Verilog单周期CPU设计报告

一，CPU设计方案综述

（一）总体设计概述

本设计为由Verilog实现的最多支持1024条指令的单周期MIPS-CPU，支持addu，subu，ori，lw，sw，beq，lui，nop, jr, jal 这十条指令。为了实现这这些功能，本实验设计出的CPU被分为了IFU（指令存储单元），GRF（寄存器组），ALU（算术逻辑运算单元），DM（数据存储器），EXT（位扩展器），Controller（控制器），spliter（分解指令）这7个模块。

（二）关键模块定义

1.GRF

A 介绍

通用寄存器组，用于临时存放和调用数据，数据为32位。

B 端口定义

寄存器堆

端口	输入输出(位宽)	描述
RA1	I [4:0]	指定寄存器堆中的一个，将其中数据输出到RD1
RA2	I [4:0]	指定寄存器堆中的一个，将其中数据输出到RD2
WA	I [4:0]	指定32个寄存器中的一个，写入WD数据
WD	I [31:0]	输入的数据
reset	I	异步复位端口
clk	I	时钟信号
WE	I	WE为1：可写入；WE为0，不可写入
RD1	O [31:0]	RA1指定寄存器的值
RD2	O [31:0]	RA2指定寄存器的值

C 功能介绍

功能名称	功能描述
读入数据	当WE为1时，可以将WD写入WA指定的寄存器中
输出数据	可以将RA1和RA2指定的寄存器中的值通过RD1和RA2输出
复位	当reset信号为1时，实现异步复位

2.DM

A 介绍

数据存储器，可以用于存放和调用数据。

B 端口定义

端口	输入输出（位宽）	描述
addr	I [5:0]	待写入（输出）地址
din	I [31:0]	待写入数据
reset	I	异步复位信号
clk	I	时钟信号
Store	I	写入使能端
dout	O [31:0]	输出数据

C 功能定义

功能名称	功能描述
读入数据	当WE为1且时钟上升沿到来时，写入数据
输出数据	将WA指定的内存中存储的数据输出
复位	当reset信号为1时，实现异步复位

3.ALU

A 介绍

算数逻辑运算单元，用于实现指令执行过程中所需要的数学运算

B 端口定义

端口	输入输出（位宽）	描述
A	I [31:0]	参与运算的第一个数据
B	I [31:0]	参与运算的第二个数据
Op	I [2:0]	000:+ 001:- 010:|
C	O [31:0]	运算结果
zero	O	C是0为1，否则为0

C 功能定义

功能名称	功能描述
数学运算	将输入的A，B值按照Op的值进行二元运算，输出结果
判0	判断运算结果是否为0

4.IFU

A 介绍

指令存储单元，内部涵盖指令计数器（PC）和指令寄存器（IM），用于取出当前执行的指令和存放下一个指令。

B 端口定义

端口	输出输出（位宽）	描述
nextPC	I [31:0]	下一条指令地址
clk	I	时钟信号
reset	I	异步复位信号
intr	O [31:0]	从IM中取出的当前指令
PC	O [31:0]	当前指令地址

C 功能定义

功能名称	功能描述
取指令	将当前指令取出用于后续逻辑处理
存指令	将处理后的写一条指令地址存入nextPC
复位	当reset信号为1，PC回归32b'0

5.EXT

A 介绍

用于进行位扩展操作，主要用于将16位立即数扩展成32位

B 端口定义

端口	输入输出（位宽）	描述
in	I [16:0]	输入16位立即数
sign	I [31:0]	输出符号扩展后的数据
unsign	I [31:0]	输出无符号扩展后的数据
bigimm	I [31:0]	输出将立即数低位扩展后的数据

C 功能定义

功能名称	功能描述
无符号扩展	对立即数进行无符号扩展
有符号扩展	对立即数进行有符号扩展
低位扩展	将立即数加载到高16位，低位补充0

6.Controller

A 介绍

中心控制器，用于产生支持各条指令数据通路的片选和控制信号。

B 端口定义

端口	输入输出（位宽）	描述
op	I [5:0]	每条指令的六位操作码
fuc	I [5:0]	每条指令的六位功能码
WDctrl	O [1:0]	寄存器写入选择信号
ALUctrl	O [1:0]	ALU B端口选择信号
MemWrite	O	是否向存储器中写入数据
RegWrite	O	是否向寄存器堆中写入数据
RegDst	O [1:0]	寄存器写入地址选择
ALUOp	O [1:0]	ALU运算模式选择信号
PCctrl	O [1:0]	nextPC选择信号

C 功能定义（真值表）

	nop	addu	subu	lw	sw	beq	jal	jr	lui	ori
op	000000	000000	000000	100011	101011	000100	000011	000000	001111	001101
function	000000	100001	100011	略	略	略	略	001000	略	略
WDctrl	00	00	00	01	00	00	10	00	11	00
ALUctrl	00	00	00	01	01	00	00	00	00	10
MemWrite	0	0	0	0	1	0	0	0	0	0
RegWrite	0	1	1	1	0	0	1	0	1	1
RegDst	00	10	10	00	00	00	01	00	00	00
ALUOp	000	000	001	000	000	001	000	000	000	010
PCctrl	11	11	11	11	11	01	10	00	11	11

7.spliter

A 介绍

用于将32位指令代码分解成易于处理的众多模块。

B 端口定义

端口	输入输出（位宽）	描述
intr	I [31:0]	输入指令
imm	O [15:0]	15位立即数
function	O [5:0]	6位功能码
rd	O [4:0]	寄存器堆输入地址
rt	O [4:0]	寄存器堆操作地址1
rs	O [4:0]	寄存器堆操作地址2
op	O [4:0]	6位操作码

C 功能定义

功能名称	功能描述
分解指令	将32位指令代码分解成imm,function,rs,rt,rd,op和j

（三）重要机制及其实现方法

1 数据通路&&控制信号搭建

首先列出数据通路表，然后进行合并，实现时在中心模块下用assign+三目运算符进行连线。

在控制信号模块直接判断指令的种类，然后对控制信号用三目运算符的方式进行编码。

二，测试方案

（一）典型样例测试

1 综合测试数据

```
subu $28 $28 $28
subu $29 $29 $29
ori $0, $0, 0
ori $1, $1, 1
ori $2, $2, 2
ori $3, $3, 3
ori $4, $4, 4
ori $5, $5, 5
ori $6, $6, 6
ori $7, $7, 7
ori $8, $8, 8
ori $9, $9, 9
ori $10, $10, 10
ori $11, $11, 11
ori $12, $12, 12
ori $13, $13, 13
ori $14, $14, 14
ori $15, $15, 15
ori $16, $16, 16
ori $17, $17, 17
ori $18, $18, 18
ori $19, $19, 19
ori $20, $20, 20
ori $21, $21, 21
ori $22, $22, 22
ori $23, $23, 23
ori $24, $24, 24
ori $25, $25, 25
ori $26, $26, 26
ori $27, $27, 27
ori $28, $28, 28
ori $29, $29, 29
ori $30, $30, 30
ori $31, $31, 31
ori $1, $1, 1
sw $1, 4($0)
sw $1, 8($0)
sw $1, 12($0)
sw $1, 16($0)
```

```
sw $1, 20($0)
sw $1, 24($0)
sw $1, 28($0)
sw $1, 32($0)
sw $1, 36($0)
sw $1, 40($0)
sw $1, 44($0)
sw $1, 48($0)
sw $1, 52($0)
sw $1, 56($0)
sw $1, 60($0)
sw $1, 64($0)
sw $1, 68($0)
sw $1, 72($0)
sw $1, 76($0)
sw $1, 80($0)
sw $1, 84($0)
sw $1, 88($0)
sw $1, 92($0)
sw $1, 96($0)
sw $1, 100($0)
sw $1, 104($0)
sw $1, 108($0)
sw $1, 112($0)
sw $1, 116($0)
sw $1, 120($0)
sw $1, 124($0)
ori $1, $1, 1
sw $1, 0($0)
lw $2, 0($0)
lw $3, 0($0)
lw $4, 0($0)
lw $5, 0($0)
lw $6, 0($0)
lw $7, 0($0)
lw $8, 0($0)
lw $9, 0($0)
lw $10, 0($0)
lw $11, 0($0)
```

lw $11, 0($0)
lw $12, 0($0)
lw $13, 0($0)
lw $14, 0($0)
lw $15, 0($0)
lw $16, 0($0)
lw $17, 0($0)
lw $18, 0($0)
lw $19, 0($0)
lw $20, 0($0)
lw $21, 0($0)
lw $22, 0($0)
lw $23, 0($0)
lw $24, 0($0)
lw $25, 0($0)
lw $26, 0($0)
lw $27, 0($0)
lw $28, 0($0)
lw $29, 0($0)
lw $30, 0($0)
lw $31, 0($0)
ori $1 $1 907
sw $0 0($0)
lw $1 0($0)
sw $1 4($0)
lw $2 4($0)
sw $2 8($0)
lw $3 8($0)
sw $3 12($0)
lw $4 12($0)
sw $4 16($0)
lw $5 16($0)
sw $5 20($0)
lw $6 20($0)
sw $6 24($0)
lw $7 24($0)
sw $7 28($0)
lw $8 28($0)
sw $8 32($0)

lw $9 32($0)
sw $9 36($0)
lw $10 36($0)
sw $10 40($0)
lw $11 40($0)
sw $11 44($0)
lw $12 44($0)
sw $12 48($0)
lw $13 48($0)
sw $13 52($0)
lw $14 52($0)
sw $14 56($0)
lw $15 56($0)
sw $15 60($0)
lw $16 60($0)
sw $16 64($0)
lw $17 64($0)
sw $17 68($0)
lw $18 68($0)
sw $18 72($0)
lw $19 72($0)
sw $19 76($0)
lw $20 76($0)
sw $20 80($0)
lw $21 80($0)
sw $21 84($0)
lw $22 84($0)
sw $22 88($0)
lw $23 88($0)
sw $23 92($0)
lw $24 92($0)
sw $24 96($0)
lw $25 96($0)
sw $25 100($0)
lw $26 100($0)
sw $26 104($0)
lw $27 104($0)
sw $27 108($0)
lw $28 108($0)

```
sw $28 112($0)
lw $29 112($0)
sw $29 116($0)
lw $30 116($0)
sw $30 120($0)
lw $31 120($0)
sw $31 124($0)
lui $1 234
lui $2 234
lui $3 234
lui $4 234
lui $5 234
lui $6 234
lui $7 234
lui $8 234
lui $9 234
lui $10 234
lui $11 234
lui $12 234
lui $13 234
lui $14 234
lui $15 234
jal con
lui $1 222
subu $16 $16 $1
subu $17 $17 $1
subu $18 $18 $1
subu $19 $19 $1
subu $20 $20 $1
subu $21 $21 $1
subu $22 $22 $1
subu $23 $23 $1
subu $24 $24 $1
subu $25 $25 $1
subu $26 $26 $1
subu $27 $27 $1
subu $28 $28 $1
subu $29 $29 $1
subu $30 $30 $1
```

```
jal end
con:
addu $16 $16 $1
addu $17 $17 $2
addu $18 $18 $3
addu $19 $19 $4
addu $20 $20 $5
addu $21 $21 $6
addu $22 $22 $7
addu $23 $23 $8
addu $24 $24 $9
addu $25 $25 $10
addu $26 $26 $11
addu $27 $27 $12
addu $28 $28 $13
addu $29 $29 $14
addu $30 $30 $15
jr $31
end:
```

运行结果：

@00003010: $ 2 <= 00000002
@00003014: $ 3 <= 00000003
@00003018: $ 4 <= 00000004
@0000301c: $ 5 <= 00000005
@00003020: $ 6 <= 00000006
@00003024: $ 7 <= 00000007
@00003028: $ 8 <= 00000008
@0000302c: $ 9 <= 00000009
@00003030: $10 <= 0000000a
@00003034: $11 <= 0000000b
@00003038: $12 <= 0000000c
@0000303c: $13 <= 0000000d
@00003040: $14 <= 0000000e
@00003044: $15 <= 0000000f
@00003048: $16 <= 00000010
@0000304c: $17 <= 00000011
@00003050: $18 <= 00000012
@00003054: $19 <= 00000013
@00003058: $20 <= 00000014
@0000305c: $21 <= 00000015
@00003060: $22 <= 00000016
@00003064: $23 <= 00000017
@00003068: $24 <= 00000018
@0000306c: $25 <= 00000019
@00003070: $26 <= 0000001a
@00003074: $27 <= 0000001b
@00003078: $28 <= 0000001c
@0000307c: $29 <= 0000001d
@00003080: $30 <= 0000001e
@00003084: $31 <= 0000001f
@00003088: $ 1 <= 00000001
@0000308c: *00000004 <= 00000001
@00003090: *00000008 <= 00000001
@00003094: *0000000c <= 00000001
@00003098: *00000010 <= 00000001
@0000309c: *00000014 <= 00000001
@000030a0: *00000018 <= 00000001
@000030a4: *0000001c <= 00000001
@000030a8: *00000020 <= 00000001

@000030ac: *00000024 <= 00000001
@000030b0: *00000028 <= 00000001
@000030b4: *0000002c <= 00000001
@000030b8: *00000030 <= 00000001
@000030bc: *00000034 <= 00000001
@000030c0: *00000038 <= 00000001
@000030c4: *0000003c <= 00000001
@000030c8: *00000040 <= 00000001
@000030cc: *00000044 <= 00000001
@000030d0: *00000048 <= 00000001
@000030d4: *0000004c <= 00000001
@000030d8: *00000050 <= 00000001
@000030dc: *00000054 <= 00000001
@000030e0: *00000058 <= 00000001
@000030e4: *0000005c <= 00000001
@000030e8: *00000060 <= 00000001
@000030ec: *00000064 <= 00000001
@000030f0: *00000068 <= 00000001
@000030f4: *0000006c <= 00000001
@000030f8: *00000070 <= 00000001
@000030fc: *00000074 <= 00000001
@00003100: *00000078 <= 00000001
@00003104: *0000007c <= 00000001
@00003108: $ 1 <= 00000001
@0000310c: *00000000 <= 00000001
@00003110: $ 2 <= 00000001
@00003114: $ 3 <= 00000001
@00003118: $ 4 <= 00000001
@0000311c: $ 5 <= 00000001
@00003120: $ 6 <= 00000001
@00003124: $ 7 <= 00000001
@00003128: $ 8 <= 00000001
@0000312c: $ 9 <= 00000001
@00003130: $10 <= 00000001
@00003134: $11 <= 00000001
@00003138: $12 <= 00000001
@0000313c: $13 <= 00000001
@00003140: $14 <= 00000001

@00003144: $15 <= 00000001
@00003148: $16 <= 00000001
@0000314c: $17 <= 00000001
@00003150: $18 <= 00000001
@00003154: $19 <= 00000001
@00003158: $20 <= 00000001
@0000315c: $21 <= 00000001
@00003160: $22 <= 00000001
@00003164: $23 <= 00000001
@00003168: $24 <= 00000001
@0000316c: $25 <= 00000001
@00003170: $26 <= 00000001
@00003174: $27 <= 00000001
@00003178: $28 <= 00000001
@0000317c: $29 <= 00000001
@00003180: $30 <= 00000001
@00003184: $31 <= 00000001
@00003188: $ 1 <= 0000038b
@0000318c: *00000000 <= 00000000
@00003190: $ 1 <= 00000000
@00003194: *00000004 <= 00000000
@00003198: $ 2 <= 00000000
@0000319c: *00000008 <= 00000000
@000031a0: $ 3 <= 00000000
@000031a4: *0000000c <= 00000000
@000031a8: $ 4 <= 00000000
@000031ac: *00000010 <= 00000000
@000031b0: $ 5 <= 00000000
@000031b4: *00000014 <= 00000000
@000031b8: $ 6 <= 00000000
@000031bc: *00000018 <= 00000000
@000031c0: $ 7 <= 00000000
@000031c4: *0000001c <= 00000000
@000031c8: $ 8 <= 00000000
@000031cc: *00000020 <= 00000000
@000031d0: $ 9 <= 00000000
@000031d4: *00000024 <= 00000000
@000031d8: $10 <= 00000000
@000031dc: *00000028 <= 00000000

@000031e0: $11 <= 00000000
@000031e4: *0000002c <= 00000000
@000031e8: $12 <= 00000000
@000031ec: *00000030 <= 00000000
@000031f0: $13 <= 00000000
@000031f4: *00000034 <= 00000000
@000031f8: $14 <= 00000000
@000031fc: *00000038 <= 00000000
@00003200: $15 <= 00000000
@00003204: *0000003c <= 00000000
@00003208: $16 <= 00000000
@0000320c: *00000040 <= 00000000
@00003210: $17 <= 00000000
@00003214: *00000044 <= 00000000
@00003218: $18 <= 00000000
@0000321c: *00000048 <= 00000000
@00003220: $19 <= 00000000
@00003224: *0000004c <= 00000000
@00003228: $20 <= 00000000
@0000322c: *00000050 <= 00000000
@00003230: $21 <= 00000000
@00003234: *00000054 <= 00000000
@00003238: $22 <= 00000000
@0000323c: *00000058 <= 00000000
@00003240: $23 <= 00000000
@00003244: *0000005c <= 00000000
@00003248: $24 <= 00000000
@0000324c: *00000060 <= 00000000
@00003250: $25 <= 00000000
@00003254: *00000064 <= 00000000
@00003258: $26 <= 00000000
@0000325c: *00000068 <= 00000000
@00003260: $27 <= 00000000
@00003264: *0000006c <= 00000000
@00003268: $28 <= 00000000
@0000326c: *00000070 <= 00000000
@00003270: $29 <= 00000000
@00003274: *00000074 <= 00000000
@00003278: $30 <= 00000000

@0000327c: *00000078 <= 00000000
@00003280: $31 <= 00000000
@00003284: *0000007c <= 00000000
@00003288: $ 1 <= 00ea0000
@0000328c: $ 2 <= 00ea0000
@00003290: $ 3 <= 00ea0000
@00003294: $ 4 <= 00ea0000
@00003298: $ 5 <= 00ea0000
@0000329c: $ 6 <= 00ea0000
@000032a0: $ 7 <= 00ea0000
@000032a4: $ 8 <= 00ea0000
@000032a8: $ 9 <= 00ea0000
@000032ac: $10 <= 00ea0000
@000032b0: $11 <= 00ea0000
@000032b4: $12 <= 00ea0000
@000032b8: $13 <= 00ea0000
@000032bc: $14 <= 00ea0000
@000032c0: $15 <= 00ea0000
@000032c4: $31 <= 000032c8
@0000330c: $16 <= 00ea0000
@00003310: $17 <= 00ea0000
@00003314: $18 <= 00ea0000
@00003318: $19 <= 00ea0000
@0000331c: $20 <= 00ea0000
@00003320: $21 <= 00ea0000
@00003324: $22 <= 00ea0000
@00003328: $23 <= 00ea0000
@0000332c: $24 <= 00ea0000
@00003330: $25 <= 00ea0000
@00003334: $26 <= 00ea0000
@00003338: $27 <= 00ea0000
@0000333c: $28 <= 00ea0000
@00003340: $29 <= 00ea0000
@00003344: $30 <= 00ea0000
@000032c8: $ 1 <= 00de0000
@000032cc: $16 <= 000c0000
@000032d0: $17 <= 000c0000
@000032d4: $18 <= 000c0000
@000032d8: $19 <= 000c0000

```
@000032dc: $20 <= 000c0000
@000032e0: $21 <= 000c0000
@000032e4: $22 <= 000c0000
@000032e8: $23 <= 000c0000
@000032ec: $24 <= 000c0000
@000032f0: $25 <= 000c0000
@000032f4: $26 <= 000c0000
@000032f8: $27 <= 000c0000
@000032fc: $28 <= 000c0000
@00003300: $29 <= 000c0000
@00003304: $30 <= 000c0000
@00003308: $31 <= 0000330c
```

符合预期。

（二）自动化生成测试数据

使用c++程序编程实现。

```cpp
#include <bits/stdc++.h>

using namespace std;

vector<int> r;
mt19937 mt(time(0));
uniform_int_distribution<int>
    imm16(0, (1 << 16) - 1),
    siz(0, 1023),
    reg(0, 2),
    grf(1, 30),
    I(1, 6),
    J(7, 9),
    IJ(1, 9);

int cnt;

void solve(int);

int getR(){
    return r[reg(mt)];
}

void addu(){
    printf("addu $%d, $%d, $%d\n", getR(), getR(), getR());
}
```

```
void subu(){
    printf("subu $%d, $%d, $%d\n", getR(), getR(), getR());
}

void ori(){
    printf("ori $%d, $%d, %d\n", getR(), getR(), imm16(mt));
}

void lw(){
    printf("lw $%d, %d($0)\n", getR(), siz(mt) * 4);
}

void sw(){
    printf("sw $%d, %d($0)\n", getR(), siz(mt) * 4);
}

void lui(){
    printf("lui $%d, %d\n", getR(), imm16(mt));
}

void beq(){
    printf("beq $%d, $%d, label%d\n", getR(), getR(), ++cnt);
    solve(I(mt));
    solve(I(mt));
    solve(I(mt));
    printf("label%d: ", cnt);
    solve(I(mt));
}

void jaljr(){
    int x = getR();
    printf("ori $%d $0 16\n",x);
    printf("jal label%d\n", ++cnt);
    solve(I(mt));
    solve(I(mt));
    printf("label%d:", cnt);
    printf("addu $%d, $%d, $31\n", x, x);
    printf("jr $%d\n", x);
    solve(I(mt));
}

void j(){
    printf("j label%d\n", ++cnt);
    solve(I(mt));
    solve(I(mt));
    solve(I(mt));
    printf("label%d: ", cnt);
    solve(I(mt));
}

void solve(int i){
    switch(i){
        case 1:
            addu();
            break;
        case 2:
            subu();
            break;
```

```
            case 3:
                ori();
                break;
            case 4:
                lw();
                break;
            case 5:
                sw();
                break;
            case 6:
                lui();
                break;
            case 7:
                beq();
                break;
            case 8:
                jaljr();
                break;
            case 9:
                j();
                break;
        }
    }

    int main(){
        r.push_back(grf(mt)), r.push_back(grf(mt)), r.push_back(grf(mt));
        freopen("test.asm", "w", stdout);
        puts("ori $28, $0, 0");
        puts("ori $29, $0, 0");
        for(int i = 1;i <= 700;i++){
            int x = IJ(mt);
            if(x > 6) i += 5;
            solve(x);
        }
    }
```

3 自动化测试方案

先用上述程序生成测试代码写入test.asm，将test.asm导入魔改mars，输出到m.out，再将机器码导出到code.txt，再利用iverilog将test.v的输出写入v.out,最后分别对文件进行排序，比对，将比对结果输出到check.txt，跳回最开始开始循环。

```
:goon
genP4.exe
java -jar Mars_Changed.jar mc CompactDataAtZero nc test.asm > m.out
java -jar Mars_Changed.jar mc CompactDataAtZero a dump .text HexText code.txt test.asm
iverilog -o tb.out -y D:\verilog\p4\P4 D:\verilog\p4\P4\test.v
vvp tb.out -v > v.out
sort.exe
m_sort.exe
fc aftersort.out maftersort.out >>  check.txt
goto goon
```

三，思考题

（一）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DM中存储器一个字占一个地址，然而mips-cpu中默认一个字节占一个地址，所以这里需要把DM输入地址（ALUout）的11到2位作为输出，相当于对地址做一个除4操作。
- addr信号的来源固定是算术逻辑运算单元的输出ALUout。

（二）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。

- 一共有两种方式
  - **指令对应的控制信号如何取值**:

```
if(ori) begin
    WDctrl=0;
    ALUctrl=2;
    MemWrite=0;
    RegWrite=1;
    RegDst=0;
    ALUOp=2;
    PCctrl=3;
end
```

    这种方式比较有针对性，对每个指令数据通路需求的控制信号展现的较为全面，不容易出错，但是修改起来比较麻烦，不够简洁。再后续添加指令后容易使代码变得冗长。

  - **控制信号每种取值所对应的指令**:

```
assign ALUctrl=(ori)? 2:
               (lw||sw)?1:
                0;
```

    这种方式比较简便，利用了真值表的方法避免了穷举的麻烦，而且后续经行修改添加的操作时也十分简便，但是这种方法比较容易出错。

（三）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?

- 这些部件都具有时序特征，可以对数据进行存储和提取，都可以在时钟上升沿到来时，在对应的位置（地址）存储输入的值。

（四）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 。

- 因为addi相对于addiu，add相对于addu多判断了一个结果是否溢出，所以在忽略溢出的情况下，add和addu，addi和addiu是等价的。

（五）根据自己的设计说明单周期处理器的优缺点

- 优点：逻辑相对比较简单，易于实现，处理结果比较精确，不容易出错。
- 缺点：效率比较低，运行指令速度较慢，无法适应工程化要求。