浙江大学

本科实验报告

课程名称：计算机组成

姓　　名：胡亮泽

学　　院：计算机科学与技术学院

　　系：计算机科学与技术系

专　　业：计算机科学与技术

学　　号：3120102116

指导教师：姜晓红

2014 年　5　月　16　日

浙江大学实验报告

课程名称：Computer Organization 实验类型：综合

实验项目名称：Lab8：单周期 CPU 实现

学生姓名：胡亮泽 专业：计算机科学与技术

学号：3120102116

同组学生姓名：王艺 指导老师：姜晓红

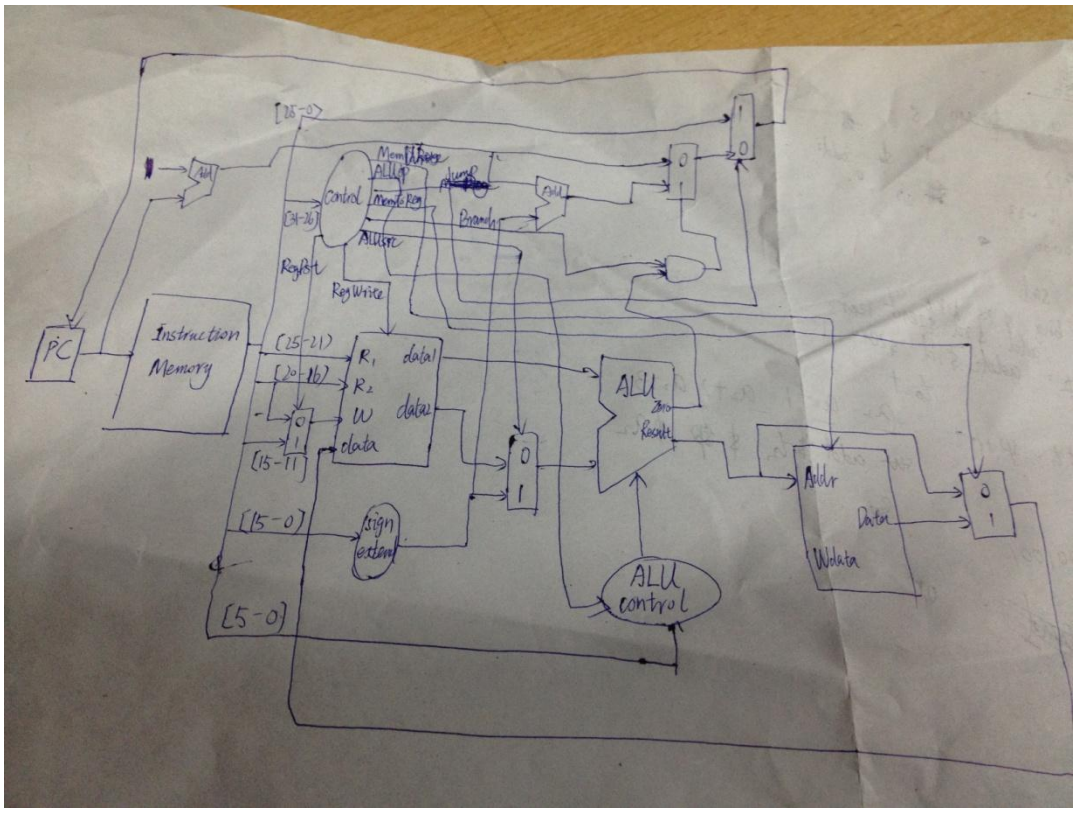
实验地点：东 4-509 实验日期：2014 年 5 月 16 日

一、 实验目的和要求

利用 verilog 代码实现单周期 CPU 的数据通路，并下载完成相应的指令操作。

二、 实验内容和原理

数据通路图如下：



本次实验中由于数据通路较为复杂，因此模块众多。主要包括存储器模块（data),指令存储器模块（Inst），ALUC , ALU ，与 pc 相关的 single_pc_plus_4, single_pc 模块，以及 control 和 5 位以及 32 位多路选择器等模块。

其中指令存储器和存储器模块主要是通过 IP 核模板生成，而 ALU，ALUC 等模块前几次实验都已经详细叙述过，这里不多做赘述。

由于是单周期 CPU，因此指令需要在一个时钟周期内完成，因此，在关于寄存器群的代码中，采用下降沿时进行写操作：

```
module single_gpr(input wire clk,
                input wire rst,
                      input wire i_wen,
                      input wire[4:0] i_addr1,i_addr2,i_addr3,
                      input wire[4:0] i_wreg,
                      input wire[31:0] i_wdata,
                      output wire[31:0] o_op1,o_op2,o_op3

    );

reg[31:0] mem[31:0];
initial begin
mem[0] = 0;
mem[1] = 0;
mem[2] = 0;
mem[3] = 0;
mem[4] = 0;
……    //初始化
end
```

```
assign o_op1 = mem[i_addr1];
assign o_op2 = mem[i_addr2];
assign o_op3 = mem[i_addr3];

always @(negedge clk,posedge rst)
if (rst) begin
   mem[0] = 0;
mem[1] = 0;
mem[2] = 0;
mem[3] = 0;
…… //重置
end
else if(i_wen)
   if(i_wreg!=0)
   mem[i_wreg] = i_wdata;

endmodule
```

需知，

```
assign o_op3 = mem[i_addr3];
```

主要用于在 top 主模块中输出验证选择寄存器内存储的值。其余内容与试验 6 完全相同，这里不多做赘述。

而 single_pc_plus_4 主要是用于输出下一个 pc+1 的值：

```
module single pc plus 4(
                    input wire[31:0] i_pc,
                            output wire[31:0] o_pc
   );
assign o_pc = i_pc + 1;

endmodule
```

由于在 IP 核中，每一行代表一条指令，而指令地址即为该指令的行数，因此在 IP 核中，指令地址是连续的整数，只需要加 1 就可以了。

至于 single_pc 模块，主要是用于将下一个 pc 的值传递给指令 IP 核作为下一条指令的地址，代码如下：

```
module single_pc(
               input wire clk,
                    input wire rst,
                    input wire[31:0] i_pc,
                    output wire[31:0] o_pc
   );
reg[31:0] t_pc;

initial t_pc = 32'hFFFFFFFF;
assign o_pc = rst?32'hFFFFFFFF:t_pc;

always @(posedge clk,posedge rst)
if(rst) t_pc <= 32'hFFFFFFFF;
else
t_pc <= i_pc;


endmodule
```

这里，我们用一个 t_pc 的变量来作为缓冲，在不重置的前提下，输出的 pc

值永远与 t_pc 值相同，但是考虑到只有在时钟上升沿的时候才需要读取新的指令，因此 t_pc 在 clk 上升沿的时候改变值。

其中 i_pc 是经过多个选择器后选择出来的 pc 值。

和 pc 需要通过加法计算下一个 pc 值类似，在 branch 操作中也需要计算下一个 pc 值，因此有了 branch_addr 模块,原理类似：

```
module branch_addr(
                input wire[31:0] pc_plus_4,
                      input wire[31:0] exten_branch,
                      output wire[31:0] branch_addr
   );


assign branch_addr = pc_plus_4 + exten_branch;

endmodule
```

Exten_branch 是符号位扩展后的偏移量。

最后，用顶层模块 CPU 模块将所有模块进行连接：

```
module CPU(
         input wire clk,   //T9
            input wire exec, //单周期时钟周期
         input wire rst,    //重置
         input wire[1:0] disp_type,  //显示类型
            input wire[4:0] reg_index,  //输出寄存器下标
            output wire[5:0] led,    //各 led 灯
            output wire[7:0] segment,
            output wire[3:0] anode

   );

wire[31:0] pc;
wire [31:0] pc_plus_4;
wire [31:0] branch_address;
wire [31:0] instruction;
wire
RegDst,ALUsrcB,MemToReg,WriteReg,MemWrite,Branch,ALUop1,ALUop0,Ju
mp;
wire[31:0] bracn_or_next,jump_or_not;
wire[4:0] rt_or_rd;
wire[31:0] addr_or_data,result_or_data;
wire [31:0] exten_branch,exten_jump,branch_or_next;
wire equal_branch;
wire[31:0] mem_data;
wire[2:0] func;
wire[1:0] ALUop;
wire[31:0] result;
wire zero;
reg[15:0] disp_num;
reg[15:0] clock;
wire[31:0] Rdata1,Rdata2,Rdata3;
wire exec_out,rst_out;

initial begin
```

```
clock = 0;

end

pbdebounce p1(clk,exec,exec_out);
pbdebounce p2(clk,rst,rst_out);

assign led[0] = (pc!=32'hFFFFFFFF);
assign led[1] = (instruction[31:26]==0&led[0]);
assign led[2] = (instruction[31:26]==35&led[0]);
assign led[3] = (instruction[31:26]==43&led[0]);
assign led[4] = (instruction[31:26]==4&led[0]);
assign led[5] = (instruction[31:26]==2&led[0]);

data
mem1(.addr(result[8:0]),.clk(clk),.din(Rdata2),.dout(mem_data),.w
e(MemWrite));
inst in1(.addr(pc[8:0]),.clk(clk),.dout(instruction));



control
c1(instruction[31:26],RegDst,ALUsrcB,MemToReg,WriteReg,MemWrite,B
ranch,ALUop1,ALUop0,Jump);

single_pc_plus_4 k1(pc,pc_plus_4);
single_pc k2(exec_out,rst_out,jump_or_not,pc);

branch_addr b1(pc_plus_4,exten_branch,branch_address);

mux_5 m1(instruction[20:16],instruction[15:11],rt_or_rd,RegDst);
assign exten_branch = {{16{instruction[15]}},instruction[15:0]};
assign equal_branch = zero & Branch;
mux_32 m2(pc_plus_4,branch_address,branch_or_next,equal_branch);
assign exten_jump = {{6{instruction[25]}},instruction[25:0]};
mux_32 m3(branch_or_next,exten_jump,jump_or_not,Jump);
mux_32 m4(Rdata2,exten_branch,addr_or_data,ALUsrcB);
mux_32 m5(result,mem_data,result_or_data,MemToReg);

assign ALUop = {ALUop1,ALUop0};
aluc a1(ALUop,instruction[5:0],func);
alu  a2(Rdata1,addr_or_data,func,zero,result);


single_gpr
g1(exec_out,rst_out,WriteReg,instruction[25:21],instruction[20:16
],reg_index,rt_or_rd,result_or_data,Rdata1,Rdata2,Rdata3);



always@(posedge exec_out,posedge rst_out) begin
if(rst_out) clock = 0;
else clock = clock + 1;
end
display d1(clk,disp_num,anode,segment);

always @* begin
case(disp_type)
2'b00:
disp_num<= Rdata3[15:0];
2'b01:
```

```
disp_num<= Rdata3[31:16];
2'b10:
disp_num<= pc[15:0];
2'b11:
disp_num<= clock[15:0];
endcase
end

endmodule
```

顶层模块变量较多，下面将一一分析其作用。

首先，exec 和 clk，前者是单周期 CPU 中的一个周期，利用按钮实现，按下按钮并放开意味着一个时钟周期的结束。而 clk 主要用于 display 模块的分频，利用开发板上的 T9 时钟实现。但是，clk 还出现在了两个 IP 核中：

```
Data
mem1(.addr(result[8:0]),.clk(clk),.din(Rdata2),.dout(mem_data),.we(MemWrite));
inst in1(.addr(pc[8:0]),.clk(clk),.dout(instruction));
```

主要原因是在数据通路的设计中，这两个模块是不需要时钟周期进行控制的，否则会出错，但是 IP 核的设计中却要求一个 clk.因此，利用一直在不断进行周期变化并且频率极高的 T9 时钟来代替。

其中 result 为 ALU 计算结果，这里计算的结果代表地址，Rdata2 为寄存器输入的第二个地址读取的数据，这里作为写入数据，mem_data 为输出数据。Pc 为当前 Pc 值，instruction 代表当前读取到的指令。

而 reg_index 为输入的寄存器下标，并输出相应寄存器中的值。

Disp_type 变量用两个开关实现，用以选择相应的数码管显示数据。选择代码：

```
always @* begin
case(disp_type)
2'b00:
disp_num<= Rdata3[15:0];
2'b01:
disp_num<= Rdata3[31:16];
2'b10:
disp_num<= pc[15:0];
2'b11:
disp_num<= clock[15:0];
endcase
end

endmodule
```

在 4 种情况下分别输出选择寄存器数据（Rdata3）的第 16 位，高 16 位，当前 pc 值，以及已经经历的时钟数。

Led[0]用以表示是否已经开始执行指令操作。

```
assign led[0] = (pc!=32'hFFFFFFFF);
```

其中 32’hFFFFFFFF 为 pc 初值，在开始运行后加一变为 0。

其余 5 个 led 灯分别代表 5 种指令类型，指令类型可以通过指令前 6 位来获得。

例如：`assign led[2] = (instruction[31:26]==35&led[0]);`

用前 6 位是否等于 35 和是否开始执行来判断代表 lw 指令的 led 灯是否要亮。

在将 instruction 指令前 6 位输入 control 模块就可以获得相应控制信号。各控制信号作用不多做赘述。

根据数据通路，在执行过程中需要很多的控制信号去选择不同的数据进行输入或者输出。

首先，利用 5 位多路选择器选择目的寄存器 rt_or_rd

```
mux_5 m1(instruction[20:16],instruction[15:11],rt_or_rd,RegDst);
```

同理，再用 32 位选择器选择出 branch_or_next（跳转指令或者 pc+1 的值），jump_or_not(跳转或不跳转的下一 pc 值)，addr_or_data(ALU 第二个操作输入数，代表地址运算或者计算)，result_or_data(寄存器的写入数据，为存储器中读取数据或者 ALU 运算结果)

最后，在寄存器模块中，

```
single_gpr
g1(exec_out,rst_out,WriteReg,instruction[25:21],instruction[20:16
],reg_index,rt_or_rd,result_or_data,Rdata1,Rdata2,Rdata3);
```

将 reg_index 输入，并输出 Rdata3 用以验收。输入 rt_or_rd 作为目的寄存器，result_or_data 作为写入数据。

至此，数据通路全部实现。

UCF 文件代码如下：

```
NET"clk" LOC="T9";

NET"exec" LOC="M13";
```

```
NET "anode[0]" LOC="D14";
NET "anode[1]" LOC="G14";
NET "anode[2]" LOC="F14";
NET "anode[3]" LOC="E13";

NET "segment[0]" LOC="E14";
NET "segment[1]" LOC="G13";
NET "segment[2]" LOC="N15";
NET "segment[3]" LOC="P15";
NET "segment[4]" LOC="R16";
NET "segment[5]" LOC="F13";
NET "segment[6]" LOC="N16";
NET "segment[7]" LOC="P16";



NET"rst" LOC="M14";
NET"led[0]" LOC="K12";
NET"led[1]" LOC="P14";
NET"led[2]" LOC="L12";
NET"led[3]" LOC="N14";
NET"led[4]" LOC="P13";
NET"led[5]" LOC="N12";


NET"disp_type[0]" LOC="F12";
NET"disp_type[1]" LOC="G12";

NET"reg_index[0]" LOC="H14";
NET"reg_index[1]" LOC="H13";
NET"reg_index[2]" LOC="J14";
NET"reg_index[3]" LOC="J13";
NET"reg_index[4]" LOC="K14";

NET "rst" CLOCK_DEDICATED_ROUTE = FALSE;
NET "exec" CLOCK_DEDICATED_ROUTE = FALSE;
```

指令 IP 核内容：

```
MEMORY_INITIALIZATION_RADIX  = 2;
MEMORY_INITIALIZATION_VECTOR =
10001100000000010000000000000000,
10001100000000010000000000000100,
10001100000001000000000000000001,
00000000001000100001100000100000,
00000000010000110001000000100000,
00000000011000010001100000100000,
00010000011001000000000000000001,
00001000000000000000000000000100,
10101100000000100000000000000011,
00001000000000000000000000000000;
```

存储器 IP 核内容：

```
MEMORY_INITIALIZATION_RADIX = 2;
MEMORY_INITIALIZATION_VECTOR =
00000000000000000000000000000001,
00000000000000000000000000001011,
00000000000000000000000000000000,
00000000000000000000000000000000,
00000000000000000000000000000000,
00000000000000000000000000000000,
00000000000000000000000000000000,
```

```
00000000000000000000000000000000,
00000000000000000000000000000000,
00000000000000000000000000000000,
00000000000000000000000000000000,
00000000000000000000000000000000,
00000000000000000000000000000000;
```

三、 实验过程和数据记录

本次实验中指令存储器中存储的具体代码为：

编好相应代码后，下载到实验板进行验证，结果如下(寄存器输出高位均为 0，下表略去)，和预期相符。

指令	clk_cnt	pc	R1	R2	R3	R4
LW $1, 0($0)	1	0	1	0	0	0
LW $2, 4($0)	2	1	1	0	0	0
LW $4, 1($0)	3	2	1	0	0	11
ADD $3, $1, $2	4	3	1	0	1	11
~ADD $2, $2, $3	5	4	1	1	1	11
ADD $3, $3, $1	6	5	1	1	2	11
BEG $3, $4, OUT	7	6	1	1	2	11
J REP	8	7	1	1	2	11
~ADD $2, $2, $3	9	4	1	3	2	11
ADD $3, $3, $1	10	5	1	3	3	11
BEG $3, $4, OUT	11	6	1	3	3	11
J REP	12	7	1	3	3	11
……	……	……	……	……	……	……
~ADD $2, $2, $3	41	4	1	55	10	11
ADD $3, $3, $1	42	5	1	55	11	11
BEG $3, $4, OUT	43	6	1	55	11	11
SW $2, 3($0)	44	8	1	55	11	11

上表中，~表示又一次循环开始之处

具体功能为实现从 1 到 10 的所有整数的相加，通过循环实现，每次的加数存储在 3 号寄存器中，和存储在 2 号寄存器中，并在最后将结果存储到存储器的 3 号地址中.

下载后，将从右到左第二个开关上拨，显示 PC 值为 FFFF，此时 LED 灯全灭，尚未开始执行指令。

时钟在每次按下按钮后自动加 1，只在重置按钮按下后归 0，这里不多做赘述。

按下执行按钮后 PC 变为 0.此时已经取得第 0 条指令的指令代码。

将开关拨至 0 后，选择 1 号寄存器，

放开按钮后发现 1 号寄存器中数据由 0 变为 1（用于加法）.

接下来的 2 条指令同理在 2 号寄存器中写入 0（用作初始化），在 4 号寄存器中写入 11（用作跳转）

再次按下按钮，执行第四条指令，同样的方法，发现在按钮松开后 3 号寄存器被写入 1.

接下来将进入循环除最后一次跳出循环按 4 次按钮，前面每次循环都是 5 下按钮一循环。

每次进入循环后将 3 号寄存器的值加到 2 号寄存器中并写入 2 号寄存器，再将 3 号寄存器加 1，用 beq 语句判断 3 号寄存器中的数字是否已经到达 11 并跳转。

因此，便有第 5 下按钮松开后 2 号寄存器中被写入 1，第 6 下 3 号被写入 2，第 7 下比较结果发现不需要跳出，第 8 下按下后跳转回第 4 条语句，pc 值重新被置为 4.

接下来进行的 10 个循环与一开始的结果类似，除了 2 号寄存器的值不断变化和每次 3 号寄存器的值加 1，其他的结果不变。而每次进入循环后第一条指令执行后 2 号寄存器的值分别为 3,6,a,f,15,1c,24,2d,37(16 进制)

最后一次跳出后，pc 值被置为 8，执行 sw 指令将 2 号寄存器的结果存入存储器中的 3 号地址，最后一下按钮按下后跳转回第 0 条指令重新执行所有指令

四、 实验结果

本次实验由于实验通路比较复杂，过程还是比较艰辛的。首先，对于 IP 核中的一些变量不是很了解，结果中的时钟使用 T9 时钟违背了单周期时钟 CPU 的初衷。

而在实验过程中经常会出现一些意想不到的情况和错误，这种时候，通过不断的猜测和实验检验来检查错误是需要花费很多时间的，硬件电路的代码检查难度明显高于其他编程的调试。

甚至可以在这次实验的过程中发现很多以前实验的模块存在很多致命错误，后来一一改正。但是一个很小的错误都是很难发现的，需要通过层层模块的检验来获得。

最后在模块连接完全正确后检查通路连接是否正确的过程中，由于变量太多，会出现变量名写错出现新的变量名，导致结果出错。由于未定义的变量不会报错，而是被默认定义为 1 位的 wire 型变量，而写错的变量名和愿变量名很相似，所以十分难查处错误原因。

最后，又要考虑到时钟周期以及上升沿下降沿对于整个通路的影响。对此，我采用将代码下载到开发板上，并根据对应指令的运行结果，一步步执行并查看每一步的结果，每次都是按下按钮后不放，观察结果，并在放开后再次检查结果，慢慢的一步步进行调试并完善，最后完成了实验。

五、 讨论与心得

本次实验让我第一次接触到了稍微复杂的数据通路，并将以往的一些模块和课堂知识真正整合并运用到了实践中，做出了具有一定功能的单周期 CPU。这个过程中的无数错误让我意识到了以前的不足，并对编程中可能遇到的一些问题有了一定的了解，而整个过程中的一些新思路，也让我对将来设计大型数据通路有了一定的基础知识准备。

更可贵的是，这次实验基本上难点就在于硬件电路的连接问题，设计过程中涉及到的行为描述很少。这就对设计者在编程前的通路设计和模块功能了解程度有了很高的要求，因为一点点错误就将导致最后结果的崩溃。

总而言之，这次的收获很大，相信会为接下来的多周期 CPU 设计等实验打下坚固的基础。