**实验八 单周期CPU的设计**

**一、实验目的**

1、掌握 MIPS R 型指令的数据通路设计，掌握指令流和数据流的控制方法；

2、掌握完整的单周期 CPU 顶层模块的设计方法；

3、实现 MIPS R 型指令的功能。

**二、实验设备**

1、装有vivado的计算机 1台

2、EGO1开发板 1块

**三、实验任务**

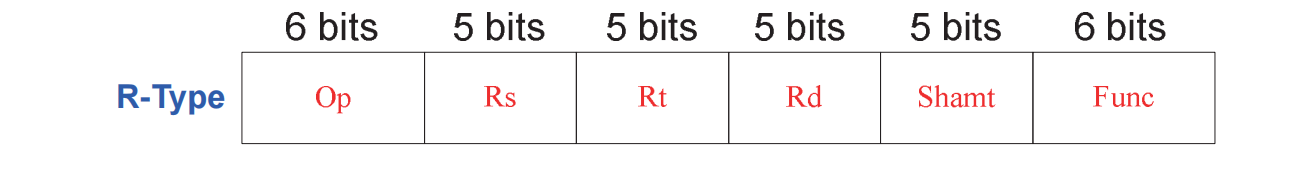
1、设计MIPS R 型指令的数据通路设计。

2、设计单周期 CPU 顶层模块。

3、设计MIPS R 型指令并且进行仿真验证。

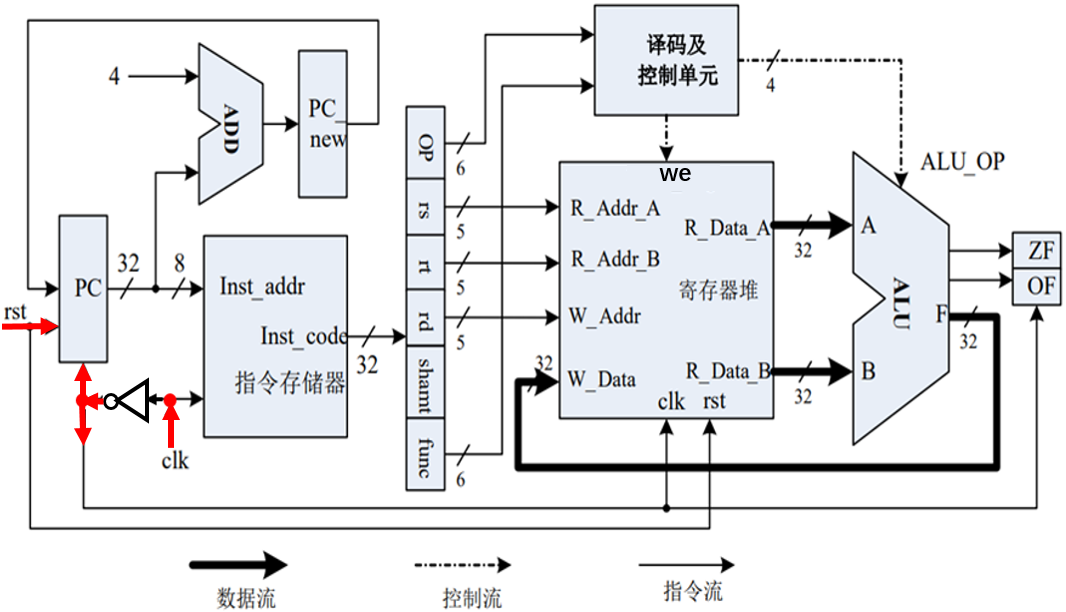
**四、实验原理**

1、MIPS R 型指令格式如图8.1所示。

图8.1

其中， Op:指令操作码；Rs:第一个源操作数寄存器号，参与运算使用；Rt:第二个源操作数寄存器号，参与运算使用；Rd:目的操作数寄存器号，保存结果使用；Shamt:位偏移量，仅在位移指令使用，在此直接置0；Func:运算类型码，用于选择Op操作中的具体运算类型。比如加法运算，在Op指令操作码中，指出它是算术运算；在Func运算类型码中，指出它是算术运算中的加法运算。最后的运算即为：$Rd=$Rs+$Rt。

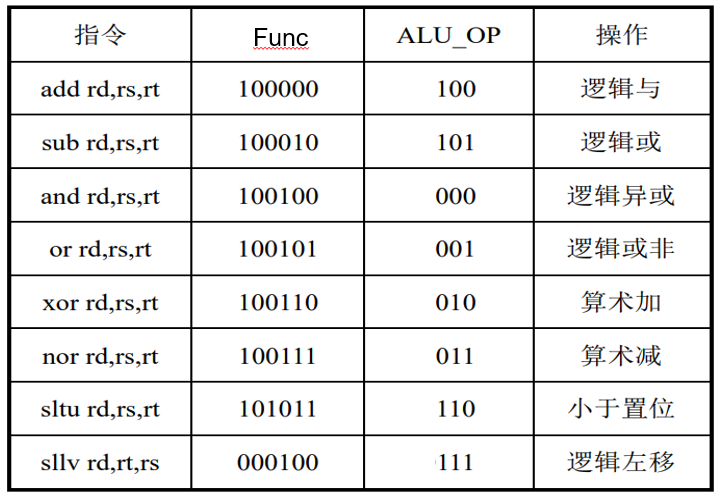
为了实现对应的机器指令，在CPU内需要有取指令部件、指令译码部件、指令执行部件和将执行结果写回的部件。硬件结构示意图如图8.2所示。

图8.2

上图8.2中，clk是时钟信号。PC是程序计数器，该寄存器里面存储的值，可以直接送出。而要往PC寄存器中写一个新的值，必须在clk的下降沿处。对于指令存储器ROM，是用来存放要执行的程序的。具体存放的指令可以参考本文末尾处的代码。对于寄存器堆，在clk上升沿处进行读；在clk下降沿处，可以往寄存器堆里进行写操作。译码和控制单元，接收来自于指令寄存器中指令操作码OP和运算类型码Func，输出为控制寄存器堆读/写允许信号we（参见实验六）和控制运算器运算类型的信号ALU-OP（参见实验五）。一般情况下，系统里时序电路模块是在clk上升沿处读，在clk下降沿处进行写，这样可以有效避免同时对同一个寄存器进行读和写的冲突。

假设现在有 MIPS指令集的一个 R 型指令核心子集，它包含 8 条指令，指令格式和编码如下表8.1所示。这8条 R 型指令的共同特征是：操作码字段 Op=000000B；指令的功能则由运算类型码字段 Func 指出。这8条 R 型指令中，每条 R 型指令的操作数有 3 个：两个源操作数分别在rs和rt字段所指定的寄存器中，而目的操作数则是 rd字段所指定的寄存器。注意，汇编助记符中，紧跟在汇编指令右边的寄存器是目的寄存器，这和指令机器码的排列顺序有差异。譬如，add r1, r2, r3（add $1, $2, $3）指令执行的功能是 r2+r3→r1，其机器指令对应编码是 32’b000000\_00010\_00011\_00001\_00000\_100000；即rs是r2（编码是 00010），rt是r3（编码是 00011），rd是r1（编码是 00001）。

表8.1



2、为实现上述 8条R型指令，接下来需要建立数据通路。将实验五和六所实现的 ALU、寄存器堆进行互联，并添加指令存储器、取指部件、指令译码与控制部件。其中，从指令存储器取出的指令，经过初级译码，将分解出的源寄存器 rs、rt 直接与寄存器堆的两个读端口A 和 B 的寄存器地址连接，而将目的寄存器rd字段与寄存器堆的写端口地址相联，寄存器读出的A口数据和 B口数据则直接连接到 ALU 的输入端 A 和 B，ALU 计算后的结果则送入寄存器堆的写数据端口。对于当前指令的 Op和 Func 字段则交由指令译码部件处理：它首先会根据 Op 字段判定 是否是R型指令，如果是，则再将Func 字段，翻译成 ALU 的控制信号 ALU\_OP，以指定 ALU 的运算功能。

接下来需要考虑指令执行的时序了。在这之前，先梳理指令执行的过程：首先取指令（根 据 PC 从指令存储器读取），然后进行指令译码（将指令各字段送到目的部件，指令译码器 则指定 ALU 的运算操作），最后运算结果送目的寄存器，PC 自增指向下一条指令。如何在一个指令周期内按序完成上述一系列操作呢？传统的硬布线控制方法很直接：一个指令周期包含若干个机器周期（又称 CPU 周期），分别用于完成取指令、分析指令和执行指令；而在一个机器周期内，又包含若干个时钟周期（又称节拍）， 用于控制数据流的流动。总之，传统的时序系统用多级的时序信号（机器周期和节拍、工作脉冲）来同步各个操作和微操作。 回到本实验的设计目标：设计实现的是单周期 CPU，即在一个时钟周期内完成一条指令， 包括取指令、分析指令和执行指令。那么问题转换为：如何在一个时钟脉冲周期内按序完成上述一系列有序的操作步骤？首先可在前述实验的基础上，分析哪些部件是时序电路，哪些 部件是组合逻辑，也即：哪些部件需要时钟脉冲同步，哪些部件不需要时钟脉冲。显然：指 令存储器的读操作、PC 值的更新、寄存器的写操作及标志寄存器的更新需要clk，其他诸如寄存器的读操作、ALU 的运算操作等皆不需要 clk，可视为组合逻辑电路。因此，可以如此设计时序：在 clk 的上升沿，启动指令存储器依据PC读出指令；在clk高电平持续期间，可以完成 PC 值的自增（+4）、指令译码、寄存器读操作，随后完成 ALU 运算；在 clk 的下降沿则完成目的寄存器的写入、PC 值的更新和标志寄存器的更新。所以，在图 8.2 中，将 clk 反相后作为寄存器堆、PC 和标志寄存器的打入脉冲。

在本实验中，只有 add 和 sub 指令是属于带符号数的算术运算，需要根据运算结果置 零标志位ZF 和溢出标志位OF，而其他指令只需要设置ZF零标志，无需设置OF溢出标志。所以每条指令在译码执行时， 需要区别对待。

**五、实验步骤**

**（一）设计一个单周期的CPU,如图8.2所示，该单周期CPU可以执行8条机器指令，对应的指令如表8.1所示。（实验室只进行仿真验收）**

1、**新建Vivado 工程**

**参照实验五的步骤创建新工程。**

**2、参考图8.2，编写子模块和顶层模块对应的verilog源代码。**

（1）参考图8.2，设计有8种运算功能的对应的运算器模块，参考代码如下：

module ALU(A,B,F,ALU\_OP,ZF,OF);

input [31:0] A,B;

input [2:0] ALU\_OP;

output reg ZF,OF;

output reg [31:0] F;

reg C;

always @(\*)

begin

C=0;

case (ALU\_OP)

3'b000:begin F=A&B; end

3'b001:begin F=A|B; end

3'b010:begin F=A^B; end

3'b011:begin F=~(A|B); end

3'b100:begin {C,F}=A+B; end

3'b101:begin {C,F}=A-B; end

3'b110:begin F=A<B; end

3'b111:begin F=B<<A; end

endcase

end

always @(\*)

begin

ZF= F==0;

OF= A[31]^B[31]^F[31]^C;

end

endmodule

（2）参考图8.2，设计寄存器堆模块，参考代码如下：

module Registerfile(input [4:0] R\_Addr\_A,

input [4:0] R\_Addr\_B,

input [4:0] W\_Addr,

input We, //低电平允许写，高电平允许读。

input [31:0] W\_Data,

input Clk,

input Reset,

output [31:0] R\_Data\_A,

output [31:0] R\_Data\_B

);

reg [31:0] REG\_Files[0:31];

reg [5:0] i;

initial //初始化

begin

for(i=0;i<32;i=i+1)

REG\_Files[i]=0;

end

assign R\_Data\_A = REG\_Files[R\_Addr\_A];

assign R\_Data\_B = REG\_Files[R\_Addr\_B];

always@(posedge Clk or posedge Reset)

begin

if(Reset) //清零重置

for(i=0;i<32;i=i+1)

REG\_Files[i]=0;

else

if(！We)

REG\_Files[W\_Addr]=W\_Data;

end

endmodule

（3）参考图8.2，设计取指令模块，参考代码如下：

module Fetch\_Inst(

input rst,

input clk,

output [31:0] Inst\_code

);

wire [31:0] PC\_new;

reg [31:0] PC;

initial

PC=32'h00000000;

ROM Inst\_addr( .clka(clk), .addra(PC[7:2]), .douta(Inst\_code) );

assign PC\_new=PC+4;

always@(posedge rst or negedge clk)

begin

if(rst)

PC=32'h00000000;

else

PC={24'h000000,PC\_new[7:0]};

end

endmodule

（4）对应ROM模块的设计，可以采用vivado里面的设计IP核的方式来实现。也可以定义一个txt文件，将指令的机器码放在在文件中，读该txt文件，从中取出每条指令赋值给寄存器组。

根据上面对指令存储器ROM输入和输出的分析，完成下面ROM模块，填写对应代码如下：

如果采用IP核的方式来实现，就不需要填写下面的与ROM有关的verilog代码了。

`timescale 1ns / 1ps

module ROM();

请在此处补充对应的verilog代码，要求在实验前完成。进实验室时会检查。验收时会围绕核心代码进行提问。

endmodule

（5）根据图8.2电路里，指令译码与控制部件模块的输入和输出，完成下面对应的代码：

`timescale 1ns / 1ps

module control();

请在此处补充对应的verilog代码，要求在实验前完成。进实验室时会检查。验收时会围绕核心代码进行提问。

endmodule

（6）根据图8.2中CPU整体电路结构，完成CPU顶层模块的设计，在下面填写对应的代码：

`timescale 1ns / 1ps

module CPU( );

请在此处补充对应的verilog代码，要求在实验前完成。进实验室时会检查。验收时会围绕核心代码进行提问。

endmodule

**3、在Vivado中进行仿真测试主模块。**

建立了单周期CPU顶层模块后，如果需要对模块进行测试的话，就需要给电路添加相应的激励输入，主要是给出时钟clk和复位信号rst的波形。观察电路中指令执行情况，观察执行结果输出波形是否正确。

**激励输入模块：**

`timescale 1ns / 1ps

module CPUtest(…… )

请在此处补充对应的verilog代码，要求在实验前完成。进实验室时会检查。验收时会围绕核心代码进行提问。

endmodule

进行仿真，验证仿真后的波形图是否正确。

在下面粘贴你的仿真截图，并分析结果的正确性。

**4、在Vivado中设计输入控制电路，将按键sw0~sw9分别对应于10条指令的启动控制信号，消除抖动的影响。每推上一个按键就执行对应的指令。**

（1）输入控制电路，将按键sw0~sw9分别对应于10条指令的启动控制信号，每推上一个按键就执行对应的指令。例如：sw6推向上方，表示执行第6条执行，sw8推向上方，表示执行第8条指令。为了消除按钮的抖动对输入的影响，可以设计一个定时时间为10ms的定时器。第一次输入读数后，延时10ms后再读第二次，如果第二次和第一次读到的对应按键的状态相同，就认为消除了抖动的影响。

**定时10ms模块：**

module counter10(…… )

感兴趣的同学请在此处补充对应的verilog代码。进实验室时不会检查。不作验收要求。

endmodule

**输入模块：**

module inputBUT(…… )

感兴趣的同学请在此处补充对应的verilog代码。进实验室时不会检查。不作验收要求。

endmodule

**5、在Vivado中进行引脚绑定，并且将指令的执行结果通过七段显示器进行动态显示。**

（1）要想在多个七段显示器上进行动态刷新显示指令执行结果，需要设计定时50ms的时钟，每隔50ms就在七段显示器上显示一条指令的执行结果中的几位。由于视觉暂停效应，人眼是看不清多个七段显示器的值是分时显示的。人眼感觉到的是多个七段显示器上值是同时显示的。

**定时50ms的模块：**

module counter50(…… )

感兴趣的同学请在此处补充对应的verilog代码。进实验室时不会检查。不作验收要求。

endmodule

**每隔50ms显示七段显示器一位的控制模块**

module showCPU(…… )

感兴趣的同学请在此处补充对应的verilog代码。进实验室时不会检查。不作验收要求。

endmodule

**6、在vivado中生成bit流文件和下载，在EGO1开发板上进行按钮控制(**不作验收要求。**),观察七段显示器上的指令执行输出结果。验证结果是否正确。**

**7、存放在ROM中的要执行的指令参考如下：**

#baseAddr 0000

nor $1, $O, $0; //Reg[1]=FFFF\_FFFF

sltu $2,$O,$1; //Reg[2]=0000\_0001

add $3,$2,$2; //Reg[3]=0000\_0002

add $4,$3,$2; //Reg[4]=0000\_0003

add $5,$4,$3; //Reg[5]=0000\_0005

add $6,$5,$3; //Reg[6]=0000\_0007

sllv $7,$6,$2; //Reg[7]=0000\_00OE

add $9,$s5,$6; //Reg[9]=0000\_000C

sllv $8,$6,$9; //Reg[8]=0000\_7000

xor $0,$1,$8; //Reg[9]=FFFF\_8FFF

add $10,$9,$1; //Reg[10]=FFFF\_8FFE

sub $11,$8,$7; //Reg[11]=0000\_6FF2

sub $12,$7,$8; //Reg[12]=FFFF\_90OE

and $13,$9,$12; //Reg[13]=FFFF\_80OE

or $14, $9,$12; //Reg[14]=FFFF\_OFFF

……

**六、实验思考：**

1、图8.2是MIPS指令系统中与R型指令有关的电路结构，如果是MIPS指令系统中的I型指令，所对应的电路结构是怎样的。

2、图8.2是MIPS指令系统中与R型指令有关的电路结构，如果是MIPS指令系统中的J型指令，所对应的电路结构是怎样的。

3、本实验实现的 sltu 指令是对无符号数的比较置位指令，如果需要实现有符号数的比较置位指令——slt 指令，请问应该如何实现？