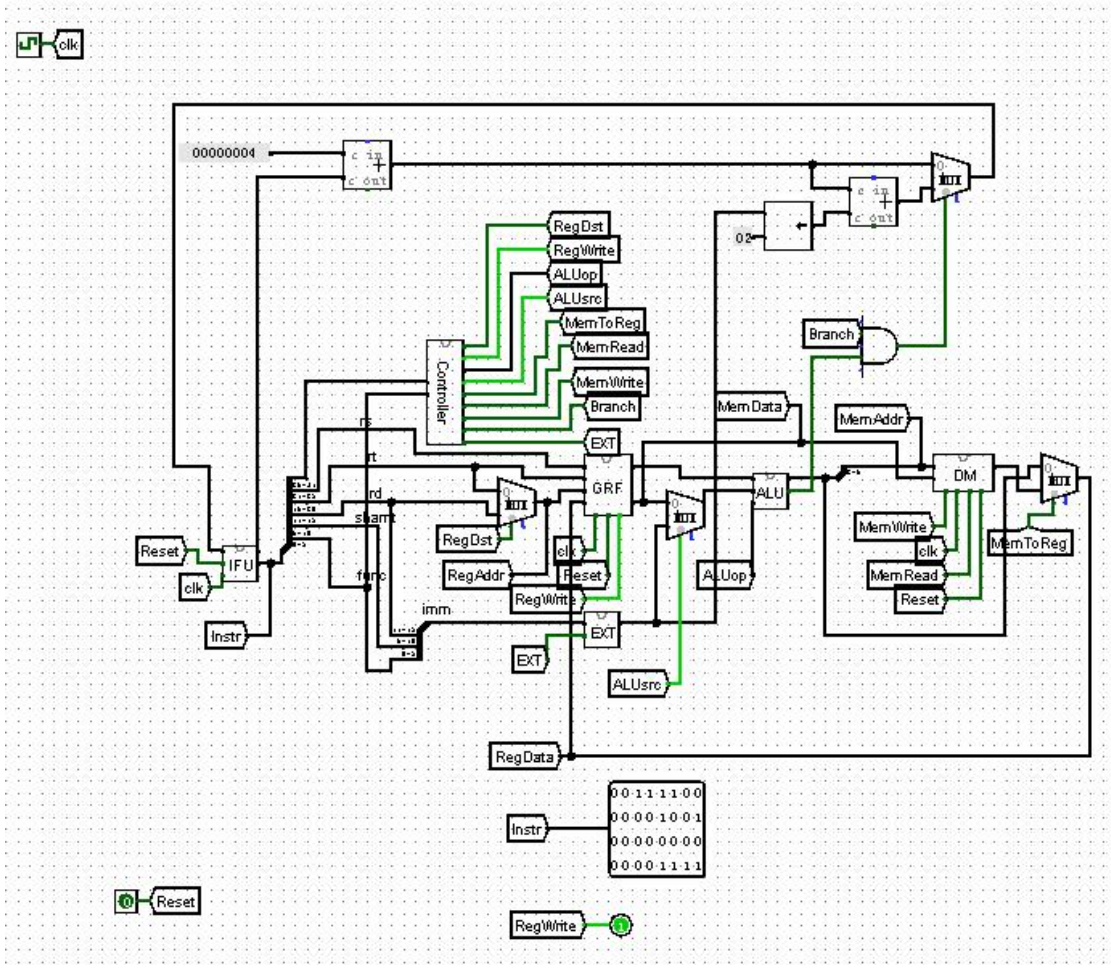
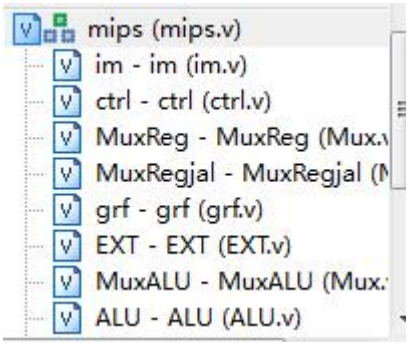
CPU 设计文档

一、总体设计





```
module mips(
    input clk,
    input reset
    );
wire [31:0]PC,PC_n,addr,WriteData,RegData1,RegData2,ALUData2,ALU_result,MemData,WriteData1,PC4,imm1,imm2,PC_B,PC_1,PC_j,PC_2;
wire [5:0]op,func;
wire [9:0]MemAddr;
wire [4:0]rs,rt,rd,shamt,wr1,wr2;
wire RegDst,RegWrite,ALUsrc,MemToReg,MemRead,MemWrite,Branch,EXTop,jump,jrop,zero,Branch1;
wire [1:0]ALUOp;
wire [15:0]imm;
im im(.clk(clk),.Reset(reset),.PC(PC),.PC_n(PC_n),.addr(addr));
assign {op,rs,rt,rd,shamt,func}=addr;
ctrl ctrl(.op(op),.func(func),.RegDst(RegDst),.RegWrite(RegWrite),.ALUop(ALUOp),.ALUsrc(ALUsrc),.MemToReg(MemToReg),
.MemWrite(MemWrite),.MemRead(MemRead),.Branch(Branch),.EXTop(EXTop),.jump(jump),.jrop(jrop));
MuxReg MuxReg(.a0(rt),.a1(rd),.sel(RegDst),.result(wr1));
MuxRegjal MuxRegjal(.a0(wr1),.sel(jump),.result(wr2));
grf grf(.WPC(PC),.ReadReg1(rs),.ReadReg2(rt),.WriteReg(wr2),.Writedata(WriteData),.clk(clk),.Reset(reset),.we(RegWrite),.Readdata1(RegData1)
assign imm={rd,shamt,func};
EXT EXT(.IMM(imm),.EXTop(EXTop),.IMM_result(imm1));
MuxALU MuxALU(.a0(RegData2),.a1(imm1),.sel(ALUsrc),.result(ALUData2));
ALU ALU(.ALUData1(RegData1),.ALUData2(ALUData2),.ALUOp(ALUOp),.ALU_result(ALU_result),.zero(zero));
assign MemAddr=ALU_result[11:2];
DM DM(.PC(PC),.MemAddr(MemAddr),.MemData(RegData2),.clk(clk),.Reset(reset),.MemWrite(MemWrite),.MemRead(MemRead),.MemData_o(MemData));
MuxMTR MuxMTR(.a0(ALU_result),.a1(MemData),.sel(MemToReg),.result(WriteData1));
Muxjr Muxjr(.a0(WriteData1),.a1(PC4),.sel(jump),.result(WriteData));
assign PC4=PC+4;
```

```
assign PC4=PC+4;
assign imm2=imm1<<2;
assign PC_B=PC4+imm2;
assign Branch1=Branch&zero;
MuxBranch MuxBranch(.a0(PC4),.a1(PC_B),.sel(Branch1),.result(PC_1));
assign PC_j={PC[31:28],rs,rt,rd,shamt,func,{2{1'b0}}};
Muxjump Muxjump(.a0(PC_1),.a1(PC_j),.sel(jump),.result(PC_2));
MuxRegPC MuxRegPC(.a0(PC_2),.a1(RegData1),.sel(jrop),.result(PC_n));
endmodule
```

二、数据通路设计

1、im

```
module im(
    input [31:0] PC_n,
    input clk,
    input Reset,
    output [31:0] PC,
    output [31:0] addr
    );
reg [31:0] im_reg[0:1023];
reg [31:0] PC1;
wire [9:0] b;
wire [31:0] a;
initial begin
   $readmemh("code.txt",im_reg);
   PC1=32'h00003000;
end
assign a=PC1-32'h00003000;
assign b=a[11:2];
assign PC=PC1;
assign addr=im_reg[b];
always @(posedge clk) begin
   if(Reset)
      PC1<=32'h00003000;
   else
      PC1<=PC_n;
end
endmodule
```

模块接口

信号名	方向	功能描述
Reset	I	复位信号,当复位信号为 1 时，PC 被设置为 0x00003000
[31:0]PC_n	I	读入下一周期 PC 的值
clk	I	时钟信号
[31:0]PC	O	当前 PC 的值
[31:0]addr	O	当前 32 位指令值

2、grf

```
module grf(
    input [4:0] ReadReg1,
    input [4:0] ReadReg2,
    input [4:0] WriteReg,
    input [31:0] Writedata,
    input [31:0]WPC,
    input clk,
    input Reset,
    input we,
    output [31:0] Readdata1,
    output [31:0] Readdata2
    );
reg [31:0] r[0:31];
assign Readdata1=r[ReadReg1];
assign Readdata2=r[ReadReg2];
integer i;
initial begin
   for(i=0;i<=31;i=i+1)
         r[i]=0;
end
always @(posedge clk) begin
   if(Reset)
      for(i=0;i<=31;i=i+1)
         r[i]<=0;
   if(Reset==0&&we==1) begin
      if(WriteReg!=0)
         r[WriteReg]<=Writedata;
      $display("@%h: $%d <= %h", WPC, WriteReg,Writedata);
   end

end
endmodule
```

模块接口

信号名	方向	功能描述
Reset	I	复位信号，当复位信号为 1 时，所有寄存器被设置为 0x00000000
we	I	寄存器使能信号，当使能信号为 1 时，寄存器可以被写入数据
clk	I	时钟信号
[4:0]ReadReg1	I	寄存器 1 的地址
[4:0]ReadReg2	I	寄存器 2 的地址
[4:0]WriteReg	I	需要写入数据的寄存器的地址
[31:0]Wirtedata	I	写入的数据

[31:0]WPC	I	当前 PC 的值
[31:0]Readdata1	O	寄存器 1 中的数据
[31:0]Readdata2	O	寄存器 2 中的数据

3、ALU

```
module ALU(
    input [31:0] ALUData1,
    input [31:0] ALUData2,
    input [1:0] ALUOp,
    output [31:0] ALU_result,
    output zero
    );
assign ALU_result=(ALUOp==0)?ALUData1+ALUData2:
                  (ALUOp==1)?ALUData1-ALUData2:
                  (ALUOp==2)?ALUData1|ALUData2:
                  (ALUOp==3)?ALUData2<<16:0;
assign zero=(ALUData1==ALUData2)?1:0;
endmodule
```

模块接口

信号名	方向	功能描述
[31:0]ALUData1	I	ALU 数据 1
[31:0]ALUData2	I	ALU 数据 2
[1:0]ALUOp	I	ALU 控制信号 00:加法运算　　01:减法运算（Data1-Data2） 10:或运算　　11:左移 16 位
[31:0]ALU_result	O	输出 ALU 计算结果
zero	O	判断 ALUData1 与 ALUData2 是否相等 若相等，则输出 1；否则，输出 0

4、DM

```
//////////////////////////////////////////////////////////////////
module DM(
    input [9:0] MemAddr,
    input [31:0] MemData,
    input [31:0] PC,
    input MemWrite,
    input clk,
    input MemRead,
    input Reset,
    output [31:0] MemData_o
    );
reg [31:0] dm_reg[0:1023];
wire [31:0]a;
integer i;
assign MemData_o=(MemRead==1)?dm_reg[MemAddr]:0;
assign a={{21'b0},MemAddr};
initial begin
   for(i=0;i<1024;i=i+1)
      dm_reg[i]=0;
end
always @(posedge clk) begin
   if(Reset)
      for(i=0;i<1024;i=i+1)
         dm_reg[i]<=0;
   if(Reset==0&&MemWrite==1) begin
         dm_reg[MemAddr]<=MemData;
         $display("@%h: *%h <= %h",PC, a<<2,MemData);
   end
end
endmodule
```

模块接口

信号名	方向	功能描述
Reset	I	复位信号,当复位信号为 1 时，所有寄存器被设置为 0x00000000
clk	I	时钟信号
MemWrite	I	内存寄存器写使能信号，当该信号为 1 时，可将数据写入内存寄存器中
MemRead	I	内存寄存器读使能信号，当该信号为 1 时，可输出内存寄存器中的数据
[9:0]MemAddr	I	内存寄存器的地址值
[31:0]MemData	I	需要写入内存寄存器的数据
[31:0]PC	I	当前 PC 值
[31:0]MemData_o	O	输出内存寄存器中的数据

5、Controller

```
module ctrl(
    input [5:0] op,
    input [5:0] func,
    output RegDst,
    output RegWrite,
    output [1:0] ALUop,
    output ALUsrc,
    output MemToReg,
    output MemRead,
    output MemWrite,
    output Branch,
    output EXTop,
    output jump,
    output jrop
    );
wire addu,subu,lw,sw,beq,ori,lui,jal,jr;
assign addu=(op==6'b000000&&func==6'b100001)?1:0;
assign subu=(op==6'b000000&&func==6'b100011)?1:0;
assign lw=(op==6'b100011)?1:0;
assign sw=(op==6'b101011)?1:0;
assign beq=(op==6'b000100)?1:0;
assign ori=(op==6'b001101)?1:0;
assign lui=(op==6'b001111)?1:0;
assign jal=(op==6'b000011)?1:0;
assign jr=(op==6'b000000&&func==6'b001000)?1:0;
```

```
assign RegDst=addu|subu;
assign RegWrite=addu|subu|lui|ori|lw|jal;
assign ALUop[1]=ori|lui;
assign ALUop[0]=subu|beq|lui;
assign ALUsrc=lw|sw|ori|lui;
assign MemToReg=lw;
assign MemRead=lw;
assign MemWrite=sw;
assign Branch=beq;
assign EXTop=lw|sw|beq;
assign jump=jal;
assign jrop=jr;
endmodule
```

模块接口

信号名	方向	功能描述
[5:0]op	I	6 为 op 信号
[5:0]func	I	6 为 func 信号
RegDst	O	写寄存器地址控制信号
RegWrite	O	GRF 写使能信号
ALUOp[1:0]	O	ALU 运算控制信号
ALUsrc	O	ALU 第二操作数选择控制信号
MemToReg	O	写入寄存器的数据选择信号
MemRead	O	RAM 读使能信号
MemWrite	O	RAM 写使能信号
Branch	O	分支指令控制信号

EXTop	O	EXT 扩展方式控制信号
jump	O	jal 跳转控制指令控制信号
jrop	O	jr 跳转指令控制信号

6、EXT

```
module EXT(
    input [15:0] IMM,
    input EXTop,
    output [31:0] IMM_result
    );
assign IMM_result=EXTop?{{16{IMM[15]}},IMM}:{{16{1'b0}},IMM};
endmodule
```

模块接口

信号名	方向	功能描述
[15:0]IMM	I	16 为待扩展数据
EXTop	I	扩展方式控制信号 0:无符号扩展　　1:有符号扩展
[31:0]IMM_result	O	扩展后数据

7、Mux

MuxReg

```
module MuxReg(
   input [4:0]a0,
   input [4:0]a1,
   input sel,
   output [4:0]result
    );
assign result=sel?a1:a0;
endmodule
```

模块接口

信号名	方向	功能描述
[4:0]a0	I	多选器 0 接口(rs)
[4:0]a1	I	多选器 1 接口(rt)
sel	I	选择信号(RegDst)
[4:0]result	O	输出选择结果

MuxRegjal

```
module MuxRegjal(
   input [4:0]a0,
   input sel,
   output [4:0]result
    );
assign result=sel?5'b11111:a0;
endmodule
```

模块接口

信号名	方向	功能描述

[4:0]a0	I	多选器 0 接口(MuxReg 输出)
sel	I	选择信号(jump)
[4:0]result	O	选择结果

MuxALU

```
module MuxALU(
    input [31:0]a0,
    input [31:0]a1,
    input sel,
    output [31:0]result
     );
assign result=sel?a1:a0;
endmodule
```

模块接口

信号名	方向	功能描述
[31:0]a0	I	多选器 0 接口(Readdata2)
[31:0]a1	I	多选器 1 接口(EXT_result)
sel	I	选择信号(ALUsrc)
[31:0]result	O	输出选择结果

MuxMTR

```
module MuxMTR(
    input [31:0]a0,
    input [31:0]a1,
    input sel,
    output [31:0]result
     );
assign result=sel?a1:a0;
endmodule
```

模块接口

信号名	方向	功能描述
[31:0]a0	I	多选器 0 接口(ALU_result)
[31:0]a1	I	多选器 1 接口(MemData_o)
sel	I	选择信号(MemToReg)
[31:0]result	O	输出选择结果

MuxRegPC

```
module MuxRegPC(
    input [31:0]a0,
    input [31:0]a1,
    input sel,
    output [31:0]result
     );
assign result=sel?a1:a0;
endmodule
```

模块接口

信号名	方向	功能描述

[31:0]a0	I	多选器 0 接口(MuxMTR 输出)
[31:0]a1	I	多选器 1 接口(PC+4)
sel	I	选择信号(jump)
[31:0]result	O	输出选择结果

MuxBranch

```
module MuxBranch(
    input [31:0]a0,
    input [31:0]a1,
    input sel,
    output [31:0]result
     );
assign result=sel?a1:a0;
endmodule
```

模块接口

信号名	方向	功能描述
[31:0]a0	I	多选器 0 接口(PC+4)
[31:0]a1	I	多选器 1 接口(Branch 跳转范围)
sel	I	选择信号(Branch&zero)
[31:0]result	O	输出选择结果

Muxjump

```
module Muxjump(
    input [31:0]a0,
    input [31:0]a1,
    input sel,
    output [31:0]result
     );
assign result=sel?a1:a0;
endmodule
```

模块接口

信号名	方向	功能描述
[31:0]a0	I	多选器 0 接口(MuxBranch 输出)
[31:0]a1	I	多选器 1 接口(jal 跳转范围)
sel	I	选择信号(jump)
[31:0]result	O	输出选择结果

Muxjr

```
module Muxjr(
    input [31:0]a0,
    input [31:0]a1,
    input sel,
    output [31:0]result
     );
assign result=sel?a1:a0;
endmodule
```

模块接口

信号名	方向	功能描述

[31:0]a0	I	多选器 0 接口(Muxjump 输出)
[31:0]a1	I	多选器 1 接口(rs)
sel	I	选择信号(jrop)
[31:0]result	O	输出选择结果

三、控制器设计

func[5:0]	100001	100011	N/A						001000	000000
op[5:0]	000000	000000	100011	101011	000100	001101	001111	000011	000000	000000
	addu	subu	lw	sw	beq	ori	lui	jal	jr	nop
RegDst	1	1	0	X	X	0	0	X	X	X
RegWrite	1	1	1	0	0	1	1	0	0	X
ALUOp[1:0]	00	01	00	00	01	10	11	XX	XX	XX
ALUsrc	0	0	1	1	0	1	1	X	X	X
MemToReg	0	0	1	X	X	0	0	X	X	X
MemRead	X	X	1	X	X	X	X	X	X	X
MemWrite	0	0	0	1	0	0	0	0	0	X
Branch	0	0	0	0	1	0	0	0	0	X
EXTop	X	X	1	1	1	0	0	X	X	X
jal	0	0	0	0	0	0	0	1	0	X
jrop	0	0	0	0	0	0	0	0	1	X

四、测试 CPU

测试程序：ori $t1,1

ori $t2,1

ori $t5,4

lui $t3,3

loop1:

subu $t1,$t2,$t1

beq $t1,$0,loop1

nop

```
beq $t1,$t2,fun1
nop
fun1:
jal fun
nop
fun:
sw $t1,0($t4)
lw $t7,0($t4)
addu $t4,$t4,$t5
jr $ra
nop
```

期望输出：

$9=00000001,$10=00000001,$13=00000004,$11=00030000, $9=00000000 ，

$9=00000001，$31=00003028 该程序为死循环，所以将把所有内存值赋为

00000001，$12 从 0 开始每次加 4，$15 始终为 1，1ns 仿真结果如下

```
@00003000: $ 9 <= 00000001
@00003004: $10 <= 00000001
@00003008: $13 <= 00000004
@0000300c: $11 <= 00030000
@00003010: $ 9 <= 00000000
@00003010: $ 9 <= 00000001
@00003024: $31 <= 00003028
@0000302c: *00000000 <= 00000001
@00003030: $15 <= 00000001
@00003034: $12 <= 0000001c
@0000302c: *0000001c <= 00000001
@00003030: $15 <= 00000001
@00003034: $12 <= 00000020
```

五、思考题

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？

文件	模块接口定义

dm.v	dm(clk,reset,MemWrite,addr,din,dout);
	input clk; //clock
	input reset; //reset
	input MemWrite; //memory write enable
	input [11:2] addr; //memory's address for write
	input [31:0] din; //write data
	output [31:0] dout; //read data

因为 mips 中内存地址按字节存储，在访问内存中的字时地址一定是 4 的倍数，而设计的 CPU 内存按字存储，所以取[11:2]相当于除以 4。该信号由 ALU 模块输出。

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？

对 PC,GRF 和 DM 进行复位操作。因为 PC 复位需要回到第一条指令的位置（0x00003000），GRF 和 DM 因为会储存数据，所以再复位后需要把所有值清零。

3、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。

case 完成操作码和编码的对应：

```
always @(*) begin
    case(op)
        6’b000000: case(func)
                        6’b100011:addu<=1;
                endcase
        6’b100011: lw<=1;
    endcase
end
```

三目运算符：

```
assign addu=(op==6'b000000&&func==6'b100001)?1:0;
assign sw=(op==6'b101011)?1:0;
```

宏定义：

```
`define sw=(op==6'b101011)?1:0
```

4、根据你所列举的编码方式，说明他们的优缺点。

case:优点：直观，便于理解和加指令。

缺点：必须写在 always 中，而 logisim 中的控制器是组合逻辑，所以可能会出现一些问题。

三目运算法：优点：利用组合逻辑实现，与预期一致。

缺点：对于 ALUop 这种多位信号需要每一位都判断，比较繁琐。

宏定义：优点：组合逻辑实现，且便于理解。

缺点：同样是多位信号的控制比较繁琐，且宏定义有许多用法，这只是我所想到的一种用法。

5、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 。

根据指令集对于这两个指令的描述，add 和 addi 首先将两个源操作数进行符号扩展成 33 位，然后进行加法，然后判断加法结果的高两位是否相等，如果不相等，表明计算结果溢出，如果不相等，表明不溢出，将结果的低 32 位作为结果存入目的寄存器。而 addu 和 addiu 指令则是直接进行 32 位的加法，并将结果存入目的寄存器。

如果忽略溢出，add 和 addi 的溢出分支不进行，则它与 addu，addiu 的处理方式完全一致，所以 add 与 addu 是等价的，addi 与 addiu 是等价的。

6、根据自己的设计说明单周期处理器的优缺点。

优点：每个周期处理一条指令，不会像多周期和流水线在顶层多许多寄存器，使顶层硬件设计比较简单。

缺点：所有指令时间周期一样，为最长的那一条指令（lw），所以 cpu 效率会比较低。

7、简要说明 jal、jr 和堆栈的关系。

jal 和 jr 一般配套使用，jal 会把 PC+4 的值存入$ra 中，然后跳转，而 jr $ra 则跳回到 jal 的下一条指令。这其实就是一个堆栈的过程，当有多个 jal 时，$ra 总会存最后一个的 PC+4 值，最后才会跳转到第一个 PC+4，符合堆栈的先进后出。