P3-Logism单周期CPU设计文档

17373499 毛豫新

目录

[一． CPU设计文档 3](#_Toc529982129)

[1. 模块规格 3](#_Toc529982130)

[(1).IFU 3](#_Toc529982131)

[(2).RF 4](#_Toc529982132)

[(3).ALU 4](#_Toc529982133)

[(4).DM 5](#_Toc529982134)

[(5).EXT 5](#_Toc529982135)

[2. 控制器设计 6](#_Toc529982136)

[(1).控制信号功能说明 6](#_Toc529982137)

[3. 测试程序 7](#_Toc529982138)

[二．思考题 10](#_Toc529982139)

[1. 模块规格问题 10](#_Toc529982140)

[(1). 若PC（程序计数器）位数为30位，试分析其与32位PC的优劣。 10](#_Toc529982141)

[(2). 现在我们的模块中 IM使用ROM， DM使用RAM， GRF使用寄存器，这种做法合理吗？ 请给出分析，若有改进意见也请一并给出。 10](#_Toc529982142)

[2. 控制器设计问题 10](#_Toc529982143)

[(1). 结合上文给出的样例真值表，给出RegDst， ALUSrc， MemtoReg，RegWrite, nPC\_Sel, ExtOp与op和func有关的布尔表达式（表达式中只能使用“与、或、非”3 种基本逻辑运算。） 10](#_Toc529982144)

[(2). 充分利用真值表中的 X 可以将以上控制信号化简为最简单的表达式， 请给出化简后的形式。 11](#_Toc529982145)

[(3). 事实上，实现nop空指令，我们并不需要将它加入控制信号真值表，为什么？请给出你的理由。 11](#_Toc529982146)

[3. 测试CPU思考题 11](#_Toc529982147)

[(1). 前文提到，“可能需要手工修改指令码中的数据偏移”，但实际上只需再增加一个 DM片选信号,就可以解决这个问题。请阅读相关资料并设计一个 DM 改造方案使得无需手工修改数据偏移。 11](#_Toc529982148)

[(2). 除了编写程序进行测试外，还有一种验证CPU设计正确性的办法——形式验证。 形式验证的含义是根据某个或某些形式规范或属性，使用数学的方法证明其正确性或非正确性。请搜索“形式验证（Formal Verification)"了解相关内容后，简要阐述相比与测试，形式验证的优劣。 12](#_Toc529982149)

# CPU设计文档

处理器为**32位单周期**处理器，支持的指令集为：**{addu, subu, ori, lw, sw, beq, lui, nop}。**

## 模块规格

该处理器包含6个模块，分别为IFU（取指令单元），RF（寄存器堆），ALU（算术逻辑单元），DM（数据存储器），EXT（扩展单元），Controller（控制器）

### (1).IFU

IFU包含存储PC的寄存器，指令存储器IM，以及由当前PC计算出下一个PC的相关逻辑。

PC的起始地址为0x00000000

IM为ROM，容量为32bit\*32

IFU模块功能

|  |  |  |
| --- | --- | --- |
| 序号 | 功能 | 功能描述 |
| 1 | 复位 | 当复位信号有效，PC被设置为0x00000000 |
| 2 | NPC选择 | 该信号有效，PC=PC+4+offset||02，否则PC=PC+4 |
| 3 | 输出 | 输出当前PC所指的指令 |
| 4 | 输入 | 读入PC的偏移量offset |

IFU端口说明

|  |  |  |  |
| --- | --- | --- | --- |
| 端口名称 | 位宽 | 方向 | 描述 |
| NpcSel | 1 | I | 当NpcSel无效时，PC=PC+4；否则PC=PC+4+偏移量 |
| reset | 1 | I | 当时钟信号在上升沿，该信号有效，则PC重置为0 |
| clk | 1 | I | 时钟信号 |
| offset | 32 | I | PC的偏移量 |
| instr | 32 | O | 输出当前PC在指令存储器所指的指令 |

### (2).RF

RF包含从1号寄存器到31号寄存器，共31个具有写使能寄存器。（0号寄存器直接由输出0信号来代替）

RF模块功能

|  |  |  |
| --- | --- | --- |
| 序号 | 功能 | 功能描述 |
| 1 | 复位 | 当复位信号有效，所有寄存器被设置为0 |
| 2 | 输入 | 该信号有效，A3所在的寄存器能够被写入数据 |
| 3 | 输出 | 输出A1和A2所在寄存器的值 |

RF端口说明

|  |  |  |  |
| --- | --- | --- | --- |
| 端口名称 | 位宽 | 方向 | 描述 |
| A1 | 5 | I | OA需要输出寄存器的地址 |
| A2 | 5 | I | OB需要输出寄存器的地址 |
| A3 | 5 | I | WD需要写入寄存器的地址 |
| WE | 1 | I | 该信号有效，寄存器堆可以写入数据 |
| WD | 32 | I | 寄存器堆需要写入的数据 |
| clk | 1 | I | 时钟信号 |
| reset | 1 | I | 当时钟信号在上升沿，该信号有效，则所有寄存器重置为0 |
| OA | 32 | O | 输出A1所指的寄存器的数据 |
| OB | 32 | O | 输出A2所指的寄存器的数据 |

### (3).ALU

ALU提供了CPU所需要的相关逻辑运算功能。

ALU模块功能

|  |  |  |
| --- | --- | --- |
| 序号 | 功能 | 功能描述 |
| 1 | 相等判断 | 当A=B时，该信号为1，否则为0 |
| 2 | 加运算 | Aluctr位000，O=A+B |
| 3 | 减运算 | Aluctr位001，O=A-B |
| 4 | 或运算 | Aluctr位010，O=AorB |
| 5 | 输出B | Aluctr位011，直接输出B数 |
| 6 | 输出 | 输出A1和A2所在寄存器的值 |

ALU端口说明

|  |  |  |  |
| --- | --- | --- | --- |
| 端口名称 | 位宽 | 方向 | 描述 |
| A | 32 | I | 读入32位数A |
| B | 32 | I | 读入32位数B |
| ALUctr | 3 | I | 逻辑运算操作信号，详细请见功能 |
| equal | 1 | O | A=B输出1，否则输出0 |
| O | 32 | O | 输出运算结果 |

### (4).DM

DM包含了一个容量位32bit\*32的RAM

DM模块功能

|  |  |  |
| --- | --- | --- |
| 序号 | 功能 | 功能描述 |
| 1 | 复位 | 当复位信号有效时，RAM数据清零 |
| 2 | 写入数据 | 当写使能信号有效，DM将数据D写入A所指的地址内 |
| 3 | 输出数据 | DM将输出A所指的地址内存储的数据 |

DM端口说明

|  |  |  |  |
| --- | --- | --- | --- |
| 端口名称 | 位宽 | 方向 | 描述 |
| A | 5 | I | 写入或输出的地址 |
| WD | 32 | I | 写入的数据 |
| WE | 1 | I | 写使能 |
| OD | 1 | O | 输出的数据 |
| clk | 1 | I | 时钟信号 |
| reset | 1 | I | 复位信号 |

### (5).EXT

EXT模块功能

|  |  |  |
| --- | --- | --- |
| 序号 | 功能 | 功能描述 |
| 1 | 符号扩展 | 当操作信号为00，输出为输入的16位数的32位符号扩展 |
| 2 | 零扩展 | 当操作信号为01，输出为输入的16位数的32位零扩展 |
| 3 | 载入高位 | 当操作信号为10，把输入的16位数载入高16位，低16位为零 |

EXT端口说明

|  |  |  |  |
| --- | --- | --- | --- |
| 端口名称 | 位宽 | 方向 | 描述 |
| A | 16 | I | 需要扩展的16位数据 |
| ExtOp | 2 | I | 扩展的操作信号 |
| O | 32 | O | 输出的扩展后的32位数据 |

## 控制器设计

### (1).控制信号功能说明

Controller模块功能

|  |  |  |
| --- | --- | --- |
| 序号 | 功能 | 功能描述 |
| 1 | PC偏移控制 | 根据操作码来输出控制PC偏移的信号 |
| 2 | RF写入寄存器选择 | 根据操作码来选择RF写入的寄存器 |
| 3 | RF写使能 | 输出RF的写使能信号 |
| 4 | 立即数扩展操作 | 输出EXT需要进行的操作信号 |
| 5 | 数据存储器写使能 | 输出DM的写使能信号 |
| 6 | 写入RF数据的选择 | 根据操作码控制写入RF数据的来源 |
| 7 | ALU操作信号 | 输出控制ALU操作的信号 |
| 8 | ALU操作数来源选择 | 控制ALU的B输入数据来源的选择信号 |

Controller端口说明

|  |  |  |  |
| --- | --- | --- | --- |
| 端口名称 | 位宽 | 方向 | 描述 |
| opcode | 5 | I | 指令的操作码 |
| funct | 5 | I | R型指令的功能码 |
| AluEqual | 1 | I | 接受ALU输出操作数是否相等的信号 |
| NpcSel | 1 | O | 输出分支指令生效的信号 |
| RegDst | 1 | O | 输出RF写入寄存器选择信号 |
| RFWrEn | 1 | O | RF的写使能信号 |
| ExtOp | 2 | O | Ext的操作信号 |
| DMWrEn | 1 | O | DM的写使能信号 |
| DtoRF | 1 | O | 写入RF的数据选择信号 |
| AluSrc | 1 | O | 参与ALU的B数据选择信号 |
| AluOp | 3 | O | ALU的操作信号 |

Controller真值表

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| funct | 100001 | 100011 | None | None | None | None | None |
| opcode | 000000 | 000000 | 001101 | 100011 | 101011 | 001111 | 000100 |
| Instruction | addu | subu | ori | lw | sw | lui | beq |
| NpcSel | 0 | 0 | 0 | 0 | 0 | 0 | alu.equal |
| RegDst(0:rt,1:rd) | 1 | 1 | 0 | 0 | x | 0 | x |
| RFWrEn | 1 | 1 | 1 | 1 | 0 | 1 | 0 |
| DMWrEn | 0 | 0 | 0 | 0 | 1 | 0 | 0 |
| DtoRF(0:alu,1:DM) | 0 | 0 | 0 | 1 | x | 0 | x |
| AluSrc(0:OB,1:imm) | 0 | 0 | 1 | 1 | 1 | 1 | 0 |
| ALuOp[2:0](alu[3]=0) | add | substrct | or | add | add | B | substract |
| ALuOp[0] | 0 | 1 | 0 | 0 | 0 | 1 | 1 |
| ALuOp[1] | 0 | 0 | 1 | 0 | 0 | 1 | 0 |
| ExtOp[1:0] | None | None | Zero\_ext | Sign\_ext | Sign\_ext | movetohi | Sign\_ext |
| ExtOp[0] | x | x | 0 | 1 | 1 | 0 | 1 |
| ExtOp[1] | x | x | 0 | 0 | 0 | 1 | 0 |

## 测试程序

测试的汇编程序：

nop

ori $1, $1, 0x0001

ori $2, $2, 0x8000

ori $3, $2, 0x7fff

addu $4, $1, $2

subu $5, $2, $1

lui $6, 0x0fff

lui $6, 0xffff

ori $7, 0x0004

sw $1, 0($7)

sw $2, 4($7)

sw $3, -4($7)

lw $1, 0($7)

lw $2, 4($7)

lw $3, -4($7)

ori $8, $8, 1

ori $9, $9, 2

beq $1, $0, next

ori $1, 0x2333

beq $0, $0, next

ori $1, 0x0001

next:

addu $1, $1, $1

sub $9, $9, $8

beq $9, $8, next

期望结果：(每一条操作指令执行后对应CPU的变化)

PC <= 0x00000004

$1 <= 0x00000001; PC <= 0x00000008

$2 <= 0x00008000; PC <= 0x0000000c

$3 <= 0x0000ffff; PC <= 0x00000010

$4 <= 0x00008001; PC <= 0x00000014

$5 <= 0x00007fff; PC <= 0x00000018

$6 <= 0x0fff0000; PC <= 0x0000001c

$6 <= 0xffff0000; PC <= 0x00000020

$7 <= 0x00000004; PC <= 0x00000024

mem[0x4] <= 0x00000001; PC <= 0x00000028

mem[0x8] <= 0x00008000; PC <= 0x0000002c

mem[0x0] <= 0x0000ffff; PC <= 0x00000030

$1 <= mem[0x4]; PC <= 0x00000034

$2 <= mem[0x8]; PC <= 0x00000038

$3 <= mem[0x0]; PC <= 0x0000003c

$8 <= 1; PC <= 0x00000040

$9 <= 2; PC <= 0x00000044

PC <= 0x00000048

$1 <= 0x00002333; PC <= 0x0000004c

PC <= 0x00000054

$1 <= 0x00004666; PC <= 0x00000058

$9 <= 1; PC <= 0x0000005c

PC <= 0x00000054

$1 <= 0x00008ccc; PC <= 0x00000058

$9 <= 0; PC <= 0x0000005c

PC <= 0x00000060

实际结果：

PC <= 0x00000004

$1 <= 0x00000001; PC <= 0x00000008

$2 <= 0x00008000; PC <= 0x0000000c

$3 <= 0x0000ffff; PC <= 0x00000010

$4 <= 0x00008001; PC <= 0x00000014

$5 <= 0x00007fff; PC <= 0x00000018

$6 <= 0x0fff0000; PC <= 0x0000001c

$6 <= 0xffff0000; PC <= 0x00000020

$7 <= 0x00000004; PC <= 0x00000024

mem[0x4] <= 0x00000001; PC <= 0x00000028

mem[0x8] <= 0x00008000; PC <= 0x0000002c

mem[0x0] <= 0x0000ffff; PC <= 0x00000030

$1 <= mem[0x4]; PC <= 0x00000034

$2 <= mem[0x8]; PC <= 0x00000038

$3 <= mem[0x0]; PC <= 0x0000003c

$8 <= 1; PC <= 0x00000040

$9 <= 2; PC <= 0x00000044

PC <= 0x00000048

$1 <= 0x00002333; PC <= 0x0000004c

PC <= 0x00000054

$1 <= 0x00004666; PC <= 0x00000058

$9 <= 1; PC <= 0x0000005c

PC <= 0x00000054

$1 <= 0x00008ccc; PC <= 0x00000058

$9 <= 0; PC <= 0x0000005c

PC <= 0x00000060

与期望结果相同

# 二．思考题

## 1. 模块规格问题

### (1). 若PC（程序计数器）位数为30位，试分析其与32位PC的优劣。

若PC位数为30位，位数不是2的整数幂，在输入到其他单元的时候会出现需要扩展的现象。其次，因为32位PC读取的指令个数是30位PC的4倍。

### (2). 现在我们的模块中 IM使用ROM， DM使用RAM， GRF使用寄存器，这种做法合理吗？ 请给出分析，若有改进意见也请一并给出。

我认为实际当中IM不应该使用ROM,因为计算机会存在从DM中读入指令的到IM当中的操作。IM与DM分开，会导致它们的数据起始地址一致，可能会出现无法分辨某条数据是IM里的还是DM里的。

## 2. 控制器设计问题

### (1). 结合上文给出的样例真值表，给出RegDst， ALUSrc， MemtoReg，RegWrite, nPC\_Sel, ExtOp与op和func有关的布尔表达式（表达式中只能使用“与、或、非”3 种基本逻辑运算。）

REGDST =~OP[0] ~OP[1] ~OP[2] ~OP[3] ~OP[4] ~OP[5]

ALUSRC =OP[0] ~OP[1] OP[2] OP[3] ~OP[4] ~OP[5]+ OP[0] OP[1] ~OP[2] ~OP[4] OP[5]

MEMTOREG=OP[0] OP[1] ~OP[2] ~OP[3] ~OP[4] OP[5]

REGWRITE = ~OP[0] ~OP[1] ~OP[2] ~OP[3] ~OP[4] ~OP[5]+ OP[0] ~OP[1] OP[2] OP[3] ~OP[4] ~OP[5]+ OP[0] OP[1] ~OP[2] ~OP[3] ~OP[4] OP[5]

NPCSEL=~OP[0] ~OP[1] OP[2] ~OP[3] ~OP[4] ~OP[5]

EXTOP=OP[0] OP[1] ~OP[2] ~OP[4] OP[5]

ADD = 00; SUBSTRACT = 01; OR = 10

ALUCTR[0]= ~OP[0] ~OP[1] ~OP[2] ~OP[3] ~OP[4] ~OP[5]+ ~OP[0] ~OP[1] OP[2] ~OP[3] ~OP[4] ~OP[5]

ALUCTR[1]= OP[0] ~OP[1] OP[2] OP[3] ~OP[4] ~OP[5]

### (2). 充分利用真值表中的 X 可以将以上控制信号化简为最简单的表达式， 请给出化简后的形式。

只给出了可以化简的2个

REGDST =~OP[0] ~OP[1] ~OP[3] ~OP[4] ~OP[5]

MEMTOREG=OP[0] OP[1] ~OP[2] OP[4] OP[5]

### (3). 事实上，实现nop空指令，我们并不需要将它加入控制信号真值表，为什么？请给出你的理由。

NOP的opcode为000000，它为R型指令，则它只能对RF中目的寄存器进行修改，而它的目的寄存器是$0，无法做到对$0寄存器进行修改，所以不需要将它加入到控制信号真值表当中。

## 3. 测试CPU思考题

### (1). 前文提到，“可能需要手工修改指令码中的数据偏移”，但实际上只需再增加一个 DM片选信号,就可以解决这个问题。请阅读相关资料并设计一个 DM 改造方案使得无需手工修改数据偏移。

DM是用RAM实现的，Logisim中的RAM是按字寻址的，不是按字节，而我们输入到DM中的地址是按字节的，因此我们只需要取输入中的2到6位就行了，而不是0到5位。

### (2). 除了编写程序进行测试外，还有一种验证CPU设计正确性的办法——形式验证。 形式验证的含义是根据某个或某些形式规范或属性，使用数学的方法证明其正确性或非正确性。请搜索“形式验证（Formal Verification)"了解相关内容后，简要阐述相比与测试，形式验证的优劣。

相比于测试，形式验证的优点是它可以做到对所有可能的情况进行验证，而测试只能通过个别数据进行验证。形式验证的可以比较快的发现自己再设计当中的错误。但是缺点是只能发现设计中的错误而不能够发现在实现当中的错误。