Project2　MIPS 流水线 CPU

2019302733 魏天昊

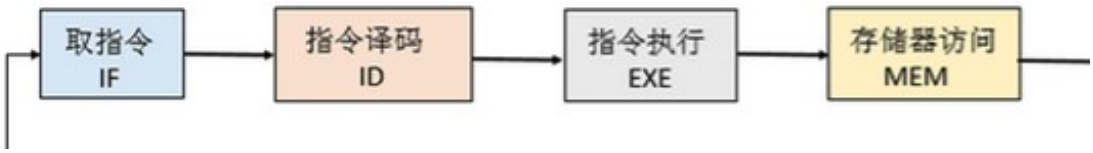
实验要求

1. 设计实现 42 条 mips 指令的流水线 CPU。
2. 处理数据冒险与控制冒险。
3. 处理溢出异常。

详细设计

1. 设计分析

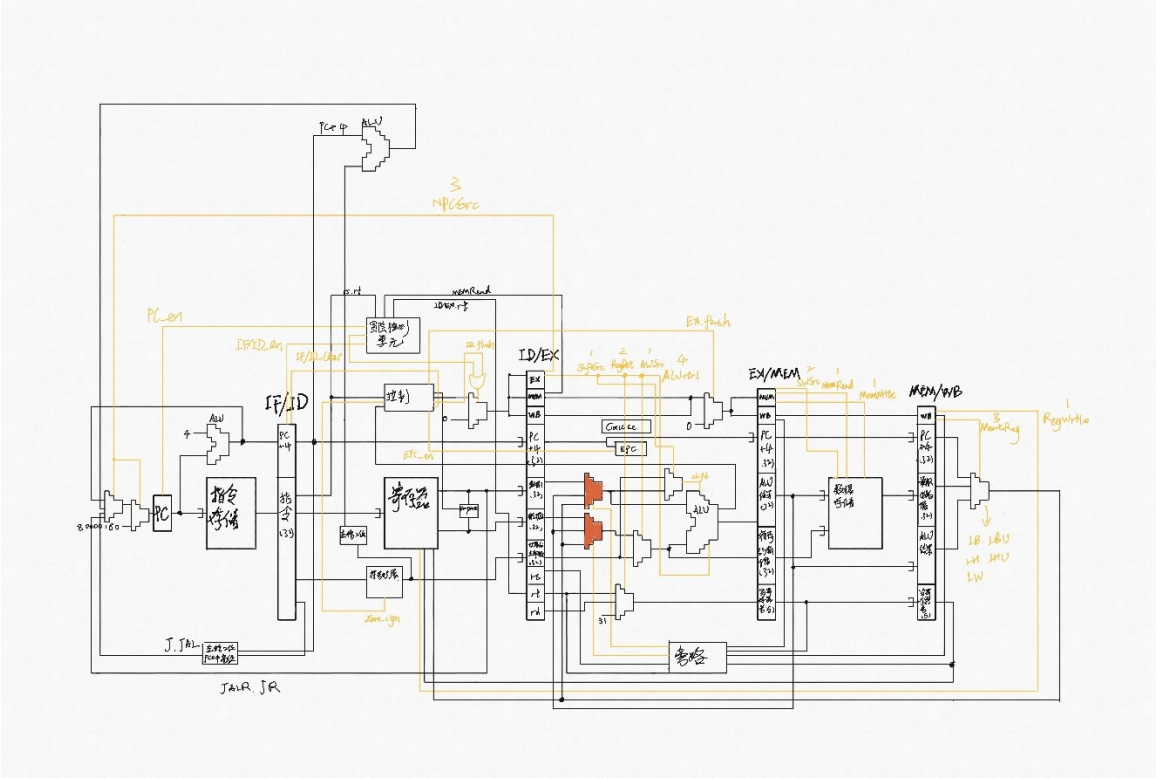
流水线 CPU 是一种利用指令间并行技术的方法，提高了系统稳定性和工作速度。根据 MIPS 处理器特点，处理过程被分为 5 个阶段：



在每个时钟周期，都有多条指令在其中运行，指令的一系列数据与控制信号将通过寄存器来存储和逐级传递。

流水线 CPU 的难点在于处理冒险。首先是数据冒险，对于一条指令需要的数据还没有写回时，就发生了数据冒险。有些冒险可以通过旁路来解决，如 R-R 型。而 LW-R 型冒险就需要阻塞一个时钟周期。具体实现上，需要将 PC 写使能和 IFID 寄存器写使能置为 0，将 ID_flush 置为 1。其次是控制冒险，对于 branch 和 jump 指令来说，他们需要改变下一个 PC 的值，而之一过程需要在 ID 级才能完成，因此发生冒险。默认情况下将选择 PC+4，当分支发生时，就需要清除上一条取到的指令，具体来说，就是将 IFID 寄存器同步清零信号置为 1。发生异常也是控制冒险的一种，在这种情况下，需要将 IFID 清零信号置为 1，将 ID_flush 和 EX_flush 置为 1，还要将地址转移到异常处理程序的地址处。

2. 数据通路



3. 重要模块实现

3.1 forward 旁路模块

```
always@(IDEX_rs,IDEX_rt,EXMEM_rw,EXMEM_RegWrite,MEMWB_rw,MEMWB_RegWrite)
begin
  if(EXMEM_RegWrite == 1&&EXMEM_rw != 0&&EXMEM_rw == IDEX_rs)
    forward_a = 2'b01;
  else if(MEMWB_RegWrite == 1&&MEMWB_rw != 0&&MEMWB_rw == IDEX_rs)
    forward_a = 2'b10;
  else
    forward_a = 2'b00;

  if(EXMEM_RegWrite == 1&&EXMEM_rw != 0&&EXMEM_rw == IDEX_rt)
    forward_b = 2'b01;
  else if(MEMWB_RegWrite == 1&&MEMWB_rw != 0&&MEMWB_rw == IDEX_rt)
    forward_b = 2'b10;
  else
    forward_b = 2'b00;
end
```

3.2 hzd 冒险检测模块

```
always@(IFID_rs,IFID_rt,IDEX_rt,IDEX_MEMRead)
  if(IDEX_MEMRead == 1&&IDEX_rt != 0&&(IDEX_rt == IFID_rs||IDEX_rt == IFID_rt))
    begin
      PC_en = 1'b0;
      IFID_en = 1'b0;
      ID_flush = 1'b1;
    end
  else
    begin
      PC_en = 1'b1;
      IFID_en = 1'b1;
      ID_flush = 1'b0;
    end
```

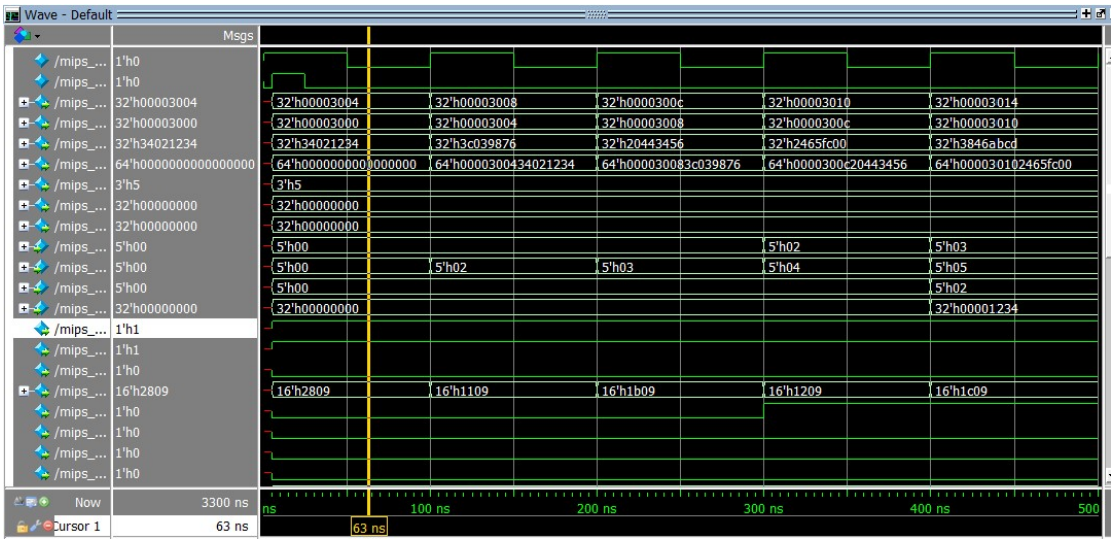
3.3 rf 模块

```
module rf(r1_addr,r2_addr,w_addr,w_data,r1_data,r2_data,clk,RegWrite);

  input [25:21] r1_addr;
  input [20:16] r2_addr;
  input [15:11] w_addr;
  input [31:0] w_data;
  input clk,RegWrite;
  output [31:0] r1_data;
  output [31:0] r2_data;

  reg [31:0] rmem[31:0];

assign r1_data = rmem[r1_addr];

assign r2_data = rmem[r2_addr];

always@(negedge clk)
  if(RegWrite == 1'b1)
    rmem[w_addr] = w_data;
  else
    rmem[w_addr] = rmem[w_addr];

endmodule
```

3.4 ctrl 模块

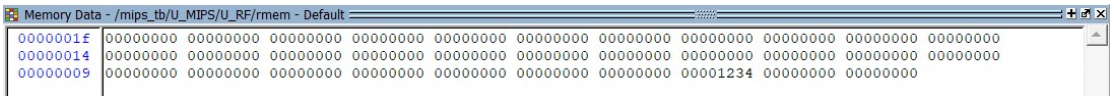
```
always@(instr)
  case(instr[31:26])
    6'b00_0000:
    begin
      zero_sign = 1'b0;
      case(instr[5:0])
        6'b10_0000:ctrlop = 16'b0_01_0_0010_00_0_0_1_001;  //add
        6'b10_0001:ctrlop = 16'b0_01_0_1100_00_0_0_1_001;  //addu
        6'b10_0010:ctrlop = 16'b0_01_0_0011_00_0_0_1_001;  //sub
        6'b10_0011:ctrlop = 16'b0_01_0_1101_00_0_0_1_001;  //subu
        6'b10_1010:ctrlop = 16'b0_01_0_0110_00_0_0_1_001;  //slt
        6'b10_1011:ctrlop = 16'b0_01_0_0111_00_0_0_1_001;  //sltu
        6'b00_0000:ctrlop = 16'b0_01_0_1000_00_0_0_1_001;  //sll
        6'b00_0010:ctrlop = 16'b0_01_0_1001_00_0_0_1_001;  //srl
        6'b00_0011:ctrlop = 16'b0_01_0_1010_00_0_0_1_001;  //sra
        6'b00_0100:ctrlop = 16'b1_01_0_1000_00_0_0_1_001;  //sllv
        6'b00_0110:ctrlop = 16'b1_01_0_1001_00_0_0_1_001;  //srlv
        6'b00_0111:ctrlop = 16'b1_01_0_1010_00_0_0_1_001;  //srav
        6'b10_0100:ctrlop = 16'b0_01_0_0000_00_0_0_1_001;  //and
        6'b10_0101:ctrlop = 16'b0_01_0_0001_00_0_0_1_001;  //or
        6'b10_0110:ctrlop = 16'b0_01_0_0100_00_0_0_1_001;  //xor
        6'b10_0111:ctrlop = 16'b0_01_0_0101_00_0_0_1_001;  //nor
        6'b00_1000:ctrlop = 16'b0_00_0_0000_00_0_0_0_000;  //jr
        6'b00_1001:ctrlop = 16'b0_01_0_0000_00_0_0_1_000;  //jalr
        default:ctrlop = 16'b0;
      endcase
    end
    6'b00_1000:  //addi
    begin
      zero_sign = 1'b1;
      ctrlop = 16'b0_00_1_0010_00_0_0_1_001;
    end
```

仿真验证

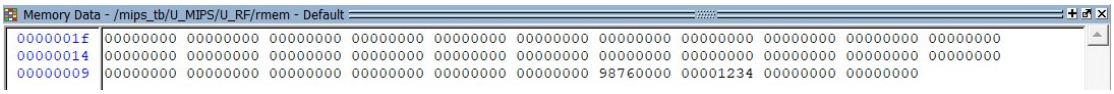


以下展示部分指令分析：

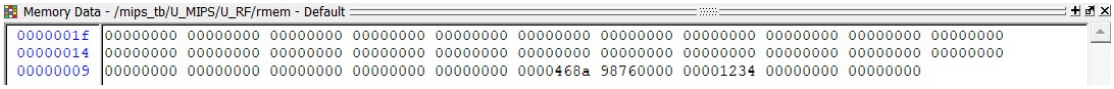
1. Ori $2 $0 0x00001234
   在第五个周期将（寄存器 0 和立即数）的或运算的结果写回寄存器 2。



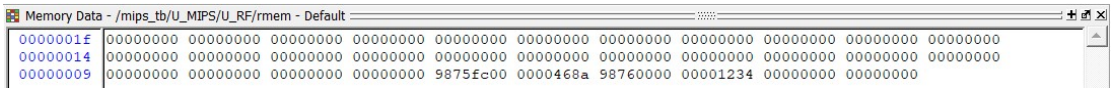
2. Lui $3 0x00009876
   在第六个周期将（加载至高位的立即数）写入寄存器 3。



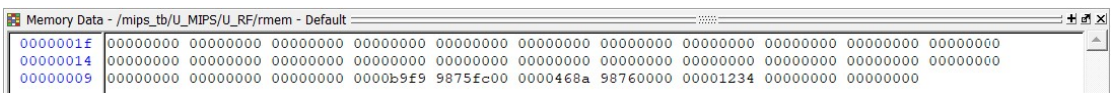
3. Addi $4 $2 0x00003456
   在第七个周期将（寄存器 2 和立即数）的和运算的结果写回寄存器 4。
   0x00001234+0x00003456 = 0x0000468a



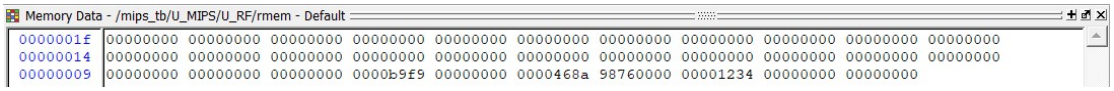
4. Addiu $5 $3 0xfffffc00
   在第八个周期将（寄存器 3 和立即数）的无符号和运算的结果写回寄存器 5。
   0x98760000+0xfffffc00 = 0x9875fc00



5. Xori $6 $2 0x0000abcd
   在第九个周期将（寄存器 2 和立即数）的异或的结果写回寄存器 6。
   0x00001234^0x0000abcd = 0x0000b9f9



6. Sltiu $5 $4 0x00000034
   在第十个周期将（寄存器 4 和立即数）的无符号小于置位的结果写回寄存器 5。
   0||0x0000468a < 0||0x00000034 = 0x00000000



疑难分析

1. 处理读写寄存器堆的数据冒险（同级数据冒险），写寄存器发生在时钟周期的前半段，读寄存器发生在时钟周期的后半段。为了能够达到这一目的，将写寄存器设置为时钟下降沿触发。

```
17  always@(negedge clk)
18    if(RegWrite == 1'b1)
19      rmem[w_addr] = w_data;
20    else
21      rmem[w_addr] = rmem[w_addr];
```

2. 在出现 LW 类指令后面紧跟 SW 指令的情况时，会发生 MEM 级数据冒险，但因为 SW 的数据需求也发生在 MEM 级，可以使用旁路来避免一次阻塞。
3. 在 ID 级的用于分支（如 beq，bgtz）的比较中，可能发生数据冒险，需要采用相应的旁路与阻塞。
4. 在 ID 级的用于 jr，jalr 指令的寄存器中的地址值可能发生数据冒险，需要采用相应的旁路与阻塞。
5. 在写 verilog 代码时，发现对于 PC 的更新，其 PC+4 的来源若是 IFID 寄存器，则会慢一个时钟周期，应该直接从 PC 值加 4 得到。
6. Slt 指令在仿真时出现错误，data1 < data2 ? 1 : 0 并不能得到想要的结果。因为 verilog 中的<是无符号比较。需要在 data1 和 data2 的声明中加入 signed。

```
2  module alu(data1,data2,shift,aluctrl,result,overflow);
3
4    input signed [31:0] data1;
5    input signed [31:0] data2;
```

   同理，在 compare 模块中也是如此。

```
2  module compare(data1,data2,result);
3
4    input signed [31:0] data1;
5    input signed [31:0] data2;
```

7. 对于 IFID_clear 信号，不能和 rst 一样设置为异步清零，可能会产生毛刺将寄存器中的值直接清零，这个后果是灾难性的。所以要设置为同步清零。在 ID 级检测到控制冒险就会将其置 1，下一个时钟上升沿时 IFID 寄存器将变为 0，也就产生了一个气泡。
8. 在书写代码时发现数据通路有一处错误，EXMEM 寄存器中待存数据与立即数无关，应与前一个多路选择器的输出相连。