计算机体系结构实验

实验6报告书

作者：李珉超

学号：515030910361

2017.4.2

1 实验介绍

1.1 实验名称

简单的类MIPS多周期流水化处理器实现

1.2 实验目的

1.理解CPU的Pipeline，对Data Hazard，Branch Hazard有初步的认识。

1.3 实验范围

1.ISE 13.4 的使用

2.Xilinx Spartan 3E 实验板的使用

3.使用VerilogHDL进行逻辑设计

4.仿真测试，下载验证

2 实验过程

2.1 4级寄存器

2.1.1 实验原理

由于pipeline要求五个流水阶段是并行的，所以需要更多的寄存器保存当前指令，在该阶段的信息。 par IF/ID 保存从instruction memory根据PC读取的指令。由于在下一个周期，又有新的指令被读取，所以IF/ID必须保存当前的32位指令。这样在instruction decode阶段，读取的就是IF/ID中的指令。另外还要保存PC+4的结果。

ID/EX 保存control unit输出的信号(作为接下来的环节的信号)，从register读取的两个32 位数据，sign extend 的结果(作为EX阶段的ALU的可能操作数)，instruction中rt，rd的数据(作为可能的写回地址)，以及PC+4(在branch指令中，需要在EX阶段加上sign extend≪2)

EX/MEM 保存与MEM和WriteBack相关的信号(这样才能保证信号是当前指令的decode结果，否则信号将会是下一条指令的decode结果)，PC+4+sign extend≪2的结果(branch地址)，ALU结果(可能写入data memory的数据)，zero信号(作为beq的信号之一)，readData2的结果(作为可能的data memory 的地址)，writeBack的地址。

MEM/WB 保存从data memory读取的数据，ALU的结果(是可能的写回的数据)，writeBack地址(可能写回的地址)，WB相关的信号(道理同EX/MEM中的信号保存的原因)。

在实验指导书中，硬件图中没有实现jump指令的data path，但是我在自己实现的时候实现了。

2.1.2 IF/ID

```
module IR(
    input clk,
    input [31:0] instruction_in,
    input [31:0] PC_in,
    output reg [31:0] instruction_out,
    output reg [31:0] PC_out,
    output reg [31:0] jump_address
    );

    always @ (negedge clk)
```

```
    begin
        instruction_out = instruction_in;
        PC_out = PC_in;
        jump_address = ((instruction_in ≪ 2) & 8'h0fffffff) + (PC_in & 8'hf0000000);
    end
endmodule
```

2.1.3 ID/EX

```
module RR(
    input clk,
    input [31:0] readData1_in,
    input [31:0] readData2_in,
    input [31:0] sign_extend_in,
    input [31:0] PC_add_in,
    input [31:0] instruction,
    input [31:0] jump_address_in,
    input regDst_in,
    input aluSrc_in,
    input memToReg_in,
    input regWrite_in,
    input memRead_in,
    input memWrite_in,
    input branch_in,
    input [1:0] aluOp_in,
    input jump_in,
    output reg [31:0] readData1_out,
    output reg [31:0] readData2_out,
    output reg [31:0] sign_extend_out,
    output reg [31:0] PC_add_out,
    output reg [31:0] jump_address_out,
    output reg [4:0] write_address1,
```

```
    output reg [4:0] write_address2,
    output reg regDst_out,
    output reg aluSrc_out,
    output reg memToReg_out,
    output reg regWrite_out,
    output reg memRead_out,
    output reg memWrite_out,
    output reg branch_out,
    output reg [1:0] aluOp_out,
    output reg jump_out,
    );

    always @ (negedge clk)
    begin
        readData1_out = readData1_in;
        readData2_out = readData2_in;
        sign_extend_out = sign_extend_in;
        PC_add_out = PC_add_in;
        jump_address_out = jump_address_in;
        write_address1 = instruction[20:16];
        write_address2 = instruction[15:11];
        regDst_out = regDst_in;
        aluSrc_out = aluSrc_in;
        memToReg_out = memToReg_in;
        regWrite_out = regWrite_in;
        memRead_out = memRead_in;
        memWrite_out = memWrite_in;
        branch_out = branch_in;
        aluOp_out = aluOp_in;
        jump_out = jump_in;
    end
endmodule
```

2.1.4 EX/MEM

```
module AR(
    input clk,
    input rst,
    input jump,
    input branch,
    input zero,
    input [31:0] jump_address,
    input [31:0] PC_add,
    input [31:0] sign_extend,
    input [31:0] ALU_result_in,
    input [31:0] readData2_in,
    input [4:0] write_address_in,
    input memRead_in,
    input memWrite_in,
    input RegWrite_in,
    input MemtoReg_in,
    output reg PCSrc,
    output reg [31:0] branch_jump,
    output reg [31:0] ALU_result_out,
    output reg [31:0] readData2_out,
    output reg [4:0] write_address_out,
    output reg memRead_out,
    output reg memWrite_out,
    output reg RegWrite_out,
    output reg MemtoReg_out
    );

    always @ (negedge rst) PCSrc = 0;

    always @ (negedge clk)
    begin
```

```
        if(jump == 1 || (branch == 1 && zero == 1)) PCSrc = 1;
        else PCSrc = 0;
        branch_jump = jump ? (jump_address) : ((sign_extend ≪ 2) + PC_add);
        ALU_result_out = ALU_result_in;
        readData2_out = readData2_in;
        write_address_out = write_address_in;
        memRead_out = memRead_in;
        memWrite_out = memWrite_in;
        RegWrite_out = RegWrite_in;
        MemtoReg_out = MemtoReg_in;
    end
endmodule
```

2.1.5 MEM/WB

```
module MR(
    input clk,
    input [31:0] ALU_result_in,
    input [31:0] readData_in,
    input RegWrite_in,
    input MemtoReg_in,
    input [4:0] write_address_in,
    output reg [31:0] ALU_result_out,
    output reg [31:0] readData_out,
    output reg RegWrite_out,
    output reg MemtoReg_out,
    output reg [4:0] write_address_out
    );

    always @ (negedge clk)
    begin
        ALU_result_out = ALU_result_in;
```

```
            readData_out = readData_in;
            RegWrite_out = RegWrite_in;
            MemtoReg_out = MemtoReg_in;
            write_address_out = write_address_in;
        end
    endmodule
```

2.2 Top

2.2.1 实验原理

在顶层模块中，需要将这4级寄存器与之前的单周期时候的模块连接起来。需要在top 模块中，声明许多wire变量。

之后的代码过程与单周期的实现一样。

顶层模块只有两个输入信号，clock和reset。

2.2.2 实验模块

由于代码冗长，且只需要连接各个模块，故展示部分代码。

```
module top(
    input clk,
    input rst
    );
    wire PCSRC;
    wire [31:0] BRANCH_JUMP;
    wire [31:0] INSTRUCTION_ADDRESS; wire [31:0] INSTRUCTION_IN; wire [31:0] INSTRUCTION_OUT;
    wire [31:0] PC_TO_IR; wire [31:0] PC_TO_RR; wire [31:0] PC_TO_AR;
    wire [31:0] JUMP_ADDRESS_TO_RR; wire [31:0] JUMP_ADDRESS_TO_AR;
    wire [4 :0] WRITE_REG;
    wire [4 :0] WRITE_ADDRESS1; wire [4 :0] WRITE_ADDRESS2; wire [4 :0] WRITE_ADDRESS_TO_MR;
    wire [31:0] REG_READ_DATA1; wire [31:0] REG_READ_DATA2; wire [31:0] REG_READ_DATA1_To_ALU;
```

```
    wire [31:0] REG_READ_DATA2_TO_ALU; wire [31:0] REG_READ_DATA2_TO_DATAMEM;
    wire [31:0] SIGN_EXTEND_TO_RR; wire [31:0] SIGN_EXTEND_TO_ALU;
    wire [31:0] ALURES; wire [31:0] ALURES_TO_AR; wire [31:0] ALURES_TO_DATAMEM;
    wire [31:0] MEM_READ_DATA_TO_MR; wire [31:0] MEM_READ_DATA;
    wire REGWRITE, REGWRITE_TO_RR, REGWRITE_TO_AR, REGWRITE_TO_MR;
    wire REGDST, REGDST_TO_RR;
    wire ALUSRC, ALUSRC_TO_RR;
    wire MEMTOREG, MEMTOREG_TO_RR, MEMTOREG_TO_AR, MEMTOREG_TO_MR;
    wire MEMREAD, MEMREAD_TO_RR, MEMREAD_TO_AR;
    wire MEMWRITE, MEMWRITE_TO_RR, MEMWRITE_TO_AR;
    wire [1:0] ALUOP; wire [1:0] ALUOP_TO_RR;
    wire BRANCH_TO_RR, BRANCH_TO_AR;
    wire JUMP_TO_RR, JUMP_TO_AR;
    wire ZERO, ZERO_TO_AR;
    wire [3:0] ALUCTR;
    //connect the module
    //...
endmodule
```

2.2.3 仿真

先初始化clk=0, rst=1

然后将rst设置为0，以初始化一些模块(register, instruction memory, PC, Control unit) 的初始数据。

注意，data memory的初始化是通过readmemh("mem_data.txt", memFile)实现的。

```
module test_for_top;
    // Inputs
    reg clk;
    reg rst;

    // Instantiate the Unit Under Test (UUT)
    top uut (
```

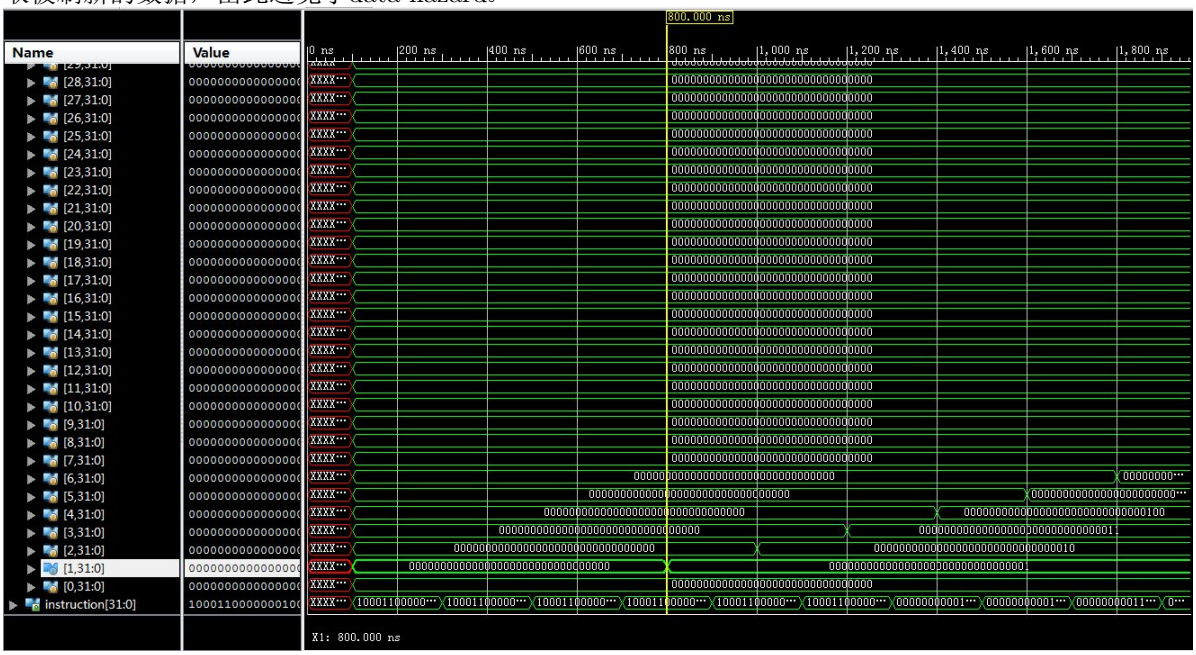
```
        .clk(clk),
        .rst(rst)
        );

        always #(100) clk = ~ clk;
        initial begin
            // Initialize Inputs
            clk = 0;
            reset = 1;

            // Wait 100 ns for global reset to finish
            #100;
            rst = 0;

        end
    endmodule
```

2.2.4 仿真图

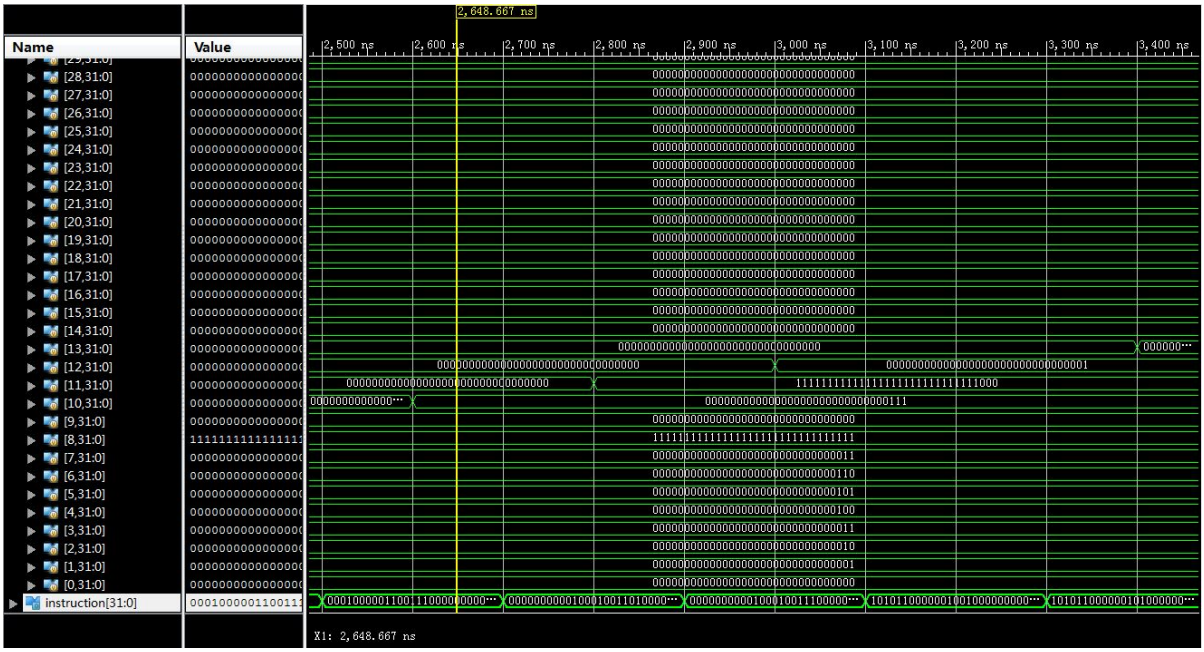
```
always @(negedge rst)
begin
   instruction_mem[0]  = 32'b10001100000000010000000000000001;// load mem[1] to reg[1]
   instruction_mem[1]  = 32'b10001100000000100000000000000010;// load mem[2] to reg[2]
   instruction_mem[2]  = 32'b10001100000000110000000000000011;// load mem[3] to reg[3]
   instruction_mem[3]  = 32'b10001100000001000000000000000100;// load mem[4] to reg[4]
   instruction_mem[4]  = 32'b10001100000001010000000000000101;// load mem[5] to reg[5]
   instruction_mem[5]  = 32'b10001100000001100000000000000110;// load mem[6] to reg[6]
   instruction_mem[6]  = 32'b00000000001000100011100000100000;// $7  = $1 + $2
   instruction_mem[7]  = 32'b00000000001000100100000000100010;// $8  = $1 - $2
   instruction_mem[8]  = 32'b00000000011001000100100000100100;// $9  = $3 & $4
   instruction_mem[9]  = 32'b00000000011001000101000000100101;// $10 = $3 | $4
   instruction_mem[10] = 32'b00000000101001100101100000100111;// $11 = $5 nor $6
   instruction_mem[11] = 32'b00000000101001100110000000101010;// $12 = $5 slt $6
   instruction_mem[12] = 32'b00010000011001110000000000000110;// jump if $7 == $3, 6 instruction
   instruction_mem[13] = 32'b00000000001000100110100000100000;// $13 = $1 + $2
   instruction_mem[14] = 32'b00000000001000100111000000100010;// $14 = $1 - $2
   instruction_mem[15] = 32'b00000000011001000111100000100100;// $15 = $3 & $4
   instruction_mem[16] = 32'b00000000011001001000000000100101;// $16 = $3 | $4
   instruction_mem[17] = 32'b10101100000010000000000000010000;// store $8  to mem[16]
   instruction_mem[18] = 32'b10101100000010010000000000010001;// store $9  to mem[17], to here!
   instruction_mem[19] = 32'b10101100000010100000000000010010;// store $10 to mem[18]
   instruction_mem[20] = 32'b10101100000010110000000000010011;// store $11 to mem[19]
   instruction_mem[21] = 32'b10101100000011000000000000010100;// store $12 to mem[20]
end
```

instruction memory在100ns的时候被初始化，以上是我初始化的内容。

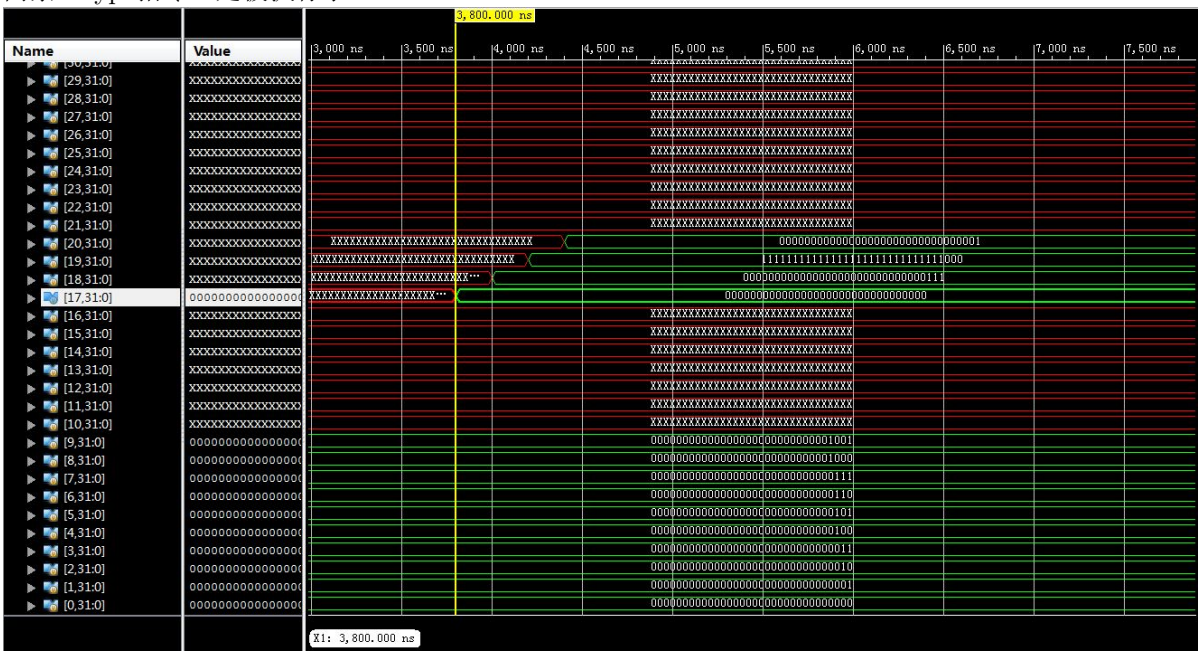
注意到由于之后对寄存器2的使用(R-type)在lw指令之后的四个周期后，所以在刚好要读取寄存器2的周期内，寄存器2的数据在前半周期的下降沿被刷新，然后在后半周期内即可正确读取被刷新的数据，由此避免了data hazard。



在100ns的时候初始化寄存器为0，之后是6条lw指令。可以看到，在从IF/ID取出指令后的第四个时钟周期时，数据被写回寄存器，这符合pipeline的实现过程。



本图中黄线位置的指令是一条beq指令。结合之前给出的预设指令，可以发现，beq 之后的4条R-typr指令只读取了两条。这是因为branch指令所给出的地址在读取指令后的2个周期，写入PC。并且跳过的指令条数也是正确的。当然在这里，本实验中，没有实现flush功能，所以那两条R-type指令还是被执行了。



对于最后的sw指令，我们从图中看到，数据被写入mem[17]到mem[20]，而mem[16]没有被写入。对比之前的仿真图，看到相应寄存器的值确实读入到了mem中。这既说明了sw 指令的正确性，同时又从另一个方面说明了branch指令的正确性。

这些仿真结果表明，多周期类MIPS流水化CPU的实现成功。

3 实验心得

由于有了之前单周期的实现经历，在语法问题上克服了不知道怎么用verilog实现的障碍。在多周期的实现中，由于有更多的模块，所以需要更多地wire变量，所以需要在命名上制定了Source_TO_Module 的命名方式，所以不至于混乱。

在其他方面难度不大，因为对各个模块的理解已经比较深刻，所以各个模块稍作调整即可使用。

在亲自实验后，真正理解到pipeline的实现原理，当然也认识到具体商业化的产品中，有更多的难题需要克服。我们实现的是最简单的pipeline处理器，真正实践中，必须要考虑control hazard，floating number computation 等问题。但至少，对于整个结构框架有了认识。这门课程对我学习计算机体系结构的帮助很大。

课程结束，很感谢那些曾经帮助过我的同学和老师，希望大家共同进步。