

实验报告

1. 实验目标

完成多周期流水线CPU的设计，需要支持MIPS-C3={ LB, LBU, LH, LHU, LW, SB, SH, SW, ADD, ADDU, SUB, SUBU, MULT, MULTU, DIV, DIVU, SLL, SRL, SRA, SLLV, SRLV, SRAV, AND, OR, XOR, NOR, ADDI, ADDIU, ANDI, ORI, XORI, LUI, SLT, SLTI, SLTIU, SLTU, BEQ, BNE, BLEZ, BGTZ, BLTZ, BGEZ, J, JAL, JALR, JR, MFHI, MFLO, MTHI, MTLO } 总共50条指令。

2. 设计思路

总共使用了五级标准流水线：取指令（IF, instruction fetch）、指令译码（ID, Instruction Decode）执行（EX, Execute），访存（MEM, Memory）、写回（WB, WriteBack），在每两级流水线之间，都加入流水线寄存器，因此总共有IF/ID, ID/EX, EX/MEM, MEM/WB四层流水线寄存器。

采用了分布式控制器架构，具体来说使用了2-控制架构，有一个主控制器Control，负责指令的译码和一部分多选器控制；还有一个冒险控制器Hazard，专门负责检测相关的指令，控制转发有关的多选器以及插入气泡进行阻塞。

为了便于对指令之间的依赖关系进行梳理和检测，我对所有MIPS-C3指令按照彼此之间的冲突关系进行分类，最终分成8个类别：

- 广义R型指令，泛指那些需要利用ALU计算出结果，并将结果送入到寄存器值中的指令。不单单包括狭义的R型指令，也包含一部分I型指令，如ori, lui等。
- load型指令，从存储器中加载数据的指令，包括lw, lh, lhu, lb, lbu
- store型指令，往存储器中存储数据的指令，包括sw, sb, sh
- branch指令，进行分支判断的指令，包括beq, bne, bgt, bge, blt, ble
- jal指令
- jr指令
- jalr指令
- 乘除法指令，进行乘除法运算或者将结果送往hi和lo寄存器的有关指令，包括MULT, MULTU, DIV, DIVU, MTHI, MTLO, MFHI, MFLO

这8个类别会使用3个二进制位来记录，在ID译码阶段由主控单元得到，沿着流水线一级一级传递，供冒险控制单元进行检测和判断使用。

为了控制各种多选器和ALU运算，在CPU中需要使用到非常多的信号量。在本次实验中我所定义和使用的全部信号量如下：

- PCSrc, 决定PC的来源，用于多选，PC的来源可能是PCAdd4, PCBranch, PCJump, 寄存器等。
- RegWriteSrc, 决定WB阶段使用哪一个数据写到寄存器中，可能是ALU计算结果，DM读出来的数据，PC+8
- RegDst, 决定WB阶段往哪一个寄存器写，可能是rt、rd，第三十一号寄存器\$ra等
- ALUOperation, 决定ALU执行何种运算
- ALUSrc, 决定ALU的第二个操作数的来源，可能是寄存器rt，也可能是经过拓展的立即数
- ALUMD, 决定使用ALU的结果还是使用MD的结果作为EX阶段的最终结果
- CMPOperation, 决定CMP模块判定跳转成立的条件
- word, 决定了内存读写操作的单位，可能是四字节（一个字），两字节（半个字），以及单个字节
- ReadSign, 决定从DM之中读出来的数据是否需要符号拓展，或者是在EX阶段决定应当对16位的立即数进行零拓展还是符号拓展。

- Type, 冒险指令类型, 上述的8类之一。

所有的旁路转发如下:

1. 从MEM送到EX。这条旁路从EX/MEM流水线寄存器出发送往EX阶段选择ALU的运算数来源的多选器, 可以解决的问题是连续两条R型指令, 并且前一条R型指令的结果是后一条R型指令的操作数, 前一指令的结果来不及写回的情况, 可以概括为R+R。
2. 从WB送到MEM。这条旁路从WB阶段决定写入寄存器的结果来源的多选器出发, 将写入到寄存器的最终结果送到EX阶段决定ALU运算数来源的多选器。这是为了解决R+无关+R的情况, 也就是R型指令的结果被隔一条的下一条指令所使用, 这种情况下会需要用到这条旁路。
3. 从MEM送到ID。这条旁路从EX/MEM流水线寄存器送出, 送到比较单元CMP之前的多选器之中。这是为了解决R型指令后面跟随着一个Branch类分支指令, 并且分支指令所判断的两个寄存器其中一个来自于上一条R型指令的结果。
4. 从WB送到MEM。这条旁路从WB阶段决定写入寄存器的结果来源的多选器出发, 将结果送到MEM阶段决定MEM写入数据来源的多选器。这条旁路是为了解决R型指令后面紧接着一条store 类型的指令, 并且store指令往存储器中写入的值恰好是上一条R型指令运算的结果

有些指令之间的数据依赖仅仅依靠旁路是没有办法解决的, 在某一些情况下必须使用阻塞插入气泡来解决。所谓气泡, 就是指插入一条nop指令, 让流水线空一个周期以等待某一些结果写入到寄存器之中。在我的实现之中, 所有的气泡全部都是插在EX阶段, 而让上一条指令阻塞在ID阶段。旗袍插入实现的机理是:

- 设置PCFrozen=1, 让PC保持原来的值而不再加四。
- 设置IF/IDFrozen=1, 让IF/ID流水线寄存器冻结不再更新, 将下一条指令保存在此
- 设置ID/EXClear=1, 让ID/EX流水线清零, 相当是下一个周期在EX阶段执行的就是nop指令

插入气泡会导致整个流水线废掉一个周期, 极端情况下, 如果每条指令都插入气泡, 就会退化成单周期流水线。因此应当尽量地减少气泡的插入, 以尽可能地提高流水线效率。经过我的梳理, 所有需要插入气泡的情况总共有以下几种:

1. load+R, 注意这里的R型指令是我自己界定的广义的R型指令。如果R型指令需要使用load从存储器中取出来的值作为自己的操作数, 就需要添加一条气泡, 使得当R型指令位于EX阶段时, load指令已经进入了WB阶段, 可以将load出来的结果直接送到EX阶段。
2. R+beq, R型指令的结果会被用作beq进行判断的依据。由于R型指令需要EX阶段结束之后才可以得到结果, 而branch类型的分支指令要求在ID阶段就要得到结果。所以, 需要插入一个气泡, 将beq推迟一个周期, 就能通过之前所说的从EX到ID的旁路, 得到EX执行的结果了。
3. load+beq, 这中指令组合最为特殊, 因为load在MEM阶段结束时才可以得到存储器中的结果, 而分支类型的指令需要在ID阶段结束时得到结果, 因此, 插入一个气泡是不够的, 这种情况会需要插入两个气泡
4. R+jalr/jr, 如果R型指令的计算结果所放的寄存器恰好是存储jalr或者jr要跳转的地址的寄存器, 产生的指令依赖情况同上述第二条非常相似, 这种情况下也需要插入一个气泡
5. load+jalr/jr, 如果load型指令的计算结果所存放的寄存器恰好就是jalr或者jr要跳转的地址寄存器, 那么这种情况下同上述第三条非常相似, 因为在我的设计下, jalr和jr同样也是需要在ID阶段就计算出下一条指令的PC。所以这种情况也会需要插入两个气泡。
6. MDbusy, 如果乘除单元目前正在执行乘法或者除法, 需要5个周期或者10个周期的延迟, 因此在这种情况下通过MDbusy位来标识。如果MDBusy=1, 就插入气泡。

3. 实现步骤

3.1 MultiDiv

乘除单元总共需要处理8种, 因此只需要3个二进制位就可以表达出这8种指令的类型。其中乘法和除法都需要分有符号和无符号两种情况, 我事先算出来有符号的结果和无符号的结果, 然后使用case语句, 跟据具体的八种指令进行具体的操作。

对于模拟延迟的实现，我使用一个计数器counter，来记录当前周期是进行完乘法或者除法操作之后的第几个周期，需不需要设置busy位以阻塞EX单元。进行模拟阻塞的代码逻辑如下：

```
always@(posedge CLK)
begin
    if(start && !MDBusy)
    begin
        MDBusy=1;
        if(MDOperation==5'b00101 || MDOperation==5'b00110)
        begin
            count=5;
        end
        if(MDOperation==5'b00111 || MDOperation==5'b01000)
        begin
            count=10;
        end
    end
    if(count)
        count=count-1;
    if(!count)
        MDBusy=0;
end
```

3.2 BitEnable

位有效单元位于MEM阶段DM单元之前，需要进行判定，如果要想正确地读取4字节、2字节或者一个单独字节，需要对地址进行修正，成为四字节对齐的形式，从而可以作为一个下标而找到对应的寄存器。还需要记录，这个寄存器之中总共有32位四个字节，应该读哪几个字节。这一部分的关键代码如下所示：

```
always @(*)
begin
    if(!word)
    begin
        writeBit=4'b1111;
    end
    if(word[0]) //half word
    begin
        if(LowAddr[1])
        begin
            writeBit=4'b1100;
        end
        else
        begin
            writeBit=4'b0011;
        end
    end
    if(word[1]) //one byte
    begin
        if(LowAddr==2'b00)
            writeBit=4'b0001;
        if(LowAddr==2'b01)
            writeBit=4'b0010;
        if(LowAddr==2'b10)
            writeBit=4'b0100;
        if(LowAddr==2'b11)
```

```

        writeBit=4'b1000;
    end
end

```

3.3 preprocess

预处理单元位于WB阶段，它将从MEM阶段DM之中读出来的32位4字节数据进行进行处理，选取其中需要的字节并进行填充，最终得到cooked加工完成的数据。这一部分的关键代码如下：

```

always @ (*)
begin
    i=0;
    if(ReadBit[0])
    begin
        cooked[i+:8]=raw[7:0];
        SignBit=cooked[i+7];
        i=i+8;
    end
    if(ReadBit[1])
    begin
        cooked[i+:8]=raw[15:8];
        SignBit=cooked[i+7];
        i=i+8;
    end
    if(ReadBit[2])
    begin
        cooked[i+:8]=raw[23:16];
        SignBit=cooked[i+7];
        i=i+8;
    end
    if(ReadBit[3])
    begin
        cooked[i+:8]=raw[31:24];
        SignBit=cooked[i+7];
        i=i+8;
    end
    for(j=i;j<32;j=j+1)
    begin
        if(ReadSign)
            cooked[j]=SignBit;
        else
            cooked[j]=0;
    end
end
end

```

3.4 DM

数据存储器和之前相比，一个重要的改变是需要适应不同的store指令，需要考虑32位写入数据的哪几个字节需要写入到对应32位寄存器的哪几个字节。我使用类似于计数器的形式结合BitEnable单元得到的WriteBit来处理，主要代码如下：

```

assign dout=dm[addr];

always@(posedge clk)begin
    if(we)
    begin

```

```

j=0;
if(writeBit[0])
begin
    dm[addr][7:0]=din[j+ :8];
    j=j+8;
end
if(writeBit[1])
begin
    dm[addr][15:8]=din[j+ :8];
    j=j+8;
end
if(writeBit[2])
begin
    dm[addr][23:16]=din[j+ :8];
    j=j+8;
end
if(writeBit[3])
    dm[addr][31:24]=din[j+ :8];
end
end;

```

3.5 Compare

compare比较单元接在ID阶段，接受两个寄存器堆从rs，rt两个寄存器之中读出来的值，比较他们是否满足跳转条件，进行跳转，实现方面主要是一个case语句：

```

always@(CMPOperand1 or CMPOperand2 or CMPOperation)
begin
    case(CMPOperation)
        3'b000:result=(CMPOperand1==CMPOperand2);
        3'b001:result=(CMPOperand1!=CMPOperand2);
        3'b010:result=($signed(CMPOperand1)>0);
        3'b011:result=($signed(CMPOperand1)>=0);
        4'b100:result=($signed(CMPOperand1)<0);
        4'b101:result=($signed(CMPOperand1)<=0);
    endcase
end

```

3.6 Hazard

冒险控制单元是控制流水线进行转发和冒险的控制单元，所处理的旁路转发类型和阻塞类型已经在前面由详述，这里不再重复，主要接受来自各个流水线寄存器的控制信号以确定确定一些多路选择的控制信号。冒险控制单元的代码实现如下所示：

```

module
Hazard(CLK,EXrs,EXrt,EXType,EXRegWrite,EXWriteReg,IDType,IDrt,IDrs,MEMWriteReg,M
EMRegWrite,MEMType,MEMrt,WBWriteReg,WBRegWrite,MDBusy,

ForwardID1,ForwardID2,ForwardEX1,ForwardEX2,ForwardMEM,PCFrozen,IFIDFrozen,IDEX
Clear);
    input CLK;
    input [4:0] IDrs,IDrt,EXrs,EXrt,EXWriteReg,MEMrt,MEMWriteReg,WBWriteReg;
    input [2:0] IDType,EXType,MEMType;
    input EXRegWrite,MEMRegWrite,WBRegWrite,MDBusy;
    output reg[1:0] ForwardEX1,ForwardEX2,ForwardID1,ForwardID2;
    output reg ForwardMEM;

```

```

output reg PCFrozen, IFIDFrozen, IDEXClear;

integer count;
initial begin
    PCFrozen=0;
    IFIDFrozen=0;
    IDEXClear=0;
    ForwardEX1=0;
    ForwardEX2=0;
    ForwardID1=0;
    ForwardID2=0;
    ForwardMEM=0;
    count=0;
end
always @(*)
begin
    PCFrozen=0;
    IFIDFrozen=0;
    IDEXClear=0;
    ForwardEX1=0;
    ForwardEX2=0;
    ForwardID1=0;
    ForwardID2=0;
    ForwardMEM=0;
    // type1
    if(EXrs==MEMwriteReg && MEMwriteReg && MEMRegWrite)
        ForwardEX1=2'b01;
    else if(EXrs==WBwriteReg && WBwriteReg && WBRegWrite)
        ForwardEX1=2'b10;
    if(EXrt==MEMwriteReg && MEMwriteReg && MEMRegWrite)
        ForwardEX2=2'b01;
    else if(EXrt==WBwriteReg && WBwriteReg && WBRegWrite)
        ForwardEX2=2'b10;

    //type2
    lw          lw          R          ori
    lw          sw          jr
    if(EXTYPE==3'b001 && (IDType==3'b000 || IDType==3'b000 || IDType==3'b001
|| IDType==3'b010 || IDType==3'b100))
    begin
        if(IDrt==EXrt && IDType==3'b000)//R
        begin
            PCFrozen=1;
            IFIDFrozen=1;
            IDEXClear=1;
            $display("type2 stall");
        end
        if(IDrs==EXrt)
        begin
            PCFrozen=1;
            IFIDFrozen=1;
            IDEXClear=1;
            $display("type2 stall");
        end
    end
end

//type3 branch
if(IDType==3'b111 && (EXwriteReg==IDrs || EXwriteReg ==IDrt) &&
EXwriteReg && EXRegWrite)

```

```

begin
    PCFrozen=1;
    IFIDFrozen=1;
    IDEXClear=1;
    $display("type3 stall1");
end//
if(IDType==3'b111 && MEMType==3'b001 && (IDrs==MEMrt || IDrt==MEMrt))
begin
    PCFrozen=1;
    IFIDFrozen=1;
    IDEXClear=1;
    $display("type3 stall2");
end
else if(IDType==3'b111 && (MEMWriteReg==IDrs || MEMWriteReg ==IDrt)&&
MEMWriteReg && MEMRegWrite)
begin
    if(MEMWriteReg==IDrs)
        ForwardID1=2'b01;
    if(MEMWriteReg ==IDrt)
        ForwardID2=2'b01;
end
if(IDType==3'b111 && (WBWriteReg==IDrs || WBWriteReg==IDrt) &&
WBWriteReg && WBRegWrite)
begin
    if(WBWriteReg==IDrs)
        ForwardID1=2'b10;
    if(WBWriteReg==IDrt)
        ForwardID2=2'b10;
end

/*
if(IDType==3'b111 && (WBWriteReg==IDrs || WBWriteReg==IDrt) &&
WBWriteReg && WBRegWrite)
begin
    PCFrozen=1;
    IFIDFrozen=1;
    IDEXClear=1;
    $display("type3 stall3,count=%h",count);
    count=count+1;
end
*/
//type4 jr & jalr
if((IDType==3'b100 || IDType==3'b101) && EXWriteReg==IDrs && EXWriteReg
&& EXRegWrite)
begin
    PCFrozen=1;
    IFIDFrozen=1;
    IDEXClear=1;
    $display("type4 stall1");
end
if((IDType==3'b100 || IDType==3'b101) && MEMWriteReg==IDrs &&
MEMWriteReg && MEMRegWrite)
begin
    PCFrozen=1;
    IFIDFrozen=1;
    IDEXClear=1;
    $display("type4 stall2");
end
end

```

```

        if((IDType==3'b100 || IDType==3'b101) && WBWriteReg==IDrs && WBWriteReg
&& WBRegWrite)
            begin
                PCFrozen=1;
                IFIDFrozen=1;
                IDEXClear=1;
                $display("type4 stall3,count=$h",count);
                count=count+1;
            end

//type5 sw
        if(MEMType==3'b010 && MEMrt==WBWriteReg && WBWriteReg && WBRegWrite)
            ForwardMEM=1;

//type6 muldiv
        if(MDBusy)
            begin
                PCFrozen=1;
                IFIDFrozen=1;
                IDEXClear=1;
                $display("type6 stall");
            end
        end
    end
endmodule

```

其余的部件单元与单周期10条指令CPU的实现是完全相同的，几乎没有改动，因此不再赘述

3.7 优化















多周期流水线之中，可以进行优化的一个关键地方是branch类型指令。由于保证所有的跳转和分支指令都具有延迟槽（紧随跳转分支指令后面的一条nop语句），所以如果branch判定条件不成立，NextPC的值可以直接设置为PC+8，而不是PC+4，因为nop语句无论执行与否都不影响结果的正确性。通过这种方式，可以减少运行所需要的周期数。















此外，如果加入一条从WB阶段送回到ID阶段的旁路，使得Branch指令可以在写回周期一开始时，就立即得到正确的判定结果，也会有一定的优化作用，可以减少汇编语句执行所需要的周期数。最终的结果，我的CPU执行test42测试程序总共需要2076个周期即可完成。

4 运行结果

4.1 test42

如下图所示是执行test42测试文件后，数据存储器前28个值

>  [27][31:0]	00b7ffe9	Array
>  [26][31:0]	010effab	Array
>  [25][31:0]	ff97009b	Array
>  [24][31:0]	002200fd	Array
>  [23][31:0]	00f6fff6	Array
>  [22][31:0]	ff97009b	Array
>  [21][31:0]	00b7ffe9	Array
>  [20][31:0]	00cf0048	Array
>  [19][31:0]	ffc00015	Array
>  [18][31:0]	002200fd	Array
>  [17][31:0]	fef1ff97	Array
>  [16][31:0]	ffc00014	Array
>  [15][31:0]	00b7ffe9	Array
>  [14][31:0]	01dd0145	Array

>  [13][31:0]	010e0069	Array
>  [12][31:0]	0022007d	Array
>  [11][31:0]	00b80069	Array
>  [10][31:0]	ffb5000a	Array
>  [9][31:0]	00cf00dc	Array
>  [8][31:0]	006d0073	Array
>  [7][31:0]	018701ff	Array
>  [6][31:0]	018701fe	Array
>  [5][31:0]	00b800ff	Array
>  [4][31:0]	00cf00ff	Array
>  [3][31:0]	00620069	Array
>  [2][31:0]	00170073	Array
>  [1][31:0]	00560000	Array
>  [0][31:0]	017000e8	Array















与Mars执行的结果进行对比，Mars执行完成后的内存如下所示：

Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)	Value (+14)	Value (+18)	Value (+1c)
0x00000000	0x017000e8	0x00560000	0x00170073	0x00620069	0x00cf00ff	0x00b800ff	0x018701fe	0x018701ff
0x00000020	0x006d0073	0x00cf00dc	0xffb5000a	0x00b80069	0x0022007d	0x010e0069	0x01dd0145	0x00b7ffe9
0x00000040	0xffc00014	0xfef1ff97	0x002200fd	0xffc00015	0x00cf0048	0x00b7ffe9	0xff97009b	0x00f6fff6
0x00000060	0x002200fd	0xff97009b	0x010effab	0x00b7ffe9	0xff90019f	0xfea6ffaf	0x002200fd	0xff90019f
0x00000080	0x0177ff0c	0x00b7ffe9	0xff3703b1	0x0141ffec	0x002200fd	0xff3703b1	0x01e7fd6d	0x00b7ffe9
0x000000a0	0xffde084a	0xfd4dff2c	0x002200fd	0xffde084b	0x02b0f9b0	0x00b7ffe9	0x00121094	0x025c00d4
0x000000c0	0x002200fd	0x00121095	0x024d2f0d	0x00b7ffe9	0x003b219b	0xfd8cfab9	0x002200fd	0x003b219b
0x000000e0	0x02c0e034	0x00b7ffe9	0x01e6441e	0x0103005f	0x002200fd	0x01e6441f	0x0285be95	0x00b7ffe9
0x00000100	0x03e388af	0xfe55ff2e	0x002200fd	0x03e388af	0x009f79fe	0x00b7ffe9	0x081d115e	0x00c400d2

经过对比可以发现结果啊完全相同，说明执行过程是正确的。

4.2 multdiv

如下图所示是我的CPU执行完multdiv测试文件之后的效果：

>  [13][31:0]	00000011	Array
>  [12][31:0]	000008ce	Array
>  [11][31:0]	00164bbf	Array
>  [10][31:0]	c070ff8d	Array
>  [9][31:0]	0000b4b4	Array
>  [8][31:0]	0000b689	Array
>  [7][31:0]	c070ff8d	Array
>  [6][31:0]	00000000	Array
>  [5][31:0]	00170073	Array
>  [4][31:0]	6a40f811	Array
>  [3][31:0]	00000001	Array
>  [2][31:0]	00a10089	Array
>  [1][31:0]	35752f2b	Array
>  [0][31:0]	000008ce	Array

同样使用Mars进行模拟，得到的执行结果为：

Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)	Value (+14)	Value (+18)	Value (+1c)
0x00000000	0x000008ce	0x35752f2b	0x00a10089	0x00000001	0x6a40f811	0x00170073	0x00000000	0xc070ff8d
0x00000020	0x0000b689	0x0000b4b4	0xc070ff8d	0x00164bbf	0x000008ce	0x00000011	0x00000000	0x00000000

经过比对两者是完全相同的，证明乘除法指令的逻辑也没有问题。