P4-Verilog 单周期 CPU 设计文档

一、设计说明

使用 Logisim 开发一个简单的 MIPS 单周期处理器，设计说明如下：

1. 处理器应支持指令集为：{addu, subu, ori, lw, sw, beq, lui, jal, jr,nop}。
2. addu,subu 可以不支持溢出。
3. 处理器为单周期设计。
4. 不需要考虑延迟槽。

二、数据通路设计

1. PC

(1) 模块接口定义

表 1 PC 模块接口定义

文件	模块接口定义
pc.v	module pc(input clk, input reset, input [31:0] nextPC, output reg [31:0] PC);

(2) 接口说明

表 2 PC 接口说明

序号	信号	方向	描述
1	clk	I	时钟信号
2	reset	I	同步复位信号 1：复位 0：无效
3	nextPC	I	PC 输入值
4	PC	O	PC 输出值

(3)功能定义

表 3 PC 功能定义

序号	功能	描述
1	复位	reset 有效时，PC 被设置为 0x00003000
2	PC 更新	时钟上升沿来临时，PC 值更新为 nextPC

2. NPC

(1) 模块接口定义

表 4 PC 模块接口定义

文件	模块接口定义
npc.v	module npc(input i_Zero, input [1:0] i_branch, input [25:0] i_jal_addr, input [31:0] i_jr_addr, input [31:0] i_offset, input [31:0] i_PC, output reg [31:0] o_nextPC, output [31:0] o_PC4);

(2) 接口说明

表 2 PC 接口说明

序号	信号	方向	描述
1	zero	I	判断 PC 是否满足 beq 指令跳转条件 1：满足 0：不满足
2	branch	I	判断当前指令是否为 beq 指令 1：是 0：不是
3	jal_addr	I	jal 指令跳转地址
4	jr_addr	I	jr 指令跳转地址
5	offset	I	beq 指令偏移量
6	PC	I	PC 输入值
7	nextPC	O	nextPC 计算结果
8	PC4	O	PC+4 输出值

(3)功能定义

表 3 PC 功能定义

序号	功能	描述
1	计算 PC 的下一个值	zero=1 且 branch=01 时，nextPC <= PC4 + (offset << 2); branch=10 时， nextPC <= { PC4[31:28], jal_addr[25:0],{2{1'b0}}}; branch=11 时, nextPC <= jr_addr; 默认情况, nextPC <= PC4;

3. IM

(1) 模块接口定义

表 4 PC 模块接口定义

文件	模块接口定义
im.v	module im(input [31:0] PC, output [31:0] instruction);

(2) 接口说明

表 2 PC 接口说明

序号	信号	方向	描述
1	PC	I	32 位 PC 值
2	instruction	O	取出 32 位指令值

(3)功能定义

表 3 PC 功能定义

序号	功能	描述
1	取指令	根据 PC 值取出指令

4. GRF

(1) 模块接口定义

表 4 PC 模块接口定义

文件	模块接口定义
grf.v	module grf(input clk, input reset, input WE, input [4:0] A1, input [4:0] A2, input [4:0] A3, input [31:0] WD, input [31:0] PC, output [31:0] RD1, output [31:0] RD2);

(2) 接口说明

表 3 GRF 端口说明

序号	信号	方向	描述

1	clk	I	时钟信号
2	reset	I	异步复位信号，将 32 个寄存器中的值全部清零 1：复位 0：无效
3	A1	I	5 位地址输入信号，指定 32 个寄存器中的一个，将其中存储的数据读出到 RD1
4	A2	I	5 位地址输入信号，指定 32 个寄存器中的一个，将其中存储的数据读出到 RD2
5	A3	I	5 位地址输入信号，指定 32 个寄存器中的一个作为写入的目标寄存器
6	WD	I	32 位写入数据
7	WE	I	写使能信号 1：可向 GRF 中写入数据 0：不能向 GRF 中写入数据
8	PC	I	当前 PC 值
9	RD1	O	输出 A1 指定的寄存器的 32 位数据
10	RD2	O	输出 A2 指定的寄存器的 32 位数据

(3)功能定义

表 4 GRF 功能定义

序号	功能	描述
1	复位	Reset 有效时，将 32 个寄存器中的值全部清零
2	读数据	读出 A1, A2 地址对应寄存器中所存储数据到 RD1, RD2
3	写数据	当 WE 有效且时钟上升沿来临时，将 WD 写入 A3 所对应的寄存器中

5. ALU

(1) 模块接口定义

表 4 PC 模块接口定义

文件	模块接口定义

alu.v	`module alu(`
	`  input [31:0] A,`
	`  input [31:0] B,`
	`  input [2:0] ALUOp,`
	`  output [31:0] Result,`
	`  output Zero);`

(2) 接口说明

表 5 ALU 端口说明

序号	信号	方向	描述

1	A	I	参与 ALU 计算的第一个 32 位数据
2	B	I	参与 ALU 计算的第二个 32 位数据
3	ALUOp	I	ALU 功能的选择信号： 000：ALU 进行与运算 001：ALU 进行或运算 010：ALU 进行加法运算 011：ALU 进行减法运算
4	Result	O	ALU 的计算结果
5	Zero	O	A,B 是否相等的标志信号 1：相等 0：不相等

(3)功能定义

表 6 ALU 功能定义

序号	功能	描述
1	与运算	$\text{Result} = A \,\&\, B$
2	或运算	$\text{Result} = A \mid B$
3	加运算	$\text{Result} = A + B$
4	减运算	$\text{Result} = A - B$
5	判断 A、B 是否相等	$Zero = (A == B)?$

6. DM

(1) 模块接口定义

表 4 PC 模块接口定义

文件	模块接口定义
dm.v	module dm(input clk, input reset, input MemWrite, input MemRead, input [31:0] addr, input [31:0] WriteData, input [31:0] PC, output [31:0] ReadData);

(2) 接口说明

表 7 DM 端口说明

序号	信号	方向	描述
1	clk	I	时钟信号
2	reset	I	异步复位信号，将 DM 中的值全部清零

			1：复位 0：无效
3	MemRead	I	写使能信号 1：可向 DM 中写入数据 0：无效
4	MemWrite	I	读使能信号 1：可读取 DM 中数据 0：无效
5	addr	I	32 位地址输入信号，对 DM 指定地址进行读写操作
6	WriteData	I	32 位写入数据
7	PC	I	当前 PC 值
8	ReadData	O	32 位输出数据

(3)功能定义

表 8 DM 功能定义

序号	功能	描述
1	复位	Reset 有效时，将 DM 中的值全部清零
2	读操作	读出 addr 地址对应存储数据到 ReadData
3	写操作	当时钟上升沿来临时，将 WriteData 写入 addr 地址对应位置

7. EXT

(1) 模块接口定义

表 4 PC 模块接口定义

文件	模块接口定义

ext.v	`module ext(`
	`  input [15:0] in,`
	`  input EXTOp,`
	`  output [31:0] out);`

(2) 接口说明

表 9 EXT 端口说明

序号	信号	方向	描述
1	in	I	16 位待扩展数据
2	EXTop	I	扩展操作信号 0：进行无符号扩展 1：进行符号扩展
3	out	O	32 位输出数据

(3)功能定义

表 10 EXT 功能定义

序号	功能	描述
1	无符号扩展	将 16 位立即数 Imm16 无符号拓展至 32 位输出 Imm32
2	符号扩展	将 16 位立即数 Imm16 符号拓展至 32 位输出 Imm32

8. MUX

(1) 模块接口定义

表 4 PC 模块接口定义

文件	模块接口定义
mux.v	module mux_2_32(input select, input [31:0] mi1, input [31:0] mi2, output [31:0] mo); module mux_4_32(input [1:0] select, input [31:0] mi1, input [31:0] mi2, input [31:0] mi3, input [31:0] mi4, output [31:0] mo); module mux_3_5(input [1:0] select, input [4:0] mi1, input [4:0] mi2, output [4:0] mo);

(2) 接口说明

表 9 EXT 端口说明

序号	信号	方向	描述
1	select	I	选择信号
2	mi（多个）x	I	输入信号
3	mo	O	输出数据

(3)功能定义

表 10 EXT 功能定义

序号	功能	描述
1	选择信号	根据 select 信号选择输入输出

三、控制器设计

在 P3 设计单周期 CPU 时我参考了牛建伟老师的作业图——

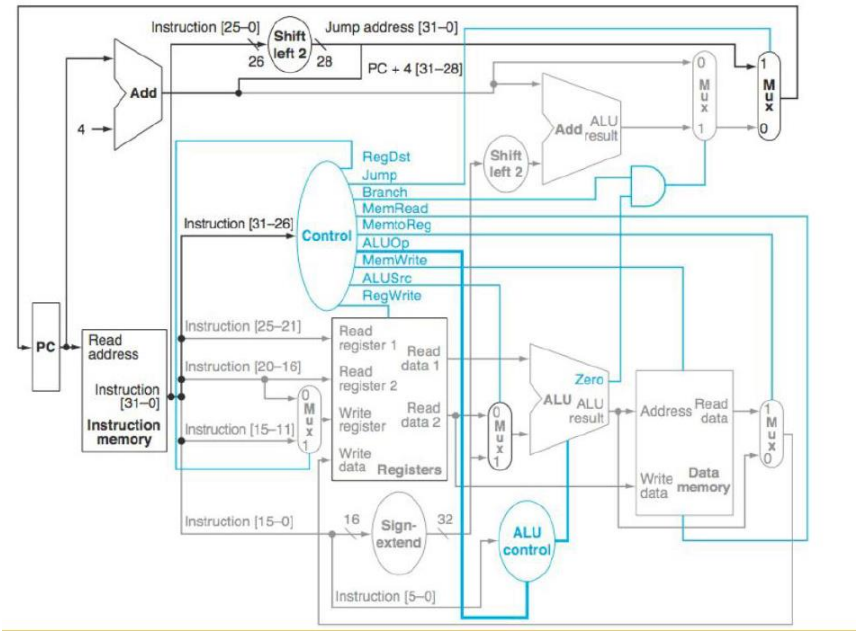

图 1 参考 CPU 电路图

将控制器部分分为 Control 模块和 ALU Control 模块，不过在 P4 中我选择将 Control 模块和 ALU Control 模块合并，从而达到减少接口以简化电路的目的。

1. Control 模块端口与功能说明

表 11 Control 模块端口说明

序号	信号	方向	描述
1	Op [5:0]	I	6 位控制信号
2	Func [5:0]	I	6 位控制信号
3	RegDst [1:0]	O	00：将 Rt 作为 GRF 的 A3 写入地址 01：将 Rd 作为 GRF 的 A3 写入地址 10：将 h1f 作为 GRF 的 A3 写入地址
4	ALUSrc	O	0：将 RD2 作为参与 ALU 计算的第二个数据 1：将 EXT 输出作为参与 ALU 计算的第二个数据
5	MemToReg [1:0]	O	00：将 ALU 计算结果写入 GRF 01：将 DM 输出值写入 GRF 10：将填充到高位的 16 位立即数写入 GRF

6	RegWrite	O	0：GRF 写使能信号无效 1：GRF 写使能信号有效
7	MemRead	O	0：DM 读使能信号无效 1：DM 读使能信号有效
8	MemWrite	O	0：DM 写使能信号无效 1：DM 写使能信号有效
9	Branch [1:0]	O	00：默认操作 01：当前指令为 beq 指令 10：当前指令为 jar 指令 11：当前指令为 jr 指令
10	ExtOp	O	0：进行无符号扩展 1：进行符号扩展
11	ALUOp [2:0]	O	ALU 功能的选择信号： 000：ALU 进行与运算 001：ALU 进行或运算 010：ALU 进行加法运算 011：ALU 进行减法运算

2. Control 模块真值表

表 12 Control 模块真值表

指令	addu	subu	ori	lw	sw	beq	lui	jal	jr
Op 字段	000000	000000	001101	100011	101011	000100	001111	000011	000000
Func 字段	100001	100011	xxxxxx						001000
RegDst	01	01	00	00	00	00	00	10	01
ALUSrc	0	0	1	1	1	0	0	0	0
MemToReg	00	00	00	01	00	00	10	11	00
RegWrite	1	1	1	1	0	0	1	1	1
MemRead	0	0	0	1	0	0	0	0	0
MemWrite	0	0	0	0	1	0	0	0	0
Branch	00	00	00	00	00	01	00	10	11
ExtOp	0	0	1	0	0	0	0	0	0
ALUOp	010	011	001	010	010	011	111	111	010

四、测试程序

1. test branch 代码

```
module tb;

    // Inputs
```

```
    reg clk;
    reg reset;

    // Instantiate the Unit Under Test (UUT)
    mips uut (
        .clk(clk),
        .reset(reset)
    );

    initial begin
        // Initialize Inputs
        clk = 0;
        reset = 1;
        #50;
        reset = 0;

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

        // Add stimulus here

    end

    always #10 clk = ~clk;

endmodule
```

2. mars 测试程序

```
ori $28, $0, 0x0
ori $29, $0, 0x0
ori $1, $0, 0x3456
addu $1, $1, $1
lw $1, 4($0)
sw $1, 4($0)
lui $2, 0x7878
subu $3, $2, $1
lui $5, 0x1234
ori $4, $0, 0x5
nop
sw $5, 65535($4)
lw $3, 65535($4)
beq $3, $5, 0x3
nop
```

```
beq $0, $0, 0x11
nop
ori $7, $3, 0x404
beq $7, $3, 0xe
nop
lui $8, 0x7777
ori $8, $8, 0xffff
subu $0, $0, $8
ori $0, $0, 0x1100
addu $10, $7, $6
ori $8, $0, 0x0
ori $9, $0, 0x1
ori $10, $0, 0x1
addu $8, $8, $10
beq $8, $9, 0xfffe
jal 0x3088
nop
addu $10, $10, $10
beq $0, $0, 0xffff
addu $10, $10, $10
jr $31
```

3. 测试程序结果

```
@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
```

五、思考题

1. 数据通路设计（L0. T2）

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

因为我们设计的 cpu 的 lw/sw 指令以字节为单位，地址是 4 的倍数。而 DM 是以字为单位，所以对 DM 进行操作时，addr 要右移两位来对齐字，由此位数是[11:2]。addr 信号来自 ALU 计算结果。

2）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 是针对哪些部件进行清零复位操作？这些部件为什么需要清零？

针对以下部件进行清零复位操作：PC、GRF、DM

PC 的初始值为 0x00003000，需要清零以回到初始值重新读取指令。

GRF 中的所有寄存器初始值为 0x00000000，DM 中所有地址的初始值为 0x00000000，需要清零，否则可能会影响之后的指令执行。

2. 控制器设计（L0. T4）

1）列举出用 Verilog 语言设计控制器的几种编码方式（至少三种），并给出代码示例。

① 利用 case 语句进行编码

```
module controller(
    input [5:0] op,
    input [5:0] Fu,
    /**************/
    output [2:0] ALUOp,
    output [1:0] RegDst,
    output ALUSrc,
    output [1:0] MemtoReg,
    output RegWrite,
    output MemRead,
    output MemWrite,
    output [1:0] Branch,
    output EXTOp);

    always@(*)
    begin
        case (op)
        6'b000000:
        begin
           ALUOp[2]     <=0;
            ALUOp[1]        <=0;
            ALUOp[0]        <=0;
            RegDst[1]       <=0;
            RegDst[0]       <=1;
            ALUSrc          <=0;
           MemtoReg[1]  <=0;
            MemtoReg[0] <=0;
            RegWrite        <=1;
            MemRead     <=0;
            MemWrite        <=0;

            ExtOp<=0;
            Branch1<=0;


            Branch2<=0;
        end
          ……
```

```
        endcase
    end
endmodule
```

② 利用 assign 语句进行与或识别

```
module controller(
    input [5:0] op,
    input [5:0] Fu,
    /**************/
    output [2:0] ALUOp,
    output [1:0] RegDst,
    output ALUSrc,
    output [1:0] MemtoReg,
    output RegWrite,
    output MemRead,
    output MemWrite,
    output [1:0] Branch,
    output EXTOp);

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

    /*************************************************************
***************/
    assign R        = !op[5] && !op[4] && !op[3] && !op[2] && !op
[1] && !op[0];

    assign addu =  Fu[5] && !Fu[4] && !Fu[3] && !Fu[2] && !Fu[1]
&&  Fu[0] && R;
    assign subu =  Fu[5] && !Fu[4] && !Fu[3] && !Fu[2] &&  Fu[1]
&&  Fu[0] && R;
    assign jr   = !Fu[5] && !Fu[4] &&  Fu[3] && !Fu[2] && !Fu[1]
&& !Fu[0] && R;

    assign ori  = !op[5] && !op[4] &&  op[3] &&  op[2] && !op[1]
&&  op[0];
    assign lw   =  op[5] && !op[4] && !op[3] && !op[2] &&  op[1]
&&  op[0];
    assign sw   =  op[5] && !op[4] &&  op[3] && !op[2] &&  op[1]
&&  op[0];
    assign beq  = !op[5] && !op[4] && !op[3] &&  op[2] && !op[1]
&& !op[0];
    assign lui  = !op[5] && !op[4] &&  op[3] &&  op[2] &&  op[1]
&&  op[0];
```

```
    assign jal  = !op[5] && !op[4] && !op[3] && !op[2] &&  op[1]
&&  op[0];
    /*************************************************************
***************/

    /*****************************************/
    /*000& 001| 010+ 011-*/
    assign ALUOp[2]         = jal||lui;
    assign ALUOp[1]         = beq||sw||lw||lui||addu||subu||jr||j
al;
    assign ALUOp[0]         = beq||lui||ori||subu||jal;
    assign RegDst[1]        = jal;
    assign RegDst[0]        = R;
    assign ALUSrc           = lw||sw||ori;
    assign MemtoReg[1]  = lui||jal;
    assign MemtoReg[0]  = lw||jal;
    assign RegWrite         = R||lui||ori||lw||jal;
    assign MemRead      = lw;
    assign MemWrite         = sw;
    assign Branch[1]        = jal||jr;
    assign Branch[0]        = beq||jr;
    assign EXTOp            = ori;
    /*****************************************/

   endmodule
```

③ 利用宏定义

```
module controller(
    input [5:0] op,
    input [5:0] Fu,
    /**************/
    output [2:0] ALUOp,
    output [1:0] RegDst,
    output ALUSrc,
    output [1:0] MemtoReg,
    output RegWrite,
    output MemRead,
    output MemWrite,
    output [1:0] Branch,
   output EXTOp);

    `define R 6'b000000
```

```
always@(*)
begin
    case (op)
    `R:
    begin
       ALUOp[2]     <=0;
        ALUOp[1]        <=0;
        ALUOp[0]        <=0;
        RegDst[1]       <=0;
        RegDst[0]       <=1;
        ALUSrc          <=0;
       MemtoReg[1]  <=0;
        MemtoReg[0] <=0;
        RegWrite        <=1;
        MemRead     <=0;
        MemWrite        <=0;

        ExtOp<=0;
        Branch1<=0;


        Branch2<=0;
    end
      ……
    endcase
end
endmodule
```

2）根据你所列举的编码方式，说明他们的优缺点。

case 方法较为直观，case 的 6 位 op 与指令的对应不够直观，需要通过注释等实现，代码长度长。

assign 方法采用与或门阵列，更接近底层实现，代码长度较短，但不能简单对应到各指令。

宏定义法相比 case,将指令直接定义为对应的 case ，更加直观；缺点是代码长度也很长。

3. 在线测试相关信息（L0.T5）

1)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 。

《MIPS32® Architecture For Programmers Volume II: The MIPS32® Instruction Set》中 addi 指令的 Operation 部分为：

```
temp ← (GPR[rs]31||GPR[rs]31..0) + sign_extend(immediate) if
temp32 ≠ temp31 then
   SignalException(IntegerOverflow)
else
   GPR[rt] ← temp
endif
```

addiu 指令的 Operation 部分为：

```
temp ← GPR[rs] + sign_extend(immediate)
GPR[rt]← temp
```

忽略溢出的前提下，addi 只完成 `GPR[rt] ← temp ← GPR[rs] + sign_extend(immediate)`，行为与 addiu 相同。

add 指令的 Operation 部分为：

```
temp ← (GPR[rs]31||GPR[rs]31..0) + (GPR[rt]31||GPR[rt]31..0) if
temp32 ≠ temp31 then
   SignalException(IntegerOverflow)
else
   GPR[rd] ← temp
endif
```

addu 指令的 Operation 部分为：

```
temp ← GPR[rs] + GPR[rt]
GPR[rd] ← temp
```

忽略溢出的前提下，add 只完成 `GPR[rd] ← temp ← GPR[rs] + GPR[rt]`，行为与 addu 相同。

2）根据自己的设计说明单周期处理器的优缺点。

优点：设计简单，容易实现。

缺点：不同指令需要的指令周期不同，单周期处理器的吞吐量低、速度慢。

3）简要说明 jal、jr 和堆栈的关系。

jal 和 jr 是配套使用的，jal 调用函数，jr 用于函数返回。

使用 jal 时，将 PC+4 的值存入$ra 寄存器中，函数调用结束后使用 jr $ra 指令返回到调用函数的下一条指令。

在调用函数时，为了避免函数内部操作对$ra 寄存器或其他在函数外部使用的寄存器进行修改，应该将寄存器的值存入堆栈中，调用结束后再从堆栈取出。