计算机系统综合课程设计

设计报告

组长：牛　星

成员：胡　诚

蒋若冰

苏艳珺

张　洁

王志斌

二〇一〇 年 一 月

设计名称	基于Minisys的异构多核片上系统				
完成时间		验收时间		成绩	
本组成员情况					
姓　名	学　号	承　担　的　任　务			个人成绩
胡　诚	09006229	编写增添指令后的汇编器内核、使用Qt设计汇编器IDE			
蒋若冰	09006302	设计流水主核的译码、回写和寄存器组单元、设计新指令、设计各阶段的测试汇编程序并协助CPU核心的调试工作			
苏艳珺	09006303	设计接口控制模块、设计可实际运行的接口驱动(键盘和彩色LCD)、BIOS编写、负责CPU与接口联调工作			
张　洁	09006307	设计多核CPU的数据Cache、指令Cache、内存模块；参加多核互联的设计和调试工作			
王志斌	09006308	设计可实际运行的接口驱动（LED七段数码管）			
牛　星	09006312	设计系统总体架构、设计流水主核的取指、执行和存储单元、设计辅核、设计辅核、接口连接部件、主持系统联调			

一、 本组设计的功能描述（含所有实现的模块的功能）

1. CPU 部分

a) 主运算控制核

本 CPU 采取异构核架构，主核在开机时就接管计算任务，当指令中有多线程任务时，主核通过辅核任务分配模块将任务传递给空闲的辅助运算核。整个系统的中断也由主核负责响应。

b) 辅助运算核

接管主核分配来的任务，执行结束后会向主核和辅核分配单元发送完成信号（或置标志位）。

c) 辅核任务分配模块

根据辅核的空闲状态，提供指向一个可分配核的“句柄”，供主核操作 PC。

d) Cache

采用哈佛结构，指令 Cache（ICache）与数据 Cache（DCache）分开，ICache 位于 CPU 与指令 ROM 之间作为指令缓冲，DCache 位于 CPU 与数据 RAM 之间作为数据缓冲。

e) 总线控制器

ICacheBus 和 DCacheBus：在多核系统中，针对多个 Cache 对 RAM/ROM 读写操作的竞争，设定各个 Cache 访问的优先级。此外 DCacheBus 还用于解决多核 Cache 间的数据一致性问题，作为侦听进行的主要模块。

f) 接口占用分配模块

当辅核接口占用冲突时，负责协调工作。

2. 接口部分

a) 彩色 LCD 控制器

以黑底白字的效果显示并且提供滚屏功能；每个字符显示大小为 16*8，整个屏幕显示 19*60 个字，屏幕像素为 304*480；每次用户显示数据默认为一排（以回车键作为结束标志）。

b) 5*4 键盘控制器（中断）

通过扫描方式读取用户按键信息，并且将键编码转化成相应键值；工作时会发出 INT0 中断。

c) 8 位七段数码管控制器

根据人眼的视觉暂留，依次扫描显示八个数码管达到多位显示；一共提供三种显示方式，即左边四位显示、右边四位显示及八位全显示。

d) 定时器控制器

提供以秒为单位的定时功能；工作时会发出 INT1 中断。

e) 接口控制模块

根据从 CPU 传来的地址，采用集中译码方式，生成所选接口的片选信号；根据从 CPU 传来的读写信号，控制接口数据的流向；选择向 CPU 写数据的接口，并将数据送到总线以供 CPU 读取

3. 汇编器部分

a) 词法分析模块

每次从文件读取程序代码的一个字符，连成一个 token，并将 token 转化成统一的格式，返回相应的 token 值。

b) 语法分析模块

根据语法栈顶的元素和当前的 token 来分别进行推导或者归约。如果匹配成功，则进行翻译，否则进行错误检查和处理。

c) 机器码生成模块

将每条指令翻译成相应的机器码，包括越界和溢出检查。

d) 读写文件模块

汇编开始时读取各种 log 文件，加载工程原有的 RAM 和 ROM 信息；汇编成功后写入相应的 log 文件，写入 RAM 和 ROM。

e) 用户界面模块

提供用户界面，提供打开、保存、汇编 BIOS、汇编 INT0、汇编 INT1、初始化用户程序、添加用户程序等功能，提供关键字功能，实时显示汇编信息和 RAM、ROM 的状态。

4. 其他

a) BIOB

提供开机自检，主要为 RAM 及 LED 的检测；系统启动自举。

二、 本组设计的主要特色

1. CPU 部分

a) 异构核架构，单主核，多辅核；
b) 主核充当操作系统功能，独立完成任务分配和中断响应；
c) 各个计算核均为 5 级流水；
d) 为线程开启、结束；接口占用、释放等设计了新的指令；
e) 采用 1 级 Cache；
f) 各模块均使用 Verilog 数据流描述；
g) Cache 替换策略采用伪 LRU 替换算法，是对 LRU 算法的简化；
h) 在数据 Cache 的一致性保持中，实现了 MESI 协议；
i) 当辅核接口占用冲突时，有模块负责协调工作。

2. 接口部分

a) 充分利用了开发板上资源，拓展使用了彩屏；

b) 键盘扫描具有防抖功能，可以得到稳定的数据；

c) 7 数码管提供多种显示方式。

3. 汇编器部分

a) 内核大量采用 STL 的 map 和 vector 容器，以简化程序，提高可读性，界面采用 QT 开发，简洁强大；

b) 词法分析过程中，统一了格式：将所有字符转化为小写，以支持大小写；支持各种进制，将所有数字的格式统一为（符号）（二进制码）（b），以方便后面的处理；

c) 支持多行查错，即使有语法错误，只要不是致命的，仍然可以继续进行，而且每行报错是独立的，程序前面的错误不会影响到后面的处理；

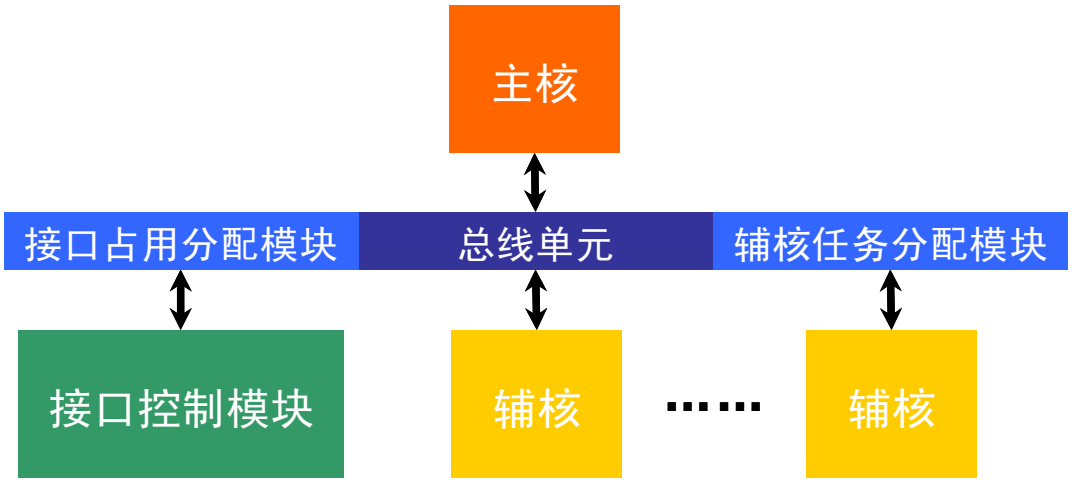
d) 细化错误类型，在报错的时候参考了语法规则，使得报错更加精确，各种类型的错误统一标号，在错误处理程序中进行处理；

e) 通过读写 log 文件，支持程序的链接，并规定了汇编程序的顺序：BIOS、INT0、INT1、初始化用户程序、添加用户程序；

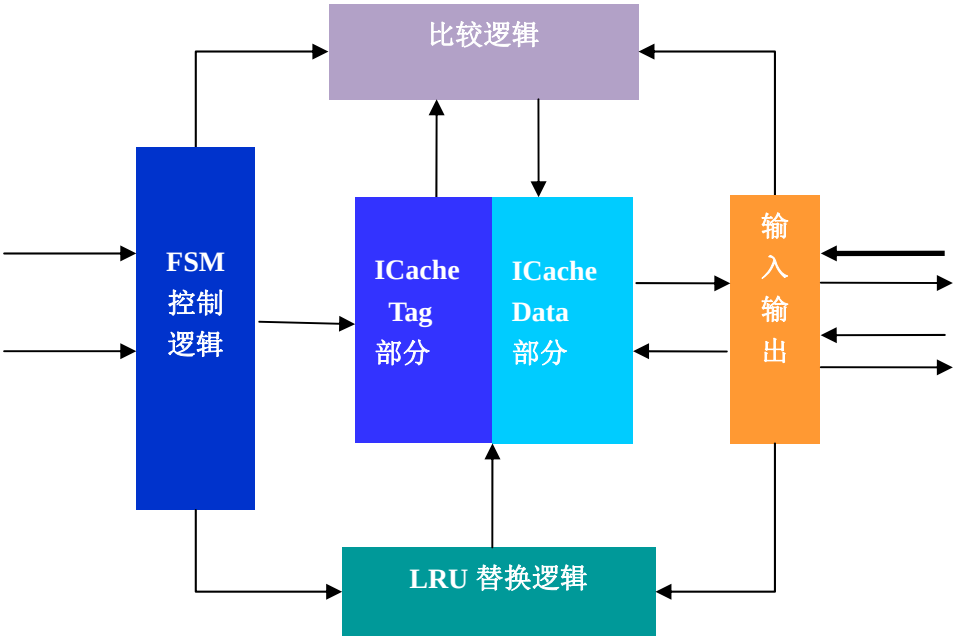
f) 用户界面可以判断工程进行到哪一步，并显示当前可以进行的操作。可以随时修改查看每一部分的代码和空间使用情况；

三、 本组设计的体系结构

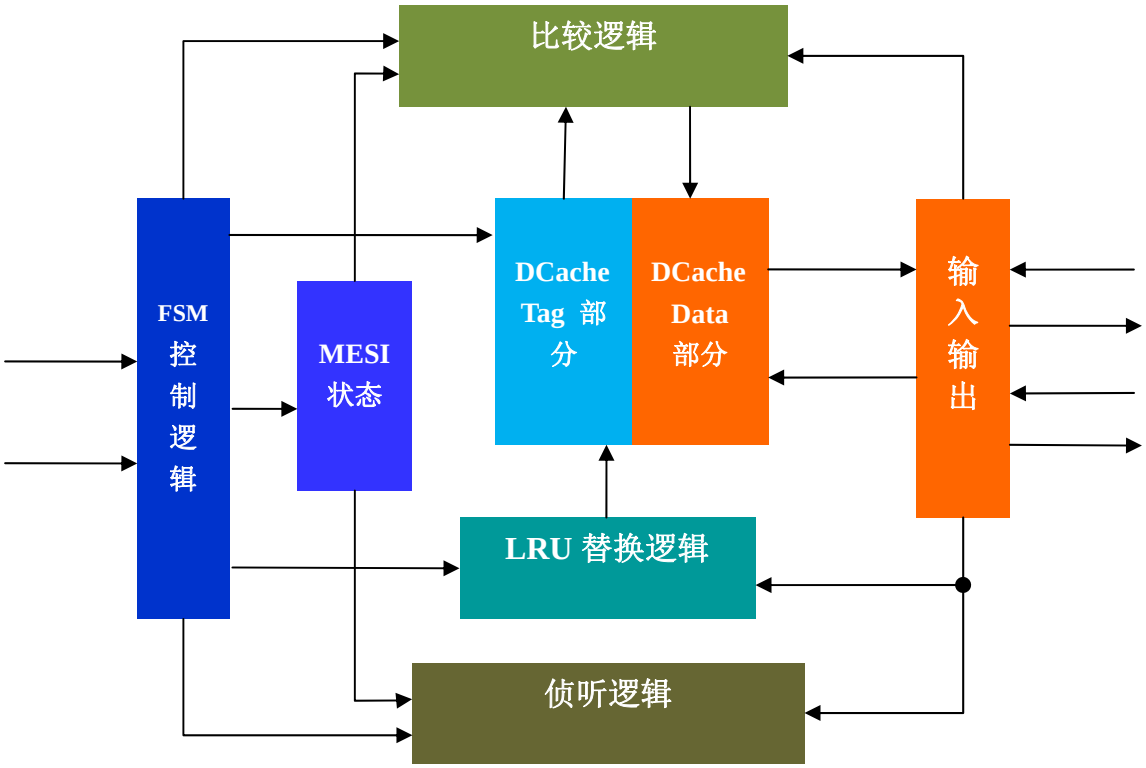
1. 硬件部分



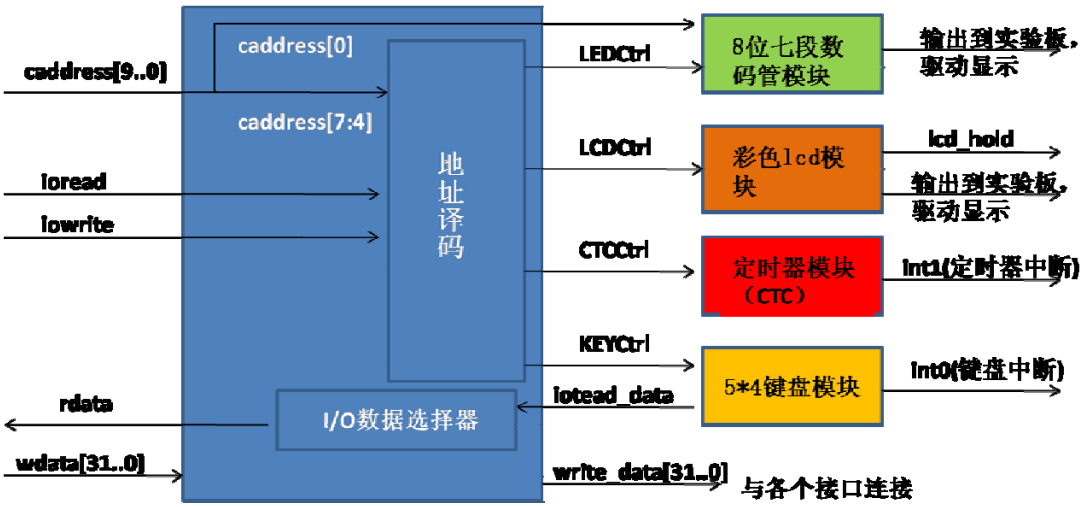
CPU 总体框架结构



ICache 结构框图

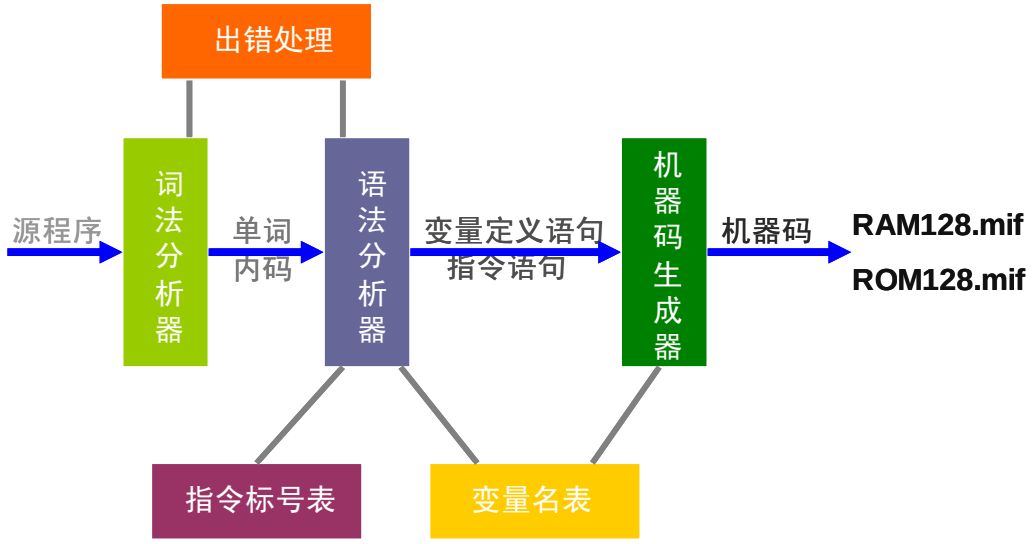


DCache 结构框图

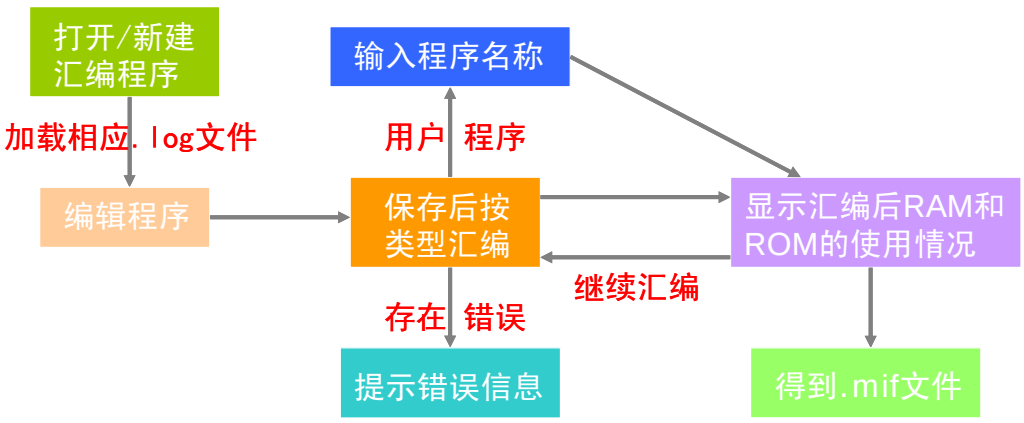


接口部件框图

2. 软件部分



汇编器内核

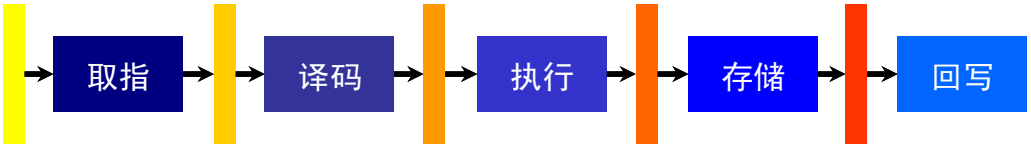


汇编器界面

四、 本组设计中各个部件的设计与特色概述（含关键代码）

1. 主运算控制核

a) 概述



运算核采用典型的五级流水：取指（IF）、译码（ID）、执行（EXE）、存储（MEM）和回写（WB）。

各流水级之间通过中间寄存器传递指令信息等，它们分别是：preIF 寄存器、IF/ID 寄存器、ID/EXE 寄存器、EXE/MEM 寄存器以及 MEM/WB 寄存器。

b) 取指单元

只要提供工作时钟，取指单元就会从 preIF 寄存器中取得 PC 值并从指令 Cache 中取指。在主控核中，preIF 的值由其自身设定，通常为 PC+1。

由于可能遇到中断、分支预测失败、无条件跳转、指令 Cache 未命中等多种情形，取指中获得的指令、计算的新 PC 都会在中途通过一系列逻辑判断，可能会送达 IF/ID 寄存器，也可能暂存在取指单元中以供以后调用。

代码展示了中间寄存器的存储形式：

```
Dff10 Dff_PC(.CLK(!CLK),.CLRN(!RST),.D(F_PC),.Q(PC));                  //prePC 寄存器
Dff10 Dff_NPC(.CLK(!CLK),.CLRN(!RST),.D(F_PC),.Q(IF_NPC));             //IF-ID PC 寄存器
Dff32 Dff_INST(.CLK(!CLK),.CLRN(!RST),.D(F_INST),.Q(IF_INST));         //IF-ID 指令寄存器
```

c) 译码单元

译码单元包括两个模块，译码模块和冲突检测模块。译码模块的主要输入信息是取址模块输出的 32 位指令码和一些突发事件的通知信号，输出信息为指令码所携带的所有信息。冲突检测模块接收寄存器的读出数据，执行模块的计算结果，内存和 IO 的读出数据以及各种突发事件的通知信号，输出数据未准备好时的暂停信号和经过判断后选择的执行模块需要的两个操作数。

译码模块通过指令码的操作码和功能码分辨各条指令，并且把所有指令执行时所需要的各项操作归类映射，最后以相应信号的高电平形式将信息传递到各个其他模块。

例如，J 类型指令的判断语句为：

```
jtype = !INST[31]&&!INST[29]&&INST[27];
//判断依据为指令码的 31 位 29 位和 27 位
```

给执行单元的“加”操作类型的判断语句为：

```
ALUOp[0] = ((RType&&INST[5]&&!INST[2]&&!INST[1])
        ||(IType&&!INST[28]&&!INST[27])||INST[31])?1'b1:1'b0;
//add
```

冲突检测模块负责解决数据相关的问题，当发生写后读相关时采用定向转发法为执行单元准备正确的计算数据。当前第一条指令或前第二条指令正要写的寄存器是当前指令两个操作数所在的寄存器时，就发生数据相关问题。可以判断相关的指

令分别执行到执行单元和访存单元，此时，只要获得这两个单元的输出值便可。

两个操作数存储在中间寄存器中，如下所示：

```
Dff32 Dff_SRCA(.CLK(!CLK),.CLRN(!RST),.D(NewSrcA),.Q(ID_SRCA));
Dff32 Dff_SRCB(.CLK(!CLK),.CLRN(!RST),.D(NewSrcB),.Q(ID_SRCB));
```

d) 执行单元

本单元的核心部件是 ALU，负责所有算术、逻辑和移位运算。

加减法运算器使用了系统库 LPM_ADD_SUB。系统库最大的好处在于以更少的资源消耗获得更健壮的功能。本例中，虽然加减运算使用 Verilog 数据流描述也仅需数行，但仍可从实际资源消耗中看到使用系统库的优势。

以下是两种写法的对比：

使用系统库——

```
SYS_ADD_SUB MyAddSub(
                .add_sub( Add_Sub ),
                .dataa( Input_A ),
                .datab( Input_B ),
                .result( Add_Sub_Result )
                );
wire[31:0] ALU_Res_Alg = ALU_Op[2] ? {31'b0, Add_Sub_Result[31]} : Add_Sub_Result;
```

数据流描述——

```
wire[31:0] Add_Result = ALU_Op[0] ? (Input_A+Input_B) : 32'h00000000;
wire[31:0] Sub_Mid_Result = Input_A - Input_B;
wire[31:0] Sub_Result = ALU_Op[1] ? Sub_Mid_Result : {31'b0,Sub_Mid_Result[31]};
wire[31:0] ALU_Res_Alg = ( ALU_Op[1]||ALU_Op[2] ) ? Sub_Result : Add_Result;
```

由于增加了指令，本单元需要额外传递 Thread 相关信号，其中 ThreadPC 将由此传出主核。中间寄存器的存储形式如下：

```
Dff32 Dff_RESULT(.CLK(!CLK),.CLRN(!RST),.D(RESULT),.Q(EXE_RESULT));
//EXE-MEM 计算结果寄存器
Dff32 Dff_ADDDATA(.CLK(!CLK),.CLRN(!RST),.D(WDATA),.Q(EXE_WDATA));
//EXE-MEM 数据寄存器
Dff10 Dff_CONTROL(.CLK(!CLK),.CLRN(!RST),.D(CONTROL),.Q(EXE_CONTROL));
//EXE-MEM 控制寄存器
dff Dff_ST(.CLK(!CLK),.CLRN(!RST),.PRN(1'b1),.D(ID_ST),.Q(EXE_ST));
//EXE-MEM Thread 开始信号寄存器
dff Dff_ET(.CLK(!CLK),.CLRN(!RST),.PRN(1'b1),.D(ID_ET),.Q(EXE_ET));
//EXE-MEM Thread 结束信号寄存器
Dff10Dff_THREAD_PC(.CLK(!CLK),.CLRN(!RST),.D(ID_THREAD_PC),.Q(EXE_THREAD_PC));
//EXE-MEM Thread PC 寄存器
```

e) 存储单元

存储阶段需要与数据 Cache 交互，Cache 未命中同样需要暂存当前状态而不向下一级流水传递。

Thread 尚有开启和结束信号需要由本单元传递下去，并由此传出主核。中间寄存器的存储形式如下：

```
Dff32 Dff_RESULT(.CLK(!CLK),.CLRN(!RST),.D(result),.Q(MEM_RESULT));
//MEM-WB 计算结果寄存器
Dff32 Dff_RDATA(.CLK(!CLK),.CLRN(!RST),.D(RDATA_MER),.Q(MEM_RDATA));
//MEM-WB 数据寄存器
Dff10 Dff_CONTROL(.CLK(!CLK),.CLRN(!RST),.D(control),.Q(MEM_CONTROL));
//MEM-WB 控制寄存器
dff Dff_ET(.CLK(!CLK),.CLRN(!RST),.PRN(1'b1),.D(et),.Q(MEM_ET));
//MEM-WB Thread 结束信号寄存器
```

f) 回写单元

回写单元包括回写模块和寄存器堆模块，回写模块主要是根据输入的控制信息从访存数据、执行单元结果和 IO 数据中过滤出最终需要回写的数据，寄存器堆模块顾名思义就是 32 个寄存器所在的模块。32 位寄存器申明语句如下：

```
wire [31:0] reg1;
Dffe32 Reg1(.CLK(CLK),.CLRN(CLRN),.ENA(CS[1]),.D(WRITEDATA),.Q(reg1));
```

寄存器堆为执行单元提供操作数，从寄存器堆读出的数据直接送译码模块经过判断和选择最终成为执行单元的操作数。寄存器堆同时也是寄存器回写的目的地。由于数据 cache 和指令 cache 的增加，当发生未命中需要延迟的情况时，需要保持输出数据不变。还有一个特殊情况是 JR 指令，当出现该指令时，作为输出数据的寄存器读出值是跳转地址，此时需要保持该地址两个指令周期，因为直接跳转指令需要暂停一个指令周期重新取址，具体的实现代码如下：

```
Dff32 Dff_IMME(.CLK(CLK),.CLRN(!RST),.D(READDATA1),.Q(olddata1));
assign READDATA1 = JR?olddata1:readdata1;
//readdata1 是当前指令读出的寄存器值，READDATA1 是最终的输出
```

2. 辅助运算核

a) 概述

辅助运算核的整体结构是与主核保持一致的，在不考虑功能冗余的情况下，除了取指单元，其余流水级单元它们均可复用。

辅助运算核的个数是可扩展的，为方便演示，本工程暂使用两个辅助核。

b) 与主核的区别

辅核与主核的取指单元是不能共用的，主要体现在两个方面：

1. 辅核的 prePC 寄存器存在于辅核任务分配模块，且其初值是由辅核任务分配模块从主核中获取而设置的，以后才由辅核自己 PC+1。
2. 由于辅核不像主核那样持续工作，当未给辅核分配任务时，辅核的取指需要停滞下来。

3. 辅核任务分配模块

a) 概述

此模块的作用在于自动寻找并指定空闲的辅核，等主核来分配任务时，就通过这个中介进行联系（如辅核的 prePC 就在此存储，主核辅核均可操作）。

b) 主要实现原理

在选择时，ACPU_PERMIT 和 ACPU_CHOOSE 寄存器起到关键作用，代码及简要解释如下：

```
wire[3:0] F_ACPU_CHOOSE;
assign F_ACPU_CHOOSE[0] = MCPU_START && !ACPU_PERMIT[0];
assign F_ACPU_CHOOSE[1] = MCPU_START && ACPU_PERMIT[0] && !ACPU_PERMIT[1];
assign F_ACPU_CHOOSE[2] = MCPU_START && ACPU_PERMIT[0] && ACPU_PERMIT[1] && !ACPU_PERMIT[2];
assign F_ACPU_CHOOSE[3] = MCPU_START && ACPU_PERMIT[0] && ACPU_PERMIT[1] && ACPU_PERMIT[2] && !ACPU_PERMIT[3];
//根据优先级找到空闲辅核
dff DFF_ACPUC1(.CLK(CLK),.CLRN(!RST),.PRN(1'b1),.D(F_ACPU_CHOOSE[0]),.Q(ACPU_CHOOSE[0]));
dff DFF_ACPUC2(.CLK(CLK),.CLRN(!RST),.PRN(1'b1),.D(F_ACPU_CHOOSE[1]),.Q(ACPU_CHOOSE[1]));
dff DFF_ACPUC3(.CLK(CLK),.CLRN(!RST),.PRN(1'b1),.D(F_ACPU_CHOOSE[2]),.Q(ACPU_CHOOSE[2]));
dff DFF_ACPUC4(.CLK(CLK),.CLRN(!RST),.PRN(1'b1),.D(F_ACPU_CHOOSE[3]),.Q(ACPU_CHOOSE[3]));
//锁存选择位
wire[3:0] ACLK;
assign ACLK[0] = ACPU_CHOOSE[0] || ACPU_END[0];
assign ACLK[1] = ACPU_CHOOSE[1] || ACPU_END[1];
assign ACLK[2] = ACPU_CHOOSE[2] || ACPU_END[2];
assign ACLK[3] = ACPU_CHOOSE[3] || ACPU_END[3];
//当选择好辅核或者辅核工作完成后提供触发
dff DFF_ACPUP1(.CLK(ACLK[0]),.CLRN(!RST),.PRN(1'b1),.D(!ACPU_PERMIT[0]),.Q(ACPU_PERMIT[0]));
dff DFF_ACPUP2(.CLK(ACLK[1]),.CLRN(!RST),.PRN(1'b1),.D(!ACPU_PERMIT[1]),.Q(ACPU_PERMIT[1]));
dff DFF_ACPUP3(.CLK(ACLK[2]),.CLRN(!RST),.PRN(1'b1),.D(!ACPU_PERMIT[2]),.Q(ACPU_PERMIT[2]));
dff DFF_ACPUP4(.CLK(ACLK[3]),.CLRN(!RST),.PRN(1'b1),.D(!ACPU_PERMIT[3]),.Q(ACPU_PERMIT[3]));
//根据上述触发，翻转辅核允许位
```

4. Cache

a) 指令 ICache

1. 基本结构

指令 Cache 行 数据域为 4 个字（16 字节、128 位），标签域为 7 位（最高位为有效位）。共有四行。

数据域：

Data			
	Data		
		Data	
			Data

127 96 95 64 63 32 31 0

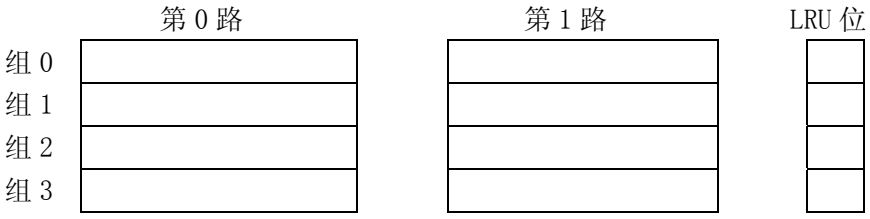
标签域：

Valid	Tag

6 5 0

2. 映射方式

采用 2 路组相联 的映射方式。结构如下：



3. 替换算法

算法思想：伪 LRU 替换算法，两路的四组 cache 行分别共用 1 个 LRU 位，当第 0 行的 cache 命中时置 0，第 1 路 cache 行命中时置 1。Cache 替换时，根据行检索相应的 LRU 位，为 0 时，替换第一路中的相应行；为 1 时，替换第 0 路的相应行。这种算法也适应于多路组相联 cache 的替换策略，简单的用 1 位表示 2 行的最近使用情况，比 LRU 算法中的计数实现起来要简单的多。

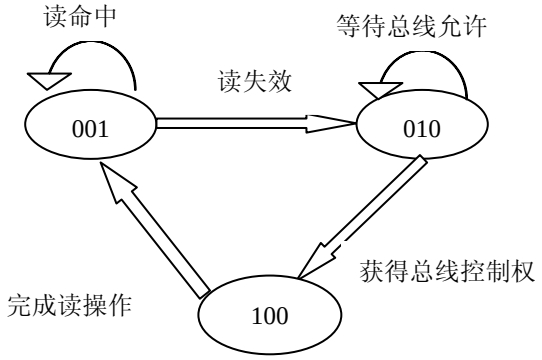
实现代码：

```
//LRU
wire RLRU;
ICacheLRU  ICache_LRU(.CLK(CLK),.CLRN(CLRN),.WE(ENL),.INDEX(ADDR[3:2]),
.D(NLRU),.Q(RLRU));

wire EN=(STATE[2])?1'b1:1'b0;
wire EN0=EN&&RLRU;
wire EN1=EN&&(!RLRU);
wire ENL=(STATE[0]&&HIT||STATE[2])?1'b1:1'b0;

wire NLRU=(STATE[0]&&HIT)?!HIT0:!RLRU;
```

4. 一致性问题

由于指令 Cache 只是从指令 ROM 中提取指令数据，CPU 再从中读指令，基本全是读操作，在标签域设置 Valid 位，保证数据的有效，一致性容易实现。

5. 读状态机



6. ICacheBus

在多核系统中，针对多个指令 Cache 对指令 ROM 读操作的竞争，设计 ICacheBus 作为 Cache 和 ROM 之间的桥梁。ICacheBus 负责接收各个 Cache 的读内存操作（MRead），并依据优先级对其进行仲裁，同一时刻，优先级高的 Cache 先获得 ROM 访问权。ICacheBus 在进行仲裁时的原则是：先到先处理，同时到按优先级处理。默认优先级为： Cache0>Cache1>Cache2…

b) 数据 ICache

1. 基本结构

数据 Cache 行 数据域为 4 个字（16 字节、128 位），标签域为 6 位，MESI 位 2 位，共有四行。

数据域：

Data			
	Data		
		Data	
			Data

127 96 95 64 63 32 31 0

标签域：

Tag

5 0

MESI 位

MESI

1 0

2. 映射方式

同指令 ICache

3. 替换算法

同指令 ICache

4. 写策略

写回法+写分配的策略。

I 写回法：信息只被写到 Cache 行中，只有当 Cache 中的脏行被替换出去时，信息才被写回到存储器中。

II 写分配：在发生写缺失时，存储器的行被读到 Cache 中，然后执行写命中时的操作。

这两种方法，通常搭配使用。相对于写直达法，写回法需要的带宽较小，对其他存储层次和存储总线的使用较少，因此，在总线资源和带宽紧张的多处

理器系统中，多采用写回法，因此，本设计采用写回+写分配的策略

5. 一致性保持

这里的一致性，包含两个方面：

（1）单核内数据 Cache 与数据 RAM 的一致性

（2）多核间数据 Cache 数据的一致性

考虑到数据 Cache 写策略的特点和多处理器的情况，本设计中采用：基于总线侦听的 MESI 一致性策略。

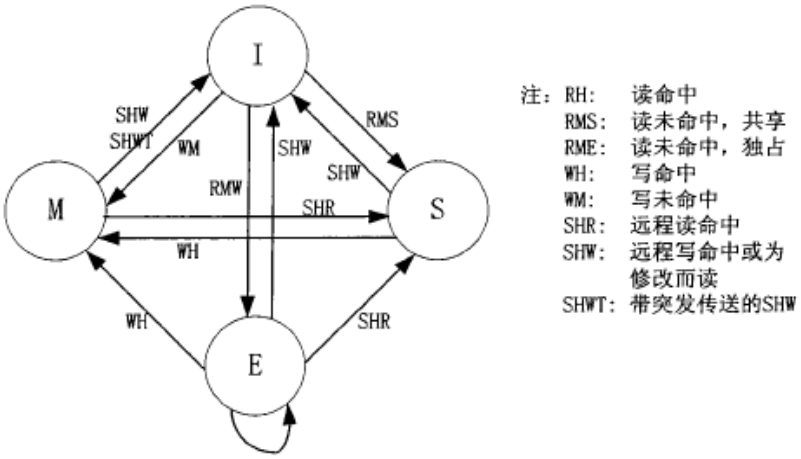


图 3-4 MESI 协议状态转换图

MESI 协议如图所示，共有四个状态，定义如下:

（1）修改状态(M):此 Cache 行已经被处理器修改过，数据值与存储器中的不同，而且仅在本 Cache 有其数据副本;

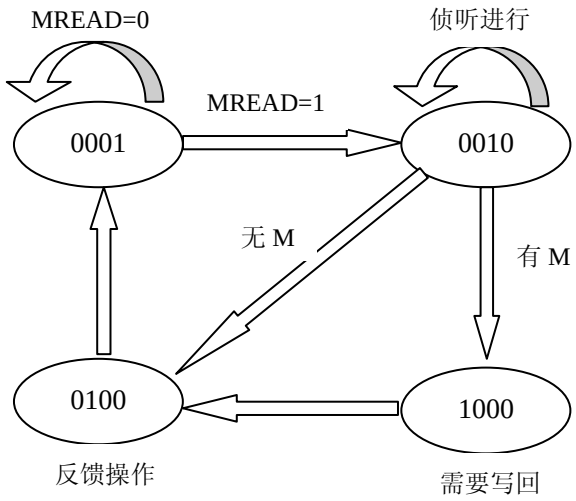
（2）专用状态(E):此 Cache 行与存储器中的数据值相同，并且仅有本 Cache 有其数据副本;

（3）共享状态(S):此 Cache 行与存储器中的数据相同，同时有两个或两个以上的 Cache 有其数据副本。

（4）无效状态(1):此 Cache 行没有包含有效数据。

该协议的实现，需要每个处理器监听共享总线上的一切操作，控制总线的处理器为主模块，其余为监听模块，各自根据广播的侦听地址和命中情况，查询自己 Cache 中是否有对应的数据，若有，则对其 MESI 状态做相应的修改。在本设计中，数据 Cache 总线负责启动侦听，各处理器负责发送侦听请求或响应侦听。

（1）总线侦听状态机如下图所示：

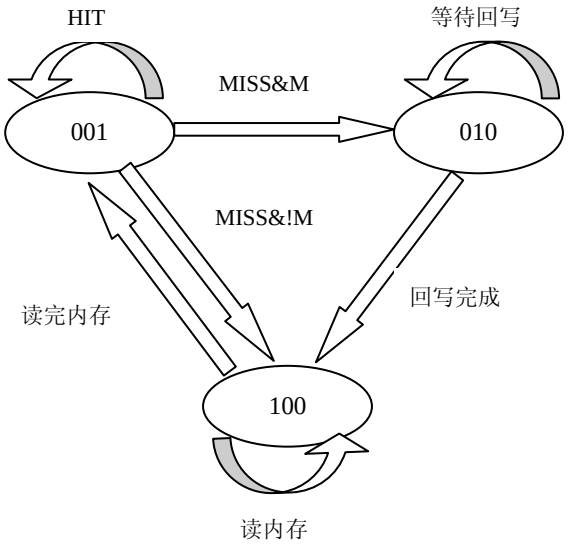


（2）各处理器侦听状态机 略

实现代码：

```
//修改 MESI
wire RH=RHIT;
wire RMS=CREAD&&STATE[2]&&BSETS;
wire RME=CREAD&&STATE[2]&&BSETE;
wire WH=WHIT;
wire WM=CWRITE&&!WHIT;
wire SHR=BS[1]&&BSETS;
wire SHW=BS[1]&&BSETI;
wire[3:0] MESI0,MESI1;
wire ENMESI0=STATE[2]&&(TH?TH0:OLRU)||BS[1]&&TH0;
wire ENMESI1=STATE[2]&&(TH?TH1:!OLRU)||BS[1]&&TH1;
wire SIM=STATE[2]&&!BS[1]&&!RDRTN;

DCacheMESI  DCacheMESI0(
                .CLK(CLK),
                .RST(RST),
                .EN(ENMESI0),
                .INDEX(ADDR[3:2]),
                .STATE(SIM),
                .RH(RH),
                .RMS(RMS),
                .RME(RME),
                .WH(WH),
                .WM(WM),
                .SHR(SHR),
                .SHW(SHW),
                .MESI(MESI0)
                );

DCacheMESI  DCacheMESI1(
                .CLK(CLK),
                .RST(RST),
                .EN(ENMESI1),
                .INDEX(ADDR[3:2]),
                .STATE(SIM),
                .RH(RH),
                .RMS(RMS),
                .RME(RME),
                .WH(WH),
                .WM(WM),
                .SHR(SHR),
                .SHW(SHW),
                .MESI(MESI1)
                );
```

6. 读写状态机



7. DCacheBus

这里的 DCacheBus 有两方面的作用：

1、用于解决上述一致性问题，作为侦听进行的主要模块。

2、在多核系统中，针对多个数据 Cache 对数据 RAM 读写操作的竞争，设计 DCacheBus 作为 Cache 和 RAM 之间的桥梁。

DCacheBus 负责接收各个 Cache 的读写内存操作（MRead、MWrite），并依据优先级对其进行仲裁，同一时刻，优先级高的 Cache 先获得 RAM 访问权。DCacheBus 在进行仲裁时的原则是：先到先处理，同时到按优先级处理。默认优先级为：写操作>读操作， Cache0>Cache1>Cache2…

5. 新增指令

a) 浮点加指令

指令格式：

31-26	25-21	20-16	15-11	10-6	5-0
010001	fs	ft	fd	s,d	001111

汇编格式：addf fd,fs,ft

指令类型：R-类型

功能描述：(fd)<-(fs)+(ft)

完成同精度浮点数之间的加运算（s:32 位；d:64 位）

汇编举例：addf $4,$2,$3

b) 浮点减指令

指令格式：

31-26	25-21	20-16	15-11	10-6	5-0
010001	fs	ft	fd	s,d	011111

汇编格式：subf fd,fs,ft

指令类型：R-类型
功能描述：（fd）<-(fs)-(ft)
完成同精度浮点数之间的减运算（s:32 位；d:64 位）
汇编举例：addf $4,$2,$3

c) 开启线程指令
指令格式：

31-26	25-23	22-10	9-0
010000	010	全 0	address

汇编格式：st address
指令类型：NEW-类型
功能描述：(PC)<-address
主核选择空闲辅核，给该核的 PC 赋初值 address，运算不涉及浮点运算
汇编举例：st 100h

d) 开启浮点线程指令
指令格式：

31-26	25-23	22-10	9-0
010000	011	全 0	address

汇编格式：stf address
指令类型：NEW-类型
功能描述：(PC)<-address
主核选择空闲辅核，给该核的 PC 赋初值 address，所有运算为浮点运算
汇编举例：stf 130h

e) 结束线程指令
指令格式：

31-26	25-23	22-0
010000	000	全 0

汇编格式：et
指令类型：NEW-类型
功能描述：辅核停止工作并向主核发送线程结束信号
汇编举例：et

f) 占用接口指令
指令格式：

31-26	25-24	23-21	20-17	16-0
010000	10	xxx	4 位接口号	全零

汇编格式：oi interface-number
指令类型：NEW-类型
功能描述：某个核占用某个接口的开始标志
汇编举例：oi ff00h

g) 释放接口指令

指令格式：

31-26	25-24	23-21	20-17	16-0
010000	11	xxx	4 位接口号	全零

汇编格式：ri interface-number

指令类型：NEW-类型

功能描述：某个核使用某个接口完毕后的释放接口标志

汇编举例：ri ff00h

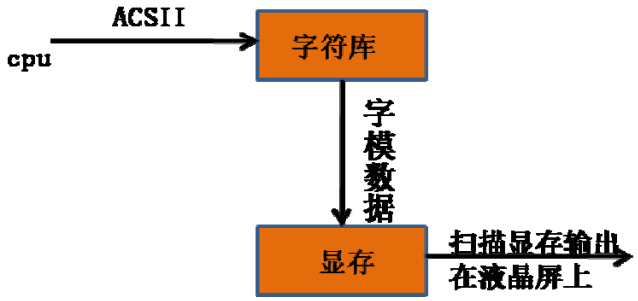
6. 接口部分

a) 彩色 LED 控制器

1. 概述

提供了大容量的显存和字符库；用户只需提供内存起始地址，取出数据（ASCII 值）后通过查字符表将用户想要显示的数据存入到显存中；为解决与 cpu 速度不匹配问题，控制器中会给出 lcd_hold 信号，信号保持到数据全部存入显存中。

2. 显示原理



3. CPU->字符库

```
module Character(sclk,reset,cs,iow,cpudata,set,set_addr,data,data_addr,
                 data_w,up,lcd_hold);
input   sclk;                   //系统时钟
input   reset;                  //复位信号
input   cs;                     //片选信号
input   iow;                    //写信号
input   [31:0] cpudata;         //cpu 数据
input   [127:0]set;             //根据 asc2 值从字符库取出的字模数据

output [6:0]   set_addr;        //字符库地址线
output [7:0]   data;            //要存入显存的数据
output [14:0] data_addr;        //显存的地址
output data_w;                  //显存写信号
output up;                      //滚屏信号
output lcd_hold;                //hold 信号
```

```
……

always @(posedge sclk)
begin
    if(iow==1 && cs==1)
        begin
        start=1;
        asc2[7:0]=cpudata[7:0];
        end

    if(reset==1)
        begin
        /*为变量置初值*/
        end

    if(start==1)
        begin
/*根据 div 值不同来进行不同的操作。
1->若 acs 不为回车，就开始将数据存入显存，并将 lcd_hold 置 1，否则直接将 pos 置成下一行开始;
18->表示已将一个完整的字符数据存储到显存中，将 lcd_hold 置 0,并且判断目前是不是存到的边界;
其他->如下，将从 set 中取出的 128 位的数据按照显示格式存入显存*/
            else
                begin
                    case (div)
                    17: data=set[7:0];
                    ……
                    2:  data=set[127:120];
                default:data=set[7:0];
                endcase
                    data_addr=pos+64*(div-1);     //显存的具体地址
                data_w=1;
                end
            div=div+1;
        end
end
endmodule
```

4. 扫描显存

根据液晶显示屏的说明书，模拟出扫描波形，具体见说明书中的波形图。

b) 5*4 键盘控制器

1. 概述

根据键盘电路图设计有效的扫描方式；设计中考虑到了电路抖动问题；以中断方式工作，每次按键都会产生 INT0 中断；键盘的中断信号将一直保持到中断处理器发出响应信号。

2. 程序实现

```
module key(sclk,reset,cs,ior,done,clk,col,line,ioread_data,int0);
.....
input   ior;          //读接口信号
input   done;         //cpu 中断完成信号
input   clk;          //工作时钟
input   [3:0]   col; //列读入信号
output [4:0]   line;//行扫描信号
output [15:0] ioread_data; //扫描数据
output int0;          //中断信号
.....

always @(posedge sclk)
begin
     //判断何时进行复位工作，将键值送到数据总线
end

always@(posedge clk)
begin
/*line=5'b00000->根据 col 输入信号判断是否有键被按下。
若有键被按下，则改变 line 的值，开始对键盘进行逐行扫描，根据 col 值得到被按键的准确位置；
否则 line 值保持不变，继续对键盘进行监控。
设计中使用 presstime 来去除抖动，只有当连续 7 次扫描到某一个键被按下时，我们才判断键盘已经稳
定，取此时的按键信息。*/
end

always @(posedge press or posedge done)
begin
     //中断信号的控制部分，只有当 done 为 1 时，才可以发中断信号
end
always @(keyvalue)
begin
     //键值编码部分，以 0~16 表示 5*4 上的各个按键
end
endmodule
```

c） 8 位七段数码管控制器

1. 概述

利用指令寄存器，实现了多种显示方式。

2. 程序实现

```
module led32(sclk,reset, cs, iow, clk, D, address,DIG, Y);
.....
```

```
input address;      //地址线的最低位
input[31:0] D;      //cpu 写向接口的数据
output[7:0] DIG;    //8 位数码管的位选择引脚
output[7:0] Y;      //7 段数码管的显示引脚

reg[31:0] C_r;      //指令寄存器
reg[31:0] D_r;      //数据寄存器
reg[2:0] scan_cnt;
reg[3:0] data;

always@(posedge sclk)
begin
        /*根据 address 判断从 cpu 传来的数据是指令还是数据。
        0->将总线上的数据存入指令寄存器;
        1->根据数码管的工作方式将总线上的数据存入数据寄存器的相应位置。*/
end

always@(posedge clk or posedge reset) //  获得扫描时钟
……

always@(scan_cnt)
begin
        case ( scan_cnt )
            3'b000 : begin
                DIG_r <= 8'b0000_0001;   //点亮第一个数码管
                data <= D_r[3:0];        //将数据后四位显示在选中的数码管上
                end
                ……
         endcase
    end
……
//译码，根据 data 的值设置对应的 Y
Endmodule
```

d) 定时器控制器

1. 概述

以中断方式工作，当计时到达用户设定的初值时，就向 CPU 发起 INT1 中断。

2. 程序实现

```
module ctc(sclk,reset,cs,iow,int_done,iowrite_data,int1);
……
input   int_done;              //中断完成信号，从 cpu 接入
input   [31:0]iowrite_data;    //cpu 写向接口的数据
```

```
output int1;                    //中断 int1 信号

reg[31:0] ini_data;             //存储计数器初值
reg[31:0] count;                //计数器

always @(posedge sclk)
begin
    ……
   if(cs==1 && iow==1)
        begin
            case(iowrite_data)
            //为 ini_data 赋值；
            endcase
            count=0;
            start=1;
        end

    if(start==1)
begin
/*count 为加计数器，当 count 的值到达 ini_data 时，则发出中断信号，count
被重置为 0，重新开始计数。*/
end
endmodule
```

e) 接口控制模块

1. 概述

分为 io 和 io_data_ctrl 两个部分，前者完成地址译码，数据转换和输入数据选择器，后者实现 I/O 数据选择器。

7. BIOS

a) 概述

对部分硬件的简单自检；BIOS 在做好自检和初始化之后，就会根据规定的位置引导用户程序。

b) BIOS 程序清单

```
CODE SEG
ORG_CODE 03CAH
START:      ADDI $SP,$ZERO,0FFFFH
            ADDI $T0,$ZERO,0
            ADDI $T1,$ZERO,0FFFFH
            ADDI $T2,$ZERO,55AAH
            ADDI $T3,$ZERO,0AA55H
            ADDI $S0,$ZERO,0E1H
```

```
CHKRAM:     LW   $T8,0($T0)
            SW   $T2,0($T0)
            LW   $T4,0($T0)
            BNE  $T4,$T2,ERROR
            SW   $T3,0($T0)
            LW   $T4,0($T0)
            BNE  $T3,$T4,ERROR
            SW   $T8,0($T0)
            BEQ  $T0,$T1,CHKLED
            ADDI $T0,$ZERO,4
            J    CHKRAM

CHKLED:     ADDI $T0,$ZERO,0
            ADDI $T1,$ZERO,0FFFFH
DIPS:       ADDI $T2,$ZERO,0FFFFH
            SW   $T0,0FF01H($0)
            ADDI $T3,$ZERO,1
LOP:        SUB  $T2,$T2,$T3
            BNE  $T2,$ZERO,LOP
            ADDI $T0,$T0,1111H
            BNE  $T0,$T1,DIPS
            J    用户程序地址

ERROR:      SW   $S0,0FF01H($ZERO)

JP:         J    JP
END START
CODE ENDS
```

8. 汇编器

a) 内核程序

1. 初始化部分

初始化所有需要初始化的变量、结构体，并调用文件读写部分初始化 RAM 和 ROM 的内容。

```
/***初始化成员变量***/
bool Assembler::InitVar() {
    n_line = 1;
    n_errorNum = 0;
    n_warnNum = 0;
    n_currentRule = 1;
    n_dataAddr = -1;
    n_comAddr = -1;
    while(!m_stack.empty())  m_stack.pop();   //初始化语法栈
    m_stack.push(-1);
```

```
    m_stack.push(N_PRO);
    m_com.clear();
    m_wb.clear();
    m_var.clear();
    m_label.clear();
    memset(m_ram,0,sizeof(m_ram));              //初始化RAM内容
    memset(m_used,false,sizeof(m_used));
    Instr *ins;
    if(n_type == 1)   {          //初始化BIOS和INT的跳转指令
        ...
    }
    if(n_type >= 4) {
        ...
    }
    else {
        ...
    }
    ReadRAM("RAM_BIOS.log");
    ReadROM("ROM_BIOS.log");
    n_orgcode = m_com.size();
    return true;
}
```

2. 词法分析部分

该模块每次调用 GetChar()从文件中读取一个字符，并进行处理，最终得到一个完整的 token，通过分析和检查，返回其合法的 token 值，或者非法的-1，所有的英文字符都被转化为小写，所有的数字都被转化为二进制的格式，第一个字符为符号，最后一个字符为 b。

```
/***获得下一个token对应的终结符***/
int Assembler::NextToken() {
    /**处理简单字符的token**/
    ...
    /**处理关键字**/
    map<string,int>::iterator iter = m_keyWords.find(m_cur);
    if(iter != m_keyWords.end()) {                    //是关键字，返回相应的终结符号
        return iter->second;
    }
    int n = m_cur.length();
    string s = m_cur;
    if(notvalid)   goto err;
    /**处理寄存器**/
    if(m_cur[0] == '(' && m_cur[n-1] == ')')     m_cur = m_cur.substr(1,n-2) , n = -1;
    if(m_reg.find(m_cur) != m_reg.end() && (top == T_RS || top == T_RT
        || top == T_RD || top == T_BRS)){
```

```
        return top;
    }
    /**处理标志符和数字**/
    ch = m_cur[0];
    if(ch >= 'a' && ch <= 'z')                          //标志符
        return T_IDNAME;
    if((ch >= '0' && ch <= '9') || ch == '-') {
        if(ch != '-')    m_cur = '+' + m_cur;
        int len = m_cur.length(), i;
        ch = m_cur[len-1];
        if(ch == 'h') {
            ...
        }
        else if(ch == 'd') {                            //十进制
            ...
        }
        else if(ch >= '0' && ch<= '9') {                //十进制
            ...
        }
        if(ch == 'b') {                                 //二进制
            for(i = 1; m_cur[i] == '0'; i++);           //去除前导0
            if(i > 1)  m_cur = m_cur[0] + m_cur.substr(i,m_cur.length()-i);
            len = m_cur.length();
            for(i = 1; i < len-1; i++ ) {
                if(m_cur[i] != '0' && m_cur[i] != '1') {
                    Error(ERROR_BIN);
                    return -1;
                }
            }
            if(top == T_SHAMT || top == T_ADDR || top == T_NUM || top ==
T_IMMEDIATE) {
                return top;
            }
            if(top == N_ADDR)    return T_ADDR;
        }
        else {                                          //否则为词法错误
            goto err;
        }
    }
err:                                                    //根据栈顶的元素报错
    m_cur = s;
    Error(ERROR_LEXICAL);
    return -1;
}
```

3. 语法分析部分

语法分析采用了 LL（1）文法，每次读取一个 token 值，根据当前下推栈顶的符号判断是进行推导还是匹配，如果匹配成功，同时进行翻译，如果语法错误导致无法推导下去，则结束程序。注意翻译仅当一直无错的情况下才会进行，因此如果存在了错误，之后的语义错误无法发现。分析成功调用 Succeed()进行写文件，并判断是否最终成功，如果失败调用 Fail()函数，并终止。

```
/**LL1分析程序**/
int top;
token = NextToken();
while( true ) {
    top = m_stack.top();
    //根据栈顶符号进行分析
    ...
    else if(top < VT_NUM) {              //栈顶为终结符，则匹配
        if(token != top) {
            if(token == T_ENDL) {
                CheckLine();       //如果此行读完，可能是指令误翻译，或者是输入中多输入了回车，分开讨论
                continue;
            }
            Error(ERROR_LEXICAL);
            if(top != T_ENDL) m_stack.pop();
        }
        else {                           //匹配成功，弹出栈首元素，并翻译，这里由于T_ENDL不会识别错误，因此是按行处理错误
            m_stack.pop();
            if(!n_errorNum) {      //如果一直没有错误，则翻译，否则不翻译
                if(!Translate())     //如果是致命错误，则直接跳出
                    break;
            }
        }
        token = NextToken();
    }
    else {                               //栈顶为非终结符，则推导
        n_lastRule = n_currentRule;     //在翻译地址的时候用到
        if((n_currentRule = n_forcast[top-50][token]) != 0) {  //LL(1)表中有对应项，推导可以进行
            Deduce();
        }
        else {
            if(token == T_ENDL) {
                n_currentRule = n_lastRule;
                token = NextToken();
```

```
                continue;
            }
            if(token == T_NOP) {
                continue;
            }
            Error(ERROR_SEMANTIC);
            if(IsFatal()) {          //如果是致命错误，则直接退出
                Error(ERROR_FATAL);
                break;
            }
            else {                   //否则，按照默认的推导方式推导
                Deduce();
            }
        }
    }
}
```

4. 机器码生成部分

该模块主要是将汇编程序中的数据和指令翻译程机器码，其中数据放在 RAM128.mif 文件，指令放在 ROM128.mif 文件，为了检查方便，同时生成了 32 位的 ROM32.mif 文件，每条指令附加注释。翻译采取的策略是每识别一个正确的单词，调用翻译程序进行翻译，翻译结合了当前使用的语法规则、前一条使用的语法规则以及当前的 token 值来进行的。如果当前 token 是指令码，则插入一条指令，否则，填充当前指令。如果指令中遇到地址为标号，则查找标号表，如果存在，把地址写入指令码，否则把指令插入回填表；如果插入标号，则插入标号的同时查找回填表，把回填表中的指令填充完整，并从回填表中删除该指令。当语法程序分析完毕后还要检查回填表是否为空，如果不为空说明有未定义的标号，报错。

```
/***翻译成机器码***/
bool Assembler::Translate() {
    if(token == T_ENDL)  return true;
    switch(n_currentRule) {                    //根据当前规则
    /**数据部分**/
    case 11:
        if(token == T_IDNAME) {
            if(n_dataAddr > RAM_SIZE) {
                Error(ERROR_RAMOVERFLOW);
                return true;
            }
            if(m_var.find(m_cur) == m_var.end())
                m_var.insert(pri(m_cur,n_dataAddr+1));
            elseWarn(WARN_REVAR);
            return true;
        }
```

```
        else if(token == T_NUM)    return InsertData();
        else return true;
    case …
    default:
        return true;                                        //其他语法规则不用翻译
    }
}
```

5. 错误处理部分

可处理常见的词法、语法、语义错误。词法和语法错误是在词法分析中发现的，而语义错误是在翻译过程中发现的。遇到错误后，调用错误总控程序 Error(int TypeOfError)，错误总控程序根据错误类型进行处理，并打印错误信息，同时调整下推栈和输入流，这样语法分析程序就可以继续进行下去。

```
/***根据当前下推栈的内容来决定是否出错，并进行处理***/
void Assembler::CheckLine() {
    int n = m_syntastic[n_currentRule].size() - 1;
    int top = m_stack.top();
    if(n_currentRule == 35 || n_currentRule == 36)
        n_currentRule = n_lastRule;
    if(m_syntastic[n_currentRule][n] == top) {        //如果当前是一条新的指令，则这是多余的换行符，忽略
        token = NextToken();
        return;
    }
    //如果推导到一半遇到换行，则认为是指令错误，根据当前的规则进行错误的判断
    while(m_stack.top() != m_syntastic[n_currentRule][0])    m_stack.pop();
    m_stack.pop();
    if(m_stack.top() == T_ENDL)    m_stack.pop();
    token = NextToken();
    n_line--;                                          //错是上一行引起的
    switch(n_currentRule) {
        …
    }
    n_line++;
}
```

6. 读写文件部分

包括读.log 文件、写.log 文件、写.mif 文件。其中.log 文件是用来记录 BIOS、INT0、INT1 和用户程序的指令和数据使用情况的文件，方便动态的增加和删除程序，具体的读写文件是根据用户当前写的程序类型而定，类型有 5 种：1--BIOS 2--INT0 3--INT1 4—初始化用户程序 5--添加用户程序。.mif 文件是用来提供给 cache 读入并继续进行处理运行的。

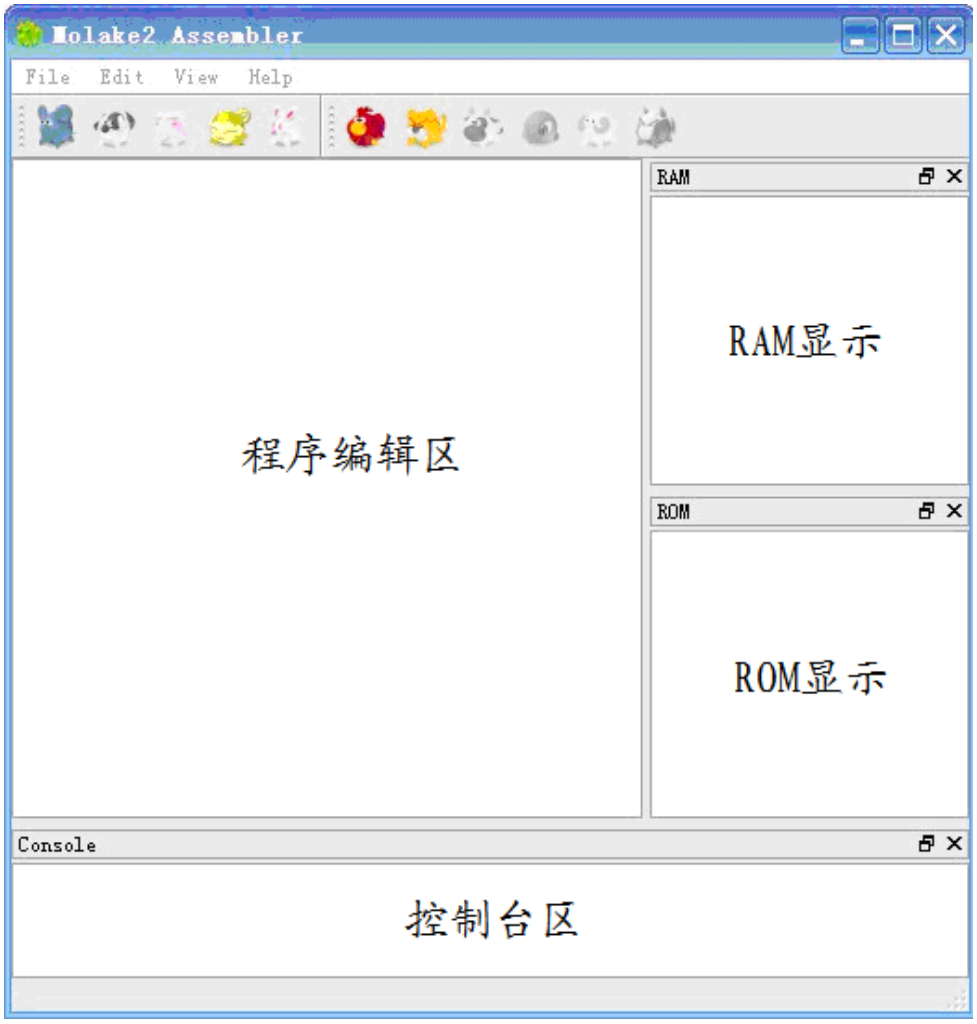
```
/***写RAM，输出数据，32位字宽，其他的类似***/
m_console << "writing RAM32.mif...\n";
```

```
m_dfout.open((m_path+"RAM32.mif").c_str());
if(m_dfout.bad()) return false;
m_dfout << "DEPTH = " << RAM_SIZE << ";\n";
m_dfout << "WIDTH = 32;\n\n";
m_dfout << "ADDRESS_RADIX = HEX;\n";
m_dfout << "DATA_RADIX = HEX;\n\n";
m_dfout << "CONTENT\n";
m_dfout << "BEGIN\n";
m_dfout << hex;                                    //以16进制输出
for(i = 0; i < RAM_SIZE; i++) {
    if(m_used[i]) {                                //地址被使用
        m_dfout << '\t' << setw(3) << setfill('0') << i << " : "
            << setw(4) << setfill('0') << m_ram[i][0]
            << setw(4) << setfill('0') << m_ram[i][1] << ";\n";
    }
    else {                                         //未被使用
        j = i++;
        while(i < ROM_SIZE && !m_used[i])     i++;
        i--;
        if(j == i) {
            m_dfout << "\t" << setw(3) << setfill('0') << j << " : 00000000;\n";
        }
        else {
            m_dfout << "\t[" << setw(3) << setfill('0') << j << "..";
            m_dfout << setw(3) << setfill('0') << i << "] : 00000000;\n";
        }
    }
}
m_dfout << "END;\n" << dec;
m_dfout.close();

/***写BIOS的信息文件，其他的类似***/
bool Assembler::WriteBIOS() {
    ofstream logout;
    int i , j;

    /**写RAM的信息文件**/
    m_console << "writing RAM_BIOS.log...\n";
    logout.open((m_path+"RAM_BIOS.log").c_str());
    if(logout.bad())    return false;
    for(i = 0; i < RAM_SIZE; i++)
        if(m_used[i])
            logout << i << ' ' << m_ram[i][0] << ' ' << m_ram[i][1] << endl;
    logout.close();
```

```
    /**写ROM的信息文件**/
    m_console << "writing ROM_BIOS.log...\n";
    logout.open((m_path+"ROM_BIOS.log").c_str());
    if(logout.bad())    return false;
    m_com[3]->content += " |***BIOS sevice***|";
    for(i = 0 , j = m_com.size(); i < j; i++) {
        logout << m_com[i]->addr<< ' '
            << m_com[i]->code       << ' '
            << m_com[i]->key        << ' '
            << m_com[i]->type       << ' '
            << m_com[i]->content << endl;
    }
    logout.close();
    return true;
}
```

b) 界面部分

界面使用开源的 Qt4.6.0 开发，整体包括四个部分：菜单栏区、程序编辑区、RAM 显示区、ROM 显示区和控制台部分，如下图所示：



1. 程序编辑区

程序编辑区使用 QTextEdit 的 Widget，通过 Hightlighter 类实现了关键字、数字、字符串和注释的高亮显示。程序部分支持中文注释，但不推荐使用中文。

2. RAM 和 ROM 显示区

这两个区使用 QtextBrowser 类，以只读的方式读取写在所建工程文件夹下的 RAM32.mif 和 ROM32.mif 文件已经使用的部分用蓝色表示，未使用的部分用灰色表示。当打开文件时，如果存在这两个文件，会自动的读取并显示。

3. 控制台区

这个区使用 QtextBrowser 类，以只读的方式读取写在 Hcasm.exe 所在文件夹下的 console.txt 文件，如果编译过程出现 error，以红色的字体显示，如果出现 warning，以黄色的字体显示。

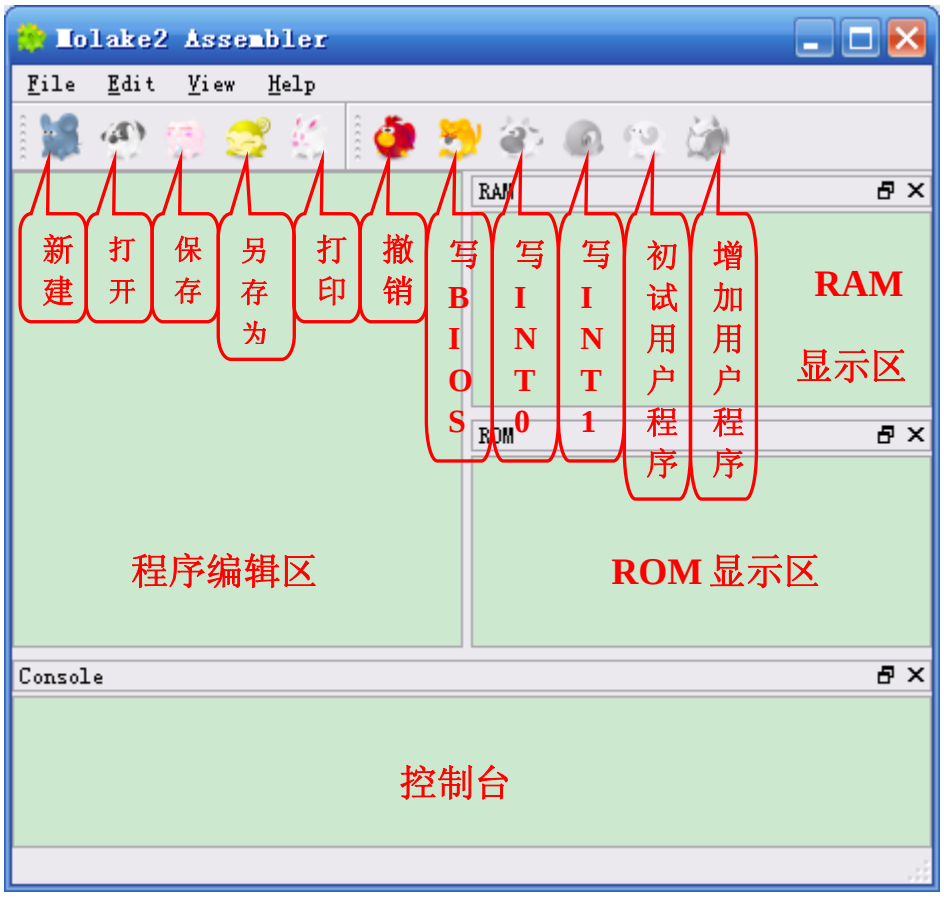
4. 菜单栏区

该区实现了简单的文本编辑功能和主要的汇编功能，并能自动将各个程序链接起来。

五、 本组设计的 MiniSys 汇编程序使用手册

1. 汇编器简介

汇编器界面使用开源的 Qt4.6.0 开发，包括五个部分：菜单栏，程序编辑区，RAM 显示区，ROM 显示区和控制台。如下图所示：



- 程序编辑区：编辑汇编程序。
- RAM/ROM 显示区：显示当前工程中 RAM 和 ROM 的使用情况。按照地址顺序，用蓝色字体显示已经使用部分的地址编号、十六进制指令码、汇编指令以及注释；用灰色字体显示未使用部分的起始地址和结束地址，指令码为全零。
- 控制台：显示当前程序的汇编结果。如果编译过程出现错误，以红色字体显示，如果出现警告，以黄色字体显示。
- 菜单栏：包括新建、打开、保存、另存为、打印、撤销等基本文件操作以及写 BIOS、写 INT0、写 INT1、初始化用户程序和增加用户程序等汇编器专用操作。

2. 工程建立步骤

一个完整的汇编程序工程主要由 BIOS 程序，中断处理程序 INT0、INT1，和用户程序组成。向工程中添加程序需要按照一定的优先级顺序，优先级最高的是 BIOS 程序，其次是 INT0 和 INT1，然后是初始化用户程序，最后是增加用户程序，当修改并写入了某一优先级程序后，较低优先级程序就会被排除出工程，但是程序文件依然在工程文件夹内，需要重新写入。相同优先级内的程序的修改不会互相影响，当然，前提是地址没有冲突。

例如，当写好 BIOS 和 INT0 之后修改了 BIOS 并重新写入时，INT0 便被排除出工程了，这时就需要重新写入 INT0。当写好 BIOS、INT0 和 INT1 后要修改 INT0 就只要直接修改和写入就好了，但是如果已经写到用户程序这一步要修改 INT0 或 INT1，那么写入后，用户程序就被排除了。

一般的编辑顺序如下：

a) 编辑并写入 BIOS 程序，使用“写 BIOS”按钮写入；
b) 编辑并写入 INT0 中断处理程序，使用“写 INT0”按钮写入；
c) 编辑并写入 INT1 中断处理程序，使用“写 INT1”按钮写入；
d) 编辑并写入用户程序，使用“初始化用户程序”按钮写入；
e) 编辑并写入其他用户程序，使用“增加用户程序”按钮写入。

3. 地址分配

0000 CPU 开始运行的第一条指令地址，汇编器在写入 BIOS 时自动在该地址填入无条件跳转指令，跳转到 BIOS 程序的首地址，CPU 转而执行 BIOS 程序。

0001 INT0 中断入口地址，汇编器写入 INT0 时自动在该地址填入无条件跳转指令，跳转到 INT0 处理程序的首地址，进行中断处理。

0002 INT0 返回地址，中断处理程序最后一条指令跳转到这里，该地址保存了跳转到 26 号寄存器(已保存了中断返回地址)内地址的指令。

0003 INT1 中断入口地址，汇编器写入 INT1 时自动在该地址填入无条件跳转指令，跳转到 INT1 处理程序的首地址，进行中断处理。

0004 INT1 返回地址，中断处理程序最后一条指令跳转到这里，该地址保存了跳转到 27 号寄存器(已保存了中断返回地址)内地址的指令。

0010 INT0 处理程序的首地址。

0020 INT1 处理程序的首地址。

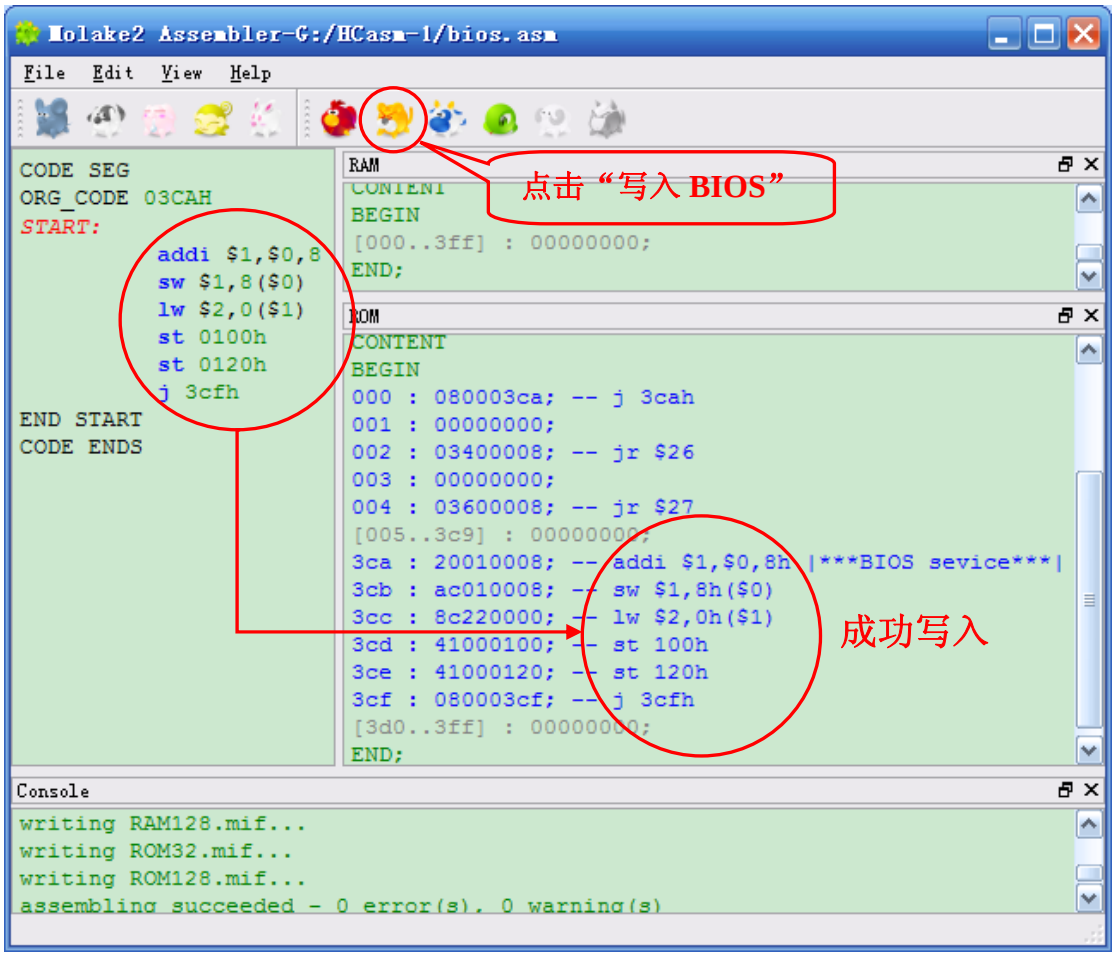
0100 用户程序的首地址。

03CA BIOS 程序的首地址。

4. 实例说明

以多核测试程序为例，BIOS 程序首先自检，再执行用户程序，本例中省去自检过程，主核程序的主要内容为启动两个线程分配给两个空闲的核，INT0 为键盘中断，INT1 为定时器中断。

首先编辑 BIOS 程序并写入 BIOS（按“写 BIOS”按钮），效果如下图：



st(start thread)指令是开启新线程指令，所带的参数是 ROM 中的指令起始地址。主核在开始两个新线程后可以继续执行其他指令或跳转到用户程序，在本例中主核执行的指令是跳转到本条指令，执行效果是无限循环。ROM 显示区中显示了 ROM 的使用情况，零号地址自动保存跳转到 BIOS 的起始地址(03ca)的指令。

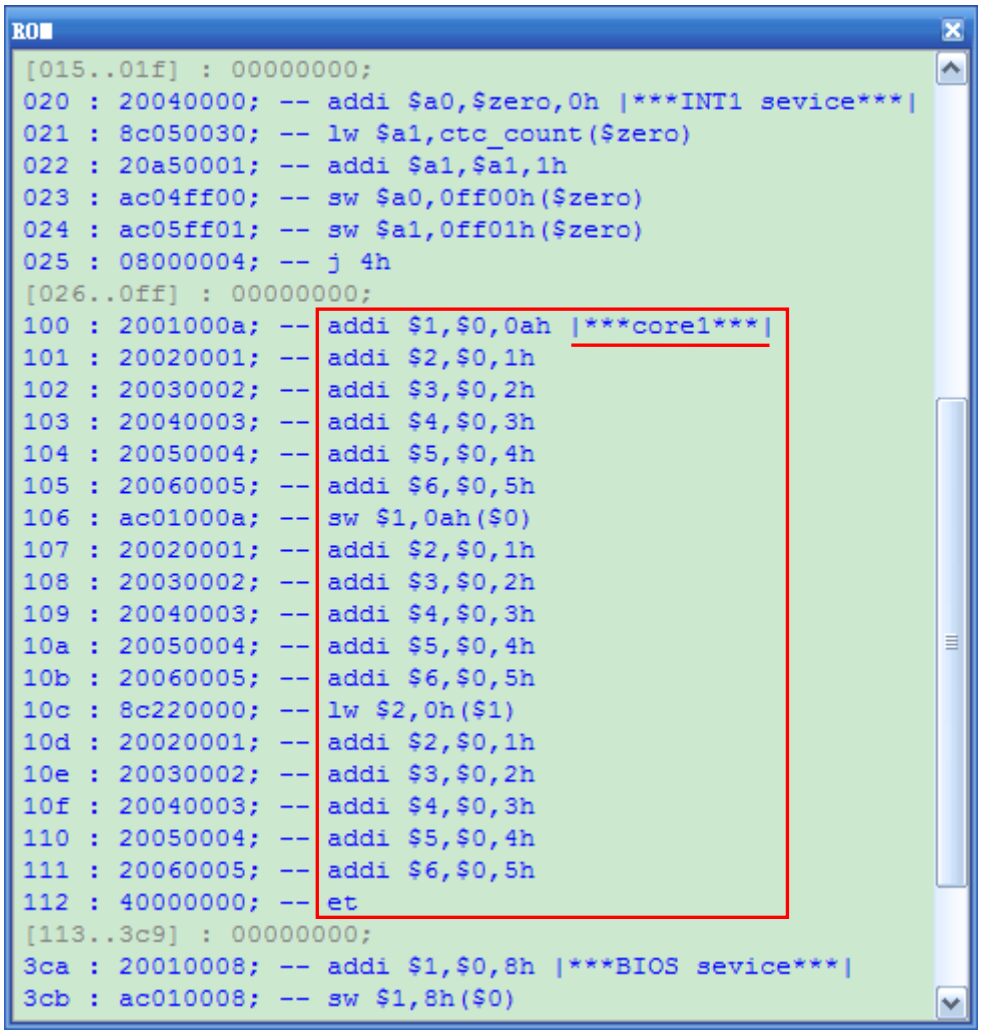
然后编辑并写入中断处理程序 INT0 和 INT1（按“写 INT0”和“写 INT1”按钮），ROM 显示区的效果如下图：

```
CONTENT
BEGIN
000 : 080003ca; -- j 3cah
001 : 08000010; -- j 10h
002 : 03400008; -- jr $26
003 : 08000020; -- j 20h
004 : 03600008; -- jr $27
[005..00f] : 00000000;
010 : 20040001; -- addi $a0,$zero,1h |***INT0 sevice***|
011 : 8c05ff10; -- lw $a1,0ff10h($zero)
012 : ac04ff00; -- sw $a0,0ff00h($zero)
013 : ac05ff01; -- sw $a1,0ff01h($zero)
014 : 08000002; -- j 2h
[015..01f] : 00000000;
020 : 20040000; -- addi $a0,$zero,0h |***INT1 sevice***|
021 : 8c050030; -- lw $a1,ctc_count($zero)
022 : 20a50001; -- addi $a1,$a1,1h
023 : ac04ff00; -- sw $a0,0ff00h($zero)
024 : ac05ff01; -- sw $a1,0ff01h($zero)
025 : 08000004; -- j 4h
[026..3c9] : 00000000;
3ca : 20010008; -- addi $1,$0,8h |***BIOS sevice***|
3cb : ac010008; -- sw $1,8h($0)
3cc : 8c220000; -- lw $2,0h($1)
3cd : 41000100; -- st 100h
3ce : 41000120; -- st 120h
3cf : 080003cf; -- j 3cfh
[3d0..3ff] : 00000000;
END;
```

ROM 中的分配情况一目了然，0000 到 0004 都自动填写了正确的跳转指令，从 010 地址开始是 INT0 处理程序，从 020 地址开始是 INT1 处理程序，从 03ca 地址开始是 BIOS 程序。

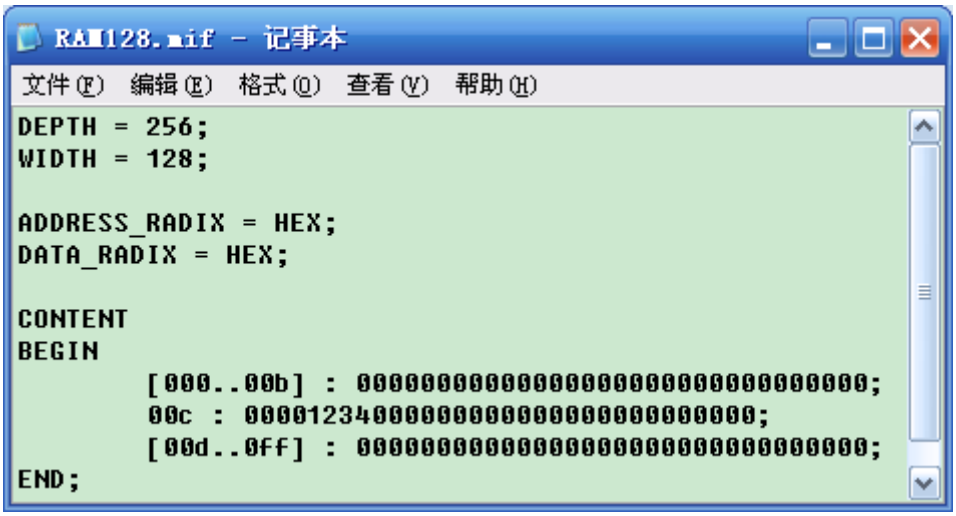
接下来初始化用户程序，编辑第一个线程执行的指令，点击初始化用户程序后弹出如下对话框，用来指定该段用户程序的名字，便于查看和管理。在这里取名为 core1：

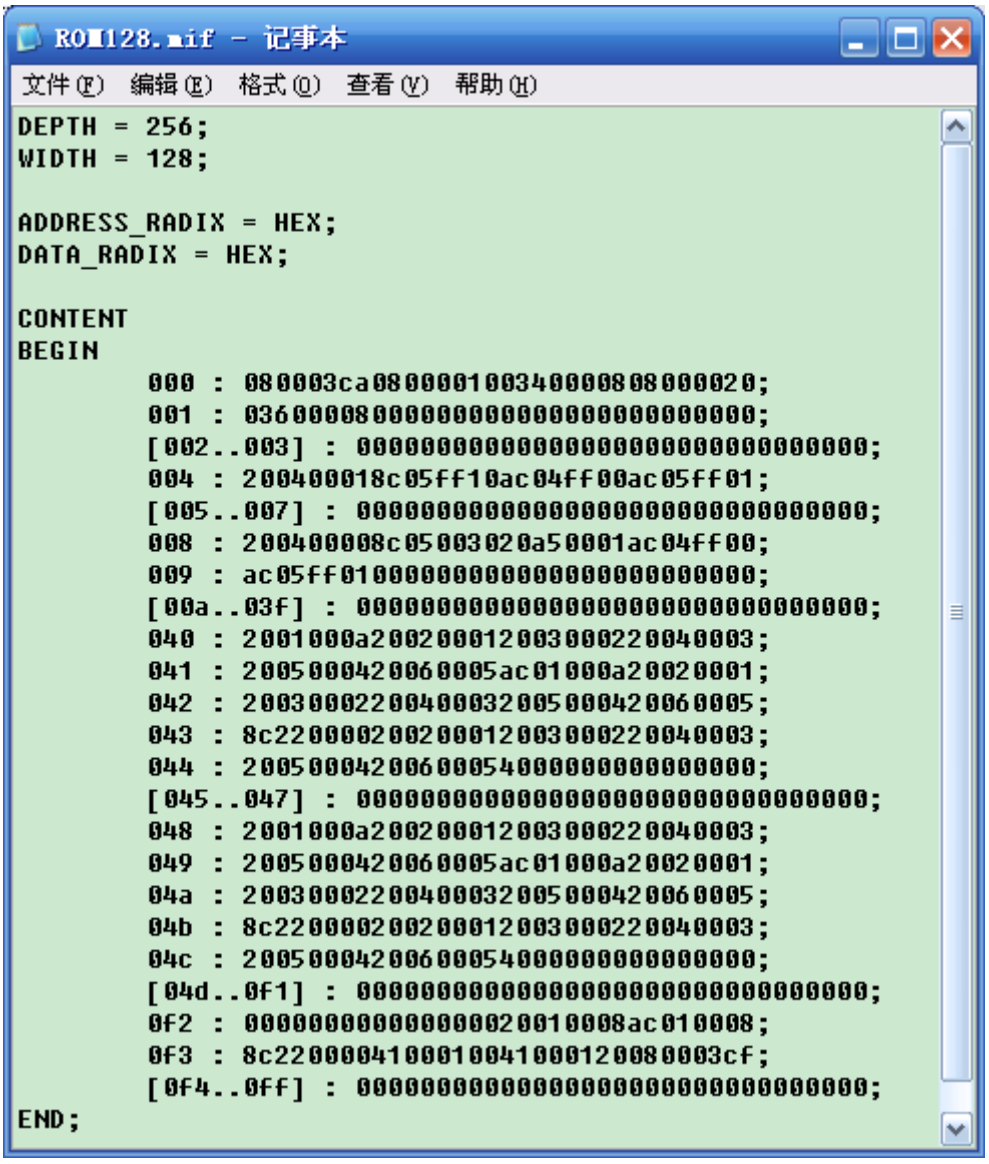
点击 ok 后便写入了初始化用户程序，此时 ROM 的效果如下图：



ROM 显示区在相应的地址显示了用户程序 core1。类似的，当输入第二段用户程序时，点击增加用户程序按钮，在弹出的对话框中输入程序名即可，图略。

这样，完整的工程就建立好了，这时候到保存源程序文件的文件夹下找到汇编器生成的 ROM128.mif 和 RAM128.mif 文件即可用于 CPU 的测试了。实例工程生成的文件如下图：





5. 编程注意事项

- 所有程序不能使用已经占用的 ROM 空间，不能使用 0000h~0004h 的地址空间；
- 编程时请注意 RAM 和 ROM 空间利用的情况，以便更好的选择所用的 RAM 和 ROM 空间；
- 编程注意语法的限制，不能实现语法之外的功能，比如子程序或者多次定义地址的起始位置等；
- 支持中文注释，但不推荐使用；
- 每个程序如果要单独保存推荐每次都新建为一个文件，防止被覆盖；
- 一个工程的程序最好保存在同一个文件夹下，方便查看和操作；
- 所有的打开、保存、另存为的路径中不可以包含中文字符，否则无法正确打开；

六、 本组设计主要测试结果（.vwf）

可演示部分（下载到实验板上的具体运行情况）：

- 键盘、LED 及定时器的联合测试

1. 测试程序

INT0： （处理来自键盘的中断）

将键值显示在左边四个数码管上

```
code seg
org_code 0010h
start:
addi   $a0,$zero,1        ;将 a0 赋值为 1
lw     $a1,0ff10h($zero);读取键值，将数据存入 a1
sw     $a0,0ff00h($zero);设定 LED 控制字，为左边四数码管显示
sw     $a1,0ff01h($zero);将数据传给 LED
j      0002h              ;INT0 中断返回
end start
code ends
```

INT1:（处理来自定时器的中断）

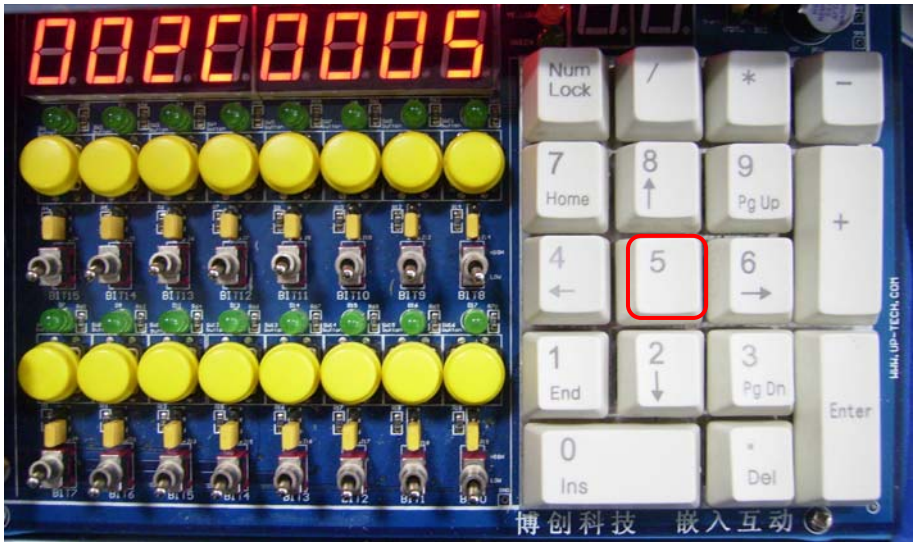
对定时器中断进行计数，将结果显示在右边四个数码管上

```
code seg
org_code 020h
start:
addi   $a0,$zero,0        ;将 a0 赋值为 0
addi   $a1,$a3,0          ;将 a3 的值赋给 a1
addi   $a3,$a3,1          ;将 a3 中的数值加一
sw     $a0,0ff00h($zero);设定 LED 控制字，为右边四数码管显示
sw     $a1,0ff01h($zero);将数据传给 LED
j      0004h              ;INT1 中断返回
end start
code ends
```

用户程序：

```
code seg
org_code 03cah
start:
 addi $a3,$zero,1        ;初始化 a3 的值为 1，a3 为记录 INT1 的个数
addi   $a1,$zero,2        ;将 a1 赋值为 2
sw     $a1,0ff30h($zero);设定 CTC 控制字，设定定时器为 2 秒定时器
loop: j     loop          ;循环，等待中断
end start
code ends
```

2. 效果图如下



左边四位显示的是定时器计数，右边四位显示的是键盘的值，5 表示我们按下了键盘上的 5

较之第一幅图，我们可以看到计数器在工作，而现在我们按下的键是 del

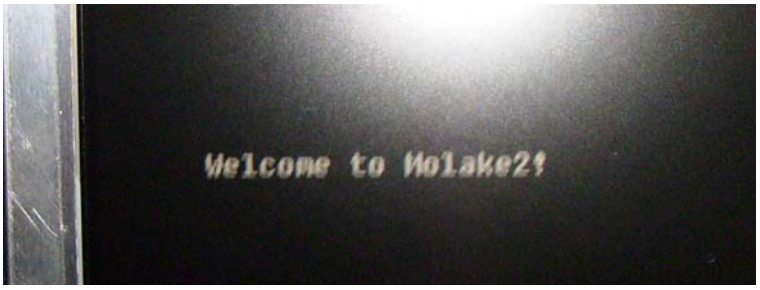
◆ 彩屏显示测试

1. 程序，利用 cpu 写的欢迎界面

```
code seg
org_code 0030h
start:    addi $a1,$zero,0dh          ; 回车
          sw $a1,0ff20h($zero)
          addi $a1,$zero,0dh
          sw $a1,0ff20h($zero)
          ……
          addi $a1,$zero,1h           ; 空格
          sw $a1,0ff20h($zero)
          ……
```

```
        addi $a1,$zero,6bh          ；显示语句
        sw $a1,0ff20h($zero)
        addi $a1,$zero,65h
        sw $a1,0ff20h($zero)
        ……
loop: j loop
end start
code ends
```

2. 效果，在屏幕中央显示了 Welcome to Molake2!



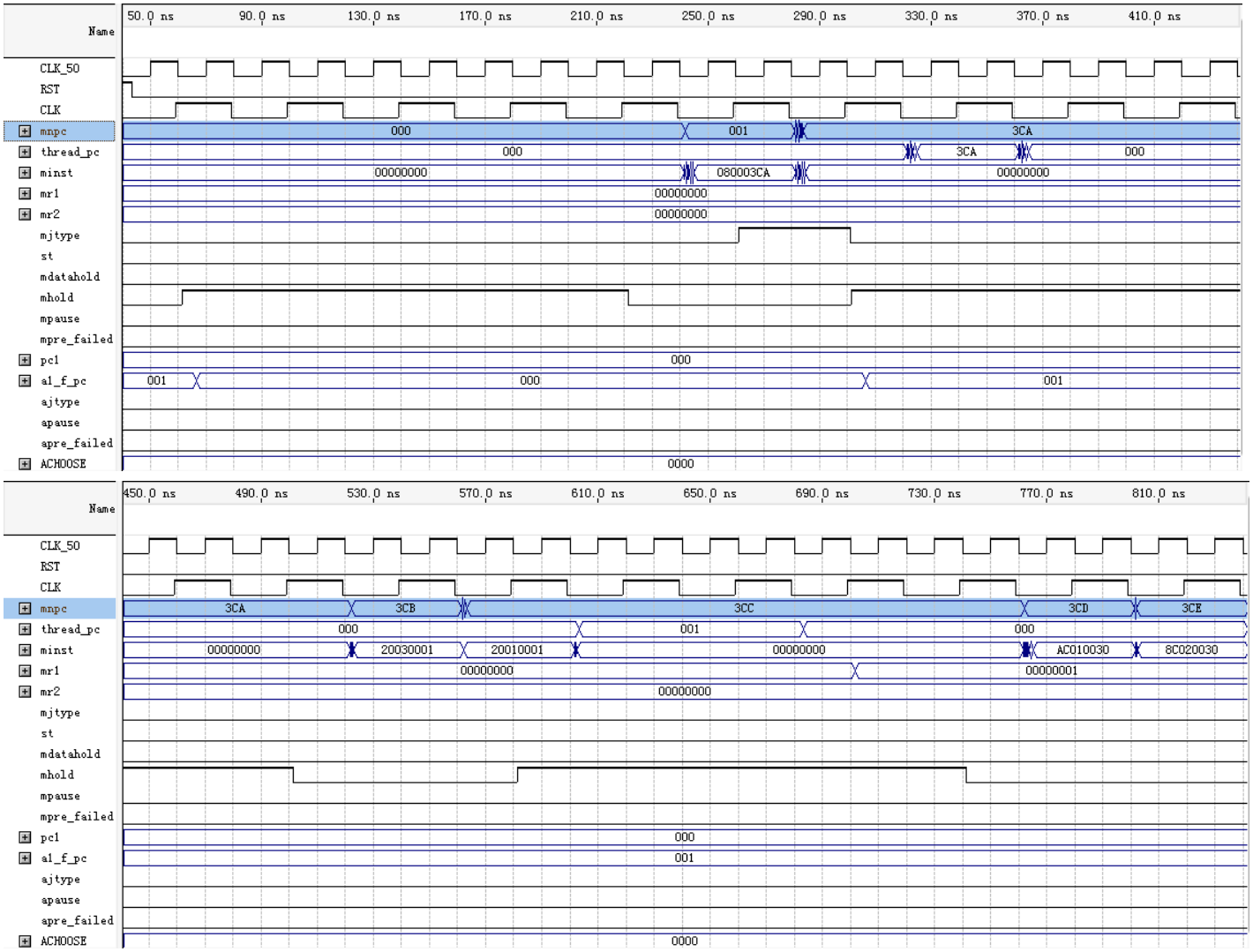
波形图部分：

- 完整测试代码：

```
000 : 080003ca;  -- j 3cah
001 : 08000010;  -- j 10h
002 : 03400008;  -- jr $26
003 : 08000020;  -- j 20h
004 : 03600008;  -- jr $27
[005..00f] : 00000000;
010 : 20090001;  -- addi $t1,$zero,1h |***INT0 sevice***|
011 : 08000002;  -- j 2h
[012..01f] : 00000000;
020 : 200a0002;  -- addi $t2,$zero,2h |***INT1 sevice***|
021 : 08000004;  -- j 4h
[022..0ff] : 00000000;
100 : 2001000a;  -- addi $1,$0,0ah |***core1***|
101 : 20020001;  -- addi $2,$0,1h
102 : 20030002;  -- addi $3,$0,2h
103 : 20040003;  -- addi $4,$0,3h
104 : 20050004;  -- addi $5,$0,4h
105 : 20060005;  -- addi $6,$0,5h
106 : ac01000a;  -- sw $1,0ah($0)
107 : 20030002;  -- addi $3,$0,2h
108 : 20040003;  -- addi $4,$0,3h
109 : 20050004;  -- addi $5,$0,4h
10a : 20060005;  -- addi $6,$0,5h
10b : 20070006;  -- addi $7,$0,6h
```

```
10c : 8c02000a;  -- lw $2,0ah($0)
10d : 20030002;  -- addi $3,$0,2h
10e : 20040003;  -- addi $4,$0,3h
10f : 20050004;  -- addi $5,$0,4h
110 : 20060005;  -- addi $6,$0,5h
111 : 20070006;  -- addi $7,$0,6h
112 : 40000000;  -- et
[113..1ff] : 00000000;
200 : 2001000b;  -- addi $1,$0,0bh |***core2***|
201 : 20020001;  -- addi $2,$0,1h
202 : 20030002;  -- addi $3,$0,2h
203 : 20040003;  -- addi $4,$0,3h
204 : 20050004;  -- addi $5,$0,4h
205 : 20060005;  -- addi $6,$0,5h
206 : ac01000a;  -- sw $1,0ah($0)
207 : 20030002;  -- addi $3,$0,2h
208 : 20040003;  -- addi $4,$0,3h
209 : 20050004;  -- addi $5,$0,4h
20a : 20060005;  -- addi $6,$0,5h
20b : 20070006;  -- addi $7,$0,6h
20c : 8c02000a;  -- lw $2,0ah($0)
20d : 20030002;  -- addi $3,$0,2h
20e : 20040003;  -- addi $4,$0,3h
20f : 20050004;  -- addi $5,$0,4h
210 : 20060005;  -- addi $6,$0,5h
211 : 20070006;  -- addi $7,$0,6h
212 : 40000000;  -- et
[213..3c9] : 00000000;
3ca : 20030001;  -- addi $3,$zero,1h |***BIOS sevice***|
3cb : 20010001;  -- addi $1,$zero,1h
3cc : ac010030;  -- sw $1,30h($zero)
3cd : 8c020030;  -- lw $2,30h($zero)
3ce : 14220002;  -- bne $1,$2,loop
3cf : 41000100;  -- st 100h
3d0 : 41000200;  -- st 200h
3d1 : 080003d1;  -- j loop
[3d2..3ff] : 00000000;
```

- 波形图演示



000 : 080003ca;

--j 3cah

开机后，从 PC=000 开始取指

指令 Cache 未命中，mhold=1

四个周期后取得 000 处指令

080003CA

本语句为跳转

3ca : 20030001;

--addi $3,$zero,1h

跳至 PC=3CA 后

指令 Cache 再次未命中

四个周期后取得 3CA 处指令

20030001

3cb : 20010001;

--addi $1,$zero,1h

3cc : ac010030;

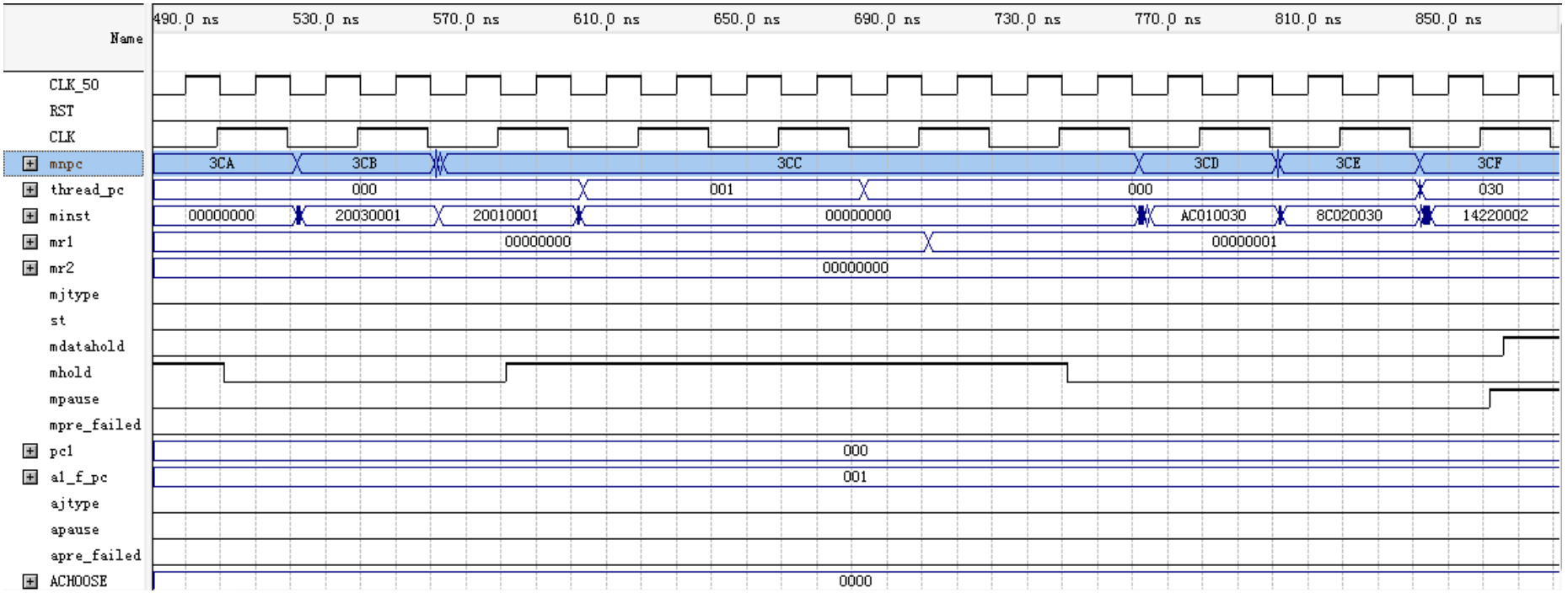
--sw $1,30h($zero)

运行至 PC=3CC 处

指令 Cache 未命中

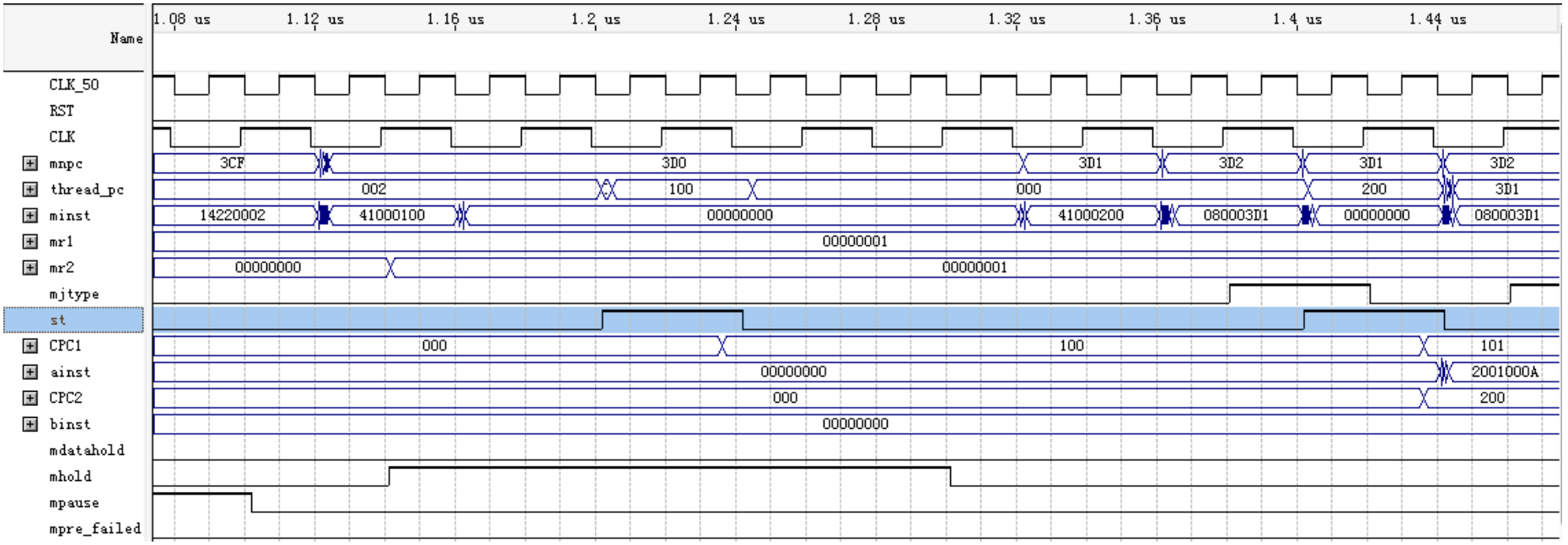
3cd : 8c020030;

--lw $2,30h($zero)



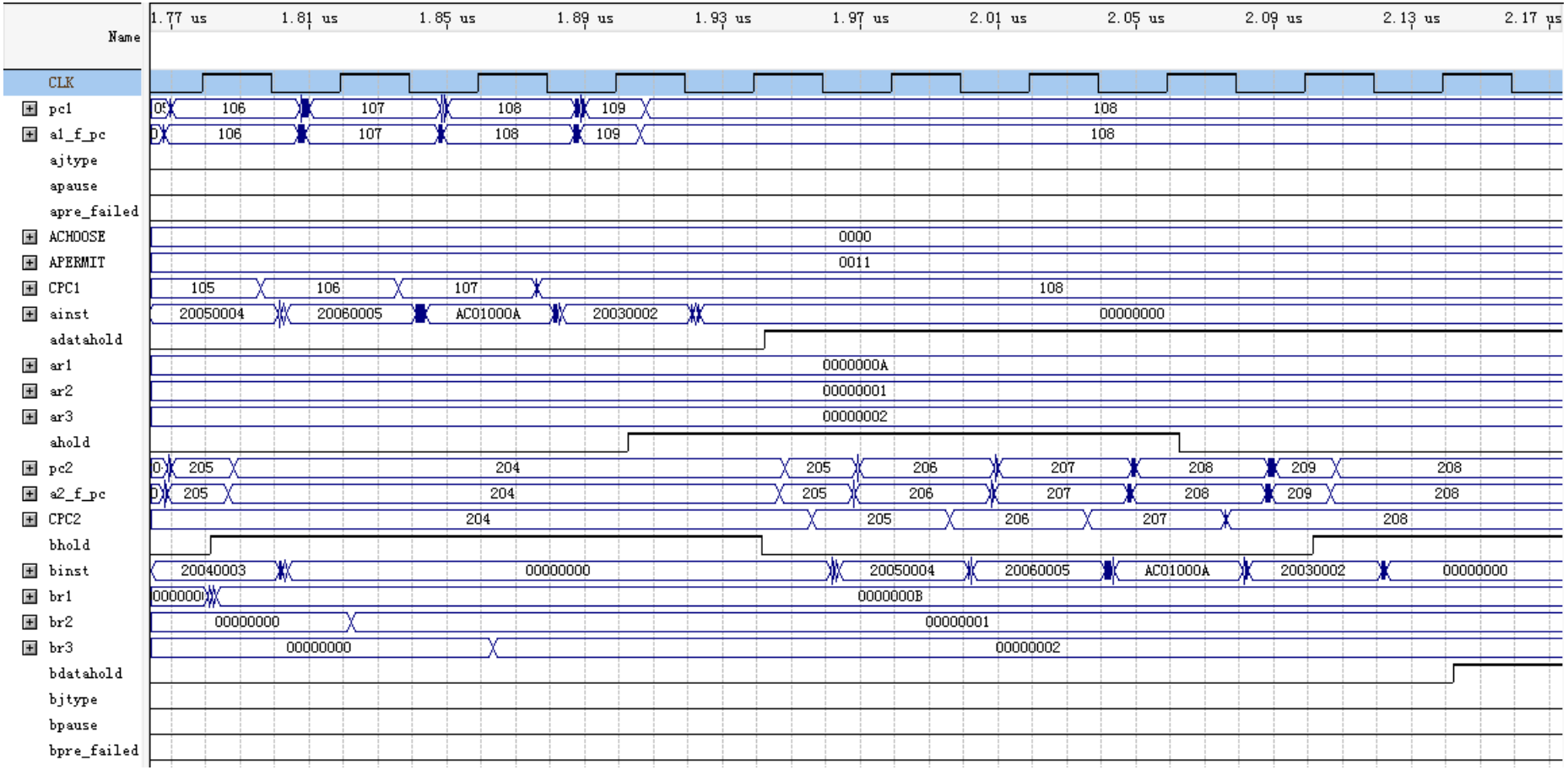
继续执行指令：

```
3ca : 20030001; -- addi $3,$zero,1h
3cb : 20010001; -- addi $1,$zero,1h
3cc : ac010030; -- sw $1,30h($zero)
3cd : 8c020030; -- lw $2,30h($zero)
3ce : 14220002; -- bne $1,$2,loop
```



继续执行指令：

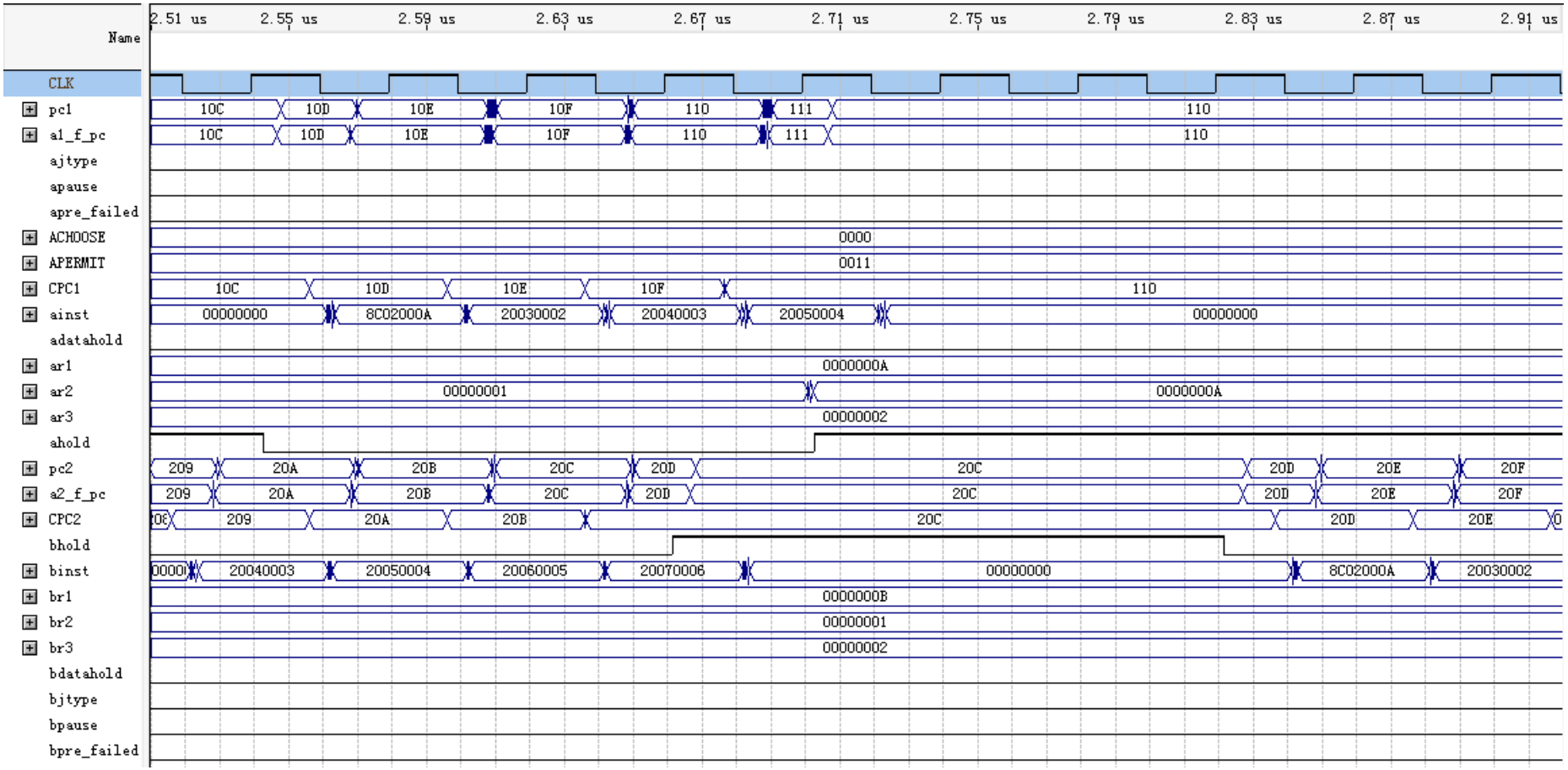
3cf : 41000100; -- st 100h	取指（41000100）两个周期后便产生 st 信号，并将 PC=100 传递给辅核 1 的 prePC：CPC1
3d0 : 41000200;-- st 200h	指令 Cache 未命中，延时一段时间后取得指令。
	取指（41000200）两个周期后再次产生 st 信号，并将 PC=200 传递给辅核 2 的 prePC：CPC2
3d1 : 080003d1;-- j loop	取指两个周期后便产生 st 信号，并将 PC=100 传递给辅核 1 的 prePC：CPC1



辅核执行指令：

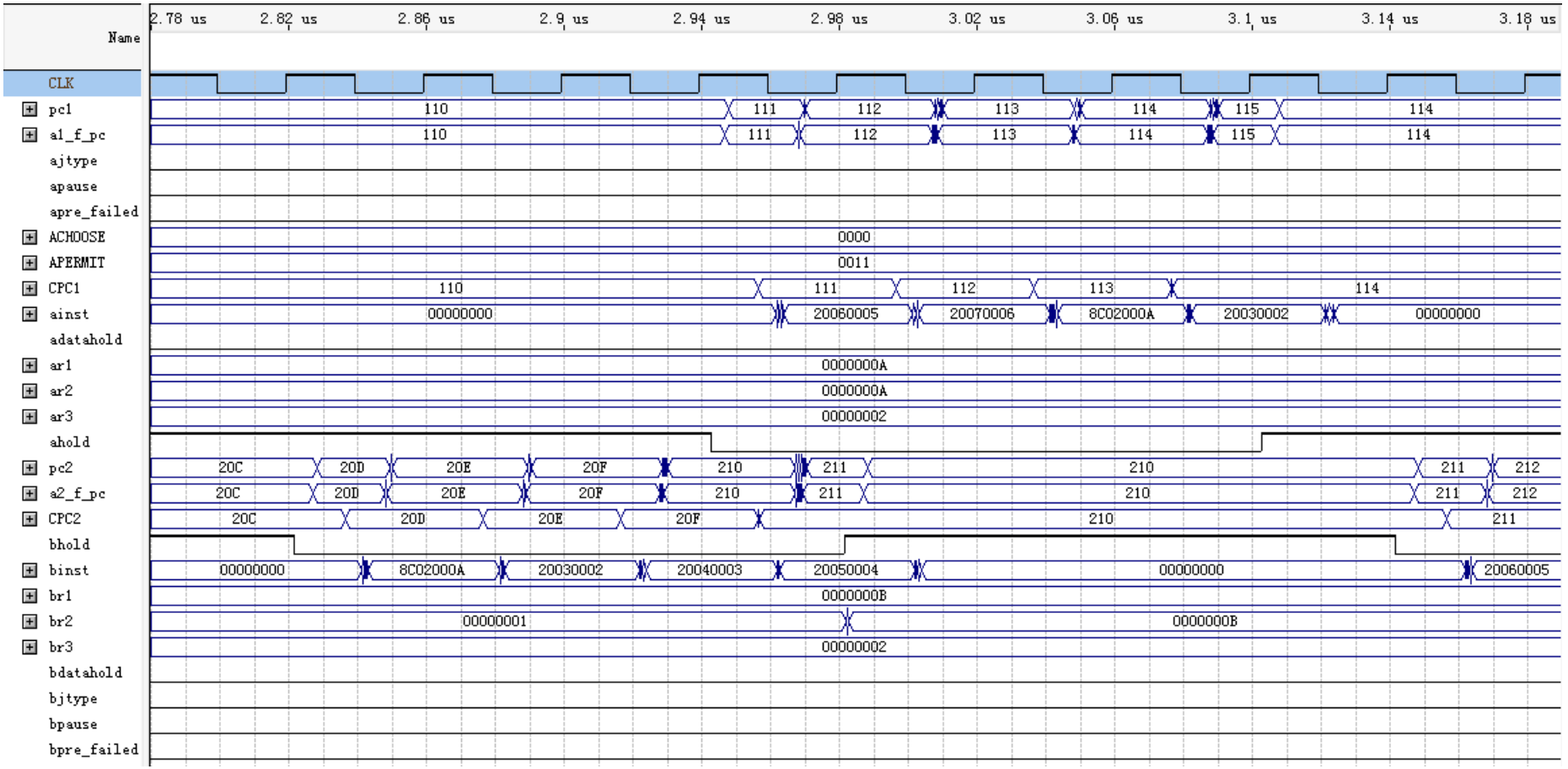
106 : ac01000a; -- sw $1,0ah($0)　　辅核 1：将$1（ar1，值为 A）中内容写入 0AH 内存。

206 : ac01000a; -- sw $1,0ah($0)　　辅核 2：将$1（br1，值为 B）中内容写入 0AH 内存。



辅核执行指令：

10c : 8c02000a; -- lw $2,0ah($0)　　辅核 1：将 0AH 内存中内容读入$2（ar2，值为 A），此时辅核 2 数据 Cache 正未命中（bhold），没写入 B。



辅核执行指令：

20c : 8c02000a; -- lw $2,0ah($0)　　辅核 2：将 0AH 内存中内容读入$2（br2，值为 B），此时辅核 2 已改写了 0AH 内存。

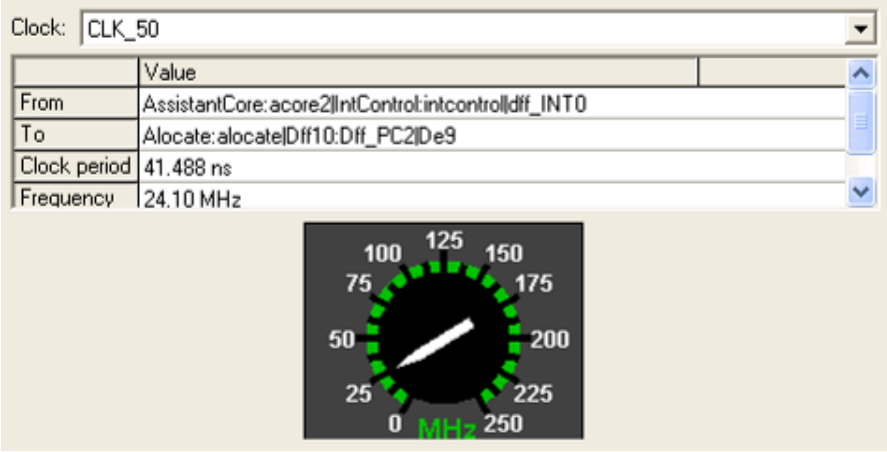
七、 本组设计的性能分析（资源情况、主频、功耗数据和自我分析）

Flow Status	Successful - Wed Mar 03 15:21:40 2010
Quartus II Version	7.2 Build 151 09/26/2007 SJ Full Version
Revision Name	MCPU
Top-level Entity Name	MCPU
Family	Cyclone II
Device	EP2C35F672C8
Timing Models	Final
Met timing requirements	No
Total logic elements	17,372 / 33,216 (52 %)
Total combinational functions	14,852 / 33,216 (45 %)
Dedicated logic registers	11,040 / 33,216 (33 %)
Total registers	11040
Total pins	431 / 475 (91 %)
Total virtual pins	0
Total memory bits	65,536 / 483,840 (14 %)
Embedded Multiplier 9-bit elements	0 / 70 (0 %)
Total PLLs	0 / 4 (0 %)

资源使用情况

PowerPlay Power Analyzer Status	Successful - Fri Sep 04 09:17:45 2009
Quartus II Version	7.2 Build 151 09/26/2007 SJ Full Version
Revision Name	MCPU
Top-level Entity Name	MCPU
Family	Cyclone II
Device	EP2C35F672C8
Power Models	Final
Total Thermal Power Dissipation	160.87 mW
Core Dynamic Thermal Power Dissipation	0.00 mW
Core Static Thermal Power Dissipation	80.09 mW
I/O Thermal Power Dissipation	80.77 mW
Power Estimation Confidence	Low: user provided insufficient toggle rate data

功耗分析数据



主频分析数据

八、 课程设计总结（包括设计的总结和还需改进的内容）

可改进意见：

1、 对于多核系统，多级 Cache 可以带来显著的性能提升。本设计中只实现了 L1 级 Cache，若能有 L2 级 Cache，就比较完备了；
2、 辅核原本设计为支持浮点运算，指令也已增加并译码，最终由于时间限制未能完成；
3、 整体设计方案为单主核、4 辅核，但为 Cache 测试方便，最终系统只有 2 个辅核，以后应当添加上。
4、 浮点加减指令码完全按照教材中给出的浮点指令码的格式，其它指令的操作码选用了指令集中没有使用到的 010000，指令码的 25 位为 0 表示有关线程的指令，为 1 表示有关接口的指令，指令码的 24 位和 23 位用以区分具体指令。新指令指令码的设计没有完全利用 32 位的空间，如有需要可以拓展功能；
5、 键盘未提供状态寄存器，采取的是及时处理的方式，工作方式就比较有局限性，只能采取中断方式；
6、 彩屏显示，没能充分利用其色彩效果，现在的显示在严格意义上与黑白液晶屏一样，还有就是代码太过于冗余，判断语句太多，整体逻辑性不是很强；
7、 辅核到接口的连接没有进一步测试；
8、 词法分析部分没有使用自动机，针对性强，虽然比较优化，可是扩展性不强；不支持子程序调用，这点也需要改进和完善；可以尝试使用 Lex 和 Yacc 来生成相应的词法和语法分析程序。

总结：

这个综合课程设计包含了本专业的很多基础知识，提高了我们对所学知识的综合运用能力，同时，我们一个团队互相合作，各司其职，克服了很多困难，形成了十分有效的工作模式，最后终于完成了整体的设计工作，虽然过程是艰辛的，但是结果是甘甜的，我们每个成员都从中受益匪浅。

验 收 报 告

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				

教师综合评价:

教师签名: _____