


Jack Ren, <https://github.com/bjrjk>

计算机组成原理课设讲解



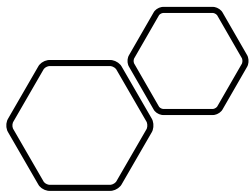
总目录

一、MIPS-Lite1 单周期处理器

二、MIPS-Lite2 多周期处理器

三、MIPS-Lite3 多周期微系统





—、

MIPS-Lite1 单周期处理器

MIPS-Lite1 单周期处理器 目录

1、重点设计
概要

2、数据通路
及控制信号

3、汇编测试
代码

4、EDA仿真
演示

MIPS-Lite1 单周期处理器 重点设计概要

与课内教学内容相同部分在此不加赘述

1、采用预处理指令定义常量

```
1 //全部宏定义头文件
2
3 `include "defs/MIPSLite1.v"
4
5 `define t 1'b1
6 `define f 1'b0
7
8 `define QBBus 31:0 // Quad Byte Bus
9 `define DBBus 15:0 // Double Byte Bus
10 `define BBus 7:0 // Byte Bus
11
12 //ALU控制信号宏定义
13 `define ALUSIG_ADD 0
14 `define ALUSIG_SUB 1
15 `define ALUSIG_OR 2
16 `define ALUSIG_LUI 3
17 `define ALUSIG_SLT 4
18
19 //译码器至控制器指令信号线对应指令下标宏定义
20 `define CTLSIG_NOP 0
21 `define CTLSIG_ADDU 1
22 `define CTLSIG_SUBU 2
23 `define CTLSIG_ORI 3
24 `define CTLSIG_LW 4
25 `define CTLSIG_SW 5
26 `define CTLSIG_BEQ 6
27 `define CTLSIG_LUI 7
28 `define CTLSIG_J 8
29 `define CTLSIG_ADDI 9
30 `define CTLSIG_ADDIU 10
31 `define CTLSIG_SLT 11
32 `define CTLSIG_JAL 12
33 `define CTLSIG_JR 13
34
35 //寄存器写目的控制信号宏定义
36 `define REGWRDSTSIG_RT 0
37 `define REGWRDSTSIG_RD 1
38 `define REGWRDSTSIG_GPR_RA 2
39
40 //位拓展器控制信号宏定义
41 `define EXTSIG_ZERO 0
42 `define EXTSIG_SIGN 1
43
44 //回写控制信号宏定义
45 `define WRBACKSIG_ALU 0
46 `define WRBACKSIG_MEM 1
47 `define WRBACKSIG_PC 2
48
49 //ALU数据源控制信号宏定义
```

```
1 //MIPS-Lite1 指令集
2
3 `define OPCODE_SPECIAL 6'b000000
4 `define OPCODE_ORI 6'b001101
5 `define OPCODE_LW 6'b100011
6 `define OPCODE_SW 6'b101011
7 `define OPCODE_BEQ 6'b000100
8 `define OPCODE_LUI 6'b001111
9 `define OPCODE_J 6'b000010
10 `define OPCODE_ADDI 6'b001000
11 `define OPCODE_ADDIU 6'b001001
12 `define OPCODE_JAL 6'b000011
13
14 `define FUNCT_ADDU 6'b100001
15 `define FUNCT_SUBU 6'b100011
16 `define FUNCT_SLT 6'b101010
17 `define FUNCT_JR 6'b001000
```

使用define语句定义常量：
使代码更易读，也容易修改。

2、控制器与译码器分离

译码器

```
1 //指令译码单元 — 译码器 Decoder
2
3 `include "defines.v"
4
5 module Decoder(
6     input [`QBBus] Inst,
7     output wire [`QBBus] DecInstBus,
8     output wire [4:0] rs,rt,rd,shamt,
9     output wire [`DBBus] imm,
10    output wire [25:0] tgtAddr
11 );
12
13    wire [5:0] opcode,funcnt;
14    reg [5:0] DecInstIndex;
15    assign opcode=Inst[31:26];
16    assign funcnt=Inst[5:0];
17    assign rs=Inst[25:21];
18    assign rt=Inst[20:16];
19    assign rd=Inst[15:11];
20    assign shamt=Inst[10:6];
21    assign imm=Inst[15:0];
22    assign tgtAddr=Inst[25:0];
23    assign DecInstBus=32'd1<<DecInstIndex;
24
25    always@ (*) begin
26        case(opcode)
27            `OPCODE_SPECIAL: begin
28                case(funcnt)
29                    `FUNCT_ADDU: DecInstIndex=(shamt==0)?(`CTLSIG_ADDU):(`CTLSIG_NOP);
30                    `FUNCT_SUBU: DecInstIndex=(shamt==0)?(`CTLSIG_SUBU):(`CTLSIG_NOP);
31                    `FUNCT_SLT: DecInstIndex=(shamt==0)?(`CTLSIG_SLT):(`CTLSIG_NOP);
32                    `FUNCT_JR: DecInstIndex=(rt==0&&rd==0)?(`CTLSIG_JR):(`CTLSIG_NOP);
33                    default: DecInstIndex=`CTLSIG_NOP;
34                endcase
35            end
36            `OPCODE_ORI: DecInstIndex=`CTLSIG_ORI;
37            `OPCODE_LW: DecInstIndex=`CTLSIG_LW;
38            `OPCODE_SW: DecInstIndex=`CTLSIG_SW;
39            `OPCODE_BEQ: DecInstIndex=`CTLSIG_BEQ;
40            `OPCODE_LUI: DecInstIndex=(rs==0)?(`CTLSIG_LUI):(`CTLSIG_NOP);
41            `OPCODE_J: DecInstIndex=`CTLSIG_J;
42            `OPCODE_ADDI: DecInstIndex=`CTLSIG_ADDI;
43            `OPCODE_ADDIU: DecInstIndex=`CTLSIG_ADDIU;
44            `OPCODE_JAL: DecInstIndex=`CTLSIG_JAL;
45            default: DecInstIndex=`CTLSIG_NOP;
46        endcase
47    end
48
49 endmodule
```

控制器

```
1 //指令译码单元 — 控制器 Controller
2
3 `include "defines.v"
4
5 module Controller(
6     input [`QBBus] DecInstBus,
7     output reg RegWrEn,RegOFWrEn,MemWrEn, //寄存器写使能, 寄存器溢出写使能, 内存写使能
8     output reg [3:0] ALUCtl, //ALU控制信号
9     output reg [1:0] RegWrDstCtl,WrBackCtl, //寄存器写目标控制信号, 回写控制信号
10    output reg ALUSrcCtl, ExtCtl //ALU数据源控制信号, 位拓展器控制信号
11 );
12
13    always@ (*) begin
14        RegWrEn=`t;
15        RegOFWrEn=`f;
16        MemWrEn=`f;
17        ALUCtl=`ALUSIG_ADD;
18        RegWrDstCtl=`REGWRDSTSIG_RT;
19        WrBackCtl=`WRBACKSIG_ALU;
20        ALUSrcCtl=`ALUSRCSIG_EXT;
21        ExtCtl=`EXTSIG_SIGN;
22        if(DecInstBus[`CTLSIG_ADDU]) begin
23            RegWrDstCtl=`REGWRDSTSIG_RD;
24            ALUSrcCtl=`ALUSRCSIG_GPR;
25        end else if(DecInstBus[`CTLSIG_SUBU]) begin
26            ALUCtl=`ALUSIG_SUB;
27            RegWrDstCtl=`REGWRDSTSIG_RD;
28            ALUSrcCtl=`ALUSRCSIG_GPR;
29        end else if(DecInstBus[`CTLSIG_ORI]) begin
30            ALUCtl=`ALUSIG_OR;
31            ExtCtl=`EXTSIG_ZERO;
32        end else if(DecInstBus[`CTLSIG_LW]) begin
33            WrBackCtl=`WRBACKSIG_MEM;
34        end else if(DecInstBus[`CTLSIG_SW]) begin
35            RegWrEn=`f;
36            MemWrEn=`t;
37        end else if(DecInstBus[`CTLSIG_BEQ]) begin
38            RegWrEn=`f;
39            ALUCtl=`ALUSIG_SUB;
40            ALUSrcCtl=`ALUSRCSIG_GPR;
41        end else if(DecInstBus[`CTLSIG_LUI]) begin
42            ALUCtl=`ALUSIG_LUI;
43        end else if(DecInstBus[`CTLSIG_J]) begin
44            RegWrEn=`f;
45        end else if(DecInstBus[`CTLSIG_ADDI]) begin
46            RegOFWrEn=`t;
47        end else if(DecInstBus[`CTLSIG_ADDIU]) begin
48
49        end else if(DecInstBus[`CTLSIG_SLT]) begin
```

译码器通过指令各字段判断指令类型；
控制器输出各部件控制信号；
两部件通过 DecInstBus 传递指令类型。

便于后续的维护、修改和拓展。

3、实现addi指令溢出写回

```
1 //读寄存器单元 — 通用寄存器组 General Purpose Register
2
3 `include "defines.v" 溢出写使能
4                        溢出标志位
5 module GPR(
6     input clk,WrEn,OFWrEn,OFFlag,
7     input [4:0] RdAddr1,RdAddr2,WrAddr,
8     input [`QBBus] WrData,
9     output reg [`QBBus] RdData1,RdData2
10 );
11
12 reg [`QBBus] regArr [`QBBus];
13
14 initial regArr[0]=0;
15
16 always @(*) begin
17     if(RdAddr1!=0)RdData1=regArr[RdAddr1];
18     else RdData1=0;
19     if(RdAddr2!=0)RdData2=regArr[RdAddr2];
20     else RdData2=0;
21 end
22
23 always @(posedge clk) begin
24     if(WrEn&&WrAddr!=0)regArr[WrAddr]<=WrData;
25     if(OFWrEn)regArr[30][0]<=OFFlag;
26 end
27
28 endmodule
29
```

在GPR内加入OFWrEn和OFFlag标志；
分别代表溢出写使能和溢出标志位。
当溢出写使能时，将标志位写入30号寄存器第0位。

4、ALU判断溢出、SLT有符号数判大小

```
1 //指令执行单元 — 算术逻辑单元 Arithmetic and Logic Unit
2
3 `include "defines.v"
4
5 module ALU(
6     input [3:0] ALUctl,
7     input [`QBBus] A,B,
8     output reg [`QBBus] C,
9     output OF,
10    output wire zero
11 );
12
13 assign OF= (A[31]==B[31])&&(C[31]!=A[31]);
14 assign zero= (C==0);
15
16 always@ (*) begin
17     case(ALUctl)
18         `ALUSIG_ADD:C=A+B;
19         `ALUSIG_SUB:C=A-B;
20         `ALUSIG_OR:C=A|B;
21         `ALUSIG_LUI:C={B[15:0],16'd0};
22         `ALUSIG_SLT:C= (A[31]==0&&B[31]==0) ? (A<B) : //都为正数，直接比
23             (A[31]==0&&B[31]==1) ? 0 : //A正B负，A肯定大于B
24             (A[31]==1&&B[31]==0) ? 1 : //A负B正，A肯定小于B
25             // (A[31]==1&&B[31]==1)，都为负数，化成绝对值后再比绝对值大的
26             ((~A)+1) > ((~B)+1)
27     ;
28     default:C=A+B;
29 endcase
30 end
31
32 endmodule
33
```

采用的实现方式：

当两个同号数相加，若所得结果符号与两数符号不同，则表明溢出。

MIPS-Lite1 单周期处理器 数据通路及控制信号

单周期处理器控制信号简介

Jack Ren <https://github.com/bjrik>

信号名	方向	描述
RegWrEn(RegWriteEnable)	O	寄存器写使能
RegOFWrEn(RegOverflowWriteEnable)	O	寄存器溢出位写使能
MemWrEn(MemoryWriteEnable)	O	内存写使能
ALUCtl(ALUControl)	O	ALU控制信号 ALUSIG_ADD: 加 ALUSIG_SUB: 减 ALUSIG_OR: 或 ALUSIG_LUI: 置高位 ALUSIG_SLT: 小于比较
RegWrDstCtl(RegWriteDestinationControl)	O	寄存器写目的控制信号 REGWRDSTSIG_RT: 寄存器rt REGWRDSTSIG_RD: 寄存器rd REGWRDSTSIG_GPR_RA: 寄存器\$ra
WrBackCtl(WriteBackControl)	O	回写控制信号 WRBACKSIG_ALU: 从ALU取结果回写 WRBACKSIG_MEM: 从内存取结果回写 WRBACKSIG_PC: 从PC下地址逻辑取PC+4回写
ALUSrcCtl(ALUSourceControl)	O	ALU数据源控制信号 ALUSRCSIG_GPR: 从寄存器堆读2端口取数据 ALUSRCSIG_EXT: 从位拓展器取数据
ExtCtl(ExtendControl)	O	位拓展器控制信号 EXTSIG_ZERO: 零扩展 EXTSIG_SIGN: 符号扩展

单周期处理器控制信号表

指令\信号	RegWrEn	RegOFWrEn	MemWrEn	ALUCtl[3:0]	RegWrDstCtl[1:0]	WrBackCtl[1:0]	ALUSrcCtl	ExtCtl
DEFAULT	1	0	0	ALUSIG_ADD	REGWRDSTSIG_RT	WRBACKSIG_ALU	ALUSRCSIG_EXT	EXTSIG_SIGN
ADDU	1	0	0	ALUSIG_ADD	REGWRDSTSIG_RD	WRBACKSIG_ALU	ALUSRCSIG_GPR	
SUBU	1	0	0	ALUSIG_SUB	REGWRDSTSIG_RD	WRBACKSIG_ALU	ALUSRCSIG_GPR	
ORI	1	0	0	ALUSIG_OR	REGWRDSTSIG_RT	WRBACKSIG_ALU	ALUSRCSIG_EXT	EXTSIG_ZERO
LW	1	0	0	ALUSIG_ADD	REGWRDSTSIG_RT	WRBACKSIG_MEM	ALUSRCSIG_EXT	EXTSIG_SIGN
SW	0	0	1	ALUSIG_ADD			ALUSRCSIG_EXT	EXTSIG_SIGN
BEQ	0	0	0	ALUSIG_SUB			ALUSRCSIG_GPR	
LUI	1	0	0	ALUSIG_LUI	REGWRDSTSIG_RT	WRBACKSIG_ALU	ALUSRCSIG_EXT	
J	0	0	0					
ADDI	1	1	0	ALUSIG_ADD	REGWRDSTSIG_RT	WRBACKSIG_ALU	ALUSRCSIG_EXT	EXTSIG_SIGN
ADDIU	1	0	0	ALUSIG_ADD	REGWRDSTSIG_RT	WRBACKSIG_ALU	ALUSRCSIG_EXT	EXTSIG_SIGN
SLT	1	0	0	ALUSIG_SLT	REGWRDSTSIG_RD	WRBACKSIG_ALU	ALUSRCSIG_GPR	
JAL	1	0	0		REGWRDSTSIG_GPR_RA	WRBACKSIG_PC		
JR	0	0	0					
NOP	0	0	0					

```
2
3 `include "defines.v"
4
5 module Controller(
6     input [`QBBus] DecInstBus,
7     output reg RegWrEn, RegOFWrEn, MemWrEn, //寄存器写使能, 寄存器溢出写使能, 内
8     output reg [3:0] ALUCtl, //ALU控制信号
9     output reg [1:0] RegWrDstCtl, WrBackCtl, //寄存器写目标控制信号, 回写控制信
10    output reg ALUSrcCtl, ExtCtl //ALU数据源控制信号, 位拓展器控制信号
11);
12
13 always@ (*) begin
14     RegWrEn=`t;
15     RegOFWrEn=`f;
16     MemWrEn=`f;
17     ALUCtl=`ALUSIG_ADD;
18     RegWrDstCtl=`REGWRDSTSIG_RT;
19     WrBackCtl=`WRBACKSIG_ALU;
20     ALUSrcCtl=`ALUSRCSIG_EXT;
21     ExtCtl=`EXTSIG_SIGN;
22     if(DecInstBus[`CTLSIG_ADDU]) begin
23         RegWrDstCtl=`REGWRDSTSIG_RD;
24         ALUSrcCtl=`ALUSRCSIG_GPR;
25     end else if(DecInstBus[`CTLSIG_SUBU]) begin
26         ALUCtl=`ALUSIG_SUB;
27         RegWrDstCtl=`REGWRDSTSIG_RD;
28         ALUSrcCtl=`ALUSRCSIG_GPR;
29     end else if(DecInstBus[`CTLSIG_ORI]) begin
30         ALUCtl=`ALUSIG_OR;
31         ExtCtl=`EXTSIG_ZERO;
32     end else if(DecInstBus[`CTLSIG_LW]) begin
33         WrBackCtl=`WRBACKSIG_MEM;
34     end else if(DecInstBus[`CTLSIG_SW]) begin
35         RegWrEn=`f;
36         MemWrEn=`t;
37     end else if(DecInstBus[`CTLSIG_BEQ]) begin
38         RegWrEn=`f;
39         ALUCtl=`ALUSIG_SUB;
40         ALUSrcCtl=`ALUSRCSIG_GPR;
41     end else if(DecInstBus[`CTLSIG_LUI]) begin
42         ALUCtl=`ALUSIG_LUI;
43     end else if(DecInstBus[`CTLSIG_J]) begin
44         RegWrEn=`f;
45     end else if(DecInstBus[`CTLSIG_ADDI]) begin
46         RegOFWrEn=`t;
47     end else if(DecInstBus[`CTLSIG_ADDIU]) begin
48
49     end else if(DecInstBus[`CTLSIG_SLT]) begin
```

为什么表格中要有一行 DEFAULT指令?

- DEFAULT指令实际并不存在, 它是为了简化Controller的Verilog代码而出现的
- 在always组合逻辑块中首先利用DEFAULT指令的各控制信值号对各控制信号线进行赋值
- 在各具体指令的判断语句中, 只需修改与DEFAULT指令不同的控制信号值即可
- 可以有效地减少代码量, 便于拓展

MIPS-Lite1 单周期处理器 汇编测试代码

(该代码同时用于测试多周期处理器)

MIPS-Lite1 单周期处理器 汇编测试代码 (1)

MIPS源程序:

```
ori $0,0x1403 # Test whether the value in $zero can be changed
lui $at,0x1804
ori $at,0x1403 # Write Student ID to $at
addu $sp,$0,$0 # Clear stack pointer
addu $s1,$0,$0
addiu $s1,$s1,1 # Change $s1 to 1
addi $a0,$0,5 # Pass parameter to FUNC_ADD
jal FUNC_ADD # Execute funcAdd
addi $a0,$0,5 # Pass parameter to FUNC_FIB
jal FUNC_FIB # Execute funcFib
jal FUNC_FUN_ADD # Execute funcFunAdd
lui $t7,0x7fff
ori $t7,0xffff # Set $t7 to 0x7fffffff
addi $t7,$t7,1 # Get $t7 Overflow
SELFLOOP:
j SELFLOOP
```

大致等价的伪C代码:

```
$0=0; //给0号寄存器赋值, 不能执行成功
```

```
$at=0x18041403; //将$at写入学号
```

```
$sp=0; //栈指针清零
```

```
$s1=1; //作为常数1的寄存器, 在之后的过程中调用
```

```
funcAdd(5); // int funcAdd(int x)
```

```
funcFib(5); // int funcFib(int n)
```

```
funcFunAdd(); // void funcFunADD()
```

```
$t7=0x7fffffff;
```

```
$t7++; //测试addi的溢出
```

```
while(1); //死循环
```

MIPS-Lite1 单周期处理器 汇编测试代码 (2)

```
FUNC_ADD:
beq $a0,$0,FUNC_ADD_TRIVIAL_RET # When x==0, jump to
trivial branch
sw $a0,0($sp) # Save paramater x to stack
sw $ra,4($sp) # Save return address to stack
subu $a0,$a0,$s1 # Decrease x
addi $sp,$sp,8 # Add $sp
jal FUNC_ADD
addi $sp,$sp,-8 # Minus $sp
lw $ra,4($sp) # Load return address to register
lw $a0,0($sp) # Load paramater x to register
addu $v0,$v0,$s1 # Increase return value
j FUNC_ADD_RET # Return
FUNC_ADD_TRIVIAL_RET:
addu $v0,$0,$0
FUNC_ADD_RET:
jr $ra
```

```
int funcAdd(int x){ //输入x, 返回值也是x
    if(x==0)return 0;
    else return funcAdd(x-1)+1;
}
```


MIPS-Lite1 单周期处理器 汇编测试代码 (3)

FUNC_FIB:

addu \$t0,\$0,\$0 # x=0

addu \$t1,\$0,\$0 # y=0

addiu \$t2,\$0,1 # z=1

addu \$t8,\$0,\$0 # i=0

FUNC_FIB_LOOP:

slt \$t9,\$t8,\$a0 # i<n?

beq \$t9,\$0,FUNC_FIB_RET # i>=n return

addu \$t0,\$0,\$t1 # x=y

addu \$t1,\$0,\$t2 # y=z

addu \$t2,\$t0,\$t1 # z=x+y

addu \$t8,\$t8,\$s1 # i++

j FUNC_FIB_LOOP

FUNC_FIB_RET:

addu \$v0,\$0,\$t2

jr \$ra

int funcFib(int n){ //计算第n个斐波那契数

int x=0,y=0,z=1;

for(int i=0;i<n;i++){

x=y;

y=z;

z=x+y;

}

return z;

}

MIPS-Lite1 单周期处理器 汇编测试代码 (4)

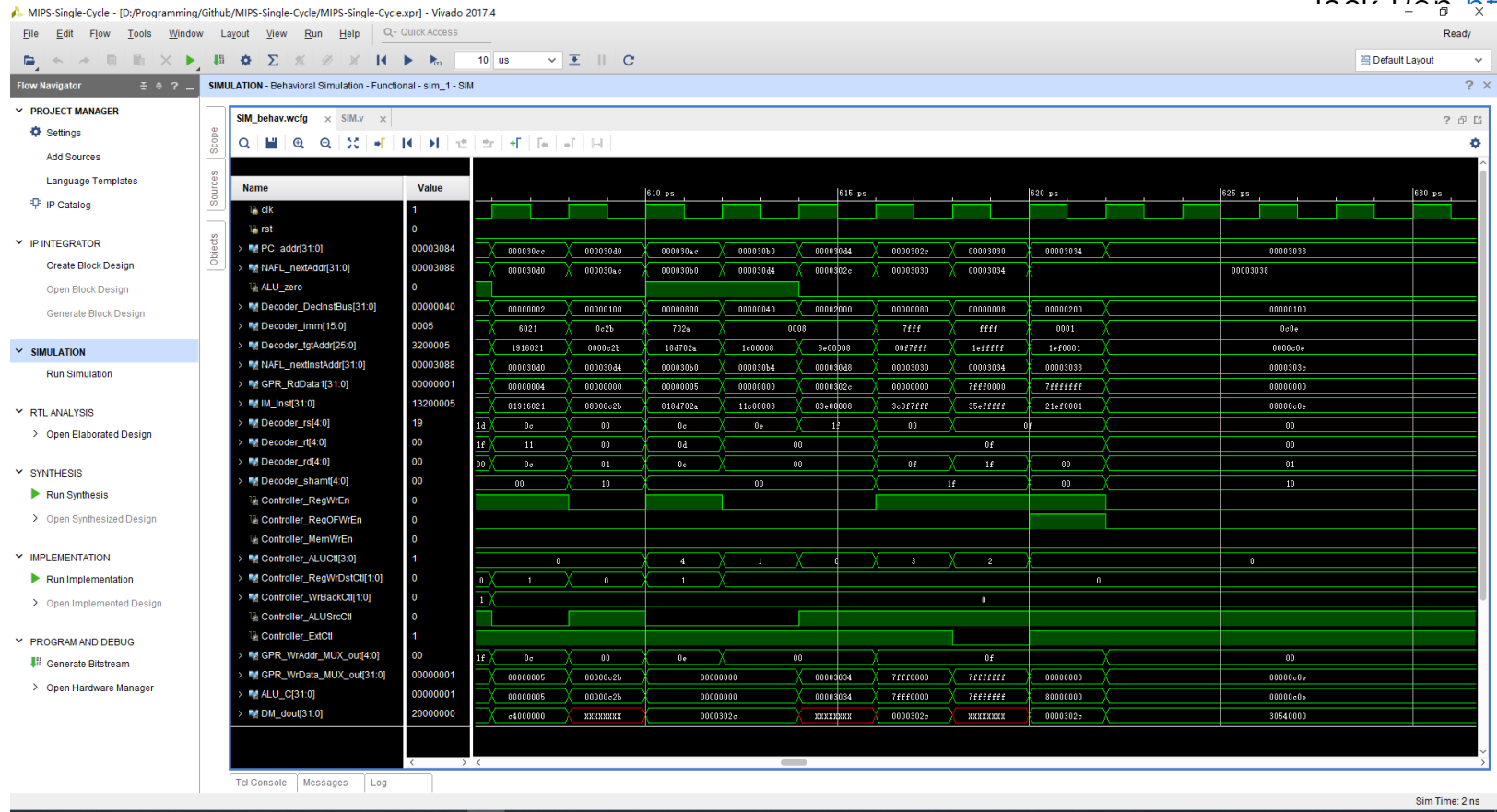
```
FUNC_FUN_ADD:
addu $t4,$0,$0 # i=0
addi $t5,$0,5 # n=5
FUNC_FUN_ADD_LOOP:
slt $t6,$t4,$t5 # i<n?
beq $t6,$0,FUNC_FUN_ADD_RET # i>=n return
sw $ra,0($sp) # Save return address to stack
addi $sp,$sp,4 # Add $sp
addu $a0,$0,$t4 # Pass parameter i
jal FUNC_ADD # Call funcAdd
addi $sp,$sp,-4 # Minus $sp
lw $ra,0($sp) # Load return address to register
addu $t4,$t4,$s1 # i++
j FUNC_FUN_ADD_LOOP
FUNC_FUN_ADD_RET:
jr $ra
```

```
void funcFunAdd(){
    int i=0;
    int n=5;

    for(i=0;i<n;i++)

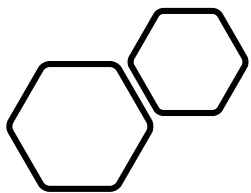
        funcAdd(i);

}
```



MIPS-Lite1 单周期处理器EDA 仿真演示

进行现场仿真演示



二、

MIPS-Lite2 多周期处理器

MIPS-Lite2 多周期处理器 目录

1、五周期处理器设计思想

2、数据通路及控制信号

3、重难点设计概要

4、汇编测试代码

5、EDA仿真演示

MIPS-Lite2 多周期处理器 五周期处理器设计思想

有问题随时询问

为什么要制作五周期处理器而不采用更多周期?

- 更好的重用各周期的代码，避免过多复杂冗余代码
- 在短时间内，更加便于扩展新指令

五周期处理器有何难点？

- 思维难度大，相对多周期处理器来说更加难以梳理清晰：
- 1、如何将这么多指令塞进 **五个周期** 里？
- 2、为了严格符合五周期处理器的设计思想（“取值、译码、执行、访存、回写”），跳转指令的实现要进行额外工作
- 在数据通路中，为跳转指令特别建立“**延迟寄存器**”，在“**重难点设计概要**”中详细介绍

单周期、多周期和流水的对比？

- 单周期处理器——一个时钟周期内干完一条指令的所有工作，因此一个时钟周期的时间必须长到足够覆盖整个关键路径的信号传输时间
- 多周期处理器——将数据通路中插入若干寄存器，从而缩小关键路径的长度，因此可减小一个时钟周期的时间长度；同时可将一些指令本不需要的执行步骤略过；但是整个处理器中同一时间只能执行一条指令
- 流水线处理器——每段关键路径的寄存器中包含全部的控制信息和数据信息；整个处理器中可以同时执行多条指令的不同阶段

*En信号和*Sel/*Op信号有什么区别？ (1)

LB控制 信号表	取指	译码	执行	访存	写回
PCWr	1	0	0	0	0
IRWr	1	0	0	0	0
WDSel	X	01	01	01	01
GPRSel	X	00	00	00	00
ExtOp	X	1	1	1	1
GPRWr	0	0	0	0	1
BSel	X	1	1	1	1
ALUOp	X	ADD	ADD	ADD	ADD
NPCOp	PC+4	X	X	X	X
DMWr	0	0	0	0	0
lb	X	1	1	1	1

将*En信号称作使能信号。其他的*Sel/*Op信号称作选择信号。

观察控制信号表，使能类信号和其他选择类信号有什么区别？

*En信号和*Sel/*Op信号有什么区别？ (2)

- 整个处理器中同一时间只能执行一条指令决定了
- **选择信号在整条指令的执行过程中都不会变**
- 多周期的特性决定其
- **使能信号随着状态机的状态转移而发生变化，但与具体指令类型无关（与大类有关）**

如何实现五周期CPU的控制信号？

- 选择信号：在一条指令的多周期执行过程中始终保持不变，且仅与指令的类型有关。因此直接利用组合逻辑电路判断当前指令类型并输出即可，**与周期无关**
- 本处理器设计中选择信号以*Ctl命名
- 使能信号：**与当前周期本身**（是取指、译码、执行、访存、回写）**有关**；与具体指令小类无关，**与是否为跳转指令相关**
- 本处理器设计中使能信号以*En命名

MIPS-Lite2 多周期处理器 数据通路及控制信号

与课内教学内容相同部分在此不加赘述

多周期处理器 控制信号简介

多周期处理器关键控制信号->
在重难点概要设计中讲述

信号名	方向	描述
PCEn	○	PC写使能
IMEn	○	IM取新指令使能
RegWrEn(RegWriteEnable)	○	寄存器写使能
RegOFWrEn(RegOverFlowWriteEnable)	○	寄存器溢出位写使能
MemWrEn(MemoryWriteEnable)	○	内存写使能
ALUCtl(ALUControl)	○	ALU控制信号 ALUSIG_ADD: 加 ALUSIG_SUB: 减 ALUSIG_OR: 或 ALUSIG_LUI: 置高位 ALUSIG_SLT: 小于比较
RegWrDstCtl(RegWriteDestinationControl)	○	寄存器写目的控制信号 REGWRDSTSIG_RT: 寄存器rt REGWRDSTSIG_RD: 寄存器rd REGWRDSTSIG_GPR_RA: 寄存器\$ra
WrBackCtl(WriteBackControl)	○	回写控制信号 WRBACKSIG_ALU: 从ALU取结果回写 WRBACKSIG_MEM: 从内存取结果回写 WRBACKSIG_PC: 从PC下地址逻辑取PC+4回写
ALUSrcCtl(ALUSourceControl)	○	ALU数据源控制信号 ALUSRCSIG_GPR: 从寄存器堆读2端口取数据 ALUSRCSIG_EXT: 从位拓展器取数据
ExtCtl(ExtendControl)	○	位拓展器控制信号 EXTSIG_ZERO: 零扩展 EXTSIG_SIGN: 符号扩展
DataSizeCtl(DataSizeControl)	○	数据大小控制信号 DATASIZESIG_W: 传输字 DATASIZESIG_B: 传输字节
NAFLCtl(NextAddressFormulationLogicControl)	○	下地址逻辑控制信号 NAFLSIG_PCNext: 取PC+4 NAFLSIG_BEQ: BEQ指令 NAFLSIG_J: J指令 NAFLSIG_JAL: JAL指令 NAFLSIG_JR: JR指令

多周期处理器选择信号表

指令\信号	ALUCtl[3:0]	RegWrDstCtl[1:0]	WrBackCtl[1:0]	ALUSrcCtl	ExtCtl	DataSizeCtl	NAFLCtl
DEFAULT	ALUSIG_ADD	REGWRDSTSIG_RT	WRBACKSIG_ALU	ALUSRCSIG_EXT	EXTSIG_SIGN	DATASIZESIG_W	NAFLSIG_PCNext
ADDU	ALUSIG_ADD	REGWRDSTSIG_RD	WRBACKSIG_ALU	ALUSRCSIG_GPR		DATASIZESIG_W	NAFLSIG_PCNext
SUBU	ALUSIG_SUB	REGWRDSTSIG_RD	WRBACKSIG_ALU	ALUSRCSIG_GPR		DATASIZESIG_W	NAFLSIG_PCNext
ORI	ALUSIG_OR	REGWRDSTSIG_RT	WRBACKSIG_ALU	ALUSRCSIG_EXT	EXTSIG_ZERO	DATASIZESIG_W	NAFLSIG_PCNext
LW	ALUSIG_ADD	REGWRDSTSIG_RT	WRBACKSIG_MEM	ALUSRCSIG_EXT	EXTSIG_SIGN	DATASIZESIG_W	NAFLSIG_PCNext
SW	ALUSIG_ADD			ALUSRCSIG_EXT	EXTSIG_SIGN	DATASIZESIG_W	NAFLSIG_PCNext
BEQ	ALUSIG_SUB			ALUSRCSIG_GPR		DATASIZESIG_W	NAFLSIG_BEQ
LUI	ALUSIG_LUI	REGWRDSTSIG_RT	WRBACKSIG_ALU	ALUSRCSIG_EXT		DATASIZESIG_W	NAFLSIG_PCNext
J							NAFLSIG_J
ADDI	ALUSIG_ADD	REGWRDSTSIG_RT	WRBACKSIG_ALU	ALUSRCSIG_EXT	EXTSIG_SIGN	DATASIZESIG_W	NAFLSIG_PCNext
ADDIU	ALUSIG_ADD	REGWRDSTSIG_RT	WRBACKSIG_ALU	ALUSRCSIG_EXT	EXTSIG_SIGN	DATASIZESIG_W	NAFLSIG_PCNext
SLT	ALUSIG_SLT	REGWRDSTSIG_RD	WRBACKSIG_ALU	ALUSRCSIG_GPR		DATASIZESIG_W	NAFLSIG_PCNext
JAL		REGWRDSTSIG_GPR_RA	WRBACKSIG_PC				NAFLSIG_JAL
JR						DATASIZESIG_W	NAFLSIG_JR
LB	ALUSIG_ADD	REGWRDSTSIG_RT	WRBACKSIG_MEM	ALUSRCSIG_EXT	EXTSIG_SIGN	DATASIZESIG_B	NAFLSIG_PCNext
SB	ALUSIG_ADD			ALUSRCSIG_EXT	EXTSIG_SIGN	DATASIZESIG_B	NAFLSIG_PCNext
NOP							NAFLSIG_PCNext

多周期处理器使能信号表

指令\信号	RegWrEn	RegOFWrEn	MemWrEn
DEFAULT	1	0	0
ADDU	1	0	0
SUBU	1	0	0
ORI	1	0	0
LW	1	0	0
SW	0	0	1
BEQ	0	0	0
LUI	1	0	0
J	0	0	0
ADDI	1	1	0
ADDIU	1	0	0
SLT	1	0	0
JAL	1	0	0
JR	0	0	0
LB	1	0	0
SB	0	0	1
NOP	0	0	0

多周期处理器执行阶段表

指令\信号	取指IF	译码/读寄存器DCD/RF	执行EXE	访存MEM	回写WB
ADDU	√	√	√		√
SUBU	√	√	√		√
ORI	√	√	√		√
LW	√	√	√	√	√
SW	√	√	√	√	
BEQ	√	√	√		
LUI	√	√	√		√
J	√	√	√		
ADDI	√	√	√		√
ADDIU	√	√	√		√
SLT	√	√	√		√
JAL	√	√	√		√
JR	√	√	√		
LB	√	√	√	√	√
SB	√	√	√	√	
NOP	√	√			

JAL指令需要对PC内容延迟写回
难点概要部分详细讲解

MIPS-Lite2 多周期处理器 重难点设计概要

1、NAFL下地址形成逻辑的设计

```
1 //指令形成单元 — 下地址形成逻辑 Next Address Formulation Logic
2
3 `include "defines.v"
4
5 module NAFL(
6     input [`QBBus] addr,
7     output reg [`QBBus] nextAddr,
8     input [2:0] NAFLCtl,
9     input beqZero,
10    input [`DBBus] beqShift, // beq指令, 16比特左移两位后变18比特再符号拓展加到PC
11    input [25:0] jPadding, // j和jal指令, 26比特左移两位后变28比特置PC低位
12    input [`QBBus] jrAddr, // jr指令从$ra直接读入的32位地址
13    output [`QBBus] nextInstAddr
14);
15
16 assign nextInstAddr = addr+4;
17
18 /*
19 在取指阶段, 产生的下地址统一为PC+4。
20 在执行阶段, 会产生跳转指令。此时PC的值已由PC变为PC+4, 所以只需做增补量。
21 这点与单周期CPU非常不同, 需要特别注意!
22 */
23
24 always@ (*) begin
25     if(NAFLCtl==`NAFLSIG_BEQ&&beqZero)nextAddr=addr+{{14{beqShift[15]}},beqShift,2'b00};
26     else if(NAFLCtl==`NAFLSIG_BEQ)nextAddr=addr;
27     else if(NAFLCtl==`NAFLSIG_J||NAFLCtl==`NAFLSIG_JAL)nextAddr={addr[31:28],jPadding,2'b00};
28     else if(NAFLCtl==`NAFLSIG_JR)nextAddr=jrAddr;
29     else nextAddr=nextInstAddr;
30 end
31
32 endmodule
33
```

明确PC的定义：在多周期处理器中一定要指向下条指令的地址，否则就没有办法做跳转指令了。

根据NAFLCtl控制信号进行操作：

Beq成立：相对当前地址加偏移量

Beq不成立：什么都不做

对于Beq指令，+4操作已经在译码阶段上升沿完成，故不需要考虑偏移量。

这个是看起来别扭的“PC相对寻址为什么要+4”的内在逻辑。

J型指令：直接PC获得传过来的立即数即可
否则就是正常指令：PC+4。

2、控制器 Controller详解 (1)

```
1 //指令译码单元 — 控制器 Controller
2
3 `include "defines.v"
4
5 module Controller(
6     input clk,rst,
7     input [`Q8Bus] DecInstBus,
8     output wire PCEn,IMEn,RegWrEn,RegOFWrEn,MemWrEn, //PC写使能, IM取新指令使能, 寄存器写使能, 寄存器溢出写使能, 内存写使能
9     output reg [3:0] ALUCtl, //ALU控制信号
10    output reg [1:0] RegWrDstCtl,WrBackCtl, //寄存器写目标控制信号, 回写控制信号
11    output reg ALUSrcCtl, ExtCtl, DataSizeCtl, //ALU数据源控制信号, 位拓展器控制信号, 数据大小控制信号
12    output reg [2:0] NAFLCtl=`NAFLSIG_PCNext //NAFL下地址逻辑控制信号
13);
14
15 /*
16  此处控制器的设计思想是, 所有的**控制信号**(以Ctl结尾), 在整个指令执行过程中不会变。可延用单周期CPU设计。
17  只有**使能信号**(以En结尾)在整个指令执行过程中根据阶段的不同会发生改变。此处是状态机需要考虑的问题。
18  根据阶段的不同, 设定不同的**阶段与**寄存器。处于什么阶段, 对应的**阶段与**寄存器为1, 其他**阶段与**寄存器为0。
19  将其与原指令所对应应有的使能信号相与输出。同时单独使用一个always块根据指令类型不同进行状态转移。
20 */
21 reg [4:0] stage=`STAGE_IF;
22 reg PCEnReg=1, IMEnReg=1, RegWrEnReg, RegOFWrEnReg, MemWrEnReg;
23 wire StageIF, StageDCDRF, StageEXE, StageMEM, StageWB;
24
25 assign StageIF= (stage==`STAGE_IF);
26 assign StageDCDRF= (stage==`STAGE_DCDRF);
27 assign StageEXE= (stage==`STAGE_EXE);
28 assign StageMEM= (stage==`STAGE_MEM);
29 assign StageWB= (stage==`STAGE_WB);
30
31 //但凡是转移指令, 必须抢先一步在执行EXE阶段就打开PC使能, 否则下一条指令取指时会取到PC+4, 而非转移的指令
32 wire INST_JUMP; 是否为跳转指令
33 assign INST_JUMP= DecInstBus[`CTLSIG_J] || DecInstBus[`CTLSIG_JAL] || DecInstBus[`CTLSIG_JR] || DecInstBus[`CTLSIG_BEQ];
34 assign PCEn= StageIF && PCEnReg ||
35             StageEXE && INST_JUMP; 跳转指令在执行阶段也要打开PC使能
36 ;
37 assign IMEn= StageIF && IMEnReg ;
38 assign MemWrEn= StageMEM && MemWrEnReg;
39 assign RegWrEn= StageWB && RegWrEnReg;
40 assign RegOFWrEn= StageWB && RegOFWrEnReg;
41
42
43 always@ (*) begin //控制信号的组合逻辑电路
44     ALUCtl=`ALUSIG_ADD;
45     RegWrDstCtl=`REGWRDSTSIG_RT;
46     WrBackCtl=`WRBACKSIG_ALU;
47     ALUSrcCtl=`ALUSRCSIG_EXT;
48     ExtCtl=`EXTSIG_SIGN;
49     DataSizeCtl=`DATASIZESIG_W;
50 end
```

输出的使能信号 均为 “对应阶段信号” 与 “原指令使能信号” 的与

选择信号除NAFLCtl之外都与周期无关

2、控制器 Controller详解 (2)

en <https://github.com/bjrik>

```
67     ALUSrcCtl=`ALUSRCSIG_GPR;
68     NAFLCtl=`NAFLSIG_BEQ;
69 end else if(DecInstBus[`CTLSIG_LUI]) begin
70     ALUCtl=`ALUSIG_LUI;
71 end else if(DecInstBus[`CTLSIG_J]) begin
72     NAFLCtl=`NAFLSIG_J;
73 end else if(DecInstBus[`CTLSIG_ADDI]) begin
74
75 end else if(DecInstBus[`CTLSIG_ADDIU]) begin
76
77 end else if(DecInstBus[`CTLSIG_SLT]) begin
78     ALUCtl=`ALUSIG_SLT;
79     RegWrDstCtl=`REGWRDSTSIG_RD;
80     ALUSrcCtl=`ALUSRCSIG_GPR;
81 end else if(DecInstBus[`CTLSIG_JAL]) begin
82     RegWrDstCtl=`REGWRDSTSIG_GPR_RA;
83     WrBackCtl=`WRBACKSIG_PC;
84     NAFLCtl=`NAFLSIG_JAL;
85 end else if(DecInstBus[`CTLSIG_JR]) begin
86     NAFLCtl=`NAFLSIG_JR;
87 end else if(DecInstBus[`CTLSIG_LB]) begin
88     WrBackCtl=`WRBACKSIG_MEM;
89     DataSizeCtl=`DATASIZESIG_B;
90 end else if(DecInstBus[`CTLSIG_SB]) begin
91     DataSizeCtl=`DATASIZESIG_B;
92 end else begin // DecInstBus[`CTLSIG_NOP] or Unexcepted Situations
93
94 end
95
96 if(StageIF)NAFLCtl=`NAFLSIG_PCNext; //原则：保证取指阶段下地址逻辑始终指向PC+4
97 end
98
99 always@ (*) begin //使能信号的组合逻辑电路
100     PCEnReg=`t; 取指阶段下地址逻辑NAFL始终指向PC+4，否则如果前
101     IMEnReg=`t; 一条指令是跳转指令，会影响下一条指令PC的正常工作
102     RegWrEnReg=`t;
103     RegOFWrEnReg=`f;
104     MemWrEnReg=`f;
105     if(DecInstBus[`CTLSIG_ADDU]) begin
106
107 end else if(DecInstBus[`CTLSIG_SUBU]) begin
108
109 end else if(DecInstBus[`CTLSIG_ORI]) begin
110
111 end else if(DecInstBus[`CTLSIG_LW]) begin
112
113 end else if(DecInstBus[`CTLSIG_SW]) begin
114     RegWrEnReg=`f;
115     MemWrEnReg=`t;
```

2、控制器

Controller详解

(3)

```
142 always@ (posedge clk or posedge rst) begin //状态转移时序逻辑
143     if(rst)stage<=`STAGE_IF;
144     else begin
145         case(stage)
146             `STAGE_IF:stage<=`STAGE_DCDRF;
147             `STAGE_DCDRF:
148                 if(DecInstBus[`CTLSIG_NOP])stage<=`STAGE_IF;
149                 else stage<=`STAGE_EXE;
150             `STAGE_EXE:
151                 //跳转至访存阶段的指令
152                 if(
153                     DecInstBus[`CTLSIG_LW] || DecInstBus[`CTLSIG_SW] || DecInstBus[`CTLSIG_LB] ||
154                     DecInstBus[`CTLSIG_SB]
155                 )
156                     stage<=`STAGE_MEM;
157                 //跳转至回写阶段的指令
158                 else if(
159                     DecInstBus[`CTLSIG_ADDU] || DecInstBus[`CTLSIG_SUBU] || DecInstBus[`CTLSIG_ORI] ||
160                     DecInstBus[`CTLSIG_LUI] || DecInstBus[`CTLSIG_ADDI] || DecInstBus[`CTLSIG_ADDIU] ||
161                     DecInstBus[`CTLSIG_SLT] || DecInstBus[`CTLSIG_JAL]
162                 )
163                     stage<=`STAGE_WB;
164                 //剩下的不论是正常不正常的指令全部跳转回去取指
165                 else
166                     stage<=`STAGE_IF;
167             `STAGE_MEM:
168                 //跳转至回写阶段的指令
169                 if(DecInstBus[`CTLSIG_LW] || DecInstBus[`CTLSIG_LB])
170                     stage<=`STAGE_WB;
171                 //剩下的跳转到取指
172                 else
173                     stage<=`STAGE_IF;
174             `STAGE_WB:stage<=`STAGE_IF;
175             default:stage<=`STAGE_IF;
176         endcase
177     end
178 end
```

3、“延迟寄存器问题” (1)

- 为了符合五周期处理器各阶段定义，JAL指令EXE阶段存入PC新的下地址，而在WB阶段才取下地址回写到GPR，此时读到的下地址是J指令修改过的下地址，而非旧的PC+4，从而导致致命错误

指令\信号	取指IF	译码/读寄存器DCD/RF	执行EXE	访存MEM	回写WB
JAL	√	√	√		√

3、“延迟寄存器问题” (2)

为了解决此问题，在PC到GPR的数据通路上加入一个寄存器，使PC的地址晚一个周期传送到GPR，即可解决回写错误PC的问题，我称之为“延迟寄存器”。

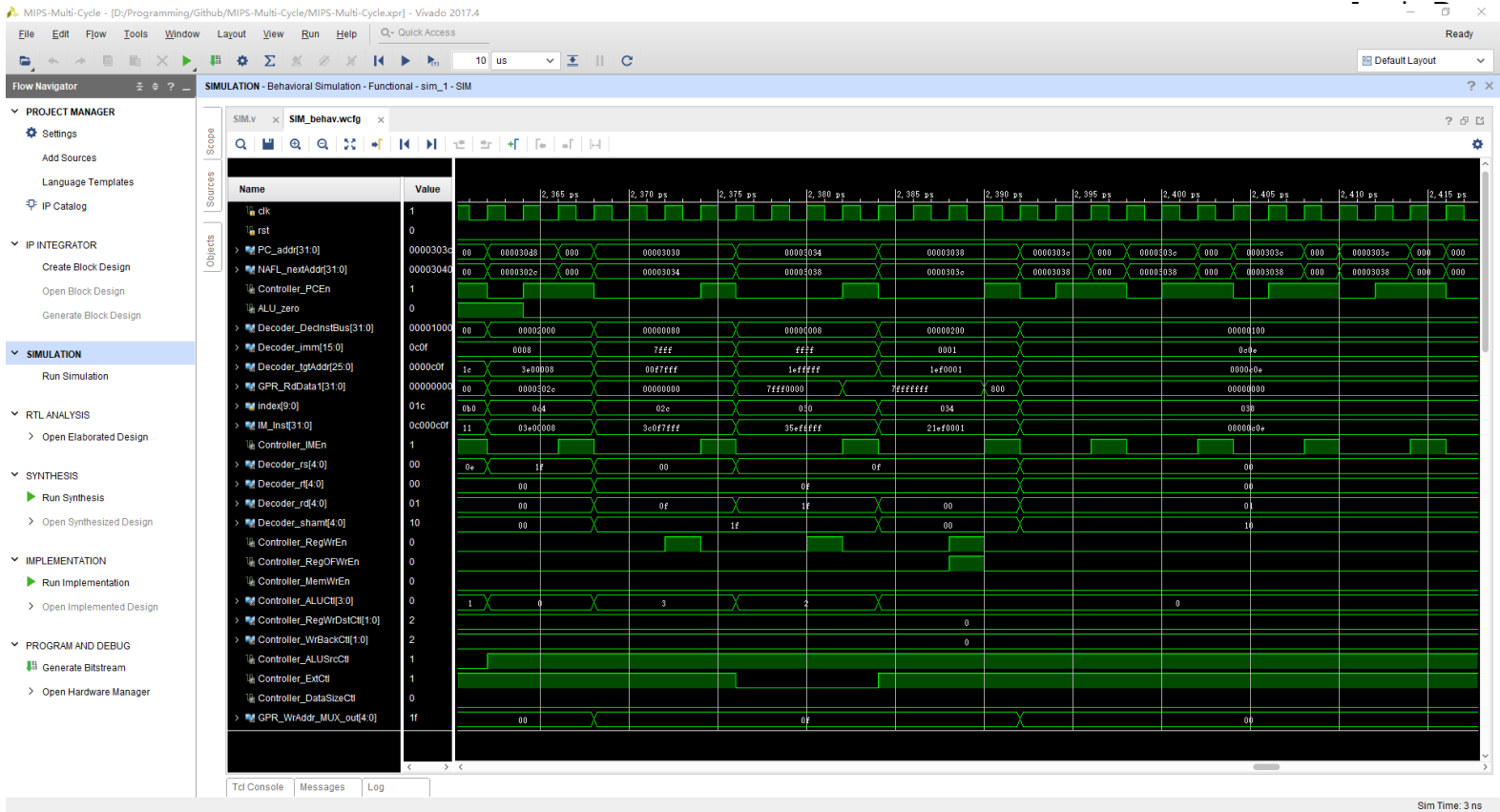
```
69 .PCEn(Controller_PCEn),
70 .IMEn(Controller_IMEn),
71 .RegWrEn(Controller_RegWrEn),
72 .RegOFWrEn(Controller_RegOFWrEn),
73 .MemWrEn(Controller_MemWrEn),
74 .ALUCtl(Controller_ALUCtl),
75 .RegWrDstCtl(Controller_RegWrDstCtl),
76 .WrBackCtl(Controller_WrBackCtl),
77 .ALUSrcCtl(Controller_ALUSrcCtl),
78 .ExtCtl(Controller_ExtCtl),
79 .DataSizeCtl(Controller_DataSizeCtl),
80 .NAFLCtl(Controller_NAFLCtl)
81 );
82
83 //GPR_WrAddr_MUX
84 wire [4:0] GPR_WrAddr_MUX_out;
85 GPR_WrAddr_MUX insGPR_WrAddr_MUX(
86 .RegWrDstCtl(Controller_RegWrDstCtl),
87 .rt(Decoder_rt),
88 .rd(Decoder_rd),
89 .out(GPR_WrAddr_MUX_out)
90 );
91
92 //多周期CPU—处理JAL抢先回写寄存器
93 wire [`QBBus] JALOut_out;
94 QBBusReg insJALOut(
95 .clk(clk),
96 .in(PC_addr),
97 .out(JALOut_out)
98 );
99
100 //GPR_WrData_MUX
101 wire [`QBBus] GPR_WrData_MUX_out;
102 wire [`QBBus] ALUOut_out,DMReg_out;
103 GPR_WrData_MUX insGPR_WrData_MUX(
104 .WrBackCtl(Controller_WrBackCtl),
105 .ALU(ALUOut_out),
106 .MEM(DMReg_out),
107 .PC(JALOut_out),
108 .out(GPR_WrData_MUX_out)
109 );
110
111 //GPR
112 wire [`QBBus] GPR_RdData2;
113 wire ALU_OF;
114 GPR insGPR(
115 .clk(clk),
116 .WrEn(Controller_RegWrEn),
117 .OFWrEn(Controller_RegOFWrEn),
```


MIPS-Lite2 多周期处理器 汇编测试代码

单周期汇编测试代码也用于测试多周期，多周期汇编代码仅测试lb/sb

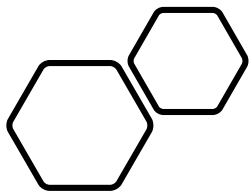
多周期 汇编测试代码

```
ori $0,0x1403 # Test whether the value in $zero can be changed
addu $1,$0,$0
ori $1,0x14ff
sw $1,0($0)
sb $1,5($0)
lw $2,0($0)
lb $3,5($0)
```



MIPS-Lite2 多周期处理器EDA 仿真演示

进行现场仿真演示



三、

MIPS-Lite3 多周期微系统

MIPS-Lite3 多周期微系统 目录

1、数据通路
及控制信号

2、重难点设
计概要

3、中断请求
及中断返回流
程

4、汇编测试
代码

5、实机演示

MIPS-Lite3 多周期微系统 数据通路及控制信号

多周期微系统 控制信号简介

信号名	方向	描述
PCEn	○	PC写使能
IMEn	○	IM取新指令使能
RegWrEn(RegWriteEnable)	○	寄存器写使能
RegOFWrEn(RegOverFlowWriteEnable)	○	寄存器溢出位写使能
MemWrEn(MemoryWriteEnable)	○	内存写使能
CP0WrEn	○	CP0寄存器“软件写”使能
EPCWrEn	○	CP0-EPC寄存器“硬件写”使能
EXLSet	○	CP0-SR寄存器EXL标志位-置1使能
EXLClr	○	CP0-SR寄存器EXL标志位-置0使能
ALUCtl(ALUControl)	○	ALU控制信号 ALUSIG_ADD: 加 ALUSIG_SUB: 减 ALUSIG_OR: 或 ALUSIG_LUI: 置高位 ALUSIG_SLT: 小于比较
RegWrDstCtl(RegWriteDestinationControl)	○	寄存器写目的控制信号 REGWRDSTSIG_RT: 寄存器rt REGWRDSTSIG_RD: 寄存器rd REGWRDSTSIG_GPR_RA: 寄存器\$ra
WrBackCtl(WriteBackControl)	○	回写控制信号 WRBACKSIG_ALU: 从ALU取结果回写 WRBACKSIG_MEM: 从内存取结果回写 WRBACKSIG_PC: 从PC下地址逻辑取PC+4回写 WRBACKSIG_CP0: 从协处理器CP0取数据回写
ALUSrcCtl(ALUSourceControl)	○	ALU数据源控制信号 ALUSRCSIG_GPR: 从寄存器堆读2端口取数据 ALUSRCSIG_EXT: 从位拓展器取数据
ExtCtl(ExtendControl)	○	位拓展器控制信号 EXTSIG_ZERO: 零扩展 EXTSIG_SIGN: 符号扩展
DataSizeCtl(DataSizeControl)	○	数据大小控制信号 DATASIZESIG_W: 传输字 DATASIZESIG_B: 传输字节
NAFLCtl(NextAddressFormulationLogicControl)	○	下地址逻辑控制信号 NAFLSIG_PCNext: 取PC+4 NAFLSIG_BEQ: BEQ指令 NAFLSIG_J: J指令 NAFLSIG_JAL: JAL指令 NAFLSIG_JR: JR指令 NAFLSIG_INT: 中断转移到中断处理子程序 NAFLSIG_EPC: 中断返回

多周期微系统选择信号表

指令\信号	ALUCtl[3:0]	RegWrDstCtl[1:0]	WrBackCtl[1:0]	ALUSrcCtl	ExtCtl	DataSizeCtl	NAFLCtl
DEFAULT	ALUSIG_ADD	REGWRDSTSIG_RT	WRBACKSIG_ALU	ALUSRCSIG_EXT	EXTSIG_SIGN	DATASIZESIG_W	NAFLSIG_PCNext
ADDU	ALUSIG_ADD	REGWRDSTSIG_RD	WRBACKSIG_ALU	ALUSRCSIG_GPR		DATASIZESIG_W	NAFLSIG_PCNext
SUBU	ALUSIG_SUB	REGWRDSTSIG_RD	WRBACKSIG_ALU	ALUSRCSIG_GPR		DATASIZESIG_W	NAFLSIG_PCNext
ORI	ALUSIG_OR	REGWRDSTSIG_RT	WRBACKSIG_ALU	ALUSRCSIG_EXT	EXTSIG_ZERO	DATASIZESIG_W	NAFLSIG_PCNext
LW	ALUSIG_ADD	REGWRDSTSIG_RT	WRBACKSIG_MEM	ALUSRCSIG_EXT	EXTSIG_SIGN	DATASIZESIG_W	NAFLSIG_PCNext
SW	ALUSIG_ADD			ALUSRCSIG_EXT	EXTSIG_SIGN	DATASIZESIG_W	NAFLSIG_PCNext
BEQ	ALUSIG_SUB			ALUSRCSIG_GPR		DATASIZESIG_W	NAFLSIG_BEQ
LUI	ALUSIG_LUI	REGWRDSTSIG_RT	WRBACKSIG_ALU	ALUSRCSIG_EXT		DATASIZESIG_W	NAFLSIG_PCNext
J							NAFLSIG_J
ADDI	ALUSIG_ADD	REGWRDSTSIG_RT	WRBACKSIG_ALU	ALUSRCSIG_EXT	EXTSIG_SIGN	DATASIZESIG_W	NAFLSIG_PCNext
ADDIU	ALUSIG_ADD	REGWRDSTSIG_RT	WRBACKSIG_ALU	ALUSRCSIG_EXT	EXTSIG_SIGN	DATASIZESIG_W	NAFLSIG_PCNext
SLT	ALUSIG_SLT	REGWRDSTSIG_RD	WRBACKSIG_ALU	ALUSRCSIG_GPR		DATASIZESIG_W	NAFLSIG_PCNext
JAL		REGWRDSTSIG_GPR_RA	WRBACKSIG_PC				NAFLSIG_JAL
JR						DATASIZESIG_W	NAFLSIG_JR
LB	ALUSIG_ADD	REGWRDSTSIG_RT	WRBACKSIG_MEM	ALUSRCSIG_EXT	EXTSIG_SIGN	DATASIZESIG_B	NAFLSIG_PCNext
SB	ALUSIG_ADD			ALUSRCSIG_EXT	EXTSIG_SIGN	DATASIZESIG_B	NAFLSIG_PCNext
ERET							NAFLSIG_EPC
MFC0		REGWRDSTSIG_RT	WRBACKSIG_CP0				NAFLSIG_PCNext
MTC0							NAFLSIG_PCNext
NOP							NAFLSIG_PCNext

多周期微系统使能信号表

指令\信号	PCEn	IMEn	RegWrEn	RegOFWrEn	MemWrEn	CP0WrEn	EPCWrEn	EXLSet	EXLClr	执行阶段PC回写
DEFAULT	1	1	1	0	0	0	0	0	0	
ADDU	1	1	1	0	0	0	0	0	0	
SUBU	1	1	1	0	0	0	0	0	0	
ORI	1	1	1	0	0	0	0	0	0	
LW	1	1	1	0	0	0	0	0	0	
SW	1	1	0	0	1	0	0	0	0	
BEQ	1	1	0	0	0	0	0	0	0	√
LUI	1	1	1	0	0	0	0	0	0	
J	1	1	0	0	0	0	0	0	0	√
ADDI	1	1	1	1	0	0	0	0	0	
ADDIU	1	1	1	0	0	0	0	0	0	
SLT	1	1	1	0	0	0	0	0	0	
JAL	1	1	1	0	0	0	0	0	0	√
JR	1	1	0	0	0	0	0	0	0	√
LB	1	1	1	0	0	0	0	0	0	
SB	1	1	0	0	1	0	0	0	0	
ERET	1	1	0	0	0	0	0	0	1	√
MFC0	1	1	1	0	0	0	0	0	0	
MTC0	1	1	0	0	0	1	0	0	0	
NOP	1	1	0	0	0	0	0	0	0	

多周期微系统执行阶段表

指令\信号	取指IF	译码/读寄存器DCD/RF	执行EXE	访存MEM	回写WB
ADDU	√	√	√		√
SUBU	√	√	√		√
ORI	√	√	√		√
LW	√	√	√	√	√
SW	√	√	√	√	
BEQ	√	√	√		
LUI	√	√	√		√
J	√	√	√		
ADDI	√	√	√		√
ADDIU	√	√	√		√
SLT	√	√	√		√
JAL	√	√	√		√
JR	√	√	√		
LB	√	√	√	√	√
SB	√	√	√	√	
ERET	√	√			
MFC0	√	√			√
MTC0	√	√			
NOP	√	√			

MIPS-Lite3 多周期微系统 重难点设计概要

1、由五周期变为六周期

- 取指、译码、执行、访存、回写
- -> (添加一个周期——中断)
- 取指、译码、执行、访存、回写、中断

2、延迟回写问题（同多周期处理器“延迟寄存器”）

```
114 .interrupt(CP0_interrupt) //CPU中断信号
115 );
116
117 //GPR_WrAddr_MUX
118 wire [4:0] GPR_WrAddr_MUX_out;
119 GPR_WrAddr_MUX insGPR_WrAddr_MUX(
120 .RegWrDstCtl(Controller_RegWrDstCtl),
121 .rt(Decoder_rt),
122 .rd(Decoder_rd),
123 .out(GPR_WrAddr_MUX_out)
124 );
125
126 //多周期CPU—处理PC延迟回写寄存器—JAL和中断使用
127 QBBusReg insPCDelayOut(
128 .clk(clk),
129 .in(PC_addr),
130 .out(PCDelayOut_out)
131 );
132
133 //微系统—处理MFC0延迟回写寄存器
134 wire [`QBBus] MFC0Out_out;
135 QBBusReg insMFC0Out(
136 .clk(clk),
137 .in(CP0_DataOut),
138 .out(MFC0Out_out)
139 );
140
141 //GPR_WrData_MUX
142 wire [`QBBus] GPR_WrData_MUX_out;
143 wire [`QBBus] ALUOut_out,DMReg_out;
144 GPR_WrData_MUX insGPR_WrData_MUX(
145 .WrBackCtl(Controller_WrBackCtl),
146 .ALU(ALUOut_out),
147 .MEM(DMReg_out),
148 .PC(PCDelayOut_out),
149 .CP0(MFC0Out_out),
150 .out(GPR_WrData_MUX_out)
151 );
152
```

JAL和中断都会碰到先写PC，后取PC+4旧值的问题

MFC0读CP0与寄存器回写是跨周期的，因此中间需要加“延迟寄存器”

3、NAFL下地址形成逻辑 ——中断和中断返回

```

1  //指令形成单元 — 下地址形成逻辑 Next Address Formulation Logic
2
3  `include "defines.v"
4
5  module NAFL(
6      input [`QBBus] addr,
7      output reg [`QBBus] nextAddr,
8      input [2:0] NAFLCtl,
9      input beqZero,
10     input [`DBBus] beqShift, // beq指令, 16比特左移两位后变18比特再符号拓展加到PC
11     input [25:0] jPadding, // j和jal指令, 26比特左移两位后变28比特置PC低位
12     input [`QBBus] jrAddr, // jr指令从$ra直接读入的32位地址
13     output [`QBBus] nextInstAddr,
14     input [`QBBus] EPC
15 );
16
17     assign nextInstAddr = addr+4;
18
19     /*
20      在取指阶段, 产生的下地址统一为PC+4。
21      在执行阶段, 会产生跳转指令。此时PC的值已由PC变为PC+4, 所以只需做增补量。
22      这点与单周期CPU非常不同, 需要特别注意!
23     */
24
25     always@ (*) begin
26         if(NAFLCtl==`NAFLSIG_BEQ&&beqZero)nextAddr=addr+{{14{beqShift[15]}},beqShift,2'b00};
27         else if(NAFLCtl==`NAFLSIG_BEQ)nextAddr=addr;
28         else if(NAFLCtl==`NAFLSIG_J||NAFLCtl==`NAFLSIG_JAL)nextAddr={addr[31:28],jPadding,2'b00};
29         else if(NAFLCtl==`NAFLSIG_JR)nextAddr=jrAddr;
30         else if(NAFLCtl==`NAFLSIG_INT)nextAddr=32'h0000_4180;
31         else if(NAFLCtl==`NAFLSIG_EPC)nextAddr=EPC;
32         else nextAddr=nextInstAddr;
33     end
34
35 endmodule
36

```

4、控制器Controller (1)

```
1 //指令译码单元 — 控制器 Controller
2
3 `include "defines.v"
4
5 module Controller(
6     input clk,rst,
7     input [`QBBus] DecInstBus,
8     output wire PCEn,IMEn,RegWrEn,RegOFWrEn,MemWrEn, //PC写使能,IM取新指令使能,寄存器写使能,寄存器溢出写使能,内存写使能
9     output reg [3:0] ALUCtl, //ALU控制信号
10    output reg [1:0] RegWrDstCtl,WrBackCtl, //寄存器写目标控制信号,回写控制信号
11    output reg ALUSrcCtl, ExtCtl, DataSizeCtl, //ALU数据源控制信号,位拓展器控制信号,数据大小控制信号
12    output reg [2:0] NAFLCtl=`NAFLSIG_PCNext, //NAFL下地址逻辑控制信号
13    output reg CP0WrEn=0,EPCWrEn=0,EXLSet=0,EXLClr=0,
14    input interrupt //中断
15);
16
17 /*
18  此处控制器的设计思想是,所有的**控制信号**(以Ctl结尾),在整个指令执行过程中不会变。可延用单周期CPU设计。
19  只有**使能信号**(以En结尾)在整个指令执行过程中根据阶段的不同会发生改变。此处是状态机需要考虑的问题。
20  根据阶段的不同,设定不同的**阶段与**寄存器。处于什么阶段,对应的**阶段与**寄存器为1,其他**阶段与**寄存器为0。
21  将其与原指令所对应应有的使能信号相与输出。同时单独使用一个always块根据指令类型不同进行状态转移。
22 */
23 reg [4:0] stage=`STAGE_IF;
24 reg PCEnReg=1,IMEnReg=1,RegWrEnReg,RegOFWrEnReg,MemWrEnReg;
25 wire StageIF,StageDCDRF,StageEXE,StageMEM,StageWB;
26
27 assign StageIF= (stage==`STAGE_IF);
28 assign StageDCDRF= (stage==`STAGE_DCDRF);
29 assign StageEXE= (stage==`STAGE_EXE);
30 assign StageMEM= (stage==`STAGE_MEM);
31 assign StageWB= (stage==`STAGE_WB);
32 assign StageINT= (stage==`STAGE_INT);
33
34 //但凡是转移指令,必须抢先一步在执行EXE阶段就打开PC使能,否则下一条指令取指时会取到PC+4,而非转移的指令
35 wire INST_JUMP;
36 assign INST_JUMP= DecInstBus[`CTLSIG_J] || DecInstBus[`CTLSIG_JAL] || DecInstBus[`CTLSIG_JR] || DecInstBus[`CTLSIG_BEQ];
37 assign PCEn= StageIF && PCEnReg || //取指阶段PC+4
38             StageEXE && INST_JUMP || //跳转指令的执行阶段
39             StageDCDRF && DecInstBus[`CTLSIG_ERET] || //中断返回的译码阶段
40             StageINT && interrupt //中断阶段且有中断
41 ;
42 assign IMEn= StageIF && IMEnReg ;
43 assign MemWrEn= StageMEM && MemWrEnReg;
44 assign RegWrEn= StageWB && RegWrEnReg;
45 assign RegOFWrEn= StageWB && RegOFWrEnReg;
46
47
48 always@ (*) begin //控制信号的组合逻辑电路
49     ALUCtl = `ALUSIG_ADD;
```

微系统PC使能时段

4、控制器Controller (2)

```
96     DataSizeCtl=`DATASIZESIG_B;
97   end else if(DecInstBus[`CTLSIG_SB]) begin
98     DataSizeCtl=`DATASIZESIG_B;
99   end else if(DecInstBus[`CTLSIG_MFC0]) begin
100     WrBackCtl=`WRBACKSIG_CP0;
101   end else if(DecInstBus[`CTLSIG_MTC0]) begin
102     CP0WrEn=`t;
103   end else if(DecInstBus[`CTLSIG_ERET]) begin
104     NAFLCtl=`NAFLSIG_EPC;
105     EXLClr=`t;
106   end else begin // DecInstBus[`CTLSIG_NOP] or Unexcepted Situations
107
108   end
109
110   if(StageIF)NAFLCtl=`NAFLSIG_PCNext; //原则：保证取指阶段下地址逻辑始终指向PC+4
111   if(StageINT && interrupt)NAFLCtl=`NAFLSIG_INT;
112 end
113
114 always@ (*) begin //使能信号的组合逻辑电路
115   PCEnReg=`t;
116   IMEnReg=`t;
117   RegWrEnReg=`t;
118   RegOFWrEnReg=`f;
119   MemWrEnReg=`f;
120   if(DecInstBus[`CTLSIG_ADDU]) begin
121
122   end else if(DecInstBus[`CTLSIG_SUBU]) begin
```

“中断周期” 且有中断信号，那么下地址控制信号为转中断处理程序

4、控制器Controller

(3)

```

162
163 always@ (posedge clk or posedge rst) begin //状态转移时序逻辑
164     if(rst)stage<=`STAGE_IF;
165     else begin
166         case(stage)
167             `STAGE_IF: begin
168                 EPCWrEn<=0;
169                 EXLSet<=0;
170                 stage<=`STAGE_DCDRF;
171             end
172             `STAGE_DCDRF:
173                 //直接跳转到中断
174                 if(
175                     DecInstBus[`CTLSIG_NOP] || DecInstBus[`CTLSIG_ERET] || DecInstBus[`CTLSIG_MTC0]
176                 )
177                     stage<=`STAGE_INT;
178                 else if(
179                     DecInstBus[`CTLSIG_MFC0]
180                 )
181                     stage<=`STAGE_WB;
182                 //剩下的都跳转到执行
183                 else
184                     stage<=`STAGE_EXE;
185             `STAGE_EXE:
186                 //跳转至访存阶段的指令
187                 if(
188                     DecInstBus[`CTLSIG_LW] || DecInstBus[`CTLSIG_SW] || DecInstBus[`CTLSIG_LB] ||
189                     DecInstBus[`CTLSIG_SB]
190                 )
191                     stage<=`STAGE_MEM;
192                 //跳转至回写阶段的指令
193                 else if(
194                     DecInstBus[`CTLSIG_ADDU] || DecInstBus[`CTLSIG_SUBU] || DecInstBus[`CTLSIG_ORI] ||
195                     DecInstBus[`CTLSIG_LUI] || DecInstBus[`CTLSIG_ADDI] || DecInstBus[`CTLSIG_ADDIU] ||
196                     DecInstBus[`CTLSIG_SLT] || DecInstBus[`CTLSIG_JAL]
197                 )
198                     stage<=`STAGE_WB;
199                 //剩下的不论是正常不正常的指令全部跳转去中断
200                 else
201                     stage<=`STAGE_INT;
202             `STAGE_MEM:

```

EPC写失能
EXL置1失能

```

177         stage<=`STAGE_INT;
178     else if(
179         DecInstBus[`CTLSIG_MFC0]
180     )
181         stage<=`STAGE_WB;
182     //剩下的都跳转到执行
183     else
184         stage<=`STAGE_EXE;
185     `STAGE_EXE:
186         //跳转至访存阶段的指令
187         if(
188             DecInstBus[`CTLSIG_LW] || DecInstBus[`CTLSIG_SW] || DecInstBus[`CTLSIG_LB] ||
189             DecInstBus[`CTLSIG_SB]
190         )
191             stage<=`STAGE_MEM;
192         //跳转至回写阶段的指令
193         else if(
194             DecInstBus[`CTLSIG_ADDU] || DecInstBus[`CTLSIG_SUBU] || DecInstBus[`CTLSIG_ORI] ||
195             DecInstBus[`CTLSIG_LUI] || DecInstBus[`CTLSIG_ADDI] || DecInstBus[`CTLSIG_ADDIU] ||
196             DecInstBus[`CTLSIG_SLT] || DecInstBus[`CTLSIG_JAL]
197         )
198             stage<=`STAGE_WB;
199         //剩下的不论是正常不正常的指令全部跳转去中断
200         else
201             stage<=`STAGE_INT;
202     `STAGE_MEM:
203         //跳转至回写阶段的指令
204         if(DecInstBus[`CTLSIG_LW] || DecInstBus[`CTLSIG_LB])
205             stage<=`STAGE_WB;
206         //剩下的跳转到中断
207         else
208             stage<=`STAGE_INT;
209     `STAGE_WB:stage<=`STAGE_INT;
210     `STAGE_INT: begin
211         if(interrupt) begin
212             EPCWrEn<=1;
213             EXLSet<=1;
214         end
215         stage<=`STAGE_IF;
216     end
217     default:stage<=`STAGE_IF;
218 endcase
219 end
220 end
221
222 endmodule
223

```

EPC写使能
EXL置1使能

5、协处理器CP0

Jack Ren <https://github.com/bjrik>

```
1 //协处理器CP0 Co-Processor 0
2
3 `include "defines.v"
4 `define CP0_SR 12
5 `define CP0_CAUSE 13
6 `define CP0_EPC 14
7 `define CP0_PRID 15
8
9
10 module CP0(
11     input clk,rst,WrEn,
12     input [4:0] addr,
13     input [`QBBus] DataIn,
14     output reg [`QBBus] DataOut,
15
16     input EPCWrEn,
17     input [`QBBus] EPCIn, //EPC入总线
18     output [`QBBus] EPCOut, //EPC出总线
19
20     input [5:0] HWInt,
21     input EXLSet,EXLClr, //置1 SR的EXL, 清0 SR的EXL
22     output interrupt //CPU中断信号
23 );
24
25 reg [5:0] IM=6'b000000;
26 reg EXL=0,IE=0;
27 wire [`QBBus] SR; //SR寄存器数据线
28 reg [`QBBus] EPC; //EPC寄存器, 不允许软件写
29 wire [`QBBus] Cause; //Cause寄存器, 不允许软件写
30 wire [`QBBus] PrID; //PrID寄存器, 不允许写
31
32 assign SR={16'd0,IM,8'd0,EXL,IE};
33 assign Cause={16'd0,HWInt,10'd0};
34 assign PrID=32'h18041403; //CPU标识号
35
36 assign EPCOut=EPC;
37 assign interrupt= (!(HWInt & IM)) & IE & !EXL ;
38
39 always@ (*) begin //寄存器数据输出
40     case(addr)
41         `CP0_SR:DataOut=SR;
42         `CP0_CAUSE:DataOut=Cause;
43         `CP0_EPC:DataOut=EPC;
44         `CP0_PRID:DataOut=PrID;
45         default:DataOut=32'd0;
46     endcase
47 end
```

```
49 //软件写寄存器SR (IM和IE), 其他寄存器软件不可写
50 always@ (posedge clk or posedge rst) begin
51     if(rst) begin
52         IM<=6'b000000; //屏蔽所有硬件中断
53         IE<=0; //全局中断失能
54     end else if(WrEn) begin
55         if(addr==`CP0_SR) begin
56             IM<=DataIn[15:10];
57             IE<=DataIn[0];
58         end
59     end
60 end
61
62 //硬件写寄存器SR (EXL)
63 always@ (posedge clk or posedge rst) begin
64     if(rst) begin
65         EXL<=0; //默认允许中断
66     end else if(EXLSet) begin
67         EXL<=1; //禁止中断嵌套
68     end else if(EXLClr) begin
69         EXL<=0; //恢复允许中断
70     end
71 end
72
73 //硬件写寄存器EPC
74 always@ (posedge clk or posedge rst) begin
75     if(rst) begin
76         EPC<=0;
77     end else if(EPCWrEn) begin
78         EPC<=EPCIn;
79     end
80 end
81
82 endmodule
83
```

6、存储器DM (内存/外存通过DM访问)

Jack Ren <https://github.com/bjrik>

```
1 // 系统内存
2
3 `include "defines.v"
4
5 module dm_12k(
6     input [`QBBus] addr,
7     input [`QBBus] din,
8     input we,
9     input clk,
10    output [`QBBus] dout,
11    input wire DataSizeCtl,
12
13    //与系统桥相连的线，除中断外全部整合到DM
14    output MIPS_WrEn,
15    output [`QBBus] MIPS_Addr,
16    input [`QBBus] Bridge_RD,
17    output [`QBBus] MIPS_DataOut
18 );
19
20
21 wire InnerMemWrEn, IsBridgeAddr;
22 assign IsBridgeAddr= (addr[31:8]==24'h0000_7F);
23 assign MIPS_Addr=addr; //访问外设地址直接输出
24 assign MIPS_DataOut=din; //外设写数据直接输出，对外设只实现LW/SW，不实现LB/SB
25 assign MIPS_WrEn= we && IsBridgeAddr; //地址前缀为外设地址且有写使能时才向系统桥发使能
26 assign InnerMemWrEn= we && !IsBridgeAddr; //内部内存写使能
27
28 reg [`BBus] dm[1023:0]; //12287, 1024方便调试
29 wire [15:0] index;
30
31 assign index=addr[15:0];
32 //Dout为小端序
33 assign dout= (!IsBridgeAddr && DataSizeCtl==`DATASIZESIG_B) ? {{24{dm[index][7]}},dm[index]} :
34    (!IsBridgeAddr) ? {dm[index+3],dm[index+2],dm[index+1],dm[index]} :
35    IsBridgeAddr ? Bridge_RD :
36    32'hEEEEEEEE
37    ;
38
39 always@ (posedge clk) begin
40     if(InnerMemWrEn) begin //不是向外设发送数据就是向内存发送
41         if(DataSizeCtl==`DATASIZESIG_B) begin
42             dm[index]<=din[7:0];
43         end else begin
44             dm[index]<=din[7:0];
45             dm[index+1]<=din[15:8];
46             dm[index+2]<=din[23:16];
47             dm[index+3]<=din[31:24];
48         end
49     end
50 end
```

7、系统桥Bridge

1ook Den <https://github.com/bjrik>

```
1 //外设 — 系统桥 Bridge
2
3 `include "defines.v"
4
5 module Bridge(
6     input clk,WrEn,
7     input [`QBBus] Addr,WD, //与CPU相连的总线
8     output [`QBBus] RD,
9     output [5:0] HWInt, //中断
10
11     //定时器DEV1, 输入设备DEV2, 输出设备DEV3
12     output DEV1_WrEn,DEV3_WrEn,
13     output [3:0] DEV_Addr, //只写定时器的地址
14     output [`QBBus] DEV_WD, //写给定时器、输出设备的输出数据; 输入设备不留输出端口
15     input [`QBBus] DEV1_RD,DEV2_RD, //从定时器和输入设备的输入数据; 输出设备不留输入端口
16     input DEV1_interrupt //定时器中断输入
17 );
18
19 wire DEV1_RdEn,DEV2_RdEn;
20 assign DEV1_RdEn= Addr[31:4]==28'h0000_7F0;
21 assign DEV2_RdEn= Addr[31:4]==28'h0000_7F1;
22
23 assign DEV1_WrEn= WrEn && Addr[31:4]==28'h0000_7F0;
24 assign DEV3_WrEn= WrEn && Addr[31:4]==28'h0000_7F2;
25 assign DEV_Addr=Addr[3:0];
26 assign DEV_WD=WD;
27
28 assign RD= DEV1_RdEn ? DEV1_RD :
29           DEV2_RdEn ? DEV2_RD :
30           32'h00000000;
31
32 assign HWInt[0]=DEV1_interrupt;
33 assign HWInt[5:1]=5'b00000;
34
35 endmodule
36
```

8、外设——定时器 TimeCounter

Jack Ren <https://github.com/bjrik>

```
1 //外设 — 定时器 TimeCounter
2
3 `include "defines.v"
4
5 module timeCounter(
6     input clk,rst,WrEn,
7     input [3:0] addr,
8     input [`QBBus] DataIn,
9     output reg [`QBBus] DataOut,
10    output wire interrupt
11);
12
13    reg [`QBBus] CTRL=0,PRESET=0,COUNT=0;
14    wire interruptWire,COUNTReload;
15    assign interrupt= CTRL[3] && CTRL[2:1]==2'b00 && interruptWire;
16    assign interruptWire= COUNT==0 ;
17    assign COUNTReload= WrEn && addr[3:2]==2'b01;
18
19
20 always@ (*) begin //数据输出
21     case(addr[3:2])
22         2'b00:DataOut=CTRL;
23         2'b01:DataOut=PRESET;
24         2'b10:DataOut=COUNT;
25         default:DataOut=32'hheeeeeee;
26     endcase
27 end
28
29 always@ (posedge clk or posedge rst) begin //数据输入
30     if(rst) begin
31         CTRL<=0;
32         PRESET<=0;
33     end else if(WrEn) begin
34         if(addr[3:2]==2'b00)CTRL[3:0]<=DataIn[3:0];
35         else if(addr[3:2]==2'b01)PRESET<=DataIn;
36     end
37 end
38
39 always@ (posedge clk or posedge rst) begin //计时功能，本课设使用模式0
40     if(rst) COUNT<=0;
41     else if(!CTRL[0] && COUNT==0 && WrEn) begin //只有停止状态下才允许重新加载计数器
42         COUNT<=DataIn;
43     end else if(CTRL[0]) begin
44         if(COUNT==0) begin
45             if(CTRL[2:1]==2'b01)COUNT<=PRESET;
46             end else COUNT<=COUNT-1;
47         end
48     end
49 end
```

MIPS-Lite3 多周期微系统 中断请求及中断返回流程

中断请求

```

1 //外设 — 定时器 TimeCounter
2
3 `include "defines.v"
4
5 module timeCounter(
6     input clk,rst,WrEn,
7     input [3:0] addr,
8     input [`QBBus] DataIn,
9     output reg [`QBBus] DataOut,
10    output wire interrupt
11 );
12
13    reg [`QBBus] CTRL=0,PRESET=0,COUNT=0;
14    wire interruptWire,COUNTReload;
15    assign interrupt= CTRL[3] && CTRL[2:1]==2'b00 && interruptWire;
16    assign interruptWire= COUNT==0 ;
17    assign COUNTReload= WrEn && addr[3:2]==2'b01;
18
19
20    always@ (*) begin //数据输出
21        case(addr[3:2])
22            2'b00:DataOut=CTRL;
23            2'b01:DataOut=PRESET;
24            2'b10:DataOut=COUNT;
25            default:DataOut=32'hheeeeeeee;
26        endcase
27    end
28
29    always@ (posedge clk or posedge rst) begin //数据输入
30        if(rst) begin
31            CTRL<=0;
32            PRESET<=0;
33        end else if(WrEn) begin
34            if(addr[3:2]==2'b00)CTRL[3:0]<=DataIn[3:0];
35            else if(addr[3:2]==2'b01)PRESET<=DataIn;
36        end
37    end
38
39    always@ (posedge clk or posedge rst) begin //计时功能, 本课设使用模式0
40        if(rst) COUNT<=0;

```

1、定时器倒计时时间到且未屏蔽中断,则将信号传入系统桥


```

2
3 `include "defines.v"
4
5 module Bridge(
6     input clk,WrEn,
7     input [`QBBus] Addr,WD, //与CPU相连的总线
8     output [`QBBus] RD,
9     output [5:0] HWInt, //中断
10
11     //定时器DEV1, 输入设备DEV2, 输出设备DEV3
12     output DEV1_WrEn,DEV3_WrEn,
13     output [3:0] DEV_Addr, //只写定时器的地址
14     output [`QBBus] DEV_WD, //写给定时器、输出设备的输出数据；输入设备不留输出端口
15     input [`QBBus] DEV1_RD,DEV2_RD, //从定时器和输入设备的输入数据；输出设备不留输入端口
16     input DEV1_interrupt //定时器中断输入
17 );
18
19 wire DEV1_RdEn,DEV2_RdEn;
20 assign DEV1_RdEn= Addr[31:4]==28'h0000_7F0;
21 assign DEV2_RdEn= Addr[31:4]==28'h0000_7F1;
22
23 assign DEV1_WrEn= WrEn && Addr[31:4]==28'h0000_7F0;
24 assign DEV3_WrEn= WrEn && Addr[31:4]==28'h0000_7F2;
25 assign DEV_Addr=Addr[3:0];
26 assign DEV_WD=WD;
27
28 assign RD= DEV1_RdEn ? DEV1_RD :
29         DEV2_RdEn ? DEV2_RD :
30         32'h00000000;
31
32 assign HWInt[0]=DEV1_interrupt;
33 assign HWInt[5:1]=5'b00000;
34
35 endmodule
36

```

2、系统桥接收到定时器中断信号并传入CPU的CP0



```
13 input [`QBBus] DataIn,
14 output reg [`QBBus] DataOut,
15
16 input EPCWrEn,
17 input [`QBBus] EPCIn, //EPC入总线
18 output [`QBBus] EPCOut, //EPC出总线
19
20 input [5:0] HWInt,
21 input EXLSet,EXLClr, //置1 SR的EXL, 清0 SR的EXL
22 output interrupt //CPU中断信号
23 );
24
25 reg [5:0] IM=6'b000000;
26 reg EXL=0,IE=0;
27 wire [`QBBus] SR; //SR寄存器数据线
28 reg [`QBBus] EPC; //EPC寄存器, 不允许软件写
29 wire [`QBBus] Cause; //Cause寄存器, 不允许软件写
30 wire [`QBBus] PrID; //PrID寄存器, 不允许写
31
32 assign SR={16'd0,IM,8'd0,EXL,IE};
33 assign Cause={16'd0,HWInt,10'd0};
34 assign PrID=32'h18041403; //CPU标识号
35
36 assign EPCOut=EPC;
37 assign interrupt= (|(HWInt & IM)) & IE & !EXL ;
38
39 always@ (*) begin //寄存器数据输出
40     case(addr)
41         `CP0_SR:DataOut=SR;
42         `CP0_CAUSE:DataOut=Cause;
43         `CP0_EPC:DataOut=EPC;
44         `CP0_PRID:DataOut=PrID;
45         default:DataOut=32'd0;
46     endcase
47 end
```

3、CP0检测是否全局中断屏蔽、已在中断处理子程序中、是否屏蔽特定中断；若非向控制器Controller发送中断信号

```

1 //指令译码单元 — 控制器 Controller
2
3 `include "defines.v"
4
5 module Controller(
6     input clk,rst,
7     input [`QBBus] DecInstBus,
8     output wire PCEn,IMEn,RegWrEn,RegOFWrEn,MemWrEn, //PC写使能, IM取新指令使能, 寄存器写使能, 寄存器溢出写使能, 内
9     output reg [3:0] ALUCtl, //ALU控制信号
10    output reg [1:0] RegWrDstCtl,WrBackCtl, //寄存器写目标控制信号, 回写控制信号
11    output reg ALUSrcCtl, ExtCtl, DataSizeCtl, //ALU数据源控制信号, 位拓展器控制信号, 数据大小控制信号
12    output reg [2:0] NAFLCtl=`NAFLSIG_PCNext, //NAFL下地址逻辑控制信号
13    output reg CP0WrEn=0,EPCWrEn=0,EXLSet=0,EXLClr=0,
14    input interrupt //中断
15 );
16
17 /*
18  此处控制器的设计思想是, 所有的**控制信号**(以Ctl结尾), 在整个指令执行过程中不会变。可延用单周期CPU设计。
19  只有**使能信号**(以En结尾)在整个指令执行过程中根据阶段的不同会发生改变。此处是状态机需要考虑的问题。
20  根据阶段的不同, 设定不同的**阶段与**寄存器。处于什么阶段, 对应的**阶段与**寄存器为1, 其他**阶段与**寄存器为0。
21  将其与原指令所对应应有的使能信号相与输出。同时单独使用一个always块根据指令类型不同进行状态转移。
22 */
23 reg [4:0] stage=`STAGE_IF;
24 reg PCEnReg=1,IMEnReg=1,RegWrEnReg,RegOFWrEnReg,MemWrEnReg;
25 wire StageIF,StageDCDRF,StageEXE,StageMEM,StageWB;
26
27 assign StageIF= (stage==`STAGE_IF);
28 assign StageDCDRF= (stage==`STAGE_DCDRF);
29 assign StageEXE= (stage==`STAGE_EXE);
30 assign StageMEM= (stage==`STAGE_MEM);
31 assign StageWB= (stage==`STAGE_WB);
32 assign StageINT= (stage==`STAGE_INT);
33
34 //但凡是转移指令, 必须抢先一步在执行EXE阶段就打开PC使能, 否则下一条指令取指时会取到PC+4, 而非转移的指令
35 wire INST_JUMP;
36 assign INST_JUMP= DecInstBus[`CTLSIG_J] || DecInstBus[`CTLSIG_JAL] || DecInstBus[`CTLSIG_JR] || DecInstBus
37 assign PCEn= StageIF && PCEnReg || //取指阶段PC+4
38             StageEXE && INST_JUMP || //跳转指令的执行阶段
39             StageDCDRF && DecInstBus[`CTLSIG_ERET] || //中断返回的译码阶段
40             StageINT && interrupt //中断阶段且有中断
41 ;
42 assign IMEn= StageIF && IMEnReg ;
43 assign MemWrEn= StageMEM && MemWrEnReg;
44 assign RegWrEn= StageWB && RegWrEnReg;
45 assign RegOFWrEn= StageWB && RegOFWrEnReg;
46
47
48 always@ (*) begin //控制信号的组合逻辑电路
49     ALUCtl=`ALUSIG_ADD;

```

4、PC使能 在中断阶段 开启

微系统PC使能时段

5、下地址逻辑准备转向中断处理程序

```
96     DataSizeCtl=`DATASIZESIG_B;
97   end else if(DecInstBus[`CTLSIG_SB]) begin
98     DataSizeCtl=`DATASIZESIG_B;
99   end else if(DecInstBus[`CTLSIG_MFC0]) begin
100     WrBackCtl=`WRBACKSIG_CP0;
101   end else if(DecInstBus[`CTLSIG_MTC0]) begin
102     CP0WrEn=`t;
103   end else if(DecInstBus[`CTLSIG_ERET]) begin
104     NAFLCtl=`NAFLSIG_EPC;
105     EXLClr=`t;
106   end else begin // DecInstBus[`CTLSIG_NOP] or Unexceped Situations
107
108   end
109
110   if(StageIF)NAFLCtl=`NAFLSIG_PCNext; //原则：保证取指阶段下地址逻辑始终指向PC+4
111   if(StageINT && interrupt)NAFLCtl=`NAFLSIG_INT;
112 end
113
114 always@ (*) begin //使能信号的组合逻辑电路
115   PCEnReg=`t;
116   IMEnReg=`t;
117   RegWrEnReg=`t;
118   RegOFWrEnReg=`f;
119   MemWrEnReg=`f;
120   if(DecInstBus[`CTLSIG_ADDU]) begin
121
122   end else if(DecInstBus[`CTLSIG_SUBU]) begin
```

“中断周期” 且有中断信号，那么下地址控制信号为转中断处理程序


```

177         stage<= `STAGE_INT;
178     else if(
179         DecInstBus[`CTLSIG_MFC0]
180     )
181         stage<= `STAGE_WB;
182     //剩下的都跳转到执行
183     else
184         stage<= `STAGE_EXE;
185     `STAGE_EXE:
186     //跳转至访存阶段的指令
187     if(
188         DecInstBus[`CTLSIG_LW] || DecInstBus[`CTLSIG_SW] || DecInstBus[`CTLSIG_LB] ||
189         DecInstBus[`CTLSIG_SB]
190     )
191         stage<= `STAGE_MEM;
192     //跳转至回写阶段的指令
193     else if(
194         DecInstBus[`CTLSIG_ADDU] || DecInstBus[`CTLSIG_SUBU] || DecInstBus[`CTLSIG_ORI] ||
195         DecInstBus[`CTLSIG_LUI] || DecInstBus[`CTLSIG_ADDI] || DecInstBus[`CTLSIG_ADDIU] ||
196         DecInstBus[`CTLSIG_SLT] || DecInstBus[`CTLSIG_JAL]
197     )
198         stage<= `STAGE_WB;
199     //剩下的不论是正常不正常的指令全部跳转去中断
200     else
201         stage<= `STAGE_INT;
202     `STAGE_MEM:
203     //跳转至回写阶段的指令
204     if(DecInstBus[`CTLSIG_LW] || DecInstBus[`CTLSIG_LB])
205         stage<= `STAGE_WB;
206     //剩下的跳转到中断
207     else
208         stage<= `STAGE_INT;
209     `STAGE_WB: stage<= `STAGE_INT;
210     `STAGE_INT: begin
211         if(interrupt) begin
212             EPCWrEn<=1;
213             EXLSet<=1;
214         end
215         stage<= `STAGE_IF;
216     end
217     default: stage<= `STAGE_IF;
218 endcase
219 end
220 end
221
222 endmodule
223

```

6、控制器执行到中断周期时，“非阻塞语句”在下一条指令取指阶段：打开EPC写使能（准备存储PC+4），EXL使能置1（屏蔽新的中断）

EPC写使能
EXL置1使能

```

163 always@ (posedge clk or posedge rst) begin //状态转移时序逻辑
164     if(rst)stage<=`STAGE_IF;
165     else begin
166         case(stage)
167             `STAGE_IF: begin
168                 EPCWrEn<=0;
169                 EXLSet<=0;
170                 stage<=`STAGE_DCDRF;
171             end
172             `STAGE_DCDRF:
173                 //直接跳转到中断
174                 if(
175                     DecInstBus[`CTLSIG_NOP] || DecInstBus[`CTLSIG_ERET] || DecInstBus[
176                 )
177                     stage<=`STAGE_INT;
178                 else if(
179                     DecInstBus[`CTLSIG_MFC0]
180                 )
181                     stage<=`STAGE_WB;
182                 //剩下的都跳转到执行
183                 else
184                     stage<=`STAGE_EXE;
185             `STAGE_EXE:
186                 //跳转至访存阶段的指令
187                 if(
188                     DecInstBus[`CTLSIG_LW] || DecInstBus[`CTLSIG_SW] || DecInstBus[`CT
189                     DecInstBus[`CTLSIG_SB]
190                 )
191                     stage<=`STAGE_MEM;
192                 //跳转至回写阶段的指令
193                 else if(
194                     DecInstBus[`CTLSIG_ADDU] || DecInstBus[`CTLSIG_SUBU] || DecInstBus
195                     DecInstBus[`CTLSIG_LUI] || DecInstBus[`CTLSIG_ADDI] || DecInstBus[
196                     DecInstBus[`CTLSIG_SLT] || DecInstBus[`CTLSIG_JAL]
197                 )
198                     stage<=`STAGE_WB;
199                 //剩下的不论是正常不正常的指令全部跳转去中断
200                 else
201                     stage<=`STAGE_INT;
202             `STAGE_MEM:

```

EPC写失能
EXL置1失能

7、新指令取指阶段上升沿:

- (1) PC置中断处理子程序地址
- (2) 打开EPC写使能、EXL使能

置1

新指令译码阶段上升沿:

- (1) EPC置旧PC+4、EXL置1
- (2) EPC写失能、EXL置1失能

(EXL寄存器为1)

中断返回

1、执行到eret语句

```
1  .text 0x00004180
2  lw $t4,0x10($gp) #Load Input Device
3  beq $t0,$t4,EQUAL
4  addu $t0,$t4,$0 #Update origin input device register
5  addu $s0,$t4,$0
6  sw $t0,0x20($gp) #Send Output Device new number
7  j CONTINUE
8  EQUAL:
9  addiu $s0,$s0,1
10 sw $s0,0x20($gp) #Send Output Device new number
11 CONTINUE:
12 sw $t0,0($gp) #Send CTRL register value, disable interrupt & timer
13 sw $t1,0x4($gp) #Send Interval
14 sw $t2,0($gp) #Send CTRL register value, enable interrupt & timer again
15 eret
16
```




2、ERET译码阶段，令下地址逻辑读EPC，EXL中断嵌套置0使能

```
76   end else if(DecInstBus[`CTLSIG_LUI]) begin
77       ALUCtl=`ALUSIG_LUI;
78   end else if(DecInstBus[`CTLSIG_J]) begin
79       NAFLCtl=`NAFLSIG_J;
80   end else if(DecInstBus[`CTLSIG_ADDI]) begin
81
82   end else if(DecInstBus[`CTLSIG_ADDIU]) begin
83
84   end else if(DecInstBus[`CTLSIG_SLT]) begin
85       ALUCtl=`ALUSIG_SLT;
86       RegWrDstCtl=`REGWRDSTSIG_RD;
87       ALUSrcCtl=`ALUSRCSIG_GPR;
88   end else if(DecInstBus[`CTLSIG_JAL]) begin
89       RegWrDstCtl=`REGWRDSTSIG_GPR_RA;
90       WrBackCtl=`WRBACKSIG_PC;
91       NAFLCtl=`NAFLSIG_JAL;
92   end else if(DecInstBus[`CTLSIG_JR]) begin
93       NAFLCtl=`NAFLSIG_JR;
94   end else if(DecInstBus[`CTLSIG_LB]) begin
95       WrBackCtl=`WRBACKSIG_MEM;
96       DataSizeCtl=`DATASIZESIG_B;
97   end else if(DecInstBus[`CTLSIG_SB]) begin
98       DataSizeCtl=`DATASIZESIG_B;
99   end else if(DecInstBus[`CTLSIG_MFC0]) begin
100       WrBackCtl=`WRBACKSIG_CP0;
101   end else if(DecInstBus[`CTLSIG_MTC0]) begin
102       CP0WrEn=`t;
103   end else if(DecInstBus[`CTLSIG_ERET]) begin
104       NAFLCtl=`NAFLSIG_EPC;
105       EXLClr=`t;
106   end else begin // DecInstBus[`CTLSIG_NOP] or Unexcepted Situations
107
108   end
109
110   if(StageIF)NAFLCtl=`NAFLSIG_PCNext; //原则：保证取指阶段下地址逻辑始终指向PC+4
111   if(StageINT && interrupt)NAFLCtl=`NAFLSIG_INT;
112 end
113
114 always@ (*) begin //使能信号的组合逻辑电路
```

3、ERET译码阶段，PC使能打开

```
32    assign StageINT= (stage==`STAGE_INT);
33
34    //但凡是转移指令，必须抢先一步在执行EXE阶段就打开PC使能，否则下一条指令取指时会取到PC+4，而非转移的指令
35    wire INST_JUMP;
36    assign INST_JUMP= DecInstBus[`CTLSIG_J] || DecInstBus[`CTLSIG_JAL] || DecInstBus[`CTLSIG_JR] || DecInstBus[`CTLSIG_JR2];
37    assign PCEn= StageIF && PCEnReg || //取指阶段PC+4
38    | StageEXE && INST_JUMP || //跳转指令的执行阶段
39    | StageDCDRF && DecInstBus[`CTLSIG_ERET] || //中断返回的译码阶段
40    | StageINT && interrupt //中断阶段且有中断
41    ;
42    assign IMEn= StageIF && IMEnReg ;
43    assign MemWrEn= StageMEM && MemWrEnReg;
44    assign RegWrEn= StageWB && RegWrEnReg;
45    assign RegOFWrEn= StageWB && RegOFWrEnReg;
46
47
```

always@ (*) begin //控制信号的组合逻辑

ALUCtl=`ALUSIG_ADD;

RegWrDstCtl=`REGWRDSTSIG_RT;

WrBackCtl=`WRBACKSIG_ALU;

ALUSrcCtl=`ALUSRCSIG_EXT;

ExtCtl=`EXTSIG_SIGN;

DataSizeCtl=`DATASIZESIG_W;

NAFLCtl=`NAFLSIG_PCNext;

CP0WrEn=`f;

EXLClr=`f;

if(DecInstBus[`CTLSIG_ADDU]) beg

RegWrDstCtl=`REGWRDSTSIG_RD;

ALUSrcCtl=`ALUSRCSIG GPR;

4、ERET的**中断阶段**上升沿:

(1) PC置原中断前
PC+4

(2) EXL寄存器置为0
新指令的译码阶段上升沿:
EXL置0信号失能

MIPS-Lite3 多周期微系统 汇编测试代码

多周期微系统 汇编 用户程序

```
mfc0 $at,$15 #PrID Store at $at
addu $gp,$0,$0
ori $gp,0x7f00 #External Device Base Address
lw $t0,0x10($gp) # Read Input Device
addu $s0,$t0,$0 # Move $s0 to $s0 to operate plus
sw $t0,0x20($gp) # Write Output Device
lui $t1,0xF
ori $t1,0x4240 #Time Interval ; When upload to FPGA,
change this
sw $t1,0x4($gp) #Send Interval
addu $t2,$0,$0
ori $t2,0x9 #TimeCounter CTRL register Value
sw $t2,0($gp) #Send CTRL register value
addu $t3,$0,$0
ori $t3,0x401 #SR Register Value
mtc0 $t3,$12 #move SR Register Value to SR
self:
j self
```

多周期微系统 汇编 内核中断处理程序

```
.text 0x00004180
lw $t4,0x10($gp) #Load Input Device
beq $t0,$t4,EQUAL
addu $t0,$t4,$0 #Update origin input device register
addu $s0,$t4,$0
sw $t0,0x20($gp) #Send Output Device new number
j CONTINUE
EQUAL:
addiu $s0,$s0,1
sw $s0,0x20($gp) #Send Output Device new number
CONTINUE:
sw $t0,0($gp) #Send CTRL register value, disable interrupt & timer
sw $t1,0x4($gp) #Send Interval
sw $t2,0($gp) #Send CTRL register value, enable interrupt & timer again
eret
```

MIPS-Lite3 多周期微系统 实机演示

FPGA演示

谢谢老师！