计算机组成原理实验报告参考模板

一、CPU 设计方案综述

（一）总体设计概述

本 CPU 为 Verilog 实现的流水线 MIPS - CPU，支持的指令集包含{addu、subu、ori、lw、sw、beq、lui、jal、jr、nop}。为了实现这些功能，CPU 主要包含了 IFU、Control、grf、EXT、ALU、DM 以及其更细化的模块 IM、and_logic、以及 or_logic，这些模块按照以 main 为顶层、IM 放置在 IFU 中、and_logic 以及 or_logic 放置在 control 中，GRF\EXT\ALU\DM 并列设计逐级展开。

（二）关键模块定义

1. IFU

实际上在设计过程中，我包含一下 IM 模块，因此先从 IM 模块进行定义。

A）IM 模块

名称	输入/输出	位数	简述
PC	I	32	当前输入的 PC 值
instr	O	32	输出的指令

B）IM 功能

1、存储指令：设置 ROM（1024 条），存储指令

2、读取指令：初始化 ROM，读取”code.txt”文件中的指令。

3、输出指令：根据输入的 PC 值，输出指令

C）IFU 模块

名称	输入/输出	位数	简述
PC	I	32	接收到的 PC 值
clr	I	1	复位信号
Instr	O	32	连接到 IM，输出

			IM 中的 instr
PCplus	O	32	输出 PC+4
PCout	O	32	输出当前的 PC 值，用于之后在确定$display 时使用

D）IFU 功能

1、初始化 PC：初始化 PC 为 0x0000_3000

2、同步复位：时钟上升沿到来时，如果 clr 有效，则 PC 为 0x0000_3000

3、时钟控制：时钟上升沿到来时，读入 PC 值到寄存器，同时寄存器分别输出当前 PC 值、PCplus 的值。

2. DM

A）模块定义

名称	输入/输出	位数	简述
Memw	I	1	写入使能
Clk	I	1	时钟
Memr	I	1	读取使能
Clr	I	1	复位
Add	I	10	十位地址
Wdata	I	32	写入数据的值
PC	I	32	当前操作下的 PC（等同于 PCout）
Rdata	O	32	读取的数据值

B）模块功能

1、初始化内存单元：设置 1024 位内存 RAM，同时初始化为 0

2、同步复位：时钟上升沿到来时，如果 clr 有效，则清空内存

3、写入数据：时钟上升沿到来时，如果 memw 有效，则通过 add 将 wdata 写进内存。

4、读取数据：只要 memr 有效，则通过 add 将内存中的值输出到 rdata。

3. ALU

ALU 的设计比较直观。这里说一下自己在具体设计时总想当做函数或者任务来进行调用，因而在 ALU 端口和 EXT 端口想通过调用来简化代码。后来了解到 function 和 task 功能的局限性，意识到还是该模块化的模块化，算得上是写的过程中的一点小体会。

A）端口设计

名称	输入/输出	位数	简述
ALUop	I	3	计算方式
Data_in_rs	I	32	待计算式 1：常常为 rs 寄存器中的值
Data_in_rt	I	32	待计算式 2：为 immediate 或者 rt 寄存器中值
Beq_flag	O	1	Beq 信号
result	O	32	输出计算结果

B）功能定义

1、计算方式：

ALUop 值	计算方式
3’b000	与运算
3’b001	或运算
3’b010	加法
3’b011	减法
3’b100	乘法
3’b101	除法
3’b110	左移（lui 专用）

2、BEQ_FLAG

当输出 result = 0 而且 ALUop 判定为减法运算时，输出为 1.

4. EXT

A）端口定义

名称	位数	输入/输出	作用
Im	16	I	指令后 16 位立即数
Zero_extern	1	I	指令零扩展到 32 位信号
Sign_extern	1	I	指令符号扩展到 32 位信号
Extern	32	0	扩展后指令

B）功能定义

序号	功能	描述
1	符号扩展	当 Zeroop=0，signop=1，extern 符号扩展
2	无符号扩展	当 Zeroop=1，signop=0，extern 无符号扩展

5、control

A）端口定义

名称	输入/输出	位数	简述
Op	I	6	Operation
Fun	I	6	Function
Regdst	O	1	选择写入的寄存器
J26	O	1	J 指令跳转信号

ALUsrc	O	1	选择 GRF 中的值或者立即数
Jal	O	1	Jal 信号
Memtoreg	O	1	将内存输入到寄存器
Jr	O	1	Jr 信号
Regw	O	1	寄存器堆写入使能
Memw	O	1	写入内存使能
Memr	O	1	读取内存使能
Beq	O	1	Beq 信号
Signop	O	1	符号扩展信号
Zeroop	O	1	零扩展信号
ALU	O	3	计算信号

B）功能定义

主要用于生成各种信号，真值表具体分为 and 和 or 真值表

And 真值表：

名称	真值表
R	Op = 000000
beq	Op = 000100
lui	Op = 001111
lw	Op = 100011
ori	Op = 001101
sw	Op = 101011
j	Op = 000010
jal	Op = 000011
jr	Op = R & fun = 001000
Addu	Op = R & fun = 100001
subu	Op = R & fun = 100011

Or 真值表：（部分）

名称	Beq	Lui	Lw	Sw	addu	j	Jr	ori	Jal
regdst	X	X	X	X	1	X	X	X	X
ALUsrc	X	1	1	1	X	X	X	1	X
memtoreg	X	X	1	X	X	X	X	X	X
regw	X	1	1	X	1	X	X	1	1
memw	X	X	X	1	X	X	X	X	X
memr	X	X	1	X	X	X	X	X	X
beq	1	X	X	X	X	X	X	X	X
Signop	1	X	1	1	X	X	X	X	X
Zeroop	X	X	X	X	X	X	X	1	X
J26	X	X	X	X	X	1	X	X	1

6、GRF

A）端口定义

名称	输入/输出	位数	简述
Data	I	32	需要写进寄存器的数据
Rs	I	5	Rs 寄存器
Rt	I	5	Rt 寄存器
Rd	I	5	Rd 寄存器
PC	I	32	当前 PC 作为输出
Clr	I	1	复位
Clk	I	1	时钟
Regw	I	1	写入使能
Data_in_rs	O	32	Rs 寄存器中数据
Data_in_rt	O	32	Rt 寄存器中数据

B）功能定义

1、复位：当时钟上升沿到来时，如果 clr = 1，则将所有寄存器清空

2、写入数据：党始终上升沿到来时，如果 regw = 1 且 Rd !=0，则将数据保存在 rd 寄存器中。

（三）重要机制实现方法

这里说实话不太清楚写一些什么好，所以把自认为比较重要的地方或者难点在这里阐述一下。

1、模块之间的引用

，以 IFU 端为例，如图所示：

```
//PC_input_design
    wire [31:0]PC;
    wire [31:0]instr;
    wire [31:0]PCplus;
    wire [31:0]PCout;
    //PC
    PC IFU(
    .PC(PC),
    .clr(reset),
    .clk(clk),
    .instr(instr),
    .PCplus(PCplus),
    .PCout(PCout)
    );
```

运用在写testbench时所采用的的“引脚”相接的方式，不用加上其他的include，从而可以实现模块的连接和调用。

说明 1：在上图中连接方式如下：

1）定义的 PC 和 IFU 中 PC 连接

2）定义的 reset 和 IFU 中 reset 连接

3）定义的 clk 和 IFU 中的 clk 连接

4）定义的 instr 和 IFU 中 instr 连接

5）定义的 PCplus 和 IFU 中 PCplus 连接

6）定义的 PCout 和 IFU 中 PCout 相连接

说明 2：有关 include 的问题

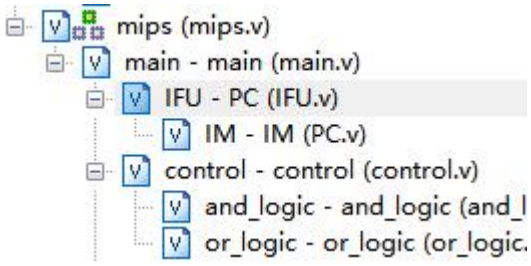
在教程中说明了 include 实际上是文件包含可用于模块调用的，然而一般来

说 include 更适合与只定义了宏的文件进行连接。根据水群中一些同学的描述，include 会将模块再次调用一遍容易出现 redeclaration 的问题。但是本人在实际操作过程中（可能是因为没有用到宏定义）从而并没有因为 include 发生问题。而且另一方面，include 并不必要（至少在本次 project 中），但是在增加 include 和删除 include 的过程似乎会出现问题。

2、模块嵌套

这个根据需要进行。由于在实际处理 P4 的过程和 P3 的定义类似，因此就对这些进行简单的说明，具体一些指令的说明就不再赘述。

如图所示：



通过多层嵌套可以使得代码变得清晰直观。

二、测试方案

（一）典型测试样例

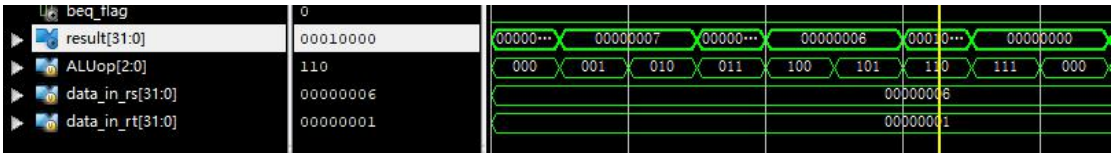
1. ALU 功能测试

Testbench 如下：

```
initial begin
   // Initialize Inputs
   ALUop = 0;
   data_in_rs = 3'b110;
   data_in_rt = 3'b001;

   // Wait 100 ns for global reset


   // Add stimulus here
end
always #5 ALUop = ALUop + 1;
```

根据之前定义的 ALUop，应该先后进行与或加减乘除左移和归零。输出波

形如下：



六个输出和预期相符。

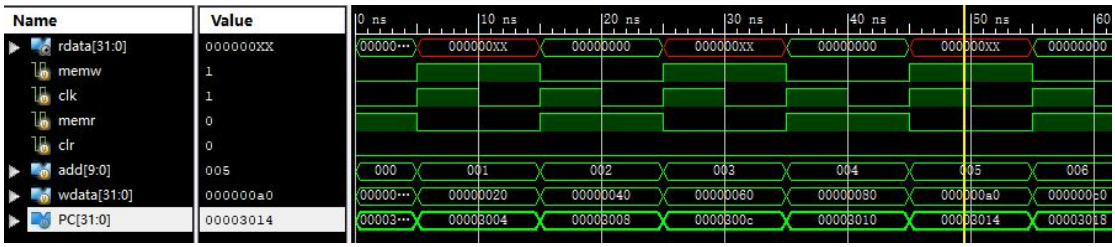
2. DM 功能测试

Testbench 如下：

```
initial begin
    // Initialize Inputs
    memw = 0;
    clk = 0;
    memr = 1;
    clr = 0;
    add = 0;
    wdata = 0;
    PC = 32'h0000_3000;

    // Wait 100 ns for global reset to finish
    #100;

    // Add stimulus here

end
always #5 clk = ~clk;
always@(posedge clk)
begin
    PC <= PC + 4;
    wdata <= wdata + 32;
    add <= add + 1;
    memw <= ~memw;
    memr <= ~memr;
end
```

期待输出如下：

```
This is a Full version of ISim.
Time resolution is 1 ps
Simulator is doing circuit initialization pr
Finished circuit initialization process.
@00003004: *00000004 <= 00000020
@0000300c: *0000000c <= 00000060
@00003014: *00000014 <= 000000a0
@0000301c: *0000001c <= 000000e0
@00003024: *00000024 <= 00000120
@0000302c: *0000002c <= 00000160
@00003034: *00000034 <= 000001a0
@0000303c: *0000003c <= 000001e0
@00003044: *00000044 <= 00000220
```

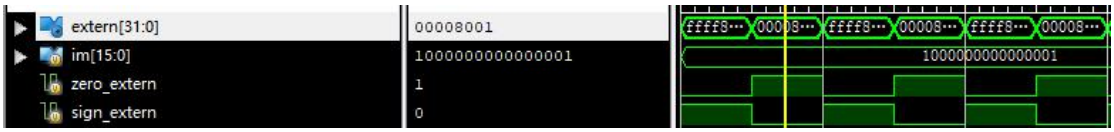


3. EXT 功能测试

Testbench 如下：

```
initial begin
    // Initialize Inputs
    im = 16'h8001;
    zero_extern = 0;
    sign_extern = 1;

    // Wait 100 ns for global reset to finish
    #100;

    // Add stimulus here

end
always #5 begin
    zero_extern = ~zero_extern;
    sign_extern = ~sign_extern;
    end
```

期待输出如下：



4. Control 功能测试

实际上在测试 control 的过程比较复杂和单一，因为没有很好地方式去测试。同时 op 和 fun 变化较多，因此手动测试过程中有点麻烦，control 的测试比较少，注重将组合逻辑写正确，一般都不会出现严重的问题。

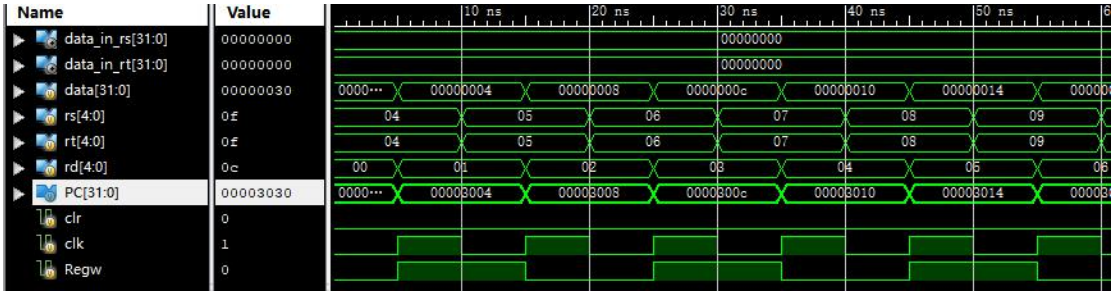
5. GRF 功能测试

Testbench 如下：

```
initial begin
    // Initialize Inputs
    data = 0;
    rs = 4;
    rt = 4;
    rd = 0;
    PC = 32'h0000_3000;
    clr = 0;
    clk = 0;
    Regw = 0;

    // Wait 100 ns for global res
    #100;

    // Add stimulus here

end
always #5 clk = ~ clk;
always@(posedge clk)
fork
    Regw = ~Regw;
    PC = PC + 4;
    data = data + 4;
    #5 rs = rs + 1;
    #5 rt = rt + 1;
    rd = rd + 1;
join
```

期待输出如下：

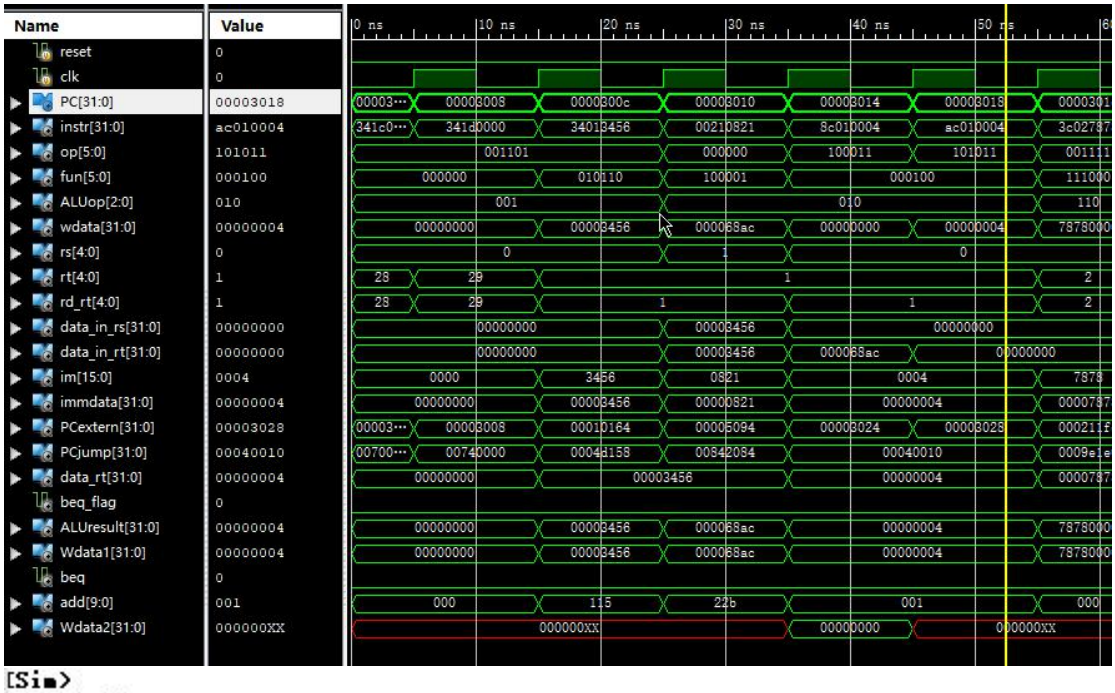
这里 fork-join 使得 rs、rt 和 rd 变化差 5ns。



6. 整体测试（main 测试）

Testbench 测试：使用第一组数据的机械码。

期待输出如下：




```
[Sim>
# run 1.00us
Simulator is doing circuit initialization pr
Finished circuit initialization process.
@00003000: $28 <= 00000000
@00003004: $29 <= 00000000
@00003008: $ 1 <= 00003456
@0000300c: $ 1 <= 000068ac
@00003010: $ 1 <= 00000000
@00003014: *00000004 <= 00000000
@00003018: $ 2 <= 78780000
@0000301c: $ 3 <= 78780000
@00003020: $ 5 <= 12340000
@00003024: $ 4 <= 00000005
@0000302c: *00000004 <= 12340000
@00003030: $ 3 <= 12340000
@00003044: $ 7 <= 12340404
@00003050: $ 8 <= 77770000
@00003054: $ 8 <= 7777ffff
@00003060: $10 <= 12340404
@00003064: $ 8 <= 00000000
@00003068: $ 9 <= 00000001
@0000306c: $10 <= 00000001
@00003070: $ 8 <= 00000001
@00003070: $ 8 <= 00000002
@00003078: $31 <= 0000307c
@00003088: $10 <= 00000002
@00003080: $10 <= 00000004
[Sim>
```

三、思考题

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

答：

实际上这里和上次 P3 的问题类似。由于最后两位一定是 0，因此实际上可以忽略。内存是每四个字节一个字，因此访问字的时候最后两位是没有用处的。

另外 add 信号来自于 ALU 端计算出的结果，取 2~11 位。

2、思考 Verilog 语言设计控制器的译码方式，给出代码示例，并尝试对比各方式的优劣。

答：

A、与或逻辑

还是根据与逻辑和或逻辑进行设计。上面已经具体给出了译码器的真值表，这里简单描述一下。

```
and (R,~op[0],~op[1],~op[2],~op[3],~op[4],~op[5]);
and (beq,~op[0],~op[1],op[2],~op[3],~op[4],~op[5]);
and (lui,op[0],op[1],op[2],op[3],~op[4],~op[5]);
and (lw,op[0],op[1],~op[2],~op[3],~op[4],op[5]);
and (ori,op[0],~op[1],op[2],op[3],~op[4],~op[5]);
and (sw,op[0],op[1],~op[2],op[3],~op[4],op[5]);
and (addu,fun[0],~fun[1],~fun[2],~fun[3],~fun[4],fun[5],R);
and (subu,fun[0],fun[1],~fun[2],~fun[3],~fun[4],fun[5],R);
and (j,~op[0],op[1],~op[2],~op[3],~op[4],~op[5]);
and (jal,op[0],op[1],~op[2],~op[3],~op[4],~op[5]);
and (jr,~fun[0],~fun[1],~fun[2],fun[3],~fun[4],~fun[5],R);
```

```
assign regdst = addu | subu;
assign ALUsrc = lui | lw | ori | sw;
assign memtoreg = lw;
assign regw = addu | subu | lui | lw | ori | jal;
assign memw = sw;
assign memr = lw;
assign Beq = beq;
assign signop = lw | beq | sw;
assign zeroop = ori;
assign ALU[0] = subu | beq | ori;
assign ALU[1] = lw | addu | subu | beq | sw | lui;
assign ALU[2] = lui;
assign j26 = j | jal;
assign Jal = jal;
assign Jr = jr;
```

其中既可以通过 assign 也可以 and&or 语句进行。通过与逻辑和或逻辑是一种较好的译码方式。

B、真值表定义

不通过与或逻辑的定义而直接通过真值表。

在设计过程中默认使用了第一种。因为比较直观，然而对于直接采用真值表进行定义有以下缺陷：

1、实现过程困难。因为是根据 op 和 fun 中每一位来进行计算，从而设计过程相当复杂，（尝试过但是觉得设计过程思路太不清晰而放弃了）而且不容易发现问题。

2、逻辑性不强。实际上直接用真值表没有任何的（中间）关系的体现，因此很难阅读。

而对于其优点可能就是在于利用真值表更容易添加新指令，同时修改地也更方便。

3、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 所驱动的部件具有什么共同特点？

答：

1、都含有存储单元，有的包括寄存器元件，有的包括内存元件，在一定条件下可以保存存储值不变。

2、都含有时钟有效端和复位端。在时钟上升沿到来时进行写入或者读取，在复位端有效时根据实际情况进行清零。

4、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 。

答：

先放上 addi 和 addiu 以及 add 和 addu 的指令操作说明。

A)addi

操作	temp ← (GPR[rs]$_{31}$‖GPR[rs]) + sign_extend(immediate)
	if temp$_{32}$ ≠ temp$_{31}$ then
	SignalException(IntegerOverflow)
	else
	GPR[rt] ← temp$_{31..0}$
	endif

B)addiu

操作	GPR[rt] ← GPR[rs] + sign_extend(immediate)

如果不考虑溢出，从而 addi 中 if 条件语句可以忽略，从而直接进行 if 条件上方的操作，而这个操作和 addiu 的操作是完全相同的，因此可以认为 addi 和 addiu 等价。

C）add

操作	temp ← (GPR[rs]$_{31}$‖GPR[rs]) + (GPR[rt]$_{31}$‖GPR[rt])
	if temp$_{32}$ ≠ temp$_{31}$ then
	SignalException(IntegerOverflow)
	else
	GPR[rd] ← temp$_{31..0}$
	endif

D）addu

操作	GPR[rd] ← GPR[rs] + GPR[rt]

解释同上，不进行 if 条件之中的操作，从而可以认为如果不考虑 溢出，两者等价。

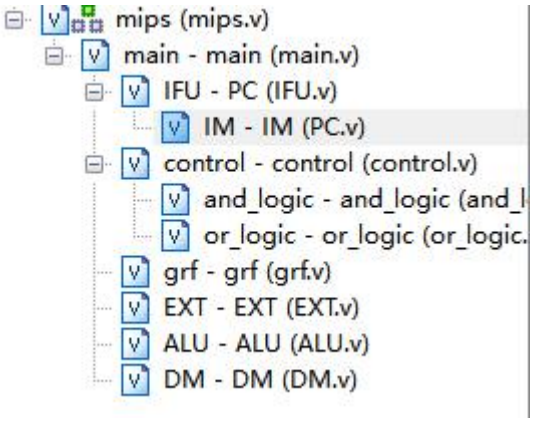
5、根据自己的设计说明单周期处理器的优缺点。

结合自身的设计以及 PPT 有关内容，我认为单周期处理器优点如下：

优点：

A、逻辑简单清楚。实际上对于所有的指令，都可以通过一个周期完成，我们可以详细地了解和控制哪个周期在进行哪一个操作，便于调试和修改。

B、层次分明，容易实现和综合化。比如个人设计的模版形式：



能将每一部分详细地阐述清楚，从而实现硬件描述，我认为这也是单周期设计的优势。

缺点：

A、对于元件的应用程度低，可能在实际开发中使得设计过程冗余复杂。比如在进行 j 指令的时候 ALU 以及后面的元件几乎不起作用，但是其同样会占据空间和时间，可能降低了效率。

B、（个人设计问题）对于添加新的指令、功能较为困难，需要修改很多元件的功能或者添加一些模块才能实现，实用性不高。