计算机组成原理-P4 Verilog 单周期实验报告

一、模块规格

1. IFU

取指令模块，内部包含寄存器 PC、nPC 计算部件以及指令存储器 IM，其中指示 nPC 如何计算的控制信号由 Controller 输出信号与 IFUCG (IFU Control Generator)共同确定，一些条件改变 PC 的语句由 IFUCG 判定条件是否成立。

端口定义表

信号名	方向	描述
clk	I	时钟信号
reset	I	重置信号
IFU_Control[1:0]	I	指示 nPC 如何计算，详见 Controller
IFU_Offset[15:0]	I	分支偏移量, b 系列指令
IFU_JAddr [25:0]	I	跳转目标地址，针对 J 指令
IFU_RegAddr[31:0]	I	跳转目标地址，针对 JR 等指令
Instr[31:0]	O	当前 PC 所指指令
IFU_PC_plus_4[31:0]	O	当前 PC+4 的值

功能定义表

序号	功能名称	功能描述
1	重置	reset 置 1 时，PC 寄存器复位至 0x00003000
2	顺序执行	IFU_Control == 0 时，Instr 为 PC 所指处指令，PC <= PC + 4
3	分支（branch）	Branch == 2’b01 时，Instr 为 PC 所指处指令，PC <= PC + 4 + branchOffset
4	跳转立即数	Jump == 2’b10 时，Instr 为 PC 所指处指令，PC <= (PC + 4)[31:28] || JAddress || 0^2
5	跳转寄存器	Jump == 2’b11 时，Instr 为 PC 所指处指令，PC <= IFU_RegAddr

2. IFUCG

IFU 控制信号产生模块，与 Controller 共同完成 IFU 控制，主要负责处理条件分支/跳转。

端口定义表

信号名	方向	描述
IFUCG_Mode[1:0]	I	nPC 计算控制信号
IFUCG_isConditional	I	是否为有条件跳转
IFUCG_Condition	I	指示 nPC 如何计算，详见 Controller
IFUCG_Output[1:0]	O	结合输入给出 IFU 的控制信号

功能：IFUCG_Output = ((IFUCG_isConditional && IFUCG_Condition) || (~IFUCG_isConditional))?IFUCG_Mode: (`IFU_NORMAL);

3. GRF

端口定义表

信号名	方向	描述
clk	I	时钟信号
reset	I	重置信号
GRF_WEnable	I	写使能信号
GRF_RAddr1[4:0]	I	读功能所读寄存器地址 1
GRF_RAddr2[4:0]	I	读功能所读寄存器地址 2
GRF_WAddr[4:0]	I	写功能所写寄存器地址
GRF_WData[31:0]	I	写功能写入数据
GRF_RData1[31:0]	O	读寄存器数据 1
GRF_RData2[31:0]	O	读寄存器数据 2

功能定义表

序号	功能名称	描述
1	重置	reset == 1 时重置 GRF，所有寄存器归零
2	读取寄存器数据	clk 接时钟信号时，GRF_RDdata1/2 输出寄存器编号为 GRF_RAddr1/2 的寄存器的值
3	写入寄存器	GRF_WEnable == 1 时，将 GRF_WData 处数据在下一个时钟上升沿存入编号为 GRF_WAddr 的寄存器

4. ALU

端口定义表

信号名	方向	描述
ALU_Operand1[31:0]	I	操作数 1
ALU_Operand2[31:0]	I	操作数 2（具体由 ALUOp2Datapath 元件确定）
ALU_Operation[3:0]	I	具体操作，0：加， 1：减， 2: 或， 其他待定
ALU_Result[31:0]	O	运算结果
ALU_isZero	O	结果是否为 0
ALU_Overflow	O	ADD 和 SUB 的结果是否溢出

功能定义表

序号	功能名称	描述
1	运算	根据 ALU_Operator 指定的操作进行运算，结果输出至 Result。具体操作：0：加， 1：减， 3: 或， 其他待定
2	判断结果是否为 0	运算结果为 0 时，ALU_isZero 置 1
2	判断结果是否溢出	运算结果为 0 时，ALU_Overflow 置 1

5. **DM**

端口定义表

信号名	方向	描述
clk	I	时钟信号
reset	I	重置数据，清零 RAM
DM_Addr[31:0]	I	操作地址
DM_WData[31:0]	I	待写入数据
DM_WEnable	I	是否写入数据
DM_WMode[1:0]	I	操作位宽，0 --- word, 1 --- half, 3 --- byte
DM_Rdata[31:0]	O	操作地址处的数据

功能定义表

序号	功能名称	描述
1	读取	以指定位宽读取指定位置数据
2	写入	以指定位宽在指定位置写入数据

注：对于不是以字为宽度的 IO 操作，输出采用符号延拓

6. **EXT**

端口定义表

信号名	方向	描述
EXT_Input[15:0]	I	输入数据
EXT_Mode[1:0]	I	操作模式： 0 - 无符号延拓 32， 1 - 有符号延拓 32，2 – 低 16 位移至高位
EXTOutput[31:0]	I	输出数据

功能定义表

序号	功能名称	描述
1	无符号延拓 32 位	EXT_Mode == 0 时，将 EXT_Input 做 32 位无符号延拓
2	有符号延拓 32 位	EXT_Mode == 1 时，将 EXT_Input 做 32 位有符号延拓
3	移至高 16 位	EXT_Mode == 2 时，将 EXT_Input 移至高 16 位,低 16 位补 0

二、控制器设计

1. 概述

该部分介绍 Controller 及各 MUX 的详细信息。控制器设计仍然分为两部分：对指令的识别以及指令被识别后控制信号的生成。两过程由一个 3 位的 instr_type 变量连接，每种指令（功能完全一致，数据可以不同）对应一个唯一的值。内体总体端口定义如下，具体信号功能请参见下文描述：

信号名	方向	描述
Instr[31:0]	I	指令
IFUCG_Mode	O	nPC 控制模式
IFUCG_isConditional	O	nPC 是否是条件性改写
GRF_WEnable	O	寄存器写使能信号
ALU_Operation[3:0]	O	ALU 操作信号
DM_WEnable	O	内存写使能
DM_Mode[1:0]	O	操作位宽
EXT_Mode	O	Extender 模式
MUX_RegWAddr_Sel[1:0]	O	寄存器写地址选择
MUX_RegWData_Sel[1:0]	O	寄存器写数据选择
MUX_ALUOp2_Sel[1:0]	O	ALU 操作数 2 选择信号

2. 相关各 MUX 控制信号说明

a) MUX_RegWAddr: 根据 MUX_RegWAddr_sel 的选择输出对应数据

端口定义表

信号名	方向	描述
MUX_RegWAddr_Sel[1:0]	I	选择信号：0 --- rd， 1 --- rt，2 --- $ra，其他待定
MUX_RegWAddr _rdIn[4:0]	I	rd 输入
MUX_RegWAddr _rtIn[4:0]	I	rt 输入
MUX_RegWAddr _Output[4:0]	O	输出

b) RegWData：根据 RegWData_Sel 的选择输出对应数据

端口定义表

信号名	方向	描述
MUX_RegWData_Sel[1:0]	I	选择信号：0 - alu 结果， 1 - mem 数据，2 - PC+4，其他待定
MUX_RegWData_ALUIn[31:0]	I	ALU 结果
MUX_RegWData_memIn[31:0]	I	MEM 结果
MUX_RegWData_linkIn[31:0]	I	PC+4 输入
MUX_RegWData_Output[31:0]	O	输出

c) MUX_ALUOp2: 根据 MUX_ALUOp2Data_Sel 的选择输出对应数据

端口定义表

信号名	方向	描述
MUX_ALUOp2_Sel[1:0]	I	选择信号，0 --- RegData2，1 --- EXTender，2,3 --- 待定
MUX_ALUOp2_regIn[31:0]	I	reg 数据输入
MUX_ALUOp2_EXTIn[31:0]	I	ext 数据输入
MUX_ALUOp2_Output[31:0]	O	输出

3. 指令识别模块

指令识别由一个 always 块描述的组合逻辑实现。在 case(Instr[31:26])和 case(Instr[5:0])内，为 instr_type 赋对应当前指令的值，真值表如下：

功能：译码 --- 当检测到某指令时 instr_type 输出其对应信号

OpCode	Funct	Instr_type
000000	000000	INSTR_NOP 32'b0
000000	100001	INSTR_ADDU 32'b1
000000	100011	INSTR_SUBU 32'b10
001101	xxxxxx	INSTR_ORI 32'b100
100011	xxxxxx	INSTR_LW 32'b1000
101011	xxxxxx	INSTR_SW 32'b10000
000100	xxxxxx	INSTR_BEQ 32'b100000
001111	xxxxxx	INSTR_LUI 32'b1000000
000011	xxxxxx	INSTR_JAL 32'b10000000
000000	001000	INSTR_JR 32'b100000000

4. ControlSignal

端口定义表及信号介绍

信号名	方向	描述
IFUCG_Mode	O	nPC 控制模式, 0 – 顺序，1 – 分支，2 – 跳转， 3 – 跳至寄存器
IFUCG_isConditional	O	nPC 是否是条件性改写。 1：IFUCG 只有在 Condition 为 1 时 Output 为 Mode，否则为 2’b00 2：IFUCG 的 Output 始终为 IFUCG_Mode
GRF_WEnable	O	寄存器写使能信号，0：GRF 不写入，1：GRF 写入
ALU_Operation[3:0]	O	ALU 操作信号，0：加，1：减，2：或
DM_WEnable	O	内存写使能，0：DM 不写入，1：DM 写入
DM_Mode[1:0]	O	操作位宽，0：字，1：半字，2：字节
EXT_Mode	O	Extender 模式，0：无符号拓展至 32 位，1：有符号拓展至 32 位， 2：低 16 位移至高位
MUX_RegWAddr_Sel[1:0]	O	寄存器写地址选择，0：rd，1：rt，2：$ra
MUX_RegWData_Sel[1:0]	O	寄存器写数据选择，0：ALU，1：Mem，2：PC+4
MUX_ALUOp2_Sel[1:0]	O	ALU 操作数 2 选择信号，0：rt，1：EXT

功能：根据输入指令信号输出对应控制信号

信号	nop	addu	subu	ori	lw	sw	beq	lui	jal	jr
IFUCG_Mode	0	0	0	0	0	0	1	0	2	3
IFUCG_isConditional	0	0	0	0	0	0	1	0	0	0
GRF_WEnable	0	1	1	1	1	0	0	1	1	0
ALU_Operation[3:0]	x	0	1	2	0	1	1	2	x	x
DM_WEnable	0	0	0	0	0	1	0	0	0	0
DM_Mode[1:0]	x	x	x	x	0	0	x	x	x	x
EXT_Mode	x	x	x	0	1	1	x	2	x	x
MUX_RegWAddr_Sel[1:0]	x	0	0	1	1	x	x	1	2	x
MUX_RegWData_Sel[1:0]	x	0	0	0	1	x	x	0	2	x
MUX_ALUOp2_Sel[1:0]	x	0	0	1	1	1	0	1	x	x

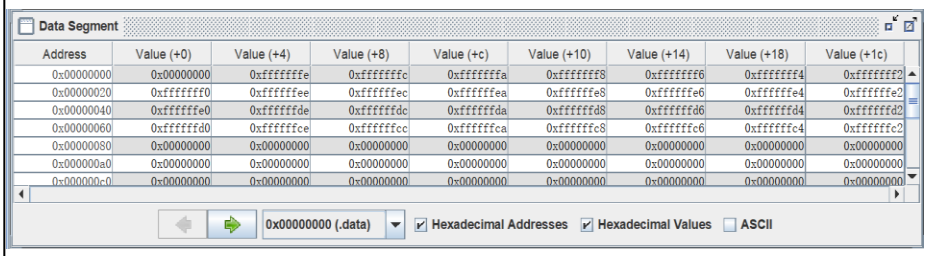
三、测试程序

以下为测试程序及期待结果，使用时机器码需去掉前两行

```
lui $28, 0
lui $29, 0
# clear, for simulation in mars
ori $1, $0, 0x1234
lui $1, 0x1078
# check if lui set lower to 0
ori $1, $1, 0x1234
# function of ori
addu $0, $1, $1
addu $2, $0, $1
addu $2, $1, $1
# check if $0 stays at 0 and addu
subu $2, $2, $1
# func of subu
ori $3, $0, 5
# check if DM is reset to 0
addu $2, $3, $2
lw $4, 4($0)
sw $2, 3($3)
# lw/sw check
# beq/jal/jr check
beq $0, $3, branch
addu $5, $5, $3
addu $5, $5, $3
branch:
addu $5, $5, $3
beq $0, $0, branch2
addu $7, $5, $3
branch2:
addu $5, $5, $3
jal jump
addu $6, $5, $3
addu $6, $5, $3
jal end
jump:
lui $7, 0
lui $8, 0
lui $9, 0
lui $10, 0
ori $8, 0x0004
ori $9, 0x0002
lui $23, 0
ori $23, 0x80
loop:
sw $10, ($7)
addu $7, $7, $8
subu $10, $10, $9
beq $7, $23, end_loop
beq $0, $0, loop
end_loop:
jr $ra
end:
```

Registers | Coproc 1 | Coproc 0

Name	Number	Value
$zero	0	0x00000000
$at	1	0x10781234
$v0	2	0x10781239
$v1	3	0x00000005
$a0	4	0x00000000
$a1	5	0x00000014
$a2	6	0x00000019
$a3	7	0x00000080
$t0	8	0x00000004
$t1	9	0x00000002
$t2	10	0xfffffffc0
$t3	11	0x00000000
$t4	12	0x00000000
$t5	13	0x00000000
$t6	14	0x00000000
$t7	15	0x00000000
$s0	16	0x00000000
$s1	17	0x00000000
$s2	18	0x00000000
$s3	19	0x00000000
$s4	20	0x00000000
$s5	21	0x00000000
$s6	22	0x00000000
$s7	23	0x00000080
$t8	24	0x00000000
$t9	25	0x00000000
$k0	26	0x00000000
$k1	27	0x00000000
$gp	28	0x00000000
$sp	29	0x00000000
$fp	30	0x00000000
$ra	31	0x00003060
pc		0x00003098
hi		0x00000000
lo		0x00000000



四、思考题

1. L0.T2.1 **根据你的理解，在下面给出的 DM 的输入示例中，地址信号 addr 位数为什么是[11:2]而不是[9:0]？这个 addr 信号又是从哪里来的?**

文件	模块接口定义

dm.v	dm(clk,reset,MemWrite,addr,din,dout);
	input clk; //clock
	input reset; //reset
	input MemWrite; //memory write enable
	input [11:2] addr; //memory's address for write
	input [31:0] din; //write data
	output [31:0] dout; //read data

因为地址信号一定是 4 的倍数（按字对齐），所以后两位必定为 2'b00。而同时 addr[11:2]来源为 ALU 运算结果的第 2-11 位

2. L0.T2.2 **在相应的部件中，reset 的优先级比其他控制信号（不包括 clk 信号）都要高，且相应的设计都是同步复位。清零信号 reset 是针对哪些部件进行清零复位操作？这些部件为什么需要清零?**

reset 针对 DM 的数据，IFU 中的 PC 寄存器以及 GRF 中各寄存器进行复位。PC 寄存器复位最为重要，它保证 CPU 复位后读取 IM 的第一条指令，从头开始执行程序。CPU 上电时，GRF 各寄存器的值以及 DM 中各内存单元的值一般都为 0。GRF 复位可以防止复位前各寄存器的值产生的影响。DM 复位清空内存信息，防止内存中已有数据产生影响。

3. L0.T4.1 **列举出用 Verilog 语言设计控制器的几种编码方式（至少三种），并给出代码示例。**

a) 利用 case 语句，指令识别与控制信号生成同时进行

```
case(Instr[31:26])
    6'b000000: begin
        case(Instr[5:0])
            `INSTR_NOP: begin
                /*Control Signal Assignment*/
            end
        endcase
    end
    `INSTR_LW: begin
        /*Control Signal Assignment*/
    end
endcase
```

b) 利用 assign 语句，模拟与或门阵列

```
assign nop = (opcode == 6'b000000);
/*Instruction Signal*/
assign RegWEnable = (addu || ori || subu)?1:0;
assign EXTMode = (ori)?2'b00:(beq?2'b01:2'b11);
/*Control Signal*/
```

c) 结合 assign 语句和 if/else

```
assign nop = (opcode == 0);

if(nop) begin
    /*Control Signal*/
end
else if(lw) begin
    /*Control Signal*/
end
```

d) 使用两个 case 语句，指令识别和控制信号生成分开进行

```
case(opcode)
    6'b000000: begin
        case(funct)
            6'b000000: instr_type = `INSTR_NOP;
            6'b001000: instr_type = `INSTR_JR;
            6'b100001: instr_type = `INSTR_ADDU;
            6'b100011: instr_type = `INSTR_SUBU;

            default: instr_type = `INSTR_NOP;
        endcase
    end

    6'b000011: instr_type = `INSTR_JAL;
```

```
case(instr_type)
    `INSTR_ADDU: begin
        IFUCG_Mode = `IFU_NORMAL;
        IFUCG_isConditional = 1'b0;
        GRF_WEnable = 1'b1;
        EXT_Mode =  `EXT_UNSIGNED;  // useless
        ALU_Operation = `ALU_ADD;
        DM_Mode = `DM_WORD;          // useless
        DM_WEnable = 1'b0;           // useless
        MUX_ALUOp2_Sel = `MUX_ALUOP2_REGSEL;
        MUX_RegWAddr_Sel = `MUX_REGWADDR_RDSEL;
        MUX_RegWData_Sel = `MUX_REGWDATA_ALUSEL;
    end
```

4. L0.T4.2 **根据你所列举的编码方式，说明他们的优缺点。**

a）方式下编码量小，便于书写，但是维护、查找 bug 以及添加新指令时会产生困难，可以利用宏使其更加清晰。b）方式实际上是直接 Logisim 电路图转换到 Verilog 的过程，在 P3 的基础上思维量较小，但随着指令数增加会变得更加麻烦。c）d）都是将指令的识别和控制信号的生成分开，区别在于 c 中采用 if/else if 形式编码可能在综合时形成优先级树，影响性能。同时过长的 if/else if 序列也会影响可读性。

5. L0.T5.1 **C 语言是一种弱类型程序设计语言。C 语言中不对计算结果溢出进行处理，这意味着 C 语言要求程序员必须很清楚计算结果是否会导致溢出。因此，如果仅仅支持 C 语言，MIPS 指令的所有计算指令均可以忽略溢出。 请说明为什么在忽略溢出的前提下，addi 与 addiu 是等价的，add 与 addu 是等价的。提示：阅读《MIPS32® Architecture For Programmers Volume II: The MIPS32® Instruction Set》中相关指令的 Operation 部分 。**

通过阅读指令手册我们可以发现，add/addi 相对于 addu/addiu 在算术部分操作相同：add/addi 相对于 addu/addiu 在计算时多出一位作为溢出的检验，但低 32 位操作相同，存入寄存器（如果不溢出）的结果也就相同。二者的区别只有在发生溢出时 addi/add 会触发异常而 addu/addiu 不会。

6. L0.T5.2 **根据自己的设计说明单周期处理器的优缺点。**

优点：设计简单，结构简单，时钟统一。

缺点：所有指令在同一周期完成，时钟长度受限于最慢的一条指令。元件利用率不高，效率低。指令吞吐量低。指令存储器 IM 和数据存储器 DM 相互独立，不符合现实

7. L0.T4.2 **简要说明 jal、jr 和堆栈的关系。**

jal 和 jr 配套使用，完成函数调用及返回功能。利用 jal 调用时$ra 会被当前地址覆盖，因此调用函数前要先将返回地址以及需要保存的值存在栈上，栈向下缩减。程序定义的变量，寄存器中保存不下的那些保存在堆中，堆从下向上增长。利用 jr 返回后，应从栈中把存储的数据释放出来，栈向上恢复。堆中内容因为调用已经结束故不再有用，可以被覆盖（堆向下释放空间）。