中断控制器的设计与实现

王凯祺　16337233
数据科学与计算机学院　2016 级计算机类教务 3 班
指导老师：李国桢

2018 年 1 月 4 日

目录

1 选题背景

在《计算机组成原理》课程中，我学习到了单周期 CPU 、多周期 CPU 和流水线 CPU 的工作原理。通过实验，我更加清楚地了解了 CPU 取指令、指令译码、指令执行、存储器访问、结果写回这 5 个阶段。本学期在以 Computer Organization and Design [2] 为课本的学习过程中，本书在第 4.9 节为我介绍了系统中断，并提供了处理异常的数据通路。但是书上并没有介绍中断程序计数器（EPC）是如何工作的，也没有告诉我如果加上处理异常的功能，CPU 的控制单元需要做哪些改变。由于此部分没有更详尽的文档，我们在实验课上就没有做异常处理的内容。我希望自己设计一个中断控制器，与我先前设计的多周期 CPU 合并成一块可以处理中断的多周期 CPU ，为之后做此项实验的同学提供一份较为详细的设计过程供参考。

2 技术路线

我选择赛灵思（Xilinx）Vivado 设计套件作为设计、编译和仿真软件。在器件设计上，我使用 Verilog HDL 设计中断控制器、中断向量表存储器以及中断堆栈。在实验课上，我已实现了不含中断控制器的 CPU 。为了测试我的中断控制器，我将该 CPU 稍加修改，加入中断周期，再连接我自主设计的中断控制器，变成可处理中断的 CPU ，就可以对中断控制器进行测试了。

3 设计过程

3.1 中断处理流程

CPU 完成中断响应动作后，首先保存当前断点、关总中断，然后查询中断向量表，并取出中断向量地址并跳转。最后还原现场，返回原断点 [1] 。

该过程示意图如图 1 所示。

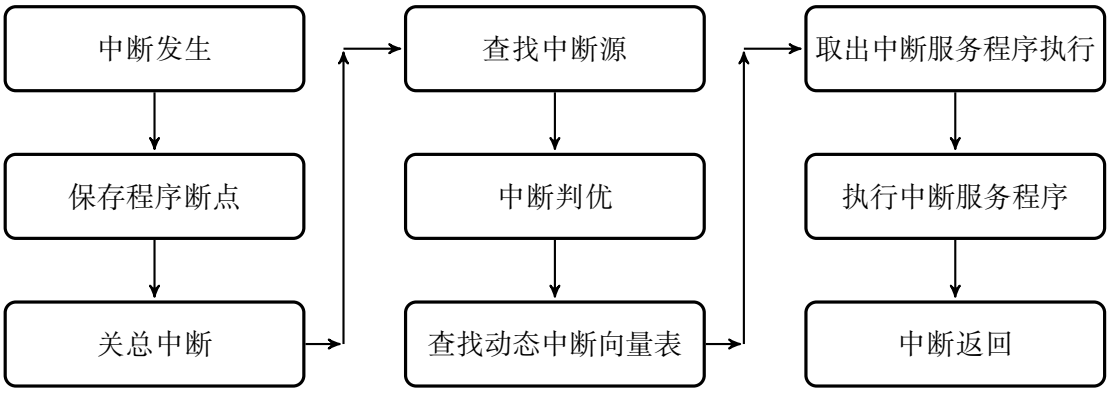


图 1: 中断处理流程示意图

3.2 中断控制器设计

中断控制器是一个中断判优的组合电路，根据输入的中断源，选择优先级最高的响应，并把对应的中断入口地址接到中断向量表存储器中，由中断向量表存储器查询中断向量地址，接到 PC 输入端。

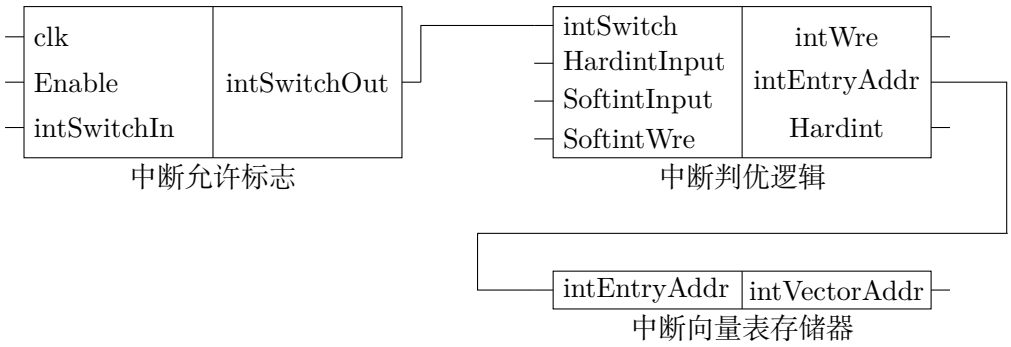


图 2: 中断控制器元器件及连线图

中断控制器元器件及连线图如图 2 所示。

中断允许标志 中断允许标志用于允许或禁止响应中断。CPU 在响应中断期间，须将中断允许标志清零，即 CPU 处于禁止中断状态。进入中断服务程序后，须将中断允许标志置 1 ，开放中断。[3]

中断允许标志使用了一个带使能端的 D 触发器，其使能端和数据输入端接 CPU ，由 CPU 控制中断允许标志是否置 1 。中断允许标志的输出接到中断判优逻辑。

中断判优逻辑 中断判优逻辑共有 4 个输入端，和 3 个输出端。

输入 intSwitch 表示中断允许标志，接中断允许标志的输出；

输入 HardintInput 表示硬中断的输入信号；

输入 SoftintInput 表示软中断的输入信号，接指令中的立即数；

输入 SoftintWre 表示软中断使能端，接 CPU；

输出 intWre 表示是否响应中断，接 CPU；

输出 intEntryAddr 表示中断入口地址，接中断向量表存储器；

输出 Hardint 表示系统响应的是否为硬中断（高电平表示是）。

若中断关闭，则不响应中断；若中断打开，选择最高优先级的中断源响应，并将中断入口地址输出。

在一个计算机系统中，存在多个中断源。中断优先级根据各个中断事件的轻重缓急程度的不同分成若干级别，给每个中断源分配一个优先权。中断判优有两种实现方法：硬件电路法和软件查询法 [3] 。为了让电路简单，我使用软件查询法来判断中断优先级。

软件查询法的流程图如图 3 所示 [3] 。

3.3 堆栈设计

中断服务程序结束后，必须能正确地返回到被中断的断点处继续原来程序的执行。为了保存断点位置，同时为了支持中断嵌套，我们需要实现一个硬件堆栈。

硬件堆栈用计数器记录栈顶位置，用存储器保存堆栈中元素的值。我们可以把它们写成一个模块，如图 4 。

在图 4 中，pushWre 表示压栈操作使能端，popWre 表示弹出操作使能端，StackTop 表示当前堆栈中实时的栈顶元素。由于弹出元素的操作会使 StackTop 立即发生改变，我们增加 D 触发器延迟该改变，保证在弹出元素的过程中仍然可以访问栈顶元素的值。

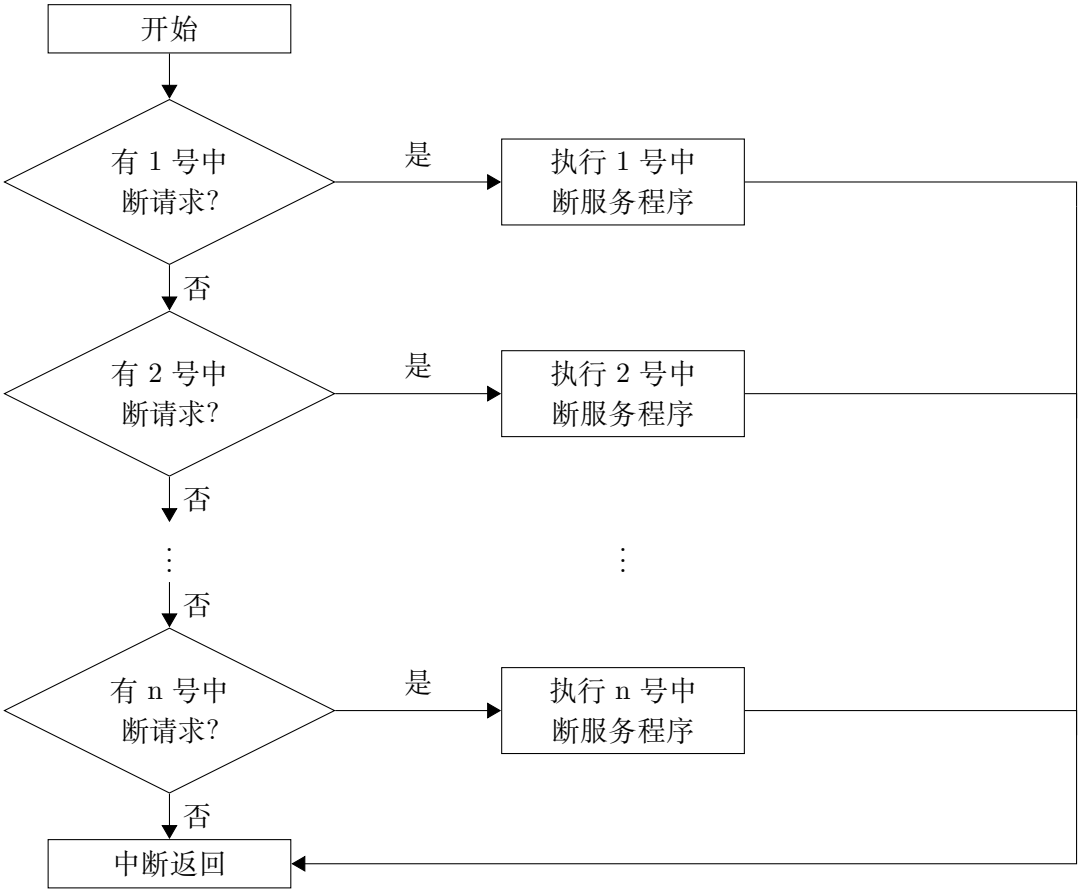


图 3: 软件查询法流程图

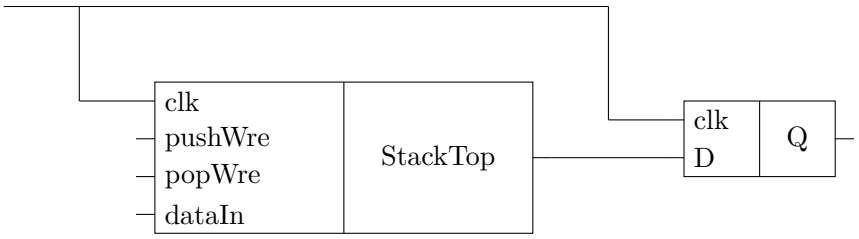


图 4: 硬件堆栈设计图

3.4 中断指令设计

我根据 MIPS 的指令集，设计了 3 条中断指令用于测试中断控制器。

3.4.1 软中断指令

功能：系统中断调用，中断入口地址为 immediate 。执行该指令时，同时关闭系统中断，将 PC+4 压入堆栈，并将 PC 转移到中断向量地址。

类型：I 类型

用法：int *immediate*

机器码：

操作码	RS	RT	立即数
101001	00000（未用）	00000（未用）	immediate（16 位）

3.4.2 开放中断指令

功能：限中断服务程序使用，用于打开系统中断。中断控制器不再屏蔽可屏蔽中断。

类型：J 类型

用法：yes

机器码：

操作码	
101000	00000000000000000000000000

3.4.3 中断返回指令

功能：限中断服务程序使用，返回到断点地址。执行该指令时，同时打开系统中断，转移到栈顶地址，并弹出堆栈。

类型：J 类型

用法：ret

机器码：

操作码	
101010	00000000000000000000000000

3.5 重新设计 CPU 状态

为了测试中断控制器，我们还需要修改多周期 CPU ，使该 CPU 可以处理中断。

3.5.1 不含中断控制器的多周期 CPU

不含中断控制器的多周期 CPU 状态转移图如图 5 所示。**本部分内容为实验课内容**。

不含中断控制器的多周期 CPU 在处理指令时，一般需要经过以下几个阶段：

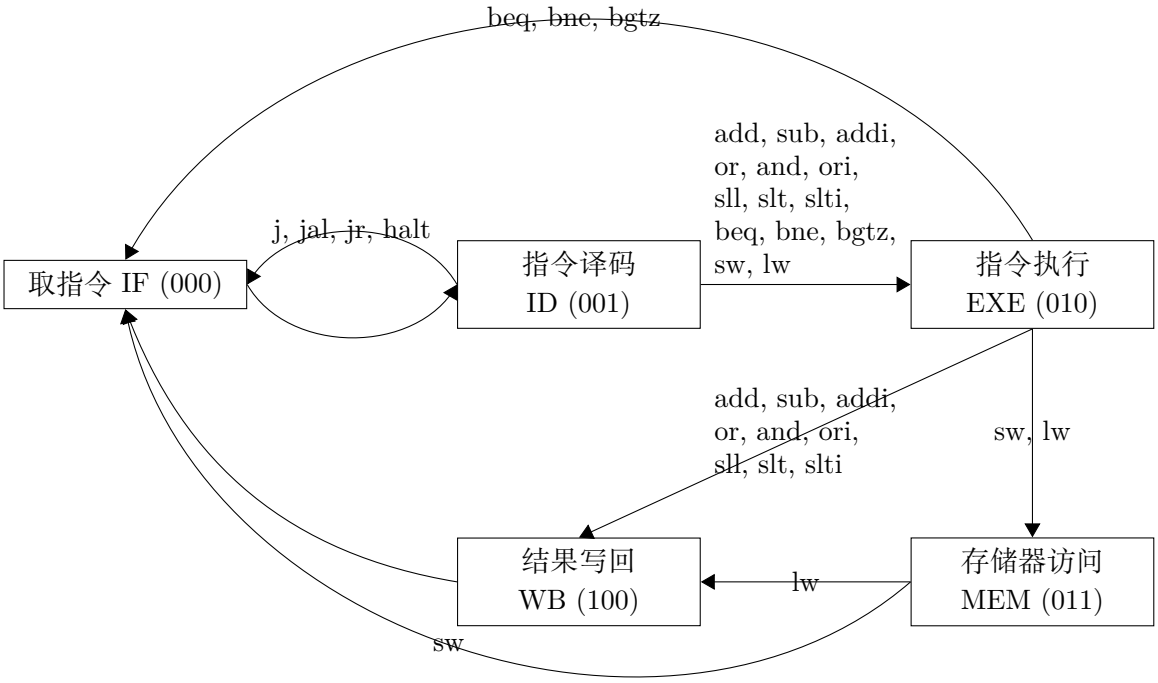


图 5: 不含中断控制器的多周期 CPU 状态转移图

1. 取指令（IF）：根据程序计数器 PC 中的指令地址，从存储器中取出一条指令，送入指令寄存器中。

2. 指令译码（ID）：对取指令操作中得到的指令进行分析并译码，确定这条指令需要完成的操作，从而产生相应的操作控制信号，用于驱动执行状态中的各种操作。

3. 指令执行（EXE）：根据指令译码得到的操作控制信号，具体地执行指令动作，然后转移到结果写回状态。

4. 存储器访问（MEM）：给定存储器的数据地址，把数据写入到存储器中数据地址所指定的存储单元或者从存储器中得到数据地址单元中的数据。

5. 结果写回（WB）：指令执行的结果或者访问存储器中得到的数据写回相应的目的寄存器中。

3.5.2 含中断控制器的多周期 CPU

含中断控制器的多周期 CPU 状态转移图如图 6 所示。图中下方的元件以及黑色和蓝色的线均为实验课上设计不含中断控制器的多周期 CPU 时何朝东老师提供的，上方的元件及绿色的线是我画的。

含中断控制器的多周期 CPU 需要能同时处理软中断和硬中断，则需在原状态转移图的基础上加上一个中断周期。这个中断周期必须在指令译码前或者整条指令执行完毕后。如果不然，某些指令的执行过程中将不包含中断周期，执行那些指令时无法响应中断，这是我们不愿意看到的。所以我把中断周期放在取指令（IF）阶段和指令译码（ID）阶段之间。

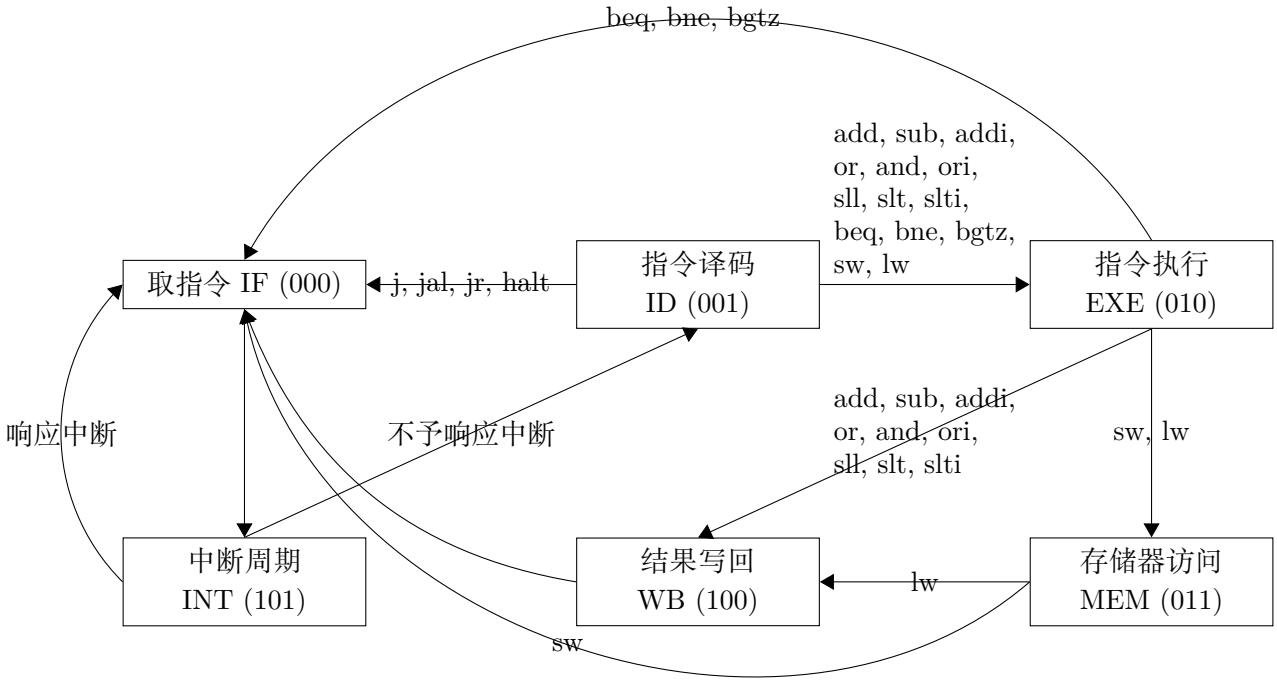


图 6: 含中断控制器的多周期 CPU 状态转移图

指令的执行步骤:

1. 首先取指令到指令寄存器中。

2. 在中断周期，将硬中断、软中断信号传输给中断控制器，由中断控制器决定是否响应中断。

3. 若中断控制器决定响应中断，则将当前 PC （硬中断）或 PC + 4 （软中断）压入堆栈，查询中断向量地址并将 PC 转移到中断向量地址，回到取指令（IF）阶段；若中断控制器决定不予响应中断，则继续执行指令译码（ID）等步骤。

含中断控制器的多周期 CPU 数据通路图如图 7 所示。

3.6 模拟仿真

3.6.1 测试程序段

我设计了一串指令序列用于测试 CPU 及中断控制器。硬中断通过开关输入中断控制器，软中断通过 int 指令输入 CPU 。

测试程序段如表 1 。

中断向量表如表 2 。

3.6.2 时序仿真

测试程序段初始化在指令存储器中，中断向量表初始化在中断向量表存储器中。

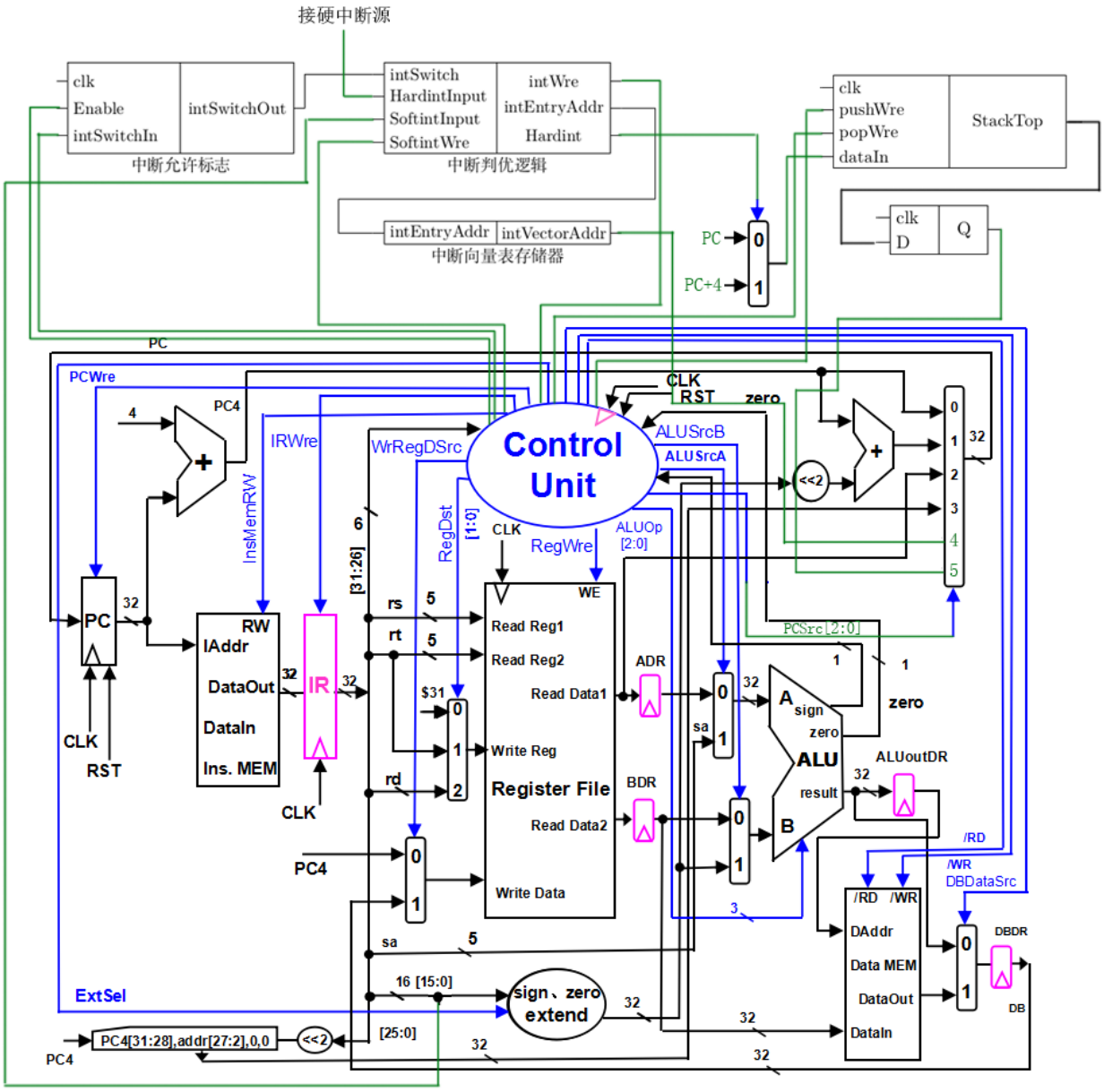


图 7: 含中断控制器的多周期 CPU 数据通路图

地址	汇编程序	op(6)	rs(5)	rt(5)	rd(5) / immediate(16)
0x00000000	addi $1, $0, 8	000010	00000	00001	0000 0000 0000 1000
0x00000004	ori $2, $0, 2	010010	00000	00010	0000 0000 0000 0010
0x00000008	or $3, $2, $1	010000	00010	00001	0001 1000 0000 0000
0x0000000C	sub $4, $3, $1	000001	00011	00001	0010 0000 0000 0000
0x00000010	or $5, $4, $2	010000	00100	00010	0010 1000 0000 0000
0x00000014	int 2	101001	00000	00000	0000 0000 0000 0010
0x00000018	addi $1, $1, 2	000010	00001	00001	0000 0000 0000 0000
0x0000001C	halt	111111	00000	00000	0000 0000 0000 0000
0x00000020		000000	00000	00000	0000 0000 0000 0000
0x00000024	yes	101000	00000	00000	0000 0000 0000 0000
0x00000028	addi $1, $1, 1	000010	00001	00001	0000 0000 0000 0001
0x0000002C	ret	101010	00000	00000	0000 0000 0000 0000
0x00000030		000000	00000	00000	0000 0000 0000 0000
0x00000034	addi $1, $1, 2	000010	00001	00001	0000 0000 0000 0010
0x00000038	ret	101010	00000	00000	0000 0000 0000 0000
0x0000003C		000000	00000	00000	0000 0000 0000 0000
0x00000040	yes	101000	00000	00000	0000 0000 0000 0000
0x00000044	addi $1, $1, 3	000010	00001	00001	0000 0000 0000 0011
0x00000048	ret	101010	00000	00000	0000 0000 0000 0000

表 1: 测试程序段

中断入口地址	中断向量地址
0x00	0x00000024
0x01	0x00000034
0x02	0x00000040

表 2: 中断向量表

时钟周期为 40 ns 。

在仿真过程中，200 ns - 300 ns 同时出现 0 号硬中断和 1 号硬中断；460 ns - 580 ns 出现 1 号硬中断。

时序仿真波形图如图 8 所示。由于版面所限，未能将完整的波形图放在正文部分。请参照附录以查看完整波形图。

通过波形图，我们可以看到该 CPU 和中断控制器能正常工作，可以同时处理软中断以及硬中断，可以处理中断嵌套的情况，可以处理多中断判优的情况。

4 效果评价

我设计的中断控制器能处理软硬中断，可以处理中断嵌套和多中断判优的情况。由于时间有限，没有来得及写输入、输出以及其它中断的中断服务程序，只提供了一段示例中断服务程序用于测试中断控制器，但在我设计的中断控制器及 CPU 上，可以在开发了输入、输出等中断服务程序的情况

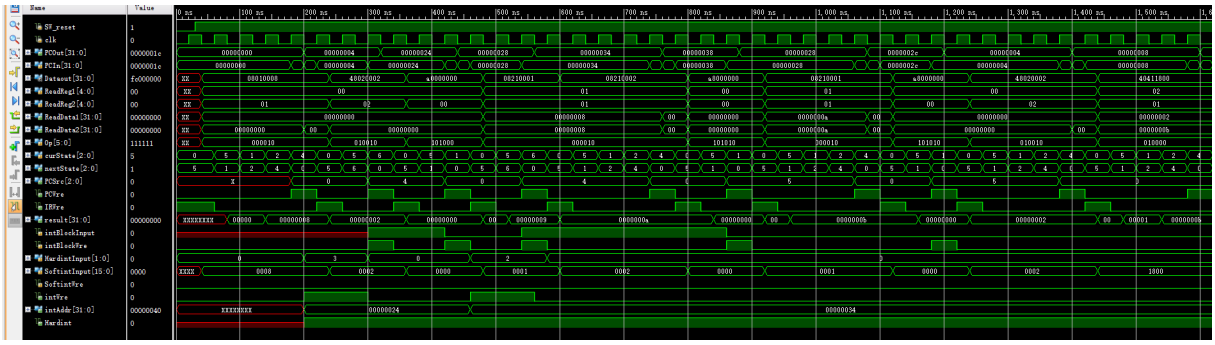


图 8: 时序仿真波形图

下处理输入、输出等中断。

自己设计一个中断控制器相比于设计 CPU 来说简单一些，但仍是一个复杂且耗时的过程，这同样需要严谨的思考和仔细的设计。在设计的过程中，我也遇到了不少的困难。由于 CPU 的控制器与中断判优逻辑均为组合电路，CPU 修改中断允许标志（intSwitchIn = 0）后，在下降沿会立即收到中断判优逻辑不予响应中断（intWre = 0）的信号，导致 CPU 控制器输出错误的指令。考虑到单个周期会产生冲突，我将中断周期拆成两个周期去完成，第一个周期先判优，将是否响应中断的结果先通过一个 D 触发器，然后给 CPU 控制器；第二个周期修改中断允许标志就不会出错了。

我的中断控制器仅在中断处理流程参考了《高校教材 计算机组成原理》[3] ，其余部分均为自主设计的。这个中断控制器填补了课本上在中断控制方面的空白。希望我设计的中断控制器可供以后学习《计算机组成原理》这门课程的同学参考。

参考文献

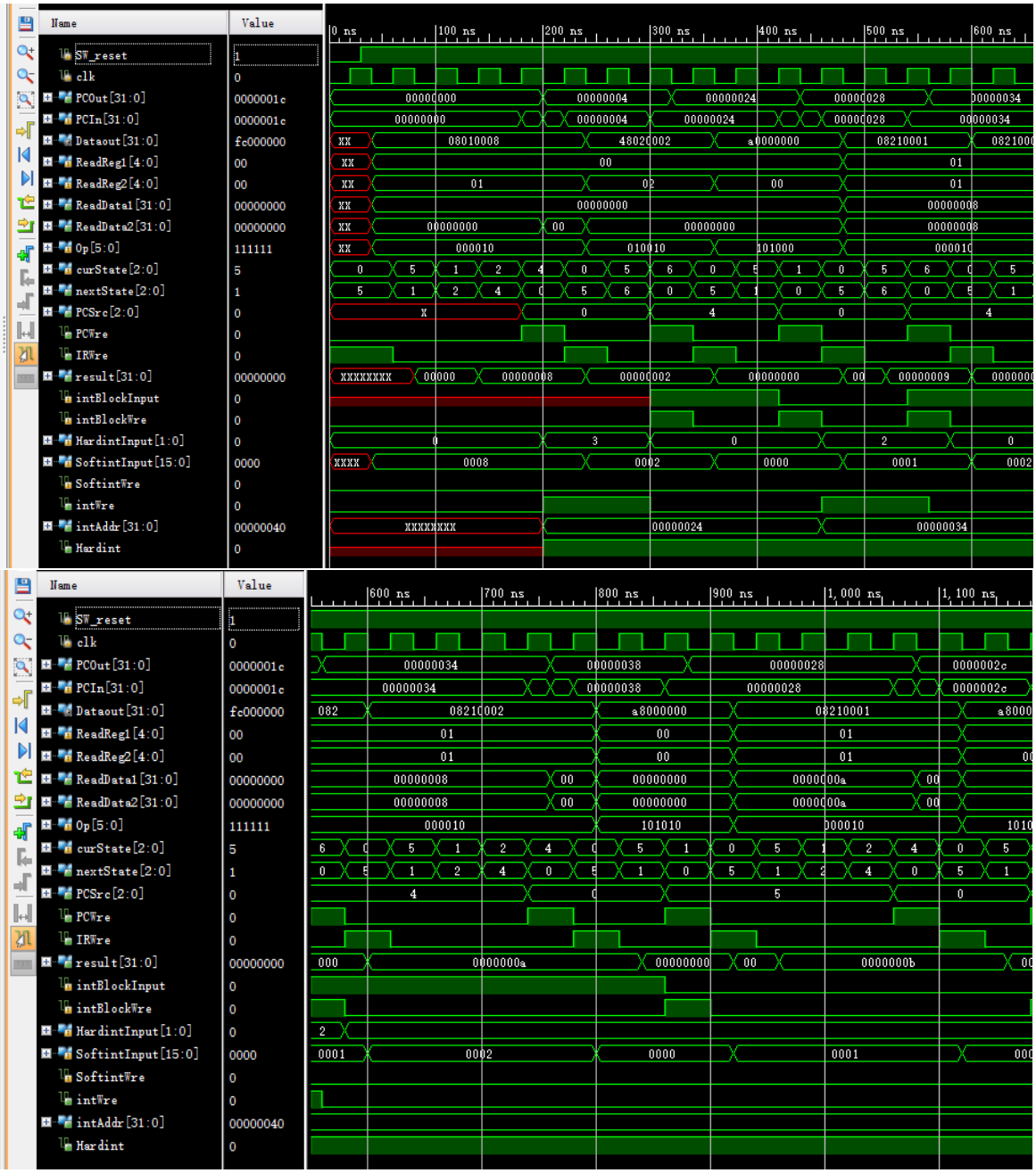
[1] SHI Jing, WANG Yi-huai, SU Yong, and SHEN Chen. Analysis of mqx interrupt mechanism and design of interrupt program frame based on arm cortex-m4. *Computer Science*, 40:43, 2013.

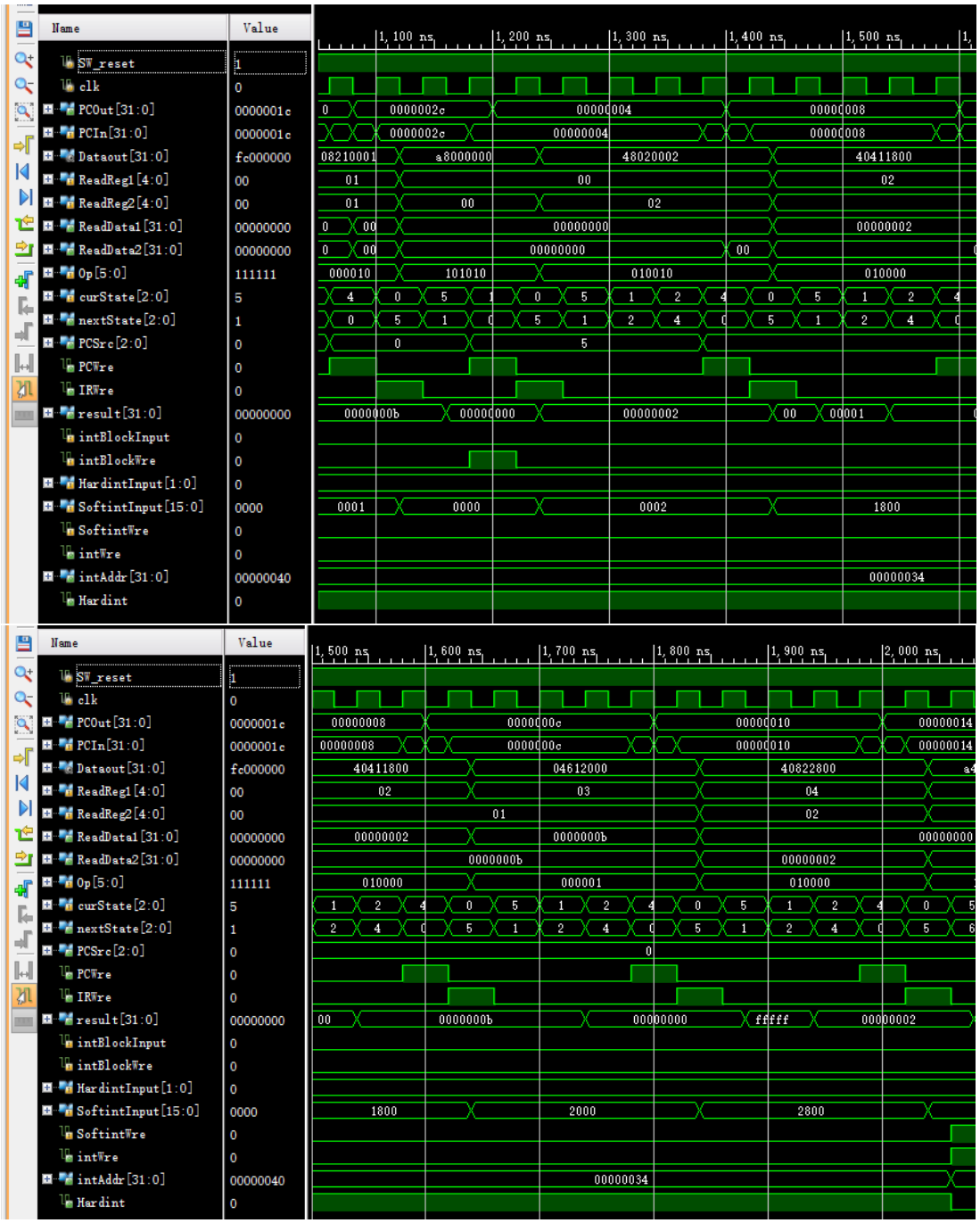
[2] David A. Patterson and John L. Hennessy. *Computer Organization and Design: The Hardware/-Software Interface (Fifth Edition)*. Elsevier (Singapore) Pte Ltd., 2014.

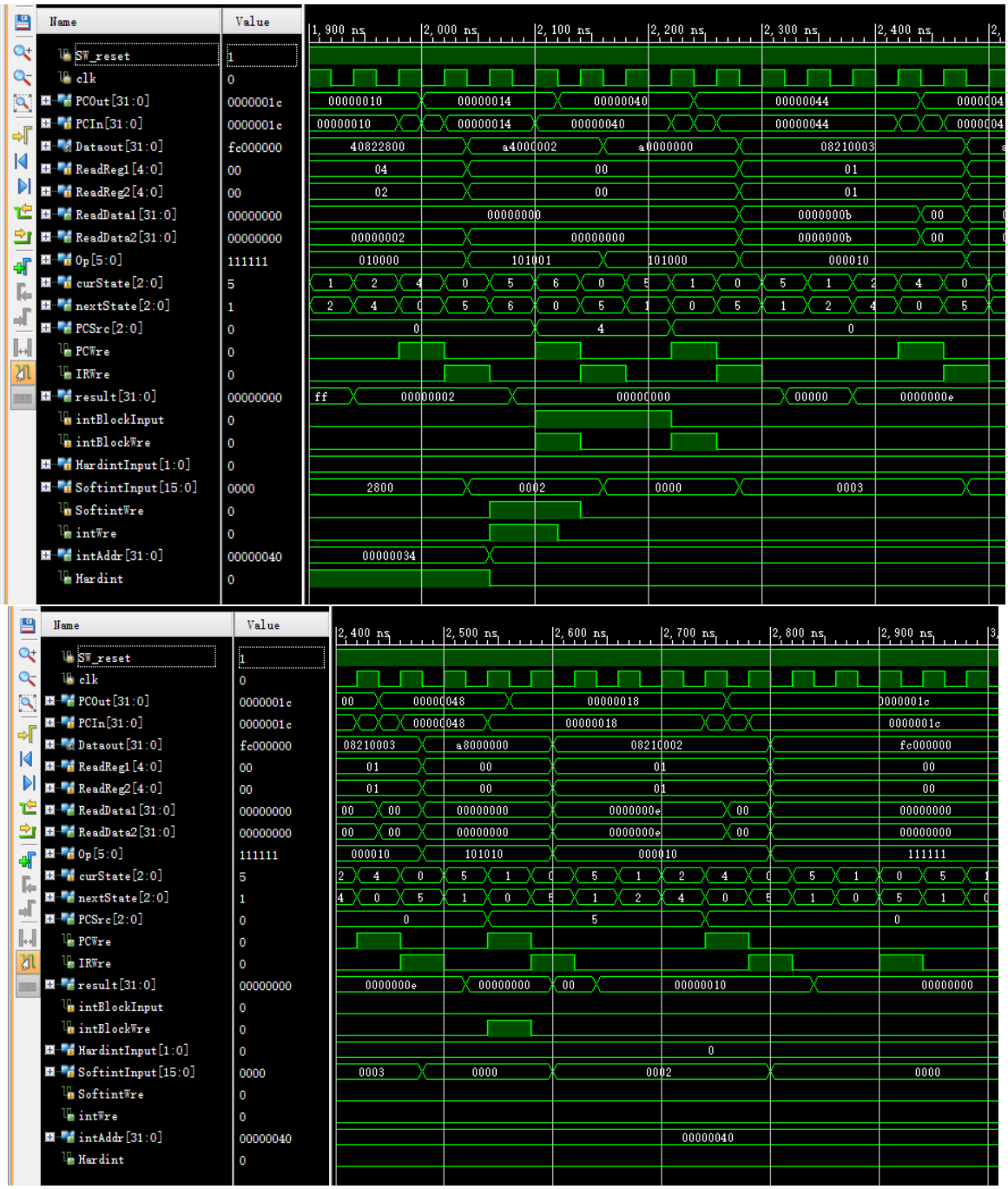
[3] 张钧良. 高校教材 计算机组成原理. 清华大学出版社 北方交通大学出版社, 2003.

Appendices

A 时序仿真波形图




Name
Value
SW_reset 1
clk 0
PCOut[31:0] 0000001c
PCIn[31:0] 0000001c
Dataout[31:0] fc000000
ReadReg1[4:0] 00
ReadReg2[4:0] 00
ReadData1[31:0] 00000000
ReadData2[31:0] 00000000
Op[5:0] 111111
curState[2:0] 5
nextState[2:0] 1
PCSrc[2:0] 0
PCWre 0
IRWre 0
result[31:0] 00000000
intBlockInput 0
intBlockWre 0
HardintInput[1:0] 0
SoftintInput[15:0] 0000
SoftintWre 0
intWre 0
intAddr[31:0] 00000040
Hardint 0
1,100 ns
1,200 ns
1,300 ns
1,400 ns
1,500 ns
1,500 ns
1,600 ns
1,700 ns
1,800 ns
1,900 ns
2,000 ns


Name
Value
SW_reset
clk
PCOut[31:0]
PCIn[31:0]
Dataout[31:0]
ReadReg1[4:0]
ReadReg2[4:0]
ReadData1[31:0]
ReadData2[31:0]
Op[5:0]
curState[2:0]
nextState[2:0]
PCSrc[2:0]
PCWre
IRWre
result[31:0]
intBlockInput
intBlockWre
HardintInput[1:0]
SoftintInput[15:0]
SoftintWre
intWre
intAddr[31:0]
Hardint
1,900 ns
2,000 ns
2,100 ns
2,200 ns
2,300 ns
2,400 ns
2,500 ns
2,600 ns
2,700 ns
2,800 ns
2,900 ns

B 中断控制器实现代码

中断控制器是使用 Verilog 实现的。

为了实现简单，中断控制器里包含了中断向量存储器。

```
1  `timescale 1ns / 1ps
2
3  module InterruptControllor(
4      input clk,
5      input intBlockInput,
6      input intBlockWre,
7      input [1:0] HardintInput,
8      input [15:0] SoftintInput,
9      input SoftintWre,
10     output reg intWre,
11     output reg [31:0] intAddr,
12     output reg Hardint
13     );
14
15     reg [7:0] rom[199:0];
16     reg valid;
17     initial begin
18         $readmemb("./interrupt_vector.txt", rom);
19         valid = 1;
20     end
21     always @ (negedge clk) begin
22         if (intBlockWre) begin
23             valid <= !intBlockInput;
24         end
25     end
26     always @ (valid or intBlockInput or intBlockWre or HardintInput or SoftintInput
           or SoftintWre) begin
27         if (!valid) begin
28             intWre = 0;
29         end else begin
30             if (HardintInput[0]) begin
31                 intAddr[31:24] = rom[0];
32                 intAddr[23:16] = rom[1];
33                 intAddr[15:8] = rom[2];
34                 intAddr[7:0] = rom[3];
35                 intWre = 1;
36                 Hardint = 1;
37             end else
38             if (HardintInput[1]) begin
39                 intAddr[31:24] = rom[4];
40                 intAddr[23:16] = rom[5];
41                 intAddr[15:8] = rom[6];
42                 intAddr[7:0] = rom[7];
43                 intWre = 1;
44                 Hardint = 1;
```

```
45              end else
46              if (SoftintWre) begin
47                  intAddr[31:24] = rom[SoftintInput * 4];
48                  intAddr[23:16] = rom[SoftintInput * 4 + 1];
49                  intAddr[15:8] = rom[SoftintInput * 4 + 2];
50                  intAddr[7:0] = rom[SoftintInput * 4 + 3];
51                  intWre = 1;
52                  Hardint = 0;
53              end else
54              begin
55                  intWre = 0;
56              end
57          end
58      end
59  endmodule
```

C 硬件堆栈实现代码

硬件堆栈是使用 Verilog 实现的。

```
`timescale 1ns / 1ps

module PCStack(
    input clk,
    input push,
    input pop,
    input [31:0] dataIn,
    output [31:0] dataOut
    );
    reg [31:0] stk[199:0];
    reg top;

    initial begin
        top = -1;
    end
    assign dataOut = (top < 0) ? 32'hz : stk[top];
    always @ (negedge clk) begin
        if (push && !pop) begin
            top = top + 1;
            stk[top] = dataIn;
        end else
        if (!push && pop) begin
            top = top - 1;
        end
    end
endmodule
```