计算机系统结构实验报告：实验六

bugenzhao@sjtu.edu.cn

July 2, 2020

目 录

1 实验概述

实验名称 类 MIPS 多周期流水线处理器设计与实现

实验目的

1. 完成多周期流水线的类 MIPS 处理器
2. 通过 Stall 和 Forward 解决流水线冒险
3. 通过 Predict-Not-Taken 进行分支预测，提高性能

实验成果

1. 实现了支持 **49 条指令**的多周期流水线 MIPS 处理器，在上次实验的基础上增加了对乘法运算的支持
2. 通过 Stall 和 Forward 解决了流水线冒险，通过 Predict-Not-Taken 进行分支预测
3. 在指令内存上实现了简单的直接映射 Cache，成功模拟了多周期取指
4. 在上次实验探索的基础上，进一步配置了 C 编译器，成功在我的 MIPS 多周期流水线处理器上运行了简单的 **C 语言程序**

2 实验设计

在本次实验中，我设计了一个支持 49 条指令的单周期 MIPS 处理器，如下表所示。其中灰色指令为编译器支持的伪指令，没有计入。

逻辑	AND	OR	XOR	NOR	ANDI	XORI	LUI	ORI	LI	
移位	SLL	SRL	SRA	SLLV	SRLV	SRAV	NOP			
算术	ADD	SUB	SLT	ADDI	SLTI	ADDU	SUBU	SLTU	ADDIU	SLTIU
跳转	JR	JALR	J	JAL						
分支	BEQ	B	BGTZ	BLEZ	BNE	BLTZ	BLTZAL	BGEZ	BGEZAL	BAL
内存	LB	LBU	LH	LHU	LW	SB	SH	SW		
乘法	MULT	MULTU	MFHI	MFLO						

相比于上次实验中单周期处理器的设计，为实现流水线、乘法和 Cache，我做了一定修改，主要有：

- 增加了四个内部阶段寄存器，用于阶段之间指令执行状态的转移，实现了 5 级流水线，即：Instruction Fetch, Instruction Decode, Execute, Memory Access, Write Back。
- 增加了 `Forward` 模块和 `Stall` 模块，用于流水线控制。

- 为支持乘法计算，将 RegisterFile 模块进行了扩展，增加了 HI、LO 寄存器和新的一个写入端口和写入信号。
- 增加了 Cache 模块，模拟了直接映射的 CPU 缓存，据此调整了指令内存的速度和带宽，为取指阶段增加了 ready 信号。

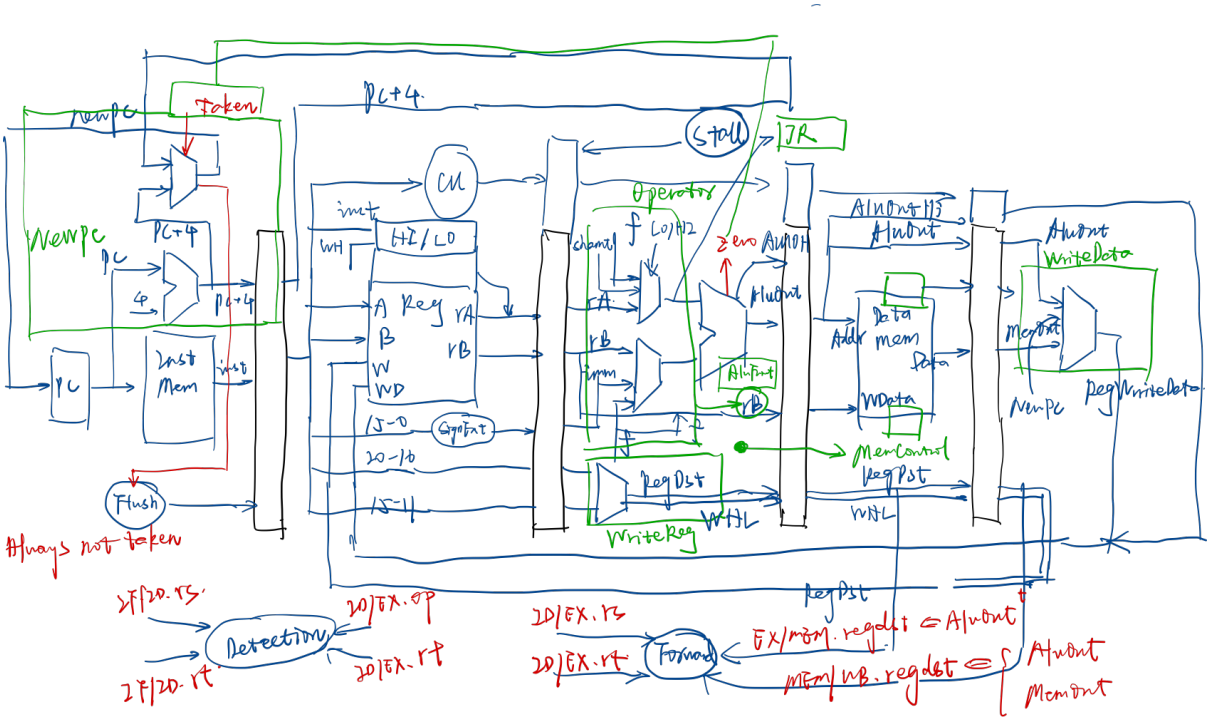

Figure 1: 多周期流水线处理器的电路设计草图

总体而言，本次实验我的电路设计草图如图 1 所示，其中绿色框为 Control Unit 拆分得到的各个小控制单元。下面我将对部分重要的模块进行解读。

2.1 乘法支持：RegisterFile、ALU 和 WriteReg 模块

设计 MIPS I ISA 支持的乘法指令是 MULT 和 MULTU，它们将得到的 64 位结果存入两个独立的 HI、LO 寄存器中，再通过后续的 MFHI（Move From Hi）和 MFLO 指令转移到通用寄存器。为方便起见，我将 HI、LO 寄存器与原先的 RegisterFile 模块整合到了一起，这就要求寄存器文件能够在一个周期内写入两个字，需要增加一个写入数据端口和控制端口。同样地，MFHI 指令从寄存器中读回 HI，这个过程是隐式的，需要根据 opcode 和 funct 加以判断。

根据我之前对于 ALU 功能控制信号的设计，由于乘法也属于 SPECIAL 指令，它的 funct 会自动被用作 ALU 功能信号，因此只需在 ALU 中加入一个新的运算功能即可。当然，由于输出结果是 64 位的，也需要加一条新的输出线路，命名为 outHi。

WriteReg 模块负责产生 writeLoHi 信号，发送给寄存器文件。

实现 RegisterFile 模块的关键代码如下所示。这里使用了嵌套三目运算符提升了可读性。

```
reg [`WORD] regFile[0:31];
reg [`WORD] lo, hi;

wire readHi = opcode == `OPC_SPECIAL && funct == `FUN_MFHI;
wire readLo = opcode == `OPC_SPECIAL && funct == `FUN_MFLO;

initial begin: init
...
end

always @(posedge clk) begin
    if (writeLoHi) begin
        lo <= writeData;
        hi <= writeDataHi;
    end
    else if (regWrite) begin
        regFile[writeReg] <= writeData;
    end
end

assign readData1 = readHi ? hi :
                   readLo ? lo :
                   readReg1 == 0 ? 0 : regFile[readReg1];

assign readData2 = readReg2 == 0 ? 0 : regFile[readReg2];
```

ALU 模块更新的部分代码如下所示。乘法相对而言并不复杂，Verilog 有内置的运算符；而对于除法等运算，使用内置运算符可能就无法综合了。

```
`FUN_MULTU:
    {outHi, out} = opA * opB;
`FUN_MULT:
    {outHi, out} = $signed(opA) * $signed(opB);
```

WriteReg 模块只需添加一行对于 writeLoHi 的控制信号。

```
writeLoHi = opcode == `OPC_SPECIAL && (funct == `FUN_MULT || funct ==
    `FUN_MULTU);
```

2.2 多周期内存：InstMemory 模块

设计 为实现 Cache 做好准备，我们首先要在 InstMemory 中模拟多周期内存访问，增加 ready 信号表示读取是否完成。同时，为了让 Cache 起到作用，我增加了指令内存的带宽到 128 位，允许一次性读取 4 条指令，并使用 Cache 中的术语将其称为 line。

基于此，我在内存内部增加了一个 lineAddrBuffer，保存上一次内存访问的 line 的起始地址。每次内存访问时，检查 buffer 中的地址是否与目前请求的 line 相同：若相同，则返回 line 对应的 128 位字，将 ready 设置为 1；若不同，则代表要进行一个新的访问，仅更新 buffer 中的值，不做读取，且将 ready 设置为 0。

将指令内存改为多周期后，CPU 内部的 NewPC 模块也要进行修改，根据 ready 信号采取不同的行为，后文中对此有详细介绍。

实现 InstMemory 模块的代码如下所示，其中 `QWORD 是一个在 ISA.v 中定义的新宏，代表 127:0 的总线宽度。

```
module InstMemory #(parameter textDump = "path/to/text/dump") (
           input clk,
           input wire [ `WORD] addr,
           output reg [`QWORD] qdata,
           output reg          ready
       );

parameter memSize = 'h1ffff;
reg [`WORD] memFile[0:memSize];

initial begin: init
    integer i;
    for (i = 0; i < memSize; i = i + 1) begin
        memFile[i] = 0;
    end
    $readmemh(textDump, memFile);
    ready = 0;
end

wire [29:0] firstIndex = addr >> 4 << 2;
reg  [29:0] lineAddrBuffer = 0;

always @(posedge clk) begin
    if (lineAddrBuffer == firstIndex) begin
        ready <= 1;
        qdata <= { memFile[firstIndex], memFile[firstIndex + 1], memFile[
    firstIndex + 2], memFile[firstIndex + 3] };
    end
```

```
28      else begin
29          lineAddrBuffer <= firstIndex;
30          ready <= 0;
31          qdata <= 128'h0;
32      end
33  end
34
35  endmodule
```

2.3 直接映射缓存：Cache 模块

设计 直接映射缓存根据地址中表示 index 的位直接确定对应的 Cache line，使用高位 tag 验证缓存是否匹配，再根据最低几位作为偏移确定 line 中对应的字。我为指令内存设计了一个每行 4 个字、64 行，总容量 256 字 =1KB 的直接映射缓存，考虑到 MIPS 指令对齐于 4 的倍数，地址结构如下所示。

tag	index	offset	00
31:10	9:4	3:2	1:0

Cache 扮演着 CPU 计算单元与外部指令内存的中间人角色，它对于 CPU 来说应该是透明的：Cache 模块接受 CPU 送来的地址，向 InstMemory 送去；接受 InstMemory 送来的 128 位字和 imReady 信号，经过处理、更新缓存后又向 CPU 送去。

由于 CPU 取指受到时序控制，因此 Cache 也并不一定需要时序进行控制。当收到 CPU 的请求时，它可能的行为有：

- 命中，直接返回对应的指令
- 未命中，但 InstMemory 已完成之前的相同请求，更新 Cache 并返回对应的指令
- 未命中，且是第一次请求，向 InstMemory 发送请求，并返回 ready 为 0

实现 Cache 模块的代码如下所示。对于更复杂的实现，可能需要添加时序来完成。

```
1  module Cache (
2              input wire [`WORD] pc,
3              output reg [`WORD] inst,
4              output reg         instReady,
5
6              output reg [ `WORD] imAddr,
7              input wire [`QWORD] imQdata,
8              input wire          imInstReady
9          );
10
```

```
`define ADDR_OFFSET   3:2
`define ADDR_INDEX    9:4
`define ADDR_TAG     31:10

reg [`WORD] blocks[0:63][0:3];
reg [21: 0] tags  [0:63];
reg         valid [0:63];

initial begin: init
    integer i, j;
    for (i = 0; i < 64; i = i + 1) begin
        tags[i] = 0;
        valid[i] = 0;
        for (j = 0; j < 4; j = j + 1) begin
            blocks[i][j] = 0;
        end
    end
    instReady = 0;
end

wire [1:0] offset = pc[`ADDR_OFFSET];
wire [5:0] index = pc[`ADDR_INDEX];
wire [24:0] tag  = pc[`ADDR_TAG];

reg [1:0] hit;

always @(pc, offset, index, tag, imInstReady, imQdata) begin
    if (tags[index] == tag && valid[index] == 1) begin
        instReady = 1;
        inst = blocks[index][offset];
        hit = 1;
    end
    else begin
        if (imInstReady == 1 && imAddr[31:4] == pc[31:4]) begin
            {blocks[index][0], blocks[index][1], blocks[index][2], blocks[
    index][3]} = imQdata;
            valid[index] = 1;
            tags[index] = tag;
            instReady = 1;
            inst = blocks[index][offset];
            hit = 2;
        end
        else begin
```

```
            imAddr = pc;
            valid[index] = 0;
            instReady = 0;
            inst = 32'h0;
            hit = 0;
        end
    end
end

endmodule // Cache
```

效果 通过添加多周期内存和 Cache，我们可以成功模拟出不同时长的取指访问，如图 2 所示。（为清晰起见，这里使用了 Scansion 工具展示波形）

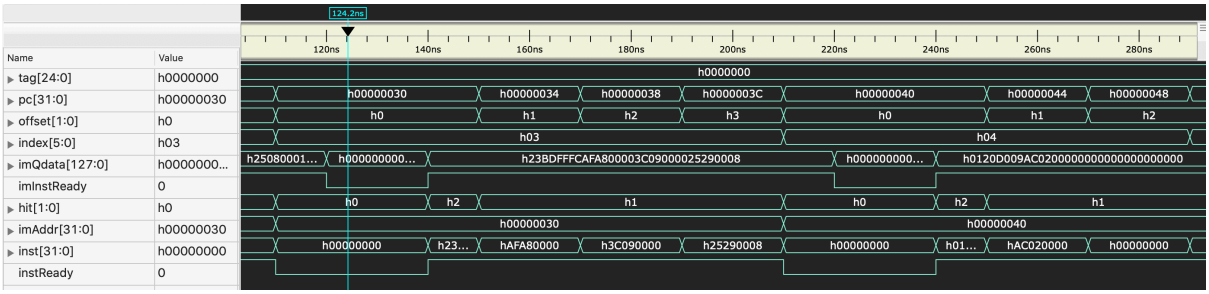

Figure 2: Cache 波形

2.4 阶段寄存器：StageRegID 等模块

设计 从单周期处理器转向流水线的第一步，是均等地将原先执行一条指令的过程划分为几个步骤，在此基础上允许多条指令同时利用多个阶段。阶段间需要暂存、转移相关的执行状态，需要一系列内部阶段寄存器。

阶段寄存器设计并不复杂，它接受上个阶段指令的相关状态，在每个指令周期开始时进行更新，将保存的状态输出给下一个阶段。而随着流水线控制信号 `flush` 和 `stall` 的加入，阶段寄存器也要遵循这些信号以实现气泡或暂停。这里以第一个阶段寄存器 StageRegID 模块为例，它需要同时处理两种信号：`flush` 为 1 时，插入指令字为 0（即 NOP）的气泡；`stall` 为 1 时，延续上一个周期的状态，即暂停；一般情况下，更新状态。

实现 该模块的关键代码如下所示。

```
module StageRegID(
            input clk,

            input wire [`WORD] ifNextPC, ifInstruction,
```

```
5           input wire          flush, stall,
6
7           output reg [`WORD] idNextPC, idInstruction
8       );
9
10 always @(negedge clk) begin
11     if (flush) begin
12         idNextPC <= ifNextPC;
13         idInstruction <= 0;
14     end
15     else if (stall) begin
16         idNextPC <= idNextPC;
17         idInstruction <= idInstruction;
18     end
19     else begin
20         idNextPC <= ifNextPC;
21         idInstruction <= ifInstruction;
22     end
23 end
24
25 endmodule // StageRegID
```

2.5 IF 阶段

2.5.1 NewPC 模块

设计 PC 寄存器与单周期的设计一致。而对于 NewPC 模块，其可选的来源依然有四种，但由于要实现流水线，这四个地址来源的阶段需要仔细斟酌。具体而言：

- nextAddr：紧接着的下一条指令，即 PC + 4，自然是当前 IF 阶段的 PC。
- branchAddr：分支目标，该地址来源的阶段取决于我们在哪个阶段得到分支结果。注意到，参考教材上只考虑 BEQ 和 BNE，因此可以直接在 ID 阶段比较取出的两个操作数是否相等而得到结果；如今我们支持更复杂的分支指令，我选择在 EX 阶段通过 ALU 得到分支结果。因此，该地址来源于 EX 阶段。
- jumpAddr：J 和 JAL 指令的跳转目标。由于跳转指令的目标直接在指令当中，我们完全可以直接在 IF 阶段进行移位计算，故该地址来源于 IF 阶段。
- jumpRegAddr：JR 和 JALR 指令的跳转目标，从寄存器读出、经过 ALU 做 +0 计算后得到。由于可能涉及到数据冒险和转发，因此，我们也只有等到 EX 阶段才能得到跳转目标，地址来源于 EX 阶段。

由于我采取 Predict-Not-Taken 预测策略，一旦 NewPC 决定使用 branchAddr，就意味着需要清除前面 2 条错误取指的指令的运行状态，因此我们可以让 NewPC 模块负责产生 flushID 和 flushEX 信号。另外，在处理加载-使用冒险时，需要暂停流水线的取指，NewPC 也需要根据 stall 信号作出反应，即保持 PC 值不变。

为支持多周期取指，NewPC 还需要根据 Cache 或指令内存反馈的 ready 信号调整行为：在 ready 为 0 时，我们应该将 nextAddr 保持为当前 PC。

实现 综上所述，NewPC 模块的关键代码如下所示。

```
module NewPC(
            input wire [`WORD] pc,
            input wire [`WORD] instruction,
            input wire         instReady,

            input wire [`WORD] jumpRegAddr,
            input wire         isJumpReg,
            input wire [`WORD] branchAddr,
            input wire         taken,
            input wire         stall,

            output reg [`WORD] newPC,
            output reg         flushID, flushEX
        );

wire [`OPC] opcode    = instruction[31:26];
wire [`FUN] funct     = instruction[ 5: 0];
wire [25:0] jumpIndex = instruction[25: 0];

wire [`WORD] nextAddr   = instReady ? pc + 4 : pc;
wire [`WORD] jumpAddr   = {nextAddr[31:28], (jumpIndex << 2)};

always @(*) begin
    flushID = 0;
    flushEX = 0;

    if (stall) begin
        newPC = pc;
        flushEX = 1;
    end
    else if (taken) begin
        newPC = branchAddr;
        flushID = 1;
        flushEX = 1;
```

```
35      end
36      else if (isJumpReg) begin
37          newPC = jumpRegAddr;
38          flushID = 1;
39          flushEX = 1;
40      end
41      else begin
42          case (opcode)
43              `OPC_J, `OPC_JAL:
44                  newPC = jumpAddr;
45              default:
46                  newPC = nextAddr;
47          endcase
48      end
49  end
50
51  endmodule
```

2.5.2 PC 模块

PC 模块可直接继承单周期版本。

2.6 ID 阶段

观察设计草图可以得知，ID 阶段中的两个模块 RegisterFile 和 SignExtend 没有因为流水线设计导致的修改。RegisterFile 为支持乘法指令进行的修改已在 2.2 节进行了分析。

2.7 EX 阶段

EX 阶段改动较大，我们需要在此阶段生成大量控制信号，还需要处理数据转发。

2.7.1 Operand 模块

设计 Operand 模块现在需要根据流水线控制模块 Forward 输出的信号决定是否采用转发的数据。为方便起见，我的设计中让 Forward 模块生成完全的转发信息，包括 rs、rt 是否转发，以及对应的转发数据。因此，Operand 模块只需根据是否转发信号 rsFwd 和 rtFwd，从输入值中选择正确的一个即可。

另一方面，回忆内存写入指令需要使用 rt 取出的值作为数据，这一值也应该得到正确的转发，输出在 writeToMemData 中。

实现 根据上述设计，该模块修改的内容很少，只需要将原先选取寄存器输出的地方改为：

```
opA = rsFwd ? rsFwdData : rsData;
opB = rtFwd ? rtFwdData : rtData;
```

另外，要转发 writeToMemData。

```
// memWriteData:
always @(*) begin
    writeToMemData = rtFwd ? rtFwdData : rtData;
end
```

2.7.2 WriteReg、MemControl 模块

这两个模块都是基本的控制模块，没有因为流水线进行的改动，为支持乘法运算进行的改动已在前文涉及。这里只是将它们移动到了 EX 阶段进行，输出结果会随着阶段寄存器传递下去。这样做的目的是为了使用 memRead 更加准确地分析各类数据冒险，从而决定是否要插入气泡或暂停流水线。

2.7.3 JumpReg、Taken 模块

JumpReg 模块的作用是判断指令是否为 JR 或 JALR，实现非常简单，是 NewPC 模块的输入之一。前文已经提到，由于 JR 和 JALR 指令需要在 EX 阶段才能得到跳转目标（ID 阶段不能接受转发），且设计已经决定在 EX 阶段检测分支结果，因此我也将 Taken 模块设计在 EX 阶段，统一处理分支和跳转。

2.7.4 ALU、ALUFunct 模块

这两个模块同样没有因为流水线进行的改动，为支持乘法运算进行的改动已在前文涉及。

2.8 MEM 阶段

观察设计草图可以得知，由于已经将 MemControl 移动至 EX 阶段，MEM 阶段中只有一个数据内存模块。回忆一下，在我单周期处理器的设计中，CPU 模块不包括指令内存和数据内存，它们的信号以 CPU 的输入、输出接口的形式给出。因此，MEM 阶段只需要几个简单的 assign 语句。

```
// --- MEM ---
assign dataAddress = memALUOut;
assign writeMemData = memWriteToMemData;
assign memRead = memMemRead;
assign memWrite = memMemWrite;
```

```
6 assign memMode = memMemMode;
```

2.9 WB 阶段

观察设计草图可以得知，WB 阶段唯一的模块 `WriteData` 没有因为流水线进行的改动。尽管添加了乘法运算，可能需要同时向寄存器写回 2 个 32 位字，但乘法的高位输出 `aluOutHi` 只可能用于写回到寄存器 `writeDataHi` 端口，这部分增改无需 `WriteData` 模块干涉。

2.10 流水线控制

2.10.1 流水线控制设计

本节将介绍该 MIPS 多周期流水线处理器对于各类流水线冒险的处理方案设计。

结构冒险 Harvard 架构的计算机将指令内存与数据内存分离，故在内存读写上不存在结构冒险。而对于寄存器而言，可能存在 ID 阶段和 WB 阶段同时访问的结构冒险。解决这一冒险的有力手段是让寄存器文件在前半周期处理写入，在后半周期处理读取。这种设计允许含有数据依赖的先后两条指令同时进行 WB 和 ID 阶段，一定程度上也缓解了数据冒险，提高了流水线处理器性能。

数据冒险 数据冒险可分为两类，对它们的处理方案如下：

1. 某一条指令是 ALU 指令，而后面的指令需要使用该指令的结果。

   解决该数据冒险的办法是进行转发。若两条指令相邻，则当二者运行到 MEM 和 EX 阶段时，可以直接将前一条指令的 ALU 运算结果从 MEM 阶段转发到 EX 阶段，用于后一条指令的计算；若两条指令中间间隔一个，则当二者运行到 WB 和 EX 阶段时，可以直接将前一条指令的 ALU 运算结果从 WB 阶段转发到 EX 阶段，用于后一条指令的计算；若两条指令间隔更大，使用上一节对于寄存器读写的处理方案，不再存在数据冒险。

2. 某一条指令是内存读取指令，而后面的指令需要使用该指令的结果，即加载-使用冒险。

   内存读取指令只有在 MEM 阶段结束才能够获得结果。若两条指令中间间隔一个，可以直接将前一条指令的内存读取运算结果从 WB 阶段转发到 EX 阶段，用于后一条指令的计算。而若两条指令相邻，我们只能在内存指令运行完 EX 阶段后，暂停流水线一个周期，等待该指令运行完 MEM 阶段后再继续，转化为之前讨论的情况。若两条指令间隔更大，使用上一节对于寄存器读写的处理方案，不再存在数据冒险。

控制冒险 控制冒险是由于分支和跳转指令不能及时处理导致的。在我们的设计中，所有不能及时处理的分支和跳转指令都是在 EX 阶段处理的。因此，可以采取 Predict-Not-Taken 策略，即 IF 阶段默认永远取下一条指令，在接收到 EX 阶段 Taken 和 JumpReg 模块汇报的预测错误信号后，采用新的地址取指即可。此时要注意清除接下来在 ID、EX 阶段的两条错误指令。

2.10.2 处理结构冒险：RegisterFile 模块

考虑到 CPU 各项阶段寄存器都是在时钟下降沿更新的，我们只需修改 RegisterFile 模块，让它在时钟上升沿进行更新，即可解决结构冒险。

```
always @(posedge clk) begin
    if (writeLoHi) begin
        lo <= writeData;
        hi <= writeDataHi;
    end
    else if (regWrite) begin
        regFile[writeReg] <= writeData;
    end
end
```

2.10.3 处理数据冒险：Forward 和 Stall 模块

Forward 模块 根据之前讨论的设计方案，可以写出 Forward 模块。Forward 模块收集大量 EX、MEM、WB 阶段的信息，负责完整地处理转发，最终输出要转发的数据 rsFwdData 和 rtFwdData，以及 rsFwd 和 rtFwd 信号指示 Operand 模块是否要选择转发的值。这样的设计可以让 Operand 模块更加清晰、简单。

由于我设计的 MIPS 流水线处理器支持近 50 条指令，Forward 模块相比于教材上的设计要增加更复杂的逻辑，主要修改有：

- 由于添加了 Link 类型的分支和跳转指令，转发内容还可能是即将写往 $ra 等寄存器的下一条指令的 PC 值。模块中实例化了一个 isLink 小模块来进行判断。
- 由于支持了乘法运算，还需要考虑 MULT 后跟 MFLO 或 MFHI 的数据冒险。

除此之外，转发应遵循“转发新数据”的原则：若 MEM 阶段和 WB 阶段的指令同时匹配，则优先选择 MEM 指令的数据。可以通过在顺序过程中先比较 WB 阶段、后比较 MEM 阶段的方式来实现，较新的数据会覆盖较老的匹配。

该模块由于比较重要，实现在下方全部给出。

```
module Forward(
           input wire [ `REG] memWriteReg, wbWriteReg,
           input wire         memMemRead, wbMemRead,
           input wire         memWriteLoHi, wbWriteLoHi,
```

```
5             input wire [`WORD] memALUOut, memALUOutHi,
6             input wire [`WORD] wbALUOut, wbALUOutHi, wbMemOut,
7             input wire [`WORD] memNextPC, wbNextPC,
8
9             input wire [`WORD] exInstruction, memInstruction, wbInstruction,
10
11            output reg         rsFwd, rtFwd,
12            output reg [`WORD] rsFwdData, rtFwdData
13        );
14
15 wire [`REG] rs = `GET_RS(exInstruction);
16 wire [`REG] rt = `GET_RT(exInstruction);
17
18 wire memIsLink, wbIsLink;
19 IsLink u_IsLink_mem(
20            .instruction (memInstruction),
21            .isLink      (memIsLink)
22        );
23 IsLink u_IsLink_wb(
24            .instruction (wbInstruction),
25            .isLink      (wbIsLink)
26        );
27
28 wire useRs, useRt, useLo, useHi;
29 RegUse u_RegUse(
30            .instruction (exInstruction),
31            .useRs       (useRs),
32            .useRt       (useRt),
33            .useLo       (useLo),
34            .useHi       (useHi)
35        );
36
37 always @(*) begin
38     rsFwd = 0;
39     rtFwd = 0;
40
41     // forward form WB
42     if (wbWriteReg != 0) begin
43         if (wbIsLink) begin
44             // forward wbNextPC from WB
45             if (wbWriteReg == rs) begin
46                 rsFwd = 1;
47                 rsFwdData = wbNextPC;
48             end
49             if (wbWriteReg == rt) begin
50                 rtFwd = 1;
51                 rtFwdData = wbNextPC;
52             end
```

```
53          end
54          else if (wbMemRead) begin
55              // forward wbMemOut from WB
56              if (wbWriteReg == rs) begin
57                  rsFwd = 1;
58                  rsFwdData = wbMemOut;
59              end
60              if (wbWriteReg == rt) begin
61                  rtFwd = 1;
62                  rtFwdData = wbMemOut;
63              end
64          end
65          else begin
66              // forward wbALUOut from WB
67              if (wbWriteReg == rs) begin
68                  rsFwd = 1;
69                  rsFwdData = wbALUOut;
70              end
71              if (wbWriteReg == rt) begin
72                  rtFwd = 1;
73                  rtFwdData = wbALUOut;
74              end
75          end
76      end
77
78      // forward from MEM
79      if (memWriteReg != 0) begin
80          if (memIsLink) begin
81              // forward memNextPC from MEM
82              if (memWriteReg == rs) begin
83                  rsFwd = 1;
84                  rsFwdData = memNextPC;
85              end
86              if (memWriteReg == rt) begin
87                  rtFwd = 1;
88                  rtFwdData = memNextPC;
89              end
90          end
91          else if (memMemRead) begin
92              // load and use hazard, should never occur
93          end
94          else begin
95              // forward memALUOut from MEM
96              if (memWriteReg == rs) begin
97                  rsFwd = 1;
98                  rsFwdData = memALUOut;
99              end
100             if (memWriteReg == rt) begin
```

```
                rtFwd = 1;
                rtFwdData = memALUOut;
            end
        end
    end


    // FOR LO, HI => only rs will use them
    // forward lo, hi from WB
    if (wbWriteLoHi) begin
        if (useLo) begin
            rsFwd = 1;
            rsFwdData = wbALUOut;
        end
        if (useHi) begin
            rsFwd = 1;
            rsFwdData = wbALUOutHi;
        end
    end

    // forward lo, hi from MEM
    if (memWriteLoHi) begin
        if (useLo) begin
            rsFwd = 1;
            rsFwdData = memALUOut;
        end
        if (useHi) begin
            rsFwd = 1;
            rsFwdData = memALUOutHi;
        end
    end
end

endmodule // Forward
```

Stall 模块 面对加载-使用冒险，我们只有通过暂停流水线来解决。我延续了教材上的设计，在 Load 指令位于 EX 阶段时识别并输出 stall 信号。前文已经提到过，stall 信号由 NewPC 模块负责处理，NewPC 再进一步决定要暂停哪些阶段，向那些阶段寄存器发送最终的 stallID、stallEX 等信号。

加载指令和使用指令在寄存器不冲突时不必暂停流水线，这样可提高流水线性能，为此，我创建了一个简单的 RegUse 模块判断寄存器是否使用。Stall 模块的实现如下。

```
module Stall(
            input wire          exMemRead,
            input wire [ `REG] exWriteReg,
            input wire [`WORD] idInstruction,
```

```
5
6             output wire         stall
7         );
8
9 wire [`REG] idRs = `GET_RS(idInstruction);
10 wire [`REG] idRt = `GET_RT(idInstruction);
11
12 wire useRs, useRt;
13 RegUse u_RegUse(
14             .instruction (idInstruction),
15             .useRs       (useRs),
16             .useRt       (useRt)
17         );
18
19 assign stall = exMemRead && ((useRs && exWriteReg == idRs) || (useRt &&
       exWriteReg == idRt));
20
21 endmodule // Stall
```

2.10.4 处理控制冒险：NewPC 模块

当分支预测错误，或加载-使用冒险要求暂停流水线时，由 NewPC 模块统一负责处理，再进一步决定要暂停哪些阶段，向那些阶段寄存器发送最终的 stallID、stallEX 等信号。这一部分我已在 2.5.1 节进行了讨论。

2.11 最终整合：PipeCPU 模块和 PipeSystem 模块

PipeCPU 模块 这里我延续了上一次实验的设计，构建了一个 PipeCPU 模块，其中不带任何内存装置，而通过如下输入输出信号与 Cache 和主存交流。由于实现了 Cache 和多周期指令内存，需要额外添加一个 instReady 输入。

```
1 module PipeCPU(
2             input clk,
3
4             output wire [`WORD] pc,
5             input  wire [`WORD] inst,
6             input  wire         instReady,
7
8             output wire [`WORD] dataAddress, writeMemData,
9             output wire         memRead, memWrite,
10            output wire [ `MMD] memMode,
11            input  wire [`WORD] readMemData
12        );
```

由于我们的设计严格遵循模块化的要求，上一次实验设计的大多数模块得以直接使用，且整个 CPU 模块中只出现了三种语句，即 wire 声明语句、assign 赋值语句、模块实例化语句，而不包含任何逻辑代码。这使得模块设计极其清晰、简洁。PipeCPU 和上一次实验的 CPU 的最大区别是，需要在其内部实例化大量的阶段寄存器，同时，对于一些跨阶段的控制模块，如 NewPC、Forward 等，需要小心地根据设计草图决定其输入来自哪个阶段。由于篇幅所限，这部分代码不再列出，请参见 PipeCPU.v。

PipeSystem 模块 即 Top 模块。构建好了 PipeCPU 模块后，我们可以最终将 Cache、主存储器和 CPU 整合起来，构建顶层 PipeSystem 模块。这里可以观察到，我设计的 Cache 模块完全扮演一个中间人角色，对于主存和 CPU 而言都是透明的。

同样地，PipeSystem 模块的代码十分简洁，它不接受输入、输出，但接受两个参数，指定指令内存的转储文件路径和时钟频率。

```
module PipeSystem #(parameter textDump = "path/to/text/dump",
                    parameter PERIOD   = 10);

// --- Clock ---
reg clk;
always #(PERIOD) clk = !clk;
initial clk = 1;

// --- Memory ---
wire [ `WORD] pc;
wire [ `WORD] instruction;
wire          instReady;
wire [ `WORD] imAddr;
wire [`QWORD] imQdata;
wire          imInstReady;

Cache u_Cache(
    .pc          (pc),
    .inst        (instruction),
    .instReady   (instReady),
    .imAddr      (imAddr),
    .imQdata     (imQdata),
    .imInstReady (imInstReady)
);

InstMemory #(textDump) u_InstMemory(
    .clk   (clk),
    .addr  (imAddr),
```

```
29      .qdata (imQdata),
30      .ready (imInstReady)
31  );
32
33
34  wire [`WORD] dataAddress;
35  wire [`WORD] writeData, readData;
36  wire [ `MMD] memMode;
37  wire         memRead, memWrite;
38  DataMemory u_DataMemory(
39      .clk       (clk),
40      .address   (dataAddress),
41      .writeData (writeData),
42      .mode      (memMode),
43      .memRead   (memRead),
44      .memWrite  (memWrite),
45      .readData  (readData)
46  );
47
48
49  // --- MIPS CPU ---
50  PipeCPU u_PipeCPU(
51      .clk          (clk),
52      .pc           (pc),
53      .inst         (instruction),
54      .instReady    (instReady),
55      .dataAddress  (dataAddress),
56      .writeMemData (writeData),
57      .memRead      (memRead),
58      .memWrite     (memWrite),
59      .memMode      (memMode),
60      .readMemData  (readData)
61  );
62
63  endmodule // PipeSystem
```

3 仿真测试

3.1 准备工作：C 编译器

在上一次实验的报告中，我介绍到我配置了 MIPS 的 gcc 交叉编译器和 binutils 二进制工具，以及将汇编代码转换为 Verilog 格式的 Makefile 流程。随着本次实验中对乘

法指令的完善，该 MIPS 处理器设计已基本实现了 MIPS I ISA 中除异常、特权、除法、乘累加等以外的全部功能，已经具有运行由通用编译器编译得到的 C 程序的能力。在此基础上，我配置了 C 交叉编译器和对应的 Makefile。

`mips32-elf-gcc` 假设编译出的结果是在真正的操作系统中运行的，因此，我们首先要配置栈顶地址和 `$ra` 寄存器的值。一般而言，这类事情是通过与汇编编写的 entry 程序进行链接实现的。为简单考虑，我向链接器传入参数`-Wl,--section-start=.text=0x8`，要求从 `0x8` 地址开始定位程序，然后直接在 `0x8` 前插入两条指令 `241d1000` 和 `241f0400`，即设置栈顶地址为 `0x1000` 和返回地址寄存器为 `0x400`，完成基本的环境配置。

除此之外，由于我设计的处理器大端的，并基于 MIPS I 指令集，因此还要向编译器传递`-mips1 -EB -mabi=32` 来指定生成的机器码，避免出现 MIPS 32 指令集中的新指令。传递该参数后，编译器也会自动关闭延迟槽。

注 该配置要求程序运行时栈大小不超过 `0x1000`、指令总长度不超过 `0x400`。如果测试程序较大，需要手动修改这两个值以满足要求，否则程序可能不能给出正确结果。

```
AS      = mips32-elf-as
CC      = mips32-elf-gcc
OBJDUMP = mips32-elf-objdump
OBJCOPY = mips32-elf-objcopy
ASFLAGS = -O0 -mips1 -EB
CCFLAGS = -O0 -mips1 -EB -mabi=32 -Wall -nostdlib -fno-zero-initialized-in-bss
    -mno-local-sdata -Wl,--section-start=.text=0x8
SRCS    = $(wildcard *.mips *.c)
_LSTS   = $(SRCS:.mips=.mem)
LSTS    = $(_LSTS:.c=.mem)
OUTDIR  = ./Products

...

%.mem: %.c
    mkdir -p $(OUTDIR)
    $(CC) -o a.out $< $(CCFLAGS)
    $(OBJDUMP) -d a.out > $(OUTDIR)/$(@:.mem=.lst)
    $(OBJCOPY) --dump-section .text=a.bin a.out
    $(OBJCOPY) -I binary -O binary --reverse-bytes=4 a.bin r.bin
    echo "241d1000\n241f0400\n" > $(OUTDIR)/$@
    od -An -v -t x4 r.bin | tr ' ' '\n' | sed '/^$$/d' >> $(OUTDIR)/$@
    rm -f a.bin a.out r.bin
```

不妨再次进行一些简单的试验。将我自己编写的递归计算 Fibonacci 数的测试 C 程序 `FibonacciR.c` 进行操作，可得到 `Resources/Products/`下两个文件：

- FibonacciR.lst 详细展示了每个指令和它的地址、二进制码，非常方便调试。这里也可以看到，代码段定位在 0x8 地址。

```
00000008 <_start>:
   8:   27bdffd8    addiu   sp,sp,-40
   c:   afbf0024    sw  ra,36(sp)
  10:   afbe0020    sw  s8,32(sp)
  14:   afb0001c    sw  s0,28(sp)
  18:   03a0f025    move    s8,sp
...
00000060 <fibonacci>:
  60:   27bdffe0    addiu   sp,sp,-32
  64:   afbf001c    sw  ra,28(sp)
  68:   afbe0018    sw  s8,24(sp)
  6c:   afb00014    sw  s0,20(sp)
...
```

- FibonacciR.mem 可直接导入 Verilog 进行测试，两句环境初始化指令也已经添加进去了。

```
241d1000
241f0400

27bdffd8
afbf0024
afbe0020
afb0001c
03a0f025
00001025
...
```

3.2 模块仿真测试

对于一些较为独立的模块，我编写了专门的测试激励文件，其中有一部分模块直接从上一次实验中继承下来，没有进行大的修改，这里就略去它们的测试结果。所有测试都使用自己撰写的 `assert 宏与预期结果进行比较。运行时，可直接观察控制台输出，如无“ASSERTION FAILED”警告，则代表测试通过。

限于篇幅，本节只引用仿真测试的波形图，具体代码请您参考对应的 tb 文件。

3.2.1 InstMemory 仿真测试

InstMemory 现在是多周期内存，一次读取 128 位字。仿真测试波形图如图 3 所示，运行过程中没有“ASSERTION FAILED”警告。

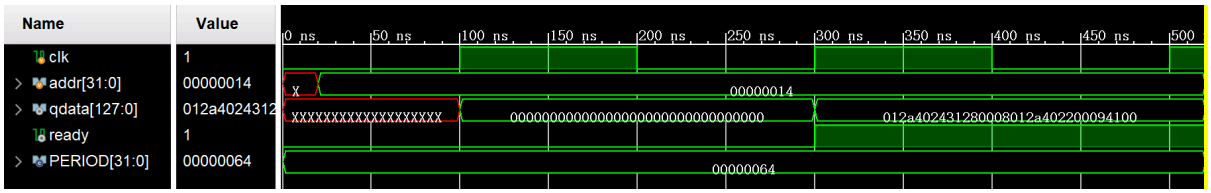

Figure 3: `InstMemory` 仿真测试

3.2.2 `RegisterFile` 仿真测试

`RegisterFile` 现在包含了乘法指令需要的 LO、HI 寄存器。仿真测试波形图如图 4 所示，运行过程中没有“ASSERTION FAILED”警告。

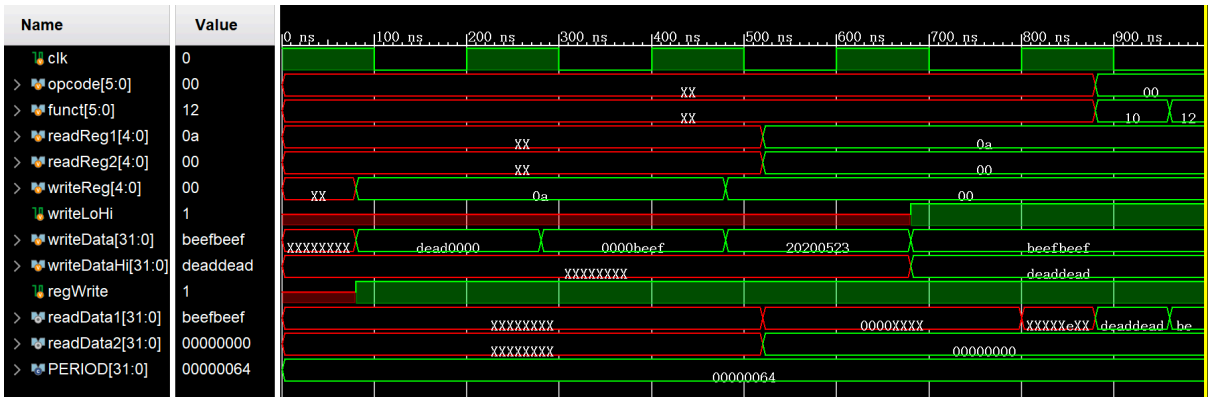

Figure 4: `RegisterFile` 仿真测试

3.2.3 `WriteReg` 仿真测试

`RegisterReg` 现在也要处理写入 LO、HI 寄存器的信号 `writeLoHi`。仿真测试波形图如图 5 所示，运行过程中没有“ASSERTION FAILED”警告。

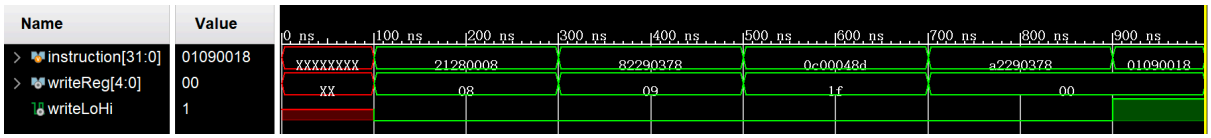

Figure 5: `WriteReg` 仿真测试

3.3 `PipeSystem` 仿真测试

本实验中，我编写了 7 个 MIPS 汇编程序（4 个继承自上次实验的功能性测试，1 个乘法测试，2 个流水线冒险测试）与 2 个 C 程序（递归计算 Fibonacci 数，迭代乘法计算 Catalan 数），汇编或编译后导入了模拟器中进行仿真。它们的激励文件为以 PS 开头的 9 个 `tb` 激励模块。

4 个继承自上次实验的功能性测试均成功通过，测试波形与单周期处理器大致相同，运行过程中没有“ASSERTION FAILED”警告，受篇幅所限，不再在这里赘述。剩下的 3 个汇编程序和 2 个 C 程序也基本覆盖了这些测试中的内容，下面将逐个进行介绍。

3.3.1 乘法运算仿真测试

汇编代码 位于 Resources/Multiplication.mips，包含有符号、无符号乘法，如下所示：

```
    .set noreorder

main:
    lw      $t1, 0($zero)
    lw      $t2, 4($zero)

    multu   $t2, $t1            # $t2 * $t1 = Hi and Lo registers
    mfhi    $t3
    sw      $t3, 8($zero)
    mflo    $t3
    sw      $t3, 12($zero)

    sub     $t2, $zero, $t2     # $t2 = $zero - $t2
    mult    $t2, $t1
    mfhi    $t4
    sw      $t4, 16($zero)
    mflo    $t4
    sw      $t4, 20($zero)
```

测试结果 数据内存波形图如图 6 所示，运行过程中没有“ASSERTION FAILED”警告，观察也可得知，符合程序运行预期结果。

3.3.2 数据冒险仿真测试

汇编代码 位于 Resources/DataHazard.mips，将许多存在数据相关的指令排布在一起，如下所示：

```
    .set noreorder

main:
    li      $t1, 1          # $t1 = 1
    li      $t5, 3          # $t5 = 3
    li      $t6, 4          # $t6 = 4
    nop
```

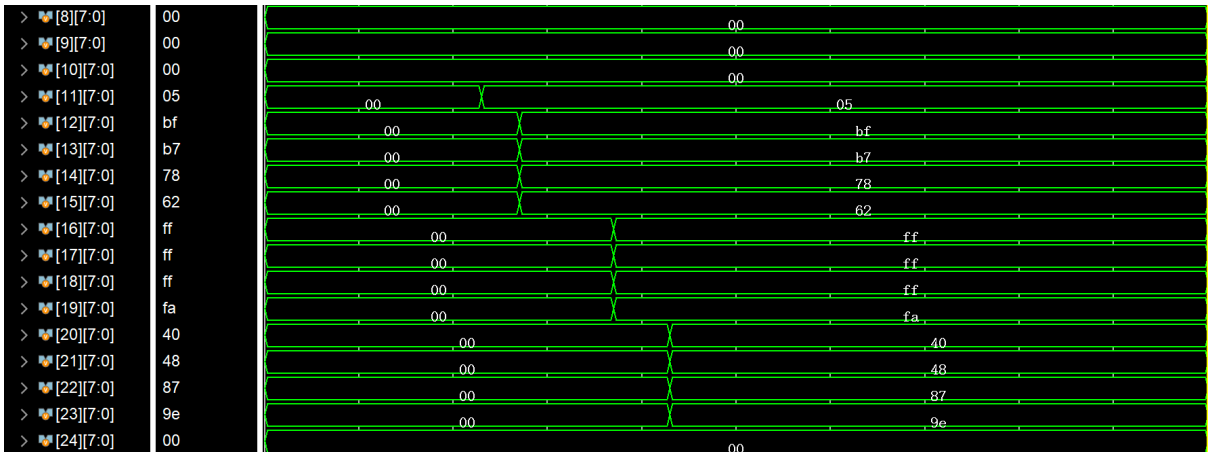

Figure 6: Multiplication 仿真测试

```
    nop
    addi    $t1, $t1, 2     # $t1 = $t1 + 2
    sw      $t1, 0($zero)
    lw      $t2, 0($zero)
    add     $t3, $t2, $t1       # $t3 = $t2 + $t1 = 6, load and use
    sub     $t4, $t3, $t1       # $t4 = $t3 - $t1 = 3
    add     $t5, $t3, $t4       # $t5 = $t3 + $t4 = 9
    bge     $t5, $t6, target    # if $t5 >= 4 then target
    addi    $t5, $t5, 3         # $t5 = $t5 + 3 = 12, should not be executed

target:
    sw      $t5, 4($zero)
```

测试结果 寄存器波形图如图 7 所示，数据内存波形图如图 8 所示，运行过程中没有“ASSERTION FAILED”警告，观察也可得知，符合程序运行预期结果。

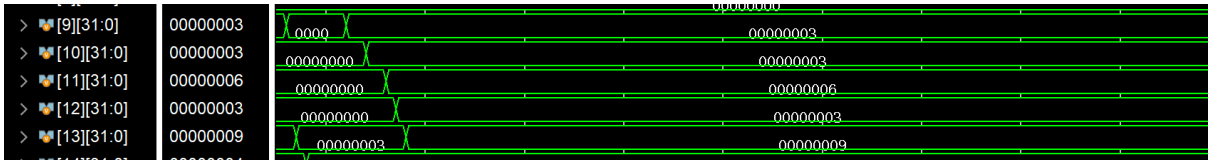

Figure 7: DataHazard 仿真测试-寄存器

3.3.3 控制冒险仿真测试

汇编代码 位于 Resources/ControlHazard.mips，主要检测流水线应对控制冒险时 flush 的效果，如下所示：

```
    .set noreorder
```

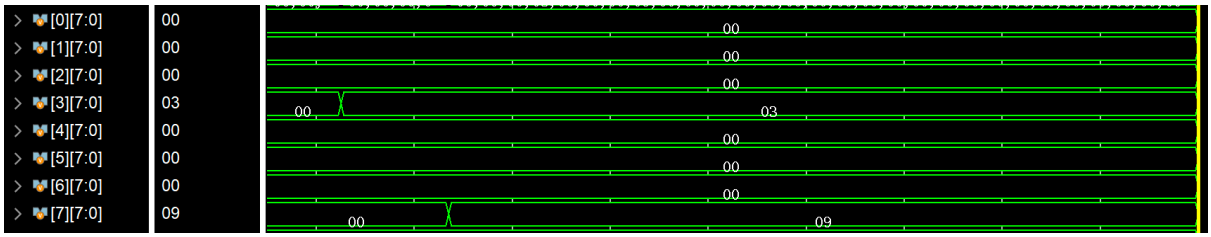

Figure 8: DataHazard 仿真测试-内存

```
2
3  main:
4      li      $t3, 0x1
5      move    $t4, $t3        # $t4 = $t3 = 0x1
6      li      $t1, 0x88       # $t1 = 0x88
7      li      $t2, 0x88       # $t2 = 0x88
8      nop
9      nop
10     nop
11     beq     $t1, $t2, l1    # if $t1 == $t2 then l1
12     addi    $t3, $t3, 1     # $t3 = $t3 + 1
13     addi    $t4, $t4, 1     # $t4 = $t4 + 1
14 l1:
15     nop
16     nop
17     # $t3, $t4 should be still 0x1
18     j       l2              # jump to l2
19     addi    $t3, $t3, 1     # $t3 = $t3 + 1
20     addi    $t4, $t4, 1     # $t4 = $t4 + 1
21 l2:
22     nop
23     nop
24     # $t3, $t4 should be still 0x1
25     jal     l3
26     addi    $t3, $t3, 1     # $t3 = $t3 + 1
27     addi    $t4, $t4, 1     # $t4 = $t4 + 1
28 l3:
29     nop
30     nop
31     # $t3, $t4 should be still 0x1
32     la      $ra, l4
33     nop
34     nop
35     jr      $ra
```

```
    addi    $t3, $t3, 1     # $t3 = $t3 + 1
    addi    $t4, $t4, 1     # $t4 = $t4 + 1
l4:
    nop
    nop
```

测试结果 寄存器波形图如图 9 所示，运行过程中没有 “ASSERTION FAILED” 警告，观察也可得知，符合程序运行预期结果。

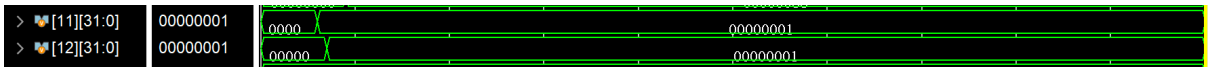

Figure 9: ControlHazard 仿真测试

3.3.4 综合仿真测试：计算 Fibonacci 数

Fibonacci 数的定义为：

$$F_0 = F_1 = 1 \quad F_n = F_{n-1} + F_{n-2}, n \geq 2$$

C 代码 位于 Resources/FibonacciR.c，从 0 地址处读取输入，将结果写入 4 地址处。这种递归计算 Fibonacci 数的效率很低，因此运行周期很长，方便我们对处理器设计的可靠性进行检测，发掘潜在的漏洞。代码如下所示：

```c
int fibonacci(int n);

void _start() {
    int n = *(int *)0;
    *(int *)4 = fibonacci(n);
}

int fibonacci(int n) {
    if (n <= 1) return 1;
    return fibonacci(n - 1) + fibonacci(n - 2);
}
```

激励模块 在激励模块中，向 0 地址写入输入 “10”，运行 30,000 周期后可得到计算结果。

```
initial begin: test
    `memFile[3] = 10; // fibonacci[10] ...
    #300000;
```

```
    `assert(`memFile[7], 89); // == 89 ?
    $finish;
end
```

测试结果 数据内存波形图如图 10 所示，运行过程中没有 “ASSERTION FAILED” 警告，观察也可得知，符合程序运行预期结果，成功得到结果 $F_{10} = 89$。

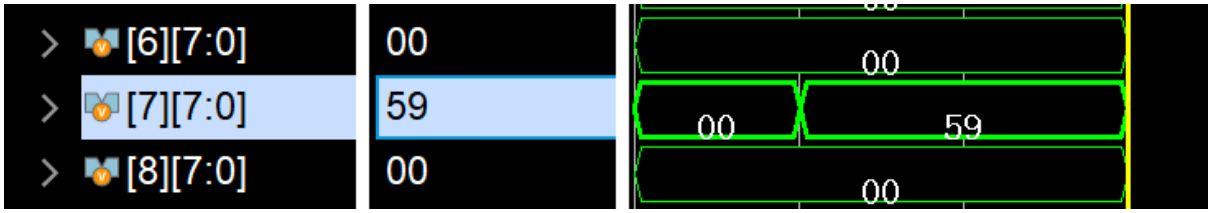

Figure 10: 综合仿真测试：计算 Fibonacci 数

注 若运行时间不足，或数据内存大小小于 `0x1000`，程序不能给出正确结果。请使用 `run 300000` 等指令手动指定运行时间进行测试。

3.3.5 综合仿真测试：计算 Catalan 数

Catalan 数是一个著名的数列，应用有二叉树个数、合法括号的个数等等，其定义为：

$$C_0 = 1 \quad C_{n+1} = \sum_{i=0}^{n} C_i C_{n-i}, n \geq 1$$

C 代码 位于 `Resources/Catalan.c`，涉及到了两重循环与乘法运算，最终将第 10 个 Catalan 数写入地址 0 处。代码如下所示：

```
void _start() {
    unsigned *catalans = (unsigned *)4;
    catalans[0] = 1;
    for (int i = 1; i < 10; i++) {
        catalans[i] = 0;
        for (int j = 0; j < i; j++) {
            catalans[i] += catalans[j] * catalans[i - 1 - j];
        }
    }
    *(int *)0 = catalans[9];
}
```

激励模块 在激励模块中，运行 10,000 周期后可得到计算结果。

```
initial begin: test
    #100000;
    `assert(`wordAt(0), 4862); // the tenth Catalan number!
    $finish;
end
```

测试结果 数据内存波形图如图 11 所示，运行过程中没有 “ASSERTION FAILED” 警告，观察也可得知，符合程序运行预期结果，成功得到结果 $C_9 = 4862$。

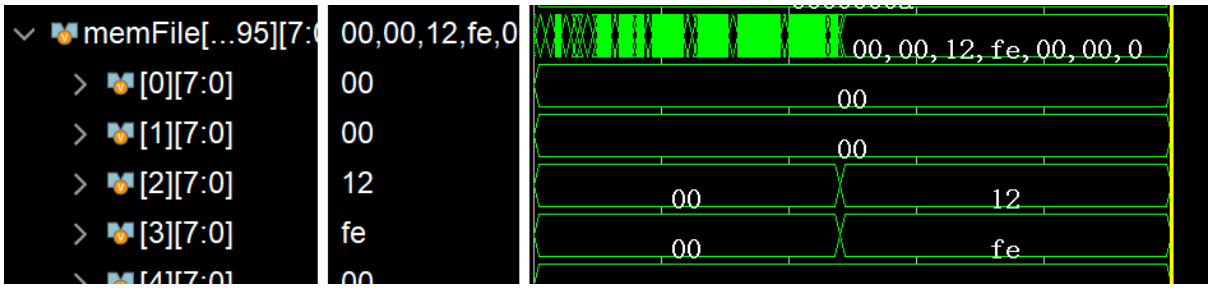

Figure 11: 综合仿真测试：计算 Catalan 数

注 若运行时间不足，或数据内存大小小于 `0x1000`，程序不能给出正确结果。请使用 `run 100000` 等指令手动指定运行时间进行测试。

4 总结与感想