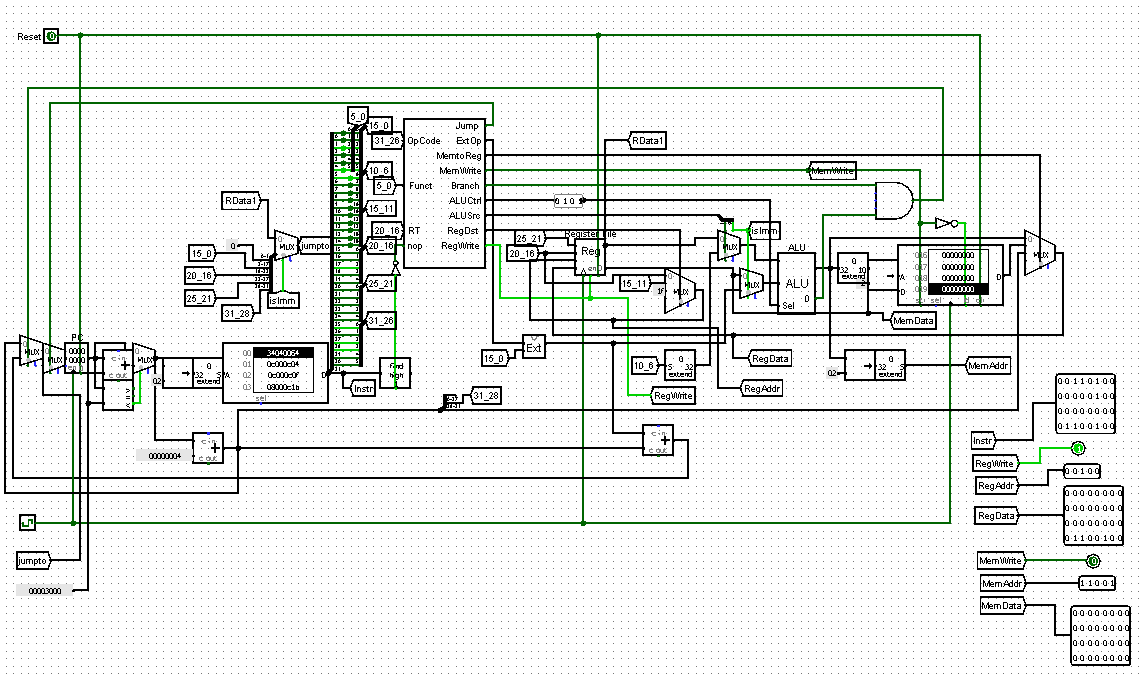
Verilog开发MIPS单周期处理器

一、整体结构：

控制器（Controller）、IFU（取指令单元）、GRF（通用寄存器组，也称为寄存器文件、寄存器堆）、ALU（算术逻辑单元）、DM（数据存储器）、EXT（位扩展器）。

处理器为32位处理器

处理器应支持的指令集为：{addu,subu,ori,lw,sw,beq,lui,jal,jr,nop}。



1. 数据通路设计：

（一）模块规格撰写

1. IFU（取指令单元）。包括PC（程序计数器，5位），IM（指令存储器，32位\*32字）

为了和Mars汇编器保持一致，PC需要初始化为0x30000000，因此，采取如下方法：

当reset信号为1时初始化为0，加上0x3000作为IM的地址。之后，如果PC值大于等于0x3000则直接作为地址，否则加上0x3000。可以证明，这个有限状态机中的值永远不小于0x3000，但保证地址不重复加0x3000。

1）PC（程序计数器）

器件：32bit寄存器

模块接口

|  |  |  |
| --- | --- | --- |
| 信号名 | 方向 | 描述 |
| next[31:0] | I | 下一条指令的地址 |
| clk | I | 时钟信号 |
| reset | I | 复位信号  1：有效  0：无效 |
| reg[31:0] | O | 当前指令地址 (IAddr=32'h00003000) |

功能定义

|  |  |  |
| --- | --- | --- |
| 序号 | 功能名称 | 功能描述 |
| 1 | 复位 | 当复位信号有效时，PC被设置为起始地址0x00003000 |

2）IM（指令存储器）

IM容量为4KB（32bit×1024字）

因为ROM中储存了1024个地址，且IM实际地址宽度仅为10位，，从而将地址的低10位（2~11位）连接到ROM选择地址端口。

模块接口

|  |  |  |
| --- | --- | --- |
| 信号名 | 方向 | 描述 |
| RAddr[9:0] | I | 输入当前的地址 |
| RData[31:0] | O | 输出读取当前的数据 |

2.Controller（控制器）

模块接口

|  |  |  |
| --- | --- | --- |
| 信号名 | 方向 | 描述 |
| cmd[31:0] | I | 32位操作编码 |
| Jump | O | 跳转信号  0为不是跳转指令  1为是跳转指令 |
| RegSrc[1:0] | O | 寄存器地址的来源 |
| MemWrite | O | DM写控制信号，写入GRF的数据选择(内存写使能信号) |
| Branch | O | 分支信号  输出0为不是Branch  输出为1是Branch |
| ALUSrc[1:0] | O | ALU操作数2的来源 |
| RegDst[1:0] | O | 寄存器地址选择  0:[20:16]  1:[15:11] |
| RegWrite | O | 寄存器写使能信号 |
| ExtOp[1:0] | O | 控制扩展方式 |
| ALUCtrl[3:0] | O | ALU功能选择信号 |

3.GRF（通用寄存器组）：内部包括32个寄存器

具有写使能的寄存器实现，寄存器总数为 32 个**0 号寄存器**的值始终保持为 0。其他寄存器初始值均为 0

模块接口

|  |  |  |
| --- | --- | --- |
| 信号名 | 方向 | 描述 |
| IAddr[31:0] | I | 相应指令存储地址 |
| clk | I | 时钟信号 |
| reset | I | 复位信号  1：有效  0：无效 |
| WEnable | I | 读写控制信号  1：写操作  0：读操作 |
| RAddr1[4:0] | I | 读寄存器1的地址 |
| RAddr2[4:0] | I | 读寄存器2的地址 |
| WAddr[4:0] | I | 5为地址输入信号，指定32个寄存器中的一个作为写入目标寄存器地址 |
| WData[31:0] | I | 向写寄存器中写入的值（数据） |
| RData1[31:0] | O | 32位输出1 |
| RData2[31:0] | O | 32位输出2 |

功能定义

|  |  |  |
| --- | --- | --- |
| 序号 | 功能名称 | 功能描述 |
| 1 | 复位 | 当复位信号有效时，所有寄存器的值被设置为0x00000000 |
| 2 | 写寄存器 | 根据输入的写寄存器地址，把输入的数据写入写寄存器中 |
| 3 | 读寄存器 | 根据输入的读寄存器地址，将数据读出 |

4.ALU（算术逻辑单元）

提供32位加、减、或运算可以不支持溢出

模块接口

|  |  |  |
| --- | --- | --- |
| 信号名 | 方向 | 描述 |
| op1[31:0] | I | ALU32位输入数据A |
| op2[31:0] | I | ALU32位输入数据B |
| sel[3:0] | I | ALU功能选择信号 |
| Result[31:0] | O | 32位数据输出 |
| Zero | O | 输出为0 |

5.EXT（位扩展器）：

模块接口

|  |  |  |
| --- | --- | --- |
| 信号名 | 方向 | 描述 |
| imm[15:0] | I | 16位imm数据输入 |
| EOp[1:0] | I | 位扩展选择信号  00: 高位符号扩展  01：高位补0  10：低位补0  11：符号扩展之后，左移两位 |
| ext[31:0] | O | 位扩展后的32位输出 |

模块接口（extbyte）

|  |  |  |
| --- | --- | --- |
| 信号名 | 方向 | 描述 |
| imm[7:0] | I | 8位imm数据输入 |
| EOp | I | 位扩展选择信号  0：高位补0  1：低位补0 |
| ext[31:0] | O | 位扩展后的32位输出 |

6.DM（数据存储器）

 DM（数据存储器，32位\*1024字）。用内置RAM实现，采用Separate load and store ports属性。地址10位。题目中要求输出5位地址，取后5位。RAM自带时钟信号、写使能、地址端和数据端，与DM要求完全相同。

起始地址：0x00003000

模块接口

|  |  |  |
| --- | --- | --- |
| 信号名 | 方向 | 描述 |
| clk | I | 时钟信号 |
| WE | I | 读写控制信号  1：写操作  0：读操作 |
| reset | I | 复位信号  1：有效  0：无效 |
| isu | I | 判断是否无符号或有符号数 |
| MemDst[1:0] | I | 写入数据的输入 |
| Addr[11:0] | I | 读寄存器的地址 |
| WData[31:0] | I | 向写寄存器中写入的值（数据） |
| IAddr[31:0] | I | 相应指令存储地址 |
| RData[31:0] | O | 32位输出 |

功能定义

|  |  |  |
| --- | --- | --- |
| 序号 | 功能名称 | 功能描述 |
| 1 | 复位 | 当复位信号有效时，所有数据被设置为0x00000000 |
| 2 | 写操作 | 根据输入的寄存器地址，把输入的数据写入 |
| 3 | 读操作 | 根据输入的寄存器地址，将其中的数据读出 |

1. 思考题
2. 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？



答：在DM中，内存是按字划分的。一个字是四字节，因此在进行DM中进行内存的存取时，应当将输入地址除以4（或者右移两位），作为实际地址，即取2~11位。

addr信号来自ALU的运算结果。

1. 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？

答：reset针对PC、GRF、DM，其中PC需要回到初始指令地址0x00003000，GRF需要清空全部寄存器的值，DM也清空。

（三）IM设计

module Instr\_Memory(

input [9:0]RAddr,

output [31:0]RData

);

reg [31:0] rom[0:1023];

integer i;

initial begin

for(i=0;i<1024;i=i+1) rom[i]=0;

$readmemh("code.txt",rom);

end

assign RData=rom[RAddr];

endmodule

（四）控制器设计

（一）设计方式

1.数据通路设计

PC：输入端接next信号，输出端接指令存储器。

IM：输出端接指令译码器。

DM：写数据端接寄存器文件输出，读数据端接寄存器文件写数据。写地址端接ALU结果。MemWrite接写使能端。

ALU：操作数接寄存器和立即数，输出端接寄存器写数据和内存写地址。还有一个零端，与Branch信号接在与门上，当二者均为真时分支有效。

next信号（下一个PC值）：当j为真时，为立即数。否则当branch和zero均为真时，为branch计算所得地址；

2.主控单元真值表

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 指令 | Opcode | Funct | Jump | ExtOp | MemtoReg | MemWrite | Branch | ALUCtrl | ALUSrc | RegDst | RegWrite |
| addu | 000000 | 100001 | 0 | xx | 0 | 0 | 0 | 0010 | 0 | 1 | 1 |
| subu | 000000 | 100011 | 0 | xx | 0 | 0 | 0 | 0011 | 0 | 1 | 1 |
| ori | 001101 | xxxxxx | 0 | 01 | 0 | 0 | 0 | 0101 | 1 | 0 | 1 |
| lw | 100011 | 0 | 00 | 1 | 0 | 0 | 0010 | 1 | 0 | 1 |
| sw | 101011 | 0 | 00 | 0 | 1 | 0 | 0010 | 1 | x | 0 |
| beq | 000100 | 0 | 11 | 0 | 0 | 1 | 0011 | 0 | x | 0 |
| lui | 001111 | 0 | 10 | 0 | 0 | 0 | 0010 | 0 | 0 | 1 |
| j | 000010 | 1 | 00 | x | 0 | x | xxxx | x | x | 0 |

1. 思考题
2. 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。

答：

1. 利用if-else（或case）完成操作码和控制信号的值之间的对应；

always@(\*)begin

case(op)

6'b000000:

begin

if(func==6'b001000)begin //jr

RegDst=2'b00;

ALUSrc=0;

MemtoReg=2'b00;

RegWrite=0;

MemWrite=0;

PCSrc=2'b00;

Extop=2'b00;

ALUop=2'b00;

jr=1;

beq=0;

end

else if(func==6'b100001)begin //addu

RegDst=2'b01;

ALUSrc=0;

MemtoReg=2'b00;

RegWrite=1;

MemWrite=0;

PCSrc=2'b00;

Extop=2'b00;

ALUop=2'b00;

jr=0;

beq=0;

end

else if(func==6'b100011)begin //subu

RegDst=2'b01;

ALUSrc=0;

MemtoReg=2'b00;

RegWrite=1;

MemWrite=0;

PCSrc=2'b00;

Extop=2'b00;

ALUop=2'b01;

jr=0;

beq=0;

end

else begin

RegDst=2'b00;

ALUSrc=0;

MemtoReg=2'b00;

RegWrite=0;

MemWrite=0;

PCSrc=2'b00;

Extop=2'b00;

ALUop=2'b00;

jr=0;

beq=0;

end

end

1. 利用assign语句完成操作码和控制信号的值之间的对应；

assign RegDst[0] = ( (op == 6'b000000 && func == 6'b100001) || (op == 6'b000000 && func == 6'b100011) )? 1 : 0 ;

1. 利用宏定义

`define state1 4'b0001

`define state2 4'b0010

`define state3 4'b0100

`define state4 4'b1000

`define 标识符(宏名) 字符串(宏内容)

如：`define signal string

它的作用是指定用标识符signal来代替string这个字符串，在编译预处理时，把程序中在该命令以后所有的signal都替换成string。这种方法使用户能以一个简单的名字代替一个长的字符串，也可以用一个有含义的名字来代替没有含义的数字和符号，因此把这个标识符(名字)称为“宏名”，在编译预处理时将宏名替换成字符串的过程称为“宏展开”。`define是宏定义命令。

1. 根据你所列举的编码方式，说明他们的优缺点。

|  |  |  |
| --- | --- | --- |
| 编码方式 | 优点 | 缺点 |
| if-else/case | 代码写起来比较容易、直观，与C语言类似 | 它们在不注意的情况下容易产生锁存器，对毛刺敏感，使其处于不确定状态，且语句较为冗长 |
| assign | 语句较为简单，看起来比较方便 | 适用范围比较窄，只能对wire型变量赋值 |
| 宏定义 | 增加可读性、可修改性好、设计的可重用性好 | 定义时容易出错，且不容易查错 |

（五）在线测试相关信息

（一）测试代码

.data

arr:.space 44

.text

ori $t0,$0,0

ori $t1,$0,1

ori $s0,$0,1

ori $s1,$0,4

ori $s2,$0,40

for:

beq $t0,$s2,next

sw $t1,arr($t0)

addu $t1,$t1,$s0

addu $t0,$t0,$s1

j for

next:

ori $t0,$0,0

ori $t1,$0,0

ori $s0,$0,0

for1:

beq $t0,$s2,next1

lw $t1,arr($t0)

addu $s0,$s0,$t1

addu $t0,$t0,$s1

j for1

next1:

sw $s0,arr($t0)

li $t0,0x44897253

导出的机器码为

34080000 34090001 34100001 34110004

34120028 11120004 ad090000 01304821

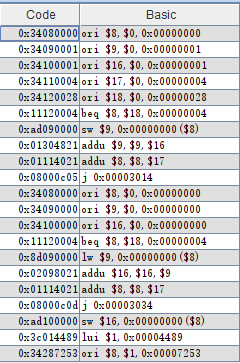
01114021 08000c05 34080000 34090000

34100000 11120004 8d090000 02098021

01114021 08000c0d ad100000 3c014489

34287253

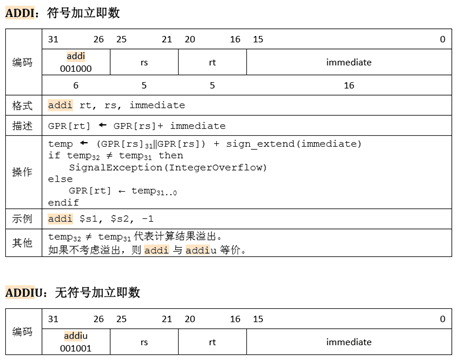
（二）预期结果

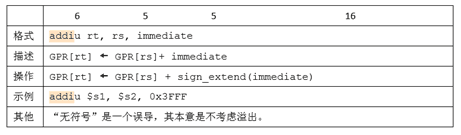


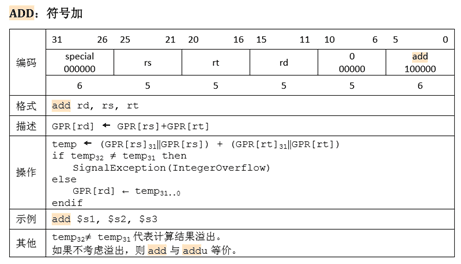
1. 思考题

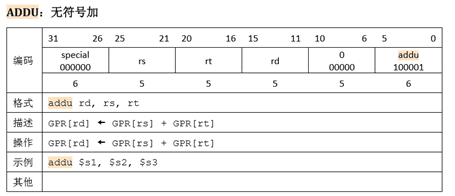
1.C语言是一种弱类型程序设计语言。C语言中不对计算结果溢出进行处理，这意味着C语言要求程序员必须很清楚计算结果是否会导致溢出。因此，如果仅仅支持C语言，MIPS指令的所有计算指令均可以忽略溢出。 请说明为什么在忽略溢出的前提下，addi与addiu是等价的，add与addu是等价的。提示：阅读《MIPS32® Architecture For Programmers Volume II: The MIPS32® Instruction Set》中相关指令的Operation部分 。

答：addi和addiu的区别只是addi在发生溢出时会报错，add与addu也是一样，在发生溢出时add会报错。所以在不考虑溢出的情况下，addi和add忽略报错，因此等价。









2.根据自己的设计说明单周期处理器的优缺点。

答：

|  |  |
| --- | --- |
| 优点 | 缺点 |
| 控制部件相比多周期CPU更简单，且实际设计起来较为简便，不容易出错 | 用一个时钟周期执行一条指令，从而确定时钟周期的时间长度要考虑执行的时间最长的指令，一次确定CPU频率，不管指令复杂度如何，单周期CPU花费相同时间执行，这造成时间上浪费 |

简要说明jal、jr和堆栈的关系

答：在跳转到指定地址实现子程序调用的同时，需要将返回地址保存到ra寄存器，即通常所说的“函数调用的现场保护”，以便子程序返回时能够继续调用之前的流程。对于跳转/分支指令，MIPS CPU将自动保存ra；若子程序需要嵌套调用其他子程序，则必须先存储ra，通常是压入栈，子程序末尾弹出之前保存的ra，然后jr到ra。这两条指令分别实现了直接和间接子程序调用。