Logisim单周期CPU设计文档

一、 CPU设计方案综述

（一） 总体设计概述

使用Logisim开发一个简单的单周期CPU，总体概述如下：

1. 此CPU为32位CPU
2. 此CPU为单周期设计
3. 此CPU支持的指令集为：

   {addu, subu, ori, lw, sw, beq, lui, nop,andi,jal,j,jr,sll}
4. nop机器码为0x00000000
5. addu, subu不支持溢出

（二） 关键模块定义

1. IFU

（1） 端口说明

表1-IFU端口说明

序号	信号名	方向	描述
1	clk	I	时钟信号
2	reset	I	异步复位信号，将PC值置为0x000000000：无效1：复位
3	branch[1:0]	I	判断转移类型0:PC+41:beq2:26位扩展3：jr指令指定寄存器
4	eq	I	A1、A2对应GRF两个寄存器中的值是否相等（是否满足beq的跳转要求）0：不相等1：相等
5	imm32[31:0]	I	32位的扩展后的立即数（16位或26位）
6	instr[31:0]	O	将IM中，要执行的指令输出
7	PC	O	满足jal指令中PC+4的存储

（2） 功能定义

表2-IFU功能定义

序号	功能	描述
1	复位	reset有效时，PC置为0x00000000
2	更新PC的值	branch为00：PC = PC + 4 branch为01： eq为0: PC = PC + 4 +sign_extend(offset||02) eq为1：PC = PC + 4 branch10: PC=PC31..28||instr_index||02 branch为11:PC = GPR[rs]
3	输出指令	根据PC的值，取出IM中的指令

2. GRF

（1） 端口说明

表3-GRF端口说明

序号	信号名	方向	描述
1	clk	I	时钟信号
2	reset	I	异步复位信号，将32个寄存器中全部清零1：清零0：无效
3	WE	I	写使能信号1：可向GRF中写入数据0：不能向GRF中写入数据
4	A1[4:0]	I	5位地址输入信号，指定32个寄存器中的一个，将其中存储的数据读出到RD1
5	A2[4:0]	I	5位地址输入信号，指定32个寄存器中的一个，将其中存储的数据读出到RD2
6	A3[4:0]	I	5位地址输入信号，指定32个寄存器中的一个，作为RD的写入地址
7	WD[31:0]	I	32位写入数据
8	RD1[31:0]	O	输出A1指定的寄存器的32位数据
9	RD2[31:0]	O	输出A2指定的寄存器的32位数据

（2） 功能定义

表4-GRF功能定义

序号	功能	描述
1	异步复位	reset为1时，将所有寄存器清零
2	读数据	将A1和A2地址对应的寄存器的值分别通过RD1和RD2读出
3	写数据	当WE为1且时钟上升沿来临时，将WD写入到A3对应的寄存器内部

3. ALU

(1) 端口说明

表5-ALU端口说明

序号	信号名	方向	描述
1	A[31:0]	I	参与运算的第一个数
2	B[31:0]	I	参与运算的第二个数
3	ALUop[2:0]	I	决定ALU做何种操作000：无符号加001：无符号减010：与011：或100：左移位运算
4	eq	O	A与B是否相等0：不相等1：相等
5	res	O	A与B做运算后的结果

(2) 功能定义

表6-ALU功能定义

序号	功能	描述
1	加运算	res = A + B
2	减运算	res = A - B
3	与运算	res = A & B
4	或运算	res = A | B
5	左移位运算	Res=A<<B

4. DM

(1) 端口说明

表7-DM端口说明

序号	信号名	方向	描述
1	clk	I	时钟信号
2	reset	I	异步复位信号0：无效1：内存值全部清零
3	WE	I	写使能信号0：禁止写入1：允许写入
4	Address[4:0]	I	读取或写入信号地址
5	WD[31:0]	I	32为写入数据
6	RD[31:0]	O	32位读出数据

(2) 功能定义

表8-DM功能定义

序号	功能	描述
1	异步复位	当reset为1时，DM中所有数据清零
2	写入数据	当WE有效时，时钟上升沿来临时，WD中数据写入A对应的DM地址中
3	读出数据	RD永远读出A对应的DM地址中的值

5. EXT

(1) 端口说明

表9-EXT端口说明

序号	信号名	方向	描述
1	imm16[15:0]	I	代扩展的16位信号
2	sign	I	无符号或符号扩展选择信号0：无符号扩展1：符号扩展
3	imm32[31:0]	O	扩展后的32位的信号

(2) 功能定义

表10-EXT功能定义

序号	功能	描述
1	无符号扩展	当sign为0时，将imm16无符号扩展输出
2	符号扩展	当sign为1时，将imm16符号扩展输出

6. Controller

(1) 端口说明

表11-Controller端口说明

序号	信号名	方向	描述
1	op[5:0]	I	instr[31:26]6位控制信号
2	func[5:0]	I	instr[5:0]6位控制信号
3	AluOp[2:0]	O	ALU的控制信号
4	WeGrf	O	GRF写使能信号0：禁止写入1：允许写入
5	WeDm	O	DM的写入信号0：禁止写入1：允许写入
6	branch	O	PC转移位置选择信号
7	AluSrc1[1:0]	O	参与ALU运算的第一个数00：RD101：RD2
8	AluSrc2[1:0]	O	参与ALU运算的第二个数，来自GRF还是imm00：RD201：imm3210: offset
9	WhichtoReg[1:0]	O	将何种数据写入GRF？00：ALU计算结果01：DM读出信号10：upperImm11: PC+4
10	RegDst[1:0]	O	GRF写入地址选择信号0：Rd1：Rt2：31号寄存器
11	SignExt	O	是否对imm16进行符号扩展0：不进行符号扩展1：进行符号扩展

真值表

端口	addu	subu	ori	lw	sw	lui	beq
op	000000	000000	001101	100011	101011	001111	000100
func	100001	100011					
AluOp	000	001	011	000	000	000	000
WeGrf	1	1	1	1	0	1	0
WeDm	0	0	0	0	1	0	0
branch	00	0	00	00	00	00	01
AluSrc1	00	00	00	00	00	00	00
AluSrc2	00	00	01	01	01	00	00
WhichtoReg	00	00	00	01	00	10	00
RegDst	00	00	01	01	01	01	01
SignExt	0	0	0	1	1	0	1
端口	**andi**	**jal**	**j**	**jr**	**sll**	**add**	**sub**
op	001100	000011	000010	000000	000000	000000	000000
func				001000	000000	100000	100010
AluOp	010	000	000	000	100	000	001
WeGrf	1	1	0	0	1	1	1
WeDm	0	0	0	0	0	0	0
branch	00	10	10	11	00	00	0
AluSrc1	00	00	00	00	01	00	00
AluSrc2	01	00	00	00	10	00	00
WhichtoReg	00	11	00	00	00	00	00
RegDst	01	10	00	00	00	00	00
SignExt	1	0	0	0	0	0	0

二、 测试方案

（1） 测试代码：

```
.text
    ori     $t1,v0, 100  # t1: 100
    ori     $t2,v0, 250  # t2: 250
    ori     $t3,v0, 200  # t3: 200
    add     $t4,t1, $t3
    # t4 = t1 + t3 (t4 = 300)
    sub     $t5,t4, $t2
```

```
    # t5 = t4 - t2 (t5 = 50)
    sw      $t5, 4(v0)
    lw      $t6, 4(v0)
Beq:
    lui     $t7, 100
    beq     $t5,t1, Beq
    beq     $t6,t5, Beq
```

(2) MARS中运行结果

Registers | Coproc 1 | Coproc 0

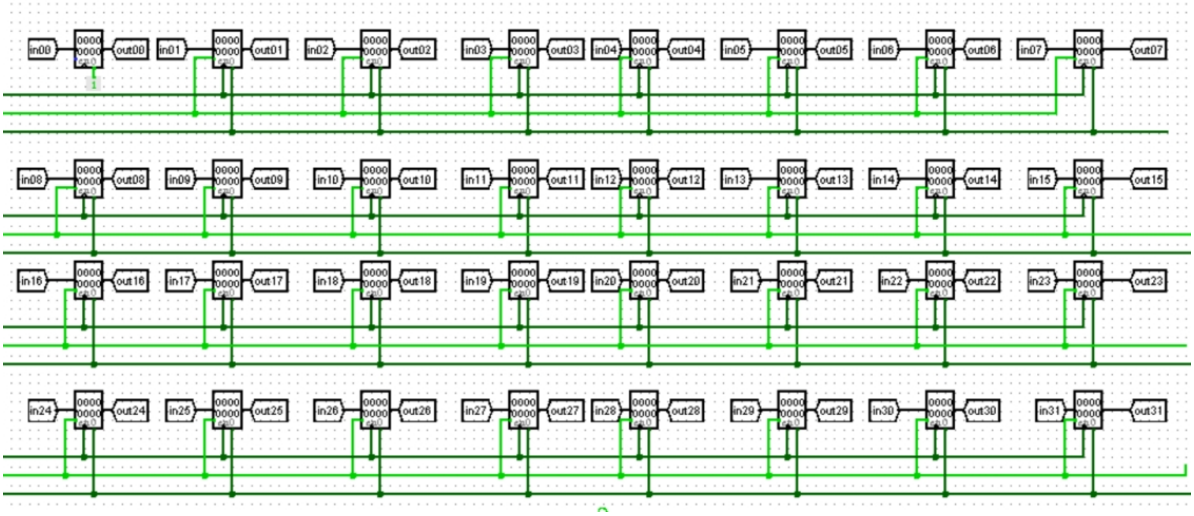
Name	Number	Value
$zero	0	0
$at	1	0
$v0	2	0
$v1	3	0
$a0	4	0
$a1	5	0
$a2	6	0
$a3	7	0
$t0	8	0
$t1	9	100
$t2	10	250
$t3	11	200
$t4	12	300
$t5	13	50
$t6	14	50
$t7	15	6553600
$s0	16	0
$s1	17	0
$s2	18	0
$s3	19	0
$s4	20	0
$s5	21	0
$s6	22	0
$s7	23	0

$t8	24	0
$t9	25	0
$k0	26	0
$k1	27	0
$gp	28	6144
$sp	29	12284
$fp	30	0
$ra	31	0
pc		12324
hi		0
lo		0

Data Segment

Address	Value (+0)	Value (+4)	Value (+8)	Value (+12)	Value (+16)	Value (+20)	Value (+24)	Value (+28)
0	0	50	0	0	0	0	0	0
32	0	0	0	0	0	0	0	0
64	0	0	0	0	0	0	0	0
96	0	0	0	0	0	0	0	0
128	0	0	0	0	0	0	0	0

（3） 该CPU运行结果



00	00000000	00000032	00000000	00000000	00000000	00000000	00000000	00000000
08	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
10	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
18	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000

三、 思考题

（一）上面我们介绍了通过 FSM 理解单周期 CPU 的基本方法，请大家画出单周期 CPU 对应有限状态机的状态转移图，并谈谈它和我们之前见过的状态转移图有什么不同。

我认为单周期CPU的主要的时序模块就是PC模块，而跟之前所学的状态机相比，我觉得PC模块更像是一个Mealy状态机和Moore状态机的结合体。

1).在立即数没有产生作用时（大多位R型指令），PC会自动加4达到下一状态，这时的CPU如同一个Moore状态机。

2).在b型和J型跳转指令的前提下，PC下一状态的值取决于PC的值和imm的输入，所以此时CPU为一个Mealy型的状态机。

3).CPU还有一个特殊的指令——jr指令，改指令的输出取决之前grf中31号寄存器存的地址和imm的输入，和PC的原状态没有任何关系，此时的CPU已经脱离的状态机的范畴。

（二） 现在我们的模块中IM使用ROM， DM使用RAM， GRF使用Register，这种做法合理吗？ 请给出分析，若有改进意见也请一并给出。

这种做法是合理的。

1).IM只需被读取，ROM只有读取功能；

2).DM既要进行读取，又要进行写入，但是一个周期只会进行读取和写入之一，RAM的单一地址和各一个的读写端口满足了这种要求，当然，用寄存器也能实现DM，但是DM需要较大的空间，使用寄存器太“浪费”；

3).GRF需要读写，且其与ALU直接连接，需要高速地读写，故使用寄存器堆搭建合理。

（三）在上述提示的模块之外，你是否在实际实现时设计了其他的模块？如果是的话，请给出介绍和设计的思路。

我在实际实现IFU模块时采用了将IFU分成PC和IM两个小模块的方法来实现的，这样更有利于两者各司其职，从而达到更好的设计效果。

（四） 事实上，实现nop空指令，我们并不需要将它加入控制信号真值表，为什么？请给出你的理由。

nop指令，整个CPU只执行PC<=PC+4指令，对于PC而言，在没有任何指令操作的前提下会自动执行PC<=PC+4，一段指令中，加与不加，执行结果没有区别。

（五） 上文提到，MARS不能导出PC与DM起始地址均为0的机器码。实际上，可以通过为DM增添片选信号，来避免手工修改的麻烦，请查阅相关资料进行了解，并阐释为了解决这个问题，你最终采用的方法。

如果寄存器中存储的DM的地址被映射在0x3000_0000到0x3fff_ffff间，而我们的DM起始地址是0，那么，我们可以将输入地址直接减去0x3000_0000，再作为DM的地址输入。

假如我们不确定寄存器中的存储的DM地址的起始值，我们可以将其与0x3000_0000比较，得到片选信号。

（六）阅读 Pre 的 “MIPS 指令集及汇编语言” 一节中给出的测试样例，评价其强度（可从各个指令的覆盖情况，单一指令各种行为的覆盖情况等方面分析），并指出具体的不足之处。

由于CPU的基础设计范围没有设计J型指令，所以对于指令设计本身的难度并不能做到很大，但就测试样例而言，在只有{add,sub,ori,nop,beq,lui}的这个前提下，这个测试样例基本的达到了常见的指令的覆盖，如正负，不同符号位等等，但调用过程中由于beq的位置相对靠后，所以测试跳转指令达到的效果并不好。