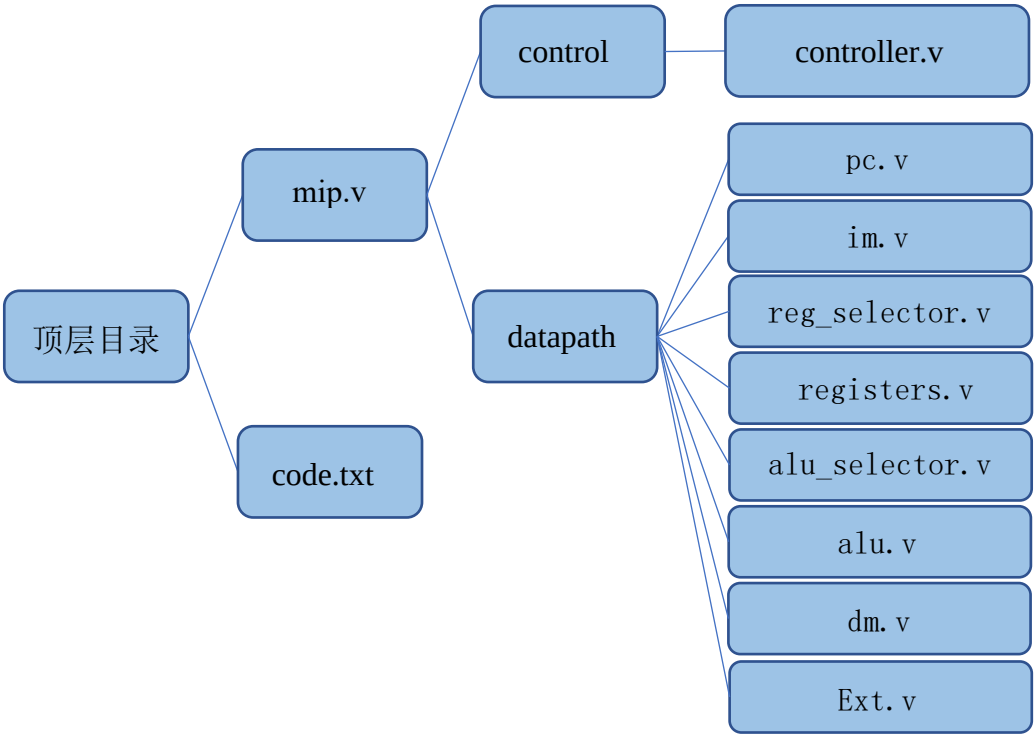
Logisim 开发 MIPS 单周期处理器

一、总体设计

单周期 CPU 顶层设计图

文件	模块接口定义

mips.v	module mips(clk,reset);
	input clk; //clock
	input reset; //reset



Ins	Adder		PC	LM Add	Registers				ALU		DM		Signext	Nadder		Shift<<2
	A	B			Reg1	Reg2	Wreg	Wdata	A	B	Add.	Wdata				
addu	PC	4	Adder	PC	Rs	Rt	Rd	ALU	Rdata1	Rdata2						
subu	PC	4	Adder	PC	Rs	Rt	Rd	ALU	Rdata1	Rdata2						
ori	PC	4	Adder	PC	Rs		Rt	ALU	Rdata1	Signext			imm16			
lw	PC	4	Adder	PC	Rs		Rt	DM	Rdata1	Signext	ALU		imm16			
sw	PC	4	Adder	PC	Rs	Rt			Rdata1	Signext	ALU	Rdata2	imm16			
beq	PC	4	Adder/Nadder	PC	Rs	Rt			Rdata1	Rdata2			imm16	Adder	Shift	imm16
lui	PC	4	Adder	PC			Rt	ALU		Signext			imm16			
jal	PC	4	Signext				31	Adder					Shift			instr_index
jr			Rdata1		31											

二、模块定义

1、控制器

（1）Controller

①基本描述

控制器将每一条机器指令中包含的信息，转化为给 CPU 各部分的控制信号。把解码逻辑分解为和逻辑和或逻辑两部分：和逻辑的功能是识别，将输入的机器码识别为相应的指令；或逻辑的功能是生成，根据输入的指令的不同，产生不同的控制信号。

文件	模块接口定义
controller.v	module controller(input [5:0] Op, input [5:0] funct, output RegA, output ALUSrc, output RegWrite, output MemWrite, output [1:0] RegDst, output [1:0] Mem2Reg, output [1:0] nPC_Sel, output [1:0] ExtOp, output [1:0] ALUOp);

②编码方式

利用 assign 语句完成操作码和控制信号的值之间的对应。

③模块接口

表 1 IFU 模块接口

信号名	方向	功能描述
Op[5:0]	I	操作码
funct[5:0]	I	指令的 function 类型
RegA	O	寄存器组的第一个操作寄存器是否为 31 号寄存器
ALUSrc	O	ALU 第二个操作数控制信号
RegWrite	O	寄存器组写控制信号
MemWrite	O	存储器写控制信号

RegDst[1:0]	O	寄存器组写地址控制信号
		//观察到表中 Op===6'b000000 的指令 RegDst[0]=1 均成立，
		//所以直接用
		//assign RegDst[0] = (Op === 6'b000000)?1:0;
		//赋值

Mem2Reg[1:0]	O	寄存器组写入数据控制信号
nPC_Sel[1:0]	O	指令判断信号
ExtOp[1:0]	O	扩展类型选择信号
ALUOp[1:0]	O	ALU 操作选择信号

2、数据通路

（1）PC（程序计数器）

①基本描述

用于存放下一条指令所在单元的地址。

文件	模块接口定义
pc.v	module pc(input clk, input reset, input zero, input [1:0] nPC_Sel, input [25:0] instr_index, input [15:0] immediate, input [31:0] rdata1, output [31:0] address, //address of updata pc output [31:0] adder, //address of pc + 4 output reg [31:0] P //address of pc before);

②模块接口

表 1 IFU 模块接口

信号名	方向	功能描述
clk	I	时钟信号
reset	I	复位信号
zero	I	ALU 的操作结果是否为 0
nPC_Sel[1:0]	I	指令类型判断信号
instr_index[25:0]	I	指令的后 26 位编码
immediate[15:0]	I	指令后 16 位编码
rdata1[31:0]	I	寄存器组第一个输出数据
address[31:0]	O	pc 更新后的结果
adder[31:0]	O	之前 pc 加 4 的结果
P[31:0]	O	之前的 pc 值

（2）IM（指令存储器）

①基本描述

用于存储带操作指令。

文件	模块接口定义
im.v	module im(input [31:0] address, output [5:0] op, output [4:0] rs, output [4:0] rt,

	output [4:0] rd, output [15:0] immediate, output [25:0] instr_index, output [5:0] funct);

②模块接口

表 1 IFU 模块接口

信号名	方向	功能描述
address[31:0]	I	待执行指令地址
op[5:0]	O	指令操作码
rs[4:0]	O	指令 rs
rt[4:0]	O	指令 rt
rd[4:0]	O	指令 rd
immediate[15:0]	O	指令后 16 位编码
instr_index[25:0]	O	指令后 26 位编码
funct[5:0]	O	指令后 6 位编码

（3）GRF MUX（寄存器组信号选择模块）

①基本描述

对多组传向寄存器组的数据进行选择。

文件	模块接口定义
reg_selector.v	module reg_selector(input [4:0] rs, input [4:0] rt, input [4:0] rd, input [31:0] result, input [31:0] RData, input [31:0] adder, input RegA, input [1:0] RegDst, input [1:0] Mem2Reg, output [4:0] A1, output [4:0] A2, output [4:0] A3, output [31:0] wdata);

②模块接口

表 1 IFU 模块接口

信号名	方向	功能描述
rs	I	指令编码的 rs 部分
rt	I	指令编码的 rt 部分

rd	I	指令编码的 rd 部分
result	I	ALU 的计算结果
RData	I	数据存储器的输出数据
adder	I	前一个 pc 加 4
RegA	I	第一个数据是否输入 31 号寄存器的地址
RegDst	I	寄存器组写入地址的选择信号
Mem2Reg	I	寄存器组写入数据的选择信号
A1	O	寄存器组的第一个操作寄存器的地址
A2	O	寄存器组的第二个操作寄存器的地址
A3	O	寄存器组写入数据的地址
wdata	O	寄存器组写入数据

（4）GRF（寄存器组）

①基本描述

用于传送和暂数据，也参与算术逻辑运算，并保存运算结果。

文件	模块接口定义
registers.v	module registers(input [4:0] A1, input [4:0] A2, input [4:0] A3, input [31:0] wdata, input [31:0] address, input clk, input reset, input RegWrite, output [31:0] rdata1, output [31:0] rdata2);

②模块接口

表 1 IFU 模块接口

信号名	方向	功能描述
A1	I	寄存器组第一个操作寄存器地址
A2	I	寄存器组第二个操作寄存器地址
A3	I	寄存器组写入数据地址
wdata	I	寄存器组写入数据
address	I	最新的 pc 值
clk	I	时钟信号
reset	I	复位信号
RegWrite	I	寄存器组写操作控制信号
rdata1	O	寄存器组第一个输出数据
rdata2	O	寄存器组第二个输出数据

（5）ALU MUX（算数逻辑单元选择模块）

①基本描述

ALU 的输入选择。

文件	模块接口定义
alu_selector.v	module alu_selector(input [31:0] rdata, input [31:0] ext, input ALUSrc, output [31:0] out);

②模块接口

表 1 IFU 模块接口

信号名	方向	功能描述
rdata	I	寄存器组第二个输出数据
ext	I	扩展模块操作结果
ALUSrc	I	ALU 第二个操作数选择信号
out	O	ALU 的第二个输入数据

（6）ALU（算数逻辑单元）

①基本描述

实现多组算术运算和逻辑运算的组合逻辑电路。

文件	模块接口定义
alu.v	module alu(input [31:0] A, input [31:0] B, input [1:0] ALUOp, output zero, output [31:0] result);

②模块接口

表 1 IFU 模块接口

信号名	方向	功能描述
A	I	ALU 第一个操作数
B	I	ALU 第二个操作数
ALUOp	I	ALU 操作类型选择信号
zero	O	ALU 计算结果是否为 0
result	O	ALU 输出数据

（7）DM（数据存储器）

①基本描述

数据存储器是用于存放程序运行的中间处理数据的，可随程序运行而随时写入或读出数据存储器的内容。

文件	模块接口定义
dm.v	module dm(input [31:0] addr, input [31:0] WD, //write data input clk, input reset, input MemWrite, input [31:0] address, output [31:0] RD //read data);

②模块接口

表 1 IFU 模块接口

信号名	方向	功能描述
addr[31:0]	I	ALU 操作结果
WD[31:0]	I	寄存器组的第二个输出数据
clk	I	时钟信号
reset	I	复位信号
MemWrite	I	数据存储器写操作控制信号
address[31:0]	I	最新的 PC 值
RD[31:0]	O	从数据存储器中取出的数据

（8）Ext（扩展模块）

①基本描述

对数据进行需要的位扩展，分为高位扩展和低位扩展。

文件	模块接口定义
Ext.v	module Ext(input [15:0] imm, input [25:0] instr_index, input [1:0] ExtOp, input [31:0] P, output [31:0] extend);

②模块接口

表 1 IFU 模块接口

信号名	方向	功能描述

imm[15:0]	I	指令后 16 位编码
instr_index[25:0]	I	指令后 26 位编码
ExtOp[1:0]	I	扩展类型选择信号
P[31:0]	I	上一个 pc 的值
extend[31:0]	O	扩展后的数据输出

三、控制器设计

（1）单周期真值表

表 12 控制器真值表设计

Funct[5:0]	100001	100011							001000
OpCode[5:0]	000000		001101	100011	101011	000100	001111	000011	000000
	addu	subu	ori	lw	sw	beq	lui	jal	jr
RegA	0	0	0	0	0	0	x	x	1
ALUSrc	0	0	1	1	1	0	1	x	x
RegWrite	1	1	1	1	0	0	1	1	0
MemWrite	0	0	0	0	1	x	x	x	x
RegDst	01	01	00	00	x	x	00	10	x
Mem2Reg	00	00	00	01	x	x	00	10	x
nPC_Sel	00	00	00	00	00	01	00	10	11
ExtOp	x	x	00	00	11	x	01	10	x
ALUOp	00	01	10	00	00	01	11	x	x

四、测试要求

（1）测试程序

#测试 ori 指令

```
ori $a0,$0,123 #test reg0 && reg4
ori $a1,$a0,456 #test return of $a0 && reg5
ori $0,$a1,789 #load reg0
ori $t0,$0,0 #test reg0
```

#测试 lui 指令

```
lui $a2,123        #符号位为 0

lui $s0,0xffff     #符号位为 1
ori $s1,$s0,0xffff # $s0 = -1 && test reg16 && reg17
```

#测试 addu 指令

```
ori $t0,$0,123 #test reg8
ori $t1,$0,456 #test reg9
addu $t2,$t0,$t1 #test ++ && reg10
```

#测试 subu 指令

```
ori $t3,$0,123 #test reg11
ori $t4,$0,456 #test reg12
subu $t5,$t3,$t4 #test reg13 && - - = negative
subu $t6,$t4,$t3 #test reg14 && - - = positive
```

#测试 sw 指令 && reg15 - reg23

```
ori $t0,$0,0x0000 #base && test 15
sw $s2,0($t0)
sw $s3,4($t0)
sw $s4,8($t0)
sw $s5,12($t0)
sw $s6,16($t0)
sw $s7,20($t0)
```

#测试 lw 指令 && test reg24 && reg25

```
lw $t8,0($t0)
lw $t9,12($t0)
sw $a0,28($t0) #retest sw
sw $a1,32($t0)
```

#测试 beq 指令

```
ori $a0,$0,1
ori $a1,$0,2
ori $a2,$0,1
beq $a0,$a1,loop1 #unequal
beq $a0,$a2,loop2 #equal

loop1: sw $a0,36($t0)

nop #测试 nop 指令

jr $31 #测试 jr 指令
```

```
loop2: sw $a1,40($t0)
```

#测试 jal 指令

```
jal loop1
```

#测试 j 指令

```
j next1
```

#测试 xori 指令

```
next2:
xori $t7,$a0,2
xori $t7,$a0,3
```

#测试 lh 指令

```
next1:
lh $t2,($t0)  #back half word
lh $t2,2($t0) #front half word
```

#测试 bgtz 指令

```
lui $t0,0xffff
bgtz $t0,srlv_test
lui $t1,0x1111
ori $t1,$zero,0xffff
```

#测试 srlv 指令

```
srlv_test:
   ori $t1,$zero,3
   srlv $t2,$t0,$t1  #right logic move 3bit
   ori $t1,$zero,32
   srlv $t3,$t0,$t1 # right logic move 32bit
```

（2）测试期望

```
@00003000: $ 4 <= 0000007b
@00003004: $ 5 <= 000001fb
@0000300c: $ 8 <= 00000000
@00003010: $ 6 <= 007b0000
@00003014: $16 <= ffff0000
@00003018: $17 <= ffffffff
@0000301c: $ 8 <= 0000007b
```

```
@00003020: $ 9 <= 000001c8
@00003024: $10 <= 00000243
@00003028: $11 <= 0000007b
@0000302c: $12 <= 000001c8
@00003030: $13 <= fffffeb3
@00003034: $14 <= 0000014d
@00003038: $ 8 <= 00000000
@0000303c: *00000000 <= 00000000
@00003040: *00000004 <= 00000000
@00003044: *00000008 <= 00000000
@00003048: *0000000c <= 00000000
@0000304c: *00000010 <= 00000000
@00003050: *00000014 <= 00000000
@00003054: $24 <= 00000000
@00003058: $25 <= 00000000
@0000305c: *0000001c <= 0000007b
@00003060: *00000020 <= 000001fb
@00003064: $ 4 <= 00000001
@00003068: $ 5 <= 00000002
@0000306c: $ 6 <= 00000001
@00003084: *00000028 <= 00000002
@00003088: $31 <= 0000308c
@00003078: *00000024 <= 00000001
@00003098: $10 <= 00000000
@0000309c: $10 <= 00000000
@000030a0: $ 8 <= ffff0000
@000030b0: $ 9 <= 00000003
@000030b4: $10 <= 1fffe000
@000030b8: $ 9 <= 00000020
@000030bc: $11 <= ffff0000
```

五、思考题

（1）数据通路设计

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] dout; //read data

答：

因为一个存储单元占 4 个字节，输入的地址均为 4 的倍数，所以取[11:2]的输入端口即为对应的 memory 空间。

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：

部件：PC（程序计数器）、registers（寄存器组）、dm（数据存储器）。

原因：PC 的 reset 可以让 CPU 从第一条指令开始执行；registers 的 reset 可以将寄存器组中的寄存器全部清零复位，不 reset 会导致数据的混淆；dm 的 reset 清零数据存储器中的所有数据，不清零会导致 lw、sw 等指令的错误。

（2）控制器设计

1、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：

a、利用 if-else（或 case）完成操作码和控制信号的值之间的对应；

```
if(Op === 6'b000000){
  if(funct === 6'b100001){
     RegA = 0;
     RegDst[1] = 0;
     RegDst[0] = 1;
     ALUSrc = 0;
     Mem2Reg[1] = 0;
     Mem2Reg[0] = 0;
     RegWrite = 1;
     nPC_Sel[1] = 0;
     nPC_Sel[0] = 0;
     ExtOp[1] = 0;
     ExtOp[0] = 0;
```

```
    ALUOp[1] = 0;
    ALUOp[0] = 0;
}
else if(funct === 6'b100011){
    RegA = 0;
    RegDst[1] = 0;
    RegDst[0] = 1;
    ALUSrc = 0;
    Mem2Reg[1] = 0;
    Mem2Reg[0] = 0;
    RegWrite = 1;
    nPC_Sel[1] = 0;
    nPC_Sel[0] = 0;
    ExtOp[1] = 0;
    ExtOp[0] = 0;
    ALUOp[1] = 0;
    ALUOp[0] = 1;
}
else{
    RegA = 1;
    RegDst[1] = 0;
    RegDst[0] = 0;
    ALUSrc = 0;
    Mem2Reg[1] = 0;
    Mem2Reg[0] = 0;
    RegWrite = 0;
    nPC_Sel[1] = 1;
    nPC_Sel[0] = 1;
    ExtOp[1] = 0;
    ExtOp[0] = 0;
```

```
        ALUOp[1] = 0;
        ALUOp[0] = 0;
      }
    }
    else if(Op === 6'b001101){
      RegA = 0;
      RegDst[1] = 0;
      RegDst[0] = 0;
      ALUSrc = 1;
      Mem2Reg[1] = 0;
      Mem2Reg[0] = 0;
      RegWrite = 1;
      nPC_Sel[1] = 0;
      nPC_Sel[0] = 0;
      ExtOp[1] = 0;
      ExtOp[0] = 0;
      ALUOp[1] = 1;
      ALUOp[0] = 0;
    }
    else{
      ......
    }
```

b、利用 assign 语句完成操作码和控制信号的值之间的对应；

```
    assign RegA = (Op === 6'b000000 && funct === 6'b001000)?1:0;
    assign RegDst[1] = (Op === 6'b000011)?1:0;
    assign RegDst[0] = (Op === 6'b000000)?1:0;
```

```
    assign ALUSrc = (Op === 6'b001101 || Op === 6'b100011
|| Op === 6'b101011 || Op === 6'b001111)?1:0;
    assign Mem2Reg[1] = (Op === 6'b000011)?1:0;
    assign Mem2Reg[0] = (Op === 6'b100011)?1:0;
    assign RegWrite = ((Op === 6'b000000 && funct ===
6'b100001) || (Op === 6'b000000 && funct === 6'b100011) ||
Op === 6'b001101 || Op === 6'b100011 || Op === 6'b001111
|| Op === 6'b000011)?1:0;
    assign MemWrite = (Op === 6'b101011)?1:0;
    assign nPC_Sel[1] = (Op === 6'b000011 || (Op ===
6'b000000 && funct === 6'b001000))?1:0;
    assign nPC_Sel[0] = (Op === 6'b000100 || (Op ===
6'b000000 && funct === 6'b001000))?1:0;
    assign ExtOp[1] = (Op === 6'b000011 || Op ===
6'b101011)?1:0;
    assign ExtOp[0] = (Op === 6'b001111 || Op ===
6'b101011)?1:0;
    assign ALUOp[1] = (Op === 6'b001101 || Op ===
6'b001111)?1:0;
    assign ALUOp[0] = (Op === 6'b000100 || (Op ===
6'b000000 && funct === 6'b100011))?1:0;
```

c、利用宏定义

```
    `define REGA(Op,funct) (Op === 6'b000000 && funct ===
6'b001000)?1:0
    `define REGDST1(Op) (Op === 6'b000011)?1:0
    `define REGDST0(Op) (Op === 6'b000000)?1:0
    `define ALUSRC(Op) (Op === 6'b001101 || Op ===
6'b100011 || Op === 6'b101011 || Op === 6'b001111)?1:0
    `define MEM2REG1(Op) (Op === 6'b000011)?1:0
    `define MEM2REG0(Op) (Op === 6'b100011)?1:0
```

```
...

module test(
   input [5:0] Op,
   input [5:0] funct,
   output RegA,
   output [1:0] RegDst,
   output ALUSrc,
   output [1:0] Mem2Reg
   ...
    );

   assign RegA = `REGA(Op,funct);
   assign RegDst[1] = `REGDST1(Op);
   assign RegDst[0] = `REGDST0(Op);
   assign ALUSrc = `ALUSRC(Op);
   assign Mem2Reg[1] = `MEM2REG1(Op);
   assign Mem2Reg[0] = `MEM2REG0(Op);
   ...
endmodule
```

2、根据你所列举的编码方式，说明他们的优缺点。

答：

a、if-else：

优点：判断清晰明了。

缺点：每一次判断后都要对所有控制信号进行赋值，代码比较冗长。

b、assign：

优点：非常简洁，逻辑性非常强。

缺点：一旦有个地方出错了，可能比较难以定位。

c、宏定义：

优点：方便修改；提高可读性以及程序的运行效率。

缺点：参数的直接替换，其合法性难以保证，存在安全隐患。

（3）综合

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。

答：

以下是 ADDI 的运算过程：

Operation:

```
temp ← (GPR[rs]31||GPR[rs]31..0) + sign_extend(immediate)
if temp32 ≠ temp31 then
   SignalException(IntegerOverflow)
else
   GPR[rt] ← temp
endif
```

以下是 ADDIU 的运算过程：

Operation:

```
temp ← GPR[rs] + sign_extend(immediate)
GPR[rt]← temp
```

忽略溢出，ADDI 中 $temp_{31}$ 与 $temp_{32}$ 不相等的情况被忽略了，永远执行 else 的部分，就与 ADDIU 的执行过程一模一样，其本质是相同的，所以说 ADDI 与 ADDIU 是等价的。

以下是 ADD 的运算过程：

Operation:

```
temp ← (GPR[rs]31||GPR[rs]31..0) + (GPR[rt]31||GPR[rt]31..0)
if temp32 ≠ temp31 then
   SignalException(IntegerOverflow)
else
   GPR[rd] ← temp
endif
```

以下是 ADDU 的运算过程：

Operation:

```
temp ← GPR[rs] + GPR[rt]
GPR[rd] ← temp
```

同理，如果忽略溢出，ADD 的运算永远执行 else 部分，其本质与 ADDU 的运算过程是一模一样的，所以说如果忽略溢出，ADD 与 ADDU 是等价的。

2、根据自己的设计说明单周期处理器的优缺点。

答：

优点：控制部件简单易实现。

缺点：一个时间周期执行一条指令，执行时间最长的指令决定了时间周期的长度，不管指令的复杂度是什么样的，都需要花费相同的时间去执行，造成了时间上的浪费。

3、简要说明 jal、jr 和堆栈的关系。

答：

在写递归函数的时候，为了让 jal 对应正确的返回地址，需要将返回地址存在堆栈中，jr 取堆栈中的地址进行跳转。