P4 Verilog 单周期 CPU 设计文档

1. 模块与层次结构

本次 CPU 设计以 Logisim 单周期 CPU（32 位）设计为基础，采用系统的层次化、结构化的设计，整体结构如下。可支持的指令集：**{addu, subu, ori, lw, sw, beq, lui, jal,jr,nop}**。

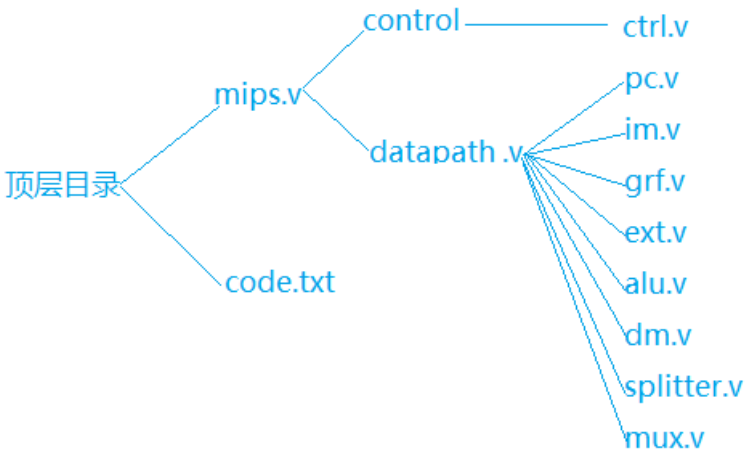

图 1 CPU 模块和层次

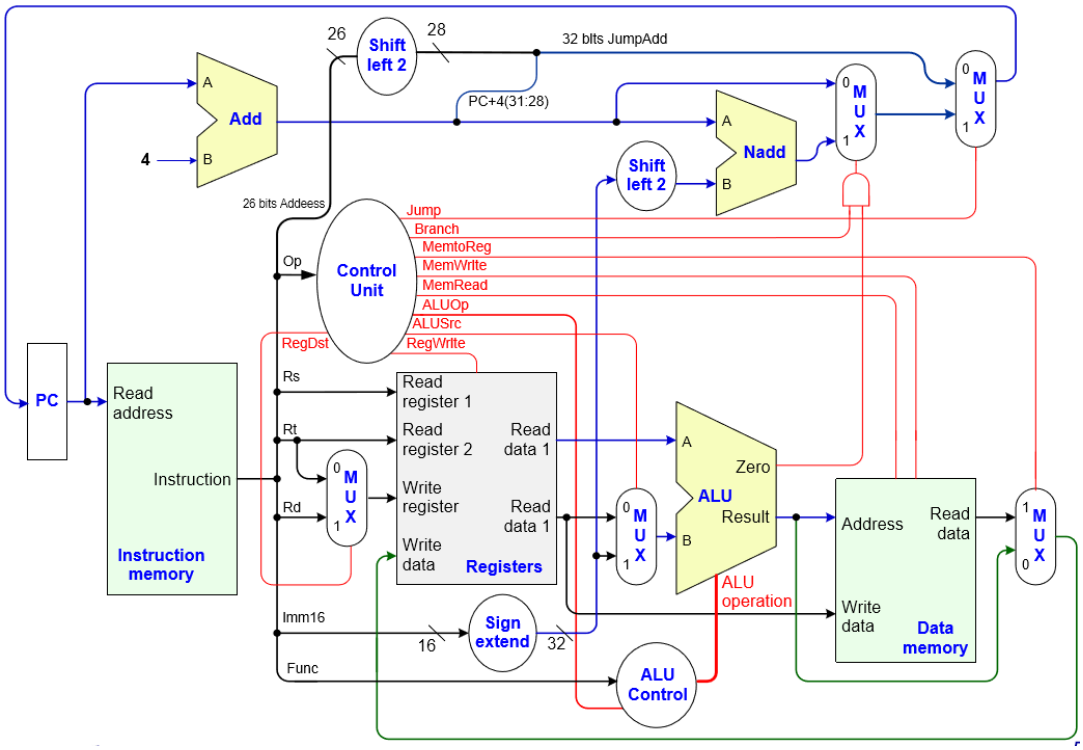

图 2 CPU 数据通路电路图

顶层文件 mips.v 模块接口定义：

表 1 mips.v 模块接口定义

文件	模块接口定义
mips.v	module mips(clk, reset); input clk,reset;//clk, reset

2. 数据通路设计

由于 P3 已经完成相关设计，由此得出的数据通路图进行设计。数据通路管理 8 个模块：pc.v, im.v, grf.v, alu.v, ext.v, splitter.v, dm.v。先完成这些部件，然后交给 datapath 统一管理，自底向上设计。

pc.v 模块接口定义：

表 2 pc.v 模块接口定义

文件	模块接口定义

pc.v	module pc(clk, reset, npc, PC);
	input clk,reset;//clk, reset
	input [31:0]npc;//next pc address
	output [31:0]PC;//current pc address

im.v 模块接口定义：

表 3 im.v 模块接口定义

文件	模块接口定义

im.v	module im(PC,Instr);
	input [31:0]PC;//current pc address
	output [31:0]Instr;//binary instructions

grf.v 模块接口定义：

表 4 grf.v 模块接口定义

文件	模块接口定义

grf.v	module grf(RA1,RA2,WA,WD,clk,reset,RegWrite,RD1,RD2);
	input [4:0]RA1,RA2,WA;//read address 1, read address 2, write address
	input clk,reset,RegWrite;//clk, reset, GRF write enable
	input [31:0]WD;//data to write
	output [31:0]RD1,RD2;//the data of read address 1, the data of read address 2

alu.v 模块接口定义：

表 5 alu.v 模块接口定义

文件	模块接口定义

alu.v	module alu(ALUctr,A,B,C,equal);
	input [1:0]ALUctr;//alu 运算选择信号
	input [31:0]A,B;//alu 运算的输入数 A，B，00：符号加法运算 01：符号减法运算 10：按位与运算
	output [31:0]C;//alu 运算结果的输出
	output equal;//equal=A==B?1:0

ext.v 模块接口定义：

表 6 ext.v 模块接口定义

文件	模块接口定义

ext.v	module ext(imme,ExtOp,datao);
	input [15:0]imme;//16 位立即数的输入
	input [1:0]ExtOp;//扩展选择信号,00:零扩展 01：符号扩展 10：低 16 位加载至高 16 位
	output [31:0]datao;//扩展结果的输出

splitter.v 模块接口定义：

表 7 splitter.v 模块接口定义

文件	模块接口定义

splitter.v	module
	splitter(Instr,op,func,rs,rt,rd,re,imme,index);
	input [31:0]Instr;//32 指令的输入
	output [25:0]index;//26 位地址的输出
	output [15:0]imme;//16 位立即数的输出
	output [5:0]op, func;//6 位 option 和 function 的输出
	output [4:0]rs,rt,rd,re;//寄存器编号的输出

dm.v 模块接口定义：

表 8 dm.v 模块接口定义

文件	模块接口定义

dm.v	module dm(addr,datai,clk,reset,MemWrite,datao);
	input [9:0]addr;//10 位地址的输入
	input [31:0]datai;//要存入 datamemory32 位数据的输入
	input clk,reset;//clk, reset
	input MemWrite;//写使能信号
	output [31:0]datao;//32 位数据的输出

datapath.v 模块接口定义：

表 9 datapath.v 模块接口定义

文件	模块接口定义

datapath.v	module datapath(clk,reset,RegDst,ALUSrc,MemtoReg,RegWrite,MemWrite,Branch,Jump,ExtOp,ALUctr,op,func);
	input clk,reset;//clk, reset
	input ALUSrc,RegWrite,MemWrite,Branch;//1 位控制信号
	input [1:0]ALUctr, ExtOp,MemtoReg,RegDst,Jump;//2 位控制信号
	output [5:0]op, func;//6 位 option 和 function

3. 控制器设计

控制的本质就是一个译码的过程，将指令包含的信息转为 CPU 各部分的控制信号，在 Verilog 中有多种实现方式，这里依旧采用与或门阵来实现。

表 10 Controller 信号说明

序号	信号名称	功能描述
1	Instr	32 位指令的输入
2	RegDst	选择寄存器堆(GRF)的写地址 00:rt 01:rd 10:31
3	ALUSrc	选择输入 ALU 的立即数 0:RD2 1:立即数
4	Memtoreg	选择从 DM 读取写入 GRF 的数据 00:aluout 01:dmout 10:GRF[rs]
5	RegWrite	GRF 写使能信号
6	MemWrite	DM 写使能信号
7	Branch	Beq 指令的表征信号
8	Jump	跳转选择信号 00:in1 01:26 位立即数抵制 10:GRF[rs]
9	ExtOp	扩展立即数选择信号 00:零扩展 01:符号扩展 10:加载至高位
10	ALUctr	ALU 运算选择信号 00:符号加法 01:符号减法 10:按位与

表 11 ctrl.v 模块端口定义

文件	模块接口定义

ctrl.v	module ctrl(op,func,RegDst,ALUSrc,MemtoReg,RegWrite,MemWrite,Branch,Jump,ExtOp,ALUctr);
	input [5:0]op,func;//6 位 option 和 function
	output ALUSrc,RegWrite,MemWrite,Branch;//1 位控制信号
	output [1:0]ExtOp,ALUctr,MemtoReg,RegDst,Jump;//2 位控制信号

表 12 Controller 真值表

op	000000	000000	001101	100011	101011	000100	001111	000011	000000
func	100001	100011	N/A						001000
	addu	subu	ori	lw	sw	beq	lui	jal	jr
RegDst	01	01	00	00	xx	xx	00	10	xx
ALUSrc	0	0	1	1	1	0	1	x	0
MemtoReg	00	00	00	01	00	00	00	10	00
RegWrite	1	1	1	1	0	0	1	1	0
MemWrite	0	0	0	0	1	0	0	0	0
Branch	0	0	0	0	0	1	0	0	0
Jump	00	00	00	00	00	00	00	01	10
ExtOp	xx	xx	00	01	01	01	10	xx	xx
ALUctr	00	01	10	00	00	00	10	xx	00

4. CPU 测试

测试代码：

```
ori $3, $0, 12
ori $1, $0, 1
ori $2, $0, 1
ori $7, $0, 1

 ori $4, $0, 0
 work:
 sw $1, 0($0)
 sw $2, 4($0)
 addu $1, $1, $2
 addu $2, $1, $2
 addu $4, $4, $7
 beq $4, $3, end
 jal work
 end:
```

测试期望：

```
@00003000: $ 3 <= 0000000c
@00003004: $ 1 <= 00000001
@00003008: $ 2 <= 00000001
@0000300c: $ 7 <= 00000001
@00003010: $ 4 <= 00000000
@00003014: *00000000 <= 00000001
@00003018: *00000004 <= 00000001
@0000301c: $ 1 <= 00000002
@00003020: $ 2 <= 00000003
@00003024: $ 4 <= 00000001
@0000302c: $31 <= 00003030
@00003014: *00000000 <= 00000002
@00003018: *00000004 <= 00000003
@0000301c: $ 1 <= 00000005
@00003020: $ 2 <= 00000008
@00003024: $ 4 <= 00000002
@0000302c: $31 <= 00003030
@00003014: *00000000 <= 00000005
@00003018: *00000004 <= 00000008
@0000301c: $ 1 <= 0000000d
@00003020: $ 2 <= 00000015
@00003024: $ 4 <= 00000003
@0000302c: $31 <= 00003030
@00003014: *00000000 <= 0000000d
@00003018: *00000004 <= 00000015
@0000301c: $ 1 <= 00000022
@00003020: $ 2 <= 00000037
@00003024: $ 4 <= 00000004
@0000302c: $31 <= 00003030
@00003014: *00000000 <= 00000022
@00003018: *00000004 <= 00000037
@0000301c: $ 1 <= 00000059
@00003020: $ 2 <= 00000090
@00003024: $ 4 <= 00000005
@0000302c: $31 <= 00003030
@00003014: *00000000 <= 00000059
@00003018: *00000004 <= 00000090
@0000301c: $ 1 <= 000000e9
@00003020: $ 2 <= 00000179
@00003024: $ 4 <= 00000006
@0000302c: $31 <= 00003030
@00003014: *00000000 <= 000000e9
@00003018: *00000004 <= 00000179
```

以上代码是将前 12 个斐波那契数列写入内存，符合期望。

5. 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？

答： MIPS 在 DM 中以字为单位存储数据，因此一个数据和下一个数据的地址偏差是 4，addr 的后两位无效，采用[11:2]位数。这个 addr 是 alu 的运算结果。

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？

答：reset 主要对 pc、grf、dm 进行清零复位操作。首先 pc 要复位到 0x0000_3000，寄存器堆和数据存储器要清零是因为这些存储器的值会影响 reset 后代码的运算。

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。

答：第一种：与或门式

```
module ctrl(op,func,RegDst,ALUSrc,MemtoReg,RegWrite,MemWrite,Branch,Jump,ExtOp,ALUctr);
   input [5:0]op;
   input [5:0]func;
   output ALUSrc,RegWrite,MemWrite,Branch;
   output [1:0]ExtOp,ALUctr,MemtoReg,RegDst,Jump;
   wire addu,subu,ori,lw,sw,beq,lui,jal,jr;
   assign addu=!op[5]&!op[4]&!op[3]&!op[2]&!op[1]&!op[0]&func[5]&!func[4]&!func[3]&!func[2]&!func[1]&func[0];
   assign subu=!op[5]&!op[4]&!op[3]&!op[2]&!op[1]&!op[0]&func[5]&!func[4]&!func[3]&!func[2]&func[1]&func[0];
   assign ori=!op[5]&!op[4]&op[3]&op[2]&!op[1]&op[0];
   assign lw=op[5]&!op[4]&!op[3]&!op[2]&op[1]&op[0];
   assign sw=op[5]&!op[4]&op[3]&!op[2]&op[1]&op[0];
   assign beq=!op[5]&!op[4]&!op[3]&op[2]&!op[1]&!op[0];
   assign lui=!op[5]&!op[4]&op[3]&op[2]&op[1]&op[0];
   assign jal=!op[5]&!op[4]&!op[3]&!op[2]&op[1]&op[0];
   assign jr=!op[5]&!op[4]&!op[3]&!op[2]&!op[1]&!op[0]&!func[5]&!func[4]&func[3]&!func[2]&!func[1]&!func[0];

   assign RegDst[0]=addu|subu;
   assign RegDst[1]=jal;
   assign ALUSrc=ori|lw|sw|lui;
   assign MemtoReg[0]=lw;
   assign MemtoReg[1]=jal;
   assign RegWrite=addu|subu|ori|lw|lui|jal;
   assign MemWrite=sw;
   assign Branch=beq;
   assign Jump[0]=jal;
   assign Jump[1]=jr;
   assign ExtOp[0]=lw|sw|beq;
   assign ExtOp[1]=lui;
   assign ALUctr[0]=subu;
   assign ALUctr[1]=ori|lui;
endmodule
```

第二种：case 式

```
module ctrl(op,func,RegDst,ALUSrc,MemtoReg,RegWrite,MemWrite,Branch,Jump,ExtOp,ALUctr);
   input [5:0]op;
   input [5:0]func;
   output ALUSrc,RegWrite,MemWrite,Branch;
   output [1:0]ExtOp,ALUctr,MemtoReg,RegDst,Jump;
   always @* begin
      case(op)
      6'b000000:begin
         if(func==6'b100001)begin
            ALUSrc=0;RegWrite=1;MemWrite=0;Branch=0;
            RegDst=2'b01;MemtoReg=2'b00;Jump=2'b00;ExtOp=2'b00;ALUctr=2'b00;
         end
         else if(func==6'b100011)begin
            ALUSrc=0;RegWrite=1;MemWrite=0;Branch=0;
            RegDst=2'b01;MemtoReg=2'b00;Jump=2'b00;ExtOp=2'b00;ALUctr=2'b01;
         end
         else if(func==6'b001000)begin
            ALUSrc=0;RegWrite=0;MemWrite=0;Branch=0;
            RegDst=2'b00;MemtoReg=2'b00;Jump=2'b10;ExtOp=2'b00;ALUctr=2'b00;
         end
         else begin
            ALUSrc=0;RegWrite=0;MemWrite=0;Branch=0;
            RegDst=2'b00;MemtoReg=2'b00;Jump=2'b00;ExtOp=2'b00;ALUctr=2'b00;
         end
      end

      6'b001101:begin
         ALUSrc=1;RegWrite=1;MemWrite=0;Branch=0;
         RegDst=2'b00;MemtoReg=2'b00;Jump=2'b00;ExtOp=2'b00;ALUctr=2'b10;
      end
      6'b100011:begin
         ALUSrc=1;RegWrite=1;MemWrite=0;Branch=0;
         RegDst=2'b00;MemtoReg=2'b01;Jump=2'b00;ExtOp=2'b01;ALUctr=2'b00;
      end
      6'b101011:begin
         ALUSrc=1;RegWrite=0;MemWrite=1;Branch=0;
         RegDst=2'b00;MemtoReg=2'b00;Jump=2'b00;ExtOp=2'b01;ALUctr=2'b00;
      end
      6'b000100:begin
         ALUSrc=0;RegWrite=0;MemWrite=0;Branch=1;
         RegDst=2'b00;MemtoReg=2'b00;Jump=2'b00;ExtOp=2'b01;ALUctr=2'b00;
      end
      6'b1001111:begin
         ALUSrc=1;RegWrite=1;MemWrite=0;Branch=0;
         RegDst=2'b00;MemtoReg=2'b00;Jump=2'b00;ExtOp=2'b10;ALUctr=2'b10;
      end
      6'b000011:begin
         ALUSrc=0;RegWrite=1;MemWrite=0;Branch=0;
         RegDst=2'b10;MemtoReg=2'b10;Jump=2'b01;ExtOp=2'b00;ALUctr=2'b00;
      end
      default:begin
         ALUSrc=0;RegWrite=0;MemWrite=0;Branch=0;
         RegDst=2'b00;MemtoReg=2'b00;Jump=2'b00;ExtOp=2'b00;ALUctr=2'b00;
      end
      endcase
   end
endmodule
```

第三种：if_else 式

```
module ctrl(op,func,RegDst,ALUSrc,MemtoReg,RegWrite,MemWrite,Branch,Jump,ExtOp,ALUctr);
   input [5:0]op;
   input [5:0]func;
   output reg ALUSrc,RegWrite,MemWrite,Branch;
   output reg[1:0]ExtOp,ALUctr,MemtoReg,RegDst,Jump;
   always @* begin
      if(op==6'b000000)begin
         if(func==6'b100001)begin
            ALUSrc=0;RegWrite=1;MemWrite=0;Branch=0;
            RegDst=2'b01;MemtoReg=2'b00;Jump=2'b00;ExtOp=2'b00;ALUctr=2'b00;
         end
         else if(func==6'b100011)begin
            ALUSrc=0;RegWrite=1;MemWrite=0;Branch=0;
            RegDst=2'b01;MemtoReg=2'b00;Jump=2'b00;ExtOp=2'b00;ALUctr=2'b01;
         end
         else if(func==6'b001000)begin
            ALUSrc=0;RegWrite=0;MemWrite=0;Branch=0;
            RegDst=2'b00;MemtoReg=2'b00;Jump=2'b10;ExtOp=2'b00;ALUctr=2'b00;
         end
         else begin
            ALUSrc=0;RegWrite=0;MemWrite=0;Branch=0;
            RegDst=2'b00;MemtoReg=2'b00;Jump=2'b00;ExtOp=2'b00;ALUctr=2'b00;
         end
      end
      else if(op==6'b001101)begin
         ALUSrc=1;RegWrite=1;MemWrite=0;Branch=0;
         RegDst=2'b00;MemtoReg=2'b00;Jump=2'b00;ExtOp=2'b00;ALUctr=2'b10;
      end
      else if(op==6'b100011)begin
         ALUSrc=1;RegWrite=1;MemWrite=0;Branch=0;
         RegDst=2'b00;MemtoReg=2'b01;Jump=2'b00;ExtOp=2'b01;ALUctr=2'b00;
      end

      else if(op==6'b101011)begin
         ALUSrc=1;RegWrite=0;MemWrite=1;Branch=0;
         RegDst=2'b00;MemtoReg=2'b00;Jump=2'b00;ExtOp=2'b01;ALUctr=2'b00;
      end
      else if(op==6'b000100)begin
         ALUSrc=0;RegWrite=0;MemWrite=0;Branch=1;
         RegDst=2'b00;MemtoReg=2'b00;Jump=2'b00;ExtOp=2'b01;ALUctr=2'b00;
      end
      else if(op==6'b1001111)begin
         ALUSrc=1;RegWrite=1;MemWrite=0;Branch=0;
         RegDst=2'b00;MemtoReg=2'b00;Jump=2'b00;ExtOp=2'b10;ALUctr=2'b10;
      end
      else if(op==6'b000011)begin
         ALUSrc=0;RegWrite=1;MemWrite=0;Branch=0;
         RegDst=2'b10;MemtoReg=2'b10;Jump=2'b01;ExtOp=2'b00;ALUctr=2'b00;
      end
      else begin
         ALUSrc=0;RegWrite=0;MemWrite=0;Branch=0;
         RegDst=2'b00;MemtoReg=2'b00;Jump=2'b00;ExtOp=2'b00;ALUctr=2'b00;
      end
   end
endmodule
```

4. 根据你所列举的编码方式，说明他们的优缺点。

答：第一种采用与或门阵的形式，书写比较复杂，但是适用于多指令设计；第二种为 case 式，思路清晰，但是只适用于支持指令较少的控制器设计；第三种是 if_else 式，设计清晰，同样不适合较多指令的 CPU 设计，易出错。

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 。

答： 在指令集查阅得知：

格式	addi rt, rs, immediate
描述	GPR[rt] ← GPR[rs]+ immediate

操作	temp ← (GPR[rs]$_{31}$‖GPR[rs]) + sign_extend(immediate)
	if temp$_{32}$ ≠ temp$_{31}$ then
	SignalException(IntegerOverflow)
	else
	GPR[rt] ← temp$_{31..0}$
	endif

示例	addi $s1, $s2, -1
其他	temp$_{32}$ ≠ temp$_{31}$ 代表计算结果溢出。 如果不考虑溢出，则 addi 与 addiu 等价。

不考虑溢出，即 temp32=temp31，都执行相同的操作 GPF[rt]=temp[31:0]=GPF[rs] + sign_extend(immediate)，所以此时 addi 与 addiu 等价，同理 add 与 addu 等价。

6. 根据自己的设计说明单周期处理器的优缺点。

答： 优点：一条指令一个周期使得上下条指令涉及的数据不会产生纠缠，准确性较高；缺点：一条指令一个周期时间占用较长，一些模块没有得到充分利用，使得整个 CPU 性能难以提高。

7. 简要说明 jal、jr 和堆栈的关系。

答：jal 是调用函数跳转，同时将下一条指令的地址写入$ra 寄存器，jr $ra 是函数返回跳转到$ra 的地址所指向的指令，在函数调用前，当前的某些参数要压入栈中，以防止在调用的函数中被错误修改，函数调用完成后，这些参数出栈继续使用。一般来说，函数调用和返回指令配合堆栈使用，完成一次函数的调用和返回。