RISC-V 性能模拟器实验报告

姓名： 张煌昭
学号： 1400017707
院系： 元培学院
邮箱： zhang_hz@pku.edu.cn
手机： 17888838127

一．实验要求

在 Lab 2.1 的单周期处理器的功能模拟器的基础之上，对其进行改造，完成多周期处理器和流水线处理器的性能模拟器，要求如下：

1. 指令阶段划分大于等于 5 阶段，建议 5 级划分；
2. 规定定点指令中除去 64 位乘（MUL）和所有除（DIV）与求模（REM）类指令，所有指令在所有阶段的周期数均为 1；
3. 定点指令中，两个 32 位数的乘指令（MULW）的执行阶段周期数为 1，所有含有 64 位数的乘指令（MUL）的执行阶段周期数为 2；
4. 定点指令中，除法（DIV）和求模（REM）类指令的执行周期都为 40，但当同时需要商和余数，即 DIV 和 REM 紧连着时，两条指令的总执行周期为 40；
5. 流水线处理器中，接连的乘法可以流水，除法不可以流水；
6. 流水线处理器中，默认没有任何的结构冒险；
7. 流水线处理器中，旁路不作要求，默认使用停顿解决数据冒险；
8. 流水线处理器中，对于 jal 指令，默认后续取值正确；对于 SB 型指令（如 beq 指令），执行条件在执行阶段获得；对于 jalr 指令，跳转地址在译码阶段获得，此处会遭遇数据冒险；默认采用静态分支预测，默认分支跳转发生；若发生预测错误，需要取消已经进入流水线的指令，并重新取指。

本次实验所用测试程序均使用 RISC-V 交叉编译工具链编译源码直接获得，详情请见 README 文档。

二．设计部件

由于单周期的功能模拟并没有考虑数据通路和控制信号，因此需要对性能模拟器设计各个阶段的部件。分为五阶段：取指（IF），译码（ID），执行（EX），访存（MEM）和写回（WB）。考虑到需要在多周期和流水线两个任务中复用这些部件，因而比较简单的方法就是将可以产生所有控制信号的指令部分打包为控制字，各个部件在内部使用专门的控制信号产生逻辑，利用其获得的控制字，产生自己需要的控制信号。各个阶段部件的数据通路和控制信号如下。

1. 取指部件（IF）

取指部件完成的工作为：将 PC 更新为 next PC 并再将新的 PC+4 写入 next PC 中，之后根据 PC 指示的地址，从指令存储器中取出 32 位定长指令，之后按照各个类型指令的位域对指令进行切分；切分完成后将 rs1，rs2，rd 这三个寄存器地址的固定位域锁存，等待向下一部件传送；IF 还需要锁存所有可能的立即数集合 Imm Set，用于产生控制信号的控制字

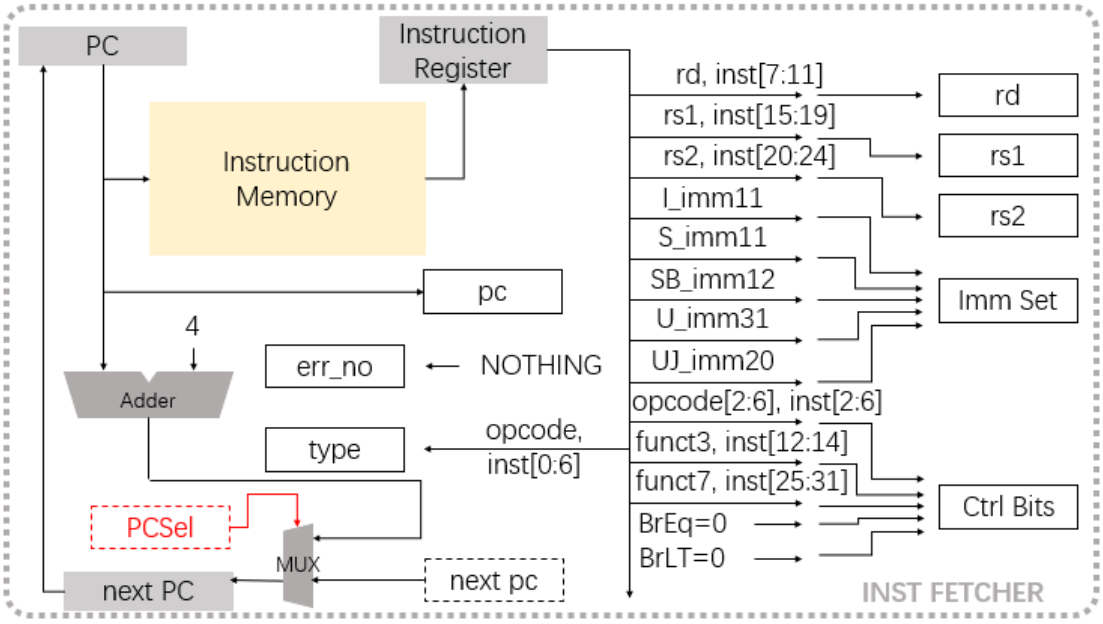
Ctrl Bits，以及状态代码 err_no 和指令类型 type；此外，取指部件还需要留有设置 next PC 的通道，以便 SB 类型指令在执行阶段获得跳转地址后设置 next PC 进行跳转。

需要特殊说明的是，type 其实并不必要，因为可以直接从 Ctrl Bits 中获得该值，增加这一数据是为了方便流水线时部件组装。

由于 IF 不需要产生任何控制信号，因此没有控制逻辑，但其会受到执行阶段产生的 PCSel 信号的控制。

IF 的 RTL 描述如下，数据通路和控制信号如下图，图中虚线框为从其它部件获得的值或信号，实线框为需要提供给其它部件的值或控制字等。

```
PC <- next_PC;
next_PC <- 2-1MUX(PC + 4, next_pc); # the 2-1MUX is controlled by PCSel signal
inst[0:31] <- IMem[PC];
rd <- inst[7:11];   rs1 <- inst[15:19];   rs2 <- inst[20:24];
ImmSet <- (I_imm11, S_imm11, SB_imm12, U_imm31, UJ_imm20)
   I_imm11 <- inst[20:31];   S_imm11 <- inst[7:11] | (inst[25:31] << 5);
   SB_imm12<-(inst[7:7]<<11)|(inst[8:11]<<1)|(inst[25:30]<<5)|(inst[31:31]<<12);
   U_imm31 <- inst[12:31] << 12;
   UJ_imm20 <- (inst[12:19] << 12) | (inst[20:20] << 11)
                  | (inst[21:30] << 1) | (inst[31:31] << 20);
CtrlBits <- (HOpcode, funct3, funct7, BrEq, BrLT)
   HOpcode <- inst[2:6];   funct3 <- inst[12:14];   funct7 <- inst[25:31];
   BrEq <- 0;   BrLT <- 0;
err_no <- NOTHING; # NOTHING is the usual status
type <- inst[0:6]
```



取指部件的实现详情，请见所附代码 simulator/if.h 和 simulator/if.cpp 中的 IF 类。

2. 译码部件（ID）

传统的译码部件进行代码切分和译码的工作，但本次试验中这些工作已经在 IF 完成，

因此 ID 只需要进行立即数扩展和读取寄存器的工作即可。ID 部件完成的工作为：首先根据控制字 Ctrl Bits 产生所需的 ImmSel 控制信号，之后立即数扩展器（Immediate Generator）根据 ImmSel 控制信号从 Imm Set 中取出所需的一条并扩展至 64 位；rs1 和 rs2 作为两个寄存器堆（Reg File）地址传入 Reg File 中，并得到对应的两个寄存器的值 rs1_content 和 rs2_content；由于在运行过程中有可能出现错误（比如未知指令），将错误状态代码锁存在 err_no 中，若未出错则将原来的 err_no 锁存；此外还需要锁存 pc，type，rd，Ctrl Bit 供之后的阶段使用。

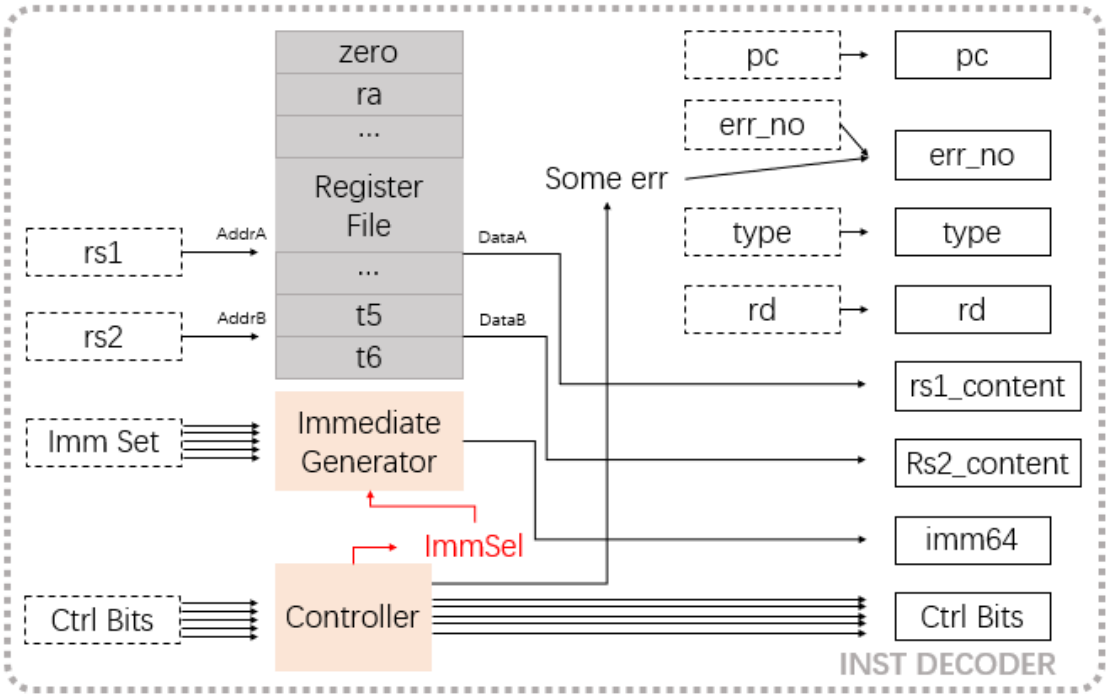
下面说明 ImmSel 的产生方式。ImmSel 共有 6 种，分别为 R_IMM，I_IMM，S_IMM，SB_IMM，U_IMM，UJ_IMM，根据 Ctrl Bits 中的 HOpcode 可以确定该指令的类型，从而确定需要使用哪一种立即数产生方式，若发现不属于上述六种中任何一种的 HOpcode，说明遭遇未知指令，设置 err_no 为 INVALID_INST。

需要特殊说明的是，err_no 被置为非 NOTHING 时，所有会改变机器状态的动作都会被禁止执行，以此来保证机器的最后正常状态。

ID 的 RTL 描述如下，数据通路和控制信号如下图。

```
ImmSel <- ID_Controller(CtrlBits);

rs1_content[0:63] <- RegFile[rs1][0:63];  rs2_content[0:63] <- RegFile[rs2][0:63];
imm64[0:63] <- ImmGenerator(ImmSet, ImmSel)[0:63];
pc <- pc_old;  type <- type_old;  rd <- rd_old;  CtrlBits <- CtrlBits_old;
err_no <- anyError ? err_no_new : err_no_old;
```



译码部件的实现详情，请见所附代码 simulator/id.h 和 simulator/id.cpp 中的 ID 类。

3. 执行部件（EX）

执行部件完成的工作为 :首先根据控制字 Ctrl Bits 产生所需的 BrUn, ASel, Bsel 和 ALUSel 控制信号（PCSel 控制信号在稍后产生）；比较器 CMP 根据 BrUn 控制信号确定使用有符号

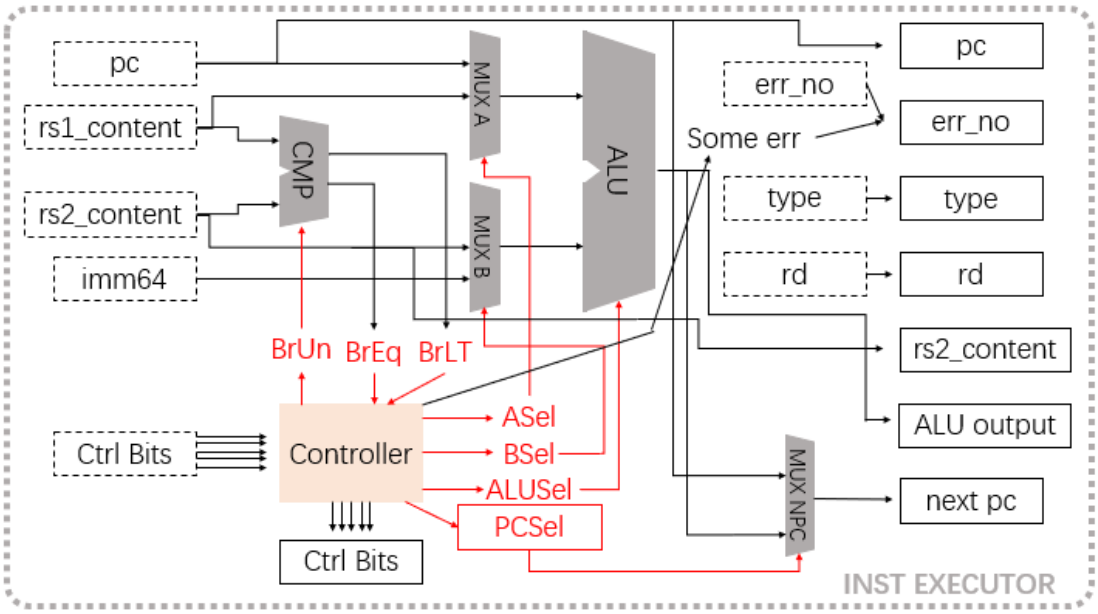
比较模式或无符号比较模式，之后将比较结果 BrEq 和 BrLT 写如 Ctrl Bits 控制字的对应位置，控制器根据 BrEq, BrLT 和控制字的其它部分产生最后一个控制信号 PCSel ;多选器 MUXA 根据控制信号 ASel 选择 pc 或 rs1_content，多选器 MUXB 根据控制信号 BSel 选择 rs2_content 或 imm64，之后算逻单元 ALU 获取 A 和 B，根据 ALUSel 控制信号选择运算模式并进行运算，运算结果锁存在 ALU output 中 ;此外还需要由多选器 MUX NPC 根据 PCSel 信号从 pc+4 和 ALU output 中选择产生 next pc，next PC 和 PCSel 信号都将传给 IF 用于跳转 ;最后将 pc，err_no，type，rd 和 rs2_content 锁存以便之后部件使用。

ASel 和 BSel 使用 Ctrl Bits 中的 HOpcode 即可产生,ALUSel 使用 Ctrl Bits 中的 HOpcode，funct7 和 funct3 组合产生；BrUn 在 HOpcode==(0x63>>2)时才会产生，否则不关心该控制信号的值，BrUn 使用 funct3 产生；PCSel 使用 Ctrl Bits 中的全部产生，若判断 HOpcode 为 B 型指令则使用 funct3，BrEq 和 BrLT 产生，若为 jal 和 jalr 则直接置为跳转。

EX 的 RTL 描述如下，数据通路和控制信号如下图。

```
(BrUn, ASel, BSel, ALUSel) <- EX_Controller_1(CtrlBits);
(CtrlBits.BrEq, CtrlBits.BrLT) <- CMP(rs1_content, rs2_content, BrUn);
PCSel <- EX_Controller_2(CtrlBits);

ALU_A <- MUXA(pc, rs1_content); # MUXA is controlled by ASel signal.
ALU_B <- MUXA(rs2_content, imm64); # MUXB is controlled by BSel signal.
ALU_output <- ALU(ALU_A, ALU_B); # ALU is controlled by ALUSel.
next_pc <- MUXNPC(ALU_output, pc+4); #MUXNPC is controlled by PCSel.
pc <- pc_old;   type <- type_old;   rd <- rd_old;   CtrlBits <- CtrlBits_old;
rs2_content <- rs2_content_old;
err_no <- anyError ? err_no_new : err_no_old;
```



执行部件的实现详情，请见所附代码 simulator/ex.h 和 simulator/ex.cpp 中的 EX 类。

4. 访存部件（MEM）

访存部件完成的工作为：首先根据控制字 Ctrl Bits 产生所需的 MemRW 控制信号，该信

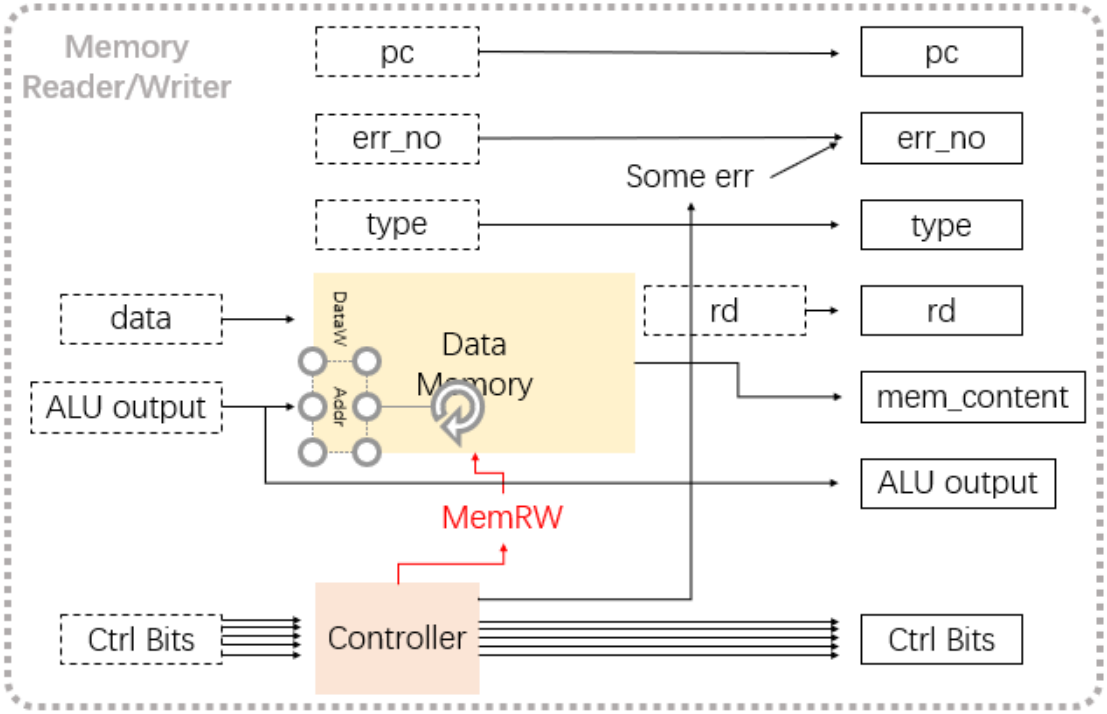
号描述了指令的读/写，以及读写的长度（Byte/ Half word/ Word/ Double word）；之后根据该控制信号将 data 写入存储器的 ALU output 地址或读出存储器的 ALU output 地址的内容；此外，将 pc，err_no，type，rd 以及 Ctrl Bits 锁存供写回阶段使用。

MemRW 控制信号的产生如下。首先根据 Ctrl Bits 中的 HOpcode 确定该指令是否是 Load 类指令或 Store 类指令，之后根据 funct3 确定读写内存的长度。

MEM 的 RTL 描述如下，数据通路和控制信号如下图。

```
MemRW <- MEM_Controller(CtrlBits);

if MemRW == READ_B:
    mem_content[0:63] <- signed_ext64(memory[ALU_output][0:7]);
elseif MemRW == READ_BU:
    mem_content[0:63] <- zero_ext64(memory[ALU_output][0:7]);
... ...
elseif MemRW == READ_D:
    Mem_content[0:63] <- memory[ALU_output][0:63];
elseif MemRW == WRITE_B:
    memory[ALU_output][0:7] <- data[0:7];
... ...
elseif MemRW == WRITE_D:
    memory[ALU_output][0:63] <- data[0:63];
else:
    Do nothing
pc <- pc_old;   type <- type_old;   rd <- rd_old;   CtrlBits <- CtrlBits_old;
ALU_content <- ALU_content_old;
err_no <- anyError ? err_no_new : err_no_old;
```



访存部件的实现详情，请见所附代码 simulator/mem.h 和 simulator/mem.cpp 中的 MEM 类。

5. 写回部件（WB）

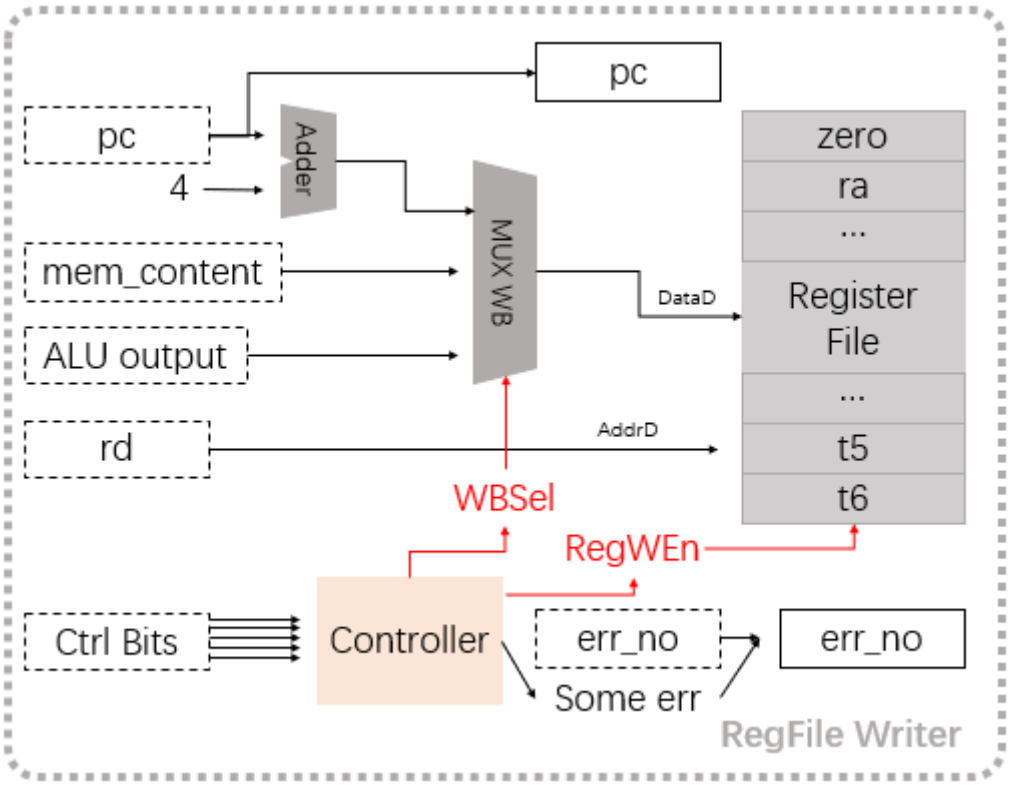
写回部件完成的工作为：首先根据控制字 Ctrl Bits 产生所需的 WBSel 和 RegWEn 控制信号；MUXWB 多选器根据 WBSel 控制信号选择 pc+4，mem_content 和 ALU output 中的一个作为 DataD；若 RegWEn 控制信号为可写，则寄存器堆 RegFile 将 DataD 写入 rd 作为地址 AddrD 的寄存器中。

WBSel 和 RegWEn 根据 Ctrl Bits 中的 HOpcode，funct3 和 funct7 产生。

WB 的 RTL 描述如下，数据通路和控制信号如下图。

```
(WBSel, RegWEn) <- WB_Controller(CtrlBits);

DataD[0:63]<-MUXWB(pc+4,mem_content,rd); #MUXWB is controlled by WBSel.
if RegWEn == Writable:
    RegFile[rd][0:63] <- DataD[0:63];
pc <- pc_old;   type <- type_old;   err_no <- anyError ? err_no_new : err_no_old;
```

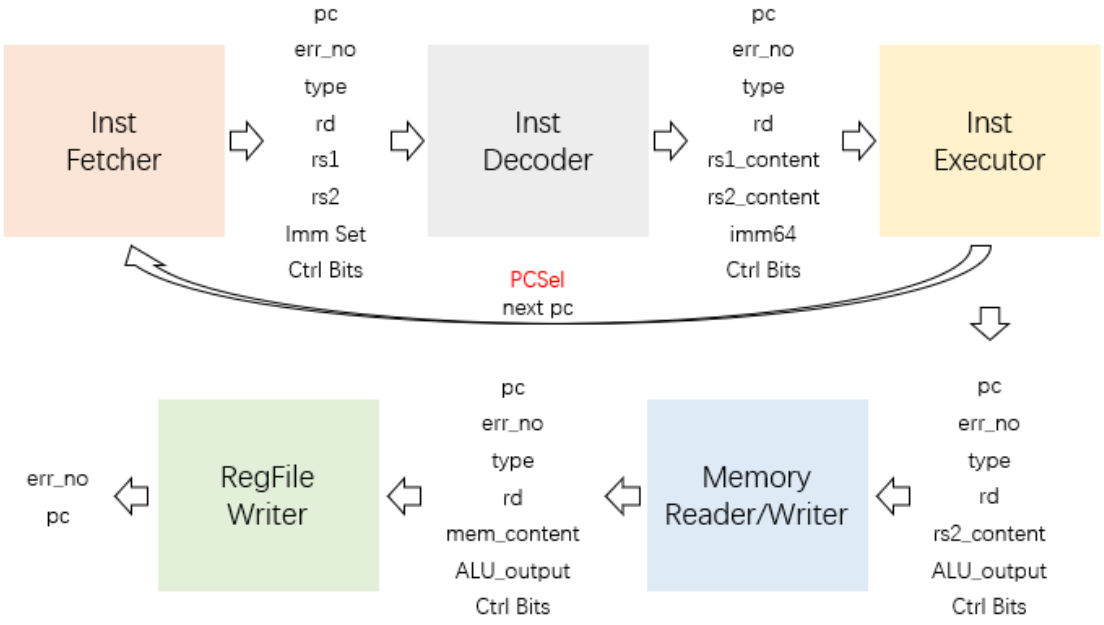


写回部件的实现详情，请见所附代码 simulator/wb.h 和 simulator/wb.cpp 中的 WB 类。

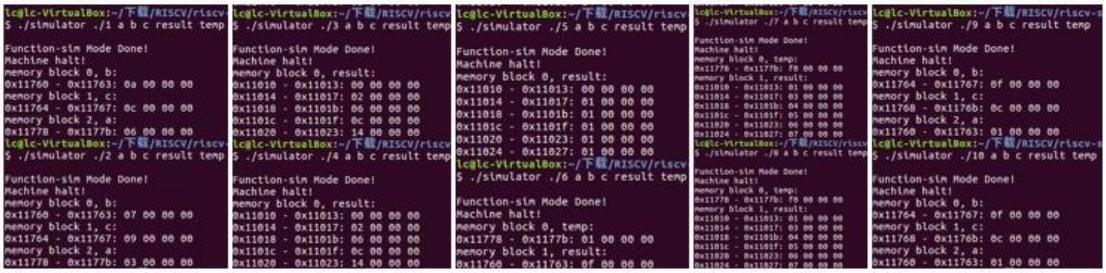
三． 组装单周期处理器模拟器

利用上述五个部件，按下图所示数据通路和控制信号连接即可组装一个单周期处理器，

指令按箭头顺序在各个部件执行一遍，最终从 WB 退出执行完毕；检查 pc 是否发生对齐错误或段错误，若有则设置 err_no ;之后检查 err_no 是否为 NOTHING 或 HALT，若为 NOTHING 则 IF 取下一条指令，若为 HALT 则停机打印各种状态，否则发生错误，停机打印错误原因。



使用测试程序 1-10 进行测试，结果如下组图。

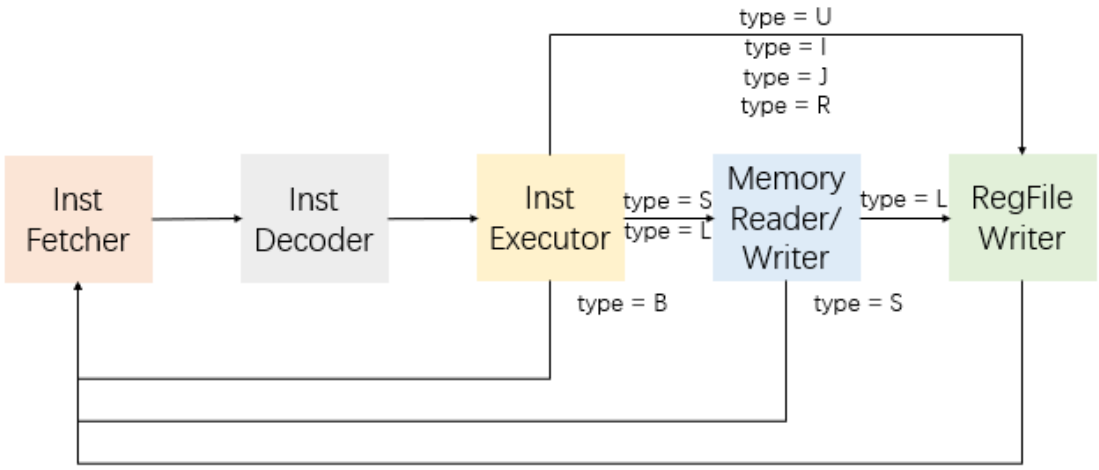


在代码 Makefile 中添加宏定义 FUNC_SIM 即可使用单周期模拟，使用方法请见所附 README 文档，详细实现请见所附代码 simulator/func_sim.h 和 simulator/func_sim.cpp 中的 Func_Sim 类。

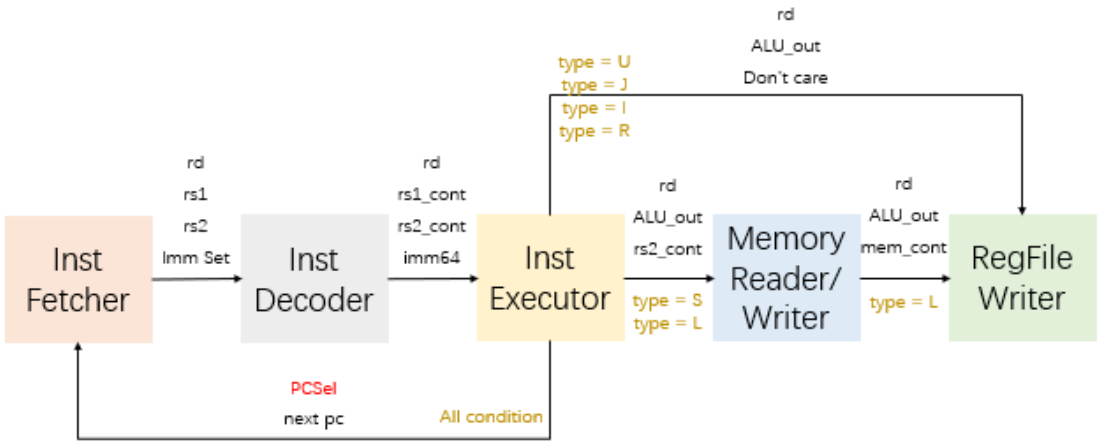
四．组装多周期处理器模拟器

1. 数据通路和控制信号

依然使用如上五个部件，对单周期处理器数据通路和控制信号进行修改，利用之前提到的特殊添加的 type 进行有限状态机的状态转移控制。有限状态机的状态转移如下图所示，连线边的标注为状态转移的条件，未标注表示无条件状态转移。



多周期处理器的数据通路和控制信号如下图所示，黑色为数据通路，红色为控制信号，黄色为该通路/信号被开启的条件。默认所有数据传输都必须传输 pc，err_no，type 和 Ctrl Bits，因此图中省去这四项。



2. 各类指令的 RTL 描述

下面举例说明 R 类，I 类（I 型中除去 Load 类），L 类（I 型中 Load 类），S 类，B 类，U 类和 J 类共 7 类指令在各个阶段的 RTL 描述，依然省去各个阶段间 pc，err_no，type 和 Ctrl Bits 四项的传输，默认这四项必须进行传输。

以 add zero, zero, zero 指令（伪指令为 nop）为例，说明 R 类，J 类，I 类和 U 类指令的 RTL 描述，各类指令在各个阶段略有不同，但大体一致。

```
add zero, zero, zero 共分作 IF，ID，EX，WB 四个阶段：
IF：  PC <- next_PC;   next_PC <- PC + 4; # 2-1MUX default 选择 PC+4
      inst[0:31] <- IMem[PC];
      rd <- inst[7:11] = 0;   rs1 <- inst[15:19] = 0;   rs2 <- inst[20:24] = 0
      HOpcode <- inst[2:6] = (0x33 >> 2);   BrEq <- False = 0;   BrLT <- False = 0
      funct3 <- inst[12:14] = 0x0;   funct7 <- inst[25:31] = 0x00;
      CtrlBits <- (HOpcode, funct3, funct7, BrEq, BrLT);
      Imm_Set <- Don't Care;
```

```
ID： rs1_content <- RegFile[rs1] = 0;  rs2_content <- RegFile[rs2] = 0;  rd <- rd = 0;
     Imm64 <- Don't Care;
EX： ALU_A <- rs1_content = 0;  ALU_B <- rs2_content = 0;  rd <- rd = 0;
     ALU_output <- ALU_A + ALU_B = 0 + 0 = 0;
     rs2_content <- Don't Care;  BrEq <- Don't Care;  BrLT <- Don't Care;
WB： wb <- ALU_output = 0;  DataD <- wb = 0;
     AddrD <- rd = 0;
     RegFile[AddrD] = zero <- DataD = 0;
```

以 beq zero, zero, offset 指令为例，说明 B 类指令的 RTL 描述。

```
beq zero, zero, offset 共分作 IF，ID，EX 三个阶段：
IF： PC <- next_PC;  next_PC <- PC + 4; # 2-1MUX default 选择 PC+4
     inst[0:31] <- IMem[PC];
     rd <- inst[7:11] = Don't Care;  rs1 <- inst[15:19] = 0;  rs2 <- inst[20:24] = 0
     HOpcode <- inst[2:6] = (0x63 >> 2);  BrEq <- False = 0;  BrLT <- False = 0
     funct3 <- inst[12:14] = 0x0;  funct7 <- Don't Care;
     CtrlBits <- (HOpcode, funct3, funct7, BrEq, BrLT);
     Imm_Set <- SB_Imm = offset;
ID： rs1_content <- RegFile[rs1] = 0;  rs2_content <- RegFile[rs2] = 0;  rd <- Don't Care
     Imm64 <- signed_ext(SB_Imm, 12); = offset
EX： BrEq <- CMP_EQ(rs1_content, rs2_content) = CMP_EQ(0x0, 0x0) = true = 1;
     BrUn <- Don't Care;  BrLT <- Don't Care;
     ALU_A <- pc;  ALU_B <- Imm64;  ALU_output <- ALU_A + ALU_B = pc + Imm64;
     PCSel <- NEXT_PC;  next_pc <- ALU_output;
```

以 lb ra, offset(sp)指令为例，说明 L 类指令的 RTL 描述。

```
lb ra, offset(sp)共分作 IF，ID，EX，MEM，WB 五个阶段：
IF： PC <- next_PC;  next_PC <- PC + 4; # 2-1MUX default 选择 PC+4
     inst[0:31] <- IMem[PC];
     rd <- inst[7:11] = 1;  rs1 <- inst[15:19] = 2;  rs2 <- Don't Care;
     HOpcode <- inst[2:6] = (0x03 >> 2);  BrEq <- False = 0;  BrLT <- False = 0
     funct3 <- inst[12:14] = 0x0;  funct7 <- Don't Care;
     CtrlBits <- (HOpcode, funct3, funct7, BrEq, BrLT);
     Imm_Set <- I_Imm = offset;
ID： rs1_content <- RegFile[rs1] = RegFile[sp];  rs2_content <- Don't Care;
     Imm64 <- signed_ext(I_Imm, 11) = offset;  rd <- rd = 1;
EX： ALU_A <- rs1_content = RegFile[sp];  ALU_B <- Imm64 = offset;  rd <- rd = 1;
     ALU_output <- ALU_A + ALU_B = RegFile[sp] + offset;
     PCSel <- PC_PLUS_4;  BrEq <- Don't Care;  BrLT <- Don't Care;
MEM：Addr <- ALU_out = RegFile[sp] + offset;  DataW <- Don't Care;
     mem_content <- signed_ext(memory[Addr][0:7]);
WB： wb <- mem_content = signed_ext(memory[Addr][0:7]);  DataD <- wb;
```

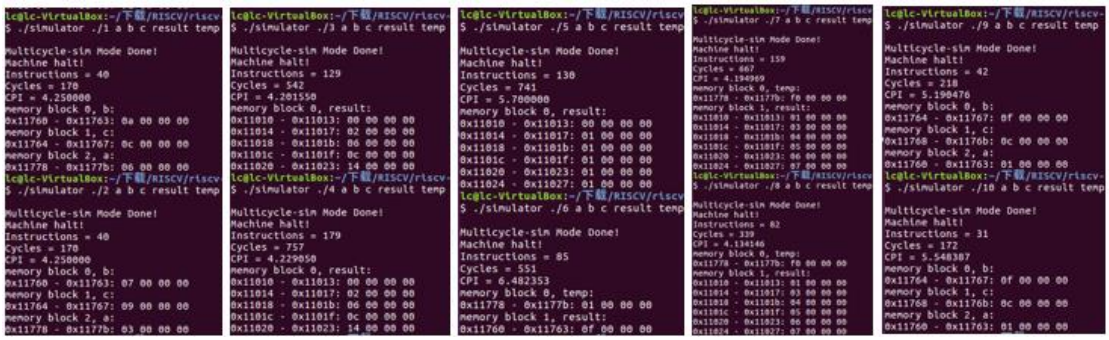
```
AddrD <- rd = 1;
RegFile[AddrD] = RegFile[ra] <- DataD = signed_ext(memory[Addr][0:7]);
```

以 sb ra, offset(sp)指令为例，说明 S 类指令的 RTL 描述。

```
sb ra, offset(sp)共分作 IF，ID，EX，MEM 四个阶段：
IF： PC <- next_PC;  next_PC <- PC + 4; # 2-1MUX default 选择 PC+4
     inst[0:31] <- IMem[PC];
     rd <- Don't Care;  rs1 <- inst[15:19] = 2;  rs2 <- inst[20:24] = 1;
     HOpcode <- inst[2:6] = (0x23 >> 2);  BrEq <- False = 0;  BrLT <- False = 0
     funct3 <- inst[12:14] = 0x0;  funct7 <- Don't Care;
     CtrlBits <- (HOpcode, funct3, funct7, BrEq, BrLT);
     Imm_Set <- S_Imm = offset;
ID： rs1_content <- RegFile[rs1] = RegFile[sp];
     rs2_content <- RegFile[rs2] = RegFile[ra];
     Imm64 <- signed_ext(S_Imm, 11) = offset;  rd <-Don't Care;
EX： ALU_A <- rs1_content = RegFile[sp];  ALU_B <- Imm64 = offset;  rd <- Don't Care;
     ALU_output <- ALU_A + ALU_B = RegFile[sp] + offset;
     rs2_content <- rs2_content = RegFile[ra];
     PCSel <- PC_PLUS_4;  BrEq <- Don't Care;  BrLT <- Don't Care;
MEM：Addr <- ALU_out = RegFile[sp] + offset;
     DataW <- rs2_content = RegFile[ra];
     Memory[Addr][0:7] <- DataW[0:7];  mem_content <- Don't Care;
```

3. 测试结果

使用测试程序 1-10 进行测试，结果如下组图，程序输出与作为对照的单周期处理器的模拟器输出一致，说明模拟器工作状态基本正常。



在代码 Makefile 中添加宏定义 MULTICYCLE_SIM 即可使用多周期模拟，使用方法请见所附 README 文档，详细实现请见所附代码 simulator/multicycle_sim.h 和 simulator/multicycle_sim.cpp 中的 Multicycle_Sim 类。

各个程序的指令数及 CPI 见下表所示。发现其中有一些程序的 CPI 较高，研究源代码发现这些程序中存在较多的除法（测试程序 5 和 6），而一个除法会使得 EX 执行 40 周期，从而使得 CPI 急速上升；而事实上，在很短的程序中的一条除法指令（测试程序 9 和 10），就

会导致 CPI 上升。

测试程序	指令数	周期数	CPI	测试程序	指令数	周期数	CPI
1	40	170	4.2500	6	85	551	6.4824
2	40	170	4.2500	7	159	667	4.1950
3	129	542	4.2016	8	82	339	4.1341
4	179	757	4.2291	9	42	218	5.1905
5	130	741	5.7000	10	31	172	5.5484

10 个测试程序的平均 CPI 为 4.8181。若按照本次试验的五阶段划分可以使得周期τ减少至 1/5，则根据下式可以使得执行时间缩短，获得提升。

$$\text{CPU time} = \text{IC} \times \text{CPI} \times \tau$$

而如果耗时的乘除法指令在真实应用中所占比例较测试程序而言更少，或者有接连的除法指令可以缩短执行时间，那么 CPI 会比测试的平均值进一步降低，使得执行时间进一步缩短，获得更多提升。

五．组装流水线处理器模拟器

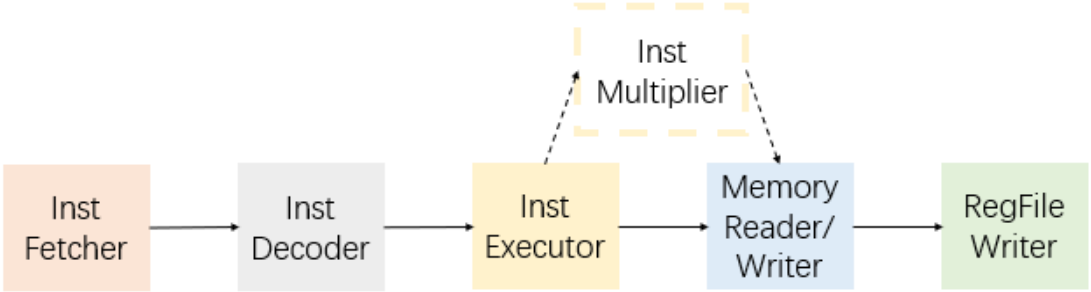
1. 新增乘法部件（EX2）

由于流水线处理器中需要对接连的乘法指令进行流水，因此增加一级“隐藏的”执行阶段——执行乘法阶段（EX2），用于对乘法流水的支持。

该部件默认处于失活状态，但其每个周期都可以被 EX 的 64 位乘指令激活，也可以被 EX 的非 64 位乘指令失活。若 EX2 失活，则流水线只有 5 级，按照 5 级流水线的方式进行流水作业；若 EX2 被激活，则该流水线呈现出 6 级，按照 6 级流水线的方式进行流水作业。

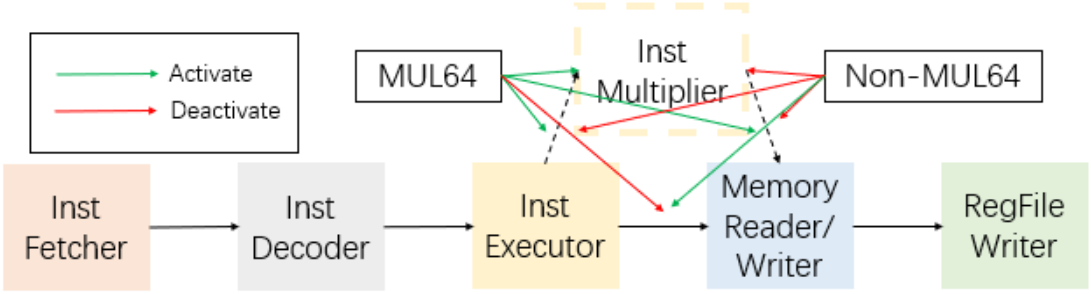
针对 EX2，考虑如下三种可能情况：

a. 接连的非 64 位乘指令，中间没有 64 位乘指令出现。这种情况下，整个处理器是 5 级流水线，按照正常流水即可。如下图所示。

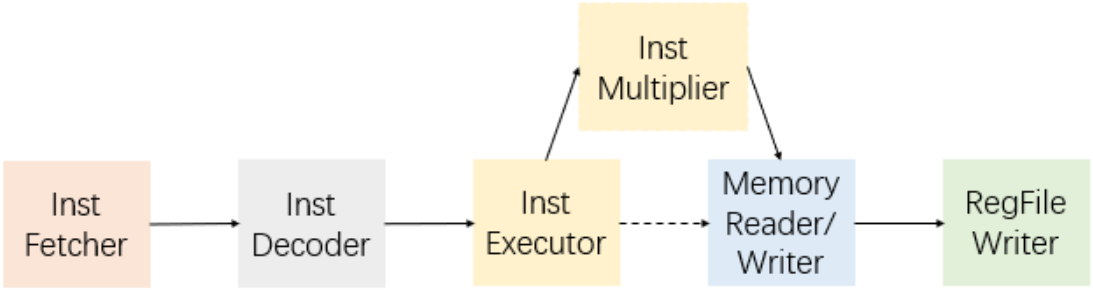


b. 接连的非 64 位乘指令之中，插入一个 64 位乘指令。这种情况下，在 64 位乘指令到达 EX 前，处理器按 5 级流水线正常工作；64 位乘指令到达 EX 阶段，运行一个周期后，激活 EX2，此时 EX 不再像 MEM 流水，而是转而向 EX2 流水，EX2 再向 MEM 流水；64 位乘指令在 EX2 阶段运行一个周期后，其后正处于 EX 阶段的非 64 位乘指令试图进入 EX2 阶段使得 EX2 失活，此时 EX2 从流水线上断开，EX 重新恢复向 MEM 流水，但由于此时处于 EX2

的 64 位乘指令和处于 EX 的非 64 位乘指令同时试图进入 MEM，会发生冒险，因此规定处于 EX 的非 64 位乘指令暂停流水（Stall）1 周期后再进入 MEM 阶段。如下图所示。



c. 接连的 64 位乘指令，中间没有非 64 位乘指令出现。这种情况下，第一个 64 位乘指令按照上述方式激活 EX2，使得处理器变为 6 级流水线，之后所有的 64 位乘指令按照 6 级流水的方式执行，如下图所示；直至出现非 64 位乘指令使得 EX2 失活，处理器恢复 5 级流水。

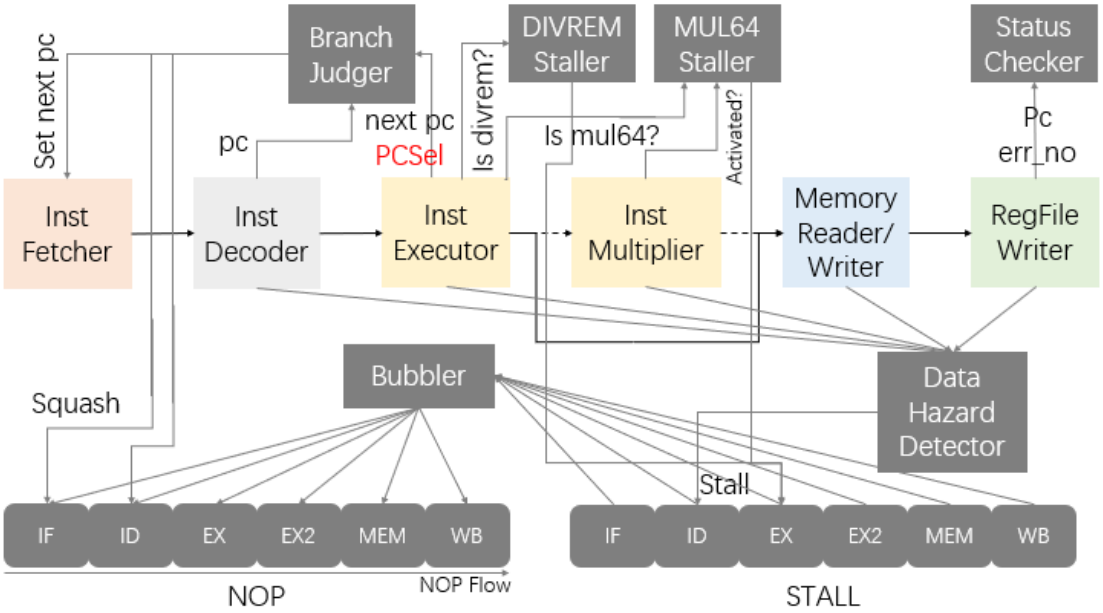


EX2 部件完成的工作为：首先接收来自 EX 的 Ctrl Bit，根据 Ctrl Bit 迅速判断是否是 64 位乘指令；若是则激活 EX2，连接 EX2 与 EX 和 EX2 与 MEM，并断开 EX 与 MEM 阶段；否则失活 EX2，断开 EX2 与 EX 和 EX2 与 MEM，连接 EX 与 MEM；若 EX2 失活则不进行任何工作，否则从 EX 接收乘法的中间结果并继续完成乘法，锁存于 ALU output。当然 EX2 必须完成 pc，Ctrl Bits，type，rd 等的传递工作。

由于 EX2 是为了便于模拟器实现而设计出来的一个“虚拟的”部件，在真实的流水线中并不一定存在这样一个”隐藏的”部件，真实情况下的乘法流水，更可能是利用两套乘法器硬件完成的。

2. 数据通路和控制信号

由于流水线冒险的存在，各个阶段的 NOP 和 STALL 控制信号必须加入其中。各个部件的功能不变，按照上面描述的形式进行连接和流水，需要新添加的部分如下图所示。NOP 表示该指令在此阶段及之后的阶段无效，因而 NOP=True 时，不会进行任何改变及其状态的操作，并且 NOP 会伴随指令一直传递下去，直至出流水线；而 STALL 表示该指令在此阶段以及处于之前所有阶段的指令停止流水（向下一阶段传播），因而 STALL=True 时，部件向下传播的数据通路被切断，并且 STALL 必须每个周期重置为 False 并重新设置。



图中 Branch Judger 用于判断是否发生错误跳转，一旦判断为是，则将错误的 IF 和 ID 的 NOP 置为 True，并给 IF 重新设置正确的 next pc；DIVREM Staller 和 MUL64 Staller 用于判断是否处于除法/求模阶段和 MUL64 指令与 Non-MUL64 指令冲突，若为是则将 EX 及之前阶段的 STALL 置为 True；Data Hazard Detector 用于检测数据冒险，一旦发现 ID 和之后阶段的 RAW，就设置 ID 及 IF 的 STALL 为 True，直至数据冒险消除；Bubbler 用于在最后一个 STALL 之后的阶段插入空泡，保证及其状态正确；此外，NOP 个位会按照规则同指令一起向下一阶段传播。

3. 冒险及解决方案

由于规定不发生结构冒险，因此该模拟器模拟的流水线处理器只会发生数据冒险和控制冒险，讨论如下。

a. 数据冒险

数据冒险分作两类，一类是 ID 读寄存器堆端口号与写存储器内容回寄存器堆端口号重叠引起的冒险，另一类是 ID 读寄存器堆端口号与写 ALU output 回寄存器堆端口号重叠引起的冒险。第一类至多可以通过 STALL 一周期等待读出存储器内容并前递解决，称作 Load-use Hazard；第二类可以通过在 EX 阶段前递 ALU output 完全消除。

由于未作要求，因此对于数据冒险不进行任何前递，仅通过 STALL 的方式消除冒险。

b. 控制冒险

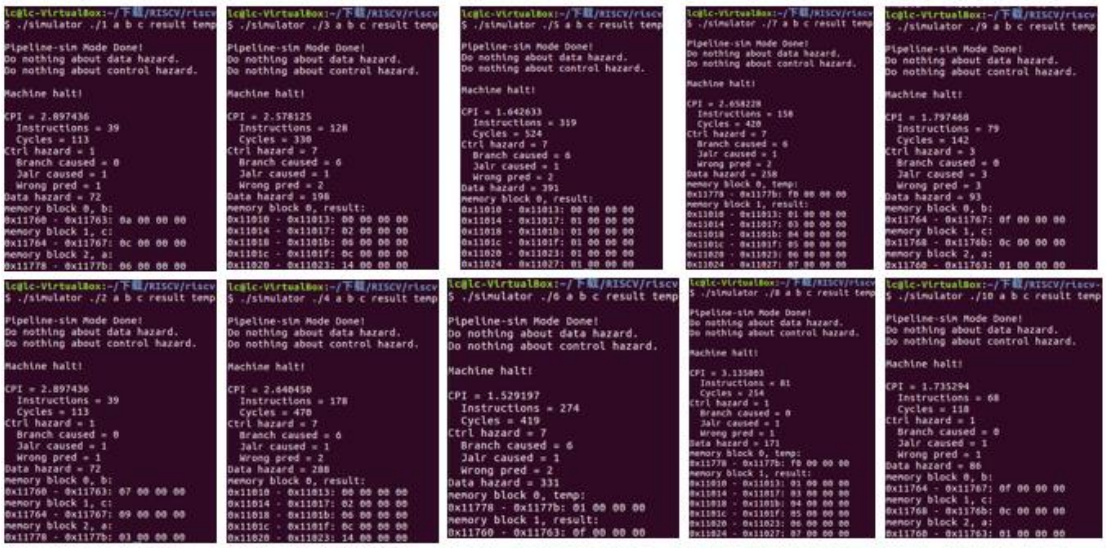
会引起控制冒险的指令有三类，一类是 B 类指令，例如 beq，其只能在 EX 阶段得到确定正确的 next pc；另一类是 jal 指令，其必然发生跳转，并且跳转地址可以根据指令确定，因而有可能在 IF 阶段直接获取到 jal 指令的跳转地址；最后一类是 jalr 指令，其必然发生跳转，但其跳转地址必须读取寄存器堆才能确定，因而 jalr 指令最早只可能在 ID 阶段获取到跳转地址。

由于进行静态预测跳转，因此对于 B 类指令默认预测不跳转，jal 指令和 jalr 指令在最早阶段（jal 指令为 IF 阶段，jalr 指令为 ID 阶段）获取跳转地址并进行跳转。

4. 测试结果

使用测试程序 1-10 进行测试，结果如下组图，程序输出与作为对照的单周期处理器的

模拟器输出一致，说明模拟器工作状态基本正常。



在代码 Makefile 中添加宏定义 PIPELINE_SIM 即可使用单周期模拟，使用方法请见所附 README 文档，详细实现请见所附代码 simulator/pipeline_sim.h 和 simulator/piepline_sim.cpp 中的 Pipeline_Sim 类。

各个程序的指令数及 CPI 见下表所示，平均 CPI 为 2.3512，相比于预期值要高。分析冒险数据，发现数据冒险所占比例极高且数目极多，因此根据 Amdahl 法则，有效考虑优化数据保险，下面采用旁路的方法优化数据冒险。

测试程序	指令数	周期数	CPI	控制冒险	转移错误	数据冒险
1	39	113	2.8974	1	1	72
2	39	113	2.8974	1	1	72
3	128	330	2.5781	7	2	198
4	178	470	2.6405	7	2	288
5	319	524	1.6426	7	2	391
6	274	419	1.5292	7	2	331
7	158	420	2.6582	7	2	258
8	81	254	3.1358	1	1	171
9	79	142	1.7975	3	3	93
10	68	118	1.7353	1	1	86

5. 旁路优化

使用旁路技术，对流水线进行优化。由于数据冒险中的一类为 ID 读寄存器堆端口号与写 ALU output 回寄存器堆端口号重叠引起的冒险，因此可以通过直接将 EX 计算出的 ALU output 传送至 ID 的寄存器读出的内容的方法，完全消除这一类冒险。

在代码 Makefile 中添加宏定义 USE_BYPASS 即可使用旁路优化，使用方法请见所附 README 文档，详细实现请见所附代码 simulator/pipeline_sim.h 和 simulator/piepline_sim.cpp 中的 Pipeline_Sim 类。

使用旁路后，各个程序的指令数及 CPI 见下表所示，平均 CPI 为 1.1961，降低了 1.1551，

有明显提升。观察冒险数据，发现数据冒险明显降低，说明旁路优化非常有效。

测试程序	指令数	周期数	CPI	控制冒险	转移错误	数据冒险
1	39	50	1.2821	1	1	6
2	39	50	1.2821	1	1	6
3	128	161	1.2578	7	2	26
4	178	221	1.2416	7	2	36
5	319	352	1.1034	7	2	26
6	274	297	1.0839	7	2	16
7	158	196	1.2405	7	2	31
8	81	95	1.1728	1	1	9
9	79	92	1.1646	3	3	4
10	68	77	1.1324	1	1	4