单周期处理器设计文档

# 模块规格

## Instruction Fetch Unit（取指令单元）

### 端口定义

表1 IFU的端口定义

|  |  |  |  |
| --- | --- | --- | --- |
| 名称 | 方向 | 端口规格 | 功能描述 |
| sign\_imm32 | input | [31:0] | 传入符号扩展后的offset，以适应beq指令 |
| addr26 | input | [25:0] | 传入j型指令的26位addr，以适应j型指令 |
| ra32 | input | [31:0] | 传入寄存器读出的32位addr，以适应jr指令 |
| PC\_choice | input | [1:0] | 传入PC的四种转移规则选择信号 |
| clk | input | [0:0] | 时钟信号 |
| reset | input | [0:0] | 同步复位信号，驱动PC同步复位成0x3000 |
| funct | output | [5:0] | 当前指令的funct段。（[5:0]段） |
| shamt | output | [4:0] | 当前指令的shamt段。（[10:6]段） |
| rd | output | [4:0] | 当前指令的rd段。（[15:11]段） |
| rt | output | [4:0] | 当前指令的rt段。（[20:16]段） |
| rs | output | [4:0] | 当前指令的rs段。（[25:21]段） |
| opcode | output | [5:0] | 当前指令的opcode段。（[31:26]段） |
| PC+4 | output | [31:0] | 下一条指令的地址，以适应jal指令 |

### 功能说明

IF中由PC计数器、PC转移器、指令存储器等组成。

PC的初始值为0x00003000。当时钟上升沿来临时，将其更新为PC转移器所计算出的下一个PC值。当reset信号有效且上升沿时，将PC计数器同步复位为0x00003000值。PC计数器的32bit输出经过截取后传给指令存储器，成为读取指令的10bit地址。

PC转移器根据当前的PC状态、若干不同种类的立即数分别计算出顺序执行、分支执行、跳转执行、跳转寄存器执行的下一个PC值，并由PC\_choice信号选择其中一路成为PC的下一个状态：

PC\_choice = 0时，对应顺序执行：PC’ = PC + 4；

PC\_choice = 1时，对应beq成立：PC’ = PC + 4 + sign\_imm32\*4；

PC\_choice = 2时，对应j型指令：PC’ = PC[31:28]||addr26||00；

PC\_choice = 3时，对应jr指令： PC’ = ra32。

指令存储器从PC计数器截取10bit指令地址，从1024\*32bit的ROM中读取32位指令并输出。

总体来说，取指令单元由时钟信号和复位信号控制，在每一个上升沿输出待执行的机器码指令。指令的执行顺序由PC计数器控制，PC的次态由当前态、立即数和指令的类型决定。

## General Register File（通用寄存器堆）

### 端口定义

表2 GRF的端口定义

|  |  |  |  |
| --- | --- | --- | --- |
| 名称 | 方向 | 端口规格 | 功能描述 |
| A1 | input | [4:0] | 读取地址为A1 |
| A2 | input | [4:0] | 读取地址为A2 |
| A3 | input | [4:0] | 写入32bit数据的目的地址A3 |
| WD | input | [31:0] | 写入到A3的32bit数据 |
| clk | input | [0:0] | 时钟信号 |
| reset | input | [0:0] | 同步复位信号 |
| WE | input | [0:0] | 写使能信号。为1时可写，为0时忽略WD |
| RD1 | output | [31:0] | 从A1读取的32bit数据 |
| RD2 | output | [31:0] | 从A2读取的32bit数据 |

### 功能说明

GRF由具有写使能的32个寄存器组成，每个寄存器均为32bit，均初始化为0，其中0号寄存器（地址为5bit 00000）始终为常数0，忽略对其的任何修改。

其余寄存器由5位地址编码，从00000到11111。在每时每刻，A1传入的地址对应的寄存器的32bit值输出到RD1中，A2传入的地址对应的寄存器的32bit值输出到RD2中。

在时钟clk处于上升沿且写使能WE=1时，寄存器将当前时刻地址为A3的寄存器中写入32bit数据WD。

当同步复位信号reset=1时，所有寄存器在clk上升沿复位成0值。

总体来说，GRF的两个读取操作（RD1 = reg[A1], RD2 = reg[A2]）在每时每刻都生效，GRF的一个写入操作（reg[A3] = WD）当且仅当clk为上升沿且WE=1时生效。

## Arithmetic Logical Unit（算术逻辑单元）

### 端口定义

表3 ALU的端口定义

|  |  |  |  |
| --- | --- | --- | --- |
| 名称 | 方向 | 端口规格 | 功能描述 |
| A | input | [31:0] | 运算对象1 |
| B | input | [31:0] | 运算对象2 |
| ALUControl | input | [2:0] | ALU运算类型控制标记，支持5种运算 |
| C | output | [31:0] | 由对象1和2得到的运算结果 |
| Zero | output | [0:0] | 结果是否为0：为0则Zero=1，否则Zero=0 |

### 功能说明

该模块为纯组合逻辑，与时钟无关。

本ALU实现暂不检测溢出。

当ALUControl = 010时，C = A + B（简单二进制加法，不考虑溢出）。

当ALUControl = 110时，C = A - B（简单二进制减法，不考虑溢出）。

当ALUControl = 000时，C = A & B（按位与）。

当ALUControl = 001时，C = A | B（按位或）。

当ALUControl = 111时，C = A < B（小于置位，若A小于B，则C置为1；否则C置为0）。

无论ALU进行什么运算操作，在任何时刻Zero = 1当且仅当C = 0；在任何时刻Zero = 0当且仅当C ≠ 0。

## Data Memory（数据内存）

### 端口定义

表4 DM的端口定义

|  |  |  |  |
| --- | --- | --- | --- |
| 名称 | 方向 | 端口规格 | 功能描述 |
| Addr | input | [31:0] | 读取/写入的目标地址，按字节寻址 |
| WD | input | [31:0] | 待写入的32bit数据 |
| clk | input | [0:0] | 时钟信号 |
| reset | input | [0:0] | 同步复位信号，驱动清零内存 |
| WE | input | [0:0] | 写使能信号。为1时可写，为0时忽略WD |
| RD | output | [31:0] | 从地址Addr读取出的32bit数据 |

### 功能说明

DM由1024个字段组成，每个字段均为32bit，均初始化为0。

DM在同一时刻只能有意义地支持读取和写入其中一种操作。

当WE = 0时，截取Addr的[11:2]位得到按字寻址的地址，读取该地址，从RD输出。此为读取。

当WE = 1时，若clk处于上升沿，则更新Addr对应的字段数据为WD中获取的数据。下一时刻clk上升沿过去，RD立即读取Addr的数据。此为写入。

当同步复位信号reset=1时，所有字段在clk上升沿复位成0值，回到初始状态。

总体来说，DM的读取操作（RD1= mem[Addr]）在WE=0时生效，DM的写入操作（mem[Addr] = WD）当且仅当clk为上升沿且WE=1时生效。

## Immediate Calculator（立即数计算器）

### 端口定义

表5 ImmCalc的端口定义

|  |  |  |  |
| --- | --- | --- | --- |
| 名称 | 方向 | 端口规格 | 功能描述 |
| imm16 | input | [15:0] | I指令中的16位立即数 |
| sign\_imm32 | output | [31:0] | 符号扩展后的立即数 |
| zero\_imm32 | output | [31:0] | 零扩展后的立即数 |
| lui\_imm32 | output | [31:0] | 将imm16加载至高16位后的立即数 |

### 功能说明

I指令的16位立即数可能作为内存操作指令lw/sw的带符号偏移量、ori的无符号参数或是lui的参数等。它们分别需要对立即数进行符号扩展、零扩展、加载至高位的操作。

立即数计算器ImmCalc获取一个16bit立即数imm16，并将其符号扩展、零扩展、加载至高位的结果sign\_imm32、zero\_imm32、lui\_imm32分别输出。

## Controller（控制器）

### 端口定义

表6 Controller的端口定义

|  |  |  |  |
| --- | --- | --- | --- |
| 名称 | 方向 | 端口规格 | 功能描述 |
| opcode | input | [5:0] | 32bit指令中的opcode段 |
| funct | input | [5:0] | 32bit指令中的funct段 |
| MemtoReg | output | [0:0] | 为0时，GRF接收ALU运算结果；  为1时，GRF接收DM读取结果。 |
| MemWrite | output | [0:0] | DM的写使能信号 |
| Branch | output | [0:0] | **优先级低**于跳转指令相关信号。非跳转指令时，Branch为0时，PC的变换规则为PC+4；Branch为1时，若beq的两个源寄存器数值相等，则PC的变换规则为PC+4+Imm\*4，否则为PC+4。 |
| ALUControl | output | [2:0] | ALU的运算控制信号 |
| ALUSrc | output | [1:0] | ALU的运算对象2的来源。 |
| RegDst | output | [0:0] | GRF的写入对象。为0时为rt；为1时为rd |
| RegWrite | output | [0:0] | GRF的写使能信号 |
| Jump | output | [0:0] | 当JR=0时，Jump为0时，PC转移由Branch决定；Jump为1时，按照26位立即数的拼接法转移PC。 |
| Link | output | [0:0] | 为0时，无特殊操作；  为1时，将PC+4存储到$ra中。 |
| JR | output | [0:0] | 为0时，无特殊操作；  为1时，令PC’ = ra32。 |

### 功能说明

Controller根据32bit指令的opcode段和funct段，指令特异性地确定各个控制信号。

MemtoReg控制GRF接收的WD（写入数据）是ALU的运算结果C，或是DM的读取结果RD。

MemWrite直接传给DM的WE，控制DM是否写使能。

Branch决定当前命令除跳转指令外，是不是分支指令。当不是跳转指令时：若不是分支指令，则Branch = 0，则PC的运算规则一定为PC’ = PC+4。若是，则Branch = 1，则PC的运算规则可能为PC’ = PC+4，也可能为PC’ = PC+4，取决于beq条件的成立与否。

ALUControl控制ALU的运算类型，直接传给ALU的ALUControl。

ALUSrc控制ALU的第二个运算对象的来源。即ALU的B可能接收GRF的读出，也可能接收立即数的符号扩展、零扩展、加载高位。

RegDst控制GRF写入的目的地址A3。可能向rt寄存器写入，也可能向rd寄存器写入。

RegWrite直接传给GRF的WE，控制GRF是否写使能。

Jump、Link、JR共同决定了j、jal、jr的PC变化和存储特性，为PC的转移增加了两条规则。

总体来说，Controller特异性识别指令，从而输出控制信号，控制数据通路中数据的流动和运算。

具体地，如何根据某条指令的机器码和其意义设计控制信号，见下文“控制器设计”中阐述。

# 数据通路设计

表7 数据通路设计表

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| op | NPC | GRF | | | | ALU | | DM | |
| **Imm** | A1 | A2 | **A3** | **WD** | A | **B** | Addr | WD |
| R | / | rs | rt | rd | ALU.C | GRF.RD1 | GRF.RD2 | / | / |
| ori | / | rs | / | rt | ALU.C | GRF.RD1 | IMM.Zero | / | / |
| lw | / | rs | / | rt | DM.RD | GRF.RD1 | IMM.Sign | ALU.C | / |
| sw | / | rs | rt | / | / | GRF.RD1 | IMM.Sign | ALU.C | GRF.RD2 |
| lui | / | rs | / | rt | ALU.C | GRF.RD1 | IMM.Lui | / | / |
| jal | INS.26 | / | / | $ra | IFU.PCp4 | / | / | / | / |
| j | INS.26 | / | / | / | / | / | / | / | / |
| jr | ALU.C | rs | rt | / | / | GRF.RD1 | GRF.RD2 | / | / |
| beq | IMM.Sign | rs | rt | / | / | GRF.RD1 | GRF.RD2 | / | / |

因此，在GRF.A3、GRF.WD、ALU.B、NPC.Imm等四处应该设置由控制信号选择的多路选择器。

# 控制器设计

## 真值表设计

此处的真值表指的是对于每一个(opcode, funct)的组合，都有一组确定的控制信号的组合。不同指令的不同控制信号组合并起来就成为了一张真值表。

支持指令集{addu, subu, ori, lw, sw, beq, lui, nop, j, jal, jr}的一张可能的真值表如下：

表8 指令-控制信号真值表

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| funct | 100001 | 100011 | / | / | / | / | 000000 |
| opcode | 000000 | 000000 | 001101 | 100011 | 101011 | 001111 | 000000 |
|  | **addu** | **subu** | **ori** | **lw** | **sw** | **lui** | **nop** |
| **MemtoReg** | 0 | 0 | 0 | 1 | x | 0 | 0 |
| **MemWrite** | 0 | 0 | 0 | 0 | 1 | 0 | 0 |
| **Branch** | 0 | 0 | 0 | 0 | 0 | 0 | x |
| **ALUControl** | 010 | 110 | 001 | 010 | 010 | 010 | xxx |
| **ALUSrc** | 00 | 00 | 10 | 01 | 01 | 11 | xx |
| **RegDst** | 1 | 1 | 0 | 0 | x | 0 | x |
| **RegWrite** | 1 | 1 | 1 | 1 | 0 | 1 | x |
| **Jump** | 0  0  0 | | | | | | |
| **Link** |
| **JR** |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| funct | / | / | 001000 | / |
| opcode | 000011 | 000010 | 000000 | 000100 |
|  | **jal** | **j** | **jr** | **beq** |
| **MemtoReg** | 0 | 0 | 0 | x |
| **MemWrite** | 0 | 0 | 0 | 0 |
| **Branch** | 0 | 0 | 0 | 1 |
| **ALUControl** | 000 | 000 | 010 | 110 |
| **ALUSrc** | 00 | 00 | 00 | 00 |
| **RegDst** | 0 | 0 | 0 | x |
| **RegWrite** | 1 | 0 | 0 | 0 |
| **Jump** | 1 | 1 | 0 | 0 |
| **Link** | 1 | 0 | 0 | 0 |
| **JR** | 0 | 0 | 1 | 0 |

## 控制信号设计规则

### MemtoReg

MemtoReg直接表示了是否将内存中数据存入寄存器中。因此当且仅当指令为lw时，才使MemtoReg=1。

为方便简化真值表，当RegWrite=0时，MemtoReg的值不必须为0。而当RegWrite=1时，除lw以外的其他指令的MemtoReg的值必须为0。

### MemWrite

MemWrite直接表示了是否将寄存器中数据存入内存中。因此当且仅当指令为sw时，才使MemWrite=1。

其余时刻，由于DM的WD段直接连接到了寄存器的RD2输出，则MemWrite必须为0。

### Branch

Branch直接表示了是否为分支指令。因此所有可能按照PC’ = PC+4+Imm\*4规则跳转的指令，Branch都应为1。其余指令除nop外，Branch都应该为0。nop对应的Branch值无所谓，因其Imm=0。

### ALUControl

ALUControl对于运算型指令，直接设为其指令意义对应的运算标志。

对于需要计算地址的lw/sw指令，由于有addr+offset的计算需求，故设为“+”运算对应的标志。

对于beq指令，由于有判断(GRF[rs]-GRF[rt])==0的需求，故设为“-”运算对应的标志。

### ALUSrc

ALUSrc表示ALU的运算对象是从寄存器取出（R型指令和beq指令），还是从立即数运算获得（I型指令）。

对于I指令，考察其指令的具体意义，可以判断其立即数需要符号扩展、零扩展还是加载到高位，由此设置ALUSrc。

### RegDst

RegDst表示存入寄存器的地址为rt还是rd。对于R型指令，目的寄存器为rd；对于I型指令，目的寄存器为rt。

为方便简化真值表，当RegWrite=0时，RegDst的值任意。

### RegWrite

RegWrite表示是否GRF写使能。对于跳转指令、sw指令、nop指令等不操作寄存器的指令，RegWrite=0。

### Jump

Jump直接表示了是否为跳转指令。因此所有应该按照PC’ = addr32规则跳转的指令，Jump都应为1。其余指令的Jump均应该为0。

### Link

Link直接决定了GRF参与储存的地址和数据是正常结果，还是$31和PC+4。当且仅当jal指令Link信号为1。

### JR

JR直接决定了PC’的第四种转移规则，故当且仅当jr指令JR信号为1。

## 控制器设计：从机器码到控制信号

对于某一条指令，其意义和控制信号被其opcode（和funct）唯一确定。

故控制器可由指令的识别和信号的生成两部分构成。

### 指令的识别

以lw指令的opcode ”100011”为例，采用以下的与门进行特异性识别指令：

一个lw识别器应当其opcode=100011时输出1，其他时候输出0。因此构造以下结构：

assign lw\_Detector = op[5] & !op[4] & !op[3] & !op[2] & op[1] & op[0];

对于R指令，其识别器应针对其funct码类似构造。

### 信号的生成

由真值表我们知道，lw对应的信号可合并为1000100101。因此可构造如下信号生成器：

assign lw\_Signal = SignExtend(lw\_Detector) & 10’b1000100101;

这样，对于每一个指令就有一个10位Signal信号。它们其中有且只有一个为目的信号，其余的均为10位0。这样，构造一个或门即可得到最终信号：

assign Signal = lw\_Signal | sw\_Signal | beq\_Signal |…;

# 测试CPU

## 测试代码

测试指令集：{addu, subu, ori, lw, sw, beq, lui, nop, j, jal, jr}

测试MIPS代码：

**jal loadimm**

**save**:

sw $t0, 0($s0) # m1 = 7 65536 7 65536

sw $t1, 4($s0) # m2 = 55 62 55 62

sw $t2, 8($s0) # m3 = 62 55 62 55

sw $t3, 12($s0) # m4 = 65536 7 65536 7

lw $t3, 0($s0) # t3 = 7 65536 7 65536

lw $t2, 4($s0) # t2 = 55 62 55 62

lw $t1, 8($s0) # t1 = 62 55 62 55

lw $t0, 12($s0) # t0 = 65536 7 65536 7

subu $t4, $t1, $t2 # t4 = 7 -7 7 -7

beq $t4, $t3, save # back down back down

**j save** # back back

**loadimm**:

ori $t0, $0, 7 # t0 = 7

ori $t1, $t0, 50 # t1 = 55

addu $t2, $t0, $t1 # t2 = 62

nop

lui $t3, 1 # t3 = 65536

subu $t4, $t3, $t2 # t4 = 65474

nop

ori $s0, $0, 4 # s0 = 4

**jr $ra**

## 预期结果

首先，jal直接跳转至loadimm标签后的指令。$t0, $t1, $t2, $t3, $t4, $s0寄存器分别按时间顺序获得值7, 55, 62, 65536, 65474, 4。之后jr直接跳回save标签后的指令。

之后，DM中按字寻址的第1位存储7，第2位存储55，第3位存储62，第4位存储65536。后将7、55、62、65536分别读到$t3, $t2, $t1, $t0中。

之后，$t4 = 62-55 = 7，与$t3相等。代码重复从标签save执行。

第二次执行后，$t4 = 55-62 = -7，与$t3不相等，代码顺序向下执行。

遇到j指令，重新跳回save标签后的指令。完成一个循环周期。

之后代码将无限循环下去，每一个循环周期内lw/sw代码段被执行两次。

# 思考题

## 数据通路设计

### input [11:2] addr

可以直接连线到32位的wire [31:0] addr上，并自动切片成10位信号：

DM(.addr(addr32))

如果使用input [9:0] addr，则还要在传入信号时手动切片：

DM(.addr(addr32[11:2]))

然而，它们都不保留完整的32位信号。

### 同步复位reset

reset针对PC计数器、GRF寄存器堆、DM数据内存进行同步复位。PC计数器设置为0x00003000，GRF和DM全部清零。

可以认为，reset的含义为重新运行程序，此时指令应该从第一条指令开始执行，故PC=0x00003000。此时上次运行的全部数据影响都应该消除，故GRF和DM恢复初始状态到全零。

## 控制器设计

### 使用条件语句

always@(\*) begin

case(opcode)

6'b100011: begin // lw

{MtoR, MWrite, Br, ALUCtrl, ALUSrc, RDst, RWrite } =

{1'b1, 1'b0, 1'b0, 3'b010, 2'b01, 1'b0, 1'b1};

end

endcase

end

优点：拓展指令和拓展信号方便。无需改变Controller电路。

缺点：没什么缺点。

### 使用assign语句仿照Logisim的与或门阵列

assign lw\_Detector = op[5] & !op[4] & !op[3] & !op[2] & op[1] & op[0];

output ALUSrc;

assign ALUSrc = ori\_Detector | lw\_Detector | sw\_Detector | … ;

优点：纯组合逻辑，在信号的更新时间上不易出错。

缺点：不够直观，与真值表对应关系弱，不好debug。

### 使用assign语句传入控制信号拼接的常数

assign lw\_Detector = op[5] & !op[4] & !op[3] & !op[2] & op[1] & op[0];

output [9:0] signals;

assign signals = (10’b10001001010 & SignExt10(lw\_Detector)) | (10’b01001001000 & SignExt10(sw\_Detector)) | …;

优点：与真值表对应关系强，纯组合逻辑。

缺点：扩展信号位数时较为麻烦。

## 综合

### 忽略溢出时的add(i) 和add(i)u

add和addu：

设被加数为，其中均为最高一位。

忽略溢出和最高位的CarryOut时，add的结果为为33位数据。当存入目的寄存器时，只保留了低32位。和的进位均因为截取被忽略，得到的结果即为忽略进位的结果，与addu相同。

addi和addiu：

设被加数为。其中为最高一位。

所以将符号扩展到33位，相当于先符号扩展至32位，再令。其中为的最高一位。

忽略溢出和最高位的CarryOut时，addi的结果为为33位数据。当存入目的寄存器时，只保留了低32位。和的进位均因为截取被忽略，得到的结果即为忽略进位的结果。

而addiu也是先符号扩展至32位。故与addi结果相同。

### 单周期处理器的优缺点

优点：实现简单

缺点：每条指令都跑满了整个数据通路和计算单元，但是可能大部分信息是没有用的，被RegWrite和MemWrite忽略掉的。然而无论如何一条指令都要等待一个时钟周期才能完成，而一个时钟周期内必须完成全部数据通路，而不是只完成所需要的部分通路。因此计算效率不高，相当于将所有指令都视为像lw一样最费时间的指令。这样并不符合common case fast原则。

### jal, jr与栈的关系

jal和jr必须搭配栈使用。jal将返回地址存在$ra中，jr读取$ra中的地址。然而在递归等使用场景中，$ra只有一个，但是层层调用的返回地址却不止一个。由于在一次call function和return中，只用到一个返回地址关系，所以可以将该层及其以上所有层的ra记录到栈中，再将$ra设为新的地址。

由于函数调用和返回的路径特性和栈的push和pop一致，所以使用栈来存储$ra的历史值。