使用Verilog HDL的五级流水线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 |
| PCAdd8 | Output | [31:0] | PC+8 |

### 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 |
| 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 | $signed(SrcA<SrcB) | |
| 11 | 小于等于 | 00001011 | $signed(SrcA<=SrcB) | |
| 12 | 大于比较 | 00001100 | $signed(SrcA>SrcB) | |
| 13 | 大于等于 | 00001101 | $signed(SrcA>=SrcB) | |
| 14 | 小于u | 00001110 | $unsigned(SrcA<SrcB) | |
| 15 | 小于等于u | 00001111 | $unsigned(SrcA<=SrcB) | |
| 16 | 大于u | 00010000 | $unsigned(SrcA>SrcB) | |
| 17 | 大于等于u | 00010001 | $unsigned(SrcA>=SrcB) | |
| 18 | 逻辑左移v | 00010010 | SrcB<<SrcA | |
| 19 | 逻辑右移v | 00010011 | SrcB>>SrcB | |
| 20 | 算术右移v | 00010100 | SrcB>>>SrcB | |

## 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。

### 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 |
| RD1 | Input | 32 | RD1 |
| RD2 | Input | 32 | RD2 |
| RT | Input | 5 | RT |
| 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 | Br=IsBr&~(|(RD1^RD2)); |

## 6．SC(Save Calculator)

### 接口定义

|  |  |  |  |
| --- | --- | --- | --- |
| 端口名 | 方向 | 位宽 | 功能简述 |
| Din | Input | 32 | 要存入的数据 |
| Adrin | Input | 32 | 地址 |
| SLCtrl | Input | 3 | 存取类型控制 |
| Dout | Output | 32 | 输出到DM的数据 |
| Adrout | Output | 32 | 输出到DM的地址 |
| ByteEN | Output | 4 | 要写入哪些位 |

## 功能描述

|  |  |  |
| --- | --- | --- |
| 序号 | 功能名 | 功能描述 |
| 1 | Save word | 存入字 |
| 2 | Save half | 存入半字 |
| 3 | Save byte | 存入字节 |

## 7．LC(Load Calculator)

### 接口定义

|  |  |  |  |
| --- | --- | --- | --- |
| 端口名 | 方向 | 位宽 | 功能简述 |
| memD | Input | 32 | 内存中读取道德数据 |
| GRFD | Input | 32 | RD2 |
| Bytesel | Input | 2 | 地址后两位 |
| SLCtrl | Input | 3 | 存取控制 |
| Dout | Output | 32 | 输出的数据 |

## 功能描述

|  |  |  |
| --- | --- | --- |
| 序号 | 功能名 | 功能描述 |
| 1 | load word | 读取字 |
| 2 | load half | 读取半字，符号拓展 |
| 3 | load halfu | 读取半字，无符号拓展 |
| 4 | load byte | 读取字节，符号拓展 |
| 5 | load byteu | 读取字节，无符号拓展 |

## 8．MDU(乘除运算单元)

### 接口定义

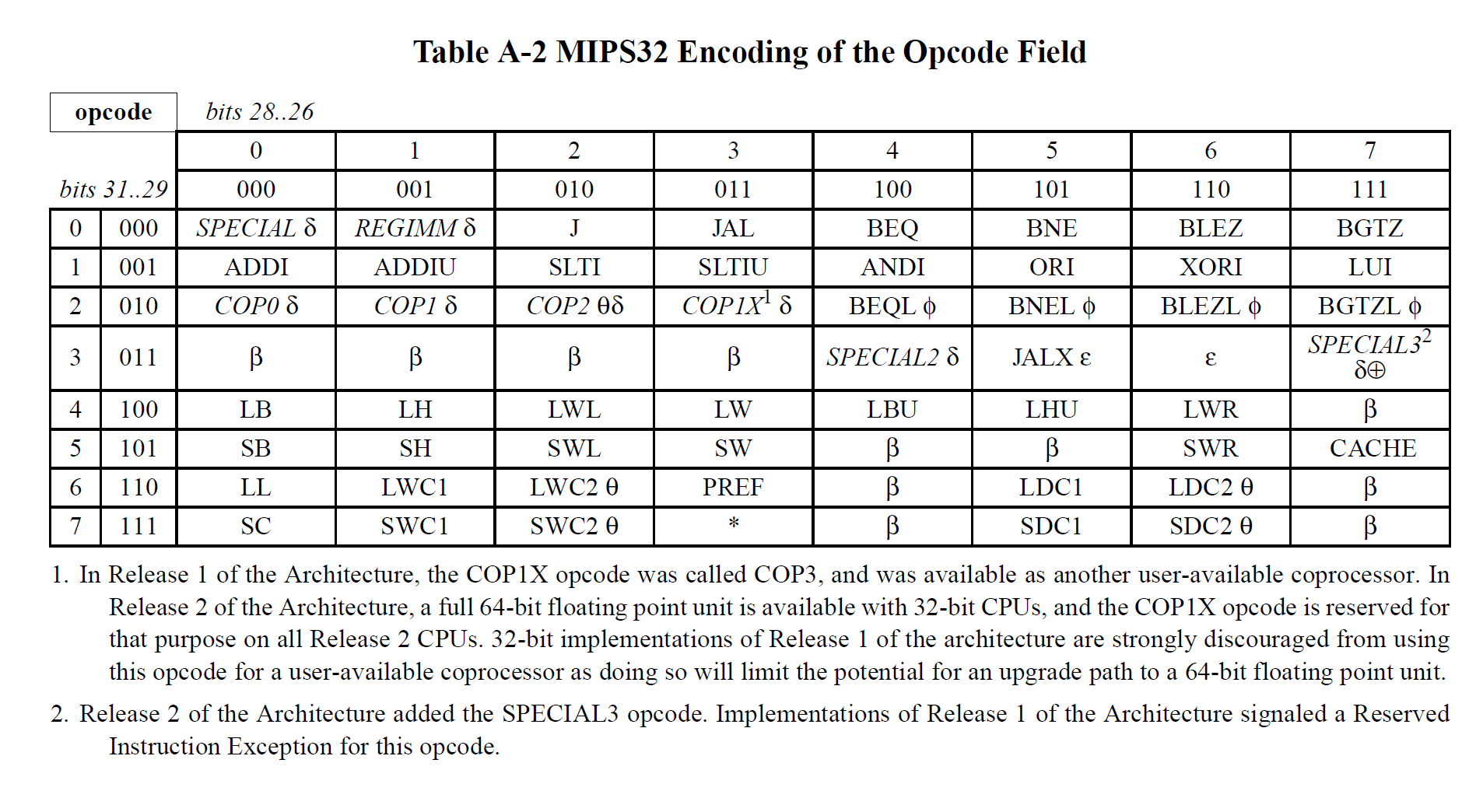
|  |  |  |  |
| --- | --- | --- | --- |
| 端口名 | 方向 | 位宽 | 功能简述 |
| Clk | Input | 1 | 时钟信号 |
| Reset | Input | 1 | 复位信号 |
| SrcA | Input | 32 | 输入A |
| SrcB | Input | 32 | 输入B |
| Start | Input | 1 | 启动信号 |
| MDUCtrl | Input | 3 | 控制信号 |
| Lo | Output | 32 | 低32位结果 |
| Hi | Output | 32 | 高32位结果 |
| Busy | Output | 1 | Busy信号 |

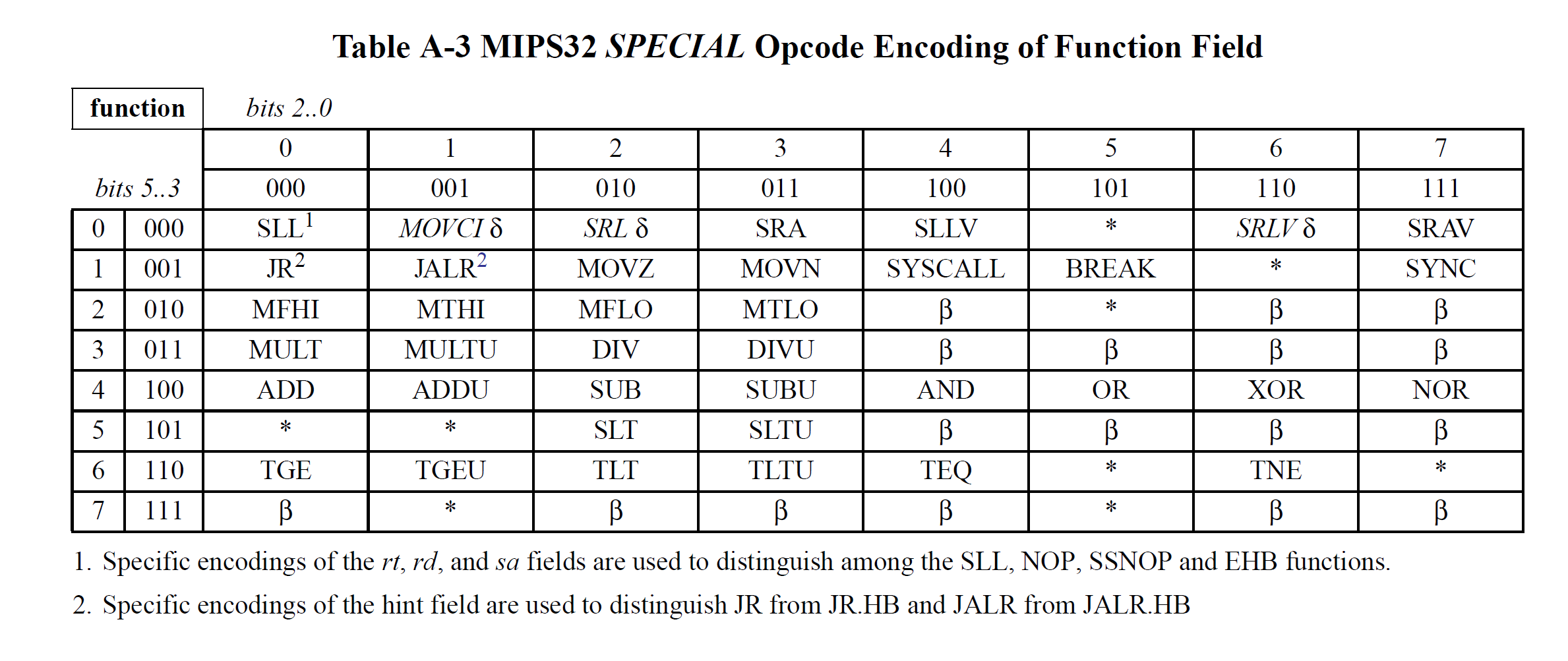
## 功能描述

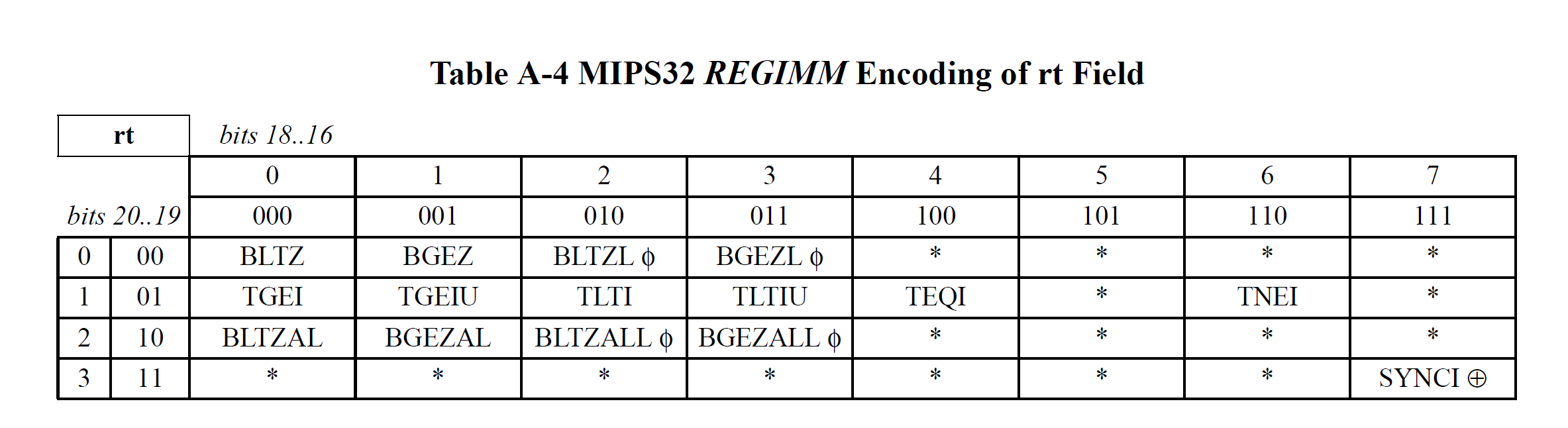
|  |  |  |
| --- | --- | --- |
| 序号 | 功能名 | 功能描述 |
| 1 | MULT | 乘 |
| 2 | MULTU | 无符号成 |
| 3 | DIV | 除 |
| 4 | DIVU | 无符号除 |
| 5 | MTHI/LO | 存入HI/LO |

# 二、功能控制信号与数据通路

## 指令编码







## 分类指令

### 指令集：

MIPS-C3={LB、LBU、LH、LHU、LW、SB、SH、SW、ADD、ADDU、SUB、 SUBU、 MULT、 MULTU、 DIV、 DIVU、 SLL、 SRL、 SRA、 SLLV、SRLV、SRAV、AND、OR、XOR、NOR、ADDI、ADDIU、ANDI、ORI、XORI、LUI、SLT、SLTI、SLTIU、SLTU、BEQ、BNE、BLEZ、BGTZ、BLTZ、BGEZ、J、JAL、JALR、JR、MFHI、MFLO、MTHI、MTLO}

### 分类

R-RD1-RD2-ALU-WD型：ADD、ADDU、SUB、SUBU、AND、OR、XOR、NOR、SLLV、SRLV、SRAV、SLT、SLTU、

R-RD1-Shamt-ALU-WD型：SLL、 SRL、 SRA、

R-RD1-RD2-MDU型：MULT、 MULTU、 DIV、 DIVU、

R-MDU-WD型：MFHI、MFLO、

R-RD1-MDU型：MTHI、MTLO

R-RD1-J：JR、

R-RD1-J-WD：JALR、

I-RD1-Imm-ALU-WD型：ADDI、ADDIU、SLTI、SLTIU、ANDI、ORI、XORI、

型：

I-Imm-WD型：LUI、

I-RD1-Imm-L-WD型：LB、LBU、LH、LHU、LW、

I-RD1-RD2-Imm-S型：SB、SH、SW、

I-RD1-RD2-Br型：BEQ、BNE、

I-RD1-Br型： BLEZ、BGTZ、

I-RD1-RegImm-Br型： BLTZ、BGEZ、

J-Imm型：J、

J-Imm-1f-WD型：JAL、

## 数据流与控制信号定义

### 数据流：

分类后，每一类的控制信号都是相同的。

<数据通路与控制信号.xlsx>

### 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。

<数据通路与控制信号.xlsx>

为了方便，以上的值为X时，均取0.

# 三、冒险

## 数据冒险：

### 分析

能转发就转发，不能就暂停

每一级流水线cpu一旦计算出结果，就可以向前转发。

一旦一个地方需要RD1/RD2，就可以接受转发。

### 策略

对于每个指令，译码出Tnew（计算出结果的时间），Tuse（最晚得到正确寄存器值的时间），使用一个控制器中的流水线寄存器记录并传递，使用大小比较决定转发还是暂停。

## Tnew与Tuse

<数据通路与控制信号.xlsx>

## 暂停机制

### 代码

**assign** A1Tnew**=** IDA1**==**DEA3**&&**DERegWE **?** DETnew **:**

IDA1**==**EMA3**&&**EMRegWE **?** EMTnew **:**

IDA1**==**MWA3**&&**MWRegWE **?** MWTnew **:**

2'd0**;**

**assign** A2Tnew**=** IDA2**==**DEA3**&&**DERegWE **?** DETnew **:**

IDA2**==**EMA3**&&**EMRegWE **?** EMTnew **:**

IDA2**==**MWA3**&&**MWRegWE **?** MWTnew **:**

2'd0**;**

**assign** stall**=** **(**IDA1**!=**0**&&**A1Tnew**>**Tuse1**)||(**IDA2**!=**0**&&**A2Tnew**>**Tuse2**);**

## 转发机制

### 转发目的地

均为外部转发。

GRF的输出的RD1，RD2；

D/Ereg和E/Mreg的输出的RD1，RD2.

## 逻辑

当源寄存器已经计算出结果（Tnew=0）并且转发目标的A1A2为要写入的A3时，进行转发。

### 代码

**assign** D1FWSel**=** IDA1**==**5'd0 **?** 2'd0 **:**

IDA1**==**DEA3**&&**DETnew**==**0**&&**DERegWE **?** 2'd1 **:**

IDA1**==**EMA3**&&**EMTnew**==**0**&&**EMRegWE **?** 2'd2 **:**

IDA1**==**MWA3**&&**MWTnew**==**0**&&**MWRegWE **?** 2'd3 **:**

2'd0**;**

**assign** D2FWSel**=** IDA2**==**5'd0 **?** 2'd0 **:**

IDA2**==**DEA3**&&**DETnew**==**0**&&**DERegWE **?** 2'd1 **:**

IDA2**==**EMA3**&&**EMTnew**==**0**&&**EMRegWE **?** 2'd2 **:**

IDA2**==**MWA3**&&**MWTnew**==**0**&&**MWRegWE **?** 2'd3 **:**

2'd0**;**

**assign** E1FWSel**=** DEA1**==**5'd0 **?** 2'd0 **:**

DEA1**==**EMA3**&&**EMTnew**==**0**&&**EMRegWE **?** 2'd1 **:**

DEA1**==**MWA3**&&**MWTnew**==**0**&&**MWRegWE **?** 2'd2 **:**

2'd0**;**

**assign** E2FWSel**=** DEA2**==**5'd0 **?** 2'd0 **:**

DEA2**==**EMA3**&&**EMTnew**==**0**&&**EMRegWE **?** 2'd1 **:**

DEA2**==**MWA3**&&**MWTnew**==**0**&&**MWRegWE **?** 2'd2 **:**

2'd0**;**

**assign** M1FWSel**=** EMA1**==**5'd0 **?** 2'd0 **:**

EMA1**==**MWA3**&&**MWTnew**==**0**&&**MWRegWE **?** 2'd1 **:**

2'd0**;**

**assign** M2FWSel**=** EMA2**==**5'd0 **?** 2'd0 **:**

EMA2**==**MWA3**&&**MWTnew**==**0**&&**MWRegWE **?** 2'd1 **:**

2'd0**;**

## 资源冒险

MDU可能会出现资源冒险。

因此一旦当前使用MDU（start==1），MDU的busy为1，当前IF/ID级需要使用MDU，则暂停。

# 四、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。

因此只需要使用复位即可

先设置reset=1；

在#100后设置reset=0

## 2、测试集

### 测试用例1

<sample1.asm>

网站上所给出的测试代码。

### 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

填充所有寄存器

### 测试用例4

**jal** init #init()

nop

**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

nop

init\_end:

**jr** $ra

nop

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)

测试各种s指令

### 测试用例6

**li** $t1 0x12345678

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

**lui** $t2, 0xffff

**ori** $t2, 0xffff

**lb** $t2, 0($0)

**lui** $t2, 0xffff

**ori** $t2, 0xffff

**lb** $t2, 1($0)

**lui** $t2, 0xffff

**ori** $t2, 0xffff

**lb** $t2, 2($0)

**lui** $t2, 0xffff

**ori** $t2, 0xffff

**lb** $t2, 3($0)

**lui** $t2, 0xffff

**ori** $t2, 0xffff

**lh** $t2, 0($0)

**lui** $t2, 0xffff

**ori** $t2, 0xffff

**lh** $t2, 2($0)

**lui** $t2, 0xffff

**ori** $t2, 0xffff

**lbu** $t2, 0($0)

**lui** $t2, 0xffff

**ori** $t2, 0xffff

**lbu** $t2, 1($0)

**lui** $t2, 0xffff

**ori** $t2, 0xffff

**lbu** $t2, 2($0)

**lui** $t2, 0xffff

**ori** $t2, 0xffff

**lbu** $t2, 3($0)

**lui** $t2, 0xffff

**ori** $t2, 0xffff

**lhu** $t2, 0($0)

**lui** $t2, 0xffff

**ori** $t2, 0xffff

**lhu** $t2, 2($0)

**lui** $t2, 0xffff

**ori** $t2, 0xffff

**lw** $t2, 0($0)

**lui** $t2, 0xffff

**ori** $t2, 0xffff

lwl $t2, 0($0)

**lui** $t2, 0xffff

**ori** $t2, 0xffff

lwl $t2, 1($0)

**lui** $t2, 0xffff

**ori** $t2, 0xffff

lwl $t2, 2($0)

**lui** $t2, 0xffff

**ori** $t2, 0xffff

lwl $t2, 3($0)

**lui** $t2, 0xffff

**ori** $t2, 0xffff

lwr $t2, 0($0)

lui $t2, 0xffff

ori $t2, 0xffff

lwr $t2, 1($0)

lui $t2, 0xffff

ori $t2, 0xffff

lwr $t2, 2($0)

lui $t2, 0xffff

ori $t2, 0xffff

lwr $t2, 3($0)

### 测试用例7

本测试用例使用代码生成，代码见思考题，枚举了各种Tuse和Tnew。

由于测试代码太大，这里放不下，测试代码附在压缩文件中。

<sample7.asm>

### 测试用例8

将题目中给出的数据去除大部分连续的nop

<sample5.asm>

### 测试用例9

乘法简单测试，并看波形。

**li** $t1, 0x12345678

**li** $t2, 0x87654321

**mult** $t1, $t2

**mflo** $t1

**mfhi** $t2

### 测试用例10

随机生成的程序。

<sample6.asm>

### 测试用例11

由程序生成的程序。

程序由思考题最后一题改造而成。

<sample8.asm>

### 测试用例12

<sample9.asm>

### 测试用例13

MDU的测试

<sample11.asm>

# 思考题

## 为什么需要有单独的乘除法部件而不是整合进ALU？为何需要有独立的HI、LO寄存器？

不是整合进ALU是因为乘除所需的时间不是单个周期，可以看作一个异步的部件。

有独立的寄存器是因为MIPS体系结构规定的。之所以这么规定，我认为

1. 是因为32位乘32位的乘法可以的到64位的结果，使用两个寄存器才能存下，而mips的R型指令只能指定一个目标寄存器
2. 是因为乘除模块得到结果大多需要超过一个周期，因此需要在MDU中存下来，之后使用MFHI/LO取出数据。

## 参照你对延迟槽的理解，试解释“乘除槽”。

延迟槽是J/Branch指令后必须执行的一条指令，产生的原因是J/Br指令在ID级才能对PC产生影响，而此时IF级会得到PC+4的指令。

而乘除槽，我觉得可能是指乘除法后面的一些指令会被延迟（虽然这和延迟槽几乎完全没有什么共性）。

## 为何上文文末提到的lb等指令使用的数据扩展模块应在 MEM/WB 之后，而不能在 DM 之后?

DM的写入时所占用的cpu的时间相比于读取是很短的。因此，实际上，CPU的关键路径是DM的读取，如果把该模块放在DM后，会计入关键路径的时间，因此不行。

## 举例说明并分析何时按字节访问内存相对于按字访问内存性能上更有优势。（Hint： 考虑C语言中字符串的情况）

一旦需要存取小于一个字的数据的时候，就会有优势。

比如c语言中的short，char，分别存取的时half和byte，如果时按字访问的话，就需要先取出值，再写入值。

## 如何概括你所设计的CPU的设计风格？为了对抗复杂性你采取了哪些抽象和规范手段？

风格：暴力转发，不行暂停。

抽象方法：把每个指令的需求和的可以转发抽象成Tues和Tnew。

规范手段：对A1，A2，A3无需求时，把A1，A2，A3设为0.

## 你对流水线CPU设计风格有何见解？

我觉得，CPU的设计风格应该简洁，统一。

尽量不要产生对某种指令特判，让同类指令的行为一致，同时保证拓展性。

将CPU分层，规定好各层之间的接口，进行抽象。

### 在本实验中你遇到了哪些不同指令组合产生的冲突？你又是如何解决的？相应的测试样例是什么样的？请有条理的罗列出来。(**非常重要**)

搭建cpu时，不用考虑**具体的**指令组合的造成冲突。

我遇到了的冲突（数据冒险）情况并解决的方法见下表：（Tnew表示某一级流水线寄存器中存在一个与写入寄存器当前FD的指令所需要的寄存器相同的指令的剩余Tnew）

|  |  |  |  |
| --- | --- | --- | --- |
|  | Tuse=0 | Tuse=1 | Tuse=2 |
| Tnew=1 | 暂停 | 转发或无影响 | 转发或无影响 |
| Tnew=2 | 暂停 | 暂停 | 转发或无影响 |
| Tnew=3 | 暂停 | 暂停 | 暂停 |

解决方法：当某一级流水线寄存器的A3和当前FD级的寄存器的**需要用到**的A1 A2一样时，参照上表进行暂停或转发。

在构建测试样例时，可以考虑具体的指令组合：

这里使用c语言编写程序，枚举了连续4条指令，前三条指令可能分别为Tnew=1、2、3的，第四条指令为Tuse为0、1、2的，生成测试程序的代码如下，所生成的代码附在压缩文件中。

#include<cstdio>

char new1**[**4**][**1000**]={**""**,**"lui $30, 0x1234"**,**"addu $30, $28, $29"**,**"lw $30,0($0)"**};**

char new2**[**4**][**1000**]={**""**,**"jal j\_%d"**,**"subu $31, $29, $28"**,**"lw $31,0($0)"**};**

char new3**[**4**][**1000**]={**""**,**"lui $31, 0x1234"**,**"addu $30, $28, $29"**,**"lw $31,0($0)"**};**

char use**[**3**][**1000**]={**"beq $30, $31, next\_%d"**,**"addu $1, $30, $31"**,**"sw $31, 4($0)"**};**

int main**()**

**{**

freopen**(**"sample2.asm"**,**"w"**,**stdout**);**

puts**(**"ori $28, $0, 0x1234"**);**

puts**(**"ori $29, $0, 0x5678"**);**

puts**(**"sw $29, 0($0)"**);**

int cnt**=**0**;**

**for(**int i**=**1**;**i**<=**3**;**i**++)**

**{**

**for(**int j**=**1**;**j**<=**3**;**j**++)**

**{**

**for(**int k**=**1**;**k**<=**3**;**k**++)**

**{**

**for(**int l**=**0**;**l**<=**2**;**l**++)**

**{**

cnt**++;**

puts**(**new1**[**i**]);**

**if(**j**==**1**)**

**{**

printf**(**new2**[**j**],**cnt**);**

putchar**(**10**);**

**}**

**else**

puts**(**new2**[**j**]);**

puts**(**new3**[**k**]);**

printf**(**"j\_%d:\n"**,**cnt**);**

**if(**l**==**0**)**

**{**

printf**(**use**[**l**],**cnt**);**

putchar**(**10**);**

**}**

**else**

puts**(**use**[**l**]);**

puts**(**"nop"**);**

printf**(**"next\_%d:\n"**,**cnt**);**

puts**(**"nop"**);**

puts**(**"nop"**);**

puts**(**"nop"**);**

puts**(**"nop"**);;**

puts**(**"nop"**);**

**}**

**}**

**}**

**}**

**}**