CPU设计文档

1. 数据通路设计
   1. pc（程序计数器）

模块端口说明如下：

表1 pc端口说明



模块功能定义如下：

表2 pc功能定义



* 1. im（指令存储器）

模块端口说明如下：

表3 im端口说明



模块功能定义如下：

表4 im功能定义



（2）grf（通用寄存器组）

模块端口说明如下：

表5 grf端口说明



模块功能定义如下：

表6 grf功能定义



（3）alu（算术逻辑单元）

模块端口说明如下：

表7 alu端口说明



模块功能定义如下：

表8 alu功能定义



1. dm（数据存储器）

模块端口说明如下：

表9 dm端口说明

模块功能定义如下：

表10 dm功能定义



1. EXT（数据扩展单元）

模块端口说明如下：

表11 ext端口说明



模块功能定义如下：

表12 ext功能定义



二、控制器设计

控制器端口说明如下：

表13 控制器端口说明



控制信号真值表如下：

表14 控制信号真值表



控制信号意义如下：

表15 控制信号意义



三、测试程序

测试程序源代码如下：

lui $s0, 0x2333

ori $s1, $s0, 0x6666

addu $s2, $s1, $s0

subu $s3, $s1, $s0

lui $0, 0x7777

ori $0, $0, 0x5555

sw $s2, 0($0)

sw $s3, 4($0)

lw $s4, 4($0)

addu $s5, $0, $0

beq $s5, $0, next

addu $s5, $0, $s1

next:

addu $s6, $0, $s1

lui $s7, 0

jal next2

jal end

lui $s7, 0x2222

next2:

ori $s7, 0x3333

jr $ra

end:

nop

beq $0, $0, end nop

期望运行结果：

寄存器s0 – s7的值分别为：

0x23330000、 0x23336666、 0x46666666、 0x00006666、

0x00006666、 0x00000000、 0x23336666、 0x00002333

数据存储器中地址0x0、0x4中的值分别为0x46666666、 0x00006666

思考题

1. addr信号截取了32信号的11-2位，用[11:2]比[9:0]更加清楚直观。

addr信号的来源为ALU的运算结果。

1. 清零信号针对PC、寄存器堆、数据存储器进行清零复位操作。

这些部件保存了当前的状态信息，因而需要进行清零。

1. coding方式
2. 利用case完成操作码和控制信号之间的对应

reg [12:0] Output\_Bus;

assign nPC\_Op = Output\_Bus[12:11];

assign RegWrite = Output\_Bus[10];

assign RegDst = Output\_Bus[9:8];

assign RegSrc = Output\_Bus[7:6];

assign EXTOp = Output\_Bus[5:4];

assign ALUOp = Output\_Bus[3:2];

assign ALUSrc = Output\_Bus[1];

assign MemWrite = Output\_Bus[0];

always @ (Op or Funct or ALU\_Zero) begin

case (Op)

6'b000000 :

case (Funct)

6'b100001 : Output\_Bus = 13'b 00\_1\_01\_00\_xx\_00\_0\_0;

6'b100011 : Output\_Bus = 13'b 00\_1\_01\_00\_xx\_01\_0\_0;

6'b001000 : Output\_Bus = 13'b 11\_0\_xx\_xx\_xx\_xx\_x\_0;

default : Output\_Bus = 13'b 00\_0\_xx\_xx\_xx\_xx\_x\_0;

endcase

6'b001101 : Output\_Bus = 13'b 00\_1\_00\_00\_01\_10\_1\_0;

6'b100011 : Output\_Bus = 13'b 00\_1\_00\_01\_00\_00\_1\_0;

6'b101011 : Output\_Bus = 13'b 00\_0\_xx\_xx\_00\_00\_1\_1;

6'b000100 : Output\_Bus = {1'b0, ALU\_Zero, 11'b0\_xx\_xx\_xx\_00\_0\_0};

6'b001111 : Output\_Bus = 13'b 00\_1\_00\_00\_10\_10\_1\_0;

6'b000011 : Output\_Bus = 13'b 10\_1\_10\_10\_xx\_xx\_x\_0;

default : Output\_Bus = 13'b 00\_0\_xx\_xx\_xx\_xx\_x\_0;

endcase

end

1. 利用assign语句完成操作码和控制信号的值之间的对应

wire r, addu, subu, jr, ori, lw, sw, beq, lui, jal;

assign r = &(~Op);

assign addu = r & Op[5] & ~Op[4] & ~Op[3] & ~Op[2] & ~Op[1] & Op[0];

assign subu = r & Op[5] & ~Op[4] & ~Op[3] & ~Op[2] & Op[1] & Op[0];

assign jr = r & ~Op[5] & ~Op[4] & Op[3] & ~Op[2] & ~Op[1] & ~Op[0];

assign ori = ~Funct[5] & ~Funct[4] & Funct[3] & Funct[2] & ~Funct[1] & Funct[0];

assign lw = Funct[5] & ~Funct[4] & ~Funct[3] & ~Funct[2] & Funct[1] & Funct[0];

assign sw = Funct[5] & ~Funct[4] & Funct[3] & ~Funct[2] & Funct[1] & Funct[0];

assign beq = ~Funct[5] & ~Funct[4] & ~Funct[3] & Funct[2] & ~Funct[1] & ~Funct[0];

assign lui = ~Funct[5] & ~Funct[4] & Funct[3] & Funct[2] & Funct[1] & Funct[0];

assign jal = ~Funct[5] & ~Funct[4] & ~Funct[3] & ~Funct[2] & Funct[1] & Funct[0];

assign nPC\_Op = {jr & lui, jr & ALU\_Zero};

assign RegWrite = addu & subu & ori & lw & lui & jal;

assign RegDst = {jal, addu & subu};

assign RegSrc = {jal, lw};

assign EXTOp = {lui, ori};

assign ALUOp = {ori & lui, subu};

assign ALUSrc = ori & lw & sw & beq;

assign MemWrite = sw;

1. 利用宏定义

`define ADDU\_FUNCT 6'b100001

`define SUBU\_FUNCT 6'b100011

`define JR\_FUNCT 6'b001000

`define R\_OP 6'b000000

`define ORI\_OP 6'b001101

`define LW\_OP 6'b100011

`define SW\_OP 6'b101011

`define BEQ\_OP 6'b000100

`define LUI\_OP 6'b001111

`define JAL\_OP 6'b000011

reg [12:0] Output\_Bus;

assign nPC\_Op = Output\_Bus[12:11];

assign RegWrite = Output\_Bus[10];

assign RegDst = Output\_Bus[9:8];

assign RegSrc = Output\_Bus[7:6];

assign EXTOp = Output\_Bus[5:4];

assign ALUOp = Output\_Bus[3:2];

assign ALUSrc = Output\_Bus[1];

assign MemWrite = Output\_Bus[0];

always @ (Op or Funct or ALU\_Zero) begin

case (Op)

`R\_OP :

case (Funct)

`ADDU\_FUNCT : Output\_Bus = 13'b 00\_1\_01\_00\_xx\_00\_0\_0;

`SUBU\_FUNCT : Output\_Bus = 13'b 00\_1\_01\_00\_xx\_01\_0\_0;

`JR\_FUNCT : Output\_Bus = 13'b 11\_0\_xx\_xx\_xx\_xx\_x\_0;

default : Output\_Bus = 13'b 00\_0\_xx\_xx\_xx\_xx\_x\_0;

endcase

`ORI\_OP : Output\_Bus = 13'b 00\_1\_00\_00\_01\_10\_1\_0;

`LW\_OP : Output\_Bus = 13'b 00\_1\_00\_01\_00\_00\_1\_0;

`SW\_OP : Output\_Bus = 13'b 00\_0\_xx\_xx\_00\_00\_1\_1;

`BEQ\_OP : Output\_Bus = {1'b0, ALU\_Zero, 11'b0\_xx\_xx\_xx\_00\_0\_0};

`LUI\_OP : Output\_Bus = 13'b 00\_1\_00\_00\_10\_10\_1\_0;

`JAL\_OP : Output\_Bus = 13'b 10\_1\_10\_10\_xx\_xx\_x\_0;

default : Output\_Bus = 13'b 00\_0\_xx\_xx\_xx\_xx\_x\_0;

endcase

end

1. 各种方法的优缺点
2. 利用case语句完成操作码和控制信号之间的对应

优点：编码方式简单易行

1. 利用assign语句完成操作码和控制信号的值之间的对应

优点：可以看到控制器电路的门级结构

缺点：编码方式不够清晰易懂

1. 利用宏定义：

优点：可以使语句更加清晰易懂

1. add和addi指令会检测运算是否发生了溢出，而addu和addiu指令不检测。因而在忽略溢出的前提下，addi与addiu是等价的，add与addu是等价的。
2. 单周期处理器优点：设计简单易行

单周期处理器缺点：

（1）需要足够长的周期来完成最慢的指令，因而效率太低。

（2）难以支持包含浮点或更复杂指令的指令集。

7、jal、jr指令主要用于函数调用，在函数调用过程中需要把一些变量以堆栈的形式存储到内存中。在函数调用中，用jal跳转后进行压栈，将栈释放后用jr指令跳回。