计算机系统综合课程设计

设计报告

组长：尹力超

成员：吴昊昆

彭　程

叶　晗

施　洵

许文星

刘　健

东南大学计算机学院

二０一０年12月

设计名称	MiniSys 系统				
完成时间	2010.12.20	验收时间	2010.12.22	成绩	
本组成员情况					
姓　名	学　号	承　担　的　任　务			个人成绩
尹力超	09007122	Cache 和 FPU 设计			
吴昊昆	09007142	流水设计与系统整合			
彭程	09007144	汇编器设计：词法分析			
叶晗	09007239	FLASH 和 SSRAM 控制器与总体测试			
施洵	09007236	接口设计			
许文星	09007341	汇编器设计：语法分析			
刘健	09007414	汇编器设计：GUI 和二进制码生成			

注：本设计报告中各个部分如果页数不够，请大家自行扩页，原则是一定要把报告写详细，能说明本组设计的成果和特色，能够反应小组中每个人的工作，**尤其要表现出每个同学完成教材中思考题的最高难度系数**。报告中应该叙述设计中的每个模块。设计报告将是评定每个人成绩的重要组成部分之一，因此要在报告中明确标明每个模块的设计者。

设计报告最后一页是验收表和教师综合评价，请大家打印报告的时候将此页一并打印装订。

本组设计的功能描述（含所有实现的模块的功能）

整个课程设计中，我们组主要完成了以下功能模块:

1、 单核 5 级流水 CPU
2、 采用直接映射的 Cache 子系统
3、 带有 FLASH 以及 SSRAM 外部存储结构，并与 Cache 构成两级存储结构
4、 符合 32 位 IEEE754 标准的浮点运算器，实现了浮点加法与减法的功能
5、 汇编 IDE，提供汇编代码的编译、Debug 等功能。

本组设计的主要特色

1、 实现了最初的 32 位 5 级流水结构的 CPU
2、 整个 MiniSys 中加入了存储结构，加大了存储容量，同时又不会影响速度。
3、 实现了 Cache 子系统。Cache 容量为 64B，采用直接映射策略。
4、 在系统中添加了 FPU，从硬件上实现了对浮点运算的支持。同时整个系统也因此成为异构双核结构。
5、 重新封装了 FLASH 和 SSRAM 控制器，实现了控制器与 MiniSys 之间一次 4 字节的交换
6、 设计了一个功能基本完整的汇编器。

本组设计的体系结构（如果只完成基础部分，无创新的可不写）

（包括体系结构框图和对结构图的简要解释，重点解释有自己创新点的部分）

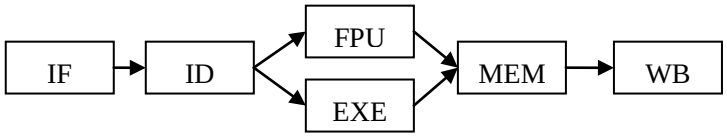
1. Cache 与存储子系统

Cache 子系统采用内部的 LE 作为存储模块，ICache 和 DCache 的大小均为 64B。采用写回法以及直接映射作为整个 Cache 子系统的策略。

整个 Cache 子系统由两部分组成：Cache 模块和后级存储接口。Cache 模块采用设计与 MiniSys 原有的相匹配，系统时钟上升沿时给出命中信号，下降沿时给出实际数据。如果上升沿没有给出命中信号，那么数据延后一个周期。

存储子系统分为 SSRAM 和 FLASH 控制器两部分。每个控制器由上层接口和物理控制器接口组成。FLASH 的物理接口与 FLASH 芯片一次交换一个字节，上层接口将 16 个字节组合，传送给 ICache。SSRAM 控制器与 SSRAM 芯片一次交换 32 位数据，上层接口将 4 个 32 位数据整合，传给 DCache。

2. 5 级流水系统



CPU 设计把一条指令的执行分为：取指、译码、执行、存储和回写五个过程。因此 CPU 为五级流水。为了使每一级流水独立工作，提高 CPU 的执行效率，同时保证整指令的正确执行，必须从四个方面考虑流水线的设计：

A. 每一级流水执行前必须保证上一级流水的执行结果及时传递给该级流水，这样才能保证该级流水及以后的流水和该条指令最终获得正确的结果。在该 CPU 流水线设计中，在时钟脉冲的下降沿上一级流水的执行结果写入与下一级流水之间的寄存器当中。当时钟的上升沿到来的下一级流水读取与上一级之间的寄存器的值。通过这种方式来保证每一级获得的信息都是自来上一级流水的执行结果。

B. 在实验材料上面已经求证过数据相关性只可能出线写后读的数据冲突，这里就不再赘述。需要说明的为了解决写后读的冲突，在 CPU 利用寄存器实现了一个长度为三的队列。由于经过三个周期数据就会写入到寄存器当中，所以只要保证在写入之前数据能够读到更新后的数据就可以了，我们就把该队列的长度设为三。每次执行模块执行完毕则将获得的数据以及寄存器的地址写入到该队列中。需要特别指出的是，LW 这条指令比较特殊，该指令执行内容必须在 EXE 模块后一个模块即 MEM 执行中获得相关的数据，这里 CPU 将插入一条空指令来等待 MEM。将获取的数据写入到队列当中。至于插入过程在后面处理跳转指令的时候一并介绍。

C. 为了保证模块之间的并行性，在 CPU 设计的过程当中就应该满足同一时刻内模块和模块之间的相互独立的。然而在 MIPS 指令集中存在 JAL、JMP、BE、BNE 和 JR 五类跳转指令。当获取跳转指令的时候，在译码的同时并不知道下一条指令的地址，因此下一条指令是一条无用的指令。但是由于在执行过程中，模块之间是相互独立的，这样就产生了跳转指令使流水获取了错误的指令。一般的解决方案主要是有定向转发或者在转移指令结束前，清空当前级间寄存器中的值。如果在跳转指令较多同时流水级数很深的情况下这样会导致整个 CPU 的执行效率变得很低。例如快要被淘汰的 P4 的架构。该 CPU 设计为了提高运

行效率，则利用定向转发的方式来处理跳转指令。

D. 由于整个 CPU 架构中设计了 ICACHE 和 DCACHE。代码中 MEMHIT 和 HIT 分别是 ICACHE 和 DCACHE 多发出的信号。主要是思路是 ICACHE 不命中的时候将会插入空指令来等待，直到获得相关的指令。而 CACHE 不命中的时候 CPU 将会维持当前级间寄存器的所有内容，保存当前现场，直到获得有效数据，然后继续执行。

3. 浮点运算器单元

MiniSys 基于 MIPS 指令架构，因此在设计浮点运算器时，参考了 MIPS 原型机的一些设计。考虑到时间问题，整个设计中对 MIPS 中的浮点运算单元做出如下简化:

浮点与整型共用一条流水线。浮点运算器最为 EXE 阶段的一个子部分、

浮点运算指令与整型指令共用 32 个寄存器，当执行整型运算时，寄存器的内容以 IEE754 标准解释。

整个浮点运算单元分为五个步骤：操作数比较、对阶、符号位生成、尾数运算以及规格化。

4. 汇编器:

采用开源工具 YACC 和 LEX 生成了一个 MIPS 汇编器

本组设计中各个部件的设计与特色概述（含关键代码）

（只完成基础部分的同学只说明完成的设计，无需贴代码，但教材中缺的代码可以贴）

1、 Cache 的设计

Cache 的基本设计思路为：假设所有读写操作均命中，尽可能早地将可能读写的数据准备完成。同时比较地址，得出是否命中的结果。如果命中，则将准备完成的数据写入 Cache 或者返回至 CPU。由于数据准备与地址比较为两个并行操作，因此命中信号可以在系统时钟下降沿之前给出。从而，系统可以在系统时钟的下降沿的时刻读取该信号线并做出相应的操作。下面以 DCache 为例：

```
wire[1:0] idx = addr[5:4];
wire[1:0] sb = addr[3:2];
wire[14:0] ptag = addr[20:6];
```

上述代码对 CPU 发送至 DCache 的地址，其中最低两位不时用。addr[3:2]表示 Cache 中列的选择。addr[5:4]表示行的选择。addr[20:6]为当前地址的标签位，用于目录表的比价。

```
wire vflag, dflag;
wire[14:0] stag;
Mux4_tag m2(
    .sel(idx), .din0(tag0), .din1(tag1), .din2(tag2), .din3(tag3), .dout(stag)
    );
Mux4_flag m3(
    .sel(idx), .din0(vflag0), .din1(vflag1), .din2(vflag2), .din3(vflag3), .dout(vflag)
    );
Mux4_flag m4(
    .sel(idx), .din0(dflag0), .din1(dflag1), .din2(dflag2), .din3(dflag3), .dout(dflag)
    );
wire cmpRst = (ptag == stag);
wire hit = cmpRst && vflag;
```

m2、m3 和 m4 为两个多路选择器，它们根据当前的 idx（行号）、sb（列号）选出当前 Cache 中，对应该 idx 的行的标签、有效位以及脏位信息。cmpRst 比较请求地址的标签信息与当前存储在 Cache 中的标签信息的比较。hit 根据 cmpRst 与 vflag 得出命中信息。

```
wire[31:0] dout0, dout1, dout2, dout3;
Mux4_data m5(
        .sel(idx),
        .din0(data0), .din1(data1), .din2(data2), .din3(data3),
        .dout({dout3, dout2, dout1, dout0}));
wire[31:0] tmpDout;
Mux4 m1(.sel(sb),
        .din0(dout0), .din1(dout1), .din2(dout2), .din3(dout3),
        .dout(tmpDout));
wire[127:0] tmpDin;
Mux4_din m6(
        .sel(sb),.d({dout3, dout2, dout1, dout0}),.din(cpuWdata), .dout(tmpDin)
        );
```

m5 和 m1 为两个多路选择器。根据 idx 和 sb 定位 Cache 中某一个数据，并将其存放至 tmpDout 中，如果 CPU 为读操作，则直接返回该数据，如果 CPU 为写入操作，那么配合 m6 可以快速完成数据的修改。m6 为一个特殊多路选择器，它更具 m5 得出的 Cache line 的内容，以及需要修改的内容，修改整个 Cache line，并保存在 tmpDin 中。

```
always@(posedge start or negedge rst)
begin
    if(!rst) begin
    else begin
        if(hit && ~wren && ~memACC) begin//read hit
            cpuRdata = tmpDout;
        end
        else if(hit && wren && ~memACC) begin //write hit
            case(idx)
            2'b00:  begin tmpWdin = tmpDin; data0 = tmpWdin; dflag0 = 1'b1; end
            2'b01:  begin tmpWdin = tmpDin; data1 = tmpWdin; dflag1 = 1'b1; end
            2'b10:  begin tmpWdin = tmpDin; data2 = tmpWdin; dflag2 = 1'b1; end
            2'b11:  begin tmpWdin = tmpDin; data3 = tmpWdin; dflag3 = 1'b1; end
            default: begin
                data0 = data0;
                data1 = data1;
                data2 = data2;
                data3 = data3;
            end
            endcase
        end
        else if(~hit && ~vflag && ~memACC)              memOpt = 2'b00;
        else if(~hit && dflag && ~memACC && vflag)      memOpt = 2'b01; //Write Back & Read in.
        else if(~hit && ~dflag && ~memACC && vflag)     memOpt = 2'b00; //Only read in.
        else if(~hit && memACC) begin
            case(idx)
            2'b00:  begin data0 = tmpMemDinWire; tag0 = addr[20:6]; vflag0 = 1'b1; dflag0 = 1'b0; end
            2'b01:  begin data1 = tmpMemDinWire; tag1 = addr[20:6]; vflag1 = 1'b1; dflag1 = 1'b0; end
            2'b10:  begin data2 = tmpMemDinWire; tag2 = addr[20:6]; vflag2 = 1'b1; dflag2 = 1'b0; end
            2'b11:  begin data3 = tmpMemDinWire; tag3 = addr[20:6]; vflag3 = 1'b1; dflag3 = 1'b0; end
            default: begin
                tag0 = tag0;
                tag1 = tag1;
                tag2 = tag2;
                tag3 = tag3;
            end
            endcase
            memOpt = 2'b10;
        end
        else memOpt = 2'b10;
    end
end
```

Cache 主状态机，其中 rst 信号相关的控制被隐藏，。rst 中可以设置初始的 Cache 内容。Cache 主状态机使用系统主时钟（20MHz），在根据 hit 信号，处理 CPU 请求。主要分为如下几个部分：读命中，写命中以及不命中。不命中的情况又分为需要写回与不需要写回两种不同的状态。如下：

```
else if(~hit && ~vflag && ~memACC)              memOpt = 2'b00;
else if(~hit && dflag && ~memACC && vflag)      memOpt = 2'b01; //Write Back & Read in.
else if(~hit && ~dflag && ~memACC && vflag)     memOpt = 2'b00; //Only read in.
```

memACC 为后级存储接口控制信号。相关说明会在后面给出。

后级存储控制器部分，通过状态机实现，0 状态为等待状态，接收由上一个 always 块中的操作信号 memOpt。

memOpt 的相关操作如下：

- 00：Cache 不命中，当前 Cache line 无需写回，直接调入新的 Cache line。
- 01：Cache 不命中：执行当前 Cache line 的写回与新 Cache line 的调入。
- 10：Cache 命中或无操作：不执行存储器操作。

Cache 操作 always 块与后级存储控制 always 状态机直接通过两个握手信号实现 memOpt 与 memACC。后级存储 always 是接收 Cache 操作 always 块中的 memOpt 指令，完成数据读取之后设置 memACC 为高。Cache 操作 always 块接收 memACC 信号，当 memACC 为高时，将新的 Cache line 写入，从而引起 hit 信号的重新计算。后级存储控制 always 状态机如下：

状态 0:

等待状态。根据 memOpt 等待 Cache 操作块中发出的指令。

```
        case(state)
        4'b0000: begin
            case(memOpt)
            2'b00:  state = 4'b0001;     //Read in
            2'b01:  state = 4'b1000; //Write back & Read in
            default: begin
                state = 4'b0000;
                memACC = 1'b0;
                memWren = 1'b0;
                memStart = 1'b0;
            end
            endcase
        end
```

读取状态：

```
        4'b0001: begin//Read in.
            memWren = 1'b0;
            memAddr = {11'b00000000000, ptag, idx, 4'b0000};
            memStart = 1'b0;
            memACC = 1'b0;
            state = 4'b0011;
        end

        4'b0011: begin//Read-in starts.0010
            memStart = 1'b1;
            state = 4'b0010;
        end

        4'b0010: begin//0011
            if(memFinish) begin //Read-in finishes
                memStart = 1'b0;
                state = 4'b0110;//Finish 1000
                memACC = 1'b1;//Read Data Missing!
                tmpMemDin = memRdata;
            end
            else state = 4'b0010;
        end

        4'b0110: begin state = 4'b0111;end// memACC = 1'b0; end
        4'b1010: begin
            if(counter == 4'b0011) begin
                state = 4'b0001;
                counter = 4'b0000;
            end
            else begin
                counter = counter + 4'b0001;
                state = 4'b1010;
            end
        end

        4'b0111:begin//0100
            if(memOpt == 2'b10)
                state = 4'b0000;
            else
                state = 4'b0111;
        end
```

需要说明的是：0110 状态为故意设置的等待状态，用于增加延迟。

写入状态:

需写回时，首先执行存储器写入操作，再执行读取操作。

```
        4'b0111:begin//0100
            if(memOpt == 2'b10)
                state = 4'b0000;
            else
                state = 4'b0111;
        end

        4'b1000:begin //Write back 0101
            memWren = 1'b1;
            memAddr = {11'b00000000000, stag, idx, 4'b0000};
            memWdata = {dout3, dout2, dout1, dout0};
            memStart = 1'b0;
            memACC = 1'b0;
            state = 4'b1001;
        end

        4'b1001:begin//Write-back starts. 0110
            memStart = 1'b1;
            state = 4'b1011;
        end

        4'b1011:begin // 0111
            if(memFinish) begin //Write-back finishes
                memStart = 1'b0;
                memWren = 1'b0;
                memACC = 1'b0;
                state = 4'b1010;//Then, read the new block into cache.
            end
                else state = 4'b1011;
        end
```

操作顺序与读取类似。需要说明的是在写入完成时，会跳转至 1010 状态。1010 状态执行四次空操作，用于增加延迟。

2、 FPU 设计

整个浮点运算单元分为五个步骤：操作数比较、对阶、符号位生成、尾数运算以及规格化。

a) 操作数比较:

```
module fCmp(e_a, e_b, cmpRst, eqlRst, e_diff, e_max);
input [7:0] e_a, e_b;
output cmpRst, eqlRst;
output [7:0] e_diff;
output [7:0] e_max;

assign cmpRst = (e_a >= e_b);
assign eqlRst = (e_a == e_b);

assign e_diff = cmpRst ? e_a - e_b : e_b - e_a;
assign e_max = cmpRst ? e_a : e_b;

endmodule
```

该模块得到比较结果 cmpRst 和是否相等的 eqlRst，以及指数差值 e_diff 和最大指数 e_max

b) 对阶模块

```
module fSft(f_a, f_b, f_min, f_max, cmpRst, eqlRst, cmpFrst, e_diff);
input [24:0] f_a, f_b;
input cmpRst, eqlRst;
input[7:0] e_diff;
output [24:0] f_min, f_max;
output cmpFrst;
reg [24:0] f_min, f_max;
reg cmpFrst;

always@(f_a or f_b or f_min or f_max or cmpRst or e_diff or eqlRst)
begin
    if(cmpRst && ~eqlRst) begin
        f_min = f_b >> e_diff;
        f_max = f_a;
        cmpFrst = 1'b1;
    end
    else if (cmpRst && eqlRst) begin
        if(f_a >= f_b) begin f_min = f_b; f_max = f_a; cmpFrst = 1'b1;end
        else begin f_min = f_a; f_max = f_b; cmpFrst = 1'b0;end
    end
    else begin
        f_min = f_a >> e_diff;
        f_max = f_b;
        cmpFrst = 1'b0;
    end
end
endmodule
```

根据 e_diff、cmpRst 以及 eqlRst，得出对阶以后的最大尾数、最小尾数以及表示 a 和 b 哪个尾数较大的 cmpFrst 信号。

c) 符号位生成模块

```
module sGen(s_a, s_b, cmpRst, eqlRst, cmpFrst, s_c);
input s_a, s_b, cmpRst, eqlRst, cmpFrst;
output s_c;

wor s_c;
assign s_c = s_a & s_b;
assign s_c = s_b & ~cmpRst;
assign s_c = s_b & eqlRst & ~cmpFrst;
assign s_c = s_a & ~eqlRst & cmpRst;
assign s_c = s_a & cmpRst & cmpFrst;

endmodule
```

根据 cmpRst、eqlRst、cmpFrst 以及操作数 a 和 b 的符号位可以得出最终 c 的符号。这里的的结果已由卡诺图化简。

d) 浮点运算

```
module fOpt(f_min, f_max, s_a, s_b, f);
input [24:0] f_min, f_max;
input s_a, s_b;
output [24:0] f;

assign f = (s_a == s_b) ? ( f_min + f_max ) : ( f_max - f_min );


endmodule
```

该模块很简单，根据符号位计算最终的尾数。

e) 规格化模块

该模块将最终生成的尾数与阶码规格化，以生成符合 IEEE754 标准的 32 位浮点数值。模块的设计采用的是一种简单的方法。这种方法会花费较

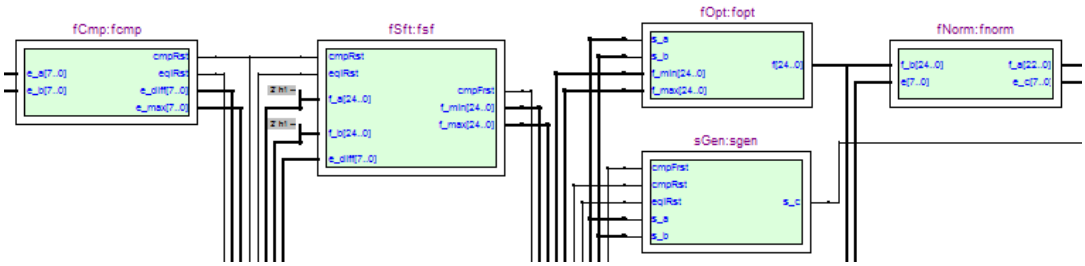
多的 LE 但是运行速度较快。

```
module fNorm(f_b, f_a, e, e_c);
input [24:0] f_b;
input[7:0] e;
output [22:0] f_a;
output [7:0] e_c;
reg[24:0] f_a_tmp;
reg[7:0] e_c;

assign f_a = f_a_tmp[23:1];

always@(f_b or e)
begin
    casex(f_b)
    25'b1_xxxx_xxxx_xxxx_xxxx_xxxx: begin f_a_tmp = f_b;       e_c = e + 8'h01; end
    25'b0_1xxx_xxxx_xxxx_xxxx_xxxx: begin f_a_tmp = f_b << 1;  e_c = e - 8'h00; end
    25'b0_01xx_xxxx_xxxx_xxxx_xxxx: begin f_a_tmp = f_b << 2;  e_c = e - 8'h01; end
    25'b0_001x_xxxx_xxxx_xxxx_xxxx: begin f_a_tmp = f_b << 3;  e_c = e - 8'h02; end
    25'b0_0001_xxxx_xxxx_xxxx_xxxx: begin f_a_tmp = f_b << 4;  e_c = e - 8'h03; end
    25'b0_0000_1xxx_xxxx_xxxx_xxxx: begin f_a_tmp = f_b << 5;  e_c = e - 8'h04; end
    25'b0_0000_01xx_xxxx_xxxx_xxxx: begin f_a_tmp = f_b << 6;  e_c = e - 8'h05; end
    25'b0_0000_001x_xxxx_xxxx_xxxx: begin f_a_tmp = f_b << 7;  e_c = e - 8'h06; end
    25'b0_0000_0001_xxxx_xxxx_xxxx: begin f_a_tmp = f_b << 8;  e_c = e - 8'h07; end
    25'b0_0000_0000_1xxx_xxxx_xxxx: begin f_a_tmp = f_b << 9;  e_c = e - 8'h08; end
    25'b0_0000_0000_01xx_xxxx_xxxx: begin f_a_tmp = f_b << 10; e_c = e - 8'h09; end
    25'b0_0000_0000_001x_xxxx_xxxx: begin f_a_tmp = f_b << 11; e_c = e - 8'h0a; end
    25'b0_0000_0000_0001_xxxx_xxxx: begin f_a_tmp = f_b << 12; e_c = e - 8'h0b; end
    25'b0_0000_0000_0000_1xxx_xxxx: begin f_a_tmp = f_b << 13; e_c = e - 8'h0c; end
    25'b0_0000_0000_0000_01xx_xxxx: begin f_a_tmp = f_b << 14; e_c = e - 8'h0d; end
    25'b0_0000_0000_0000_001x_xxxx: begin f_a_tmp = f_b << 15; e_c = e - 8'h0e; end
    25'b0_0000_0000_0000_0001_xxxx: begin f_a_tmp = f_b << 16; e_c = e - 8'h0f; end
    25'b0_0000_0000_0000_0000_1xxx: begin f_a_tmp = f_b << 17; e_c = e - 8'h10; end
    25'b0_0000_0000_0000_0000_01xx: begin f_a_tmp = f_b << 18; e_c = e - 8'h11; end
    25'b0_0000_0000_0000_0000_001x: begin f_a_tmp = f_b << 19; e_c = e - 8'h12; end
    25'b0_0000_0000_0000_0000_0001: begin f_a_tmp = f_b << 20; e_c = e - 8'h13; end
    25'b0_0000_0000_0000_0000_0000_1xxx: begin f_a_tmp = f_b << 21; e_c = e - 8'h14; end
    25'b0_0000_0000_0000_0000_0000_01xx: begin f_a_tmp = f_b << 22; e_c = e - 8'h15; end
    25'b0_0000_0000_0000_0000_0000_001x: begin f_a_tmp = f_b << 23; e_c = e - 8'h16; end
    25'b0_0000_0000_0000_0000_0000_0001: begin f_a_tmp = f_b << 24; e_c = e - 8'h17; end
    25'b0_0000_0000_0000_0000_0000_0000: begin f_a_tmp = 0;         e_c = e - 8'h18; end
    default: f_a_tmp = 0;
    endcase
end
endmodule
```

浮点运算部件自成流水，如下图:



3、 FLASH 控制器与 SSRAM 控制器

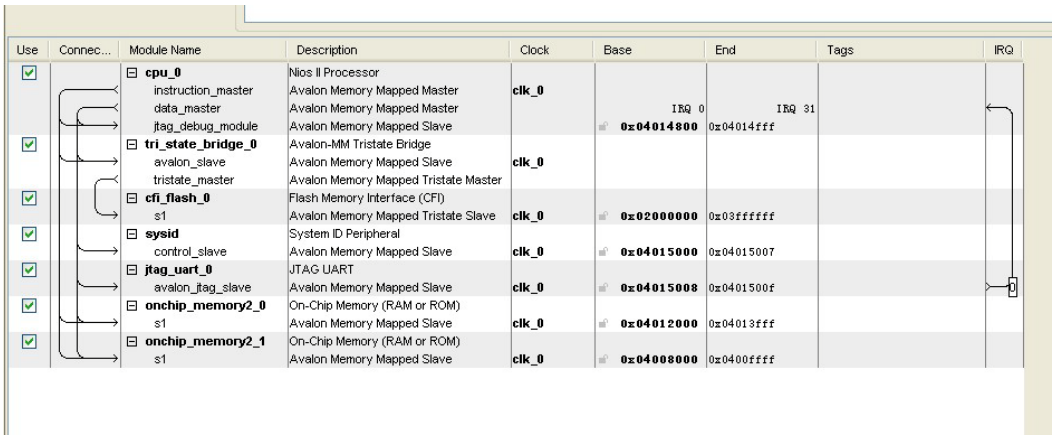
FLASH 一次交换 1 个字节，因此状态机循环 32 次，获取 4 个字，将其整合。

SSRAM 一次交换 4 字节，因此状态机循环 4 次，获取四个字，并将其整合。其中一次循环如下:

```
A1:
begin
writedata = host_in_data[31:0];
cs = 1;
if(host_write)
    write = 1;
else
    read = 1;
counter = 0;
status = A2;
end
A2:
begin
counter = counter + 2'b1;
if(counter > 2)
begin
cs = 0;
status = B1;
address = host_address + 18'h4;
if(host_write)
    write = 0;
else
    begin
    host_out_data[31:0] = readdata;
    read = 0;
    end
end
else
    status  = A2;
end
```

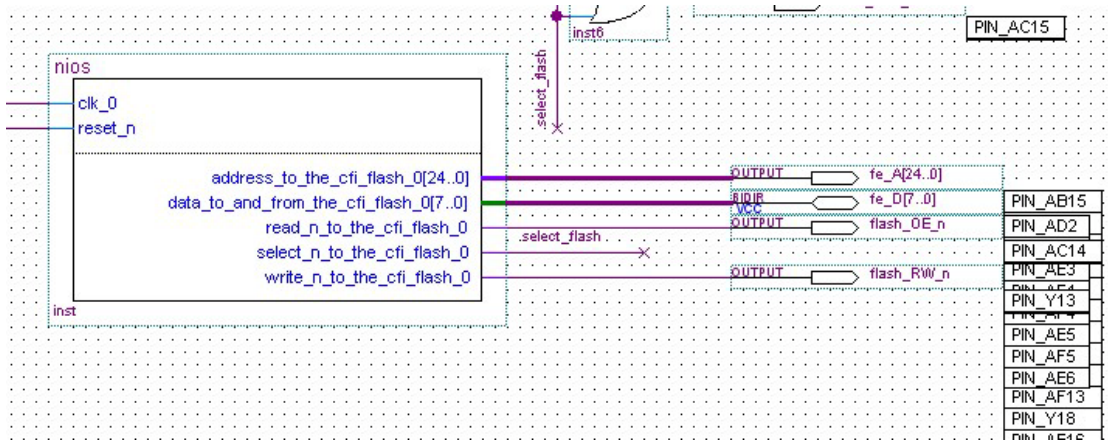
详细内容请参考源代码。

4、 FLASH 烧写程序：FLASH 烧写程序通过 NIOS 软核实现。



Use	Connec...	Module Name	Description	Clock	Base	End	Tags	IRQ
☑		cpu_0	Nios II Processor					
		instruction_master	Avalon Memory Mapped Master	clk_0				
		data_master	Avalon Memory Mapped Master					
		jtag_debug_module	Avalon Memory Mapped Slave		0x04014800	0x04014fff		
☑		tri_state_bridge_0	Avalon-MM Tristate Bridge					
		avalon_slave	Avalon Memory Mapped Slave	clk_0				
		tristate_master	Avalon Memory Mapped Tristate Master					
☑		cfi_flash_0	Flash Memory Interface (CFI)					
		s1	Avalon Memory Mapped Tristate Slave	clk_0	0x02000000	0x03ffffff		
☑		sysid	System ID Peripheral					
		control_slave	Avalon Memory Mapped Slave	clk_0	0x04015000	0x04015007		
☑		jtag_uart_0	JTAG UART					
		avalon_jtag_slave	Avalon Memory Mapped Slave	clk_0	0x04015008	0x0401500f		
☑		onchip_memory2_0	On-Chip Memory (RAM or ROM)					
		s1	Avalon Memory Mapped Slave	clk_0	0x04012000	0x04013fff		
☑		onchip_memory2_1	On-Chip Memory (RAM or ROM)					
		s1	Avalon Memory Mapped Slave	clk_0	0x04008000	0x0400ffff		

所生成的 NIOS 核在 Quartus 中的原理图表示如下



5、 流水设计

在流水设计中，我们主要考虑的三个问题。

1. 每一级流水执行前必须保证上一级流水的执行结果及时传递给该级流水，这样才能保证该级流水以及以后的流水和该条指令最终获得正确的结果。在该 CPU 流水线设计中，在时钟脉冲的下降沿上一级流水的执行结果写入与下一级流水之间的寄存器当中。当时钟的上升沿到来的下一级流水读取与上一级之间的寄存器的值。通过这种方式来保证每一级获得的信息都是自来上一级流水的执行结果。例如:

```
always @(negedge clock or negedge reset )
begin
    if(~reset) ID_EXE=144'b0;
    else if(memhit==1) begin
        if(jmp==1||jal==1)  begin
            ID_EXE[31:0]= 32'b0 ;
            ID_EXE[63:32]= 32'b0;
            ID_EXE[95:64]= 32'b0;
            ID_EXE[127:96]=32'b0;
            ID_EXE[143:128]=16'b0;
        end
        else begin
            ID_EXE[31:0]= instruction ;
            ID_EXE[63:32]= read_data_1;
            ID_EXE[95:64]= read_data_2;
            ID_EXE[127:96]=signed_extend;
            ID_EXE[159:128]={16'b0,opcplus4};
        end
    end
    else ID_EXE=ID_EXE ;
end
```

该段代码就表示 EXE 模块中的执行结果，在下降沿的时候写入到 EXE 模块与 MEM 模块之间的寄存器当中的。下面代码则表示该模块获取该模块执行的任务所需数据以及后面流水所需数据是在上升沿进行的。

```
always@(posedge clock)begin
    if(ID_EXE[44:39]==6'b010001) float_format=1 ;
    else float_format=0;
end

 always@(posedge clock)
 begin
    ADD_Result=ID_EXE[156:141]+ID_EXE[124:109];
    if(read_1==read_2) Zero=1;
    else Zero=0;
    READ_DATA_1=read_1;
    Jrn_EXE=ID_EXE[176];
    Branch_EXE=ID_EXE[173];
    nBranch_EXE=ID_EXE[174];
end

 always@(posedge clock)
 begin
   regwrite=ID_EXE[1];
   regdst=ID_EXE[0];
end
```

2. 在实验材料上面已经求证过数据相关性只可能出现写后读的数据冲突，这里就不再赘述。需要说明的为了解决写后读的冲突，在 CPU 利用寄存器实现了一个长度为三的队列。由于经过三个周期数据就会写入到寄存器当中，所以只要保证在写入之前数据能够读到更新后的数据就可以了，我们就把该队列的长度设为三。每次模块执行完毕，将获得的数据以及寄存器的地址写入到该队列中。需要特别指出的是，LW 这条指令比较特殊，该指令执行内容必须在 EXE 模块后一个模块即 MEM 执行中获得相关的数据，这里 CPU 将插入一条空指令来等待 MEM。将获取的数据写入到队列当中。至于插入过程在后面处理跳转指令的时候一并介绍。下面是核心代码:

```
always@(negedge clock or negedge reset)begin
   if(~reset) begin
       data_3=32'b0;
       data_2=32'b0;
       data_1=32'b0;
       add_3=5'b0;
       add_2=5'b0;
       add_1=5'b0;
   end
   else if(memhit==1)begin
       data_3=data_2;
       data_2=data_1;
       if(From_MEM==1) data_1=rdata ;
       else if(regwrite==1&&float_format==0) data_1=ALU_Result;
       else if(regwrite==1&&float_format==1) data_1=float_result;
       else data_1=32'b0;
       add_3=add_2;
       add_2=add_1;
       if(From_MEM==1) add_1=add_1;
       else if(regwrite==1&&regdst==1) add_1=ID_EXE[28:24];
       else if(regwrite==1&&regdst==0) add_1=ID_EXE[33:29];
       else add_1=5'b0;
   end
end
```

利用寄存器来模拟队列操作解决写后读以及等待 LW 处理过程中 MEM

发出的 FROM_MEM 的信号

3. 为了保证模块之间的并行性，在 CPU 设计的过程当中就应该满足同一时刻内模块和模块之间的相互独立的。然而在 MIPS 指令集中存在 JAL、JMP、BE、BNE 和 JR 五类跳转指令。当获取跳转指令的时候，译码的同时并不知道下一条指令的地址，因此下一条指令是一条无用的指令。但是由于在执行过程中，模块之间是相互独立的，这样就产生了跳转指令使流水获取了错误的指令。一般的解决方案主要是有定向转发或者在转移指令结束前，清空当前级间寄存器中的值。如果在跳转指令较多同时流水级数很深的情况下这样会导致整个 CPU 的执行效率变得很低，例如 Pentium 4 的架构。本 CPU 设计中为了提高运行效率，使用定向转发的方式来处理跳转指令。

```
always @(negedge clock or negedge reset) begin
    if(~reset)begin
        nextPC=18'b0;
        PC=18'b0;
        IF_ID[47:0]=48'b0;
    end
    else if(memhit==1) begin
        Temp_Instruction=Jpadr;
        if(lw==1||jrn==1||branch==1||nbranch==1)  nextPC=PC;
        else  begin
           if((jmp == 1) || (jal == 1))
                  nextPC[17:0] = {IF_ID[15:0],2'b0};
           else if(!((jmp == 1) || (jal == 1)))   begin
                if(((Branch_EXE == 1) && (Zero == 1 )) || ((nBranch_EXE == 1) && (Zero == 0)))
                    nextPC = {Add_result,2'b00};
                else if(Jrn_EXE == 1)
                    nextPC = Read_data_1[17:0];
                else if(fail==1'b1&&fail_cnt==4'b1)
                    nextPC= PC;
                else
                    nextPC=PC+18'd4;
            end
        end
        if(fail==0) PC=nextPC ;
        if(fail==1||lw==1||jrn==1||jmp==1||branch==1||nbranch==1||Jrn_EXE==1||jal==1)
            IF_ID[47:0]=48'b0;
        else  if(((Branch_EXE == 1) && (Zero == 1 )) || ((nBranch_EXE == 1) && (Zero == 0)))
            IF_ID[47:0]=48'b0;
        else begin
            IF_ID[31:0]=Temp_Instruction;
            IF_ID[47:32]=Temp_OpcPlus4;
        end
    end
    else begin IF_ID = IF_ID ; PC=PC ; end
end
endmodule
```

上面是处理跳转指令的核心代码。需要说的是 BE 与 BNE 以及 JRN 这三条指令需要用到寄存器的值，这样同样存在数据相关性的冲突。因此需要两个插入两个空指令才能分析出跳转的 PC 值。所以 JRN 和 JRN_EXE 分别是从 ID 和 EXE 模块发出的，这样做是为了插入两条空指令。而 JMP 和 JAL 只需要插入一条空指令，在 ID 模块中获取到相关信息即可。同时需要指出的是由于整个 CPU 架构拥有 ICACHE 和 DCACHE 的子系统。代码中 MEMHIT 和 HIT 分别是 ICACHE 和 DCACHE 发出的命中信号。主要是思路是 ICACHE 不命中的时候将会插入空指令来等待，直到获相关的指令。而 DCACHE 不命中的时候 CPU 将会维持当前级间寄存器的所有内容，保存当前现场，直到数据获得，然后继续执行。

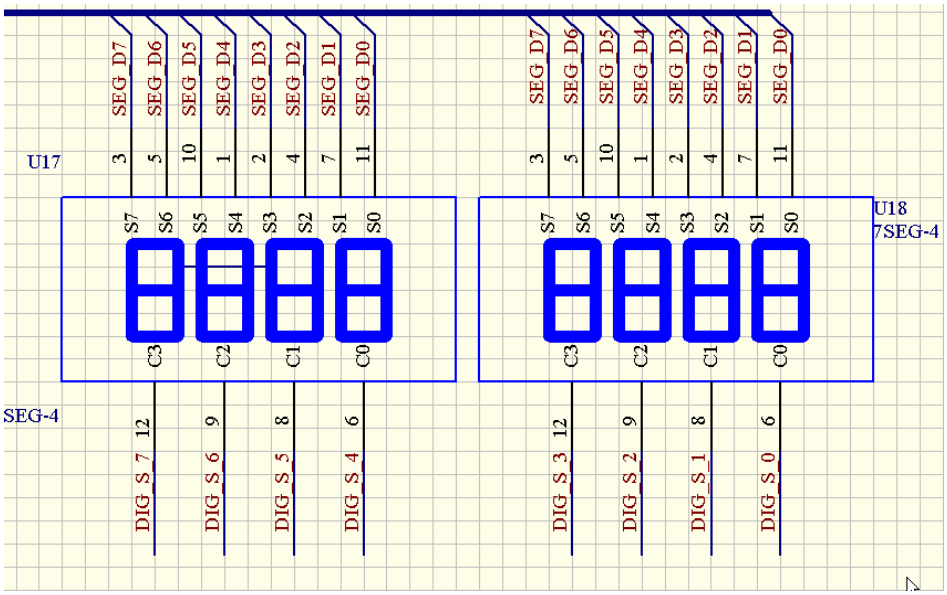
6、 接口设计

由于开发板与书本中的设计不匹配，因此我们对其进行了重新设计

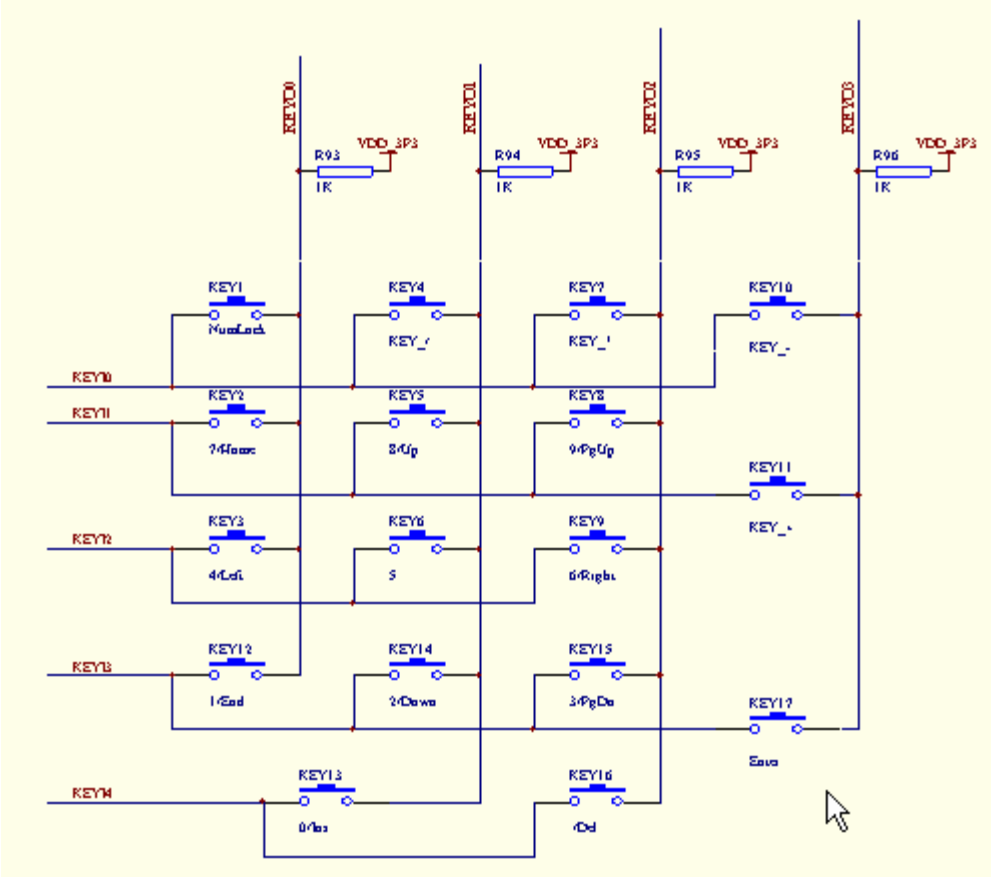
数码管

根据开发板的原理图与说明书，使用的是动态的共阳极七段数码管，

由于是动态的多个数码管，一般来说，多个数码管的连接并不是把每个数码管都独立的与可编程逻辑器件连接，而是把所有的 LED 管的输入连在一起。我们采用扫描的方式输出数码管。



键盘



按键设置在行列线交叉点，行列线分别连接到按键开关的两端。列线通过上拉电阻接 3.3V 电压，即列线的输出被钳位到高电平状态。

判断键盘中有无按键按下式通过行线送入扫描线好然后从列线读取状

态得到的。其方法是依次给行线送低电平，检查列线的输入。如果列线全是高电平，则代表低电平信号所在的行中无按键按下；如果列线有输入为低电平，则代表低电平信号所在的行和出现低电平的列的交点处有按键按下。

7、 汇编器设计

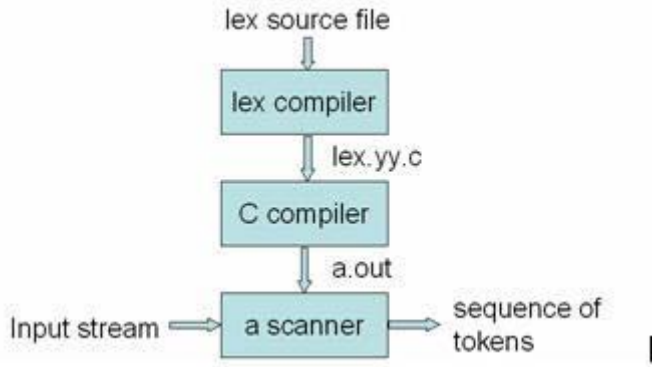
汇编器总体分为四个部分：界面、词法分析、语法分析、二进制代码生成部分。

1. 界面部分

界面基于 MFC 单文档程序进行编写。界面分为菜单栏、工具栏和视图区。菜单栏和工具栏可完成基本的文件打开、保存、新建等操作以及对源程序汇编操作。视图区采用 CSplitterWnd 类，将视图分为四个部分（四个 View），其中一个部分为编辑区，用于显示和编辑源程序，另外的三个部分分别显示对源程序汇编后的相关结果。

2. 词法分析部分

词法分析部分采用 Lex 自动生成。Lex 是 LEXical compiler 的缩写，是 Unix 环境下非常著名的工具,主要功能是生成一个词法分析器(scanner)的 C 源码,描述规则采用正则表达式(regular expression)。描述词法分析器的文件 *.l，经过 lex 编译后，生成一个 lex.yy.c 的文件，然后由 C 编译器编译生成一个词法分析器。词法分析器，简单来说，其任务就是将输入的各种符号，转化成相应的标识符(token)，转化后的标识符 很容易被后续阶段处理。过程如图 。



Lex 文件详见 Asm.l。

- 1-15 行是定义段，主要包括头文件的引用、变量定义、函数声明等。
  - 其中，第 2 行在”mipsAsm.tab.h”中包含了词法分析程序中的常量定义（语法符号的内码）。
  - 第 6 行定义了一个 int 型变量 lineno，用来保存当前分析的汇编程序源文件的行号，以便查到错误时报告位置。
  - 第 7 行是 str2inth 函数的声明，该函数的作用是将字符串转为 16 进制 int 型整数。第 10 行表示让 Lex 生成默认版本的 yywrap() 函数。
  - 12-15 行定义了一些可能在规则段中用到的变量，alpha 表示英文字母，digitd 表示十进制数字，digith 表示十六进制数字，alphanum 表示英文字母或十进制数字。
- 18-242 行是规则段，每行均由一个正规表达式和一段 C 语言程序组成，表示当词法分析器

根据正规表达式匹配到了一个字符串时，执行这段 C 语言程序。

```
\n              lineno++;//18 行表示分析到换行符，lineno 自增。
"data"      return DATA;
…
")"             return RPARA;
```

- 20-39 行是一些汇编伪指令和标点符号，规则很简单，词法分析器只要遇到这些符号直接返回一个常量，把控制权交给语法分析器。

```
"add"       return ADD;
…
"NOP"           return NOP;
```

- 41-105 行是 mips 的 31 条指令（外加一条 nop 指令），同样，当词法分析器分析到这些字符串时，将返回一个常量给语法分析器。

```
"$0"        return REG0;
…
"$RA"       return REG31;
```

- 107-233 行是寄存器的各种表示方法，规则同上。

```
{alpha}{alphanum}*    yylval.strType = new char[strlen(yytext)+1];
strcpy(yylval.strType, yytext); return ID;
```

- 237 行表示词法分析器分析到一个标识符，将把该字符串传给变量 yylval，然后返回一个常量；这里需要解释的是 yytext 是 Lex 的一个内置变量，存放当前分析到的字符串，yylval 是 Yacc 的一个内置变量，表示语法符号的语义属性。

```
{digitd}+{digith}*   yylval.intType = str2inth(yytext); return NUM;
```

- 238 行表示当分析到一个十六进制整数，将其转换为 int 型，然后赋值给 yylval。

```
-{digitd}+{digith}*  yylval.intType = str2inth(yytext); return NEGNUM;
```

- 239 行表示当分析到一个十六进制负整数，将其转换为 int 型，然后赋值给 yylval。

```
[ \t\r]             ;//ignore white space
```

- 241 行表示忽略空白符（包括空格、制表符、回车）。

```
cout<<lineno<<": unexpected character '"<<yytext<<"'\n"; return 0;
```

- 第 242 行表示如果上面 18-241 行所述的正规式都未得到匹配，那么词法分析器将报错给用户。
- 246-252 行是用户定义段。实现了 str2inth()函数。

3. 语法分析部分

语法分析部分采用 Yacc 自动生成。Yacc(Yet Another Compiler Compiler)是 Unix/Linux 上一个用来生成编译器的编译器（编译器代码生成器）。yacc 生成的编译器主要是用 C 语言写成的语法解析器（Parser），需要与词法解析器 Lex 一起使用，再把两部份产生出来的 C 程序一并编译。

Yacc 完整程序见文件 Asm.y。

Yacc 采用 LALR(1)语法分析方法进行语法分析，项目中一并在 yacc 中完成代码生成工作，下面对该程序进行详细解释。

- 1-33 行主要是头文件声明和变量函数定义部分

其中声明了两个在词法分析部分的定义的变量，yyin 是输入文件的

指针，lineno 是行号。以及词法分析函数 yylex()的声明和错误处理函数 yyerror()的声明，数据段输出文件流定义和代码段输出文件流定义。

- 26 行 定义了一个 int 型变量 addr，用于保存指令或者数据所在存储器中的地址。
- 27 行定义了一个预编译标记，因为在实施过程中将先做一次语法分析将所有标号读入保存，因此设定一个标号标记是第一次语法分析还是翻译过程。
- 28 行定义了一个从字符串到整型的一个映射，作为符号表，map 模板的第三个参数是用于比较两个字符串的大小。
- 30-33 行表示语法符号的语意属性可以是 int 型，也可以是 char* 型。

```
%token DATA SEG ENDS ORGDATA DW COMMA ID NUM NEGNUM
…
%token REG16 REG17 REG18 REG19 REG20 REG21 REG22 REG23
REG24 REG25 REG26 REG27 REG28 REG29 REG30 REG31
```

- 35-42 行定义了一些终结符。

```
%type <intType> NUM
%type <intType> NEGNUM
%type <strType> ID
%type <intType> inst;
%type <intType> reg
```

- 44-48 行为一些语法符号指定了语意属性的类型。
- 50-188 行是规则段。这里用上下文无关文法定义了 mips 指令集汇编的语法规则。各条产生式中有花括号围起来的是语义规则。
  这个汇编器是语义制导翻译的，即边分析边输出翻译结果，为了能够处理标号翻译正确的问题，我们在第一次语法分析的过程中只保存所有的标号信息，在第二次编译时再生成翻译的结果。

```
program : dataseg codeseg     {if(!precompile){cout<<"Translation
completed.\n";}}
```

- 52-53 行的产生式是说，一个程序由数据段和代码段组成。

```
dataseg  : DATA SEG dparts DATA ENDS {dataout<<"END;\n\n";
if(!precompile){cout<<"Data segment completed.\n";}}

dparts    : ORGDATA NUM {addr = $2/4;} vars dparts

vars : ID {symbols[$1] = addr*4} DW nums vars

nums    :    nums    COMMA    NUM
 {dataout<<uppercase<<hex<<setw(3)<<setfill('0')<<addr++<<"    :
"<<setw(8)<<setfill('0')<<$3<<";\n";}          | NUM
 {dataout<<uppercase<<hex<<setw(3)<<setfill('0')<<addr++<<"    :
"<<setw(8)<<setfill('0')<<$1<<";\n";}
```

- 55-73 是数据段语法的定义。

```
codeseg : CODE SEG cparts END ID CODE ENDS
  {codeout<<"END;\n\n"; if(!precompile)cout<<"Code segment completed.\n";}
…
| NOP                                {$$ = 0;}
```

- 75-145 是代码段语法的定义。
- 184-188 行定义了错误处理函数，这个汇编器在发现语法错误的时候会报告错误行号。最后是整个汇编器的入口函数 lexmain()。

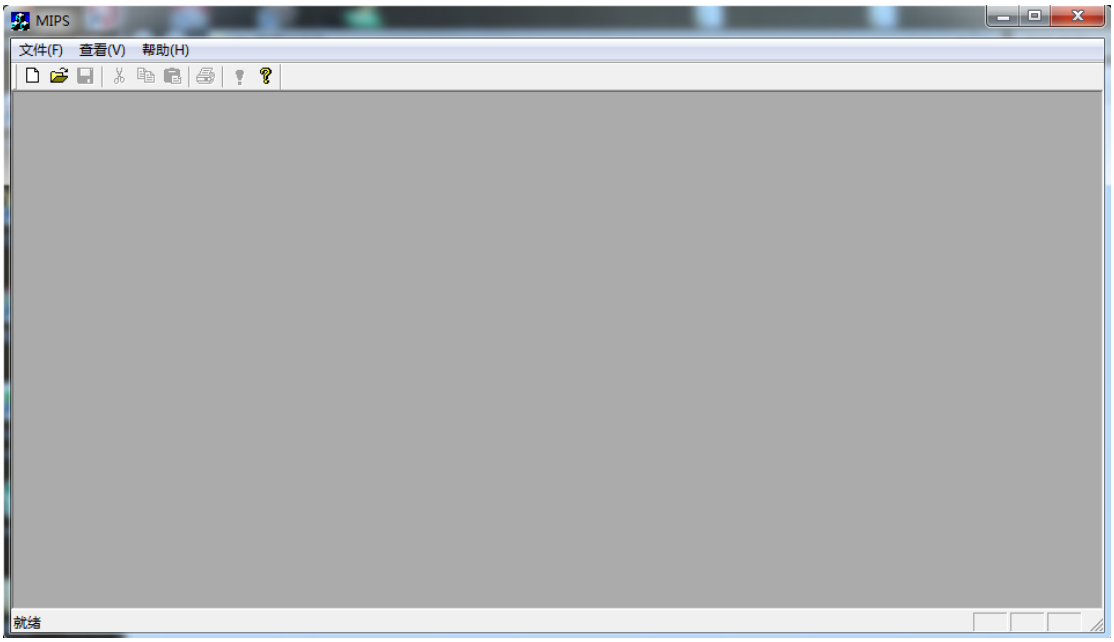
4. 二进制代码生成部分

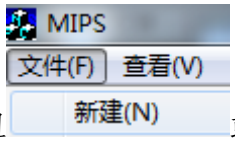
若源程序通过汇编，将运行二进制代码生成函数生成汇编指令对应的二进制文件 out.dat。函数体内定义一个 unsigned int 类的变量 ir 表示一条 32 位二进制的汇编指令。函数读取词法分析部分生成的 prgmip32.mif 文件，寻找到其指令部分。先获取该条指令在内存中的地址，将 out.dat 的文件流指针定位到相应的地址，再将字符串表示的 8 位十六进制的指令转化为 unsigned int 类的变量 ir，并将 ir 写入文件 out.dat。

本组设计的 MiniSys 汇编程序使用手册

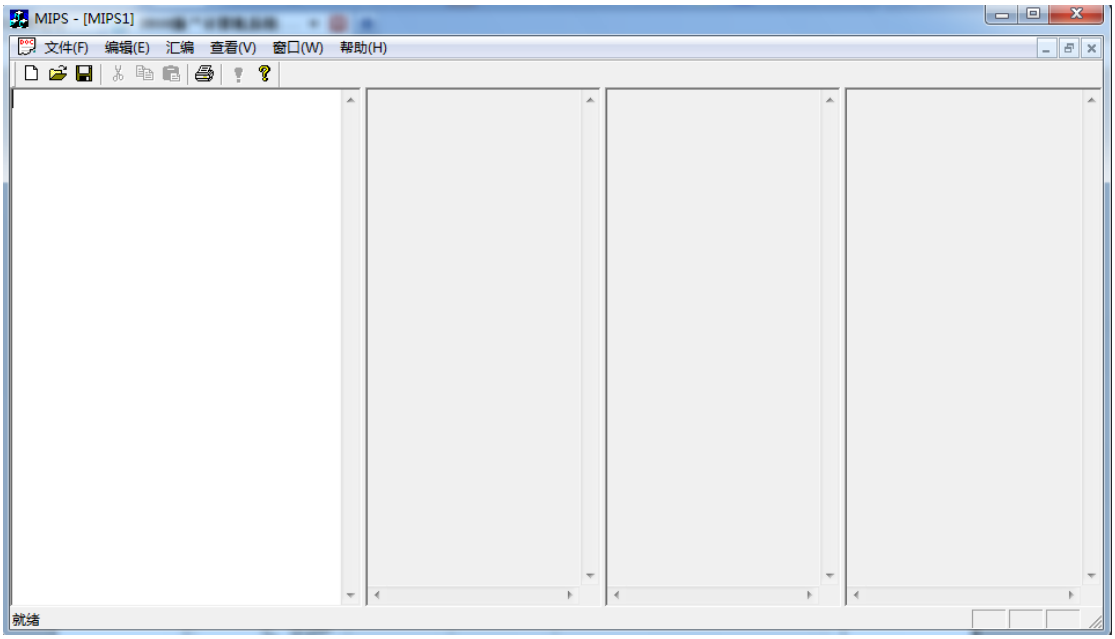
1. 单击图标 MIPS.exe，打开程序

2. 程序初始化界面

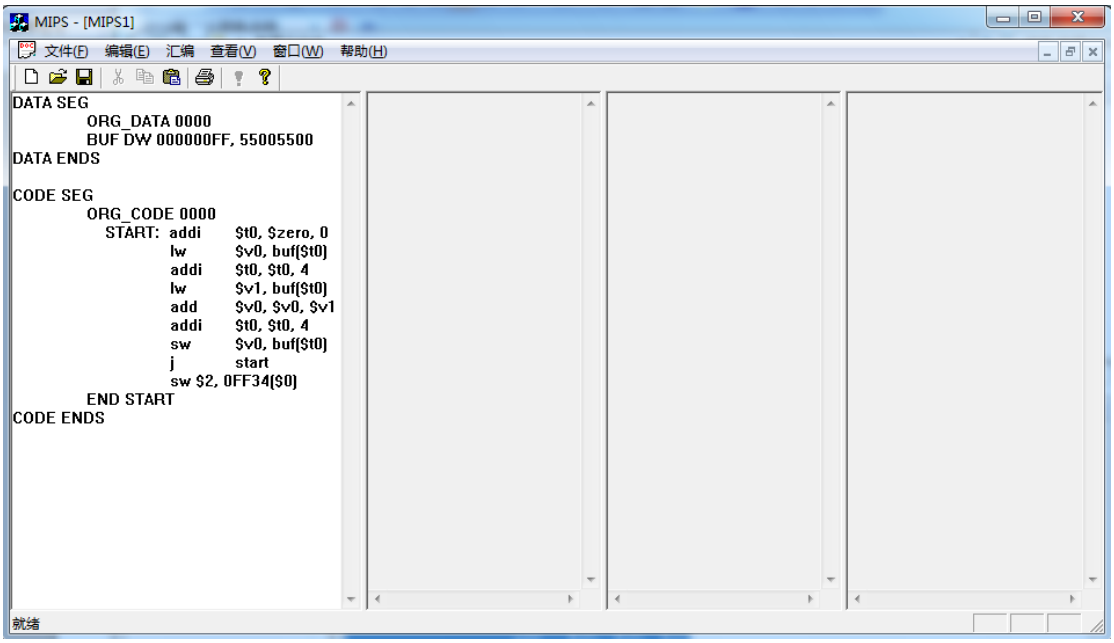


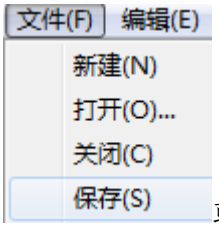
3. 单击菜单栏“新建”按钮 

或者工具栏  ，新建文件

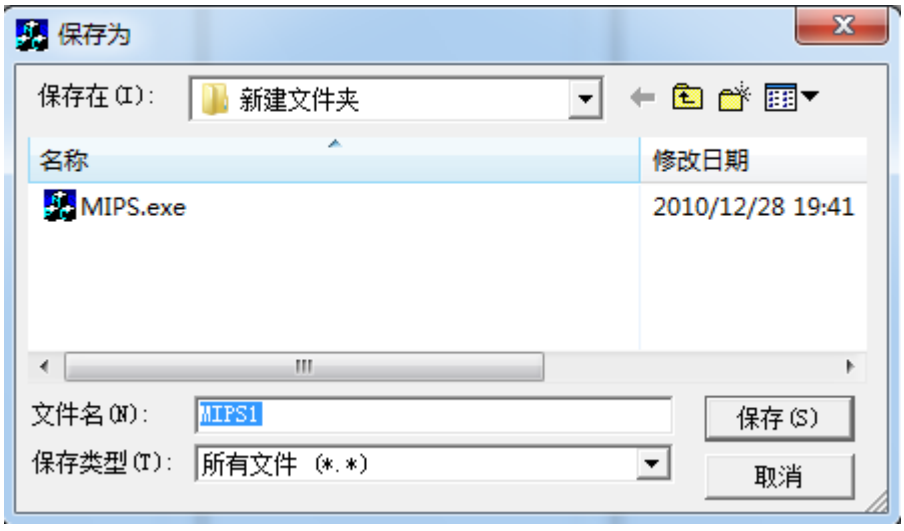


4. 在左侧的编辑框中输入源程序

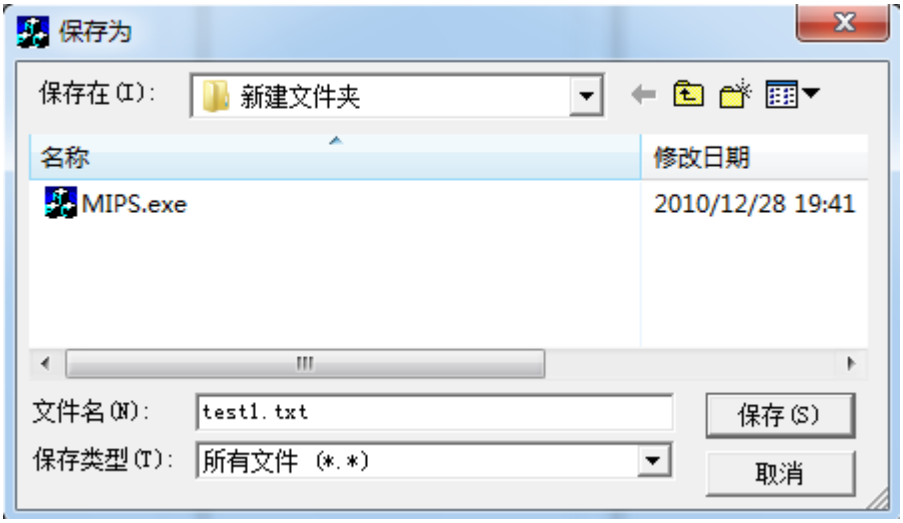


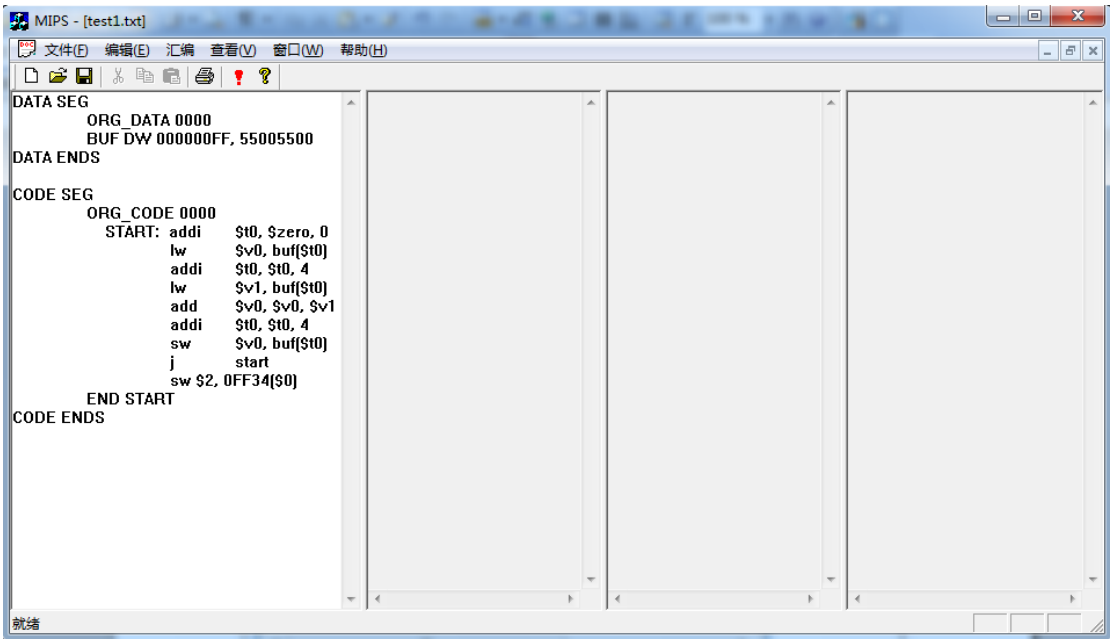
5. 单击菜单栏“保存”按钮， 

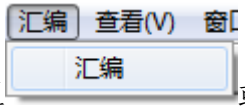
或者工具栏  ，弹出文件保存对话框



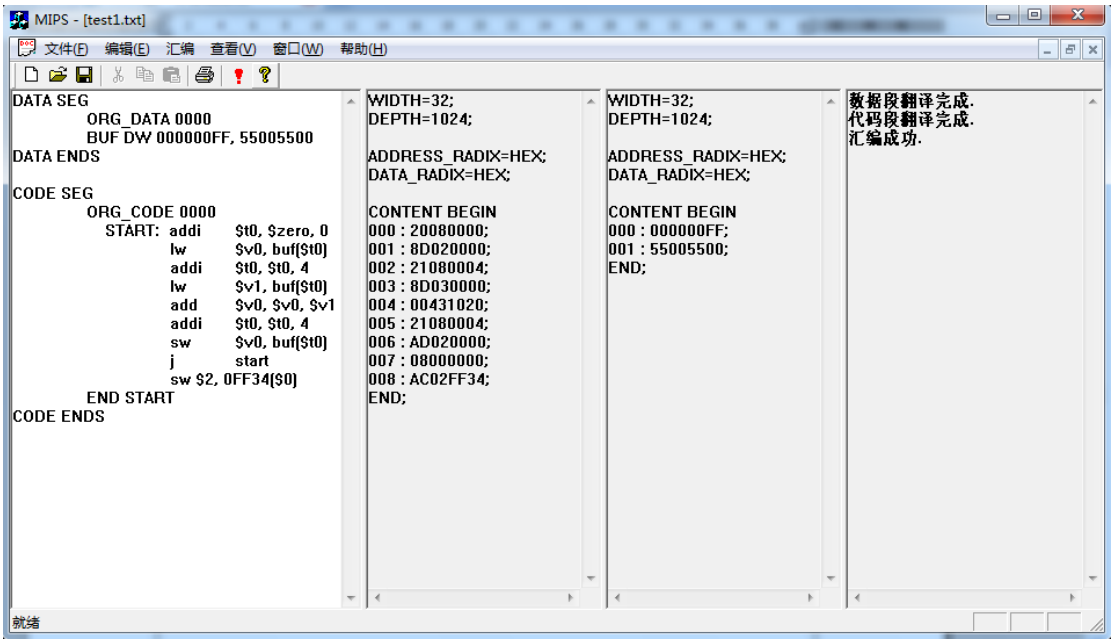
6. 将文件命名为"test1.txt"，保存在“新建文件夹”下，单击保存



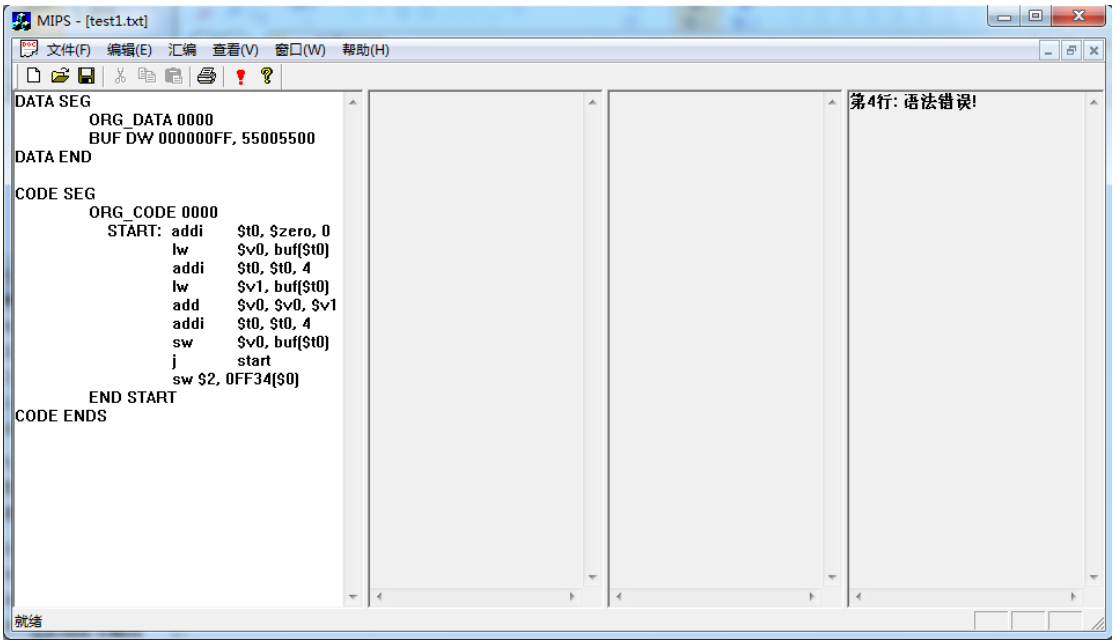


7. 单击菜单栏“汇编”按钮 

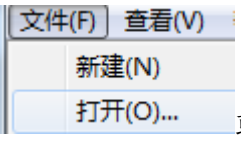
或者工具栏  ，对源程序进行汇编若源程序正确，则显示相应的信息



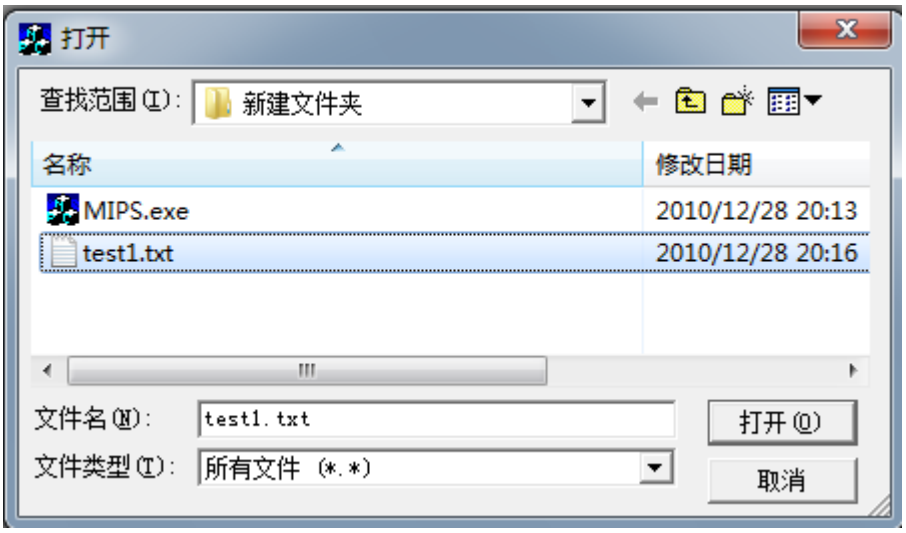
若源程序错误，则报错



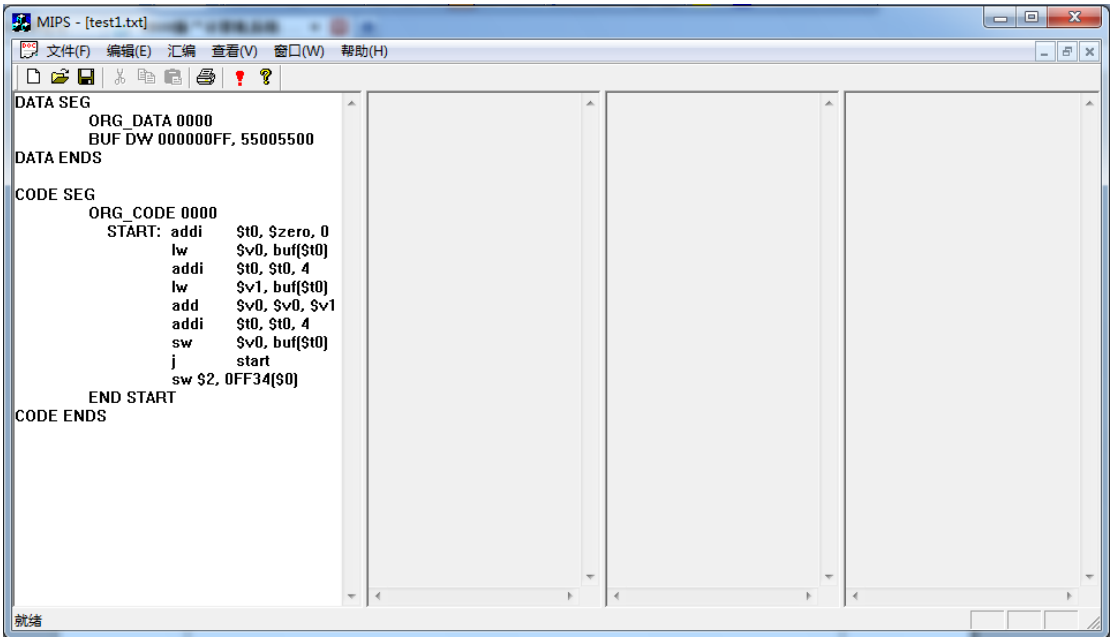
8. 对源程序修改后可直接单击汇编按钮进行汇编

已存在的源文件，也可单击菜单栏“打开”按钮 

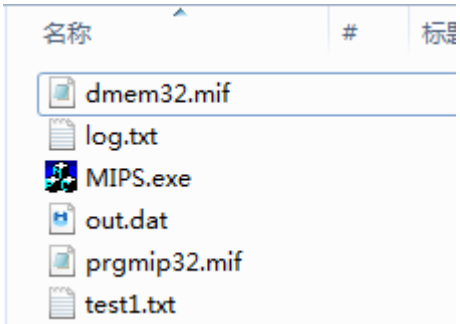
或者工具栏  ，弹出打开文件对话框



选择文件"test1.txt"打开



若源程序通过汇编，会在源程序的文件夹下生成相应的文件

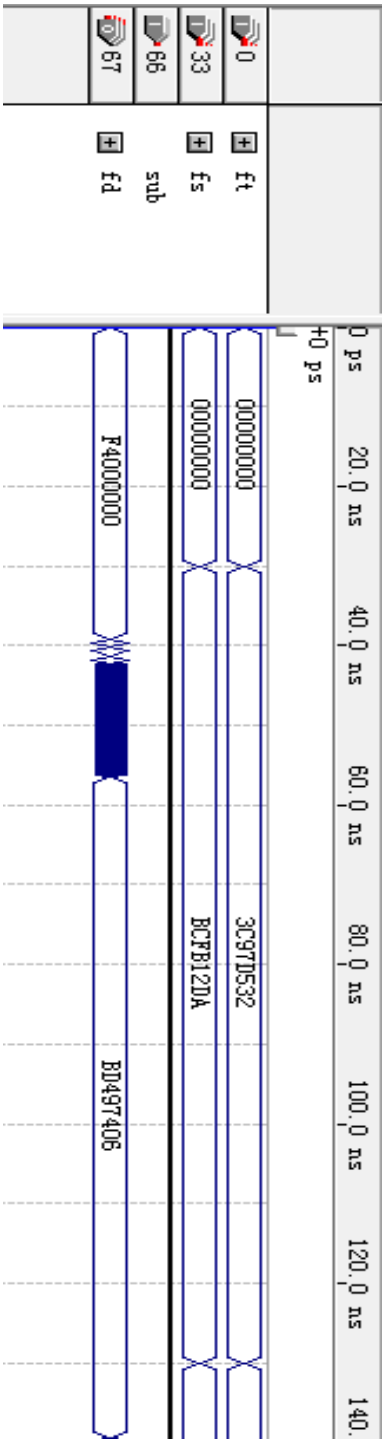


本组设计主要测试结果（.vwf）

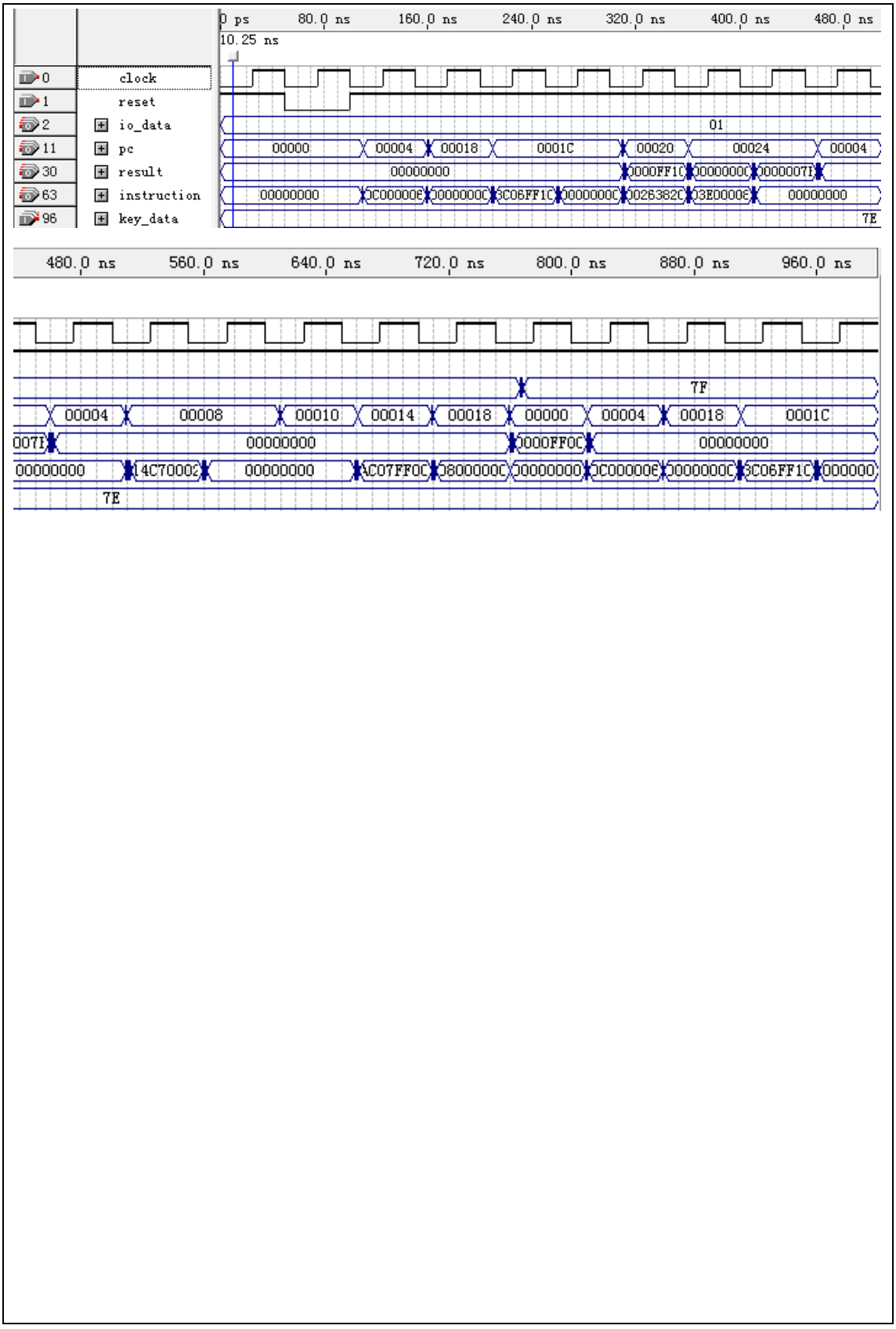
（贴关键的能说明问题的时序图，可以竖贴）

ICache、DCache 单独测试时序：见附录

FPU：单独测试时序



MiniSys 流水时序图


0 ps
80.0 ns
160.0 ns
240.0 ns
320.0 ns
400.0 ns
480.0 ns
10.25 ns
0 clock
1 reset
2 io_data
11 pc
30 result
63 instruction
98 key_data
01
00000
00004
00018
0001C
00020
00024
00004
00000000
00000000
7E
480.0 ns
560.0 ns
640.0 ns
720.0 ns
800.0 ns
880.0 ns
960.0 ns
7F
00004
00008
00010
00014
00018
00000
00004
00018
0001C
00000000
00000000
7E

本组设计的性能分析

（资源使用情况、主频、功耗数据和自我分析，可参考验收表的项目测试）
采用 Quartus 9.1 编译的结果可能与 Quartus 9.0 不太相同:

Flow Status	Successful - Wed Dec 29 15:55:46 2010
Quartus II Version	9.1 Build 222 10/21/2009 SJ Full Version
Revision Name	minisys
Top-level Entity Name	top
Family	Cyclone II
Device	EP2C35F672C8
Timing Models	Final
Met timing requirements	No
Total logic elements	7,491 / 33,216 (23 %)
Total combinational functions	5,861 / 33,216 (18 %)
Dedicated logic registers	3,727 / 33,216 (11 %)
Total registers	3727
Total pins	129 / 475 (27 %)
Total virtual pins	0
Total memory bits	0 / 483,840 (0 %)
Embedded Multiplier 9-bit elements	0 / 70 (0 %)
Total PLLs	2 / 4 (50 %)

功耗分析

PowerPlay Power Analyzer Status	Successful - Wed Dec 29 15:56:33 2010
Quartus II Version	9.1 Build 222 10/21/2009 SJ Full Version
Revision Name	minisys
Top-level Entity Name	top
Family	Cyclone II
Device	EP2C35F672C8
Power Models	Final
Total Thermal Power Dissipation	199.34 mW
Core Dynamic Thermal Power Dissipation	50.32 mW
Core Static Thermal Power Dissipation	80.22 mW
I/O Thermal Power Dissipation	68.80 mW
Power Estimation Confidence	Low: user provided insufficient toggle rate data

Cache 单独测试:

Timing Analyzer Summary

	Type	Slack	Required Time	Actual Time	From	To	From Clock	To Clock	Failed Paths
1	Worst-case tsu	N/A	None	9.853 ns	addr[4]	IC...	--	start	0
2	Worst-case tco	N/A	None	13.753 ns	ICache:cache|ta...	hit	start	--	0
3	Worst-case tpd	N/A	None	17.432 ns	addr[4]	hit	--	--	0
4	Worst-case th	N/A	None	-1.074 ns	rst	IC...	--	CLK_...	0
5	Clock Setup: 'start'	N/A	None	161.97 MHz (period = 6.174 ns)	ICache:cache|ta...	IC...	start	start	0
6	Clock Setup: 'CLK_50Mhz'	N/A	None	243.96 MHz (period = 4.099 ns)	flash_control:flas...	fla...	CLK_...	CLK_...	0
7	Total number of failed paths								0

其中 start 信号即 Cache 的主时钟信号。

课程设计总结

（包括设计的总结和还需改进的内容以及收获）

总的来说，在全体小组成员的努力下，本机完成了设计报告中的大部分内容。同时，包括 5 级流水 CPU、Cache 存储子系统、SSRAM 和 Flash 控制器、汇编器等等。

下面主要谈谈需要改进的地方吧：

1、 由于 Cache 以及片外存储器的加入，导致了流水实现难度的增加。因此 CPU 本身没有太多的硬件优化。因此通过对流水部件的线路优化和结构优化，尤其是 EXE 部件的优化，可以大大提高主频，降低功耗。

2、 接口部分相对简单，测试期间，由于总是出现一些未知问题，因此进行上板调试的进度很慢。种种原因，最终只实现了矩阵键盘和 LED 的功能。可以充分利用开发板上的资源，如 LCD 液晶屏等，增加系统的应用。

3、 Cache 部分采用的直接映射策略虽然速度快，但是冲突概率高，因此可以改变策略算法，提高命中率。

4、 汇编部分。虽然可以编译，但是 IDE 功能较弱，没有很好的代码提示、Debug 纠错功能。

小组成员签名：

验　收　报　告（此表由验收人员填）

主频	MHz	逻辑单元数	(　　%)	功　耗	mW
CPU 类型		☐单周期　☐多周期　☐流水线　☐超标量			
CPU 设计	☐通过　☐未通过		接口电路设计	☐通过　☐未通过	
汇编器设计	☐通过　☐未通过		合　成	☐通过　☐未通过	
中断	☐有　☐无　☐未通过		BIOS	☐有　☐无　☐未通过	
应用软件	☐有　☐无　☐未通过		验收答辩	☐通过　☐未通过	
加分项目					
存在问题					
验收结论	☐优秀　☐良好　☐中等　☐及格　☐不及格				
验收人签字					

教师综合评价:

教师签名: _____