg0	0xD070_0000	64K	0xD070_FFFF

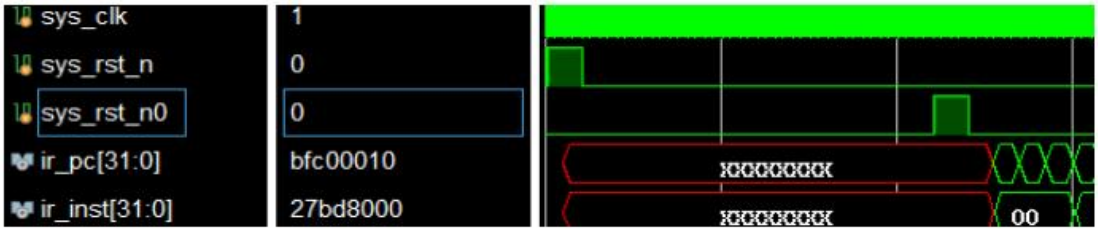
在完成上述步骤之后，我们需要测试一下 AXI 通信是否能够正常使用，我们采用仿真的方式，只要 LS132R 能够正常的读取指令并且执行，就可以认为 axi 与 flash 之间的通信是正常的。之后对于串口和 gpio 的测试也是如此。同时我们还需要对 flash 的读取代码进行稍微的修改，如下所示：

```
/* flash_top.v*/
//这部分是程序下入板子时用
// rdata[31:24]<=temp_rdata[7:0];
// rdata[23:16]<=temp_rdata[15:8];
// rdata[15:8]<=temp_rdata[23:16];
// rdata[7:0]<=temp_rdata[31:24];
。。。。
/**/
//这部分程序，是仿真时候用的，直接读取文件中的数据
//两者之间相互冲突。
 rdata <= memory[(raddr>>2)];00005820
。。。。
reg[31:0] memory[0:2000];
initial
begin
 $readmemh("./haha13.data",memory);
end
```

在对上述的 flash 的读取代码进行修改之后，我们需要进行仿真观察具体的波形来观察是否正常，仿真的代码如下：

```
module test_tb();
reg sys_clk;
reg sys_rst_n;
reg sys_rst_n0;
always #10 sys_clk = ~sys_clk;
initial begin
 sys_clk = 1'b0;    sys_rst_n = 1'b0;   sys_rst_n0 = 1'b0;
 #100   sys_rst_n = 1'b1;
 #2000  sys_rst_n = 1'b0;
 #20000 sys_rst_n0 = 1'b1;
 #2000  sys_rst_n0 = 1'b0;
end
ls132r_soc_wrapper(
 .GPIO_0_tri_io(),  .UART_0_rxd(),  .UART_0_txd(),
 .clk_in1_0(sys_clk),   .flash_clk_0(), .flash_cs_0(),
 .flash_din_0(),    .flash_dout_0(),
 .reset_0(sys_rst_n), //总线先 reset,
 .reset_1(sys_rst_n0) //CPU 后 reset,也许您的设计有所不同注意
);
endmodule
```

仿真后的波形图，如下：



我们观察到了上述波形，可以知道我们当前的 AXI 实验已经顺利完成，axi 和 flash 之间的通信正常。

（4）汇编点亮 LED 实验

本实验是在上个实验的基础上进行，在本次实验中我们需要将时钟分频到 20MHz 左右。在实验中我们首先需要进行分频处理，并正常生成 bit 文件，之后我们需要修改 flash_top 文件以改变数据读取方式。现在应该正常生成 bit 流文件，这是最终的 fpga 程序了。

现在我们需要开始生成烧入 flash 所需要的.bin 文件，找到 led_asm.s 文件所在的对应目录下，然后进行 cmd 命令的输入。输入的内容如下：

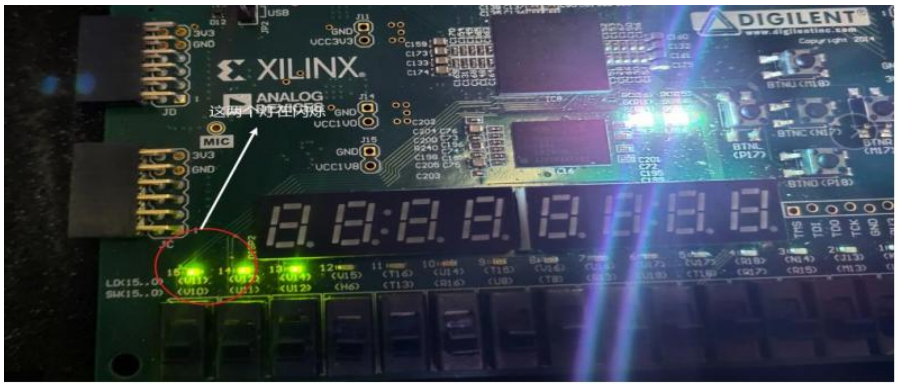
```
..\\bin\\mips-mti-elf-as.exe -32 -mips32 led_asm.s -o led_asm.o

..\\bin\\mips-mti-elf-ld.exe -T mytest.ld led_asm.o -o led_asm.om

..\\bin\\mips-mti-elf-objcopy.exe -O binary led_asm.om led_asm.bin

/*反汇编与仿真结合，可能会使检查错误容易一些*/
..\\bin\\mips-mti-elf-objdump.exe -D led_asm.om > led_asm.asm
```

下板之后的最终实验结果如下：



（5）C 语言版点亮 LED 实验

我们需要了解 gpio 的 IP 核的相关文档，GPIO_DATA 就是引脚输出值的寄存器，输出 0 对应引脚就是低电平，输出高对应引脚就是高电平。如果是输入引脚的话，读取的就是对应引脚上的值。GPIO_TRI 用于定义输入还是输出。GIER

意思是是否打开全局中断，IP_IER 意思是是否使能对应通道中断，IP_ISR 是对应通道的中断状态，如果您往 IP_ISR 中写零，就可以清除相应中断。而 GPIO_DATA 寄存器地址 0xD000_0000，GPIO_TRI 寄存器地址 0xD000_0004。

我们可以观察到 main 函数的内容为：

```c
#include "../include/minicrt.h"
#include "../include/system.h"
#include "../include/gpio.h"
int main(void)
{
unsigned int* gpio_tri_addr = GPIO_TRI_ADDR;
unsigned int* gpio_data_addr = GPIO_DATA_ADDR;
*gpio_tri_addr = 0x0; //地址 0xD000_0004 写入 0x0
while(1){
*gpio_data_addr = 0xffffffff; //地址 0xD000_0000 写入 0xffffffff
udelay(200000);
*gpio_data_addr = 0xff3fffff;
udelay(200000);
}
return 0;
}
```

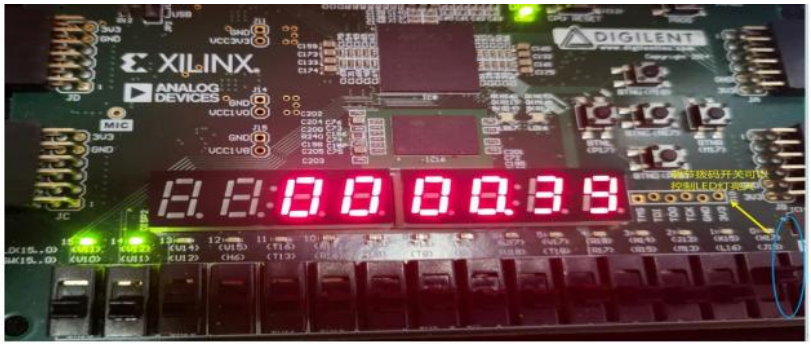
按照教程操作，我们可以观察到下板结果：

（6）C 语言版时钟实验：

在本次实验中，我们按照指导书上的时钟程序代码进行实验：

```
int main(void)
{
set_gpio_tri(0xffff0000,false);//设置低 16 位为输出。
set_gpio_tri(0x02000000,true); //设置第 25 位为输入，L16
。。。。
while(1){
send = get_seconds();
。。。。
digital_led(5,(hour /10)%10);
udelay(3000);
flag = read_gpio(0x03000000); //测试 GPIO 读功能是否正常
if(flag !=0){
set_led(true);
}else{
set_led(false);
}
}
return 0;
}
```

下板后的实验结果如下：



三、程序修改说明

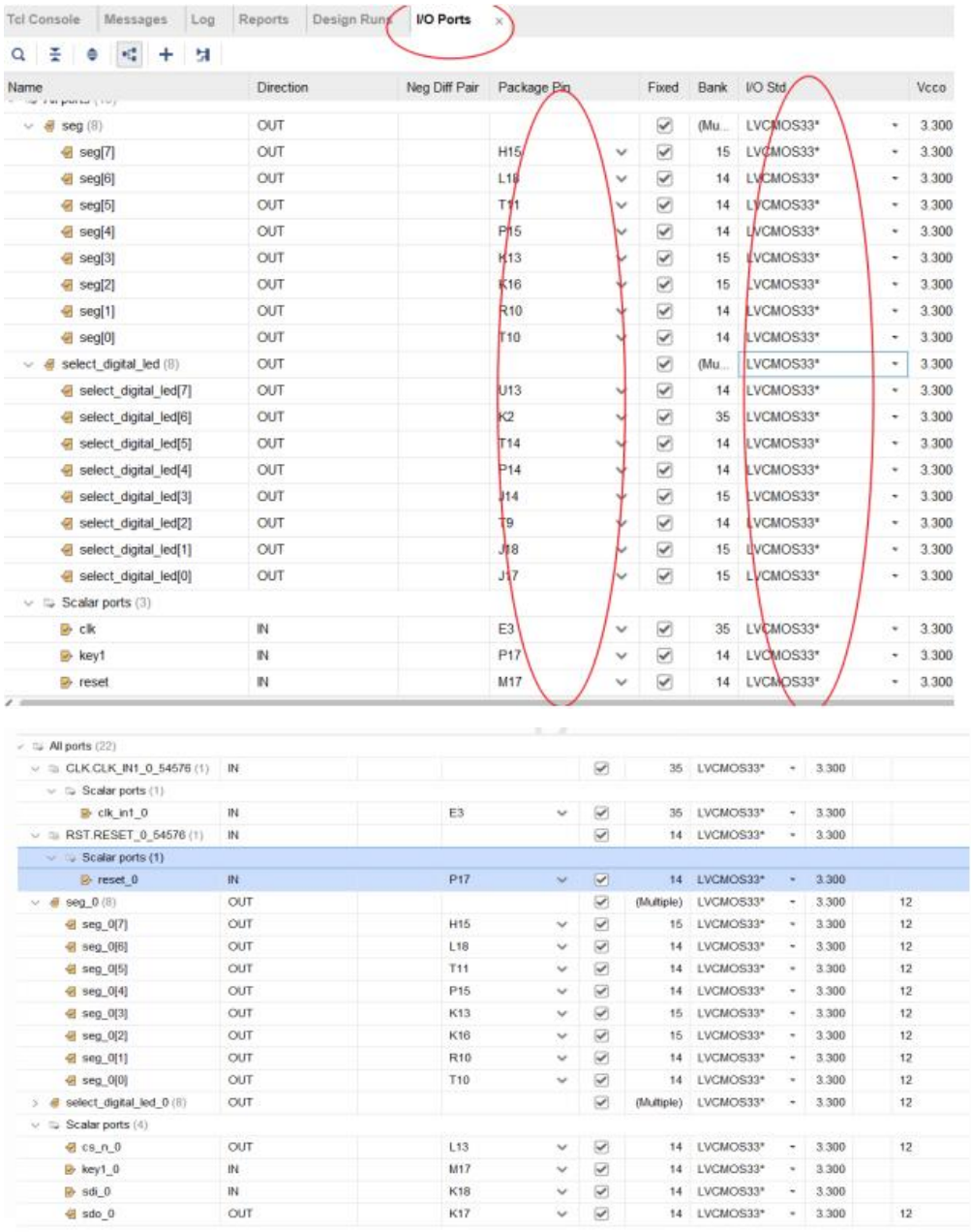
尽管教程中的内容已经非常的翔实，我们根据教程进行操作，就能完成本次实验。同时也给了我们具体的实例项目工程文件，但是为了适应代码中相应的需求，我们仍然需要根据实验要求进行了相应的程序修改。首先就是注释掉 LS132R-CPU 模块中的硬件配置参数，同时取消了对代码中 INST SRAM 和 DATASRAM 的地址空间分配，并注释掉后续源码中对这两部分地址空间的操作，

使得我们的程序可以在自己的板子上运行。

通过上述的程序修改，我们能够顺利地完成整个实验。

四、约束文件修改说明

在本次实验中的约束文件配置过程如下：



在 LS132R 处理器核的硬件初始化过程中，非常重要的一步是将所有的电路状态机重置到一个安全的初始状态。这个过程与核心软件的运行无关，主要通过 reset_0 和 reset_1 管脚来实现。

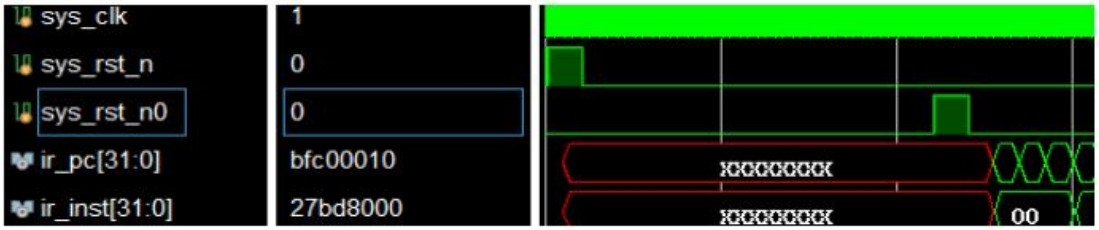
在 N4 开发板的 XDC 文件中，这些管脚已经 进行了配置，以方便 LS132R CPU

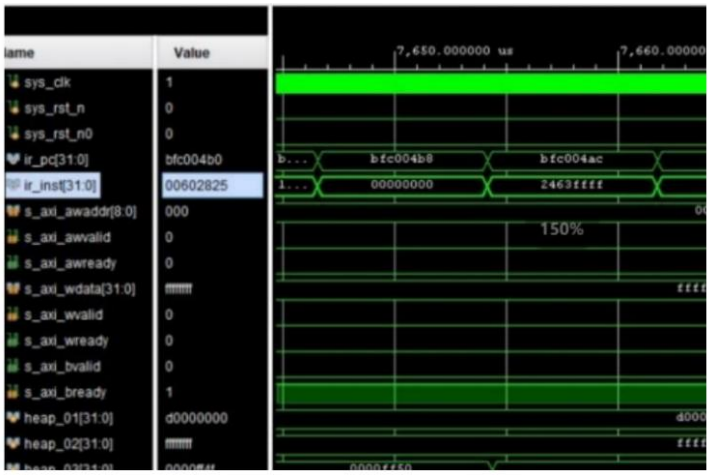
的移植与使用。reset_0 管脚用于总线重置，而 reset_1 管脚用于 M1CPU 的重置。通过将这些管脚连接到适当的引脚上，并在适当的时机将其置高或置低，可以触发重置过程。当重置信号被激活时，所有的电路状态机会被强制重置到一个安全的初始状态，以确保处理器核在启动时处于可控的状态。

这个硬件初始化过程是移植 LS132R CPU 的重要一步，它确保了处理器核在开始执行软件代码之前的可靠性和可预测性。通过正确配置和使用 reset_0 和 reset_1 管脚，我们可以确保 LS132R CPU 在目标 FPGA 开发板上能够正常运行，并且具备可靠的启动和重置功能。

五、仿真分析

整个实验中的仿真波形如下：





六、性能验证数学模型及算法程序

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;

$$c[i]=\begin{cases} a[i], & 0\le i\le 9 \\ a[i]+b[i], & 10\le i\le 29 \\ (a[i]*b[i])\ll 1, & 30\le i\le 49 \end{cases}$$

$$d[i]=\begin{cases} b[i]+c[i], & 0\le i\le 9 \\ a[i]*c[i], & 10\le i\le 29 \\ c[i]*b[i]/(d[i-1]\gg 1), & 30\le i\le 49 \end{cases}$$

七、性能验证程序下板测试过程与实现

在实验中我们通过显示管输出计时结果（微秒级），从而完成对顶点运算性能的测试。下面是我们的性能验证程序：

MIPS 汇编程序：

```
    .file
    "temp.c"
    .text
    .globl set_gpio_tri
    .type set_gpio_tri, @function
set_gpio_tri:
.LFB23:
    .cfi_startproc
    endbr64
    ret
    .cfi_endproc
.LFE23:
    .size set_gpio_tri, .-set_gpio_tri
    .globl read_gpio
    .type read_gpio, @function
read_gpio:
.LFB24:
    .cfi_startproc
```

```
    endbr64
    movl $0, %eax
    ret
    .cfi_endproc
.LFE24:
    .size read_gpio, .-read_gpio
    .globl digital_led
    .type digital_led, @function
digital_led:
.LFB25:
    .cfi_startproc
    endbr64
    ret
    .cfi_endproc
.LFE25:
    .size digital_led, .-digital_led
    .globl udelay
    .type udelay, @function
udelay:
.LFB26:
    .cfi_startproc
    endbr64
    ret
    .cfi_endproc
.LFE26:
    .size udelay, .-udelay
    .globl get_seconds
    .type get_seconds, @function
get_seconds:
.LFB27:
    .cfi_startproc
    endbr64
    movl $0, %eax
    ret
    .cfi_endproc
.LFE27:
    .size get_seconds, .-get_seconds
    .globl main
    .type main, @function
main:
.LFB28:
    .cfi_startproc
    endbr64
    movl $50, %eax
```

```
.L7:
    subl $1, %eax
    jne .L7
.L8:
    jmp .L8
    .cfi_endproc
.LFE28:
    .size main, .-main
    .ident "GCC: (Ubuntu 11.4.0-1ubuntu1~22.04) 11.4.0"
    .section
    .note.GNU-stack,"",@progbits
    .section
    .note.gnu.property,"a"
    .align 8
    .long 1f - 0f
    .long 4f - 1f.long 5
0:
    .string "GNU"
1:
    .align 8
    .long 0xc0000002
    .long 3f - 2f
2:
    .long 0x3
3:
    .align 8
4:
```

C 语言程序：

```c
#include <stdio.h> // 补全可能缺失的头文件
#include <stdbool.h>
#define M 50
int main(void) {
set_gpio_tri(0xffff0000, false); // 设置低 16 位
为输出。
set_gpio_tri(0x02000000, true); // 设置第 25 位为
输入，L16
```

```
int a[M], b[M], c[M], d[M];
a[0] = 0, b[0] = 1;
int start_time = get_seconds();
for (int i = 0; i < M; ++i) {
if (i <= 9) {
c[i] = a[i];
d[i] = b[i] + c[i];
}
if (i >= 1 && i <= 9) {
a[i] = a[i - 1] + i;
b[i] = b[i - 1] + 3 * i;
}
if (i >= 10 && i <= 29) {
c[i] = a[i] + b[i];
d[i] = a[i] * c[i];
}
if (i >= 30 && i < M) {
c[i] = (a[i] * b[i]) << 1;
d[i] = (c[i] * b[i]) / (d[i - 1] >> 1);
}
}
```

```
int end_time = get_seconds();
int compute_time = end_time - start_time;
unsigned int flag;
while (1) {
flag = read_gpio(0x03000000);
if (flag != 0) {
digital_led(0, compute_time % 10);
udelay(1000);
digital_led(1, (compute_time / 10) % 10);
udelay(1000);
digital_led(2, (compute_time / 100) %
10);
udelay(1000);
digital_led(3, (compute_time / 1000) %
10);
udelay(1000);
```

```
} else {
digital_led(0, c[M - 1] % 10);
digital_led(4, d[M - 1] % 10);
udelay(1000);
digital_led(1, (c[M - 1] / 10) % 10);
digital_led(5, (d[M - 1] / 10) % 10);
udelay(1000);
digital_led(2, (c[M - 1] / 100) % 10);
digital_led(6, (d[M - 1] / 100) % 10);
udelay(1000);
digital_led(3, (c[M - 1] / 1000) % 10);
digital_led(7, (d[M - 1] / 1000) % 10);
udelay(1000);
}
}
return 0;
}
```

在完成上述的LS132R CPU的移植与程序编写后，我们按照之前的C语言时钟实验的方法，修改toolchain/build/src目录下的main.c文件，添加性能测试的代码，并在显示管上输出程序的运行时间。通过命令行窗口执行make指令编译程序，生成bin文件，观察并记录输出结果。

对于汇编程序，先在Mars程序中完成MIPS汇编，然后仿照之前汇编程序点亮LED实验的方法，将mips文件放置在toolchain/build目录下，并使用指令生成bin文件。接下来，将bin文件进行flash烧录，并下载bit文件，记录数据以进行性能比较。通过对LS132R CPU性能测试的实验，我们可以进一步了解和评估其性能表现，并据此做出相应的优化和调整。

八、实验体会

在进行C语言和MIPS指令编写性能验证程序的实验中，我获得了一些宝贵的体会：

（1）编程体验的多样性：通过编写性能验证程序，我体验到了C语言和MIPS指令编程的不同特点和优势。C语言作为高级语言，提供丰富的库函数和易读易写的特性，使得编程变得更加便捷。相比之下，MIPS指令更接近底层硬件，需要更多的细节和低级操作。这种多样性的编程体验让我更全面地了解不同层次的编程方式。

（2）性能与效率的权衡：在编写性能验证程序时，我需要平衡程序的运行时间和MIPS值。C语言编写的程序通常具有较高的抽象级别和易用性，但在某些情况下可能会带来性能损失。相比之下，使用MIPS指令编写的程序更有利于性能优化。在实验过程中，我需要权衡性能与编程效率之间的关系，选择适合需求的编程方式。

（3）硬件和软件的协作：在进行性能验证程序实验时，我必须结合硬件平台和编译器等软件工具。不同的硬件平台和编译器优化策略可能会影响程序的性能表现。因此，我需要了解所用硬件和软件工具的特点，进行适当的配置和调整，以获得准确可靠的性能测试结果。

（4）综合考量的性能评估：在实验中，我需要综合考虑多个因素来评估CPU的定点运算性能。除了运行时间和MIPS值外，还需要考虑编译器优化、指令集支持、程序复杂度等因素。这种综合考量有助于更全面地评估CPU的性能，帮助

我做出准确的结论和分析。

总的来说，通过编写 C 语言和 MIPS 指令的性能验证程序，我不仅加深了对不同编程方式和指令集的理解和体验，还提升了在性能优化和编程效率方面的能力。这次实验经历对于未来深入研究和应用 CPU 性能优化具有重要的参考价值。这个实验让我更加熟悉硬件和软件协作的重要性，同时也培养了我解决问题和挑战的能力。