

单片机系统篇

第 5 章

本篇涵盖单片机系统微处理器核、片上系统虚拟平台和智能硬件三大模块。快速入门学习可采用“做中学”方法,实验日志方式记录“基本实验、问题及解答、主动实验”。综合实践部分作为拓展实践适合兴趣小组项目驱动学习。

5.1 AVR 架构

5.1.1 基本知识点

区分微控制器 CPU 核、单片机芯片、开发板;建立处理器基本模型,说明处理器运行过程。深入 Harvard 结构的流水线概念,能设计满足单周期并行取指与指令执行的时序节拍电路。VHDL 模块编程、模块实例化与多文件编程;建立状态机,实现时序发生电路逻辑与逻辑仿真验证;配置 DDA-III A 便携板的 Cyclone II 型芯片与硬件验证。

5.1.2 实验设备

DDA-III A 便携板、Quartus II Web 9.0。

5.1.3 实验概述

本教程处理器以 ATmega328P 为原型,它属于 Atmel 厂家一款 CMOS 低功耗 8 位 AVR 增强型 RISC 架构微控芯片。1.8~5.5V 宽幅供电同时带来近 1MIPS/MHz(最高 20MHz)处理速度;丰富的存储资源,如 32KB 带 ISP 的 Flash、1KB 的 EEPROM、2KB 的 SRAM;片内接口类型充足,23 个通用 I/O、3 个 Timer、6 通道 10 位 ADC、USART、TWI、SPI 各一,带内部时钟 WatchDog 等。

1. AVR CPU

AVR CPU 采用了 Harvard 结构,具有独立的数据和指令总线,如

图 5-1-1 虚线框内部分所示。8 位数据总线处理 1 字节,16 位指令总线一次取 1 字。

32 个 8 位通用寄存器构成寄存器堆文件,支持单时钟周期的 ALU 操作。16 位的 X、Y、Z 寄存器可作间接寻址寄存器指针以寻址数据空间或作为指针备份。

ALU 支持寄存器间、寄存器与立即数、单寄存器值与位的算术和逻辑运算,同时运算结果给出标志判断改变状态寄存器 SREG 相应比特位数值。运算完成之后状态寄存器的内容得到更新以反映操作结果。

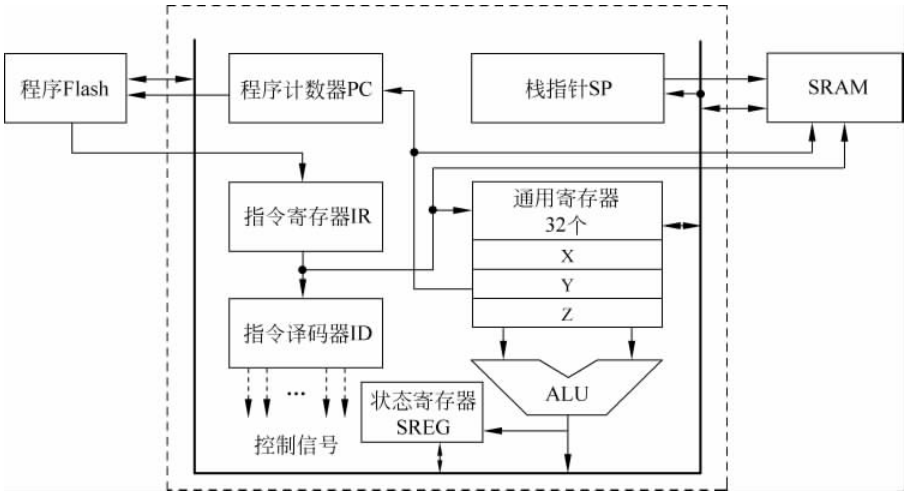


图 5-1-1 处理器框图

状态寄存器 SREG 是一个 8 位标志寄存器,如图 5-1-2 所示,用来存放指令执行后的有关状态和结果的标志,各位状态通常是在指令执行过程中自动产生的,但也可以由用户根据需要用专用指令加以改变。

位	7	6	5	4	3	2	1	0	
	I	T	H	S	V	N	Z	C	SREG
读/写	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
初始值	0	0	0	0	0	0	0	0	

图 5-1-2 SREG

I 位是全局中断使能位,T 位是位拷贝存储,H 位是半进位标志,C 位是进位标志,Z 位是零标志,符号位 S。

2. 并行取指与指令运行

处理器是处理数据的器件。AVR CPU 处理一条典型 ALU 操作指令过程,首先改变 PC 值来访问程序存储器 IF,然后经 ALU 运算执行 EX。IF 与 EX 运行通过流水线并行执行,如图 5-1-3 所示。T1 完成第 1 指令 IF,至 T4 完成第 3 指令 EX 与第 4 指令 IF。每个周期都能完成一条指令,理想性能达到 1MIPS/MHz。

在图 5-1-4 所示的一个完整 ALU 操作 EX 时间 T1 内,CPU 完成操作数读取寄存器,送往 ALU 运算,最后结果写回目的寄存器。

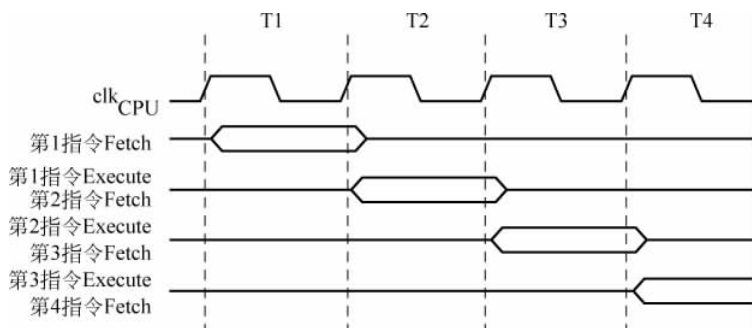


图 5-1-3 并行 IF 与 EX

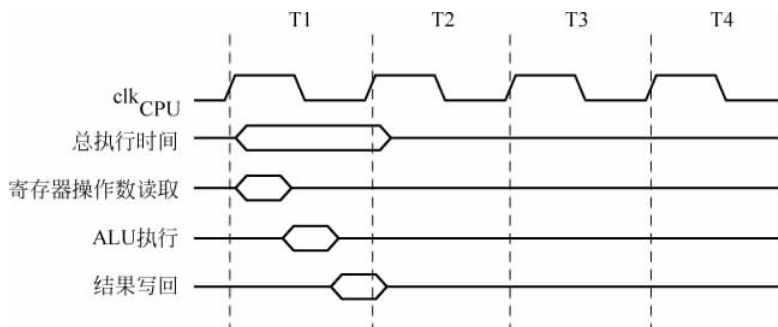


图 5-1-4 ALU 操作

5.1.4 实验内容

BE: 时序节拍电路

内容：深入 Harvard 结构的流水线概念，设计满足单周期并行取指与指令执行的时序节拍电路，包含启动、单拍等控制信号，完成功能波形图，如图 5-1-5 所示。

分析：图 5-1-5 显示 clock 时钟、reset 复位（高电平有效）属于电路必须系统控制信号。

qd 启动（低电平有效）、tj 停机（高电平有效）、dp 单拍周期（高电平有效）用于控制节拍产生模式。

停止模式：t1、t2、t3 一直处于无波形输出。

连续输出模式：qd 启动后，tj 与 dp 无效，循环顺序输出 t1、t2、t3，如第一节拍周期连续产生一组 t1、t2、t3，而 t3 结束也是下一新节拍 t1 开始，再又是 t2，如此往复。

停机模式：连续输出模式下，遇到 tj 有效，dp 无效，发生电路一直输出当前的节拍，如第二节拍周期 t3 节拍输出时遇上了 tj，延长了 t3 输出时间。

单拍周期模式：连续输出模式下，遇到 tj 无效，dp 有效，发生电路输出一组 t1、t2、t3，便进入停止模式。

EX 单周期，如果采用 t1、t2、t3 节拍正好完成寄存器堆取操作数 OF、运算 ALU、写寄存器。

如果不用节拍 t1、t2、t3，将 t1 节拍归到流水线前一级间寄存器操作，t3 节拍归到后一级间寄存器操作，节拍 t2 操作就只能用组合逻辑，适合低速运行情况。

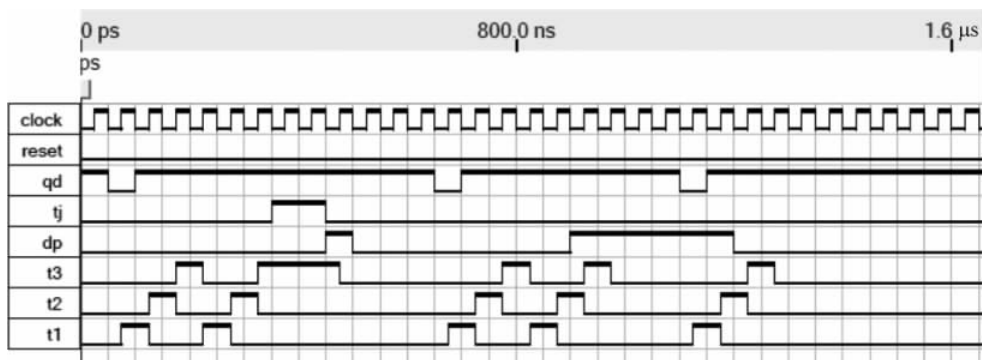


图 5-1-5 时序发生信号波形

5.1.5 思考题

AVR CPU 芯片由系统时钟直接驱动指令运行,其他如定时器可以通过分频设置调慢运行速度。设计一款时钟选择电路,功能参考见表 5-1-1,并分析它与节拍电路时钟控制方式的区别。

表 5-1-1 时钟信号选择

CS02	CS01	CS00	功能说明
0	0	0	无时钟源
0	0	1	$clk_{i/o}$
0	1	0	$clk_{i/o}/8$
0	1	1	$clk_{i/o}/64$
1	0	0	$clk_{i/o}/256$
1	0	1	$clk_{i/o}/1024$
1	1	0	T0 引入的外部时钟源,下降沿有效
1	1	1	T0 引入的外部时钟源,上升沿有效

5.2 AVR 汇编

5.2.1 基本知识点

认识 AVR 指令如何操作, AVR 寄存器约定、指令类型与寻址、常用指令;安装 Atmel Studio,建立工程、目标配置、Hex 输出,调试基础实例验证;掌握 AVR 汇编程序模板,并能汇编语言编程实现简单算法。

5.2.2 实验设备

Atmel Studio 6.1。

5.2.3 实验概述

1. 工作环境安装

选择 astudio61net.exe(WIN7 WIN8.1 64bits)安装 Atmel studio 6.1。首先,接受协议安装.NET Framework 4.0 和 Visual Studio 2010 Shell 组件。继续安装 Atmel USB Driver,过程中系统弹出组件安装许可,选同意安装。最后,弹出 Atmel Studio 6.1 安装向导,单击 Next 按钮,选择好安装目录,如:d:\Atmel\Atmel Studio 6.1,最后单击 Finish 按钮,完成安装。

2. AVR 指令

AVR 指令总线宽度为 16 位,一条指令的长度大多为 16 位,还有少部分为 32 位。操作码长度不固定,是全 16 位还是其中某几位依指令而不同。AVR 的指令的一般格式如图 5-2-1 所示。

操作码	操作数 1 或操作数地址	操作数 2 或操作数地址
-----	--------------	--------------

图 5-2-1 AVR 指令格式

指令执行时 CPU 先要依 PC 地址获取指令,指令译码后直接或间接得到操作数,最后运算操作结果也按指令要求地址保存于储存器或寄存器。参与运算数据的获取方式是指令寻址方式。

下面介绍部分 AVR 的部分指令识读及示例。

(1) 二进制数值表示操作码。d dddd 表示目的寄存器 Rd 地址,r rrrr 表示寄存器 Rr 地址,范围(0,31),对应 32 个通用寄存器。

ADD 0000 11rd dddd rrrr 单字寄存器寻址

例: add r2,r1 ;将 r1 的值累加到 r2。

(2) KKKK KKKK 表示常量,范围(0,255)。

ORI 0110 KKKK dddd KKKK 单字寄存器寻址

例: ori r13,\$F0 ;设置 r13 寄存器的高 4 位全为 1。

(3) AA AAAA 表示 I/O 寄存器地址,范围(0,63),对应 64 个 I/O 寄存器。

IN 1011 0AA d dddd AAAA 单字 I/O 寄存器寻址

例: in r23,\$15 ;读取端口 15 数据送 r23。

(4) 带偏移或不带偏移的间接寻址指针(X、Y、Z)寄存器和栈指针寄存器(SP)也可指向数据或程序空间。

LD 1001 000d dddd 1101 单字数据空间 X 寄存器间接寻址

例: ld r0,X+; 数据区中 X 寄存器指定地址的数据装载到寄存器 r0 中(X 值加 1)。

LPM 1001 000d dddd 0100 单字数据空间 Z 寄存器间接寻址

例: lpm r14,Z ; Z 指向的程序空间内容送 R14。

PUSH 1001 001d dddd 1111 单字数据空间 SP 寄存器间接寻址

例: push r13; 保存 r13 中的数据到堆栈。

POP 1001 000d dddd 1111 单字数据空间 SP 寄存器间接寻址

例: pop r13; 还原 r13。

(5) 数据存储空间地址 kkkk kkkk kkkk kkkk 表示, 支持 16 位。

LDS 1001 000d dddd 0000
kkkk kkkk kkkk kkkk 双字数据空间直接寻址

例: lds r1, \$FF00; 数据空间中位于 \$FF00 的数据装入 r1。

(6) 程序空间直接与相对寻址, 决定新的 PC 地址。

JMP 1001 010k kkkk 110k
kkkk kkkk kkkk kkkk 双字程序空间直接寻址

例: jmp \$0034; 无条件跳转至程序空间 \$0034。

RJMP 1100 kkkk kkkk kkkk 单字程序空间相对寻址

例: RJMP \$0010; 跳转至程序空间 PC + \$0010 + 1。

5.2.4 实验内容

BE: CPU 编译器

内容: 新建 C 工程, 能以 ATmega328p 目标芯片, 运行 AVR CPU 模拟器, 设置断点观察 Processor、Memory、Register 相关信息, 编译生成或查看汇编语言源代码。查看工程目录下 makefile 文件内容, 说明 AVR GCC 编译过程。

分析:

(1) 加法运算函数参考代码。

```
//返回两数和
char AddNum(char a, char b)
{ return a + b;
}
int main(void)
{
    //调用 AddNum 函数
    char c = 0;
    c = AddNum(12, 45);
}
```

注意: MCU 必须的头文件统一用 #include <avr/io.h>, 它已经包含了所有的单片机头文件。AS6 汇编嵌入编程中可使用“asm(“xxx”);”格式或者用小写的“sei();”(中断开关位操作)。

(2) 查看 Disassembly。

设置 C 编译器优化选项。单击快捷工具图标 Atmega328P, 或选择菜单 Project → Properties(Alt + F7), 打开工程属性窗, 在 Tools 标签页, 设置 Selected Debugger/

Programmer 为 Simulator。在 Toolchain 标签页, AVR GNU C Compiler 设置 Optimization Levels 为 None, 如图 5-2-2 所示。

程序 Build, 菜单 Build→Build。

错误信息栏无错误后, 按 Alt+F5 进入 Debug 调试, 打开 Disassembly 窗口。

黄色箭头指示下一步程序执行处, 如 Main() 入口所在, 单步调试 F11 执行。

Processor 窗口, 观察 AVR CPU 运行状况、时钟频率(可修改)、寄存器值等。

Locals 窗口, 观察变量 name 和 value。

Memory1 窗口, 选择 data IRAM 可以观察数据存储(栈 stack pointer: 0x08FD)。

I/O 窗口, 观察输入输出值, 这里也可以修改。

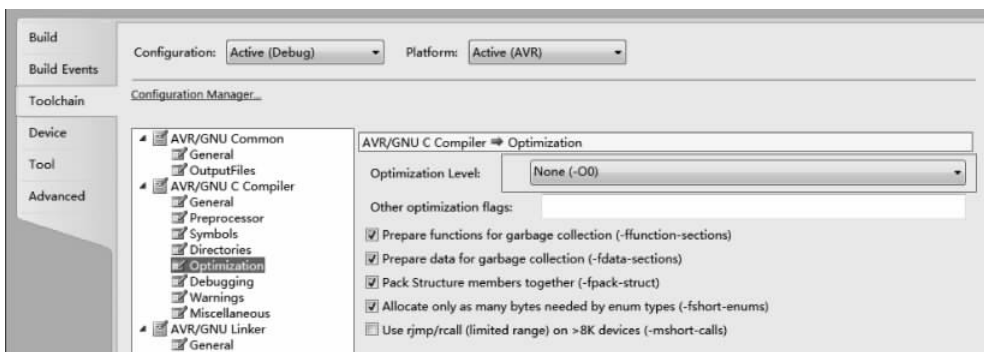


图 5-2-2 编译器优化设置

Disassembly 窗口, 右击快捷菜单, 可勾选 Show code byte 显示机器语言字节码。

断点添加, F9 或右击快捷菜单。

Solution Explorer 窗口, Output Files 列表显示输出文件. elf, . lss, . hex 等。

(3) 根据 Disassembly 显示, 表 5-2-1 记录程序执行部分。

表 5-2-1 汇编代码

(代码)

BE: CPU 模拟器

内容: 利用 AVR CPU 模拟器 Debug 分析汇编语言源代码, 掌握 AVR 指令格式, 表 5-2-2 记录上一实验编译中出现过指令的机器码、操作数、功能操作、时钟数、影响标志。

表 5-2-2 指令详细说明

助记符	操作数	指令格式	功能	影响标志位	时钟周期数
例: ADD					
...					

BE: AVR 循环与分支指令

内容：利用前两小节实验方法分析 C 语言程序对应 AVR 汇编语言格式,表 5-2-3 中说明流程控制中条件分支、循环处理、函数调用的实现方法。

分析：for 与 if 参考代码如下,左边代码 for 循环体调用 MySub 函数,右边 if 代码判断 a 变量值进行条件选择执行 MySub1 或 Mysub2。

这里 MySub、MySub1、MySub2 函数体里均是空操作,可以适应编译器及处理器约束要求,方便通过 CPU 模拟器和汇编代码熟悉程序运行行为。

<pre>//参考代码: for //定义 MySub 函数 void MySub() { //无操作 } int main(void) { char i; for (i = 0;i < 10;i++) { MySub(); } }</pre>	<pre>//参考代码: if //定义 MySub1 函数 void MySub1() { //无操作 } //定义 MySub2 函数 void MySub2() { //无操作 } //定义 MySub3 函数 void MySub3() { //无操作 } int main(void) { char a = 123; if (a > 100) { MySub1(); } else if (a < 50) { MySub2(); } else { MySub3(); } }</pre>
---	--

表 5-2-3 流程控制说明

(代码)

5.2.5 思考题

采用延时函数,汇编编程实现算法案例,如 LED 闪烁来表示 SOS 求救信号;注意分析 AVR 汇编语言格式,掌握 AVR 指令的寻址方式和寻址空间,记录函数调用过程中栈变化、

参数传入与值返回。

说明：

(1) 汇编代码必备头文件，例：目标芯片 ATmega328P，开头写“. INCLUDE "m328Pdef.inc"；CPU 配置文件。”

(2) CPU 配置文件.inc 一般位于程序安装路径位置..\Atmel\Atmel Toolchain\AVR Assembler\Native\2.1.39.232\avrassembler\include。

(3) 循环体内，可设置一处断点，同时注释掉延时子程序；RCALL DELAY，再利用 STEP over，可以加快调试，便于在 I/O view 中观察 LED 引脚对应比特位值交替变化。

5.3 数据存储

5.3.1 基本知识点

认识指令和数据存储器件，深入 AVR 结构存储空间。能够定制 IP 核实现 ROM 逻辑与功能仿真验证；VHDL 编程设计三端口 RAM 类型的 Regfiles 电路逻辑与功能仿真验证；配置 DDA-III A 便携板的 Cyclone II 型芯片与硬件验证。

5.3.2 实验设备

DDA-III A 便携板、Quartus II Web 9.0。

5.3.3 实验概述

AVR 结构具有两个主要的存储器空间：数据存储器空间和程序存储器空间。另外还有保存数据的 EEPROM 存储器。

ATmega328P 具有 32KB 的在线编程 Flash，用于存放程序指令代码，如图 5-3-1 所示。

ATmega328P 的 SRAM 空间组织结构也是平面型，如图 5-3-2 所示。它包含全部 32 个通用寄存器、64 个 I/O 寄存器及 2048 个字节的内部数据 SRAM，可以通过常见寻址模式进行访问。起始的 96 个地址为寄存器文件与 I/O 存储器，接下来 2048 个字节归内部数据 SRAM。

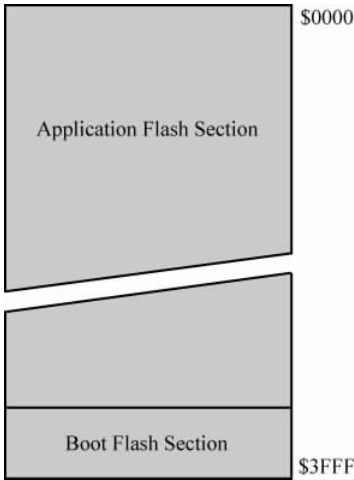


图 5-3-1 程序存储空间

5.3.4 实验内容

BE: ROM 与 RAM

内容：深入 AVR 结构存储空间，能设计满足 CPU 运行的 RAM 与 ROM。

(1) 记录程序 ROM 的大小 32KB、格式 16K×16、地址范围，HDL 设计 ROM 实现读操作。

(2) 记录数据 RAM 的大小 2KB、格式 2K×8、地址范围，HDL 设计 RAM 实现读/写操作。

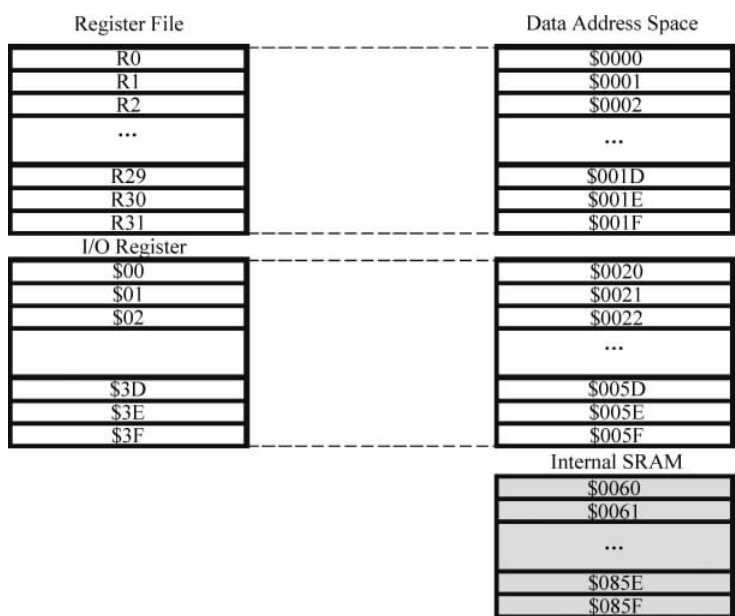


图 5-3-2 数据存储空间

分析: AVR 指令为 16 位或 32 位,故而 Flash 组织成 $16\text{K} \times 16$ 位的形式。数据存取访问时间如图 5-3-3 所示,同一 CLK 周期,RAM 可读可写,读还需要两路用于读取寄存器 Rd 和 Rr 做运算。

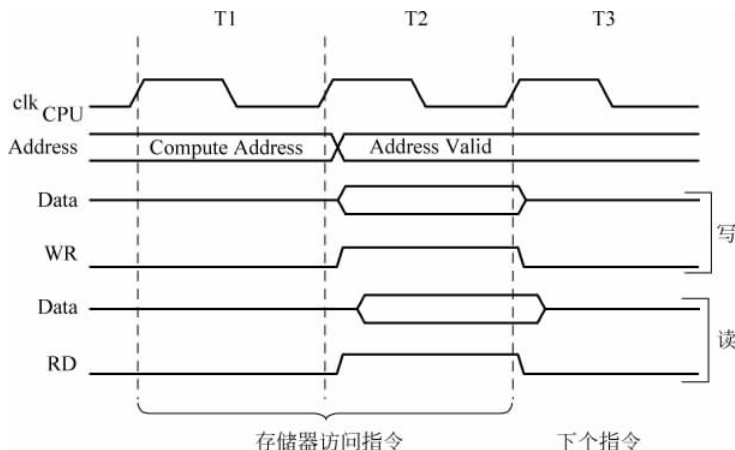


图 5-3-3 RAM 读写

波形图分析时利用 Register 类节点来观察通用寄存器,而不是通过访问内存。VHDL 编程时可借用“type regfiles is array”快速生成多个寄存器。

注意: 警告信息“warning: Inferred RAM node "<name>" from synchronous design logic. Pass-through logic has been added to match the read-during-write behavior of the original”。

解决方法: 选择 Settings→Analysis & Synthesis Settings→More 选项中关闭 Auto RAM Replacement logic 的设置。

5.3.5 思考题

FPGA 实现 RAM 时,一种方法利用 Quartus 软件 LPM 定制,另一种方法使用 HDL 纯文本的描述方式设计生成同样功能的 RAM 块,查看编译报告,分析比较两种方法资源利用情况。

如果使用 Cyclone II 器件,编译过程中可能报错“Error: M4K memory block WYSIWYG primitive...”。

解决方法:选择 ASSIGNMENTS→Settings→Analysis & Synthesis Settings→Default Parameters 页面,Parameter 选项中 Name 栏中输入 CYCLONEII_SAFE_WRITE,Default setting 栏中输入 VERIFIED_SAFE 并单击 Add 按钮添加配置。

5.4 微处理器软核

5.4.1 基本知识点

认识状态机及流水线两种模型;能设计单周期并行处理 ORI 单指令的两级流水线式 AVR 微处理器软核;ALU 分类操作划分单周期状态和操作;结合 AVR 架构单总线特点为各模块完成时序分配;VHDL 编程完成微处理器逻辑与功能仿真验证。

5.4.2 实验设备

Quartus II Web 9.0。

5.4.3 实验概述

经典 MIPS 架构 5 级流水线,包括取指 IF、译码 ID、运算 ALU、存储 MEM、写回 WB 过程,如图 5-4-1 所示。

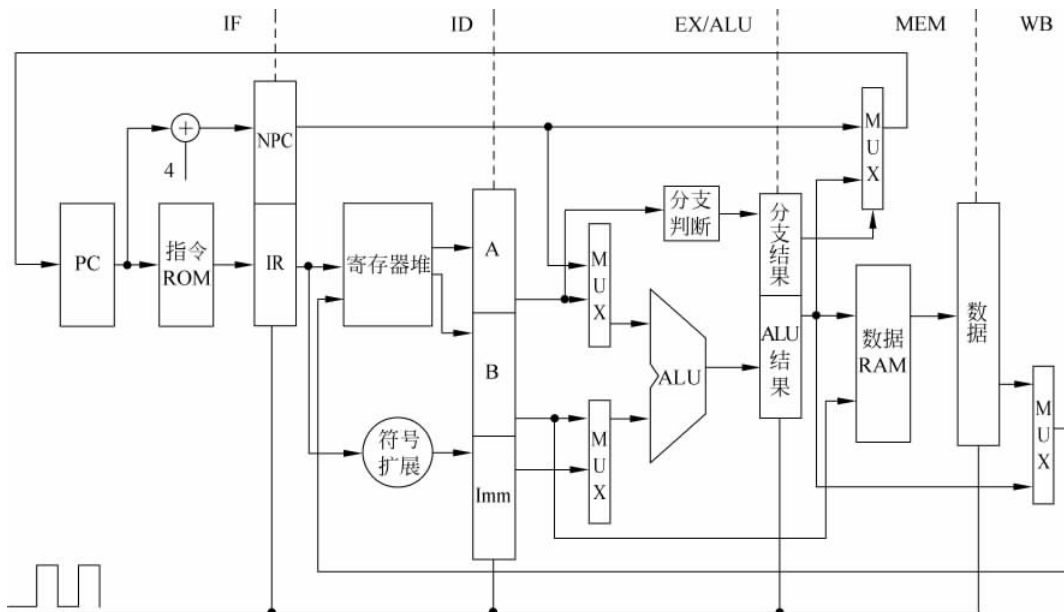


图 5-4-1 流水线框图

每一级是封闭式分立总线,后一级只访问前一级间已寄存好的信息并加工处理为新的中间数据。每级更关注组合逻辑如何处理好数据,级间寄存器如何以及哪些信息要输出或要寄存。

相比上述设计,AVR 流水线的设计将 5 级压缩到 2 级 IFD(负责 IF、ID)与 EX(负责 ALU、MEM、WB)。而且,AVR 微架构内部存在一条开放的公共的 8 位数据总线。本节实验采用统一的节拍 t1t2t3 单周期并行取指与指令执行,重点考虑为 CPU 控制器及每条指令在每个节拍划分任务。

5.4.4 实验内容

BE: c3avr_core

内容: 考虑单指令 ORI,建立 Mealy 型状态机模型,设计单周期并行处理 ORI 单指令的 AVR 两级流水线微处理器软核,完成 c3avr_core 功能仿真验证。

分析:

(1) 以状态图来表示 ALU 指令并行执行情况。

程序处理过程采用图 5-4-2 来表示,两个状态 IFD 和 EX/IFD。IFD 状态先取指令并译码,指令经过译码,跳转至 EX/IFD 状态执行运算,同时取出下一条指令并译码,直至所有指令执行完。

按时序分配任务,图 5-4-2 中第二状态里的 EX 与 IFD 必须分别考虑指令与数据两种情况,也要综合考虑单总线数据传输同一时刻独占机制,且如果流水线是封闭式多数据总线可忽略后者。

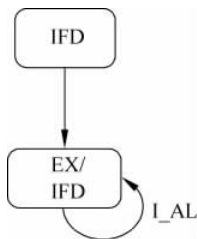


图 5-4-2 指令执行

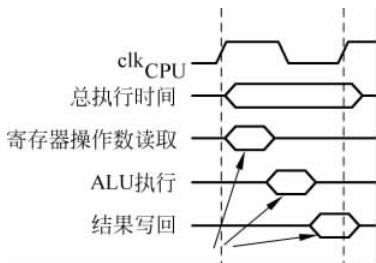


图 5-4-3 ALU 单周期子操作

(2) 结合图 5-4-3,时序划分参考表 5-4-1(也可自行设计)。

控制模块各节拍内操作基本原则是运算 t1,取指 t2,写回 t2,译码 t3 以前,状态判定 t3,读取寄存器堆 t3。

流水线处理过程中指令操作需叠加,如 ex_alu 状态时,除了正常运算操作还附上了取指译码。

bus_reg_t1、bus_reg_t2、bus_reg_t3 表示数据总线上各节拍寄存器。这一类寄存器一方面作为流水线级间数据传递,类似 D 触发器链逐级缓冲;另一方面符合硬件描述禁止多个并行进程同时对一信号操作行为。所有可叠加的操作先划入各个节拍下方便同一进程描述。

表 5-4-1 时序节拍下各状态的操作表

当前状态	t1 ↑	t2 ↑	t3 ↑	下一状态
CU 控制模块 的基本操作	运算	bus_reg2<-bus_reg1 取指 写回 regf	bus_reg3<-bus_reg2 state 判定 读 regf	
IFD NOP	rom_ce<-1	op<-Rom	PC+1 Regfiles_rd Id 判定 (Rd,Rr,k)<-op	Id 判定
Ex_ALU	运算 bus_reg1<-Alu (sreg_o 可自选) rom_ce<-1	写去哪由 Id 判定 ori;rd op<-Rom	PC+1 Regfiles_rd Id 判定 (Rd,Rr,k)<-op	Id 判定

(3) 依据表 5-4-1 完成软核代码,参考如下,建立波形图给定激励输入从状态转移和输出两方面分析代码。代码中指令依据指令集格式书写。

```

-----
-- entity:c3avr_core with no ROM
-- file:c3avr_core.vhd
-- author:zhanghongjie
-- description:8bits avr 2-stage unsigned integer pipeline
-----

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;
use IEEE.std_logic_arith.all;
entity c3avr_core is
port(
    -- core control
    Rst,t1,t2,t3:in STD_LOGIC;
    -- prog ROM
    ROM_Addr_o:out STD_LOGIC_VECTOR(13 downto 0);
    ROM_data_i:in STD_LOGIC_VECTOR(15 downto 0);
    ROM_CE :out STD_LOGIC;
    -- bus
    bus_inst_o:out STD_LOGIC_VECTOR(15 downto 0);
    bus_data_o:out STD_LOGIC_VECTOR(7 downto 0)
);
end c3avr_core;
architecture rtl of c3avr_core is
    ----- signal declare -----
    -- pc Reg
    signal pc:STD_LOGIC_VECTOR(13 downto 0);
    -- bus Reg
    signal ALU : STD_LOGIC_VECTOR (7 downto 0);
    signal bus_Reg_t1,bus_Reg_t2,bus_Reg_t3 : STD_LOGIC_VECTOR (7 downto 0);

```

```

signal op: STD_LOGIC_VECTOR(15 downto 0 );
-- op flags
signal is_ori, is_lds, is_sts :STD_LOGIC;      -- ori
-- regfiles is a 32 * 8 RAM with 2 read ports & one Wr port.
type regfiles is array(0 to 31) of STD_LOGIC_VECTOR(7 downto 0);
signal regf32:regfiles:= (others => x"00");
-- reg for computing
signal regf_d_i      : STD_LOGIC_VECTOR (7 downto 0);
signal regf_rdad_Rd   : STD_LOGIC_VECTOR (4 downto 0);
signal regf_rdad_rr   : STD_LOGIC_VECTOR (4 downto 0);
signal regf_wrad      : STD_LOGIC_VECTOR (4 downto 0);
signal regf_Rd_o      : STD_LOGIC_VECTOR (7 downto 0);
signal regf_rr_o      : STD_LOGIC_VECTOR (7 downto 0);
signal k               : STD_LOGIC_VECTOR (7 downto 0);
----- control unit ,a mealy state machine -----
-- build an enumerated type for the state machine
type state_type is (ifd, ex_ALU, ex_lds, ex_ldst, ex_sts);
-- register to hold the current state
signal state : state_type;

begin
-- wire output
ROM_Addr_o<= pc;
bus_inst_o<= op;
bus_data_o<= bus_Reg_t3;
-- based on the current state and the input ,
-- determine the next state synchronously.
cu_state:process (t3, Rst)
begin
if Rst = '1' then
state<= ifd;    -- initial
elsif (rising_edge(t3)) then
case state is
when ifd | ex_ALU | ex_ldst =>
-- id
if ( is_ori = '1' ) then
state<= ex_ALU;
endif;
when ex_lds | ex_sts =>
state<= ex_ldst;
end case;
end if;
end process;
-- at t1, determine what should do and the output based only on
-- the current state and the input .
cu_o_t1:process (Rst, state, t1)
begin
ROM_CE<= '1' ;
if Rst = '1' then
ROM_CE<= '0';
ALU<= (others => '0');

```

```

        bus_Reg_t1 <= (others => '0');
    elsif (rising_edge(t1)) then
        case state is
            when ifd =>
                when ex_ALU =>
                    ALU <= regf_Rd_o or k;
                    bus_Reg_t1 <= regf_Rd_o or k;
                when others =>
                    end case;
        end if;
    end process;
-- at t2, determine what should do and the output based only on
-- the current state and the input .
cu_o_t2: process (Rst, state, t2)
begin
    if Rst = '1' then
        op <= (others => '0');
        bus_Reg_t2 <= (others => '0');
    elsif (rising_edge(t2)) then
        bus_Reg_t2 <= bus_Reg_t1;
        case state is
            when ifd =>
                op <= ROM_data_i;
            when ex_ALU =>
                -- id -- a:Rd b:rr
                if (is_ori = '1') then
                    regf32(conv_integer(regf_wrad)) <= ALU;
                end if;
                op <= ROM_data_i;
            when others =>
                end case;
        end if;
    end process;
-- instruction hex code to op flag, Rd_Addr, rr_Addr.
-- these regs, such as Rd, rr, are used to Reg a and b for ALU computing,
-- and also attribute these to reduce disturbing between 2 stage in pipeline.
is_ori <= '1' when op(15 downto 12) = "0110" else '0';
-- at t3, determine what should do and the output based only on
-- the current state and the input .
cu_o_t3: process (Rst, state, t3)
begin
    if Rst = '1' then
        pc <= (others => '0');
        bus_Reg_t3 <= (others => '0');
        regf_Rd_o <= (others => '0');
        regf_rr_o <= (others => '0');
        k <= (others => '0');
        regf_rdad_Rd <= (others => '0');
        regf_rdad_rr <= (others => '0');
        regf_wrad <= (others => '0');
    elsif (rising_edge(t3)) then

```

```

bus_Reg_t3 <= bus_Reg_t2;
pc <= pc + 1;
case state is
    when ifd | ex_ALU =>
        -- id -- a:Rd b:rr
        if ( is_ori = '1' ) then
            regf_Rd_o <= regf32(conv_integer( '1' & op(7 downto 4) ));
            k <= op(11 downto 8) & op(3 downto 0); -- k
            regf_rdad_Rd <= '1' & op(7 downto 4) ; -- 16 < d < 31
            regf_wrad <= '1' & op(7 downto 4) ;
        end if;
    wheno thers =>
end case;
end if;
end process;
end rtl;

```

5.4.5 思考题

举例说明本节实验处理器控制模块采用的流水线如何来满足指令并行要求。

5.5 SOPC 模型

5.5.1 基本知识点

VHDL 模块编程、模块实例化与多文件编程；能够建立最小的 SOPC 模型，添加 ROM 与时序节拍器件；书写简单汇编程序、AVR 模拟器调试并转化为 ROM 初始文件；完成功能仿真验证模型机的执行周期、寄存器读与写回、复位信号。

5.5.2 实验设备

DDA-III A 便携板、Quartus II Web 9.0。

5.5.3 实验概述

miniSOPC 电路图如图 5-5-1 所示，模块 rom16k16 由 5.3 节实验依 LPM 定制，用户代码 prog.mif。模块 c3avr_core 为 5.4 节 CPU 软核，此图中没有添加 5.1 节时序节拍电路。

5.5.4 实验内容

BE: miniSOPC

内容：比照图 5-5-2 书写简单 prog.mif 程序，完成功能仿真验证，分析模型机的执行周期、寄存器读与写回、复位信号。

参考波形图如图 5-5-3 所示。rst、t1、t2、t3 设置激励仿照 5.1 节时序节拍信号。bus_o

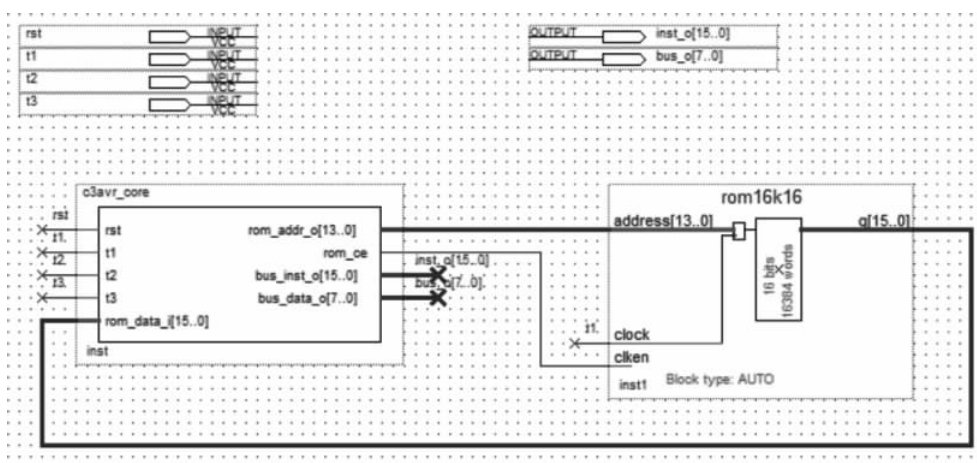


图 5-5-1 电路总图

Addr	+0	+1	+2	+3	+4	+5	+6	+7
0000	6311	6123	0000	0000	0000	0000	0000	0000
0008	0000	0000	0000	0000	0000	0000	0000	0000
0010	0000	0000	0000	0000	0000	0000	0000	0000
0018	0000	0000	0000	0000	0000	0000	0000	0000

图 5-5-2 ROM 内容

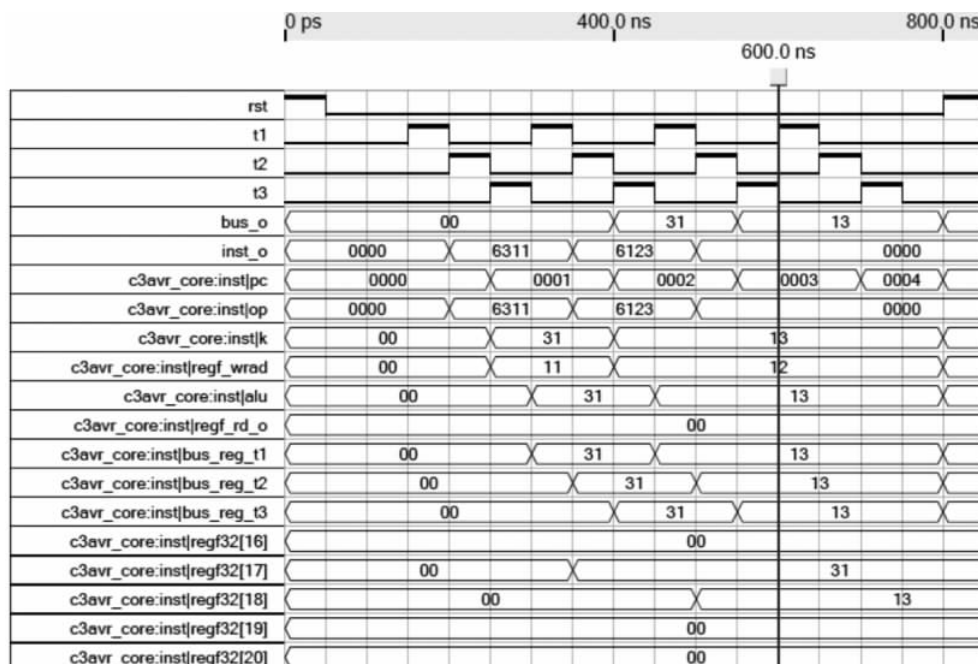


图 5-5-3 功能仿真结果

和 inst_o(op) 分别表示数据总线和指令总线。PC 程序计数从 0x0000 开始,观察下一条指令地址指示。k 寄存着 ORI 指令运算赋值的常量。regf 前缀代表寄存器堆,regf_rd_o 是寄存器堆中取出运算数寄存器,regf_wrad 写入寄存器号,regf32[17] 表示寄存器 r17。alu 寄存器保存运算结果。

功能仿真波形显示:三个周期结束(600ns 游标处),寄存器 r17 与 r18 先后经 ORI 赋值为 0x31 与 0x13。

分析:若将代码中“is_ori<='1' when op(15 downto 12)="0110" else '0';”修改成“is_ori<='1' when rom_data_i(15 downto 12)="0110" else '0';”两条 ORI 是否两周周期结束?VWF 中添加 registers 类节点,通过观察 Regfiles 寄存器的数值来验证其复位、清零等情况。

5.5.5 思考题

为 c3avr_core 添加一条 ALU 指令,如 OR、LDI,验证其功能,说明它们内在的联系。

5.6 硬件测试

5.6.1 基本知识点

配置 DDA-III A 便携板的 Cyclone II 型芯片与硬件验证。阅读工程综合结果,包括芯片、引脚、LE 资源,了解编译与综合规则的影响;认识 Fmax 与 slack 概念,完成时序分析;应对数据相关 RAW 问题,设计数据旁路逻辑并完成时序仿真验证。

5.6.2 实验设备

DDA-III A 便携板、Quartus II Web 9.0。

5.6.3 实验概述

1. 综合

RTL 验证功能成功后,RTL 代码可以由综合器优化得到网表。Quartus 软件里影响综合器结果的因素主要是 Analysis & Synthesis Settings 设置,如图 5-6-1 所示。

通常情况,默认 Balance 综合策略,可兼顾时序与面积的要求。速度优先策略会使用更多的面积实现。相对地,面积优先牺牲速度适合资源较紧张的设计。More Settings 按钮可以打开 Quartus 更详细的综合策略选项。

2. 布局布线

布局布线结果完全依赖用户提供约束,包括芯片、引脚、时序等与应用硬件环境紧密相关的信息。Quartus 中提供 Fitter 来实现布局布线,如图 5-6-2 所示。

全编译过程中,往往容易忽略布局布线过程。实际上,布局与布线紧密联系,依靠优化策略的努力程度来满足时序要求。

3. 时序分析

布局布线再如何加大努力程度都无法满足时序要求的情况下,那就可能不仅仅是时序优化的问题,需要分析延时比较大的关键路径,甚至人工优化 RTL 代码。TimeQuest 静态

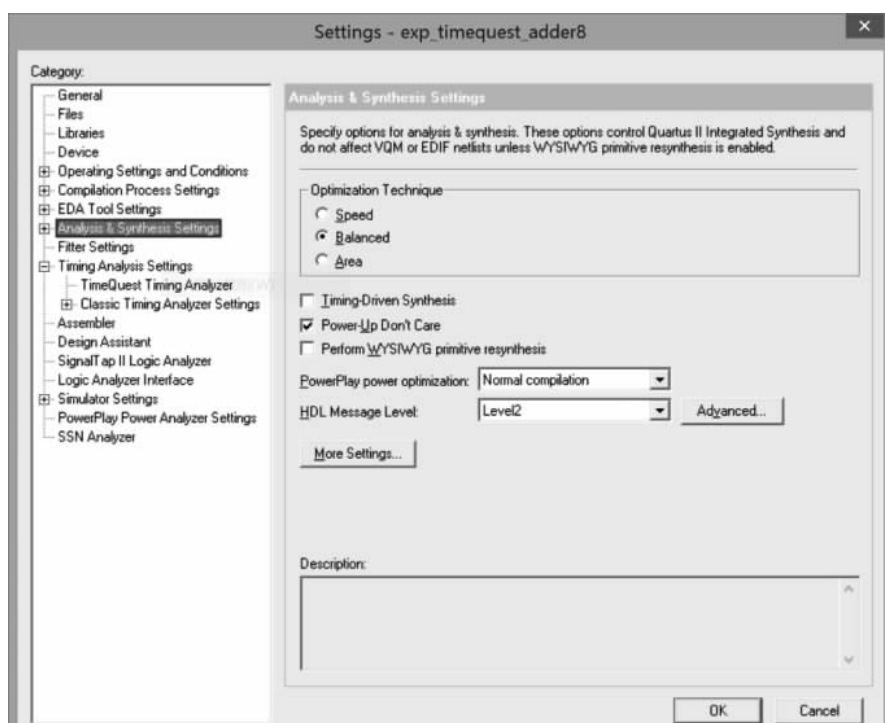


图 5-6-1 Settings 设置框 1

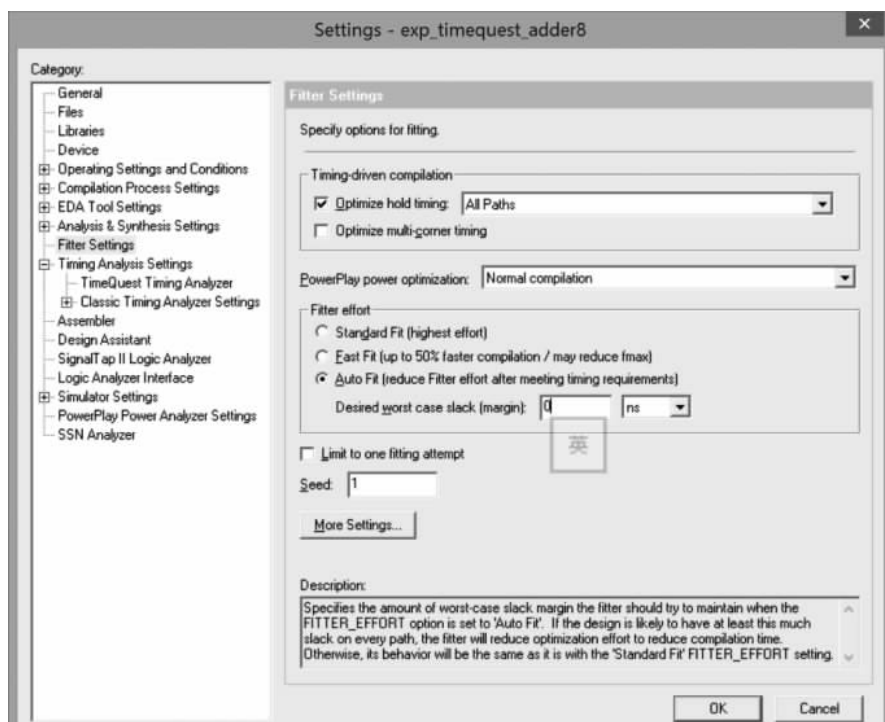


图 5-6-2 Settings 设置框 2

时序分析在约定的时钟频率 $F_{\max}$ 下,检测每一条时序路径延时余量 Slack 情况,正为富余,负为违约。

4. RAW 问题

假设指令 2 是在指令 1 后面执行的指令,写后读(Read After Write,RAW)表示指令 1 将数据写入寄存器后,指令 2 才能从这个寄存器读取数据。如果指令 2 在指令 1 写入寄存器前尝试读出该寄存器的内容,将得到不正确的数据。

数据前推或旁路将计算结果从其产生处直接送到其他指令所需的功能单元处。

5.6.4 实验内容

BE: TimeQuest 时序分析

内容:以 8 位加法器为例,编程完成 RTL 功能,查看工程目录下综合后的编译报告 map.rpt。记录以下数据并分析优化:

- (1) Summary 等表中芯片、逻辑单元、寄存器、引脚、顶层文件利用信息。
- (2) Settings 中综合器优化策略、功耗优化策略等选项。

参考代码如下:

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;
entity exp_timequest_adder8 is
    generic
        (    data_width : natural := 8
        );
    port
        (    a,b,c : in unsigned ((data_width-1) downto 0);
            Clk      : in STD_LOGIC;
            sum : out unsigned ((data_width-1) downto 0)
        );
end entity;
architecture rtl of exp_timequest_adder8 is
    signal a_Reg,b_Reg,c_Reg,sum_Reg:unsigned ((data_width-1) downto 0);
begin
    process(Clk)
    begin
        if (Clk'event and Clk = '1') then
            a_Reg <= a;
            b_Reg <= b;
            c_Reg <= c;
            sum_Reg <= a_Reg + b_Reg + c_Reg;
        end if;
    end process;
    sum <= sum_Reg;
end rtl;
```

分析:

- (1) 先期准备。新建工程同名 Synopsys Design Constraints File,扩展名 SDC,保存至

工程目录；确保 Fitter 编译工作已经完成；Settings→Timing Analysis Settings 中选择 Use TimeQuest，下一级 Time Quest Analyzer 中选择 SDC file to include in the project 并添加。

打开 Timequest，选择 Quartus 窗口 Tools→Time Quest Timing Analyzer 菜单，主要关注 Tasks 面板和 Report 面板。任务面板提供了获得一个设计的时序信息的一系列命令动作，这些命令动作包括建立时序网表 creating a timing netlist，读取时序约束文件 reading a timing constraints file，执行时序分析 performing timing analysis，生成报告 generating reports 和保存时序约束文件 saving a timing constraints file。报告面板包含了设计的详细时序信息，这些报告使用任务面板的命令动作来生成。

(2) 生成时序报告。Tasks 列表中，双击执行 Create Timing Netlist。若提示错误，请重新执行一次 Fitter 编译。此时，Tasks→Reports 下面就是新报告。

(3) 记录常用时序报告并阅读。

影响时序最大的 Fmax 工作时钟报告 Report Fmax Summary，如图 5-6-3 所示下方有说明文字。Fmax 没设置时，软件默认采用最快 1GHz。

常用的建立余量报告 Report Setup Summary。Slack 负值红色醒目地表示违约，正值绿色为正常。如图 5-6-4 中 -3.434 表示 1ns 时钟周期太短，少了 3.434ns。

Fmax Summary				
	Fmax	Restricted Fmax	Clock Name	Note
1	225.53 MHz	225.53 MHz	clk	

图 5-6-3 执行速度报告

Summary (Setup)				
	Clock	Slack	End Point TNS	
1	clk	-3.434	-23.282	

图 5-6-4 时序摘要

(4) 速度与路径时序分析。

本工程时钟频率 Fmax 达到最快 1GHz 么？周期重新设置至少为多少 ns？

为获取更详细的时序信息，如违约的关键时序路径，右击已给时钟所在行，然后选择右键菜单中的选项 Report Timing，打开的新对话框，如图 5-6-5 所示。

Clocks 区域中，选择指定发出数据的时钟信号到锁存数据的时钟信号 clk，将让分析器仅报告寄存器到寄存器路径。

Targets 区域指定需要报告的特定路径。此处留白不填，报告所有的指定时钟路径。

Analysis type 和 Paths 区域：Analysis type 区域用来指定分析的时序类型，包括建立时间 Setup、保持时间 Hold、恢复时间 Recovery 和移除时间 Removal。此处设置分析 Setup 且最多报告 10 条违反时序约束路径。

Output 区域可以指定输出的格式和报告的详细程度。此处默认选项，即 Detail level 设置 Path only 并输出在 Setup: clock 窗口中。

Tcl command 区域显示可执行命令，默认设置。

单击 Report Timing 按钮可以生成报告窗口，如图 5-6-6 所示。

图 5-6-6 中，点开 Waveform 标签页图形化显示各延时，右下方 Data Path 标签页显示路径经过部件及时延。Waveform 用户测量时序，可移动游标至起点，再按住左键拖动到测量终点。Data Arrival 和 Data Delay 两个波形显示了信号从源寄存器传播到目的寄存器的时间。

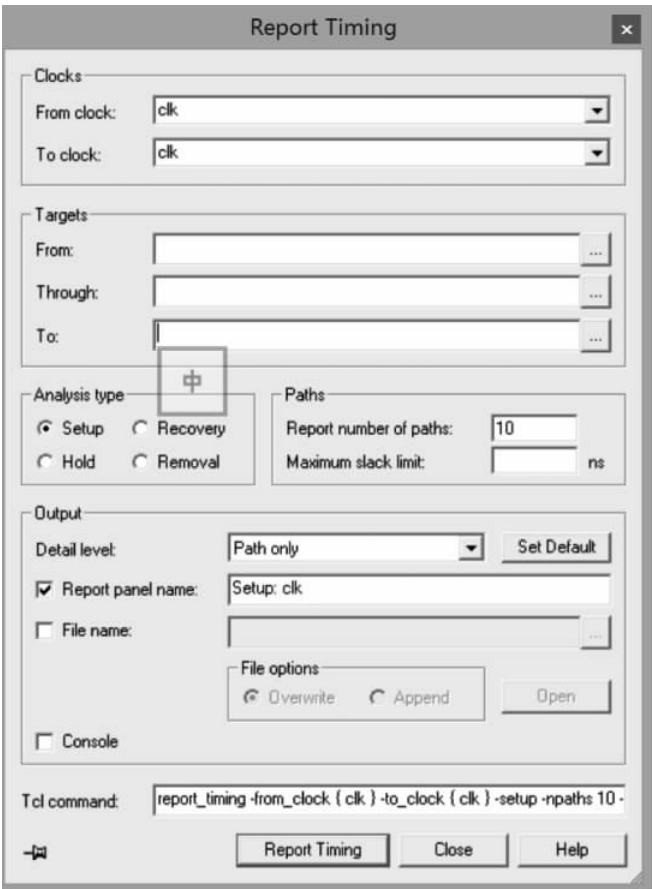


图 5-6-5 Report Timing 设置框

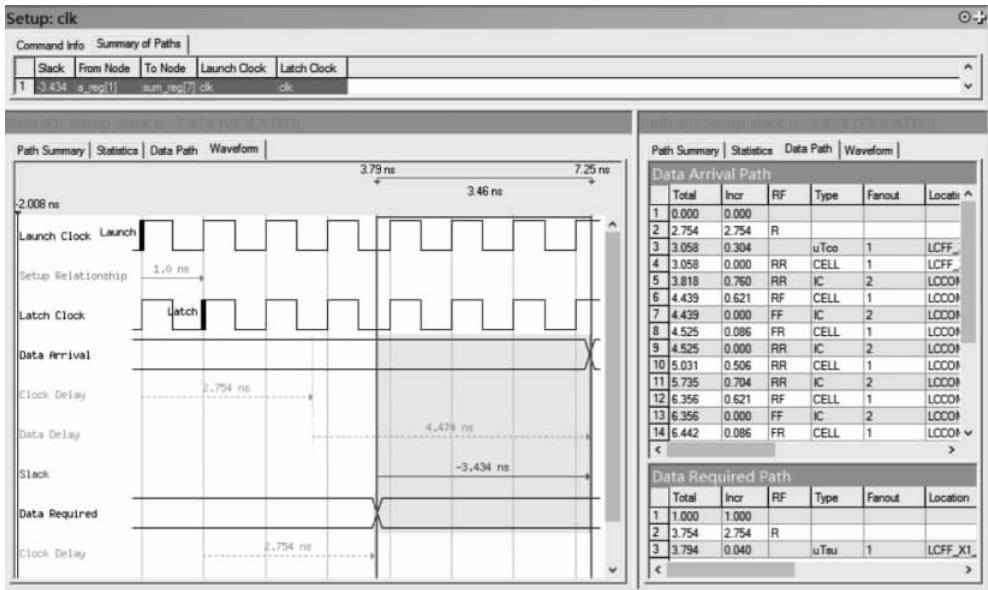


图 5-6-6 生成报告

(5) 违约与优化分析。

分析：Slack 项显示红色区域为时序违约，指示 Data Arrival 和 Data Required 时间先后关系？

设计：设计时钟信号约束，优化 Slack。

TimeQuest 窗口选择 Constraints→Create Clock 设置下一次编译中应用指定的时钟约束，打开 Create Clock 对话框，如图 5-6-7 所示。Clock name 栏输入本例 clk。Period 设置合适周期，默认 1ns。Waveform edges 区域定制电平，相应改变会在图示中显示，空白默认占空比 50%。

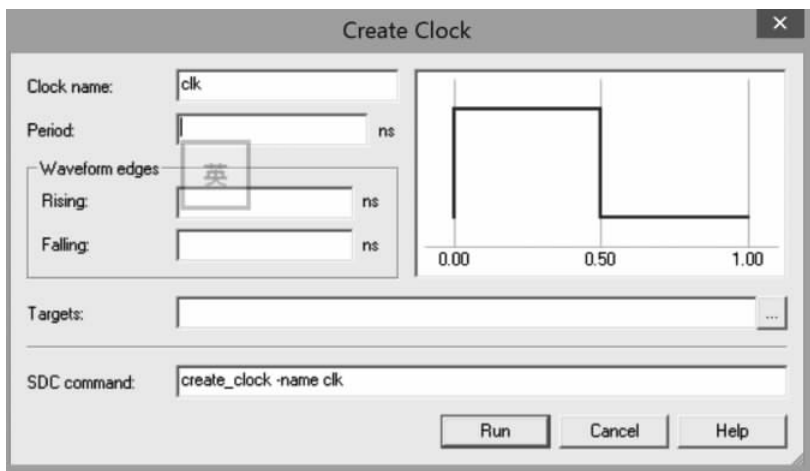


图 5-6-7 创建时钟 1

单击 Targets 右边浏览按钮，弹出 Name Finder 对话框，如图 5-6-8 所示，Matches 区域单击 List 按钮，左侧列表中选中 clk 双击至右侧 selected name，单击 OK 按钮返回 Create Clock 对话框，单击 Run 按钮执行。

写入 SDC：TimeQuest 窗口选择 Constraints→Write SDC 选项，保存或覆盖工程同名 .sdc 文件。

重新编译 Fitter 后，查看 TimeQuest 分析。Slack 不合格，则重复上述步骤，直至满足时序约束。

5.6.5 思考题

设计一段包含相邻两条以上类似 ORI Rn, N(n, N 为常量)指令的程序，手工编译成 prog. mif，利用上一节的 miniSOPC 执行功能验证，从波形图分析 c3avr_core 处理器如何解决 RAW 相关问题。

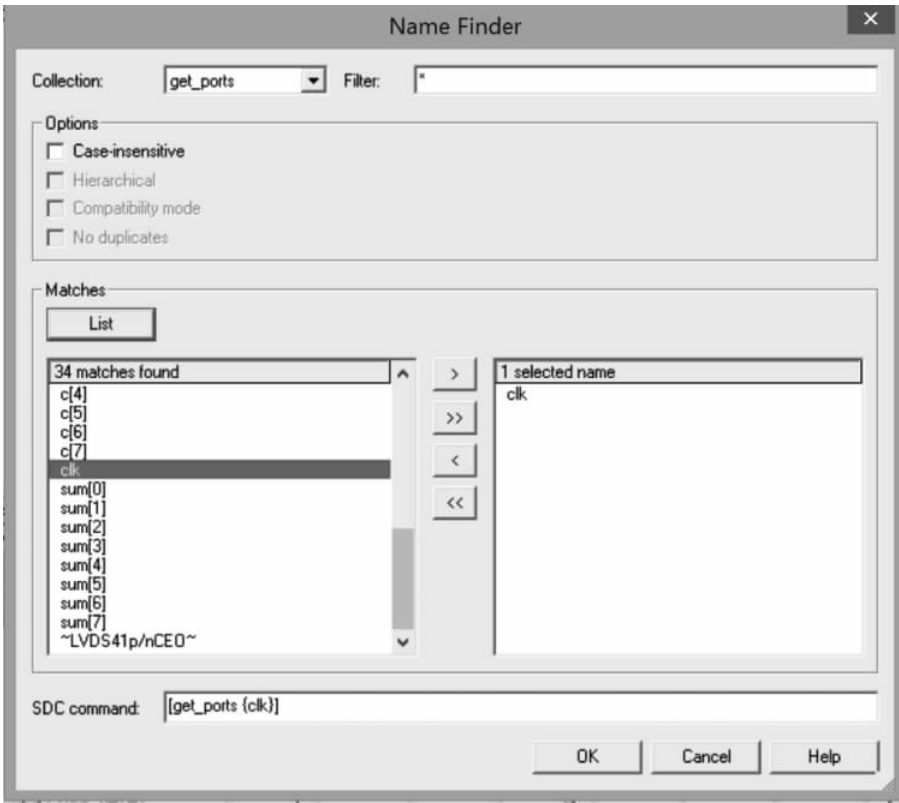


图 5-6-8 创建时钟 2

5.7 综合实践 1

5.7.1 基本知识点

从 AVR 的 LED 闪烁应用入手,改进微处理器实现延时、跳转与条件分支、SREG 状态位判断、I/O 模块、调用与栈、位操作及中断开关等相关指令,在 FPGA 开发板上完成功能验证。

5.7.2 实验设备

DDA-IIIA 便携板、Quartus II Web 9.0、Atmel Studio 6.1。

5.7.3 实验概述

1. 延时函数

_delay_ms(double __ms)和_delay_us(double __us),是 AS6 中经常使用的两个标准延时函数,使用时需要加载头文件#include <util/delay.h>,参考代码如下。参数虽然是 double 型,但可赋整型值。另外,调用前定义#define F_CPU 1600000UL。延时 300ms 的

函数写为 `_delay_ms(300)`。

```
#include <avr/io.h>
#define F_CPU 16000000UL
#include <util/delay.h>
```

一般情况下只要晶振填写准确,这两个函数延时很精确。若没有指定时钟频率 `F_CPU`,编译器会默认 10MHz 且报警。

2. 库文件

库文件系统路径一般位于 Atmel Studio 程序安装根目录下\Atmel Toolchain\AVR8 GCC\Native\3.4.2.939\avr8-gnu-toolchain\avr\include,如图 5-7-1 所示,另可在工程属性中查看。

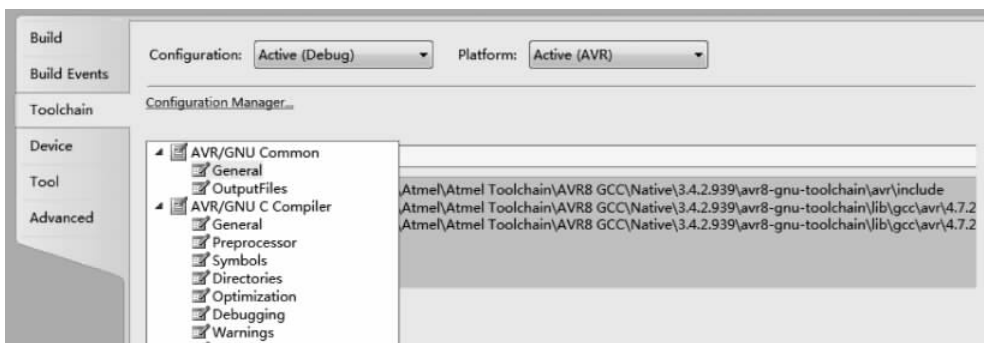


图 5-7-1 库文件路径

3. 状态字判别

指令运算后处理器还会根据结果考虑更新状态寄存器 SREG 相应比特值,判别逻辑的英文版手册部分如图 5-7-2 所示。

Status Register (SREG) and Boolean Formula:

	I	T	H	S	V	N	Z	C
	—	—	↔	↔	↔	↔	↔	↔

H: $Rd3 \bullet Rr3 \bullet Rr3 \bullet \overline{R3} \bullet Rd3$
Set if there was a carry from bit 3; cleared otherwise

S: $N \oplus V$, For signed tests.

V: $Rd7 \bullet Rr7 \bullet \overline{R7} \bullet Rd7 \bullet \overline{Rr7} \bullet R7$
Set if two's complement overflow resulted from the operation; cleared otherwise.

图 5-7-2 SREG 判定

4. Stack 栈

AVR 的栈是向下生长的,即新数据推入栈时,栈指针的数值将减小。栈指针 SP 主要用来保存临时数据、局部变量和中断/子程序的返回地址。SP 总是指向堆栈的顶部,初始值为 Internal SRAM 最大地址,当发生 PUSH 动作,SP 存储的地址由大变小。SP 值由寄存器 SPH 与 SPL 分别存储。

Data Memory	
32 Registers	0x0000~0x001F
64 I/O Registers	0x0020~0x005F
160 Ext I/O Reg.	0x0060~0x00FF
Internal SRAM (512/1024/1024/2048×8)	0x0100
	0x04FF/0x04FF/0x0FF/0x08FF

图 5-7-3 数据空间映射

5. ASF 框架

Atmel 软件框架(Atmel Software Framework, ASF)为不同的 MCU 提供软件驱动和库来加快开发过程,ASF 的架构如图 5-7-4 所示。Your Application 对应要实现的软件应用,Utilities 对应开发工具和宏,Boards 对应开发板或者用户自定义板。

ASF 由 drivers、components 和 services 三部分组成(各版本分组会有所不同)。

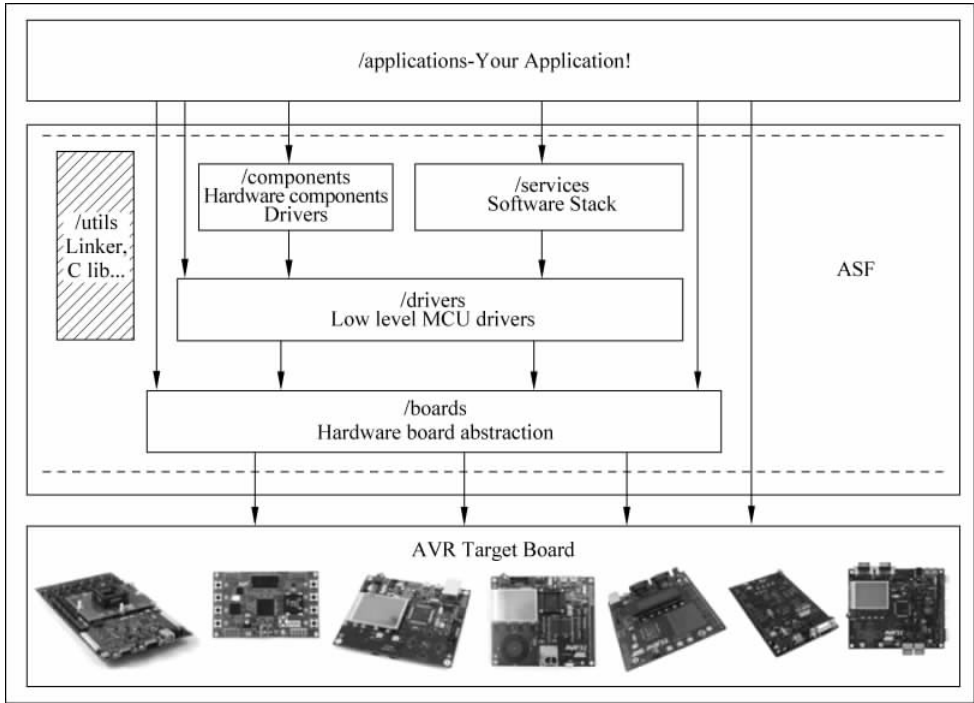


图 5-7-4 ASF 功能

5.7.4 实验内容

BE: FPGA 的 C 应用

内容: 利用 AVR 的 C 语言编写一个 LED 闪烁应用并指定为 PORTB5 输出显示; 编译成功将 Hex 自动翻译成 mif 内存初始文件; 详细分析汇编代码, 硬件上为 c3avr_core 实现各条指令; 最后在完整 SOPC 模型上完成系统功能硬件验证。

最终结果: 如图 5-7-5 所示, QD:1->0, 连续观察到 IOR_o(5) 交替亮灭(对应 PIN: 80

上的 LED 灯 Z5)。

原代码是 16MHz 主频,但板载是 6MHz 时钟输入,显示延时会有所增加。

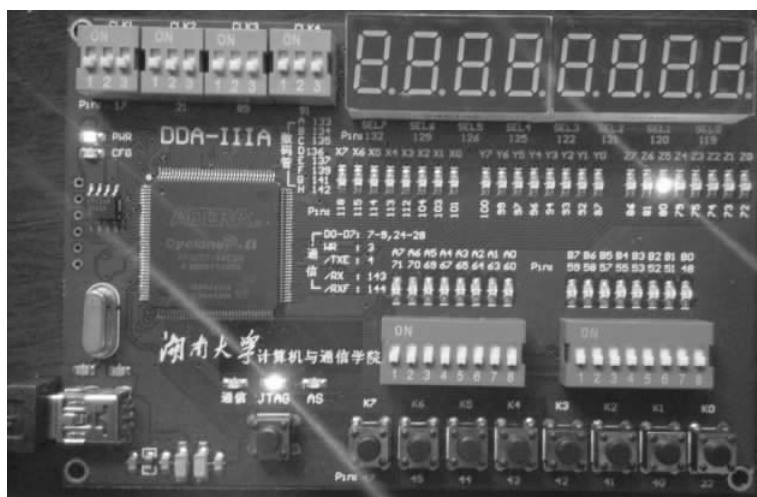


图 5-7-5 硬件验证结果

1. 软件设计

初始化 I/O 口后 while(1)主循环内完成逻辑电平翻转与延时,参考代码如下:

```
#include <avr/io.h>
#define F_CPU 16000000UL
#include <util/delay.h>
#define LED1ON (PORTB| = 0x20)
#define LED1OFF (PORTB&= 0xDF)
int main(void)
{
    DDRB| = 0x20;
    while(1)
    {
        LED1ON;
        _delay_ms(300);
        LED1OFF;
        _delay_ms(300);
    }
}
```

Build 编译通过,OUTPUT 信息窗口显示“Program Memory Usage :176 bytes 0.5 % Full”。打开工程目录 Debug/Blinkled.iss 汇编文件。注意:需要选择编译器优化,否则会报警告,优化后代码更简洁。

```
#ifndef __OPTIMIZE__
#warning "Compiler optimizations disabled; functions from <util/delay.h> won't work as designed"
#endif
```

分析：根据 Disassembly 显示，记录程序执行部分及基本运行情况。

2. 硬件设计

(1) 利用 Quartus 将 Hex 转化为 mif。

直接将 ex 文件转化为 mif,会报警告如下：

Warning: Byte addressed memory initialization file "Blinkled.hex" was read in the word-addressed format

解决方法：设置 Quartus 中菜单项 Tools→Options,打开设置对话框。选择 Category→Memory Editor,勾选 Read or write Hexadecimal Files (.hex) using byte addressing (Intel Format)。

Quartus 中打开 Hex 文件,再选择 File→Save as,确定保存 word 字长 16,文件名 prog.mif。

注意：数据在程序存储器中是按照字存放的。处理器从中读出 1 个字(2 个字节),与处理器的指令格式字节次序不一致,即存放次序是先低字节后高字节。

prog.mif								
Addr	+0	+1	+2	+3	+4	+5	+6	+7
00	0C94	3400	0C94	3E00	0C94	3E00	0C94	3E00
08	0C94	3E00	0C94	3E00	0C94	3E00	0C94	3E00
10	0C94	3E00	0C94	3E00	0C94	3E00	0C94	3E00
18	0C94	3E00	0C94	3E00	0C94	3E00	0C94	3E00
20	0C94	3E00	0C94	3E00	0C94	3E00	0C94	3E00
28	0C94	3E00	0C94	3E00	0C94	3E00	0C94	3E00
30	0C94	3E00	0C94	3E00	1124	1FBE	CFEF	D8E0
38	DEBF	CDBF	0E94	4000	0C94	5600	0C94	0000
40	259A	2D9A	2FEF	85EA	9EE0	2150	8040	9040
48	E1F7	00C0	0000	2D98	2FEF	85EA	9EE0	2150
50	8040	9040	E1F7	00C0	0000	EBCF	F894	FFCF

图 5-7-6 ROM 内容信息

(2) 分析汇编语言源代码,列表记录指令的机器码、操作数、功能操作、时钟数、影响标志,见表 5-7-1。

表 5-7-1 指令说明

助记符	操作数	指令格式	功能	影响标志位	时钟周期数
例: ADD					
...					

(3) 分析指令(下面以 Debugging 中显示为准),列表记录指令处理的时序节拍下的操作说明。Quartus 中执行 prog.mif 一步步功能仿真一条条指令。参考时序节拍下各状态的操作表见表 5-7-2。

表 5-7-2 节拍下各状态操作表

当前状态	t1 ↑	t2 ↑	t3 ↑	下一状态
CU 控制模块 的基本操作	rom_ce<-1	bus_reg2<-bus_reg1	bus_reg3<-bus_reg2	
IFD NOP		op<-ROM i0: op2<-ROM	Id 判定 Is state 判定 Is:PC+1 Is:(Rd,Rr,k)<-op Is:Regf32 读数 I1: PC+1 I0: k<-op2 PC<-op2 分支 BRNE: Z0: ex_jmp K<-op Z0: PC+k+1 Z1:PC+1 RJMP: ex_jmp K<-op PC<=PC+K+1 CLI; PC+1 Sreg(7)<-’0’	Id 判定
Ex_ALU	Is:ALU<-运算结果 Is:bus_reg1<-运算结果	is:rd<-ALU SREG 判定 op<-Rom	PC+1 Regf32 读数 Is:(Rd,Rr,k)<-op	Id 判定
Ex_jmp	bus_reg1<-k	op<-Rom	PC+1 Regf32 读数 Is:(Rd,Rr,k)<-op	Id 判定
Ex_st	Is: bus_reg1<-Rr	Is:ior<-Rr op<-ROM	PC+1 Regf32 读数 Is:(Rd,Rr,k)<-op	Id 判定
IF2		op2<-rom 压栈 ram(wrad)<-pc+2	K2<-op2	Ex_call1
Ex_call1	bus_reg1<-k2		PC<-k2 SPH,SPL 读取	Ex_call2

续表

当前状态	t1 ↑	t2 ↑	t3 ↑	下一状态
Ex_call2	Sp<-sp-2	op<-ROM ior(SPH,SPL)<-sp	PC2<-PC+2; PC<-PC+1; ram_wrad<-(sph,spl) Regf32 读数 Is:(Rd,Rr,k)<-op	Id 判定
Ex_sbi	Ior_d_i<-ior_rr or k Bus_reg1<-结果	Is:ior(ior_wrad)<-ior_d_i Op<-ROM	PC+1 Is:(Rd,Rr,k)<-op	Id 判定

(4) 依据指令处理的时序节拍下操作说明,逐条在 5.4 节的软核代码框架上增加指令,仿真波形观察结果直到完成 prog. mif 所有字节功能。

注意:实际上存储字节次序是小端,与 5.4 节的软核代码框架中按指令集中指令格式来写顺序有所不同,需要另行修改其中 ID 判定寄存器赋值部分。

op2 用来存储 JMP 第 2 个字。标志 i 初始为 1,与 is_jmp 结合来区分第几次取指(1 为首次,0 表示已经取指一次)。

SREG 正常结果应该遵循 Atmel Studio 中 CPU 模拟器 Debug 时信息指示。t1 时,指令运算,同时更新寄存器 rd、rr、alu。t1 结束后,组合逻辑部分完成 sreg 每位判定,t2 时更新至 sreg。沿用前面实验所给判定公式。编译之后,SREG 从来没有被读取,无法在波形图 VWF 中添加 Registers 类型信号观察,可以采用添加 OUTPUT 方法输出显示。

I/O 输出指令 OUT,先设计容量是 64×8bit 的 I/O 寄存器。同样,这个寄存器 ior 编译优化后会消失,依然采用电路图添加 OUTPUT 方法观察波形图结果。

AVR 的 stack 向下生长,设计一个 RAM 地址 0x0100~0x08FF。压栈设计状态 IF2 将 PC+2 写 RAM,SP 写入 ior 需加入两 EX_CALL 状态。

条件分支 BRNE 判断 SREG 的 Z 位是否为 0 决定下一状态及新 PC 是否为 PC+K+1,K 需要符号扩展与 PC 位宽一致。

3. 验证

(1) 验证剩余指令按功能重写 prog. mif 并验证功能。代码顺序:RJMP 与 NOP 测试相对跳转;1 位翻转实验 CBI(ior 位清零)与 SBI(ior 位置 1)、SBI(开总中断位)与 CLI(关总中断位)两对位操作。

(2) 搭建 SOPC 最终电路,见图 5-7-7,下载到 FPGA 板上观察 LED 闪烁。

BE: 软件库 ASF 编程

内容:用 ASF 编程实现 GPIO 灯闪烁。

分析:软件框架应用步骤如下所述。

新建 GCC C 工程。工程命名 Blinkled,目标芯片 ATmega328P,工程创建完成,在生成的 main 函数里编写相应代码。

导入 ASF:进行开发时,借用官方提供的驱动。

5.8.3 实验概述

硬件实验箱经常会受硬件条件限制,不利于学习者释放积极创新和探索的热情。电路设计初期操作中容易造成仪器和元器件损坏,从而加大开发周期和反复购置造成的开发成本浪费。Proteus 这款 EDA 软件能帮助初学者进行单片机外围硬件设计、近距离观看软件编程最终结果以及波形图分析时序。

1. 环境配置与基础例程

安装 Proteus。打开 Proteus 8 Pro 文件夹,按常规步骤完成程序安装与授权。

建立工程。选择菜单 File→New project,打开工程向导。Start 页 Name 栏输入 avr_pro_blinkled,Path 栏默认。Schematic Design 页,选择 create→Landscape A4 模板。PCB Layout 页选择 Do not。Firmware 页,选择 Create,Family 为 AVR,Controller 为 ATmega 328P。Summary 页,查看摘要信息,单击 Finish 按钮,完成工程的创建。

如图 5-8-1 所示,①菜单与②③⑥工具栏和常用 EDA 原理图布局类似,各版本略有不同。新建工程完成后,“⑦原理图编辑窗”中显示工程用“⑧主芯片”,在此基础上一般原理图绘制流程为:③⑤⑦,即由③选取库中元件至⑤,再将⑤元件放置在⑦图纸上布局,最后布线。

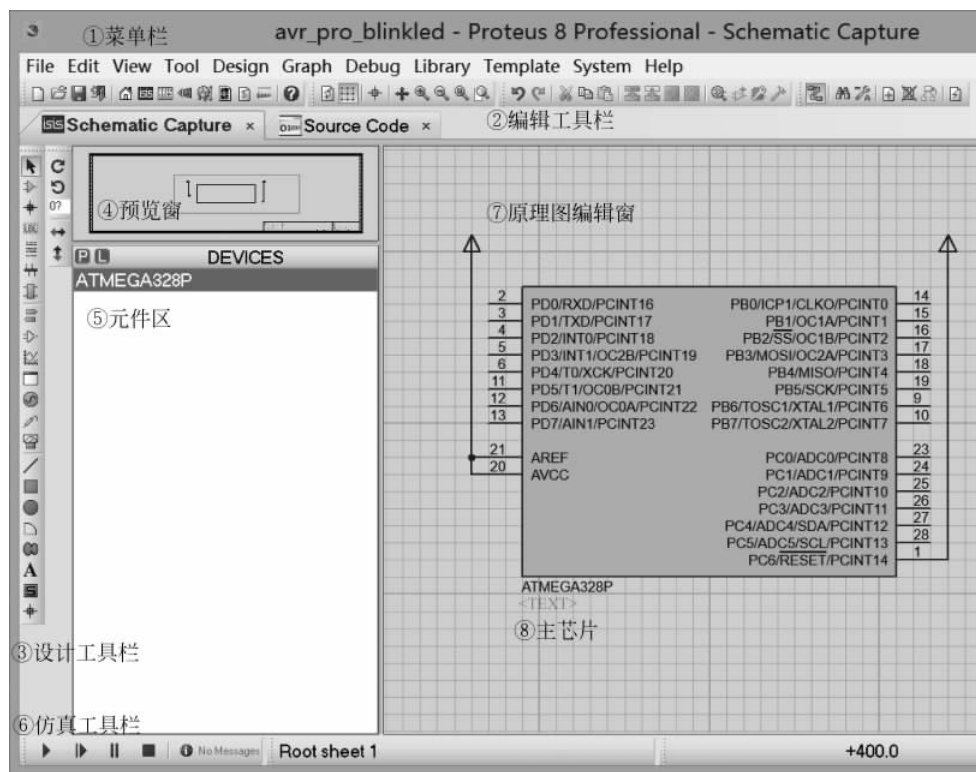


图 5-8-1 Proteus 的 AVR 工程界面

外围设计。5.7节 Blinkled 工程使用了一个 LED, 对应引脚 PORTB(5)。

元件查找及添加。单击 Proteus 左侧工具栏按钮 , 进入元件模式, 再次单击  按钮, 即可调出元件库在搜索关键词部分, 输入所需元件的关键字 LED_Yellow。如果库中有相应元件, 会在元件区域列出所选元件, 双击 LED_Yellow 型号, 将元件添加到电路图的 DEVICES, 单击此处元件名 LED_Yellow, 元件符号嵌在鼠标光标上, 将鼠标移到图纸空白处单击完成放置。

元件库分类说明如图 5-8-2 所示。

Analog Ics 模拟电路集成库	Microprocessor ICs CPU 库
Capacitors 电容库	Operational Amplifiers 运放库
CMOS 4000 series CMOS 4000 库	Optoelectronics 光电元件库
Connectors 插座、插针等电路接口连接库	PLDs & FPGAs 可编程逻辑器件
Data Converters ADC,DAC 数/模、模/数库	Resistors 电阻库
Debugging Tools 调试工具	Simulator Primitives 简单类模拟元件库
Diodes 二极管库	Speakers & Sounders 扬声器、蜂鸣器库
ECL 10000 Series ECL 10000 库	Switches & Relays 开关及继电器库
Electromechanical 电机库	Switching Devices 开关类元件库
Inductors 电感库	Thermionic Valves 热电子元件库
Laplace Primitives 拉普拉斯变换库	Transducers 传感器库
Memory ICs 存储元件库	Transistors 晶体管库
Miscellaneous 元件混合类型库	余下皆为 TTL74 或 TTL74LS 系列库
Modeling Primitives 简单模式库	

图 5-8-2 库分类

少部分常用元件对应搜索关键字列举如图 5-8-3 所示。

数码管	7SEG	电池	BATTERY
电阻	RES	电机	MOTOR
电容	CAP	或与非门	OR AND NOT
二极管	LED	可变电阻器	POT-LIN
晶振	CRYSTAL	扬声/蜂鸣器	SPEAKERS
液晶	LCD	拨码开关	DIPSW
开关	SWITCH	排阻	RESPACK
按键开关	BUTTON		

图 5-8-3 器件命名

放置引脚。在工具栏中单击  按钮, 单击 Ground, 鼠标移到电路图纸空白处单击放置。

绘制虚拟平台电路图。鼠标移动到元件端点处, 当鼠标光标变形成铅笔时, 单击便可以绘制导线连接元件。

5.8.4 实验内容

BE: 虚拟 LED 灯

内容: Proteus 上选择合适的虚拟元器件,设计完成 PB5 灯外围电路,参考图 5-8-4,仿真整个虚拟平台并分析。

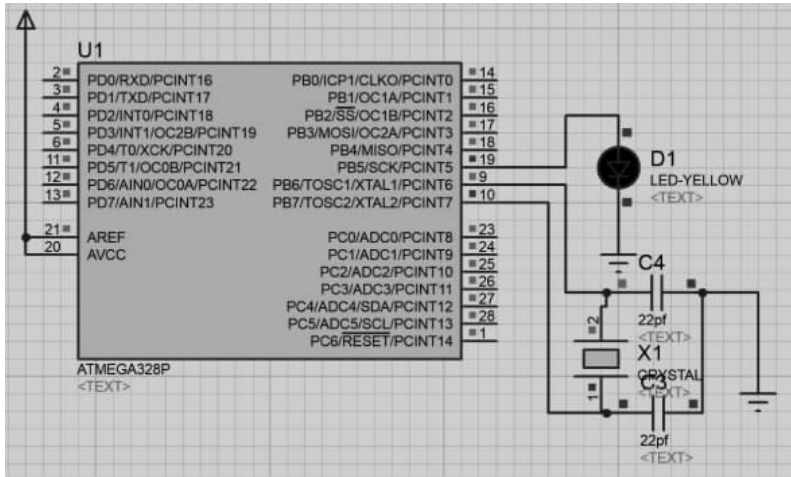


图 5-8-4 LED 电路原理图

分析: 加载 Hex 文件,启动仿真。每次都要双击 CPU 弹出 Edit Component 窗口,如图 5-8-5 所示设置好 Program File 运行代码、外部晶振 CKSEL 和 SUT 启动时间、Clock Frequency 频率,确认退出。

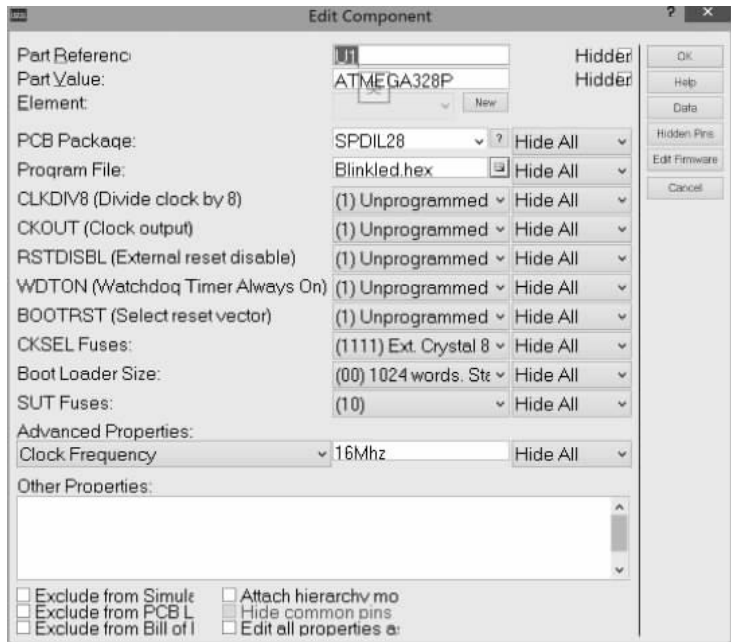


图 5-8-5 处理器设置

仿真工具栏启动仿真(三角形开始,双竖线暂停,方形结束),在电路图窗口观察 D1 闪烁黄光,如图 5-8-6 所示。

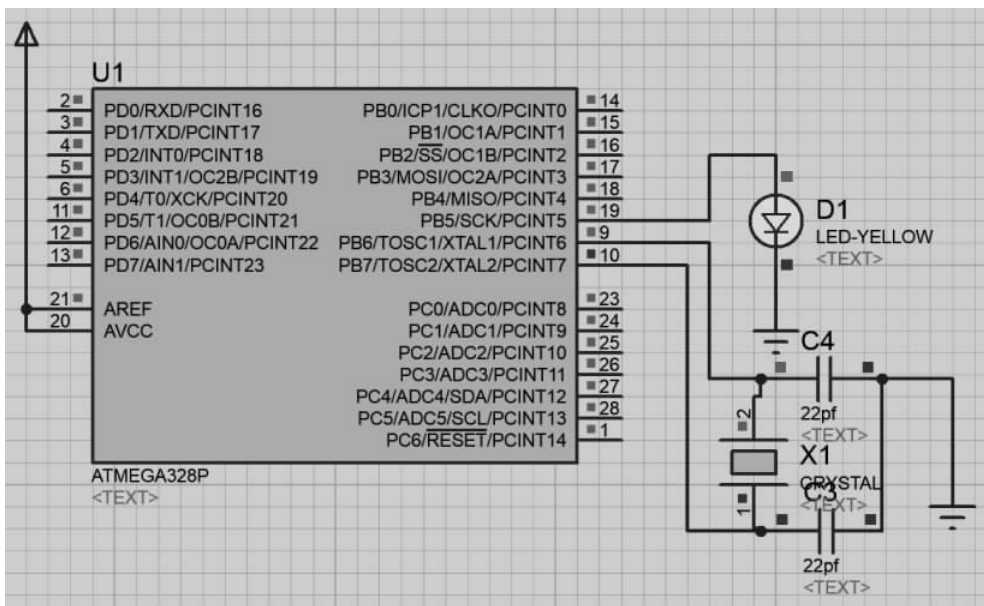


图 5-8-6 LED 灯闪烁运行结果

注意: 多次尝试添加 Program file 无果导致仿真不能正确执行的,可以试试关闭 Proteus 程序再重新打开。

5.8.5 思考题

判断 `delay_ms(300)` 延时 300ms 是否合格,如果不合格如何改进程序? 在 Proteus 完成绘制 AVR 核 ATmega328P 仿真平台且使用虚拟仪表和数字图形图表仿真。

5.9 AVR 的 Timer 与 ADC

5.9.1 基本知识点

设计复杂外围电路实例,下载仿真基础实例;依据开发板原理图,采用 Protues 设计 AVR 虚拟平台模块;编程完成模块功能并下载验证。

5.9.2 实验设备

Proteus 8 Pro 软件。

5.9.3 实验概述

1. 数码管 7SEG-MPX4-CC

7SEG-MPX4-CC 属于共阴数码管,如图 5-9-1 所示,abcdefgp 段选信号组合显示图案,

逻辑电平高就亮。位选信号 $w_3w_2w_1w_0$ 从左往右顺序选择数码管 1234。

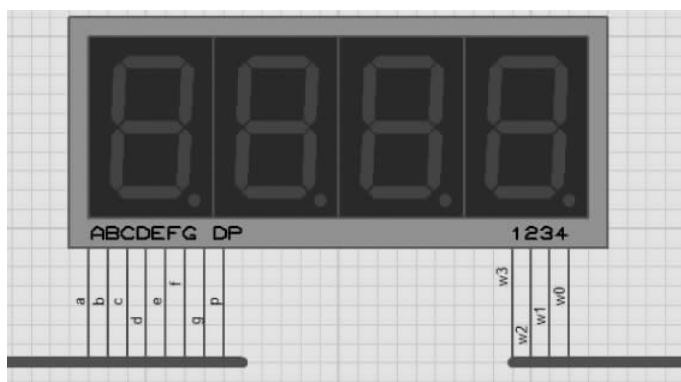


图 5-9-1 共阴数码管

2. 简易电压表

简易电压表电路如图 5-9-2 所示。数码管采用动态扫描方式输出显示电压值。ATmega328P 的 PORTD 作为数码管段选信号,PORTC[3..0]为 4 位数码管位选信号。PC5 即 ADC5 作为模拟电压测量的输入口。

改变 RV2 (POT-HG 可变电阻) 的阻值,引发电压变化经 PC5 单端输入,再经 ATmega328P 内部 ADC 转换,转换后的电压测量结果显示在数码管上。

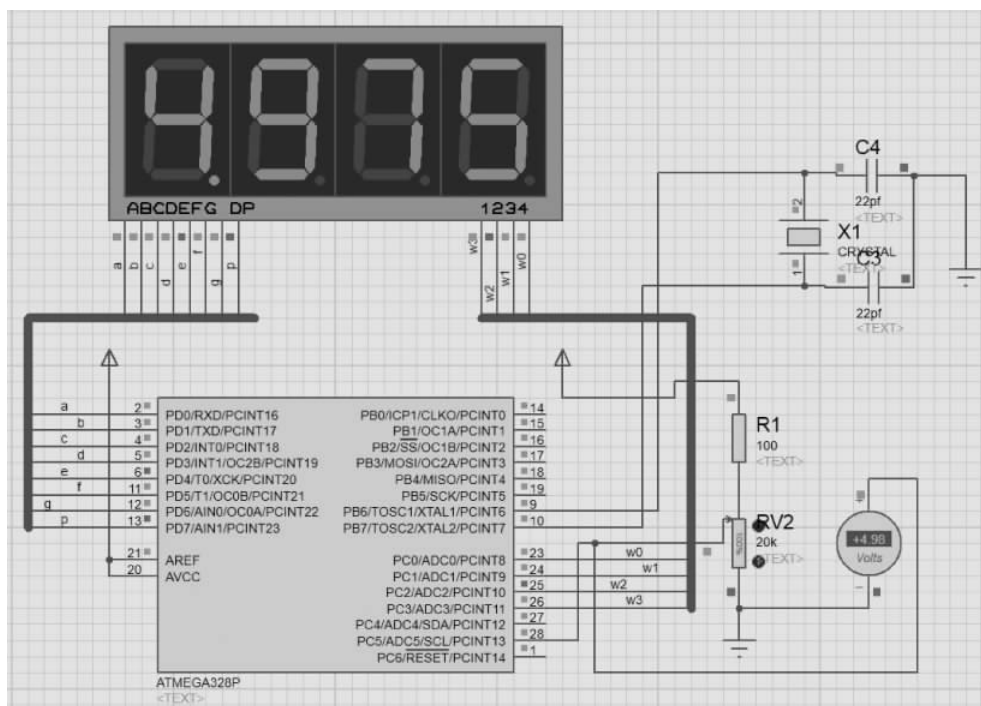


图 5-9-2 简易 0~5V 电压表电路

5.9.4 实验内容

BE: 电压表

内容：利用 Proteus 设计一个 0~5V 简易电压表电路并用 C 语言编程完成软件。

分析：

ATmega328P 的 PC0~PC3 作为 4 个数码管的位控制线使用,但 PC 口初始化时需要注意 PC5 设置成输入方式,且不能使用该口内部的上拉电阻,否则会影响到输入的模拟电压值。

程序中采用 T/C0 溢出中断,每 2ms 中断一次。该定时中断除了使 PB5 灯闪烁,还可作为开启 ADC 自动触发转换的触发信号。

在 ADC 的初始化代码中,设置 ADC 时钟的分频系数为 32。系统 4MHz 时钟经过 32 分频产生 125kHz 的 ADC 时钟,满足了逐次比较转换电路达到最大精度时需要的 50~200kHz 采样时钟要求。ADC 单端输入转换时间为 13 个 ADC 时钟周期,则一次 ADC 转换的时间为 $13/125\text{kHz}=0.11\text{ms}$ 。因此,2ms 的触发转换间隔时间远超出完成一次 ADC 转换时间 0.11ms 与一次数值格式转换时间(Debug 断点显示约 0.07ms)之和,不会影响 ADC 转换过程,同时每秒内完成的 ADC 转换达 500 次。

在 ADC 转换完成中断服务中,采用了整型数将 ADC 转换结果换算成电压值。为了保证计算产生不溢出,先将 adc_data 强行转换成 long 型,然后再乘 5000(这里假定 AVCC 参考电压为 5V),最后再除 1024,保证了换算的正确性。

软件设计：参见代码如下。

```
/* avr_pro_dianyaji.c */
#include <avr/io.h>
#include <avr/interrupt.h>
#define F_CPU 4000000UL
#include <util/delay.h>
#define CNTNUM 0x7C
//共阴数码管段选信号数组 p g f e d c b a
unsigned char led_7[10] = {0x3F, 0x06, 0x5B, 0x4F, 0x66, 0x6D, 0x7D, 0x07, 0x7F, 0x6F};
unsigned char position[6] = {0xfe, 0xfd, 0xfb, 0xf7};
unsigned char display_buff[4] = {0, 0, 0, 0}, posit = 0;
unsigned char adc2disp;
unsigned int adc_v;
//ADC 初始化函数
void AD_Init(void)
{
    //参考电压 AVCC, 电容稳压, 右对齐 ADC5 单端输入
    ADMUX |= (0 << REFS1) | (1 << REFS0) | (0 << ADLAR) | (1 << MUX2) | (1 << MUX0);
    //AD 使能, 连续转换, 中断使能, 32 分频, 4M/32 = 125kHz
    ADCSRA |= (1 << ADEN) | (1 << ADATE) | (1 << ADIE) | (1 << ADPS2) | (1 << ADPS0) | (0 << ADIF);
    //AD 开始转换
    ADCSRA |= (1 << ADSC);
}
// ADC 电压值输出显示缓冲函数
void adc_to_displaybuff(unsigned int adc)
```

```

{
    unsigned char num4,num3,num2,num1;
    num1 = adc % 10;
    num2 = adc % 100/10;
    num3 = adc % 1000/100;
    num4 = adc/1000;
    display_buff[3] = num4;
    display_buff[2] = num3;
    display_buff[1] = num2;
    display_buff[0] = num1;
}
// Timer 0 比较匹配中断服务
ISR(TIMERO_OVF_vect)
{
    PORTB = ~PORTB ;
    TIFR0 = 0x00;
    TCNT0 = CNTNUM;
    // 允许 T/CO 比较中断
    TIMSK0 = 0x01;
    //AD 开始转换
    ADCSRA |= (1 << ADSC);
}
// ADC 转换完成中断服务函数
ISR(ADC_vect)
{
    unsigned int adc_data;
    unsigned char AL,AH;
    AL = ADCL;
    AH = ADCH;
    //读取 ADC 置换结果
    adc_data = AH;
    adc_data = adc_data << 8;
    adc_data = adc_data | AL;
    //电压值换算
    adc_v = (unsigned long)adc_data * 5000/1024;
    adc2disp = 1;
}
//主程序
int main(void)
{
    // LED 显示控制 I/O 端口初始化
    DDRB = 0x20;
    PORTB = 0x20;
    DDRC = 0x0f;
    PORTC = 0x0f;
    DDRD = 0xff;
    PORTD = 0x00;
    // T/CO 初始化
    // 内部时钟,64 分频(4MHz/64 = 62.5kHz),基本模式
    // 允许 T/CO 比较中断
    TCCR0A = 0x00;

```

```

TCCR0B = 0x03;
TCNT0 |= CNTNUM;
TIFR0 = 0x00;
TIMSK0 = 0x01;
// ADC 初始化
AD_Init();
// 开放全局中断
asm("sei");
// 主循环
while (1)
{
    if (adc2disp)
        adc_to_displaybuff(adc_v);
    switch(posit)
    {
        default:
            PORTC = 0x0F & position[posit];
            PORTD = (0 << PD7);
            PORTD = led_7[display_buff[posit]];
            posit++;
            break;
        case 3:
            PORTC = 0x0F & position[posit];
            PORTD = (1 << PD7) | led_7[display_buff[posit]];
            posit = 0;
            break;
    }
    // _delay_ms(2);
}
}

```

5.9.5 思考题

- (1) 如何解决 Proteus 仿真警告 ADC 寄存器锁定？
- (2) 如何解决 Proteus 仿真时，明明已经在处理器属性框中加载了 Hex 程序代码，运行仿真后却无法观察到实验现象的问题？

5.10 AVR 的 Arduino 仿真

Arduino 是众多低成本微控类创客硬件中的优秀平台。有了 Arduino，避免了在单片机学习过程中，长期陷入无尽 Debug、原理图读图等基础例程上。

5.10.1 基本知识点

安装 Arduino IDE，芯片配置 AVR Mega328P；依据开发板原理图，搭建外围电路，下载仿真基础实例；编程完成模块功能并下载验证，如电机、串口等。

5.10.2 实验设备

Arduino IDE、Arduino 开发板。

5.10.3 实验概述

Arduino UNO 是一块采用 ATmega328P 直插式主控初学者开发板,如图 5-10-1 所示。开发板除了拥有主控芯片上计算、存储、接口等丰富资源外,集成了方便的 USB 接口在线可编程下载调试方式。另外,板上提供了 4 组可扩展 I/O 插座包括:电源 1 组 3.3V、5V、GND 和外部 Vin;模拟输入 1 组 A0~A5 对应 ATmega328 芯片内 5 路带 ADC 端口;数字端口 2 组 13~8 号属于 PB 端口,7~0 号属于 PD 端口。PWM 模式有 11、10、9、6、5、3 共 6 处。

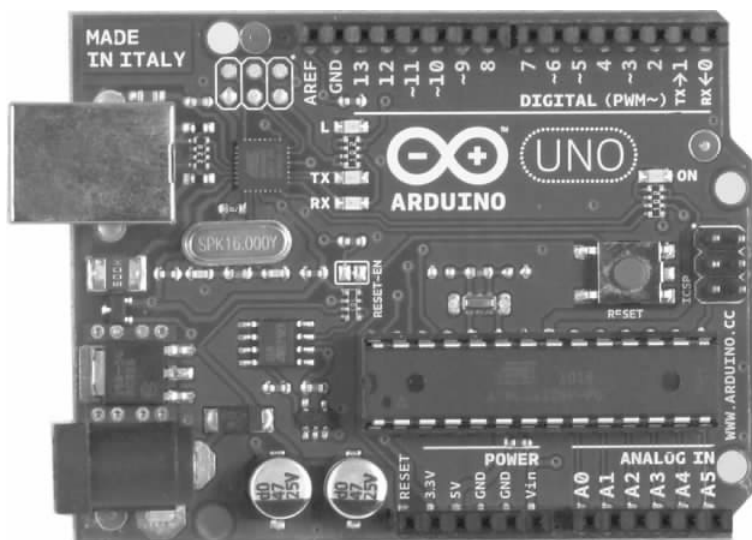


图 5-10-1 Arduino UNO 开发板

1. Arduino 仿真方法

Arduino 开源硬件在创客界十分流行,国内代理商如 DFRobot、Seeed Studio 公司,这些公司提供了丰富的物美价廉的 Arduino 及周边产品。当然,初学者也可以先不着花钱买硬件设备,可以通过仿真方法进行前期学习。

1) 在线仿真

123D 是一个由 Autodesk 和 Circuits.io 合作开发的在线电路仿真和互助设计新平台(2013 年秋)。它满足初学者通用 Windows 浏览器开发 Arduino、Raspberry Pi 或其他平台的需求。用户可以在线访问 Arduino Simulator(<https://circuits.io/>)。

2) Proteus 的 Arduino 元件

Proteus 中用 Arduino 有两种办法:一种方法是根据 Arduino 版本芯片自己画出 Arduino 外围器件,例如 Arduino UNO 用的是 ATmega328 芯片;另一方法是利用官方 Arduino 的 Proteus 元件库,目前只有 Arduino UNO 和 Arduino 2560 两种版本。

2. Arduino IDE 安装与运行

Arduino 官方网站 <http://arduino.cc> 可下载其最新版本开发软件和学习教程。开发软件 Arduino IDE 支持 Windows、MAC OS、Linux 等多种操作系统,本书采用版本 1.6.5。

解压已下载的 Arduino 软件至本地硬盘目录,即可完成安装。如图 5-10-2 所示,每次双击程序 `arduino.exe` 蓝色图标,即可运行 Arduino IDE。



图 5-10-2 Arduino IDE 文件夹

其中,drivers 文件夹存放了 Arduino 控制板的驱动文件,记住其所在路径,供下一步使用。

3. Arduino 驱动安装

第一次通过 USB 数据线缆将 Arduino 开发板连接至计算机 USB 接口时,计算机往往无法识别硬件,还需要安装驱动文件。Windows 7 系统下安装驱动方法如下,其他操作系统下安装方法请另行查阅。

先确保 Arduino UNO 板已经 USB 数据线连接至计算机。

(1) 在 Windows 桌面的“计算机”图标上右击,在弹出快捷菜单中选择“管理”,打开计算机管理器。

(2) 在左侧列表中选中“设备管理器”,右侧会显示设备结果,其他设备中有一项前面带感叹号的未知设备。

(3) 右击未知设备,在弹出的快捷菜单中选择“更新驱动程序软件”,将会弹出“驱动搜

索”对话框。

(4) 在“驱动搜索”对话框中,选择“浏览计算机以查找驱动程序软件”,将会弹出“驱动位置选择”对话框。

(5) 在弹出的对话框中单击“浏览”按钮,选择驱动路径指向 Arduino 解压路径下 /drivers 文件夹,单击“确定”按钮,并稍等片刻,将会弹出“驱动安装”对话框。

(6) 单击“确定”按钮,直到完成 Arduino 驱动文件安装。

驱动正确安装后,设备管理器中未知设备的黄色感叹号消失,端口列表中已经识别出 Arduino UNO(COM8)。这表示 Arduino 已经连接上计算机,控制板的型号是 Arduino UNO,连接端口为 COM8。每次 USB 线缆连接 Arduino 板,计算机可能会分配不同的端口号,用户需要确认当时控制板型号和端口号。

4. Arduino IDE 硬件环境设置

启动 Arduino IDE 开发环境,如图 5-10-3 所示。选择“工具”→“板: "Arduino Uno"”→ Arduino Uno。

在“工具”→“端口”中根据实际端口号配置,如 COM8。至此,Arduino IDE 硬件环境设置完毕,可以准备下一步编写代码。

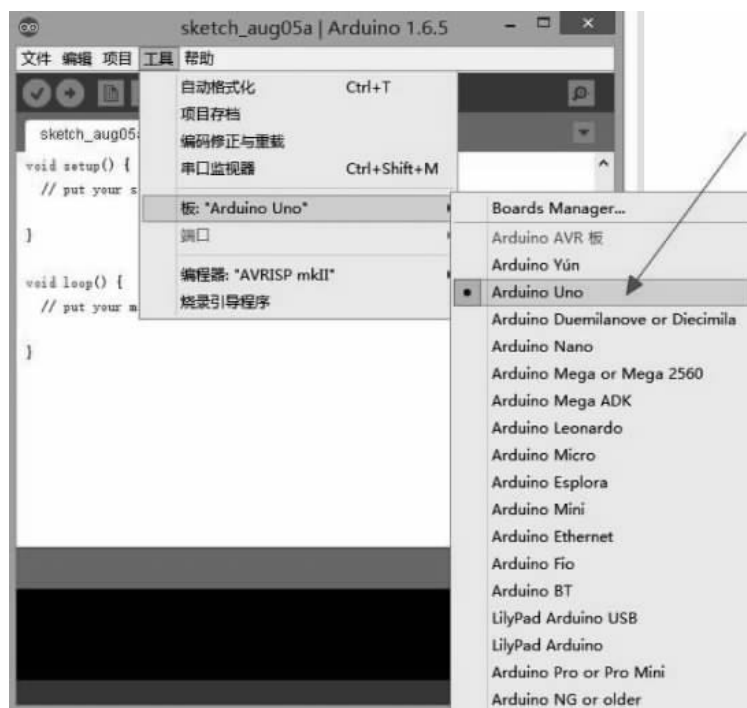


图 5-10-3 IDE 开发板设置

5.10.4 实验内容

BE: 远程开发

内容: 在线完成一个 Arduino 案例开发,观察虚拟设备显示。

分析：打开 <https://circuits.io>, 注册 Autodesk 账号并登录。项目库中有电子项目、PCB、电路原理图项目。打开一个电子项目, 选择案例库中现有的键盘项目 HelloKeypad, 如图 5-10-4 所示。



图 5-10-4 仿真案例硬件

单击 Code Editor 按钮展开编辑代码窗口, 如图 5-10-5 所示, 这里设计了 12 键键盘且串口打印键入值。



图 5-10-5 案例程序

启动仿真,需单击 Upload&Run 按钮经编译成功后,单击 Serial Monitor 按钮打开串口观察窗,如图 5-10-6 所示。依次单击图 5-10-4 中键盘按键可以在串口窗中观察到对应输入键值。

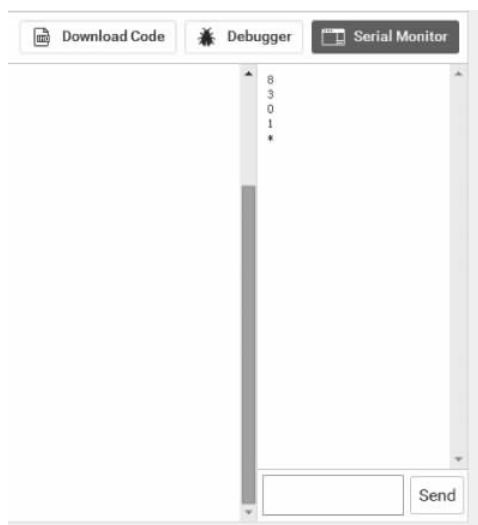


图 5-10-6 仿真运行结果

BE: 本地开发

内容: 产生程序代码 Hex 文件,添加 Arduino 的 Proteus 元件,完成一个 Arduino 案例开发,观察虚拟设备显示。

分析:

1. Arduino IDE 1.6.5 编译器设置

Arduino IDE 默认在编译下载之后自动删除 Hex 文件,这使得软件仿真无法使用。如图 5-10-3 所示,选择“文件”→“首选项”,打开“首选项”对话框。

在“首选项”对话框最下方“在首选项中还有更多选项可以直接编辑”处,单击下一行有 preferences.txt 的文字,会自动打开文件所在路径。

使用记事本打开 preferences.txt 文件,找到文本最后,并添加 build.path = E:\Arduino。E:\Arduino 为 Hex 生成路径,可自行修改。每次 Arduino IDE(CTRL+R)编译通过后,将会在此路径下产生 xx.cpp.hex 文件,可用于单片机直接下载,也可用于 Proteus 仿真。

先关闭 Arduino IDE,然后保存 preferences.txt。

2. 查看示例修改程序

在 Arduino IDE 文件菜单下的示例中选择 01Basics→Blink,打开程序代码,修改端口号配置如下。

```
/*
  Blink
  Turns on an LED on for one second, then off for one second, repeatedly.
```

Most Arduinos have an on-board LED you can control. On the Uno and Leonardo, it is attached to digital pin 13. If you're unsure what pin the on-board LED is connected to on your Arduino model, check the documentation at <http://www.arduino.cc>

```

* /
// the setup function runs once when you press reset or power the board
void setup() {
    // initialize digital pin 13 as an output.
    pinMode(13, OUTPUT);
}
// the loop function runs over and over again forever
void loop() {
    digitalWrite(13, HIGH);      // turn the LED on (HIGH is the voltage level)
    delay(1000);                // wait for a second
    digitalWrite(13, LOW);      // turn the LED off by making the voltage LOW
    delay(1000);                // wait for a second
}

```

3. Proteus 工程建立

Proteus 库：请将库文件解压后复制至 Proteus 软件安装路径，如 E:\ProgramData\Labcenter Electronics\Proteus 8 Professional\LIBRARY 下。新建 Proteus 工程，添加 Arduino UNO R3、Groud、LED-YELLOW 元件绘制电路，如图 5-10-7 所示。

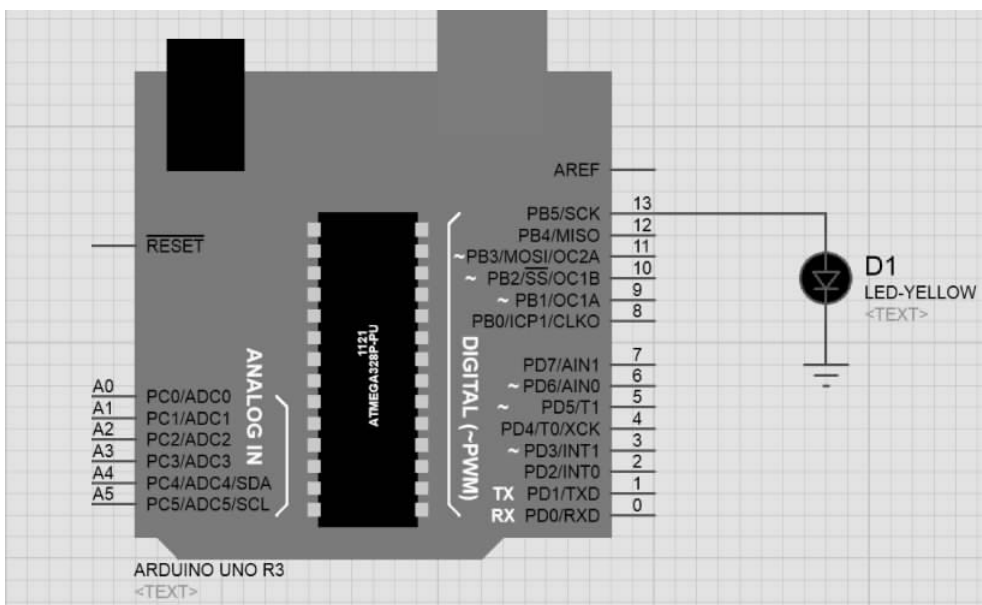


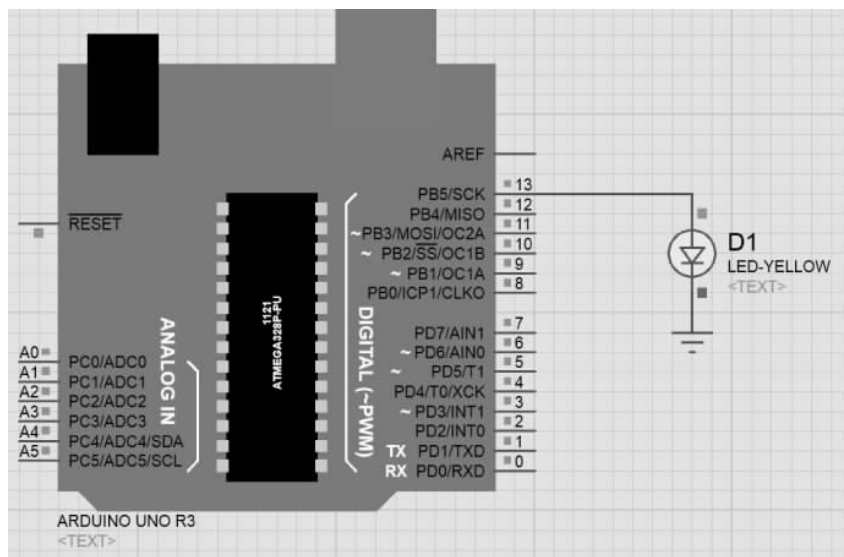
图 5-10-7 Blink 电路图

双击 Arduino UNO，打开 Edit Component 设置对话框，并完成如下设置：Program File 栏中选择 Arduino IDE 生成的 hex 文件，例如：D:\Arduino\xxx.cpp.hex；CKSEL Fuses 栏中下拉选择 (1111)Ext. Crystal 8.0MHz；Boot Loader Size 栏中下拉选择 (00) 1024 words. Starts at 0x1C；SUT Fuses 栏中下拉选择 (10)；Clock Frequency 栏设置为

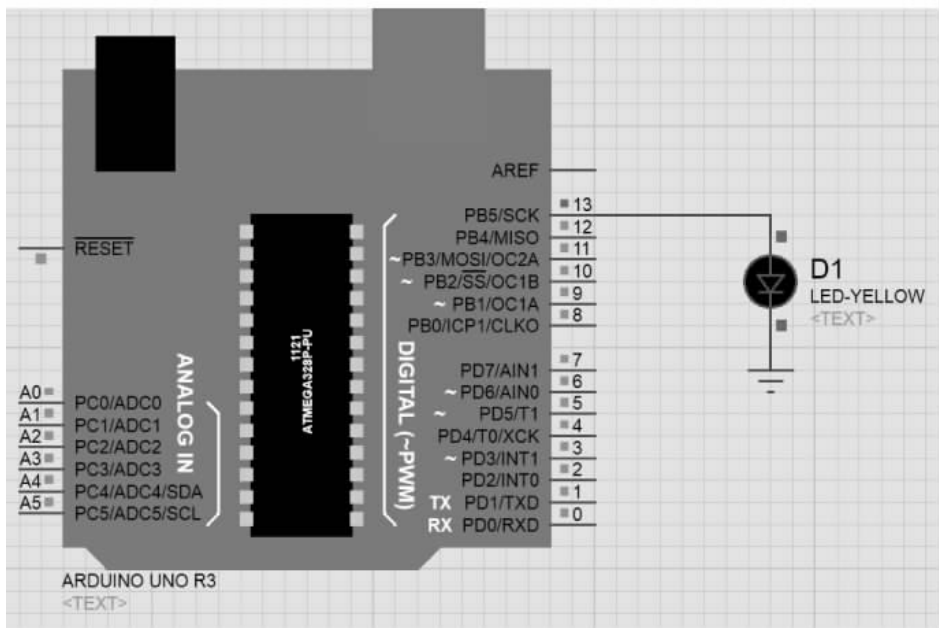
16MHz; 其他可默认设置。

4. Proteus 仿真结果

LED_YELLOW 黄灯 D1 会在亮灯状态(见图 5-10-8(a))与灭灯状态(见图 5-10-8(b))之间反复切换实现闪烁效果。



(a)



(b)

图 5-10-8 LED 闪烁结果

相比 5.8 节实验方法,本节直接调用 Arduino 板,比 Proteus 中调用 ATmega328P 元件的外围电路更简单、直观。Arduino IDE 编程示例丰富,简单易用。

5.10.5 思考题

说明比较本节提供的几种 Arduino 仿真方法,能否另行查找其他仿真工具并说明推荐理由。

5.11 综合实践 2

Arduino 最初被设计用于艺术和设计项目的开发,专注于简化微控制器的编程复杂度,而且并未削减其平台原有的强大处理能力。使用 Arduino,我们可以制作的项目有环境检测、驱动电机、智能循迹、机器人交互、工业机械手、音乐机器人、机器昆虫等。

5.11.1 基本知识点

能在软硬件平台上,深入微处理器和外设应用开发,学习安全加工,自行提出问题设计实验且记录日志报告。

5.11.2 实验设备

Arduino UNO、电机驱动板等。

5.11.3 实验概述

1. 电机驱动模块

L298 是 SGS(通标标准技术服务有限公司)公司的产品,比较常见的是 15 脚 Multiwatt 封装的 L298N,内部包含 4 通道逻辑驱动电路,如图 5-11-1 所示。

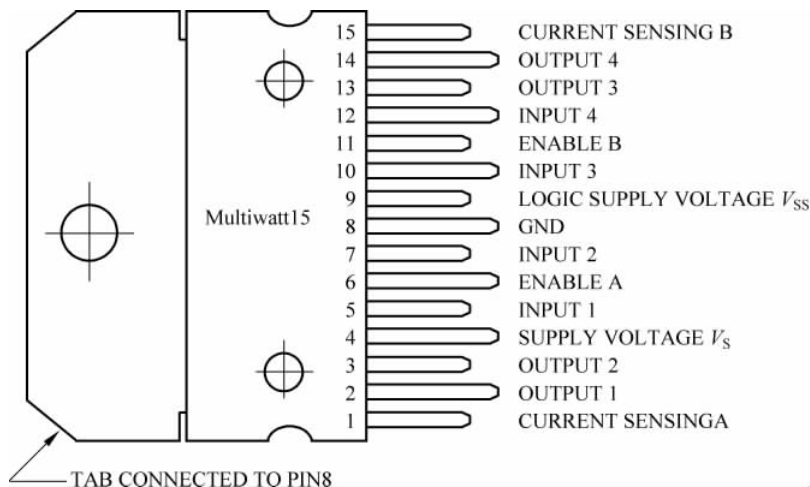


图 5-11-1 L298 芯片

引脚 2、3、13、14 为 L298N 芯片接入到电机的输出端,其中引脚 2 和 3 能控制两相电机,对于直流电机,即可控制一个电机。同理,引脚 13 和 14 可控制另一个直流电机。引脚 6 和 11 脚为电机的使能接线脚。引脚 5、7、10、12 为单片机接入到 L298N 芯片的输入引脚。

L298 是一种二相和四相电机的专用驱动器,即内含 2 个 H 桥的高电压大电流双全桥式驱动器,接收标准 TTL 逻辑电平信号,可驱动 46V、2A 以下的电机,常见驱动模块如图 5-11-2 所示。

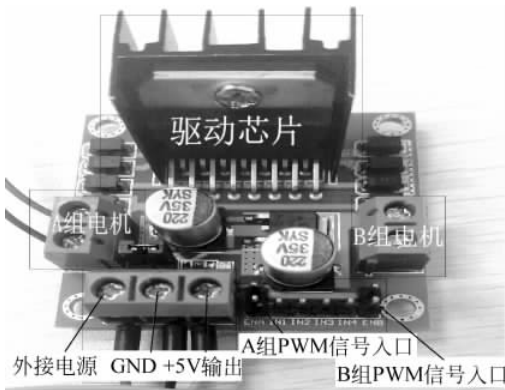


图 5-11-2 L298N 电机驱动板

表 5-11-1 是 L298N 驱动模块使能、输入引脚和输出引脚的逻辑关系。

表 5-11-1 L298N 驱动模块功能表

ENA(ENB)	IN1(IN3)	IN2(IN4)	运转状态
0	X	X	停止
1	1	0	正转
1	0	1	反转
1	1	1	刹车
1	0	0	停止

2. Arduino UNO 与舵机连接

俗称的舵机是伺服电机的一种,SG90 9G 舵机如图 5-11-3 所示,是一种控制角度的驱动器,内部包含直流电机、电机驱动板、角度传感器、减速齿轮组,基本参数为:重量为 9g,尺寸为 23mm×12.2mm×29mm,无负载操作速度为 0.12s/60°(4.8V)、0.10s/60°(6.0V),扭矩为 1.6kg·cm (4.8V),使用温度为-30~+60℃,死区设定为 5μs,工作电压为 3.5~6V,配件为三种功能舵角、固定螺钉,线长 25cm。

舵机 GND 负极(棕色)线、VCC 正极(红色)线、Signal 信号(橙色)线分别连到 Arduino 板上 GND、5V、PWM 模式数字端口(11、10、9、6、5、3 共 6 处,或其他数字端口处经程序虚拟 PWM 模式输出)。



图 5-11-3 SG90 舵机

```
//变量定义
int vi;

//初始化

void setup() {
    // put your setup code here, to run once:
    pinMode(4, OUTPUT);
    pinMode(7, OUTPUT);
    pinMode(10, OUTPUT);
    Serial.begin(9600);
}

//主循环

void loop() {
    // put your main code here, to run repeatedly:
```

```

vi = analogRead(A0);
Serial.println(vi);
digitalWrite(4,LOW);
digitalWrite(7,HIGH);
analogWrite(10,map(vi,0,1023,0,255));
}

```

注意：变量 vi 用以接收变阻器的电压输入，借此调节直流电机转速。数字输入引脚 4、7，用以控制直流电机正反装。数字引脚 10，此处用作 PWM 输出，输出脉宽由变量 vi 大小决定。pinMode() 函数决定数字引脚是输入还是输出。analogRead() 函数读入端口 A0 的模拟电压大小。digitalWrite() 函数向数字端口输出电平。analogWrite() 函数向端口 10 输出 PWM 波。map() 函数将变量 vi 由 0~1023 的范围按比例映射成 0~255。Serial.begin() 确定串口波特率。由于 Serial.println() 函数的存在，可以在 Arduino IDE 中打开串口监视器观察输入电压大小。ArduinoUNO 接收 0~5V 的电压输入，在串口监视器中，可以读到的是模数转换后的(0,1023)间的数值。

BE：舵机控制

内容：Arduino UNO 控制舵机，设计外围电路图，搭建实物，观察并记录功能是否正常情况。

分析：

1. 了解 Servo 类

servo 类成员函数以下：

```

attach();           //挂载舵机,用于设定舵机的接口(范围是 2~13 号)
write();            //角度控制,函数用于设定舵机旋转角度(范围是 0°~180°).这里写入的角
                    //度数值作为舵机直接旋转到角度数值,不是以当前位置为起始点旋转多
                    //少角度数值.舵机通过控制 PWM 的宽度来决定转的角度

writeMicroseconds(); //用于设定舵机旋转角度的语句,直接用 μs 作为参数
read();              //读上一次舵机转动角度,用于读取舵机角度的语句,可理解为读取最后一
                    //条 write()命令中的值

attached();          //判断舵机参数是否已发送到舵机所在接口
detach();             //断开舵机连接

```

2. 代码下载观察舵机运行情况

在 Arduino IDE 软件的菜单“文件”→“示例”中，选择 Servo→Sweep 程序示例，下载观察此示例是否符合 180°舵机旋转要求，如图 5-11-5 所示。

3. 舵机脉冲宽度调制

大多数的舵机都是由脉冲宽度调制(Pulse Width Modulation,PWM)控制的，特殊定制舵机除外。舵机控制板输出的是 PWM 信号，所以舵机控制板都可以控制 180°~360°舵机。

360°舵机由 PWM 波控制旋转速度和旋转方向，脉

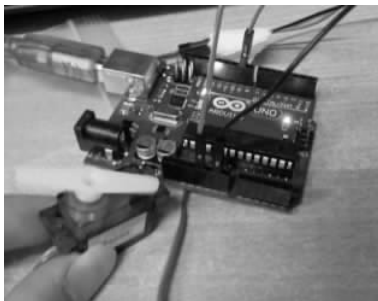


图 5-11-5 舵机运行结果

宽 $1000\sim 1500\mu\text{s}$ 的 PWM 波是控制正转,值越小,旋转速度越大;脉宽 $1500\sim 2000\mu\text{s}$ 的 PWM 波是控制反转,值越大,旋转速度越大。脉宽 $1500\mu\text{s}$ 的 PWM 波是控制停止的标准中位。实际每个舵机的中位可能会不一样,所以有些舵机可能需要脉宽 $1520\mu\text{s}$ 的 PWM 波才能控制停下来,这就需要用户制作过程中实际测量舵机的中位。

360° 舵机是一个普通的直流电机与一个电机驱动板的组合,只能连续旋转,不能定位,也无法知道它的角度和圈数,除非用户在舵机外面另行加装其他传感器。

4. 舵机驱动测试参考代码

(1) 创建舵机对象、引脚约束、正转与反转控制测试。

```
#include <Servo.h>
//创建一个舵机控制对象
//使用 Servo 类最多控制 8 个舵机
Servo myservo;
// 存储舵机角度位置变量
int pos = 0;
//初始化
void setup()
{
    //舵机与 Arduino 引脚连接
    myservo.attach(9);
}
//主循环
void loop()
{
    // 0~180°运动,每次步进 1°
    for(pos = 0; pos < 180; pos += 1)
    {
        //指定舵机转向的角度
        myservo.write(pos);
        // 等待 15ms,舵机到达指定位置
        delay(15);
    }
    //从 180°到 0°运动
    for(pos = 180; pos >= 1; pos -= 1)
    {
        myservo.write(pos);
        delay(15);
    }
}
```

(2) 如果增加舵机数,与上述都是一样的,可以把舵机信号线放在数字端口 2~13 中的任何一个,所以最多可以有 12 个舵机。双舵机对象测试参考代码如下:

```
#include <Servo.h>
// 创建 2 个 servo
// 最多控制 12 个 servo
Servo myservo1;
Servo myservo2;
```

```
// servo 位置变量
int pos1 = 0;
int pos2 = 0;
void setup()
{
    //舵机 1 分配引脚 9
    //舵机 2 分配引脚 10
    myservo1.attach(9);
    myservo2.attach(10);
}
void loop()
{
    for(pos1 = 0; pos1 <= 180; pos1 += 1)
    {
        myservo1.write(pos1);
        myservo2.write(pos1);
        delay(10);
    }
    for(pos1 = 180; pos1 >= 0; pos1 -= 1)
    {
        myservo1.write(pos1);
        myservo2.write(pos1);
        delay(10);
    }
}
```

5.11.5 思考题

(1) 利用两减速电机、轮子、洞洞板、构件等装配一个小车平台,在此基础上加入控制板、电机驱动板、蓝牙通信、传感器等模块实现智能小车硬件,并编写软件完成小车行走控制算法。

(2) 设计一个简易机械臂控制器如图 5-11-6 所示,底座、左臂、右臂、爪四路舵机基本要求:所有舵机有上电起始姿态;前三个转向速度快慢可调;爪子张开角度限位。

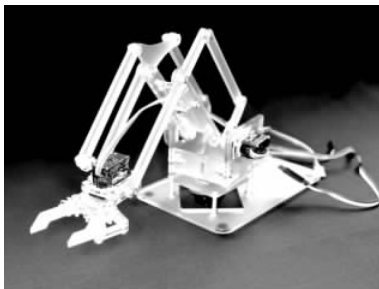


图 5-11-6 简易机械臂