CPU设计方案综述

总体设计概述

本 CPU 为 Verilog 实现的单周期 MIPS-CPU，支持的指令集包括 `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，GRF，DM，ALU，CTRL` 这些模块。

关键模块定义

1.IFU

介绍

取指令单元，内部包括 PC(程序计数器)、IM(指令存储器)及相关逻辑，其中 PC 具有同步复位功能，IM 的容量为 容量为 32bit * 1024，起始地址为 0x00003000。

端口定义

端口	输入/输出	位宽	描述
nPC	I	32	设置下一个 PC 值。
clk	I	1	时钟信号。
reset	I	1	同步复位信号。
I	O	32	当前指令。
PC	O	32	当前 PC 值。

功能定义

序号	功能名称	功能描述
1	同步复位	当时钟上升沿到来且同步复位信号有效时，将 PC 设置为 0x00003000。
2	取指令	当时钟上升沿到来时，I 输出当前 PC 对应的指令。
3	取PC值	当时钟上升沿到来时，PC 输出当前 PC 值。
4	设置PC值	当时钟上升沿到来时，将 nPC 设置为当前 PC 值。

2.GRF

介绍

通用寄存器组，也称为寄存器文件、寄存器堆，可以存取 32 位数据，具有同步复位功能。寄存器标号为 0 到 31，其中 0 号寄存器读取的结果恒为 0。

端口定义

端口	输入/输出	位宽	描述
A1	I	5	指定 32 个寄存器中的一个，将其存储的数据读出到 RD1。
A2	I	5	指定 32 个寄存器中的一个，将其存储的数据读出到 RD2。
A3	I	5	指定 32 个寄存器中的一个，作为写入的目标寄存器。
WD	I	32	写入寄存器的数据信号。
WE	I	1	写使能信号，高电平有效。
clk	I	1	时钟信号。
reset	I	1	同步复位信号。
RD1	O	32	输出 A1 指定的寄存器中的数据。
RD2	O	32	输出 A2 指定的寄存器中的数据。

功能定义

序号	功能名称	功能描述
1	同步复位	当时钟上升沿到来且同步复位信号有效时，将所有寄存器的值设置为 0x00000000。
2	读数据	读出 A1 和 A2 地址对应寄存器中存储的数据到 RD1 和 RD2。
3	写数据	当 WE 有效且时钟上升沿到来时，将 WD 的数据写入 A3 对应的寄存器中。

3.ALU

介绍

算术逻辑单元，提供 32 位按位与、按位或、加法、减法、左移 16 位、判断相等的功能。

端口定义

端口	输入/输出	位宽	描述
A	I	32	参与 ALU 计算的第一个值。
B	I	32	参与 ALU 计算的第二个值。
ALUOp	I	4	ALU 功能的选择信号，具体见功能定义。
AO	O	32	ALU 的计算结果。
Zero	O	1	当 A = B 时为 1，否则为 0。

功能定义

序号	功能名称	功能描述
1	按位与	当 ALUOp = 0 时，AO = A & B。
2	按位或	当 ALUOp = 1 时，AO = A | B。
3	加法	当 ALUOp = 2 时，AO = A + B。
4	减法	当 ALUOp = 3 时，AO = A - B。
5	左移 16 位	当 ALUOp = 4 时，AO = A << 16。
6	判断相等	当 A = B 时，Zero 为 1，否则为 0。

4.DM

介绍

数据存储器，可以存取 32 位数据，容量为 32bit * 1024，具有同步复位功能，起始地址为 0x00000000。

端口定义

端口	输入/输出	位宽	描述
A	I	5	读取或写入数据的地址。
WD	I	32	写入 DM 中的数据。
WE	I	1	写入数据信号，高电平有效。
clk	I	1	时钟信号。
reset	I	1	同步复位信号。
RD	O	32	输出 A 指定地址的数据。

功能定义

序号	功能名称	功能描述
1	同步复位	当时钟上升沿到来且同步复位信号有效时，将 DM 中所有值设为 0x00000000。
2	读数据	读出 A 地址对应的数据到 RD。
3	写数据	当 WE 有效且时钟上升沿到来时，将 WD 的数据写入 A 对应的地址。

5.CTRL

介绍

控制器，产生控制信号。

端口与功能定义

端口	输入/输出	位宽	描述
Op	I	6	所有指令的操作码，对应 I[31:26]。
Func	I	6	R 指令中辅助识别的操作码，对应 I[5:0]。
PCSrc	O	2	0：PC = PC + 4。 1：如果 Zero = 1，PC 跳转到 beq 指令对应的跳转地址；否则依旧执行 PC = PC + 4。 2：跳转到 jal 指令对应的跳转地址。 3：跳转到 jr 指令对应的跳转地址。
MemtoReg	O	2	0：GRF 的 WD 输入端选择 ALU 的 AO 输出端。 1：GRF 的 WD 输入端选择 DM 的 RD 输出端。 2：GRF 的 WD 输入端选择 PC + 4。
MemWrite	O	1	0：DM 的写使能端 WE 无效。 1：DM 的写使能端 WE 有效。
ALUOp	O	4	ALU 功能的选择信号，具体见 ALU 模块的功能定义。
ALUSrc	O	2	0：ALU 的 B 输入端选择 GRF 的 RD2 输出端。 1：ALU 的 B 输入端选择 IFU 的 I[15:0] 的 32 位无符号扩展。 2：ALU 的 B 输入端选择 IFU 的 I[15:0] 的 32 位有符号扩展。
RegWrite	O	1	0：GRF 的写使能端 WE 无效。 1：GRF 的写使能端 WE 有效。
RegDst	O	2	0：GRF 的 A3 输入端选择 IFU 的 I[20:16]。 1：GRF 的 A3 输入端选择 IFU 的 I[15:11]。 2：GRF 的 A3 输入端选择 31。

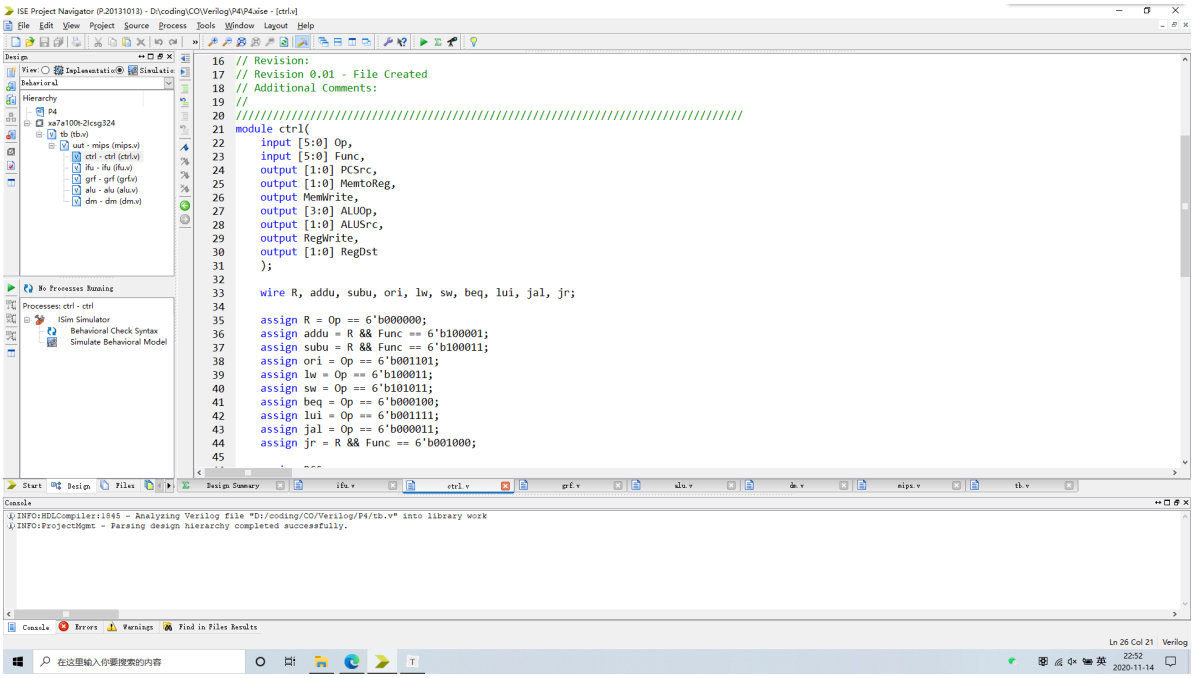
控制信号真值表

		R	addu	subu	ori	lw	sw	beq	lui	jal	jr
	Op	000000			001101	100011	101011	000100	001111	000011	
	Func		100001	100011							001000
PCSrc		0			0	0	0	1	0	2	3
MemtoReg		0			0	1			0	2	
MemWrite		0			0	0	1	0	0	0	0
ALUOp			2	3	1	2	2		4		
ALUSrc		0			1	2	2	0	1		
RegWrite		1			1	1	0	0	1	1	0
RegDst		1			0	0			0	2	

重要机制实现方法

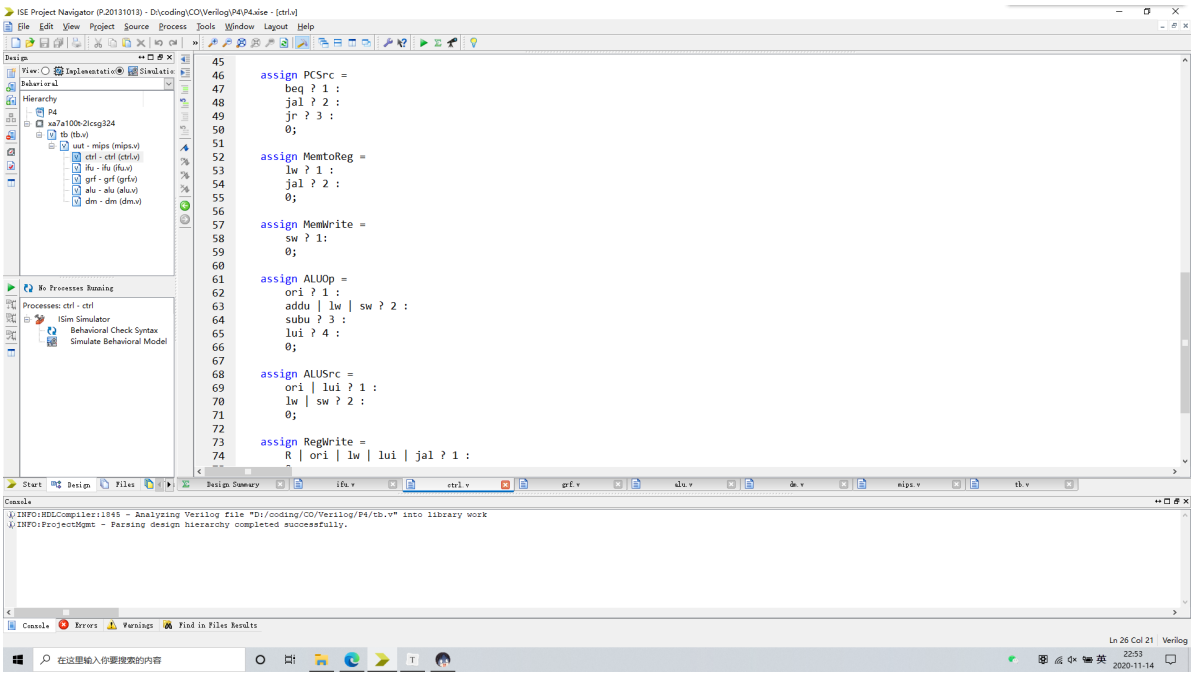
使用逻辑运算符构造控制信号

对比 Logisim，使用 `==` 和 `&&` 代替与门，如下图所示：



```
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////
module ctrl(
    input [5:0] Op,
    input [5:0] Func,
    output [1:0] PCSrc,
    output [1:0] MemtoReg,
    output MemWrite,
    output [3:0] ALUOp,
    output [1:0] ALUSrc,
    output RegWrite,
    output [1:0] RegDst
    );

    wire R, addu, subu, ori, lw, sw, beq, lui, jal, jr;

    assign R = Op == 6'b000000;
    assign addu = R && Func == 6'b100001;
    assign subu = R && Func == 6'b100011;
    assign ori = Op == 6'b001101;
    assign lw = Op == 6'b100011;
    assign sw = Op == 6'b101011;
    assign beq = Op == 6'b000100;
    assign lui = Op == 6'b001111;
    assign jal = Op == 6'b000011;
    assign jr = R && Func == 6'b001000;
```

再使用三目运算符代替或门，如下图所示：



```
    assign PCSrc =
        beq ? 1 :
        jal ? 2 :
        jr ? 3 :
        0;

    assign MemtoReg =
        lw ? 1 :
        jal ? 2 :
        0;

    assign MemWrite =
        sw ? 1:
        0;

    assign ALUOp =
        ori ? 1 :
        addu | lw | sw ? 2 :
        subu ? 3 :
        lui ? 4 :
        0;

    assign ALUSrc =
        ori | lui ? 1 :
        lw | sw ? 2 :
        0;

    assign RegWrite =
        R | ori | lw | lui | jal ? 1 :
```

测试方案

典型测试样例

```
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 $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:
ori $1 $0 1
ori $2 $0 2
beq $1 $2 beq1
addu $1 $1 $1
beq1:
ori $12 $0 1
ori $13 $0 1
beq $12 $13 beq2
addu $2 $2 $2
beq2:
jal con2
jal end2
addu $6 $6 $6
jal end2
con2:
addu $15 $0 $31
ori $5 $0 4
addu $31 $31 $5
jr $15
addu $1 $1 $1
end2:
```

自动测试工具

全自动化对拍器

- 重复下述过程无数次：
  - 使用 C++ 随机化生成一段 MIPS 汇编代码。
  - 使用魔改版的 Mars，执行汇编指令时会按照格式输出评测机需要的信息（即需要 $display 的信息）。
  - 使用 Mars 的命令行代码编译并执行，将输出导入到 `m.out`。
  - 使用 Mars 的命令行代码编译并将机器码导出到 `code.txt`。
  - 使用 IVerilog 命令行代码将 testbench 文件转换为可进行仿真的 `tb.out` 文件。
  - 使用 IVerilog 命令行代码 执行仿真，并将输出导入到 `v.out`。
  - 使用 C++ 删掉 `v.out` 中多余的信息（例如一些警告），将 `m.out` 和 `v.out` 分别排序。
  - 使用 `fc` 将 `v.out` 和 `m.out` 进行比对，将比对信息输入到 `log.txt`。
- 代码如下：
  - 数据生成器：

```
#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, 8),
    IJ(1, 8);

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));
    printf("label%d: ", cnt);
    solve(I(mt));
}

void jaljr(){
    printf("jal label%d\n", ++cnt);
    solve(I(mt));
    int x = getR();
    printf("label%d: ori $%d, $0, 20\n", cnt, x);
    printf("addu $%d, $%d, $31\n", x, x);
    printf("jr $%d\n", x);
    solve(I(mt));
    solve(I(mt));
}

void j(){
    printf("j label%d\n", ++cnt);
    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();
```

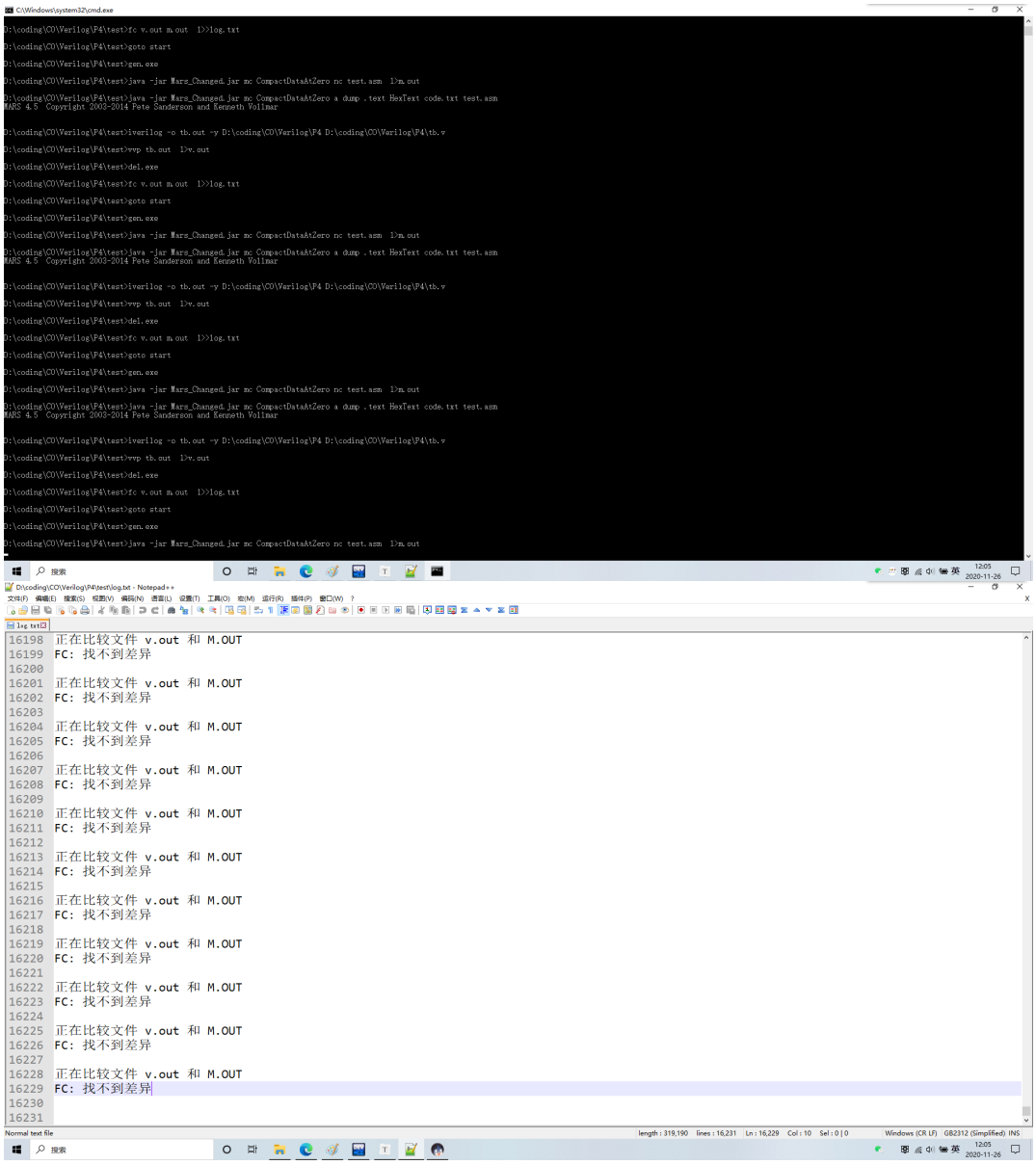
```
            break;
        case 9:
            j();
            break;
    }
}

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);
    }
}
```

- 格式处理器：

```
#include <bits/stdc++.h>
#define maxn 100086

using namespace std;

vector<string> v, w;
char s[maxn];

int main(){
    freopen("v.out", "r", stdin);
    while(gets(s) != NULL){
        string S = s;
        v.push_back(s);
    }
    freopen("v.out", "w", stdout);

    for(int i = 0;i < v.size();i++){
        if(v[i][0] != '@') continue;
        w.push_back(v[i]);
    }
    sort(w.begin(), w.end());
    for(int i = 0;i < w.size();i++) printf("%s\n", w[i].c_str());

    v.clear();
    freopen("m.out", "r", stdin);
    while(gets(s) != NULL){
        string S = s;
        v.push_back(s);
    }
    freopen("m.out", "w", stdout);
    sort(v.begin(), v.end());
    for(int i = 0;i < v.size();i++){
        if(v[i][0] == '@') printf("%s\n", v[i].c_str());
    }
}
```

- bat 代码：

```
:start
gen.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:\coding\CO\Verilog\P4
D:\coding\CO\Verilog\P4\tb.v
vvp tb.out > v.out
del.exe
fc v.out m.out >> log.txt
goto start
```

- 效果图如下：
  - 
  -

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

- DM 实现过程中存储方式为 32bit * 1024，按字存储，而 addr 是以字节为单位，因此要除以 4，由 [9:0] 变为 [11:2]。
- addr 信号来自 ALU 的计算结果输出。

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。
   - 指令对应的控制信号如何取值：

```
always @ * begin
    if(ori) begin
        PCSrc = 0;
        MemtoReg = 0;
        MemWrite = 0;
        ALUOp = 1;
        ALUSrc = 1;
        RegWrite = 1;
        RegDst = 0;
    end
end
```

   - 控制信号每种取值所对应的指令：

```
assign ALUOp =
        ori ? 1 :
        addu | lw | sw ? 2 :
        subu ? 3 :
        lui ? 4 :
        0;
```

   - 对于第一种方式，所有控制信号需要使用 reg 型且每添加一个指令就要把所有信号写一遍，每添加一个信号就要在所有指令里写一遍；对于第二种方式，所有控制信号只需要使用 wire 型，添加一个信号或指令可以省略掉绝大部分指令相同的值（例如0）。因此第二种方式更优。

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?
   - 它们都包含时序逻辑，都存在被复位的需求。

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 。
   - addi 相比 addiu 多了一个如果溢出就抛出 SignalException 异常，如果忽略掉这个异常显然两者等价；add 和 addu 同理。

5. 根据自己的设计说明单周期处理器的优缺点。
    - 优点：数据通路简单，设计精简，易于维护和拓展。
    - 缺点：同一时间只能执行一个指令，耗时长，时间效率低。