使用Verilog HDL的单周期MIPS CPU设计文档

17373436 林昱同

# 一、模块规格

## 1、NextPC(PC计算)

### 端口定义：

|  |  |  |  |
| --- | --- | --- | --- |
| 端口名 | 方向 | 位宽 | 功能简述 |
| curPC | Input | [31:0] | 当前PC |
| BrImm | Input | [31:0] | 拓展后的BrImm |
| JImm | Input | [25:0] | J指令的Imm |
| JRImm | Input | [31:0] | JR指令的目标地址 |
| Br | Input | 1 | 是否为分支指令 |
| Jump | Input | 1 | 是否为跳转指令 |
| JType | Input | 1 | 使用哪种跳转 |
| NPC | Output | [31:0] | 下一个PC |
| PCAdd4 | Output | [31:0] | PC+4 |

### 功能描述

|  |  |  |
| --- | --- | --- |
| 序号 | 功能名 | 功能描述 |
| 1 | 下一条指令 | 根据指令情况计算PC |

## 2、GRF单元（通用寄存器单元）

### 端口定义

|  |  |  |  |
| --- | --- | --- | --- |
| 端口名 | 方向 | 位宽 | 功能简述 |
| A1 | Input | [4:0] | 读寄存器编号1 |
| A2 | Input | [4:0] | 读寄存器标号2 |
| A3 | Input | [4:0] | 写寄存器编号 |
| WD | Input | [31:0] | 写入数据 |
| clk | Input | 1 | 时钟信号 |
| reset | Input | 1 | 复位信号 |
| WE | Input | 1 | 写入使能 |
| RD1 | Output | [31:0] | 寄存器值1 |
| RD2 | Output | [31:0] | 寄存器值2 |

### 功能描述

|  |  |  |
| --- | --- | --- |
| 序号 | 功能名 | 功能描述 |
| 1 | 复位 | 当reset为1时，所有寄存器值均变为0 |
| 2 | 读取值 | RD1 RD2始终为A1和A2编号的寄存器的值 |
| 3 | 写入 | 当clk上升沿来临时，如WE为1，向A3号寄存器写入WD |

## 3、ALU（算术逻辑单元）

### 接口定义

|  |  |  |  |
| --- | --- | --- | --- |
| 端口名 | 方向 | 位宽 | 功能简述 |
| SrcA | Input | [31:0] | 数据A |
| SrcB | Input | [31:0] | 数据B |
| ALUCtrl | Input | [7:0] | ALU功能控制信号 |
| Shamt | Input | [4:0] | 移位控制 |
| Zero | Output | 1 | 运算结果是否为零 |
| ALUResult | Output | [31:0] | 运算结果 |

### 功能描述

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 序号 | 功能名 | 功能描述 | | |
| ALUCtrl | | ALUResult |
| 0 | 加 | 00000000 | SrcA+SrcB | |
| 1 | 减 | 00000001 | SrcA-SrcB | |
| 2 | 与 | 00000010 | SrcA&SrcB | |
| 3 | 或 | 00000011 | SrcA|SrcB | |
| 4 | 异或 | 00000100 | SrcA^SrcB | |
| 5 | 或非 | 00000101 | !(SrcA|SrcB) | |
| 6 | 逻辑左移 | 00000110 | SrcB<<shamt | |
| 7 | 逻辑右移 | 00000111 | SrcB>>shamt | |
| 8 | 算术右移 | 00001000 | $signed(SrcB>>>shamt) | |
| 9 | 等于比较 | 00001001 | SrcA==SrcB | |
| 10 | 小于比较 | 00001010 | SrcA<SrcB | |
| 11 | 小于等于 | 00001011 | SrcA<=SrcB | |
| 12 | 大于比较 | 00001100 | SrcA>SrcB | |
| 13 | 大于等于 | 00001101 | SrcA>=SrcB | |
| 14 |  |  |  | |
| 15 |  |  |  | |

## 4、DM（数据储存器）

### 接口定义

|  |  |  |  |
| --- | --- | --- | --- |
| 端口名 | 方向 | 位宽 | 功能简述 |
| A | Input | [31:0] | 地址，只有[4:0]有意义 |
| WD | Input | [31:0] | 写入数据 |
| Clk | Input | 1 | 时钟信号 |
| WE | Input | 1 | 写入使能 |
| Reset | Input | 1 | 初始化信号 |
| RD | Output | [31:0] | 读取数据 |

## 功能描述

|  |  |  |
| --- | --- | --- |
| 序号 | 功能名 | 功能描述 |
| 1 | 写入 | 当时钟上升沿来临时，如果Reset为0且WE为1，则再A的位置写入WD |
| 2 | 读取 | RD始终为地址为A的数据的值 |
| 3 | 清空 | Reset为1时，所有数据清0 |
|  |  |  |

## 5、EXT（拓展器）

### 接口定义

|  |  |  |  |
| --- | --- | --- | --- |
| 端口名 | 方向 | 位宽 | 功能简述 |
| Imm | Input | [15:0] | 输入立即数 |
| ExtCtrl | Input | [1:0] | Extender控制信号 |
| Result | Output | [31:0] | 拓展结果 |

## 功能描述

|  |  |  |
| --- | --- | --- |
| 序号 | 功能名 | 功能描述 |
| 1 | 0拓展 | Result={{16{0}},Imm} |
| 2 | 符号拓展 | Result={{16{Imm[15]}},Imm} |
| 3 | 加载到高位 | Result={Imm, {16{0}}} |
| 4 | 1拓展 | Result={{16{1}},Imm} |

## 6、BC(Branch \_Control分支控制)

分支的控制信号既关乎数据流，也关乎控制信号，因此在下面定义控制信号之前定义描述。

由于只需要一个beq，所以这里直接使用一个与门即可。

### 接口定义

|  |  |  |  |
| --- | --- | --- | --- |
| 端口名 | 方向 | 位宽 | 功能简述 |
| Is\_Br | Input | 1 | 是否为分支指令 |
| Zero | Input | 1 | 结果是否为0 |
| Br | Output | 1 | 是否分支跳转 |

## 功能描述

|  |  |  |
| --- | --- | --- |
| 序号 | 功能名 | 功能描述 |
| 1 | 分支判断 | nPC\_Ctrl=Br&!Zero |

# 二、控制信号

## 1、指令编码

### 各个指令的opcode和funct

|  |  |  |
| --- | --- | --- |
|  | opcode[5:0] | funct[5:0] |
| ADDU | 000000 | 100001 |
| SUBU | 000000 | 100011 |
| NOP (SLL) | 000000 | 000000 |
| JR | 000000 | 001000 |
| ORI | 001101 | \ |
| LW | 100011 | \ |
| SW | 101011 | \ |
| BEQ | 000100 | \ |
| LUI | 001111 | \ |
| JAL | 000011 | \ |

NOP即为SLL。

# 2、数据流与控制信号定义

### 数据流：

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | GRF  .A1 | GRF  .A2 | GRF  .A3 | GRF  .WD | ALU  .SrcA | ALU  .SrcB | DM  .Adr | DM  .WD |
| ADDU | IM  .Instr  [25:21] | IM  .instr  [20:16] | IM  .instr  [15:11] | ALU  .Result | GRF  .RD1 | GRF  .RD2 | \ | \ |
| SUBU | IM  .Instr  [25:21] | IM  .instr  [20:16] | IM  .instr  [15:11] | ALU  .Result | GRF  .RD1 | GRF  .RD2 | \ | \ |
| SLL | \ | IM  .instr  [20:16] | IM  .instr  [15:11] | ALU  .Result | \ | GRF  .RD2 | \ | \ |
| JR | IM  .Instr  [25:21] | \ | \ | \ | \ | \ | \ | \ |
| ORI | IM  .instr  [25:21] | \ | IM  .instr  [20:16] | ALU  .Result | GRF  .RD1 | EXT  .Re | \ | \ |
| LW | IM  .instr  [25:21] | \ | IM  .instr  [20:16] | DM  .RD | GRF  .RD1 | EXT  .Re | ALU  .Re | \ |
| SW | IM  .instr  [25:21] | IM  .instr  [20:16] | \ | \ | GRF  .RD1 | EXT  .Re | ALU  .Re | GRF  .RD2 |
| BEQ | IM  .instr  [25:21] | IM  .instr  [20:16] | \ | \ | GRF  .RD1 | GRF  .RD2 | \ | \ |
| LUI | IM  .instr  [25:21] | \ | IM  .instr  [20:16] | ALU  .Result | GRF  .RD1 | EXT  .Re | \ | \ |
| SLL | \ | IM  .instr  [20:16] | IM  .instr  [15:11] | ALU  .Result | \ | GRF  .RD2 | \ | \ |
| JAL | \ | \ | const  “1f” | PC+4 | \ | \ | \ | \ |
| 汇总 | IM  .instr  [25:21] | IM  .instr  [20:16] | IM  .instr  [15:11]  IM  .instr  [20:16]  Const  “1f” | ALU  .Result  DM  .RD  PC+4 | GRF  .RD1 | GRF  .RD2  EXT  .Re | ALU  .Re | GRF  .RD2 |
| 对应控制信号 | \ | \ | RegA3Sel  SaveRA | Data  toReg | \ | ALUBSel | \ | \ |

这里有几个显然固定的数据流没有标识，比如control的opcode和funct、EXT的Imm，BC的zero，ALU的shamt，CPC的BrImm，JRImm，JImm。

通过这些数据通路，确定选择器的控制信号，并绘制出数据通路如下：(下图未加sll与j指令);

### 控制信号真值表：

通过以上的数据通路列表，确定选择信号，并通过器件的使用情况来确定各个元件的写使能信号和模式。

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | Br | Jump | JType | RegA3Sel | SaveRA | DatatoReg | RegWE | ALUBSel | DMWE | EXTCtrl | SLCtrl |
| ADDU | 0 | 0 | X | 0 | 0 | 00 | 1 | 0 | 0 | X | X |
| SUBU | 0 | 0 | X | 0 | 0 | 00 | 1 | 0 | 0 | X | X |
| SLL | 0 | 0 | X | 0 | 0 | 00 | 1 | 0 | 0 | X | X |
| JR | 0 | 1 | 0 | X | X | X | 0 | X | 0 | X | X |
| JALR | 0 | 1 | 0 | 0 | 0 | 10 | 1 | X | 0 | X | X |
| ORI | 0 | 0 | X | 1 | 0 | 00 | 1 | 1 | 0 | 00 | X |
| LW | 0 | 0 | X | 1 | 0 | 01 | 1 | 1 | 0 | 01 | 000 |
| SW | 0 | 0 | X | X | X | X | 0 | 1 | 1 | 01 | 000 |
| BEQ | 1 | 0 | X | X | X | X | 0 | 0 | 0 | 01 | X |
| LUI | 0 | 0 | X | 1 | 0 | 00 | 1 | 1 | 0 | 10 | X |
| JAL | 0 | 1 | 1 | X | 1 | 10 | 1 | X | 0 | X | X |
| J | 0 | 1 | 1 | X | X | X | 0 | X | X | X | X |

为了方便，以上的值为X时，均取0.

|  |  |
| --- | --- |
|  | ALUCtrl |
| ADDU | Add |
| SUBU | Sub |
| SLL | Sll |
| JR | X |
| ORI | Or |
| LW | Add |
| SW | Add |
| BEQ | Eq |
| LUI | Or |
| JAL | X |

## 3、连接控制信号，完成cpu

# 三、CPU的测试

## 1、测试电路

### TB编写

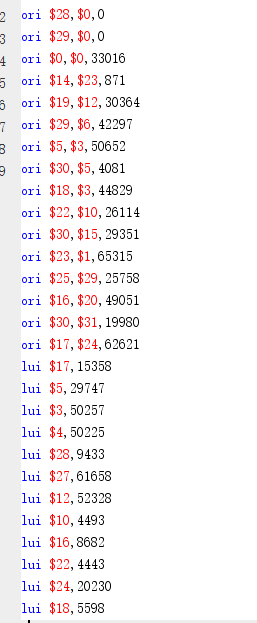
由于写入寄存器和DM时都有输出，可以通过这些输出来调试。

因此只需要使用以下复位即可

## 2、测试集

### 测试用例1

题目给出的测试代码：

图源自讨论区

### 测试用例2

**ori** $s1, $0, 32

**ori** $s2, $0, 0

**ori** $s0, $0, 0

for\_1\_begin:

**beq** $s0, $s1, for\_1\_end

**sw** $s0, 0($s2)

**ori** $t0, $0, 4

**addu** $s2, $s2, $t0

**ori** $t0, $0, 1

**addu** $s0, $s0, $t0

**beq** $0, $0, for\_1\_begin

for\_1\_end:

**nop**

简单的把内存0-31填充上数字的程序。

期望下，内存中0-31地址分别被填充0-31

### 测试用例3

**ori** $0, $0, 0xffffffff

**ori** $1, $0, 0xffffffff

**ori** $2, $0, 0xffffffff

**ori** $3, $0, 0xffffffff

**ori** $4, $0, 0xffffffff

**ori** $5, $0, 0xffffffff

**ori** $6, $0, 0xffffffff

**ori** $7, $0, 0xffffffff

**ori** $8, $0, 0xffffffff

**ori** $9, $0, 0xffffffff

**ori** $10, $0, 0xffffffff

**ori** $11, $0, 0xffffffff

**ori** $12, $0, 0xffffffff

**ori** $13, $0, 0xffffffff

**ori** $14, $0, 0xffffffff

**ori** $15, $0, 0xffffffff

**ori** $16, $0, 0xffffffff

**ori** $17, $0, 0xffffffff

**ori** $18, $0, 0xffffffff

**ori** $19, $0, 0xffffffff

**ori** $20, $0, 0xffffffff

**ori** $21, $0, 0xffffffff

**ori** $22, $0, 0xffffffff

**ori** $23, $0, 0xffffffff

**ori** $24, $0, 0xffffffff

**ori** $25, $0, 0xffffffff

**ori** $26, $0, 0xffffffff

**ori** $27, $0, 0xffffffff

**ori** $28, $0, 0xffffffff

**ori** $29, $0, 0xffffffff

**ori** $30, $0, 0xffffffff

**ori** $31, $0, 0xffffffff

### 测试用例4

**jal** init #init()

**ori** $s0, $0, 1

**ori** $s1, $0, 1024

for\_1\_begin:

**beq** $s0, $s1, for\_1\_end #for i = 1 to 1024

**nop**

**sll** $s2, $s0, 2

**lw** $t0, -4($s2) #get a[i-1]

**ori** $t1, $0, 0x8000 #try zero extern (16'b1000...)

**subu** $t0, $t0, $t1

**sw** $t0, 0($s2) #a[i]=a[i-1]-0x8000 // can less than 0

**ori** $t0, $0, 1

**addu** $s0, $s0, $t0

**j** for\_1\_begin

**nop**

for\_1\_end:

**jal** end #end of program

init: #void init()

**ori** $t0, $0, 0x7111 #to try the lui //lui is {imm,{16{0}}}, not {imm, $rs}

**lui** $t0, 0x00ff

**ori** $t0, 0x8000 #0x0ff8000(9 continouns zero begin from 16's bits)

**sw** $t0, 0($0) #a[0] = 0x00010000

**j** init\_end

init\_end:

**jr** $ra

end:

期望下内存从0x00078000填填充至0xfff80000

### 测试用例5

**lui** $t1, 0x1234

**ori** $t1, $t1, 0x5678

**sb** $t1,0($0)

**sb** $t1,1($0)

**sb** $t1,2($0)

**sb** $t1,3($0)

**sh** $t1,4($0)

**sh** $t1,6($0)

swl $t1,8($0) #8+0

swl $t1,13($0) #12+1

swl $t1,18($0) #16+2

swl $t1,23($0) #20+3

swr $t1,24($0) #24+0

swr $t1,29($0) #28+1

swr $t1,34($0) #32+2

swr $t1,39($0) #36+3

**sw** $t1, 40($0)

# 四、思考题

## 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？

因为DM内部是按字储存的，因此只需要[11:2]这些位数。Addr从ALU来。

### 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？

DM，PC的寄存器，GRF都需要清零。因为如果把CPU看成状态机的话，这些部件就是状态储存器。

## 控制器设计（L0.T4）

### 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. 如logisim一样，设计出表达式

wire sll=op[5]& op[4]& op[3]& op[2]& op[1]& op[0]

1. 使用assign和三目运算符

assign aluop= opcode==…… ? 8’b0:

opcode==…… ? ……

1. 行为级描述，使用switch

case(opcode)

6’b000000:

……

### 根据你所列举的编码方式，说明他们的优缺点

1. 难写难用
2. 好看，很结构级，输出可以直接是wire型变量
3. 更好看，可读性高

## 在线测试相关信息（L0.T5）

### 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 。

因为补码的编码方式就是为了直接使用简单的加法就可以实现正负数的加法，因此直接是否有符号，仅在翻译编码的时候有意义。

### 根据自己的设计说明单周期处理器的优缺点。

优点：

1. 结构简单
2. 每个指令的时长相等
3. 可以没有延迟槽

缺点：

1. 频率严重依赖于关键路径
2. 即使是简单的指令，时间也很长（由于1）
3. 相比于流水线，所有元件相当于只有一个（或几个）可以同时工作，效率低下。

### 简要说明jal、jr和堆栈的关系。

我觉得没有关系。

即：在MIPS的CPU中，在实际的指令执行和cpu的体系结构和实际实现中，这些东西没有关系，基本不用考虑堆栈的问题。

但是，在写汇编代码，即软件实现上，会使用jal和jr来进行程序跳转（子程序调用）时原地址的记录和跳回，并把记录到$ra的地址使用$sp存入堆栈。因此，从这个角度来说，x86体系结构中的堆栈的操作，在MIPS中使用jal jr 和 $sp实现了。