流水线 MIPS 处理器设计
——夏季学期综合实验报告

暮月

目录

1 设计方案

1.1 总体设计

使用 Verilog，结合 Vivado 的 IP Core 生成的 Distributed Memory，设计了支持 MIPS 核心指令子集的五级流水线。其总体设计图如图1。最终设计与设计图略有不同，但差异很小，主体结构与设计一致。

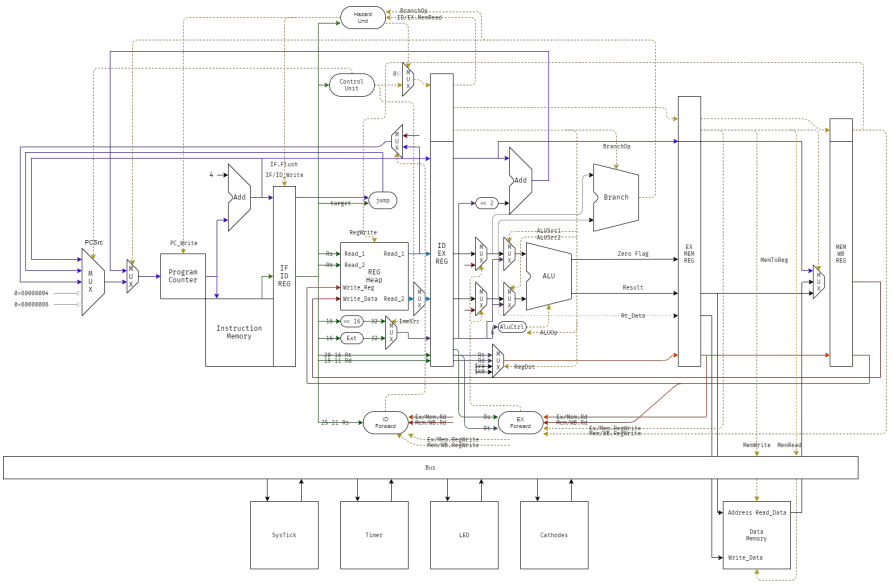


图 1: 流水线总设计图

此 CPU 支持指令如表

表 1: 支持指令

类型	指令
R 型	sll, srl, sra, jr, jalr, add, addu, sub, subu, and, or, xor, nor, slt, sltu
I 型	bltz, bgez, beq, bne, blez, bgtz, addi, addiu, slti, sltiu, andi, ori, xori, lui, lw, sw
J 型	j, jal

CPU 为五级流水线的设计，即分为取指令（IF）、译指令（ID）、执行（EX）、访存（MEM）、写回（WB）五个阶段。后文均用缩写代称各阶段。

CPU 有 2048Bytes 的指令存储器和数据存储器。

CPU 仅支持处理最简单的“指令不支持”异常和由总线来的中断请求。采用完全的 Forwarding 解决数据依赖的冒险，并模拟寄存器堆的先写后读。对于 Load-Use 冒险，采取 stall 一周期的方式进行解决。

分支指令在 EX 阶段进行跳转，J 型指令与 jr 和 jalr 在 ID 阶段进行跳转。jr 和 jalr 前一条指令如果写入目标寄存器，需要在汇编层面插入 nop 指令以保证跳转位置正确。下面逐项介绍此 CPU 的设计细节。

1.2 IF 阶段

IF 阶段进行 PC 的更新和指令存储器的访问。PC 采用 Verilog 代码编写的上升沿触发寄存器，指令存储器为使用 Vivado 的 Distributed Memory Generator 生成的 ROM。使用这种 ROM 的好处是可以使用 COE 文件对其中数据进行初始化，实验中的汇编代码均使用 Mars 模拟器转为 16 进制后放入 COE 文件，再使用 Vivado 对指令存储器进行初始化，之后再进行综合与实现。

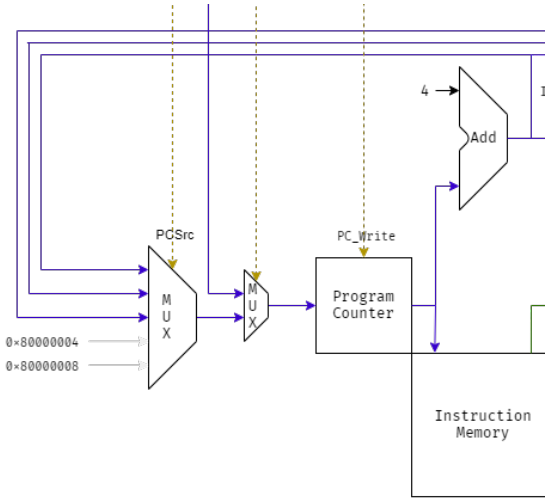


图 2: IF 阶段设计图

IF 阶段设计图如图2。当时钟上升沿到来时，PC 会根据控制信号进行更新，更改为 PC+4、jump_target、jr_target、branch_target、0x80000004、0x80000008 中的一个，分别对应顺序执行、跳转和分支的目标、中断、异常。指令存储器为 ROM，无时钟控制，根据输入地址输出对应指令。

PC 的写入受控制信号 PC_Write 的控制，具体细节见节1.7。写入的中断和异常对应的控制信号细节见节1.9。

1.3 ID 阶段

ID 阶段将 IF 阶段取出的指令进行译码，并访问寄存器堆取值。此阶段还进行立即数的扩展、控制信号的生成、冒险的判断、跳转目标的生成、ID 阶段的转发。其设计图如3。

IF 和 ID 间为上升沿触发的流水线寄存器 IF_ID_Reg，存储 IF 阶段取出的指令和 PC+4。

寄存器堆为 Verilog 代码实现的类似 RAM 的结构，读取寄存器为组合逻辑，写寄存器需时钟和 MEM_WB_Reg 中存储的 RegWrite 信号一起触发。采用 ID_Forward 对寄存器取值进行转发，模拟先写后读的功能。注意，这里设计图的转发有一定错误。寄存器的两个输出都进行转发，其中 Read_1 的输出数据进行转发后再作为 jr 的目标地址和下一阶段的输入数据。其结构与对 Read_2 的转发类似，但由于设计图已基本绘制完毕，不方便更改而遗留了这一处错误。

立即数扩展部分进行三种扩展方式：左移 16 位、符号扩展、零扩展，采用控制信号 ExtOp 和 ImmSrc 进行控制。

控制信号产生模块接收指令的 OpCode 和 Funct，以及中断请求 IRQ 和监督信号 Su-

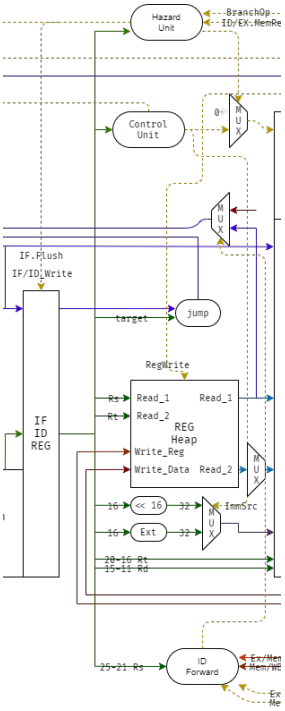


图 3: ID 阶段设计图

pervised，产生 CPU 绝大部分的控制信号。部分控制信号写入流水线寄存器，部分直接连接到其他模块进行控制。

冒险模块和转发模块的细节见节1.8。

1.4 EX 阶段

EX 阶段主要使用 ALU 进行整型运算，同时生成分支目标和 branch_hazard 信号。其设计图如4。

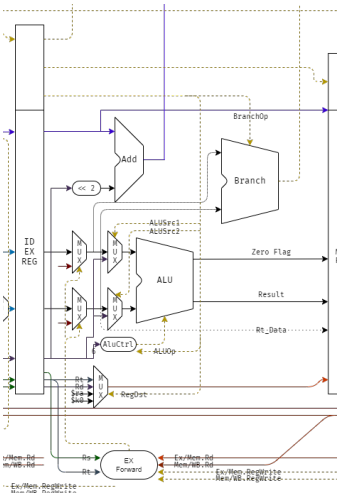


图 4: EX 阶段设计图

ALU 接收两个 32 位整型数据，支持位运算（与、或、异或、或非、左移、逻辑右移、代数右移）、加法、减法、小于比较功能。其控制信号由 ALUControl 接收 ID_EX_Reg 存

储的控制信号后产生。

Branch 接收两个 32 位整型数据，根据控制信号进行等于、不等于、小于等于零、大于等于零、小于零、大于零的判断。其输出为 branch_hazard 信号，即为高电平时需要进行分支跳转。同时，将 ID_EX_Reg 存储的立即数左移两位后，与存储的 PC+4 求和，得到分支目标。

来自 ID_EX_Reg 的寄存器堆读出数据进行 EX 阶段的转发，然后与立即数一起接收控制信号的选择，由指令功能决定是否输入 ALU。其转发细节见节1.8。

EX 阶段还根据控制信号，在 Rt、Rd、$ra、$k0 中选择一个作为该指令的写入寄存器。

1.5 MEM 阶段

MEM 阶段对数据存储器进行访存。由于此 CPU 设计中需要加入外设，故设计了总线，并将外设和数据存储器作为其子模块。故对于 CPU，此阶段为与总线的交互。MEM 阶段的设计图见5，总线的设计图见6。

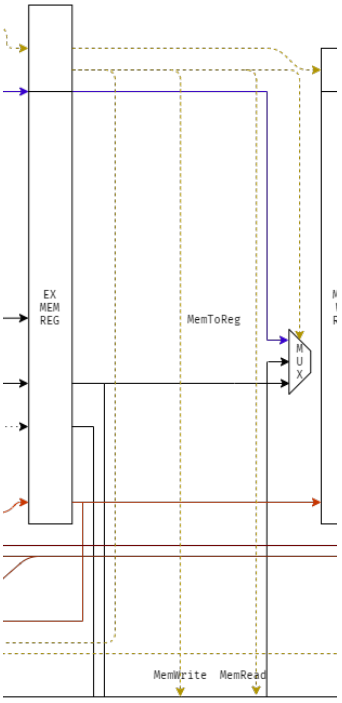


图 5: MEM 阶段设计图

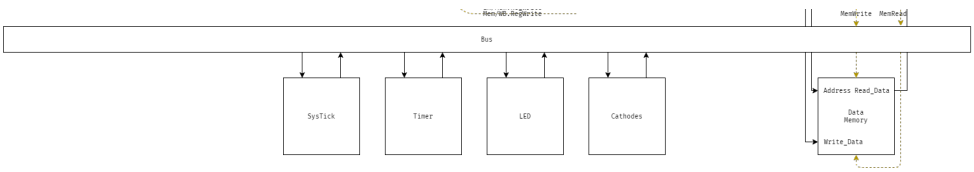


图 6: Bus 设计图

对于抽象出来的总线，CPU 在此阶段有控制信号读使能、写使能，以及 32 位地址、输入、输出。由于有中断请求和输出到数码管与 LED 的需要，CPU 接收来自总线的相关信

号。在时钟上升沿，如果读使能，总线输出地址对应的数据；如果写使能，将相应数据写入。由于此处理器一个周期只能执行写或者读，故 32 位地址共享，不作区分。

具体各外设和数据存储器的实现见节1.10。

MEM 阶段还根据 EX_MEM_Reg 中存储的 MemToReg 信号，从 PC、PC+4、Rt 内容、访存结果中选择写入寄存器堆的数据。

1.6 WB 阶段

本阶段不产生任何信号，仅在时钟上升沿根据控制信号决定是否写入寄存器堆。为了模拟先写后读，本阶段的写入数据会参入其他阶段的转发。

由于寄存器堆的写入位上升沿触发，且与流水线寄存器为同一时钟控制，实际写入会晚一周期。不过由于采用转发机制，对汇编程序的执行没有影响。

1.7 分支与跳转

CPU 支持 j、jal、jr、jalr 以及多条分支指令，分别在 ID 阶段和 EX 阶段进行判断和执行。

对于 j、jal、jr、jalr 四条指令，在 ID 阶段控制信号产生模块生成 jump_hazard 信号，传入冒险处理模块。冒险处理模块产生 IF_Flush 信号，将 IF_ID_Reg 的指令置为 32 位 0，从而清理掉下一个周期 ID 阶段将译码的信号。

由于 jr 和 jalr 需要访问寄存器 Rs，故使用转发将前前条和前前前条指令对应要写入的指令转发。而对于前一条指令，可以采用类似 Load-Use 冒险判断的方式，进而延迟一个周期。但由于连线较多，改为汇编程序中添加一个 nop 指令的方式进行解决。即需要采用类似下方的语句以保证跳转结果的正确性。

```
to_user_mode:
la      $ra, main
nop
jr      $ra
```

对于分支指令，处理器默认不跳转，即 IF 和 ID 不暂停工作。在分支指令的 ID 阶段，控制信号产生模块根据分支指令的类型，产生 BranchOp 并存储到流水线寄存器中。在 EX 阶段，Branch 模块接收转发后的 rs_data 和 rt_data 作为输入，并进行相关判断。当判断为真时，branch_hazard 信号拉高，冒险模块随即产生 IF_Flush 和 ID_Flush 指令，将对应的 IF_ID_Reg 和 ID_EX_Reg 中的关键信号改为 0，即清理掉错误的指令。而当判断为假时，处理器继续运行，可以减少一定的 CPI。简言之，此 CPU 对分支的预测总为假。

PC 的更新会先根据 PCSrc 在 PC+4、跳转目标、异常和中断处理地址中进行选择，再根据 branch_hazard 决定是否跳转去分支目标。这个设计是有一定缺陷的，错误地将分支的优先级置于异常和中断之上，在特殊情况下会出现错误，应进行修改。

1.8 冒险的应对

MIPS 流水线处理器中主要有三类冒险：结构冒险、数据冒险和控制冒险。在节1.7中已经介绍了处理控制冒险的 Flush 信号，而此处理器各部分一周期只进行一项工作，不存在结构冒险。下面介绍数据冒险的应对。

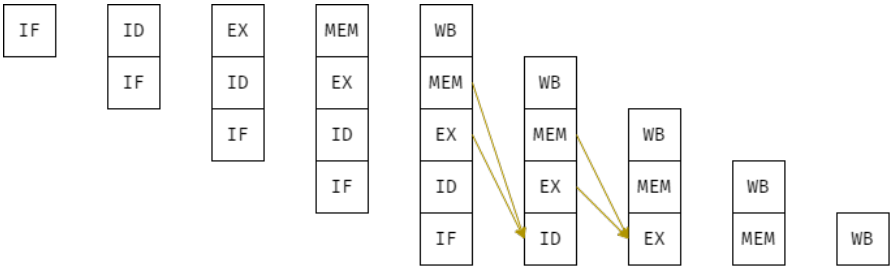


图 7: 数据冒险处理示意图

数据冒险表现为当前指令需要寄存器 Rs 和 Rt 中的数据，而前面的三条指令的写入寄存器 Rd 可能恰好是这两个寄存器。此 CPU 采用转发以解决此问题。下面阐述图7中的四条转发路径的设计思路和其解决的问题。

1) MEM_WB 转发到 ID

由于寄存器堆和流水线寄存器使用了同一时钟进行控制，实际写寄存器在 WB 阶段之后的一个周期开始时。将 MEM_WB_Reg 中的 write_data 转发到 ID，模拟了先写后读，保证 WB 阶段的同时的 ID 可以拿到要写入的值。

对于这一问题，也可以将写入寄存器的操作提前到 MEM 进行，即 MEM 准备好写使能信号和数据，在 WB 开始时便可以直接写入寄存器堆，使用组合逻辑的读端口也能拿到数据。不过涉及改动较大，此 CPU 未采用这种策略。

2) EX_MEM 转发到 ID

对于前前条执行写入寄存器的指令，不可能从寄存器中读取数据。而 jr 和 jalr 指令需要读取 Rs 寄存器的数值。故将 EX 阶段的结果从 EX_MEM_Reg 转发到 ID 阶段，从而不需要添加 nop 指令或者 stall 便可以在 ID 阶段进行跳转。

3) MEM_WB 转发到 EX

对于前前条、执行访存或者其他需要写入寄存器的指令，可以采取从 MEM_WB_Reg 转发到 EX 阶段的策略，将同一时刻正在 WB 阶段的写入数据用于 EX 的计算功能中。

4) EX_MEM 转发到 EX

对于前一条、需要写入寄存器的指令，可以采取从 EX_MEM_Reg 转发到 EX 阶段的策略，将同一时刻正在 MEM 阶段的写入数据用于 EX 的计算功能中。

此外还有 Load-Use 这种特殊的数据冒险，即访存指令后紧接着要使用访存得到的数据这一冒险。由于访存发生在 MEM 阶段，无法直接转发到 ID 或者 EX 阶段。故需要先 stall 一个周期，再将 MEM_WB_Reg 中存储的数据转发到 EX，从而解决这一冒险。stall 的具体实现为将 PC 和的 IF_ID_Reg 写使能信号拉低，从而让 PC 和 IF_ID_Reg 暂停更新一个周期。

与之类似，jr 和 jalr 前如果向对应寄存器写数据，同样会导致使用数据时还未写入正确的数据。可以增加电路实现类似的 stall 机制，但涉及改动较多，CPU 未实现此机制。需要在汇编代码中添加 nop 指令。

1.9 中断与异常

此 CPU 支持最简单的“指令不支持”异常。其实现逻辑为在 ID 阶段的控制信号生成时根据 OpCode 和 Funct 判断是否为已实现指令，如果不是且当前不是内核态，则置 Exception 为 1。相关的 PCSrc、BranchOp 等信号也根据 Exception 信号进行变化，保证

此时准确的跳转到 0x80000008。

中断的处理与异常类似，当外部中断请求进入控制信号生成单元后，调整控制信号，让 PC 准确地更改为 0x80000004。

更加具体地，当异常或中断触发时，控制信号生成单元将 PCSrc 改为 011 或 100，对应 0x80000004 和 0x80000008 两个地址；将 BranchOp 改为 0，即禁止分支跳转功能；将 RegWrite 置为 1，将 RegDst 置为 11（对应选择 $k0），即将发生异常或者中断的地址写入寄存器（$k0）；将 MemToReg 置为 10，对应选择 PC+4 作为写入寄存器数据的来源。

在控制信号发生变化后，PCSrc 阻止冒险模块将 IF_Flush 信号拉高，确保 PC 可以顺利更改为中断和异常处理程序的入口地址 0x80000004 和 0x80000008。

当进入异常和中断的处理程序后，PC 最高位为 1，对应内核态。为保证内核态的监督信号在中断和异常处理时准确的一直为高，监督信号为 PC 和 IF_ID_Reg 中的 PC+4 最高位的或。当监督信号被拉高后，控制信号生成单元中保持 Exception 为 0，同时也不接受外部中断请求，直到相应的处理程序通过 jr 或者 jalr 将 PC 最高位置为 0，监督信号被拉低。

在此 CPU 的设计中，中断请求只来自总线的 IRQ 信号。

1.10 总线与外设

此 CPU 的数据存储器和其他外设一齐被抽象为总线的子模块，CPU 访存阶段仅直接和总线交互，由总线将数据根据地址转接到各个外设。下面逐个介绍各外设。

1.10.1 数据存储器

数据存储器与指令存储器类似，为使用 Vivado 的 Distributed Memory Generator 生成单口 RAM。其使用一个时钟的上升沿和一个写使能信号控制写入，而读取数据为组合逻辑。使用 Vivado 生成这样一个 RAM 的好处在于使用 COE 文件对其进行初始化，减少了仿真的麻烦。对于实际上板应用，应考虑添加串口对数据存储器中数据进行初始化。

数据存储器的使用与寄存器堆基本一致，即读没有延迟，而写会因为流水线寄存器和总线传递给数据存储器的时钟一致，导致晚一个周期。然而一般 MIPS 的汇编代码中基本不可能出现先存数据再读同一地址的行为，故此问题应可以忽略。

1.10.2 LED

LED 接受一个 32 位的输入，在时钟上升沿且写使能时将输入存储到寄存器中。当复位信号使能时，LED 的寄存器会清空。

由于此 CPU 的设计目标为运行在指定的 FPGA 板上，LED 数量有限，故仅将 LED 寄存器的低八位接出，作为总线的输出信号，进而连接到 CPU 的最外层输出，绑定在板卡的相应管脚上。

1.10.3 七段数码管（SSDT）

七段数码管（seven segment digital tube, SSDT）接收一个 32 位的输入，在时钟上升沿且写使能时将其低 12 位存入寄存器中。

在复位信号使能时，寄存器值置为 FFFF_FFFF_FFFF。最高 4 位为数码管位选信号，低 8 位为数码管段选信号，高电平点亮。

由于此外设并不进行译码，需要汇编指令译码。参照汇编指令的 switch 可以使用下方指令进行译码（待译码四位信号存于 $t2，位选信号存于 $t0，部分指令已省略）：

```
BCD_switch:
    la      $t1, BCD
    lui     $t3, 0x8000
    sll     $t4, $t2, 2
    add     $t4, $t4, $t1
    add     $t4, $t4, $t3
    nop
    jr      $t4

BCD:
    j       bcd_0
    j       bcd_1
...

bcd_0:
    li      $t1, 0x3F
    add     $t0, $t0, $t1
    j       interruptExit
...
```

此代码可以根据待译码的数字大小跳转到对应的`j bcd_X`，进而跳转到对应的信号合成部分，得到正确的 12 位控制信号。将这个信号存入 SSDT 后即可正确点亮。

1.10.4 系统时钟计数器（SysTick）

SysTick 为一个计数器，时钟上升沿时加一，不能写。当总线读使能时，可以取出其计数的值。

1.10.5 定时器

定时器有 3 个 32 位寄存器，分别成为 TH、TL、TCON，因而占据总线上的连续 12 字节空间。当 TCON 最低位的计时使能为 1 时，TL 不断增加直到变为 0xFFFFFFFF，然后变为 TH 后再次自增。同时如果 TCON 的第 2 位允许中断为 1，则 TCON 的第 3 位被拉高，视为对 CPU 的中断请求，输出到总线的 IRQ，其后 CPU 的行为见节1.9。

2 关键代码与文件清单

2.1 文件清单

源代码均位于 src 文件夹中,其中 ip 文件夹下的 dist_mem_gen_ins 和 dist_mem_gen_data 为 Vivado 的 IP Core 生成文件。

```
src
├─configs
│      cpu.xdc                 约束文件
├─designs
│   │  ALU.v                   ALU
│   │  ALUControl.v            ALU 控制信号生成单元
│   │  Branch.v                分支判断
│   │  Bus.v                   总线
│   │  Control.v               控制信号生成单元
│   │  CPU.v                   CPU 顶层文件
│   │  DataMem.v               数据存储器
│   │  EX_Forward.v            EX 阶段转发控制
│   │  EX_MEM_Reg.v            EX MEM 间流水线寄存器
│   │  Hazard.v                冒险单元
│   │  ID_EX_Reg.v             ID EX 间流水线寄存器
│   │  ID_Forward.v            ID 阶段转发控制
│   │  IF_ID_Reg.v             IF ID 间流水线寄存器
│   │  InstructionMem.v        指令存储器
│   │  MEM_WB_Reg.v            MEM WB 间流水线寄存器
│   │  ProgramCounter.v        PC 寄存器
│   │  Register.v              寄存器堆
│   ├─ip
│   │   ├─dist_mem_gen_data    数据存储器核心
│   │   └─dist_mem_gen_ins     指令存储器核心
│   ├─ip_init
│   │       data_init.coe      数据存储器初始化文件
│   │       ins_init.coe       指令存储器初始化文件
│   └─peripherals
│           LED.v              LED
│           SSDT.v             七段数码管
│           SysTick.v          系统时钟计数器
│           Timer.v            定时器
└─tests
    │  tb_CPU.v                CPU testbench
    │  tb_InstructionMem.v     ins_mem testbench
    └─asm
            a.in                        Mars 测试用读入文件
            a.out                       Mars 测试用输出文件
            bubble_sort.asm             冒泡排序汇编
            bubble_sort.hex             冒泡排序汇编 16 进制码
            bubble_sort_Mars.asm        Mars 测试用冒泡排序汇编
            gen.py                      a.in 生成用 py
```

2.2 关键代码

CPU.v PC 更新代码

```
assign PC_p4 = {PC[31], PC[30: 0] + 31'd4};
assign PC_next =
       branch_hazard ? branch_target :
       (PCSrc == 3'b000) ? PC_p4 :
       (PCSrc == 3'b001) ? jump_target :
       (PCSrc == 3'b010) ? jr_target :
       (PCSrc == 3'b011) ? 32'h80000004 :
       (PCSrc == 3'b100) ? 32'h80000008 :
       32'h80000000;
ProgramCounter program_counter(
                 .clk(clk),
                 .reset(reset),
                 .wen(PC_wen),
                 .pc_next(PC_next),
                 .pc(PC)
               );
```

CPU.v ID 阶段转发

```
assign rs_data_forward_id =
    (id_forward_1 == 2'b00) ? rs_data :
    (id_forward_1 == 2'b01) ? ex_mem.alu_out :
    mem_wb.write_data;
assign rt_data_forward_id = id_forward_2 ?
        mem_wb.write_data :
        rt_data;
```

CPU.v EX 阶段转发及 ALU 输入控制

```
assign rs_data_forward_ex =
    (ex_forward_1 == 2'b01) ? ex_mem.alu_out :
    (ex_forward_1 == 2'b10) ? mem_wb.write_data :
    id_ex.rs_data;
assign rt_data_forward_ex =
    (ex_forward_2 == 2'b01) ? ex_mem.alu_out :
    (ex_forward_2 == 2'b10) ? mem_wb.write_data :
    id_ex.rt_data;

assign alu_src1 =
    (id_ex.ALUSrc[1 : 0] == 2'b01) ? id_ex.Imm :
    (id_ex.ALUSrc[1 : 0] == 2'b10) ? 32'h0 :
```

```
    rs_data_forward_ex;
assign alu_src2 = id_ex.ALUSrc[2] ?
        id_ex.Imm :
        rt_data_forward_ex;
```

CPU.v 写回数据选择

```
assign write_data =
    (ex_mem.MemToReg == 2'b10) ?
    (ex_mem.Rd == 5'd26 ?
     ex_mem.PC_p4 - 32'h4 :
     ex_mem.PC_p4) :
    (ex_mem.MemToReg == 2'b01) ? mem_out :
    ex_mem.alu_out;
```

Hazard.v 冒险控制信号

```
assign load_use_hazard =
    reset ? 1'b0 :
    ID_EX_MemRead &&
    (ID_EX_Rt == IF_ID_Rs ||
     ID_EX_Rt == IF_ID_Rt);

assign PC_wen = ~load_use_hazard;
assign IF_wen = ~load_use_hazard;

assign IF_Flush =
    reset ?
    1'b0 :
    (jump_hazard ||
    branch_hazard) &&
    (PCSrc != 3'b011 &&
    PCSrc != 3'b100);
assign ID_Flush = reset ? 1'b0 : branch_hazard;
```

Bus.v 总线与外设使能信号

```
assign Data_Mem_en = (Address <32'h40000000) && en;
assign Data_Mem_wen = (Address <32'h40000000) && wen;

assign Timer_en =
    (Address >= 32'h40000000 &&
    Address <= 32'h4000000B) ?
    en : 0;
```

```
assign Timer_wen =
    (Address >= 32'h40000000 &&
    Address <= 32'h4000000B) ?
    wen : 0;

assign LED_en = (Address == 32'h4000000C) &&en;
assign LED_wen = (Address == 32'h4000000C) & wen;

assign SysTick_en = (Address ==32'h40000014) && en;

assign SSDT_en = (Address == 32'h40000010) & en;
assign SSDT_wen = (Address == 32'h40000010) & wen;

assign led = LED_dout[7 : 0];

assign dout =
       Data_Mem_en ? Data_Mem_dout :
       Timer_en ? Timer_dout :
       SysTick_en ? SysTick_dout :
       LED_en ? LED_dout :
       SSDT_en ? {20'b0, ssdt} : 32'h0;
```

3 综合与实现情况

3.1 资源使用与时序性能

调整 Vivado 的 flatten_hierarchy 为 none、rebuild、full 三种情况，针对 100MHz 时钟进行综合和实现，得到性能概览如图8。

Name	Constraints	Status	WNS	TNS	WHS	THS	TPWS	Total Power	Failed Routes	LUT	FF
synth_1 (active)	**constrs_1**	**synth_design Complete!**								**1635**	**1564**
impl_1 (active)	**constrs_1**	**route_design Complete!**	**0.591**	**0.000**	**0.069**	**0.000**	**0.000**	**0.111**	**0**	**2006**	**1583**
synth_rebuilt	constrs_1	Synthesis Out-of-date								1602	1564
impl_rebuilt	constrs_1	Implementation Out-of-date	0.512	0.000	0.085	0.000	0.000	0.116	0	1985	1583
synth_full_flatten_hierarchy	constrs_1	Synthesis Out-of-date								1901	1564
impl_full_flatten_hierarchy	constrs_1	Implementation Out-of-date	0.370	0.000	0.099	0.000	0.000	0.117	0	1988	1583

图 8: 综合实现性能概览

总体上，不打平层次的时序裕量最多，达到 0.591ns，但使用的 LUT 也最多，综合使用了 1635 个 LUT 和 1564 个触发器，实现后上涨到 2006 个 LUT 和 1583 个触发器。相比之下，rebuild 和 full 两种模式进行层次打平后时序裕量下降，使用板上资源也没有较大的减幅，故下面针对不打平的情况进行分析。

逻辑使用情况见表2。总体上实现使用了近 10% 的 LUT 资源和近 4% 的寄存器资源。一个较为复杂的 MIPS5 级流水线处理器只需要不到 10% 的资源，这体现出了 FPGA 的巨大潜力。

表 2: 逻辑资源使用情况

Site Type	Used	Fixed	Available	Util%
Slice LUTs	2006	0	20800	9.64
LUT as Logic	1750	0	20800	8.41
LUT as Memory	256	0	9600	2.67
LUT as Distributed RAM	256	0		
LUT as Shift Register	0	0		
Slice Registers	1583	0	41600	3.81
Register as Flip Flop	1583	0	41600	3.81
Register as Latch	0	0	41600	0.00
F7 Muxes	384	0	16300	2.36
F8 Muxes	64	0	8150	0.79

由于没有打平层次，可以进一步查看各层次使用的资源，使用情况见表3。可以看到，使用资源最多的时寄存器堆，其次是总线和 ALU。进一步优化板上资源应考虑减少这几处不必要的或者冗余的资源。

表 3: 逻辑资源分层使用情况

Name	LUTs	Regs	F7 Mux	F8 Mux	Slice	LUT Logic	LUT Mem
CPU	2006	1583	384	64	796	1750	256
(ALU)	406	0	0	0	111	406	0
(ALUControl)	9	0	0	0	6	9	0
(Branch)	22	0	0	0	12	22	0
(Bus)	448	192	128	64	162	192	256
(Control)	26	0	0	0	12	26	0
(EX Forward)	14	0	0	0	6	14	0
(EX MEM)	0	106	0	0	69	0	0
(Hazard)	7	0	0	0	4	7	0
(ID EX)	2	160	0	0	69	2	0
(ID Forward)	10	0	0	0	3	10	0
(IF ID)	1	63	0	0	29	1	0
(InsMem)	110	0	0	0	34	110	0
(MEM WB)	0	38	0	0	26	0	0
(PC)	0	32	0	0	8	0	0
(Register)	607	992	256	0	418	607	0

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	0.591 ns	Worst Hold Slack (WHS):	0.069 ns	Worst Pulse Width Slack (WPWS):	3.750 ns
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	5420	Total Number of Endpoints:	5420	Total Number of Endpoints:	1840

All user specified timing constraints are met.

图 9: 实现时序裕量

实现后的时序裕量为 0.591ns，最长路径的电路见图10，细节见图11。。可以看出，这一条路径是从 MEM_WB 向 EX 阶段转发寄存器，进入分支判断模块后再进入冒险单元产生 IF_Flush 信号。这表明此处理器的控制信号产生较为复杂，需要多个模块逐级产生。也因为过于复杂，优化难度较大，经过尝试后我认为需要对总设计做出较大的改动，遂放弃。

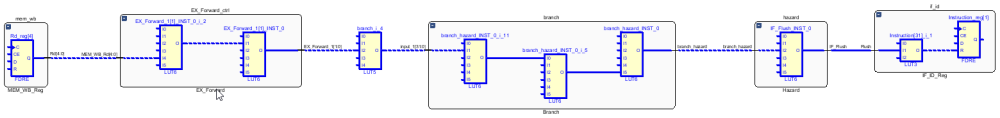
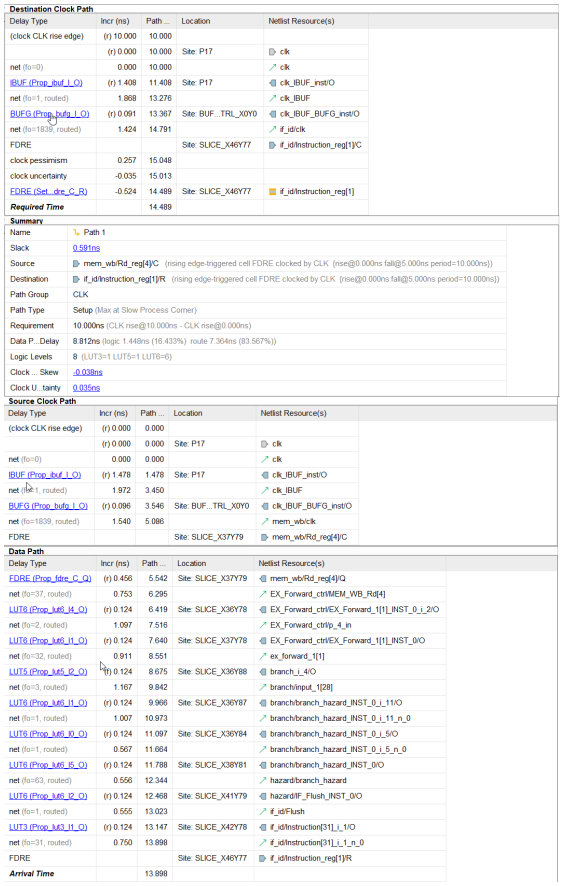


图 10: 最长路径



Destination Clock Path

Delay Type	Incr (ns)	Path ...	Location	Netlist Resource(s)
(clock CLK rise edge)	(r) 10.000	10.000		
	(r) 0.000	10.000	Site: P17	clk
net (fo=0)	0.000	10.000		clk
IBUF (Prop_ibuf_I_O)	(r) 1.408	11.408	Site: P17	clk_IBUF_inst/O
net (fo=1, routed)	1.868	13.276		clk_IBUF
BUFG (Prop_bufg_I_O)	(r) 0.091	13.367	Site: BUF...TRL_X0Y0	clk_IBUF_BUFG_inst/O
net (fo=1839, routed)	1.424	14.791		if_id/clk
FDRE			Site: SLICE_X46Y77	if_id/Instruction_reg[1]/C
clock pessimism	0.257	15.048		
clock uncertainty	-0.035	15.013		
FDRE (Set...dre_C_R)	-0.524	14.489	Site: SLICE_X46Y77	if_id/Instruction_reg[1]
Required Time		14.489		

Summary

Name	Path 1
Slack	0.591ns
Source	mem_wb/Rd_reg[4]/C (rising edge-triggered cell FDRE clocked by CLK {rise@0.000ns fall@5.000ns period=10.000ns})
Destination	if_id/Instruction_reg[1]/R (rising edge-triggered cell FDRE clocked by CLK {rise@0.000ns fall@5.000ns period=10.000ns})
Path Group	CLK
Path Type	Setup (Max at Slow Process Corner)
Requirement	10.000ns (CLK rise@10.000ns - CLK rise@0.000ns)
Data P...Delay	8.812ns (logic 1.448ns (16.433%) route 7.364ns (83.567%))
Logic Levels	8 (LUT3=1 LUT5=1 LUT6=6)
Clock ... Skew	-0.038ns
Clock U...tainty	0.035ns

Source Clock Path

Delay Type	Incr (ns)	Path ...	Location	Netlist Resource(s)
(clock CLK rise edge)	(r) 0.000	0.000		
	(r) 0.000	0.000	Site: P17	clk
net (fo=0)	0.000	0.000		clk
IBUF (Prop_ibuf_I_O)	(r) 1.478	1.478	Site: P17	clk_IBUF_inst/O
net (fo=1, routed)	1.972	3.450		clk_IBUF
BUFG (Prop_bufg_I_O)	(r) 0.096	3.546	Site: BUF...TRL_X0Y0	clk_IBUF_BUFG_inst/O
net (fo=1839, routed)	1.540	5.086		mem_wb/clk
FDRE			Site: SLICE_X37Y79	mem_wb/Rd_reg[4]/C

Data Path

Delay Type	Incr (ns)	Path ...	Location	Netlist Resource(s)
FDRE (Prop_fdre_C_Q)	(r) 0.456	5.542	Site: SLICE_X37Y79	mem_wb/Rd_reg[4]/Q
net (fo=37, routed)	0.753	6.295		EX_Forward_ctrl/MEM_WB_Rd[4]
LUT6 (Prop_lut6_I4_O)	(r) 0.124	6.419	Site: SLICE_X36Y78	EX_Forward_ctrl/EX_Forward_1[1]_INST_0_i_2/O
net (fo=2, routed)	1.097	7.516		EX_Forward_ctrl/p_4_in
LUT6 (Prop_lut6_I1_O)	(r) 0.124	7.640	Site: SLICE_X37Y78	EX_Forward_ctrl/EX_Forward_1[1]_INST_0/O
net (fo=32, routed)	0.911	8.551		ex_forward_1[1]
LUT5 (Prop_lut5_I2_O)	(r) 0.124	8.675	Site: SLICE_X36Y88	branch_i_4/O
net (fo=3, routed)	1.167	9.842		branch/input_1[28]
LUT6 (Prop_lut6_I1_O)	(r) 0.124	9.966	Site: SLICE_X36Y87	branch/branch_hazard_INST_0_i_11/O
net (fo=1, routed)	1.007	10.973		branch/branch_hazard_INST_0_i_11_n_0
LUT6 (Prop_lut6_I0_O)	(r) 0.124	11.097	Site: SLICE_X36Y84	branch/branch_hazard_INST_0_i_5/O
net (fo=1, routed)	0.567	11.664		branch/branch_hazard_INST_0_i_5_n_0
LUT6 (Prop_lut6_I5_O)	(r) 0.124	11.788	Site: SLICE_X38Y81	branch/branch_hazard_INST_0/O
net (fo=63, routed)	0.556	12.344		hazard/branch_hazard
LUT6 (Prop_lut6_I2_O)	(r) 0.124	12.468	Site: SLICE_X41Y79	hazard/IF_Flush_INST_0/O
net (fo=1, routed)	0.555	13.023		if_id/Flush
LUT3 (Prop_lut3_I1_O)	(r) 0.124	13.147	Site: SLICE_X42Y78	if_id/Instruction[31]_i_1/O
net (fo=31, routed)	0.750	13.898		if_id/Instruction[31]_i_1_n_0
FDRE			Site: SLICE_X46Y77	if_id/Instruction_reg[1]/R
Arrival Time		13.898		

图 11: 最长路径细节

实现结果表明，此 CPU 的主频可以达到 106.281MHz。

3.2 仿真验证

仿真验证分两部分：Vivado 仿真与 Mars 验证。使用的汇编代码见附录A，其内容为使用冒泡排序对一组设计好的，放在数据存储器 0-127 行的数据进行排序。为节省仿真时间，排序用的数据如图12。即只需将少数数字排序便全部有序。

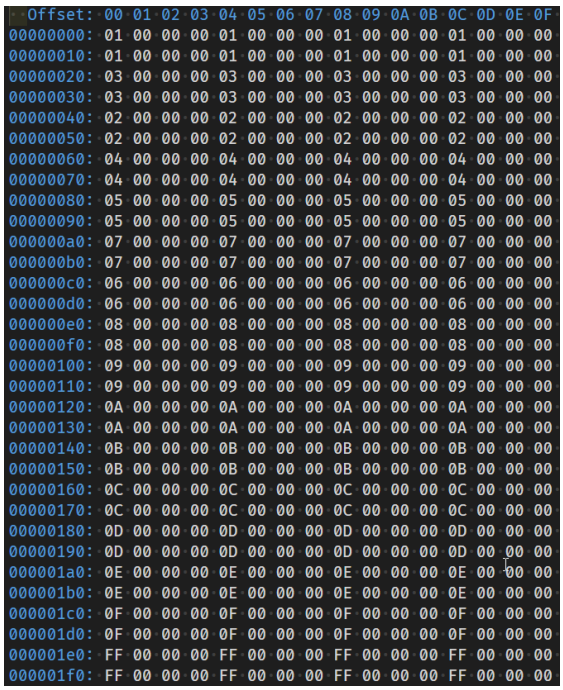
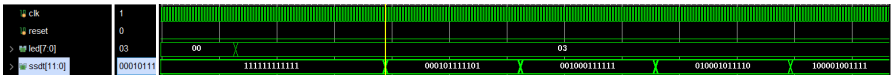


图 12: 排序数据

由于没有进行层次打平，可以非常容易地查看寄存器中数据，下面查看仿真时用到的寄存器和最后输出的 SSDT 信号。

(a) 排序开始寄存器数据

(b) 排序结束寄存器数据



(c) ssdt 信号

图 13: 仿真波形

可以看到，CPU 顺利执行了冒泡排序，将设计好的数据进行了从小到大的排序，并且输出了周期“0x3D06”，即执行了 15622 个周期。

使用脚本生成同样的数据的二进制文件，结合之前理论课汇编作业的代码和附录A中的冒泡排序，用 Mars 模拟器统计排序过程中的指令数如图14。得到总共指令数为 11541，故 CPI 为 1.3536。

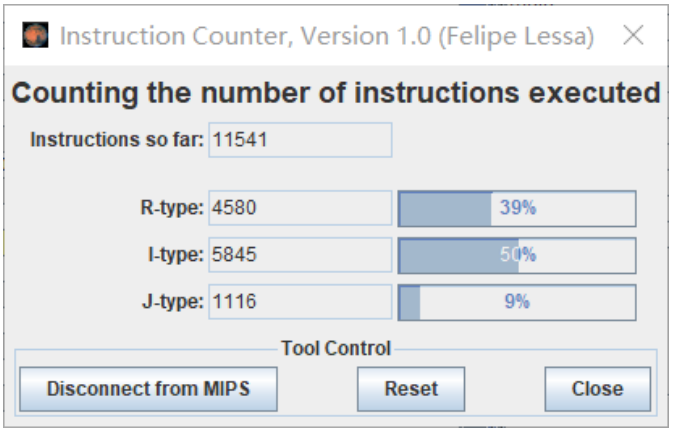


图 14: 指令数量

由于此 CPU 的设计并未将分支跳转前移至 ID 阶段，且汇编代码并未特意按照分支发生的可能性进行编写，故 CPI 并不算很低。如果经过精心设计，CPI 可能可以有较大的缩减。

综上，此 CPU 的主频为 106.281MHz，测试用程序的 CPI 为 1.3536。与单周期 MIPS 32 CPU 的 66.1857MHz 比起来有非常显著的提升。

4 经验体会

本次实验比较全面地体会了硬件设计的流程，学会了一些设计的思路。但是没有能够对设计进行比较好的优化，也未能尝试仿照《计算机组成与设计：硬件/软件接口》所叙述的超长指令字、超标量、分支预测等方法提高 CPI，略有遗憾。

回顾本次实验，主要经验可以概括为下面两部分：

(1) **先设计，再实现，事半功倍**

本次实验中，我先着手使用 drawio 绘制了 CPU 的总设计图，然后基于此设计图进行实现。因为在绘制设计图时已经思考了大部分信号的依赖关系，所以参照设计图使用 Verilog 进行实现节省了很多时间。但是由于时序电路设计经验不足，加之流水线处理器的细节处理知识还不够扎实，有些设计是不很正确甚至是整个设计的拖累。

如果从头开始进行本实验，我认为我会在最初花更多的精力进行总设计，减少本次实验中设计完成后发现功能有问题而不断修修补补的方式。或许这样可以减少电路中的冗余部分，节省资源，优化时序性能。

(2) **利用好设计软件 Vivado**

在设计前，我咨询了几位学长使用 Vivado 进行硬件设计的经验。其中得到了两条对于本实验帮助很大的经验：“使用好 IP Core”和“少用行为仿真，多用时序仿真”。

基于前一条经验，我使用搜索引擎，结合 Xilinx 的 Vivado 使用指南，将单周期 MIPS 32 CPU 采用的组合逻辑指令存储器和人工编写的数据存储器换为 Distributed Memory Generator 生成的存储器。这样做最大的好处是方便修改代码和测试数据，节省了不少时间。

最初我不太理解后一条经验，认为应该行为仿真没有问题再进行时序仿真。我实际进行实验后发现，行为仿真没有延时信息，与实际执行过程差异较大。为了保证 CPU 正常工作，直接进行综合实现后的时序仿真可以更快发现问题，并可以直接对着近乎实际的波形图进行分析，节省了大量时间。

在实验的后半段，我基本上都是直接进行综合实现，然后针对有延时后的时序波形进行调整。这种调整比对着行为仿真的标准波形目测更加直观有效。

此外，我还学会了配置仿真波形的显示窗口。在充分利用了 Group 和 Divider 后，波形窗口中不同阶段的信号很好地被区分开来，加快了寻找问题的速度。

参考文献

[1] XILINX. Vivado design suite user guide: Synthesis[K/OL]. 2017.1 ed. 2017[2020-09-20]. https://www.xilinx.com/support/documentation/sw_manuals/xilinx2017_1/ug901-vivado-synthesis.pdf.

[2] XILINX. Distributed memory generator v8.0[K/OL]. 2015[2020-09-20]. https://www.xilinx.com/support/documentation/ip_documentation/dist_mem_gen/v8_0/pg063-dist-mem-gen.pdf.

[3] DAVIDA.PATTERSON, JOHNL.HENNESSY. 计算机组成与设计: 硬件/软件接口[M]. 王党辉安建峰, 译. 北京: 机械工业出版社, 2020.

A 仿真验证用汇编

```
.text
    j   to_user_mode
    j   interrupt
    j   exception

to_user_mode:
    la  $ra,    main
    nop
    jr  $ra

main:
    # load SysTick => count clocks
    li      $s7, 0x40000000         # $s7 0x40000000
    lw      $s6, 20($s7)            # $s6 (0x40000014)

    # call bubble sort
    ori     $a0, $0, 0              # $a0 0x00000000 start addr
    ori     $a1, $0, 127            # $a1 n
    jal     bubble_sort

    # load SysTick
    lw      $s5, 20($s7)
    sub     $s4, $s5, $s6

    # lower 16 bits
    ori     $t0, $0, 0xF
    and     $s0, $t0, $s4
    srl     $s4, $s4, 4
    and     $s1, $t0, $s4
    srl     $s4, $s4, 4
    and     $s2, $t0, $s4
    srl     $s4, $s4, 4
    and     $s3, $t0, $s4

    # use led
    sw      $s4, 12($s7)

    ori     $s6, $0, 0

    # Timer interrupt
    subi    $t0, $0, 0x000F
    sw      $t0, 0($s7)             # TH = 0xFFFFFFF1
    subi    $t0, $0, 1
    sw      $t0, 4($s7)             # TL = 0xFFFFFFFF
    ori     $t0, $0, 3
    sw      $t0, 8($s7)
```

```
loop:
    j       loop

bubble_sort:
    addi    $sp, $sp, -12
    sw      $ra, 0($sp)
    sw      $s1, 4($sp)
    sw      $s0, 8($sp)

    move    $s0, $a0                # array addr
    move    $s1, $a1                # n

    move    $s2, $0                 # sorted = false

L0:
    bne     $s2, $0, L0end
    ori     $s2, 0x1

    li      $t0, 1
L1:
    bge     $t0, $s1, L1end

    sll     $t1, $t0, 2
    add     $t1, $t1, $s0
    lw      $t2, 0($t1)
    lw      $t3, -4($t1)
    ble     $t3, $t2, else
    sw      $t3, 0($t1)
    sw      $t2, -4($t1)
    move    $s2, $0
else:
    addi    $t0, $t0, 1
    j       L1
L1end:
    subi    $s1, $s1, 1
    j       L0
L0end:

    lw      $s0, 8($sp)
    lw      $s1, 4($sp)
    lw      $ra, 0($sp)
    addi    $sp, $sp, 12

    move    $v0, $0
    nop
    jr      $ra

interrupt:
    ori     $t0, $0, 1
```

```
    sw      $t0, 8($s7)

    la      $t1, show
    lui     $t2, 0x8000
    sll     $t0, $s6, 2
    add     $t0, $t0, $t1
    add     $t0, $t0, $t2
    nop
    jr      $t0

show:
    j       show_0
    j       show_1
    j       show_2
    j       show_3

show_0:
    li      $t0, 0x00000100
    move    $t2, $s0
    j       BCD_switch

show_1:
    li      $t0, 0x00000200
    move    $t2, $s1
    j       BCD_switch

show_2:
    li      $t0, 0x00000400
    move    $t2, $s2
    j       BCD_switch

show_3:
    li      $t0, 0x00000800
    move    $t2, $s3
    j       BCD_switch

BCD_switch:
    la      $t1, BCD
    lui     $t3, 0x8000
    sll     $t4, $t2, 2
    add     $t4, $t4, $t1
    add     $t4, $t4, $t3
    nop
    jr      $t4

BCD:
    j       bcd_0
    j       bcd_1
    j       bcd_2
```

```
    j       bcd_3
    j       bcd_4
    j       bcd_5
    j       bcd_6
    j       bcd_7
    j       bcd_8
    j       bcd_9
    j       bcd_A
    j       bcd_B
    j       bcd_C
    j       bcd_D
    j       bcd_E
    j       bcd_F

bcd_0:
    li      $t1, 0x3F
    add     $t0, $t0, $t1
    j       interruptExit
bcd_1:
    li      $t1, 0x6
    add     $t0, $t0, $t1
    j       interruptExit
bcd_2:
    li      $t1, 0x5B
    add     $t0, $t0, $t1
    j       interruptExit
bcd_3:
    li      $t1, 0x4F
    add     $t0, $t0, $t1
    j       interruptExit
bcd_4:
    li      $t1, 0x66
    add     $t0, $t0, $t1
    j       interruptExit
bcd_5:
    li      $t1, 0x6D
    add     $t0, $t0, $t1
    j       interruptExit
bcd_6:
    li      $t1, 0x7D
    add     $t0, $t0, $t1
    j       interruptExit
bcd_7:
    li      $t1, 0x7
    add     $t0, $t0, $t1
    j       interruptExit
bcd_8:
    li      $t1, 0x7F
    add     $t0, $t0, $t1
```

```
    j       interruptExit
bcd_9:
    li      $t1, 0x6F
    add     $t0, $t0, $t1
    j       interruptExit
bcd_A:
    li      $t1, 0x77
    add     $t0, $t0, $t1
    j       interruptExit
bcd_B:
    li      $t1, 0x7C
    add     $t0, $t0, $t1
    j       interruptExit
bcd_C:
    li      $t1, 0x39
    add     $t0, $t0, $t1
    j       interruptExit
bcd_D:
    li      $t1, 0x5E
    add     $t0, $t0, $t1
    j       interruptExit
bcd_E:
    li      $t1, 0x79
    add     $t0, $t0, $t1
    j       interruptExit
bcd_F:
    li      $t1, 0x71
    add     $t0, $t0, $t1
    j       interruptExit

interruptExit:
    addi    $s6, $s6, 1
    andi    $s6, $s6, 3

    sw      $t0, 16($s7)
    li      $t0, 3
    sw      $t0, 8($s7)           # TCon = 3
    jr      $k0

exception:
    j       exception
```