计算机组成原理实验报告

一、CPU 设计方案综述

总体设计综述

本 **CPU** 为 **Verilog** 实现的单周期 **CPU**，支持指令集包含 `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,InstrDecoder,Controller,DataType,GRF, Extender,ALU,DM` 。

关键模块定义

IFU

信号名	方向	信号描述
`clk`	I	时钟信号
`reset`	I	复位信号
`Branch`	I	分支信号
`Jump`	I	跳转信号
`JumpAddr[31:0]`	I	跳转地址
`BranchAddr[31:0]`	I	分支地址
`Instr[31:0]`	O	当前指令
`pc[31:0]`	O	指令地址

序号	功能名称	功能描述
1	取指令	根据 `PC` 地址将指令读出
2	复位	当复位信号有效时，将 `PC` 设置为 `0x00003000`
3	计算下一个 `PC`	若 `Jump` 有效，则 `PC = JumpAddr` 若 `Branch` 有效，则 `PC = BranchAddr` 否则，`PC = PC + 4`

InstrDecoder

信号名	方向	信号描述
`Instr[31:0]`	I	当前指令
`opcode[5:0]`	O	当前 `opcode` 操作数
`rs[4:0]`	O	当前 `rs` 寄存器地址
`rt[4:0]`	O	当前 `rt` 寄存器地址
`rd[4:0]`	O	当前 `rd` 寄存器地址
`shamt[4:0]`	O	当前移位数
`funct[5:0]`	O	当前 `funct` 操作数
`imm16[15:0]`	O	16 位立即数
`imm26[25:0]`	O	26 位立即数

Controller

信号名	方向	信号描述
`Instr[31:0]`	I	当前指令
`RegDst`	O	选择 `rd` 和 `rt` 作为写入目标寄存器
`AluSrc`	O	控制运算的数值
`RegWrite`	O	寄存器堆写入使能
`MemToReg`	O	控制 `DM` 数据写入寄存器堆
`MemWrite`	O	`DM` 写入使能
`Branch[2:0]`	O	检测条件偏移指令
`Jump`	O	检测 `J` 指令信号
`Link`	O	将 `PC + 4` 写入寄存器信号
`LinkReg`	O	写入地址进寄存器并有指定寄存器地址信号
`Return`	O	将寄存器的地址写入 `PC` 信号
`Byte`	O	传输 `Byte` 使能信号
`Half`	O	传输 `Half` 使能信号
`Sign`	O	传输 `Sign` 使能信号
`AluCtrl[3:0]`	O	控制运算方式
`Ext_Op[1:0]`	O	控制扩展方式

funct	100000	100010	-	-	-	-	-	000000	-	001000	001001	-	-	-	-	000000	000000	000000
opcode	000000	000000	001101	100011	101011	000100	001111	000000	000011	000000	000000	100000	100001	101000	101001	000000	000010	000011
Instruction	addu	subu	ori	lw	sw	beq	lui	nop	jal	jr	jalr	lb	lh	sb	sh	sll	srl	sra
RegDst	1	1	0	0	x	x	0	1	0	x	0	0	0	x	x	1	1	1
AluSrc	0	0	1	1	1	1	1	0	0	0	0	1	1	1	1	0	0	0
RegWrite	1	1	1	1	0	0	1	1	1	0	1	1	1	0	0	1	1	1
MemToReg	0	0	0	1	0	0	0	0	0	0	0	0	0	1	1	0	0	0
MemWrite	0	0	0	0	1	0	0	0	0	0	0	0	0	1	1	0	0	0
Branch[2:0]	x	x	x	x	x	equal	x	x	x	x	x	x	x	x	x	x	x	x
Jump	0	0	0	0	0	0	0	0	1	1	1	0	0	0	0	0	0	0
Link	0	0	0	0	0	0	0	0	1	0	1	0	0	0	0	0	0	0
LinkReg	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0
Return	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0
Byte	0	0	0	0	0	0	0	0	0	0	0	1	0	1	0	0	0	0
Half	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	0	0	0
Sign	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0
ALUCtrl[3:0]	ADD	SUB	OR	ADD	ADD	COMP	ADD	SLL	x	x	x	ADD	ADD	ADD	ADD	SLL	SRL	SRA
Ext_Op[1:0]	x	x	sign16	sign16	sign16	sign16	upper	x	zero26	x	x	sign16	sign16	sign16	sign16	x	x	x

DataType

信号名	方向	信号描述
`Sign`	I	符号信号
`Offset[1:0]`	I	偏移量
`Data[31:0]`	I	输入数据
`ByteData[31:0]`	O	输出 `Byte` 数据
`HalfData[31:0]`	O	输出 `Half` 数据
`WordData[31:0]`	O	输出 `Word` 数据

GRF

信号名	方向	信号描述
`clk`	I	时钟信号
`reset`	I	复位信号
`RegWrite`	I	写入使能信号
`A1[4:0]`	I	指定 32 个寄存器中的一个，输出其中数据到 `RD1`
`A2[4:0]`	I	指定 32 个寄存器中的一个，输出其中数据到 `RD2`
`A3[4:0]`	I	指定 32 个寄存器中的一个，写入 `WriteData` 数据
`PC[31:0]`	I	`PC` 地址
`WriteData[31:0]`	I	输入数据
`RD1[31:0]`	O	`A1` 指定寄存器中的数据
`RD2[31:0]`	O	`A2` 指定寄存器中的数据

序号	功能名称	功能描述
1	复位	当复位信号有效时，寄存器清零
2	写数据	读出 `A1,A2` 的寄存器数据到 `RD1,RD2`
3	读数据	当 `RegWrite` 有效时且时钟上升沿时，将 `WriteData` 写入指定寄存器内

Extender

信号名	方向	信号描述
`imm16[15:0]`	I	16 位立即数
`imm26[25:0]`	I	26 位立即数
`ExtCtrl[1:0]`	I	控制扩展方式信号
`imm[31:0]`	O	指定扩展后 32 位数据

序号	功能名称	功能描述
1	扩展	按照 `ExtCtrl` 信号指定扩展数据和方式

ALU

信号名	方向	信号描述
`A[31:0]`	I	第一运算数
`B[31:0]`	I	第二运算数
`S[4:0]`	I	移位数
`AluCtrl[3:0]`	I	控制运算方式信号
`D[31:0]`	O	运算结果

序号	功能名称	功能描述
1	运算	按照 `AluCtrl` 信号进行运算
2	比较	若 `AluCtrl = 6`，进行两数比较并将比较结果输出在 `D` `D = 001` 表示小于 `D = 010` 表示等于 `D = 100` 表示大于

DM

信号名	方向	信号描述
`clk`	I	时钟信号
`reset`	I	复位信号
`MemWrite`	I	写入存储器使能信号
`Byte`	I	`Byte` 信号
`Half`	I	`Half` 信号
`Addr[31:0]`	I	写入地址
`PC[31:0]`	I	`PC` 地址
`DataWrite[31:0]`	I	写入数据
`DataRead[31:0]`	O	读出数据

序号	功能名称	功能描述
1	复位	当 `reset` 信号有效则将 `DM` 数据清零
2	写数据	当 `MemWrite` 有效且时钟上升沿时，将 `DataWrite` 数据写入指定地址
3	读数据	从 `Addr` 读出数据

二、测试方案

典型测试样例

汇编代码

```
ori $t0, $0, 0x4123
ori $t1, $0, 0x1234
addu $t0, $t0, $t1
subu $t1, $t0, $t1
lui $t0, 0xffff
lui $t1, 0x1111
lui $t2, 0x0000
sw $t0, ($0)
sw $t1, 4($0)
sw $t2, 8($0)
lw $s0, 8($0)
lw $s1, 4($0)
lw $s2, ($0)
beq $t0, $s2, front
nop
nop
nop
front:
addu $t0, $0, $t1
ori $t2, $0, 1
addu $t0, $t0, 1
back:
subu $t0, $t0, 1
beq $t0, $t1, back
ori $t0, $0, 0
ori $t1, $0, 10
ori $t2, $0, 1
for:
    beq $t0, $t1, for_end
    addu $t3, $0, $t0
    addu $t0, $t0, $t2
    jal for
for_end:
    jal fun
    jal return
fun:
    ori $t0, $0, 0x1234
    lui $t1, 0x1234
    jr $ra
return:
    nop
```

机器码

```
34084123
34091234
01094021
01094823
3c08ffff
3c091111
3c0a0000
ac080000
ac090004
ac0a0008
8c100008
8c110004
8c120000
11120003
00000000
00000000
00000000
00094021
```

```
340a0001
3c010000
34210001
01014021
3c010000
34210001
01014023
1109fffc
34080000
3409000a
340a0001
11090003
00085821
010a4021
0c000c1d
0c000c23
0c000c26
34081234
3c091234
03e00008
00000000
```

期望结果

```
@00003000: $ 8 <= 00004123
@00003004: $ 9 <= 00001234
@00003008: $ 8 <= 00005357
@0000300c: $ 9 <= 00004123
@00003010: $ 8 <= ffff0000
@00003014: $ 9 <= 11110000
@00003018: $10 <= 00000000
@0000301c: *00000000 <= ffff0000
@00003020: *00000004 <= 11110000
@00003024: *00000008 <= 00000000
@00003028: $16 <= 00000000
@0000302c: $17 <= 11110000
@00003030: $18 <= ffff0000
@00003044: $ 8 <= 11110000
@00003048: $10 <= 00000001
@0000304c: $ 1 <= 00000000
@00003050: $ 1 <= 00000001
@00003054: $ 8 <= 11110001
@00003058: $ 1 <= 00000000
@0000305c: $ 1 <= 00000001
@00003060: $ 8 <= 11110000
@00003058: $ 1 <= 00000000
@0000305c: $ 1 <= 00000001
@00003060: $ 8 <= 1110ffff
@00003068: $ 8 <= 00000000
@0000306c: $ 9 <= 0000000a
@00003070: $10 <= 00000001
@00003078: $11 <= 00000000
@0000307c: $ 8 <= 00000001
@00003080: $31 <= 00003084
@00003078: $11 <= 00000001
@0000307c: $ 8 <= 00000002
@00003080: $31 <= 00003084
@00003078: $11 <= 00000002
@0000307c: $ 8 <= 00000003
@00003080: $31 <= 00003084
@00003078: $11 <= 00000003
@0000307c: $ 8 <= 00000004
@00003080: $31 <= 00003084
@00003078: $11 <= 00000004
@0000307c: $ 8 <= 00000005
@00003080: $31 <= 00003084
@00003078: $11 <= 00000005
@0000307c: $ 8 <= 00000006
@00003080: $31 <= 00003084
@00003078: $11 <= 00000006
```

```
@0000307c: $ 8 <= 00000007
@00003080: $31 <= 00003084
@00003078: $11 <= 00000007
@0000307c: $ 8 <= 00000008
@00003080: $31 <= 00003084
@00003078: $11 <= 00000008
@0000307c: $ 8 <= 00000009
@00003080: $31 <= 00003084
@00003078: $11 <= 00000009
@0000307c: $ 8 <= 0000000a
@00003080: $31 <= 00003084
@00003084: $31 <= 00003088
@0000308c: $ 8 <= 00001234
@00003090: $ 9 <= 12340000
@00003088: $31 <= 0000308c
```

汇编代码

```
ori $a0, $0, 0x1000
ori $t0, $0, 0x3018
sw $t0, ($a0)
lb $t1, 1($a0)
lh $t2, 2($a0)
lw $t3, ($a0)
ori $t0, $0, 0xffff
sw $t0, 4($a0)
lb $t1, 5($a0)
lh $t2, 4($a0)
lw $t3, 4($a0)
lui $t0, 0x1234
sw $t0, -4($a0)
lb $t1, -1($a0)
lh $t2, -2($a0)
lw $t3, -4($a0)
ori $a0, $0, 0x0000
ori $t0, $0, 0x3018
sw $t0, ($a0)
ori $t0, $0, 0x1234
sb $t0, 2($a0)
sh $t0, 6($a0)
```

机器码

```
34041000
34083018
ac880000
80890001
848a0002
8c8b0000
3408ffff
ac880004
80890005
848a0004
8c8b0004
3c081234
ac88fffc
8089ffff
848afffe
8c8bfffc
34040000
34083018
ac880000
34081234
a0880002
a4880006
```

期望结果

```
@00003000: $ 4 <= 00001000
@00003004: $ 8 <= 00003018
@00003008: *00001000 <= 00003018
@0000300c: $ 9 <= 00000030
@00003010: $10 <= 00000000
@00003014: $11 <= 00003018
@00003018: $ 8 <= 0000ffff
@0000301c: *00001004 <= 0000ffff
@00003020: $ 9 <= ffffffff
@00003024: $10 <= ffffffff
@00003028: $11 <= 0000ffff
@0000302c: $ 8 <= 12340000
@00003030: *00000ffc <= 12340000
@00003034: $ 9 <= 00000012
@00003038: $10 <= 00001234
@0000303c: $11 <= 12340000
```

```
@00003040: $ 4 <= 00000000
@00003044: $ 8 <= 00003018
@00003048: *00000000 <= 00003018
@0000304c: $ 8 <= 00001234
@00003050: *00000000 <= 00343018
@00003054: *00000004 <= 12340000
```

汇编代码

```
main:
    lui $t0, 0xffff
    ori $t0, $t0, 0xffff
    sll $t1, $t0, 2
    srl $t2, $t1, 2
    sra $t3, $t1, 2
    jal fun
fun:
    move $s1, $ra
    jalr $ra, $s1
```

机器码

```
3c08ffff
3508ffff
00084880
00095082
00095883
0c000c06
001f8821
0220f809
```

期望结果

```
@00003000: $ 8 <= ffff0000
@00003004: $ 8 <= ffffffff
@00003008: $ 9 <= fffffffc
@0000300c: $10 <= 3fffffff
@00003010: $11 <= ffffffff
@00003014: $31 <= 00003018
@00003018: $17 <= 00003018
@0000301c: $31 <= 00003020
@00003018: $17 <= 00003020
@0000301c: $31 <= 00003020
```

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

来源来自于 ALU 计算结果，如果直接取 ALU 的 $[11:2]$ 位直接获得了 DM 的 `word` 地址，不需要进行移位。

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。

第一种，使用 if else 来进行判断

```
always @ (*) begin
    if(opcode == 6'b000000) begin // R指令
        if(funct == 6'b100001) begin // addu 指令
            RegDst = 1;
            AluSrc = 0;
            RegWrite = 1;
            MemWrite = 0;
            .
            .
            .
        end
        else if(funct == 6'b??????)
    end

    else if(opcode == 6'b??????) begin // 其他指令
        .
        .
        .
    end
end
```

简单构造，搭建思路比较清晰，但代码会比较冗长

利用 case 判断

```
always @ (*) begin
    case(opcode)
        6'b000000: begin
            case(funct)
                6'b??????: begin end
            endcase
        end
        6'b??????: begin end
        6'b??????: begin end
        6'b??????: begin end
        6'b??????: begin end
    endcase
end
```

比if else来得整齐，但代码还是会比较冗长

利用与或逻辑来判断

```
wire R = opcode == 6'b000000; // R指令
wire addu = R & funct == 6'b100001; // addu 指令
wire ori = opcode == 6'b001101; // ori 指令

assign RegWrite = addu | ori;
assign AluSrc = ori;
```

加指令较容易，搭建思路也较清晰，比较难出现漏接线的问题。

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？

   DM, GRF, IFU，为了重新初始化数据。

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 。

1. ADD：符号加

编码	31 26	25 21	20 16	15 11	10 6	5 0
	special 000000	rs	rt	rd	0 00000	add 100000
	6	5	5	5	5	6
格式	add rd, rs, rt					
描述	GPR[rd] ← GPR[rs]+GPR[rt]					

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					

示例	add $s1, $s2, $s3					
其他	temp$_{32}$≠ temp$_{31}$ 代表计算结果溢出。 如果不考虑溢出，则 add 与 addu 等价。					

4. ADDU：无符号加

编码	31 26	25 21	20 16	15 11	10 6	5 0
	special 000000	rs	rt	rd	0 00000	addu 100001
	6	5	5	5	5	6
格式	addu rd, rs, rt					
描述	GPR[rd] ← GPR[rs] + GPR[rt]					
操作	GPR[rd] ← GPR[rs] + GPR[rt]					
示例	addu $s1, $s2, $s3					
其他						

`add` 操作需添加数据最高位的信号为，然后再相加，相加后若最高位和次高位信号不同则发出异常，否则就和 `addu` 等价。

`addi` 和 `addiu` 同理。

5. 根据自己的设计说明单周期处理器的优缺点。

单周期处理器较容易搭建，基本上除了写入操作都是组合逻辑，但处理指令的速度较慢。