思考题

1. 上面我们介绍了通过 FSM 理解单周期 CPU 的基本方法。请大家指出单周期 CPU 所用到的模块中，哪些发挥状态存储功能，哪些发挥状态转移功能。

> 状态存储功能的模块有：PC（程序计数器），它存储当前指令的地址。GRF（通用寄存器组）也具有状态存储功能，用于存储寄存器的值。DM（数据存储器）同样具有状态存储功能，用于存储数据。

> 状态转移功能的模块有：NPC（下一个程序计数器），它根据当前指令和条件计算下一条指令的地址。Controller（控制器）也具有状态转移功能，根据指令的操作码和功能码生成相应的控制信号，指导数据通路的操作。

2. 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。IM 使用 ROM 是因为指令存储器通常是只读的，程序代码在运行时不会被修改，因此使用 ROM 可以节省成本并提高可靠性。DM 使用 RAM 是因为数据存储器需要支持读写操作，程序在运行过程中会频繁地读写数据，因此使用 RAM 是合适的。GRF 使用 Register 是因为寄存器组需要快速访问和更新寄存器的值，寄存器具有高速读写能力，适合存储临时数据和操作数。

3. 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。

> 在实际实现中，设计了一个 Splitter 模块，用于将指令的各个字段（如 opcode、rs、rt、rd、shamt、funct、imm16）分离出来，方便后续模块使用。设计思路是根据 MIPS 指令格式，将 32 位指令按照位域划分，提取出各个字段，并通过输出端口传递给其他模块使用。

4. 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？

> 因为 nop 指令实际上不执行任何操作，它不会改变任何寄存器或内存的状态。因此，在控制信号真值表中不需要为 nop 指令单独设置控制信号。执行 nop 指令时，CPU 只需保持当前状态，不进行任何数据处理或状态更新即可。

5. 评价如下测试样例的强度（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。

> 测试样例覆盖了多种指令类型，包括 R 型指令（add）、I 型指令（ori、lui、lw、sw、beq），并且测试了不同的操作数组合（正数与正数、正数与负数、负数与负数）。此外，还测试了内存读写操作以及分支跳转的行为。 然而，测试样例中存在一些不足之处：
>
> 1. 缺少对 sub 指令的测试，无法验证减法操作的正确性。
> 2. 对于 lui 指令，仅测试了符号位为 0 和 1 的情况，缺少对边界值（如 0x8000）的测试。
> 3. 对于 beq 指令，仅测试了相等和不相等的情况，缺少对边界条件（如寄存器值为最大或最小整数）的测试。
> 4. 缺少对异常情况的测试，如内存地址越界、寄存器写入冲突等。(目前单周期CPU设计中未考虑异常处理，但在更复杂的设计中应予以考虑)
> 5. 缺少对 nop 指令的测试，虽然它不改变状态，但仍应验证其执行不会影响其他操作。

单周期CPU 设计文档

几个模块：

1. IFU：取指令单元
2. GRF：通用寄存器组
3. ALU：算术逻辑单元
4. DM：数据存储器
5. EXT：扩展单元
6. Controller：控制器

设计思路： 新指令加入

1. 操作码对应信号 Opcode/Funct
2. 控制信号,对应着Controller中的各类控制信号，需要缜密填表，连接到对应控制信号端口的或门
3. 数据通路设计：

1. 分析RTL语言描述的数据通路
2. 增量设计法：Add- Sub- Or- LW-SW-BEQ
3. 硬件需求：
    1. 需要哪个模块
    2. 回写寄存器选择

IFU（取指令单元）

模块描述： 包含PC、NPC、IM三个子模块 （此处填写模块功能、内部结构、实现要点等）

PC（Program Counter）模块

模块功能简述：

- PC 模块保存当前执行的指令地址，在每个时钟周期根据控制信号更新为 next_PC 或保持不变
- PC具有异步复位的功能，复位值为起始地址**0x00003000**

端口名	功能说明	位宽	方向	模块
clk	时钟信号	1	I	IFU
reset	异步复位信号，清零 PC	1	I	IFU
DI[31:0]	输入	32	I	IFU
DO[31:0]	当前 PC 输出	32	O	IM / IFU

NPC（next program counter） 模块

- NPC模块根据当前PC模块输出的当前指令，计算下一条指令的地址
- 如果是顺序执行类指令，则进行PC+4
- 如果是分支指令beq +
    - 若rs与rt相等(zero==1)，则输出PC+4+立即数；否则输出PC+4

端口名	功能说明	位宽	方向	连接模块
PC[31:0]	当前指令地址	32	I	PC
imm[15:0]	16位的指令行偏移	16	I	IM
br	beq分支指令标志	1	I	IM
zero	rs==rt	1	I	IM
NPC[31:0]	下一条指令地址	32	I	PC

IM（Instruction Memory）模块

模块功能简述： IM 模块根据输入的地址输出对应的机器指令，用 ROM 实现。

- 容量为 4096（字 word） × 32bit，即存储了4096条指令(4096 = 2 ^ 12)
    - 所以ROM的地址宽度为12
- 每个地址相差4，所以除以4（右移两位）可得到第几行指令

端口名	功能说明	位宽	方向	连接模块
IM_addr[31:0]	指令地址输入（来自 PC）	32	I	PC / IFU
IM_data[31:0]	输出对应地址的指令内容	32	O	IFU

Splliter 模块

输入： 32位机器指令Instr[31:0]

输出： 各字段信号

字段	位数	起始位（Bit）	结束位（Bit）	字段含义
opcode	6	31	26	标识指令类型，R型指令固定为000000，I型/J型指令通过不同值区分具体操作
rs	5	25	21	源操作数寄存器1的编号（0-31），用于提供第一个操作数
rt	5	20	16	源操作数寄存器2的编号（部分场景下作为目标寄存器，如I型立即数指令）
rd	5	15	11	R型指令中存储运算结果的目标寄存器编号
shamt	5	10	6	移位指令中指定移位的位数（非移位指令固定为0）
funct	6	5	0	配合opcode（R型指令）确定具体运算（如加法、减法等）
imm16	16	15	0	I型指令中的16位有符号立即数，可符号扩展为32位，用于运算或地址偏移

GRF（通用寄存器组）

模块描述： 用于存储处理器运行时使用的寄存器数据， 包含32个32位寄存器。

模块接口

端口名	方向	位宽	功能说明
A1[4:0]	I	5	读到RD1的地址
A2[4:0]	I	5	读到RD2的地址
A3[4:0]	I	5	写入地址
WD[31:0]	I	5	写入数据
RD1[31:0]	O	32	读出的数据1
RD2[31:0]	O	32	读出的数据2
WE	I	1	写入使能信号
Clk	I	1	时钟信号
Reset	I	1	异步复位

功能定义

功能项
寄存器写入
异步复位
读出端口
$0号寄存器常为0$

ALU（算术逻辑单元）

模块描述： 支持 ADDU SUBU ORI LW SW BEQ 几个指令 集成方法：只需要使用加法、或者、判0模块即可

模块接口

端口名	方向	位宽	功能说明
A[31:0]	In	32	输入1
B[31:0]	In	32	输入2

端口名	方向	位宽	功能说明
ALUCtrl[1:0]	In	2	控制信号
Result	Out	1	结果
Zero	Out	1	是否相等

功能定义

功能项	功能描述	对应操作码
加法		000
减法		001
逻辑或		010
大小比较		/

DM（数据存储器）

模块描述： 用于lw, sw指令。 存储程序的结果，能够读出存储器的数据 利用RAM实现，容量为3072 * 32bit，地址位宽为12 异步复位，初始化为0x00000000

模块接口

端口名	方向	位宽	功能说明
Addr[4:0]	I	32	写入的地址
WD[31:0]	I	32	写入的数据
RD[31:0]	O	32	读出的数据
We	I	1	写使能信号
Clk	I	1	时钟信号
reset	I	1	异步复位

功能定义

功能项
读操作
写操作
写使能
异步复位

EXT（扩展单元）

模块描述： 将16位数进行0扩展为32位数

模块接口

端口名	方向	位宽	功能说明
Imm16[15:0]	I	16	16位输入
Ext[15:0]	I	16	32位0扩展结果

功能定义

功能项	功能描述
符号扩展	按照符号数进行扩展
零扩展	用0进行扩展

Controller（控制器）

模块描述： 将机器指令的每一个信息，转化为CPU各部分的控制信号 拆解：

- 和逻辑
- 或逻辑

模块接口

端口名	方向	位宽	功能说明
opcode	I	6	输入的指令操作码，用于识别指令类型
funct	I	6	R型指令的功能码，配合opcode确定具体R型操作
RegDst	O	1	控制寄存器目标地址选择，1选rd字段，0选rt字段(RTL语言箭头左侧的寄存器)
ALUSrc	O	1	控制ALU源操作数选择，1选立即数，0选寄存器rt中的值
MemtoReg	O	1	控制存储器到寄存器的数据选择，1选存储器数据，0选ALU运算结果
RegWrite	O	1	控制寄存器写使能，1允许寄存器写操作
MemWrite	O	1	控制存储器写使能，1允许存储器写操作
nPC_sel	O	1	控制下一个PC选择，1选分支目标地址，0选PC+4
ExtOp	O	1	控制立即数扩展方式，1为符号扩展，0为零扩展
ALUctrl[2:0]	O	3	控制ALU运算类型（如加法、减法、或运算等）

功能定义

功能项
识别
主控制信号生成
运算控制

数据通路表

指令	func	op	RegDst	ALUSrc	MemtoReg	RegWrite	MemWrite	nPC_sel	ExtOp	ALUctr<2:0>	功能描述
add	10 0000	00 0000	1	0	0	1	0	0	X	000	R型加法
sub	10 0010	00 0000	1	0	0	1	0	0	X	001	R型减法
ori	n/a	00 1101	0	1	0	1	0	0	0	010	立即数或
lw	n/a	10 0011	0	1	1	1	0	0	1	000	读内存 (Load)

指令	func	op	RegDst	ALUSrc	MemtoReg	RegWrite	MemWrite	nPC_sel	ExtOp	ALUctr<2:0>	功能描述
sw	n/a	10 1011	X	1	X	0	1	0	1	000	写内存（Store）
beq	n/a	00 0100	X	0	X	0	0	1	X	001	分支相等跳转
lui	n/a	00 1111	0	1	0	1	0	0	X	011	加载立即数到寄存器高位
nop	00 0000	00 0000	X	0	X	0	0	0	X	000	无操作指令

- **lui（Load Upper Immediate）**：RTL语言描述为 `Reg[rt] <- {imm16, 16'b0}`,
- **nop（No Operation）**：

解释

信号名	全称	含义	取值解释
RegDst	Register Destination	选择写回寄存器的目标：是 `rd`（R型）还是 `rt`（I型）。	`1` → 写入 `rd`（如 add、sub） `0` → 写入 `rt`（如 ori、lw）
ALUSrc	ALU Source	控制 ALU 第二个操作数来源：寄存器（rt）还是立即数（imm）。	`0` → 来自寄存器（R型、beq） `1` → 来自立即数（I型、lw、sw、ori）
MemtoReg	Memory to Register	控制写回寄存器的数据来自哪：ALU 结果还是内存。	`0` → 写回 ALU 结果（add、sub、ori） `1` → 写回内存（lw）
RegWrite	Register Write	是否允许写寄存器文件。	`1` → 启用写回（add、lw、ori） `0` → 不写（sw、beq）
MemWrite	Memory Write	是否写入内存。	`1` → 写内存（sw） `0` → 不写（其他）
nPC_sel	Next PC select	控制下一个 PC 的来源：是 PC+4（顺序执行）还是分支目标（跳转）。	`0` → 正常执行（PC+4） `1` → 分支跳转（beq 并且条件满足）
ExtOp	Sign/Zero Extend Operation	控制立即数扩展方式：符号扩展还是零扩展。	\
ALUctr<2:0>	ALU Control	告诉 ALU 要执行什么运算。	000=Add 001=Subtract 010=Or 等