Verilog 单周期 CPU 设计文档

作者：李健健

一、 CPU 设计方案综述

（一）总体设计概述

使用 Verilog 开发一个简单的单周期 CPU，总体概述如下：

1. 此 CPU 为 32 位 CPU
2. 此 CPU 为单周期设计
3. 此 CPU 支持的指令集为：

   {addu, subu, ori, lw, sw, beq, lui, jal, jr,nop}
4. nop 机器码为 0x00000000
5. addu, subu 不支持溢出

（二）关键模块定义

1. IM

（1） 端口说明

表 1-IM 端口说明

序号	信号名	方向	描述
1	PC[31:0]	I	时钟信号
2	instr[31:0]	O	指令

（2） 功能定义

表 2-IM 功能定义

序号	功能	描述
1	取指令	就是取指令

2. PC

（1） 端口说明

表 3-PC 端口说明

序号	信号名	方向	描述
1	clk	I	时钟信号
2	reset	I	复位信号
3	NPC	I	下一条指令所在 IM 地址
4	PC	O	当前指令所在 IM 地址

（2） 功能定义

表 4-PC 功能定义

序号	功能	描述
1	存储指令的地址	保存当前执行指令在 IM 中的地址

3. NPC

（1） 端口说明

表 5-NPC 端口说明

序号	信号名	方向	描述
1	branch	I	分支信号
2	j	I	跳转信号
3	jr	I	指令是不是 jr
4	PC[31:0]	I	当前 PC 值
5	RegJump[31:0]	I	跳转寄存器中地址值
6	imm26[25:0]	I	26 位立即数
7	PC4[31:0]	O	PC+4
8	NPC[31:0]	O	根据各种指令计算出的下一个 PC 值

（2） 功能定义

表 6-NPC 功能定义

序号	功能	描述
1	计算下一个 PC 的值	
2	输出 PC4	为 jr 指令写入寄存器做准备

4. GRF

（1） 端口说明

表 7-GRF 端口说明

序号	信号名	方向	描述
1	clk	I	时钟信号
2	reset	I	异步复位信号，将 32 个寄存器中全部清零 1：清零 0：无效
3	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
4	A1[4:0]	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
5	A2[4:0]	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

6	A3[4:0]	I	5 位地址输入信号，指定 32 个寄存器中的一个，作为 RD 的写入地址
7	WD[31:0]	I	32 位写入数据
8	RD1[31:0]	O	输出 A1 指定的寄存器的 32 位数据
9	RD2[31:0]	O	输出 A2 指定的寄存器的 32 位数据

（2） 功能定义

表 8-GRF 功能定义

序号	功能	描述
1	异步复位	reset 为 1 时，将所有寄存器清零
2	读数据	将 A1 和 A2 地址对应的寄存器的值分别通过 RD1 和 RD2 读出
3	写数据	当 WE 为 1 且时钟上升沿来临时，将 WD 写入到 A3 对应的寄存器内部

5. ALU

(1)端口说明

表 9-ALU 端口说明

序号	信号名	方向	描述
1	A[31:0]	I	参与运算的第一个数
2	B[31:0]	I	参与运算的第二个数
3	AluOp[2:0]	I	决定 ALU 做何种操作 000：无符号加 001：无符号减 010：与 011：或 100：将 B[15:0]做为 res[31:16],res[15:0]=0
4	eq	O	A 与 B 是否相等 0：不相等 1：相等
5	res[31:0]	O	A 与 B 做运算后的结果

(2)功能定义

表 10-ALU 功能定义

序号	功能	描述
1	加运算	res = A + B
2	减运算	res = A - B
3	与运算	res = A & B
4	或运算	res = A | B
5	加载高位运算	res = {B[15:0], 16'h0}

6. DM

(1)端口说明

表 11-DM 端口说明

序号	信号名	方向	描述
1	clk	I	时钟信号
2	reset	I	异步复位信号 0：无效 1：内存值全部清零
3	WE	I	写使能信号 0：禁止写入 1：允许写入
4	MemAddr[31:0]	I	读取或写入信号地址
5	WD[31:0]	I	32 为写入数据
6	RD[31:0]	O	32 位读出数据

(2)功能定义

表 12-DM 功能定义

序号	功能	描述
1	异步复位	当 reset 为 1 时，DM 中所有数据清零
2	写入数据	当 WE 有效时，时钟上升沿来临时，WD 中数据写入 A 对应的 DM 地址中
3	读出数据	RD 永远读出 A 对应的 DM 地址中的值

7. EXT

(1)端口说明

表 13-EXT 端口说明

序号	信号名	方向	描述
1	imm16[15:0]	I	代扩展的 16 位信号
2	sign	I	无符号或符号扩展选择信号 0：无符号扩展 1：符号扩展
3	imm32[31:0]	O	扩展后的 32 位的信号

(2)功能定义

表 14-EXT 功能定义

序号	功能	描述
1	无符号扩展	当 sign 为 0 时，将 imm16 无符号扩展输出
2	符号扩展	当 sign 为 1 时，将 imm16 符号扩展输出

8. MUX

（1） 功能说明

行为级建模 Multiplexer

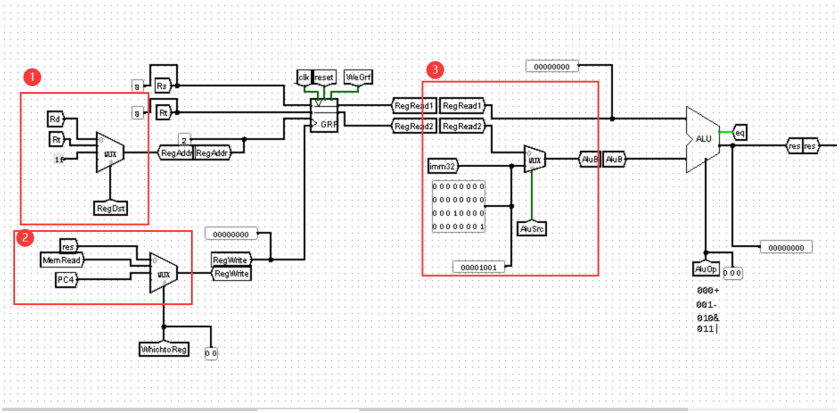

图 1-MUX 应用之处

分别应用于 GRF 写入地址、GRF 写入数据和 ALU 第二个运算符的选择上

9. Controller

(1)端口说明

表 15-Controller 端口说明

序号	信号名	方向	描述
1	instr[31:0]	I	instr[31:26],6 位控制信号
2	eq	I	RegRead1 和 RegRead 是否相等
3	WeGrf	O	GRF 写使能信号 0：禁止写入 1：允许写入
4	WeDm	O	DM 的写入信号 0：禁止写入 1：允许写入
5	RegDst[1:0]	O	GRF 写入地址选择信号 0：Rd 1：Rt
6	WhichtoReg[1:0]	O	将何种数据写入 GRF? 00：ALU 计算结果 01：DM 读出信号 11：upperImm
7	AluSrc	O	参与 ALU 运算的第二个数，来自 GRF 还是 imm 0：来自 GRF 1：imm
8	AluOp[2:0]	O	ALU 的控制信号
9	sign	O	是否对 imm16 进行符号扩展 0：不进行符号扩展 1：进行符号扩展
10	branch	O	instr 是否为 beq 信号 0：不是 1：是

11	JType	O	是不是 J 型指令，只有 j 和 jal 是
12	jr	O	是不是 jr 指令

(2)真值表

表 16-Controller 内部真值对应

端口	addu	subu	ori	lw	sw	lui	beq	jal	jr
op	000000	000000	001101	100011	101011	001111	000100	000011	000000
func	100001	100011							001000
WeGrf	1	1	1	1	0	1	0	1	0
WeDm	0	0	0	0	1	0	0	0	0
RegDst	00	00	01	01	00	01	00	10	00
WhichtoReg	00	00	00	01	00	00	00	10	00
AluSrc	0	0	1	1	1	1	0	0	0
AluOp	000	001	011	000	000	100	000	000	000
sign	0	0	0	1	1	0	1	0	0
branch	0	0	0	0	0	0	1	0	0
JType	0	0	0	0	0	0	0	1	0
jr	0	0	0	0	0	0	0	0	1

二、 测试方案

（1）测试代码：

```
.text
ori $a0,$0,0x100
ori $a1,$a0,0x123
lui $a2,456
lui $a3,0xffff
ori $a3,$a3,0xffff
addu $s0,$a0,$a2
addu $s1,$a0,$a3
addu $s4,$a3,$a3
subu $s2,$a0,$a2
subu $s3,$a0,$a3
sw $a0,0($0)
sw $a1,4($0)
sw $a2,8($0)
sw $a3,12($0)
sw $s0,16($0)
sw $s1,20($0)
sw $s2,24($0)
```

```
sw $s3,44($0)
sw $s4,48($0)
lw $a0,0($0)
lw $a1,12($0)
sw $a0,28($0)
sw $a1,32($0)
ori $a0,$0,1
ori $a1,$0,2
ori $a2,$0,1
beq $a0,$a1,loop1
beq $a0,$a2,loop2
loop1: sw $a0,36($t0)
loop2: sw $a1,40($t0)
jal loop3
jal loop3
sw $s5,64($t0)
ori $a1,$a1,4
jal loop4
loop3:sw $a1,56($t0)
sw $ra,60($t0)
ori $s5,$s5,5
jr $ra
loop4: sw $a1,68($t0)
sw $ra,72($t0)
```

（2）MARS 中运行结果

Name	Number	Value
$zero	0	0x00000000
$at	1	0x00000000
$v0	2	0x00000000
$v1	3	0x00000000
$a0	4	0x00000001
$a1	5	0x00000006
$a2	6	0x00000001
$a3	7	0xffffffff
$t0	8	0x00000000
$t1	9	0x00000000
$t2	10	0x00000000
$t3	11	0x00000000
$t4	12	0x00000000
$t5	13	0x00000000
$t6	14	0x00000000
$t7	15	0x00000000
$s0	16	0x01c80100
$s1	17	0x000000ff
$s2	18	0xfe380100
$s3	19	0x00000101
$s4	20	0xfffffffe
$s5	21	0x00000005
$s6	22	0x00000000
$s7	23	0x00000000
$t8	24	0x00000000
$t9	25	0x00000000
$k0	26	0x00000000
$k1	27	0x00000000
$gp	28	0x00001800
$sp	29	0x00002ffc
$fp	30	0x00000000
$ra	31	0x0000308c
pc		0x000030a4
hi		0x00000000
lo		0x00000000

Address	Value (+0)	Value (+4)	Value (+8)	Value (+12)	Value (+16)	Value (+20)	Value (+24)	Value (+28)
0	0x00000100	0x00000123	0x01c80000	0xffffffff	0x01c80100	0x000000ff	0xfe380100	0x00000100
32	0xffffffff	0x00000000	0x00000002	0x00000101	0xfffffffe	0x00000000	0x00000002	0x00003080
64	0x00000005	0x00000006	0x0000308c	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000

（3）该 CPU 运行输出结果

```
@00003000: $ 4 <= 00000100
@00003004: $ 5 <= 00000123
@00003008: $ 6 <= 01c80000
@0000300c: $ 7 <= ffff0000
@00003010: $ 7 <= ffffffff
@00003014: $16 <= 01c80100
@00003018: $17 <= 000000ff
@0000301c: $20 <= fffffffe
@00003020: $18 <= fe380100
@00003024: $19 <= 00000101
@00003028: *00000000 <= 00000100
@0000302c: *00000004 <= 00000123
@00003030: *00000008 <= 01c80000
@00003034: *0000000c <= ffffffff
@00003038: *00000010 <= 01c80100
@0000303c: *00000014 <= 000000ff
@00003040: *00000018 <= fe380100
@00003044: *0000002c <= 00000101
@00003048: *00000030 <= fffffffe
@0000304c: $ 4 <= 00000100
@00003050: $ 5 <= ffffffff
@00003054: *0000001c <= 00000100
@00003058: *00000020 <= ffffffff
@0000305c: $ 4 <= 00000001
@00003060: $ 5 <= 00000002
@00003064: $ 6 <= 00000001
@00003074: *00000028 <= 00000002
@00003078: $31 <= 0000307c
@0000308c: *00000038 <= 00000002
@00003090: *0000003c <= 0000307c
```

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

MIPS 中以字节为单位，我们的 DM 中，以 32 位的 register 为单位。

addr 是 ALU 单元的输出端口接过来的，代表的是要读取的 DM 存储器的地址。

（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。

if-else, assign, case-endcase

```
always@(*) begin
    if(branch) begin
        NPC = PC + 4 + {{14{imm26[15]}}, imm26[15:0], 2'b00};
    end
    else if (JType) begin
        NPC = {PC[31:28], imm26, 2'b00};
    end
    else if (jr) begin
        NPC = RegJump;
    end
    else begin
        NPC = PC + 4;
    end
end
```

```
assign RegAddr =    (RegDst == 2'b00)? rd:
                    (RegDst == 2'b01)? rt:
                    (RegDst == 2'b10)? 5'h1f:
                    0;
```

```
output reg [31:0] res
);
assign eq = (A == B)? 1:0;
always@(*) begin
    case(AluOp)
        `ADDU :
            res = A + B;
        `SUBU :
            res = A - B;
        `AND :
            res = A & B;
        `OR :
            res = A | B;
        `LUI:
            res = {B[15:0], 16'h0};
        default:
            res = 0;
    endcase
end
```

assign 不需要自己再定义寄存器；case-endcase 和 assign 可以通过宏定义的方式，使代码更加美观，增强可读性。

（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？

特点就是都是存储器，PC、GRF、DM

（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。

addi与addiu的区别在于，当出现溢出时，
addiu忽略溢出，将溢出的最高位舍弃；
addi会报告SignalException(IntegerOverflow)。
故忽略溢出，二者等价。

（五）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，扩展性好，要加什么指令，一目了然

缺点：时钟频率取决于执行时间最长的指令，拖了执行时间短的指令的后腿。