CPU设计方案综述

（一）总体设计概述

1. 处理器为 单周期32位处理器。
2. 处理器支持的指令集为：{`addu`, `subu`, `ori`, `lw`, `sw`, `beq`, `lui`, `nop`}。
3. `nop` 机器码为 `0x00000000`， 即空指令，不进行任何有效行为（修改寄存器等）
4. 需要采用模块化和层次化设计。顶层有效的驱动信号仅包括 `reset` 。

（二）关键模块定义

1. GRF

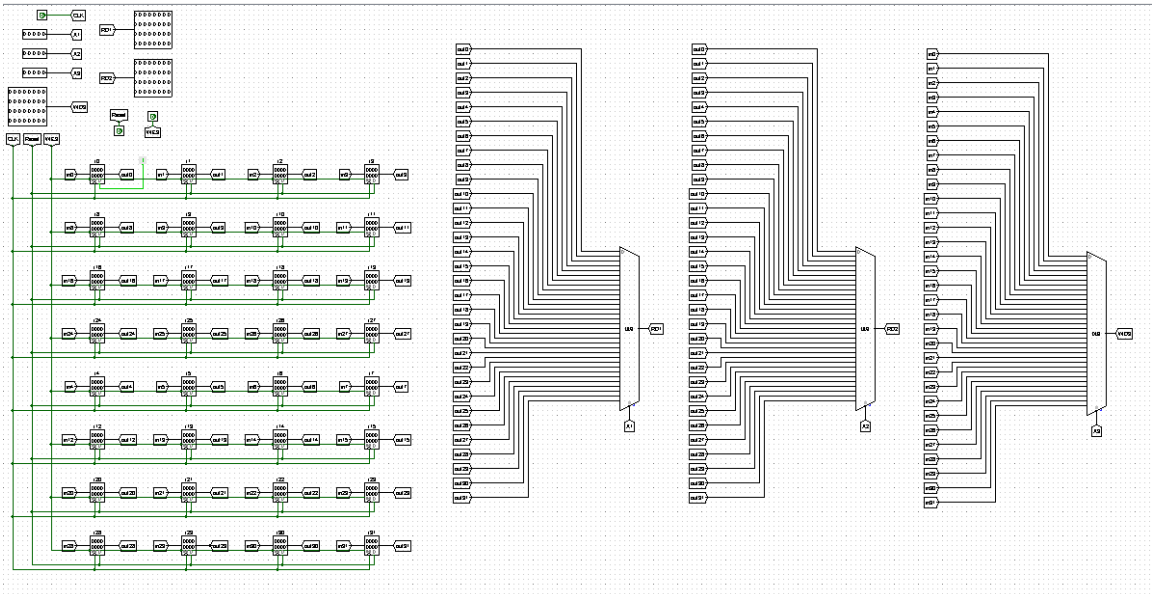
模块接口

信号名	方向	功能描述
CLK	I	时钟信号(Clock)
Reset	I	复位信号(Reset)
WE3	I	写入控制信号(随时都可以读出)(WriteEnable)
A1[4:0]	I	读寄存器地址1(rs_Address1)
A2[4:0]	I	读寄存器地址2(rt_Address2)
A3[4:0]	I	写寄存器地址3(rd/rt_Address2)
WD3[31:0]	I	待写入数据(WriteData)
RD1[31:0]	O	读出数据1(rs)
RD2[31:0]	O	读出数据2(rt)

功能定义

序号	功能名称	功能描述
1	复位	异步复位，调试用
2	读寄存器	不受时钟上升沿控制，输出指定序号寄存器内当前存储值
3	写寄存器	受时钟上升沿与 `WE3` 信号控制，当 `WE3 = 1` 且处于时钟上升沿时，`A3` 指定序号寄存器将 `WD3` 值写入

- 设计注意事项：
  - 使用 `Demultiplexer` 选择传递写入数据 `WD3` 时，需设置'Three State' 选项，使未被选择的其他31条通路处于 `floating` 状态，这样可以使目标寄存器存入值时，其余寄存器的内容不受影响。
  - 或者，也可以将 `WD3` 值直接接入给所有寄存器，将写入使能信号 `WE3` 通过 `DMX` 传递给各寄存器的 `En` 使能端。上述操作也可以达到相同的目的。



2. ALU

模块接口

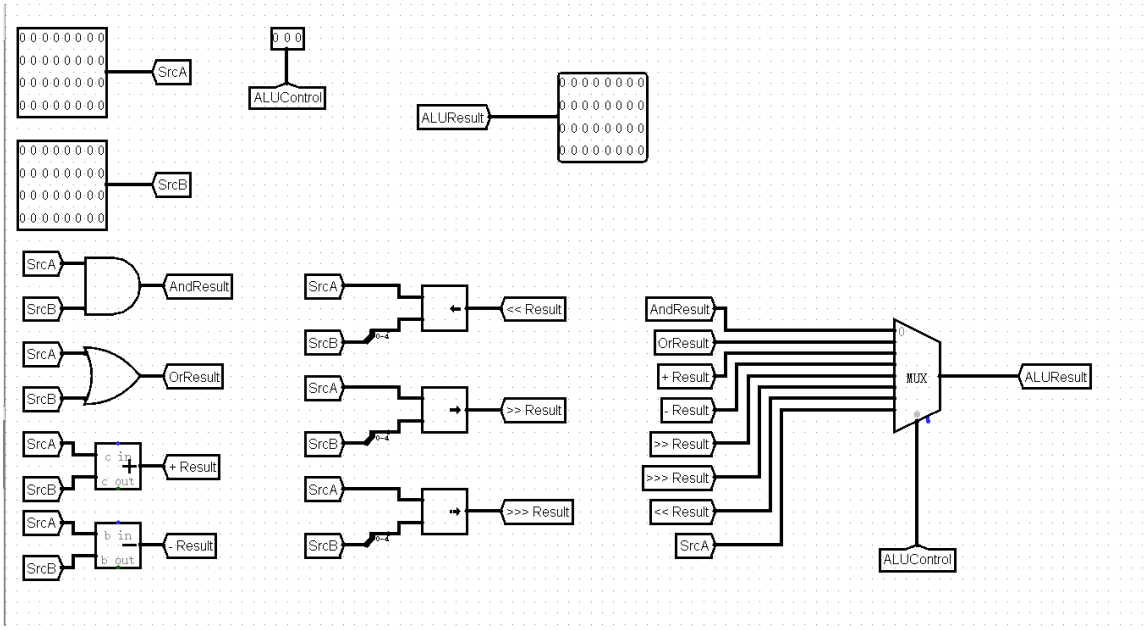
信号名	方向	功能描述

信号名	方向	功能描述
ALUControl[2:0]	I	ALU控制信号(功能与信号具体对应关系见下表)
SrcA[31:0]	I	运算因子A
SrcB[31:0]	I	运算因子B
ALUResult[31:0]	O	计算结果

功能定义

序号	功能名称	功能描述
1	& (000)	A & B
2	|	(001)
3	+ (010)	A + B
4	- (011)	A - B
5	>> (100)	A >> B
6	>>> (101)	A >>> B
7	<< (110)	A << B
8	A (111)	A

- 关于 Zero 端口，笔者也并没有将其加入 ALU 的一部分，而是单独建设了一个 BranchIf 模块， 便于 Branch 类系列指令的实现。 具体思路即，将 beqbne``bgtzbltz 四条指令从 CU 单独引出，并且将四条指令对应 ALUControl[2:0]输出均定为 011(-)。Branchif模块中利用组合逻辑输出是否跳转指令。



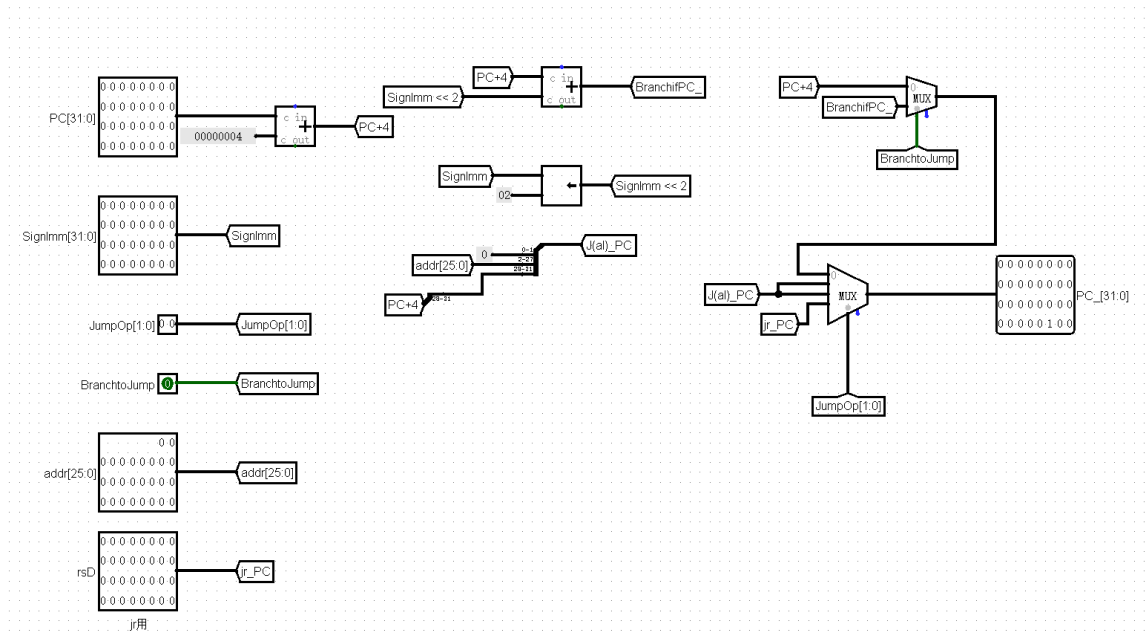
3. NPC

模块接口

信号名	方向	功能描述
PC[31:0]	I	PC_CurrentValue
SignImm[31:0]	I	Immediate_SignExtended
JumpOp[1:0]	I	JumpOperator(详情参见 CU部分)
BranchtoJump	I	Output_of_BranchIf_Unit
addr[25:0]	I	Instruction的后26位
rsD[31:0]	I	Output_of_RD1_in_GRF([rs])(jr 指令用)
PC_[31:0]	O	PC_NextStateValue

功能定义

序号	功能名称	功能描述
1	Default (JumpOp == 00 && BranchtoJump == 0)	PC_ = PC + 4
2	beq/bne/bgtz/bltz (JumpOp == 00 && BranchtoJump == 1)	PC_ = PC + 4 + (SignImm << 2)
3	j/jal (JumpOp == 01|| JumpOp == 10)	PC_ = {(PC + 4)[31 : 28], addr[25 : 0], 2'b0}
4	jr (JumpOp == 11)	PC_ = [rs]



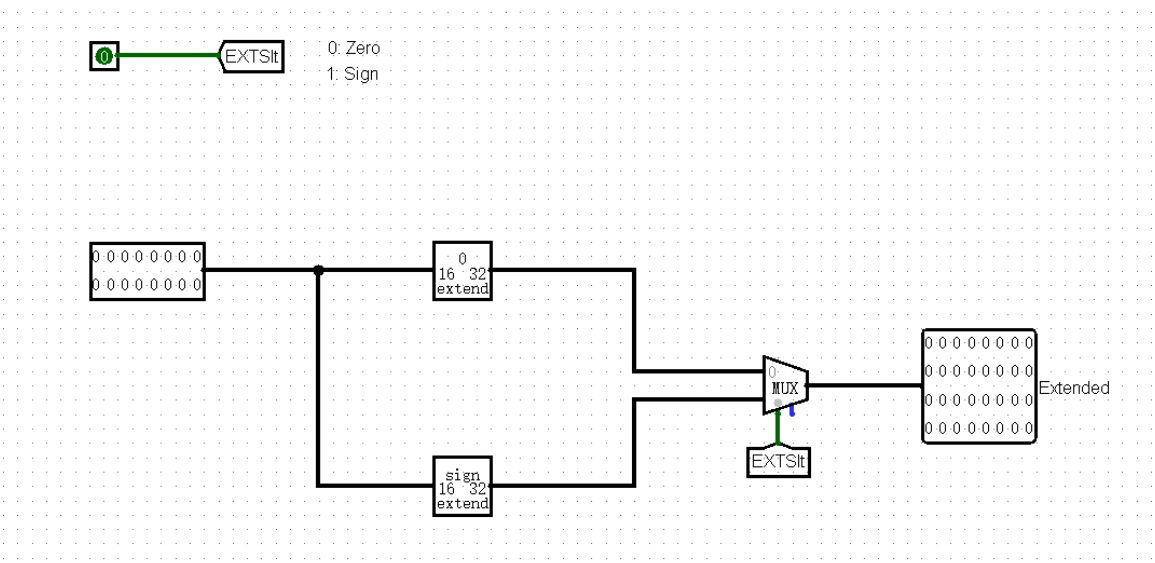
4. EXT

模块接口

信号名	方向	功能描述
EXTOp	I	EXTOperator
Imm[15:0]	I	Immediate
ExtendedImm[31:0]	O	Extended_Immediate

功能定义

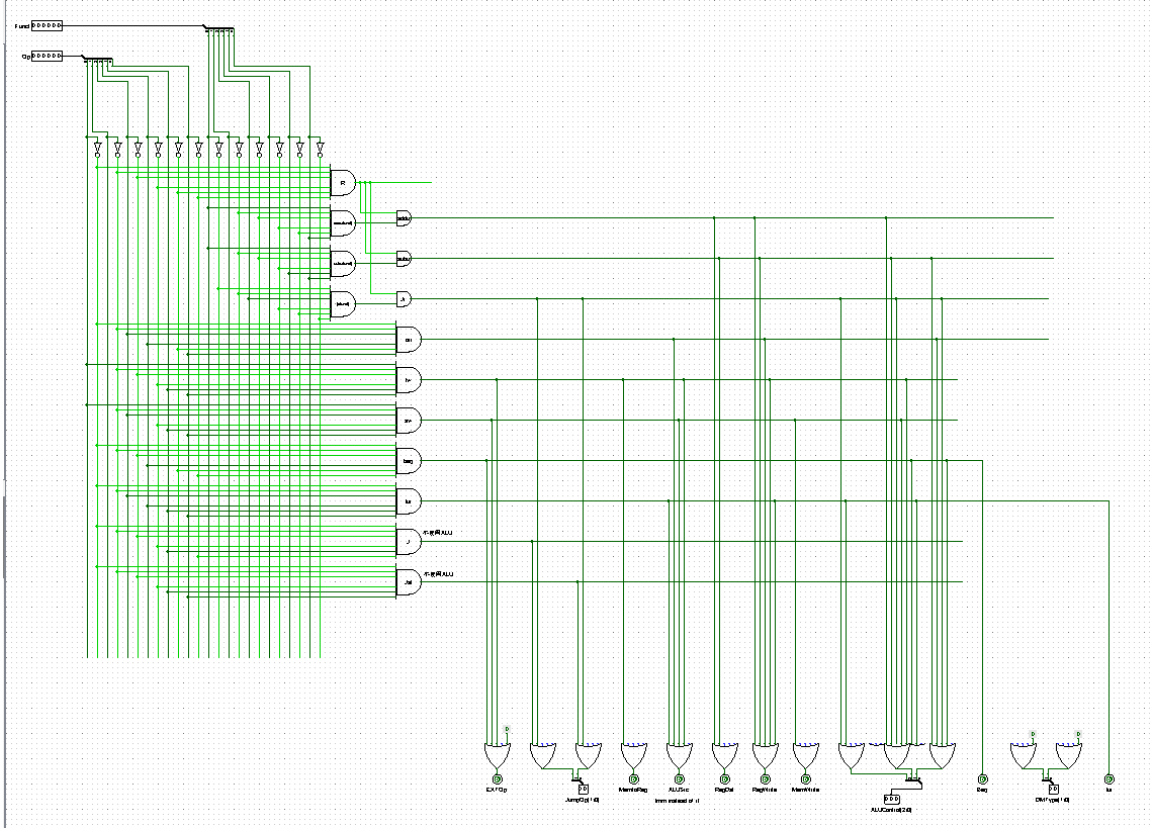
序号	功能名称	功能描述
1	ZeroExtendImmediate	ExtendedImm = {16'b0, Imm}
2	SignExtenedImmediate	ExtendedImm = {16{Imm[15]}, Imm}



5. CU

模块接口

信号名	方向	功能描述
Op[5:0]	I	Operator, Instruction[31:25]
Funct[5:0]	I	Function, Instruction[5:0]
EXTOp	O	ExtenderOperator
JumpOp[1:0]	O	JumpOperator
MemtoReg	O	Memory_to_register_Multiplexer_Op
ALUSrc	O	Control_the_source_of_SrcB
RegDst	O	Write_Data_to_`rs` *or* `rt`
RegWrite	O	WriteEnable_of_GRF
MemWrite	O	WriteEnable_of_DM
ALUControl[2:0]	O	Details can be Reffered to in `ALU` Unit
Beq	O	set 1 if the instruction is `beq` , Input of `Branchif` Unit
DMType[1:0]	O	用于处理 `lw`、`sw`、`lh`、`sh`、`lb`、`sb`等指令
lui	O	set 1 if the instruction is `lui`



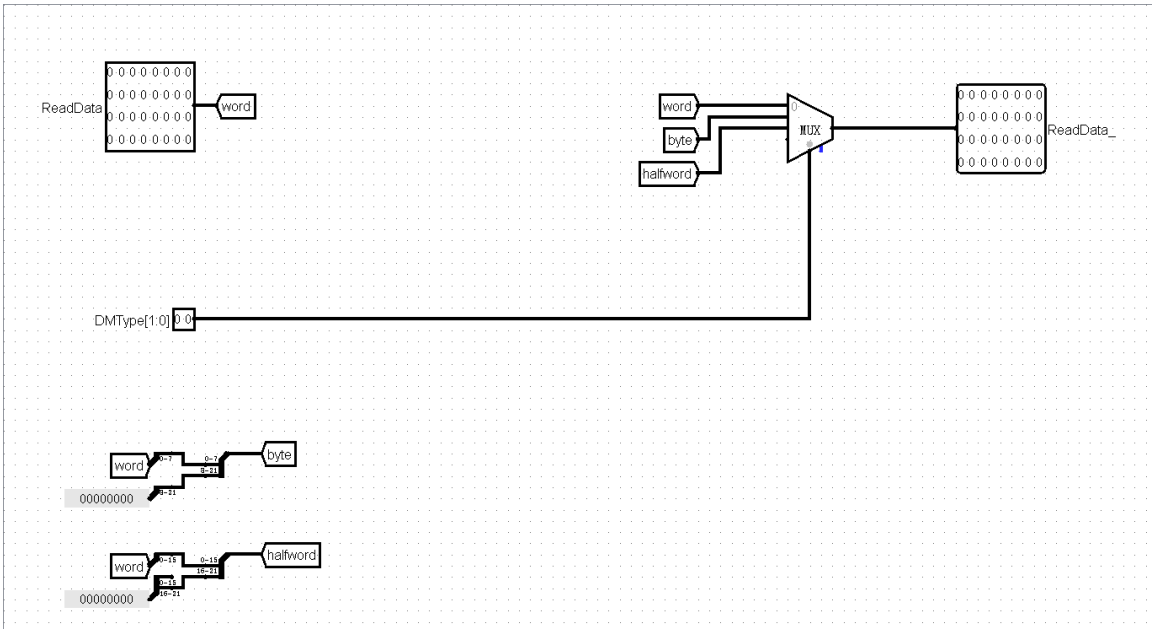
6. DMPreReading

模块接口

信号名	方向	功能描述
ReadData[31:0]	I	RD_of_DM
DMType[1:0]	I	控制元件功能
ReadData_[31:0]	O	Operated_RD

功能定义

序号	功能名称	功能描述
1	Transform_to_word(DMType == 00)	ReadData_ = ReadData
2	Transform_to_Byte(DMType == 01)	ReadData_ = {24'b0, ReadData[7:0]}
2	Transform_to_halfword(DMType == 10)	ReadData_ = {16'b0, ReadData[15:0]}



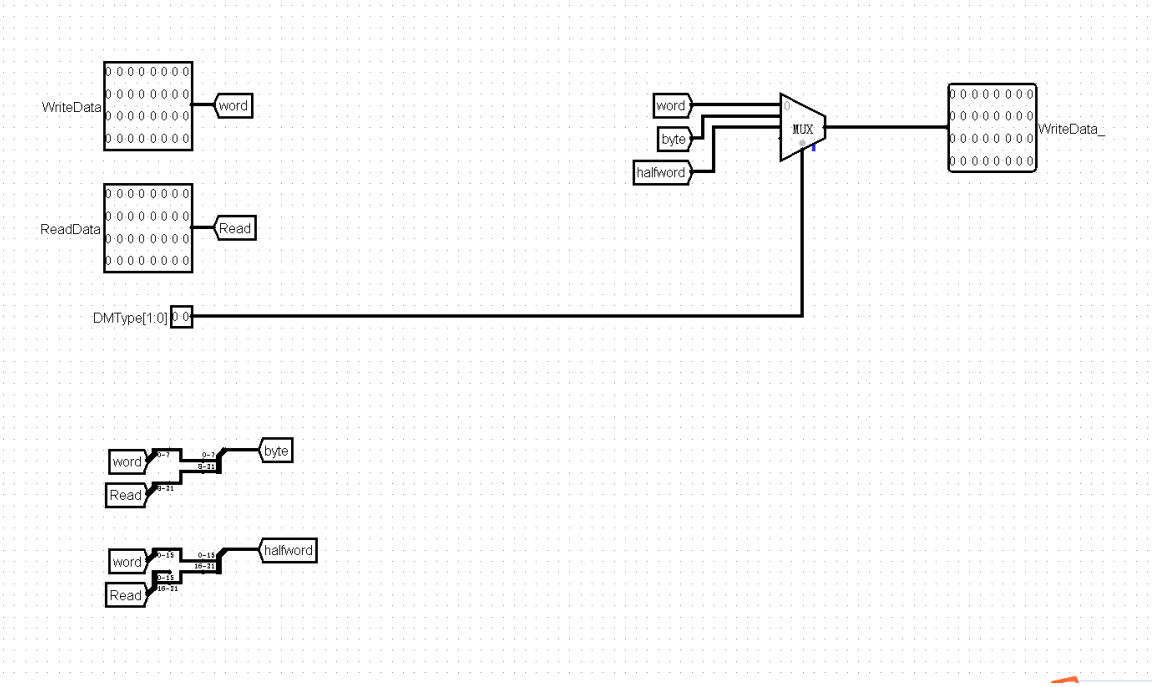
7. DMPreWriting

模块接口

信号名	方向	功能描述
WriteData[31:0]	I	未处理的待写入数据
ReadData[31:0]	I	RD_of_DM
DMType[1:0]	I	控制元件功能
WriteData_[31:0]	O	Operated_WriteData, 处理后的待写入数据(WD_of_DM)

功能定义

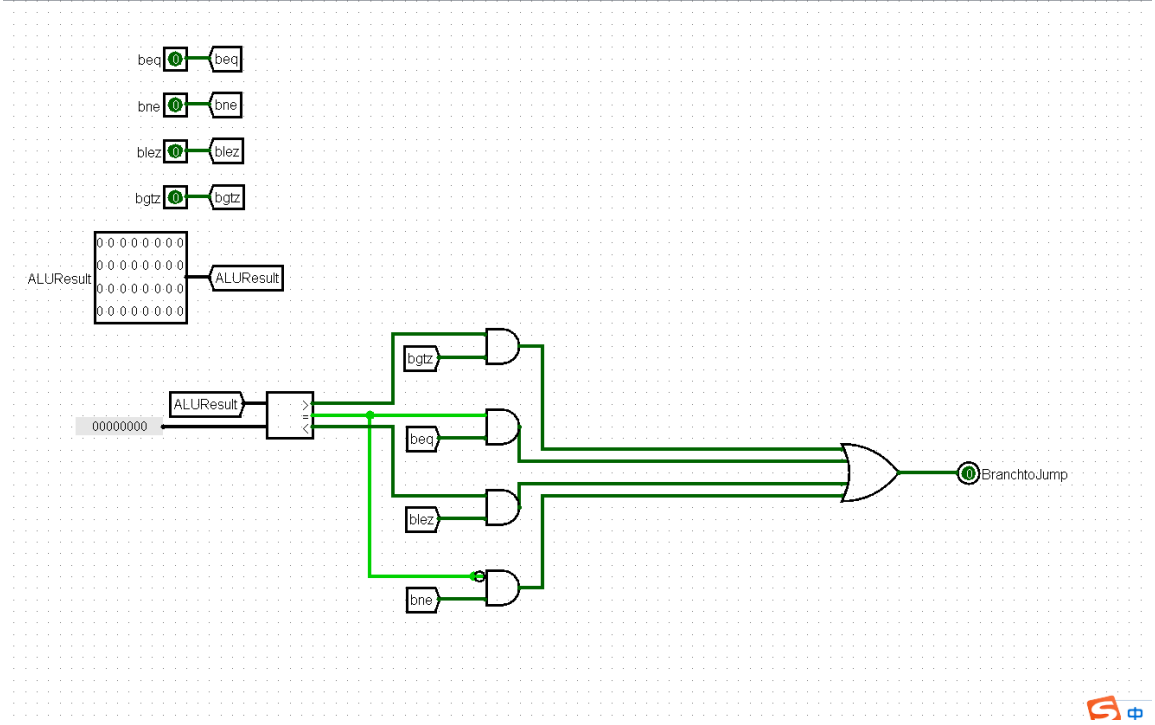
序号	功能名称	功能描述
1	Transform_to_word(DMType == 00)	WriteData_ = WriteData
2	Transform_to_Byte(DMType == 01)	WriteData_ = {ReadData[31:8], WriteData[7:0]}
2	Transform_to_halfword(DMType == 10)	WriteData_ = {ReadData[31:16], WriteData[15:0]}



8. Branchif

模块接口

信号名	方向	功能描述
beq	I	判断指令是否为 beq
bne	I	判断指令是否为 bne
bgtz	I	判断指令是否为 bgtz
bltz	I	判断指令是否为 bltz
ALUResult	I	SrcA - SrcB
BranchtoJump	O	若 ALUResult与对应指令相符合，则输出1



（三）重要机制实现方法

1. 跳转指令

将`beq`系列指令，及`j`，`jal`，`jr`三条指令判定集合成2位信号，引入`NPC`中进行组合逻辑操作，并将`PC`值对应的`Instruction`机器码引入`NPC`，根据跳转类型对指令进行分位与计算操作。

2. 实现对字节与半字的操作

搭建`DMPreReading` `DMPreWriting`两个模块。基本思路，仍将数据以字为基本单位存入，通过对 待写入数据 与 读出数据进行预处理，实现更小单元的写入与读出。这里以字节操作为例，读出字节很简单，只需将完整字读出后，将其前24位置为0即可；而存入字节，我的实现方法为先将`DM`对应地址的`RD`32位输出先提取出来，接下来用`RD`的前24位与待写入数据的后8位拼接后作为`DM`输入即可。具体操作也较为简单，使用 `Splitter` 与 `MUX` (使用DMType[1:0]操作单位判断结果作为选择信号) 即可实现这一功能，如前图。

3.CU 中指令判断与各输出信号间的关系

Instruction	Opcode	Funct	EXTOp	JumpOp[1:0]	MemtoReg	ALUSrc	RegDst	RegWrite	MemWrite	ALU
addu	000000	100001	X	00	0	0	1	1	0	
addu	000000	100001	X	00	0	0	1	1	0	
subu	000000	100011	X	00	0	0	1	1	0	
ori	001101	X	0	00	0	1	0	1	0	
sw	101011	X	1	00	X	1	X	0	1	
lw	100011	X	1	00	1	1	0	1	0	
beq	000100	X	1	00	X	0	X	0	0	
lui	001111	X	X	00	X	X	0	1	0	

测试方案设计

lui, ori, addu, subu 指令检测

```
li      $s0, 4558445
ori     $s1, 1
addu    $s2, $s0, $s1
subu    $s3, $s0, $s1
```

将 简化指令 转化为 基本指令 后结果如下

Bkpt	Address	Code	Basic	
☐	0x00003000	0x3c010045	lui $1, 69	1: li $s0, 4558445
☐	0x00003004	0x34308e6d	ori $16, $1, 36461	
☐	0x00003008	0x36310001	ori $17, $17, 1	2: ori $s1, 1
☐	0x0000300c	0x02119021	addu $18, $16, $17	3: addu $s2, $s0, $s1
☐	0x00003010	0x02119823	subu $19, $16, $17	4: subu $s3, $s0, $s1

MARS运行后预期结果如下：

Name	Number	Value
$zero	0	0
$at	1	4521984
$v0	2	0
$v1	3	0
$a0	4	0
$a1	5	0
$a2	6	0
$a3	7	0
$t0	8	0
$t1	9	0
$t2	10	0
$t3	11	0
$t4	12	0
$t5	13	0
$t6	14	0
$t7	15	0
$s0	16	4558445
$s1	17	1
$s2	18	4558446
$s3	19	4558444
$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		12308
hi		0
lo		0

将指令对应机器码导入 IM运行后，结果与预期一致。

sw, lw 指令检测

```
li      $s0, 4558445
ori     $s1, 1
addu    $s2, $s0, $s1
subu    $s3, $s0, $s1
sw      $s3, 4($0)
sw      $s2, 8($0)
lw      $s4, 4($0)
lw      $s5, 8($0)
```

将 简化指令 转化为 基本指令 后结果如下

Bkpt	Address	Code	Basic	Source
	0x00003000	0x3c010045	lui $1, 69	1: li $s0, 4558445
	0x00003004	0x34308e6d	ori $16, $1, 36461	
	0x00003008	0x36310001	ori $17, $17, 1	2: ori $s1, 1
	0x0000300c	0x02119021	addu $18, $16, $17	3: addu $s2, $s0, $s1
	0x00003010	0x02119823	subu $19, $16, $17	4: subu $s3, $s0, $s1
	0x00003014	0xac130004	sw $19, 4($0)	5: sw $s3, 4($0)
	0x00003018	0xac120008	sw $18, 8($0)	6: sw $s2, 8($0)
	0x0000301c	0x8c140004	lw $20, 4($0)	7: lw $s4, 4($0)
	0x00003020	0x8c150008	lw $21, 8($0)	8: lw $s5, 8($0)

MARS运行后预期结果如下：

Name	Number	Value
$zero	0	0
$at	1	4521984
$v0	2	0
$v1	3	0
$a0	4	0
$a1	5	0
$a2	6	0
$a3	7	0
$t0	8	0
$t1	9	0
$t2	10	0
$t3	11	0
$t4	12	0
$t5	13	0
$t6	14	0
$t7	15	0
$s0	16	4558445
$s1	17	1
$s2	18	4558446
$s3	19	4558444
$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		12308
hi		0
lo		0

将指令对应机器码导入 IM运行后，结果与预期一致。

思考题

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

合理。

ROM是只读存储器，IM作为指令存储器只被用来读出指令，提前将待执行指令机器码存入即可。

而DM却需要在内存中存储并读出数据，使用RAM正合适。

而GRF是寄存器堆，速度要快，选择寄存器，寄存器是这三种原件中速度最快的。

2.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指令对应机器码为0X00000000,只是做了PC=PC+4，别的什么都没有修改，没对逻辑电路电路中的元件进行任何操作，存在与否对电路没有影响。

3.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。

假设DM有256MB容量,并且映射在0x3000_0000-0x3FFF_FFFF区间,那么只需要把高4位地址与0x3进行比较,结果就是DM的片选信号。之前的 DM存满后就从 0x3000_0000~0x3FFF_FFFF存储数据。这次的 DM 最多存到0x0000_00fc,所以不需要片选信号。

4.除了编写程序进行测试外，还有一种验证 CPU 设计正确性的办法——形式验证。 形式验证的含义是根据某个或某些形式规范或属性，使用数学的方法证明其正确性或非正确性。请搜索“形式验证（Formal Verification)”了解相关内容后，简要阐述相比于测试，形式验证的优劣之处。

- 所谓形式验证，是指 从数学上完备地证明或验证电路的实现方案是否确实实现了电路设计所描述的功能 。形式验证方法分为 等价性验证 、模型检验 和 定理证明 等。
- 组合逻辑电路的逻辑验证
    1. 转换为单一抽象模型比较。通过对单一表示的结构进行比较，得出其功能等价的结论。在最坏的情况下，布尔函数为正，表示随输入个数指数增加，其过大的内存需求限制了一般布尔函数的验证能力。
    2. 利用测试输入向量进行验证。探寻使两个电路具有不同输出的输入测试向量，若存在这样的测试向量，则电路在功能上等价。在最坏情况下，这种方法需要穷举所有可能的输入测试矢量，运行时间又成为一个主要问题。
- 时序逻辑电路的验证

    对一个时序电路而言，可以把它看成一个有限状态机。电路功能的等价可以用有限状态机的等价来判断。假定有两个状态机A和B，要对它们进行比较。直观的说，当A和B有相同的接口，而且从相同的初始状态出发，两者对有效输入值序列产生相同的输出值序列，则可以说A和B等价。
- 形式验证的优点：
    1. 形式验证技术是借用数学上的方法将待验证电路和功能描述或参考设计直接进行比较，不需要开发测试激励。
    2. 形式验证是对指定描述的所有可能的情况进行验证，不是仅仅对其中的一个子集进行多次试验，因此有效地克了测试验证的不足。对指定描述的所有可能的情况进行验证，覆盖率达到了100%。
    3. 形式验证可以进行从系统级到门级的验证，验证时间短，有利于尽早、尽快地发现和改正电路设计中的错误，缩短设计周期。
- 形式验证的缺点：

    形式验证到目前为止仍然不能有效的验证电路的性能，如电路的时延和功耗等

参考资料

MIPS Instructions

B

This appendix summarizes MIPS instructions used in this book. Tables B.1–B.3 define the `opcode` and `funct` fields for each instruction, along with a short description of what the instruction does. The following notations are used:

- `[reg]`: contents of the register
- `imm`: 16-bit immediate field of the I-type instruction
- `addr`: 26-bit address field of the J-type instruction
- `SignImm`: sign-extended immediate
  `= {{16{imm[15]}}, imm}`
- `ZeroImm`: zero-extended immediate
  `= {16'b0, imm}`
- `Address`: `[rs] + SignImm`
- `[Address]`: contents of memory location `Address`
- `BTA`: branch target address[1]
  `= PC + 4 + (SignImm << 2)`
- `JTA`: jump target address
  `= {(PC + 4)[31:28], addr, 2'b0}`

[1] The SPIM simulator has no branch delay slot, so BTA is PC + (SignImm << 2). Thus, if you use the SPIM assembler to create machine code for a real MIPS processor, you must decrement the immediate field by 1 to compensate.

Table B.1 Instructions, sorted by opcode

Opcode	Name	Description	Operation
000000 (0)	`R-type`	all R-type instructions	see Table B.2
000001 (1) (`rt` = 0/1)	`bltz/bgez`	branch less than zero/ branch greater than or equal to zero	`if ([rs] < 0) PC = BTA/` `if ([rs] ≥ 0) PC = BTA`
000010 (2)	`j`	jump	`PC = JTA`
000011 (3)	`jal`	jump and link	`$ra = PC+4, PC = JTA`
000100 (4)	`beq`	branch if equal	`if ([rs]==[rt]) PC = BTA`
000101 (5)	`bne`	branch if not equal	`if ([rs]!=[rt]) PC = BTA`
000110 (6)	`blez`	branch if less than or equal to zero	`if ([rs] ≤ 0) PC = BTA`
000111 (7)	`bgtz`	branch if greater than zero	`if ([rs] > 0) PC = BTA`
001000 (8)	`addi`	add immediate	`[rt] = [rs] + SignImm`
001001 (9)	`addiu`	add immediate unsigned	`[rt] = [rs] + SignImm`
001010 (10)	`slti`	set less than immediate	`[rs]<SignImm ? [rt]=1 : [rt]=0`
001011 (11)	`sltiu`	set less than immediate unsigned	`[rs]<SignImm ? [rt]=1 : [rt]=0`
001100 (12)	`andi`	and immediate	`[rt] = [rs] & ZeroImm`
001101 (13)	`ori`	or immediate	`[rt] = [rs] | ZeroImm`
001110 (14)	`xori`	xor immediate	`[rt] = [rs] ^ ZeroImm`
001111 (15)	`lui`	load upper immediate	`[rt] = {Imm, 16'b0}`
010000 (16) (`rs` = 0/4)	`mfc0`, `mtc0`	move from/to coprocessor 0	`[rt] = [rd]/[rd] = [rt]` `(rd is in coprocessor 0)`
010001 (17)	`F-type`	`fop` = 16/17: F-type instructions	see Table B.3
010001 (17)	`bc1f/bc1t`	`fop` = 8: branch if `fpcond` is	`if (fpcond == 0) PC = BTA/`

(rt = 0/1)		FALSE/TRUE	if (fpcond == 1) PC = BTA
100000 (32)	lb	load byte	[rt] = SignExt ([Address]$_{7:0}$)
100001 (33)	lh	load halfword	[rt] = SignExt ([Address]$_{15:0}$)
100011 (35)	lw	load word	[rt] = [Address]
100100 (36)	lbu	load byte unsigned	[rt] = ZeroExt ([Address]$_{7:0}$)
100101 (37)	lhu	load halfword unsigned	[rt] = ZeroExt ([Address]$_{15:0}$)
101000 (40)	sb	store byte	[Address]$_{7:0}$ = [rt]$_{7:0}$

(*continued*)

Table B.1 Instructions, sorted by opcode—Cont'd

Opcode	Name	Description	Operation
101001 (41)	sh	store halfword	$[Address]_{15:0} = [rt]_{15:0}$
101011 (43)	sw	store word	[Address] = [rt]
110001 (49)	lwc1	load word to FP coprocessor 1	[ft] = [Address]
111001 (56)	swc1	store word to FP coprocessor 1	[Address] = [ft]

Table B.2 R-type instructions, sorted by funct field

Funct	Name	Description	Operation
000000 (0)	sll	shift left logical	[rd] = [rt] << shamt
000010 (2)	srl	shift right logical	[rd] = [rt] >> shamt
000011 (3)	sra	shift right arithmetic	[rd] = [rt] >>> shamt
000100 (4)	sllv	shift left logical variable	[rd] = [rt] << $[rs]_{4:0}$ assembly: sllv rd, rt, rs
000110 (6)	srlv	shift right logical variable	[rd] = [rt] >> $[rs]_{4:0}$ assembly: srlv rd, rt, rs
000111 (7)	srav	shift right arithmetic variable	[rd] = [rt] >>> $[rs]_{4:0}$ assembly: srav rd, rt, rs
001000 (8)	jr	jump register	PC = [rs]
001001 (9)	jalr	jump and link register	$ra = PC + 4, PC = [rs]
001100 (12)	syscall	system call	system call exception
001101 (13)	break	break	break exception
010000 (16)	mfhi	move from hi	[rd] = [hi]
010001 (17)	mthi	move to hi	[hi] = [rs]
010010 (18)	mflo	move from lo	[rd] = [lo]
010011 (19)	mtlo	move to lo	[lo] = [rs]
011000 (24)	mult	multiply	{[hi], [lo]} = [rs] × [rt]
011001 (25)	multu	multiply unsigned	{[hi], [lo]} = [rs] × [rt]
011010 (26)	div	divide	[lo] = [rs]/[rt], [hi] = [rs]%[rt]

(*continued*)

Table B.2 R-type instructions, sorted by funct field—Cont'd

Funct	Name	Description	Operation
011011 (27)	divu	divide unsigned	[lo] = [rs]/[rt], [hi] = [rs]%[rt]
100000 (32)	add	add	[rd] = [rs] + [rt]
100001 (33)	addu	add unsigned	[rd] = [rs] + [rt]
100010 (34)	sub	subtract	[rd] = [rs] − [rt]
100011 (35)	subu	subtract unsigned	[rd] = [rs] − [rt]
100100 (36)	and	and	[rd] = [rs] & [rt]
100101 (37)	or	or	[rd] = [rs] | [rt]
100110 (38)	xor	xor	[rd] = [rs] ^ [rt]
100111 (39)	nor	nor	[rd] = ~([rs] | [rt])
101010 (42)	slt	set less than	[rs] < [rt] ? [rd] = 1 : [rd] = 0
101011 (43)	sltu	set less than unsigned	[rs] < [rt] ? [rd] = 1 : [rd] = 0

Table B.3 F-type instructions (fop = 16/17)

Funct	Name	Description	Operation
000000 (0)	add.s/add.d	FP add	[fd] = [fs] + [ft]
000001 (1)	sub.s/sub.d	FP subtract	[fd] = [fs] − [ft]
000010 (2)	mul.s/mul.d	FP multiply	[fd] = [fs] * [ft]
000011 (3)	div.s/div.d	FP divide	[fd] = [fs]/[ft]
000101 (5)	abs.s/abs.d	FP absolute value	[fd] = ([fs] < 0) ? [−fs] : [fs]
000111 (7)	neg.s/neg.d	FP negation	[fd] = [−fs]
111010 (58)	c.seq.s/c.seq.d	FP equality comparison	fpcond = ([fs] == [ft])
111100 (60)	c.lt.s/c.lt.d	FP less than comparison	fpcond = ([fs] < [ft])
111110 (62)	c.le.s/c.le.d	FP less than or equal comparison	fpcond = ([fs] ≤ [ft])