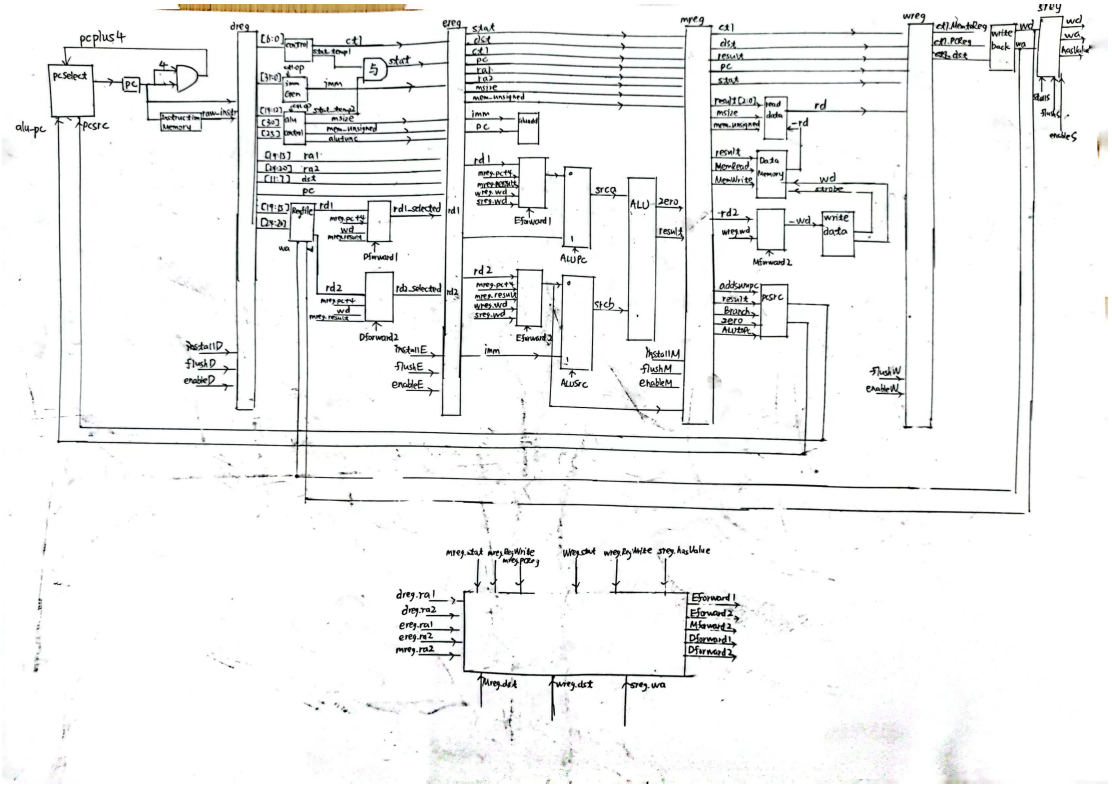
简易电路图：



结构冒险：

由于指令内存和数据内存在实际中是同一块内存，且读和写无法同时进行，所以 fetch 阶段和 memory 阶段会产生结构冒险。

被助教们用仲裁解决了(∩ ..∩) ♡ 。

控制冒险：

当遇到跳转指令时，流水线预测成功，则正常执行，若预测失败，需要将跳转之后的指令都清除，否则 CPU 会得到错误的结果。

解决方案：流水线在 Memory 阶段决定是否跳转。若不跳转，流水线正常执行，若跳转，将 IF/ID,ID/EX,EX/MEM 寄存器清空，并将 pc 值改为跳转值。此时若 fetch 阶段访存未结束，则先阻塞 pc，IF/ID,ID/EX,EX/MEM，并清空 MEM/WB 寄存器。直到 fetch 阶段访存结束后再执行操作。

电路实现：在 core.sv 中设置 stallF,stallD,flushD,stallE,flushE,stallM,flushM,flushW 来控制流水线寄存器，对于控制冒险，表达式的变量有 ireq.valid,iresp.data_ok,PCSrc(为 1 则跳转)通过真值表得出表达式并化简，最终解决控制冒险。

数据冒险：

（1）alu-alu 数据依赖:前一条指令 alu 计算出来的结果要写入目的寄存器，该目的寄存器是后一条指令 alu 的数据来源。前一条指令的结果存储在 EX/MEM 流水线寄存器中，最早在 Memory 阶段可以给出，后一条指令需要在 Execute 阶段用到该结果。

解决方案：转发，将 EX/MEM 流水线寄存器中存储的 alu 计算结果转发给 Execute 阶段。

电路实现：pipeline/forward/fwt.sv 中，输入 EX/MEM 流水线寄存器的 dst(Mdst)，RegWrite(MRegWrite)，stat(Mstat),ID/EX 流水线寄存器的 ra1(Era1)，ra2(Era2)，输出 Eforward1,Eforward2 来控制 Execute 阶段的 rd1,rd2。当 Eforward1 或 Eforward2==Result 时，用 EX/MEM 寄存器中的 result（alu 计算出来的结果）代替 rd1 或 rd2，当 Eforward1 或 Eforward2 的值为 Regs 时，不代替。

（2）alu-sd 数据依赖:前一条指令 alu 计算出来的结果要写入目的寄存器，该目的寄存器是后一条指令 memory 阶段写入数据内存的数据来源。前一条指令的结果存储在 EX/MEM 流水线寄存器中，最早在 Memory 阶段可以给出，后一条指令需要在 Memory 阶段用到该结果。

解决方案：转发，将 EX/MEM 流水线寄存器中存储的 alu 计算结果转发给 Execute 阶段，并将转发的值在时钟上升沿时存入 EX/MEM 寄存器中。

电路实现：在（1）的基础上，将 rd2 转发后的值存入 mreg（EX/MEM 寄存器）中。

（3）pc-alu 数据依赖：前一条指令的 pc+4 的值要写入目的寄存器，该目的寄存器是后一条指令 alu 的数据来源。前一条指令的 pc 存储在各个流水线寄存器中，后一条指令需要在 Execute 阶段用到该结果。

解决方案：转发，将 EX/MEM 寄存器中 pc+4 的值转发给 Execute 阶段。

电路实现：在（1）的基础上，在 fwt.sv 中增加输入 EX/MEM 寄存器控制信号 PCReg(控制将 pc+4 的值给寄存器)。当 Eforward1 或 Eforward2==PCplus4 时，用 EX/MEM 寄存器中的 pc+4 代替 rd1 或 rd2。

（4）pc-sd 数据依赖：前一条指令的 pc+4 的值要写入目的寄存器，该目的寄存器是后一条指令 memory 阶段写入数据内存的数据来源。

在(1),(2),(3)的基础上已解决。

（5）load-alu 数据依赖：前一条指令从数据内存中取出的值要写入目的寄存器，该目的寄存器是后一条指令 alu 的数据来源。前一条指令的结果存储在 MEM/WB 流水线寄存器中，最早在 Writeback 阶段可以给出，后一条指令需要在 Execute 阶段用到该结果。

解决方案：阻塞加转发，插入一条气泡使得当后一条指令运行到 Execute 阶段时停顿一周期（暂不考虑访存延迟），再将 writeback 阶段的结果转发给 Execute 阶段，

电路实现：在 core.sv 中，

```
assign ld_aluBubble = dataE.ctl.MemRead & dataE.ctl.RegWrite & (dataD.ra1 == dataE.dst | dataD.ra2 == dataE.dst)
```

将 stallE(阻塞 ID/EX 寄存器)的值设为 ld_aluBubble(暂不考虑访存延迟，之后可通过或运算加上去)。flushM 的值设为 ld_aluBubble & enableW。这样就实现了阻塞。

转发则是在 fwt.sv 中新增输入 MEM/WB 流水线寄存器的 dst（Wdst），RegWrite（WRegWrite）,stat(Wstat)，输出 Eforward1 或 Eforward2==Wd 时，用 writeback 阶段的结果 wd 代替 rd1 或 rd2。

（6）load-sd 数据依赖：前一条指令从数据内存中取出的值要写入目的寄存器，该目的寄存器是后一条指令 memory 阶段写入数据内存的数据来源。前一条指令的结果存储在 MEM/WB 流水线寄存器中，最早在 Writeback 阶段可以给出，后一条指令需要在 Memory 阶段用到该结果。

解决方案：转发，将 writeback 阶段的结果转发给 memory 阶段。

电路实现：fwt.sv 中新增输出 Mforward2，当值为 Wd 时，用 writeback 阶段的结果代替 rd2。

（7）load-alu-alu 数据依赖:第一条指令从数据内存中取出的值要写入目的寄存器，该目的寄存器是 2,3 条指令 memory 阶段写入数据内存的数据来源。由于 load-alu 数据依赖，第一条指令到达 writeback 阶段时，第三条指令到达 decode 阶段。由于 writeback 阶段后没有寄存器，当第三条指令到达 execute 阶段时，寄存器中已经没有第一条指令的结果了。

解决方案：转发，将 writeback 阶段的结果转发给 decode 阶段。

电路实现：fwt.sv 中新增输入 IF/ID 流水线寄存器的 ra1(Dra1),ra2(Dra2)，输出 Dforward1，Dforward2.当 Dforward1 或 Dforward2 等于 Wd 时，用 writeback 阶段的结果代替 rd1，rd2。

（8）alu-load-alu 数据依赖：第一条指令 alu 计算出来的结果要写入目的寄存器 1，第二条指令从数据内存取出的值要写入目的寄存器 2，目的寄存器 1 和 2 是第三条指令 alu 的数据来源。第一条指令到达 writeback 阶段时，第三条指令处于 execute 阶段。由于 load-alu 数据依赖，下一个时钟周期，第三条指令依旧处于 execute 阶段，但第一条指令已写回，寄存器中不再有第一条指令的数据。

解决方案：转发，将 EX/MEM 流水线寄存器中存储的 alu 计算结果转发给 decode 阶段，并将转发的值在时钟上升沿时存入 ID/EX 寄存器中。

电路实现：当 Dforward1 或 Dforward2 等于 Result 时，用 EX/MEM 寄存器中的 result 代替 rd1，rd2.

（9）load-load-alu 数据依赖：第一条指令和第二条指令的从数据内存取出的值要写入不同的目的寄存器 1 和 2，目的寄存器 1 和 2 是第三条指令 alu 的数据来源，且是不同的目的寄存器。load 指令的结果最早在 writeback 阶段给出。当第一条指令到达 writeback 阶段时，第二条指令处于 memory 阶段，第三条指令处于 execute 阶段。此时 2,3 两条指令间存在 load-alu 数据依赖，解决方案是阻塞加转发，但阻塞会导致第三条指令在下一个时钟周期仍处于 execute 阶段，此时其依旧需要第一条指令的结果，但 writeback 阶段后没有寄存器，这使得寄存器中已经没有第一条指令的结果了。

解决方案：增加寄存器，保存第一条指令 writeback 阶段的结果，将结果转发到 execute

阶段。

电路实现：pipeline/regfile 目录下新增 sreg.sv，wa,wd 分别对应 writeback 阶段的 wa，wd，hasValue 标识该寄存器是否有效。设置 enableS 来控制寄存器是否需要更新。当不满足 load-load-alu 情况时，用 flushS 来清零寄存器。此外，考虑到访存延迟，用 installS 来阻塞 sreg。fwt.sv 中新增输入 sreg 的 wa，hasValue，输出 Eforward1 或 Eforward2==Sregwd 时，用 sreg 的 wd 代替 rd1 或 rd2。

此外，当有多个转发来源时，取最近阶段的数据，保证转发的结果是最新的。这可以通过改变 fwt.sv 中 if 语句的条件来实现。

以上是主要的数据冒险问题，通过解决这些问题，流水线已经能够正常运转了。

随机延迟的支持：

读指令，读写数据内存会产生延迟。

解决方法：当读指令未完成时，阻塞 pc 值，而 decode 及之后的阶段并不受影响，可以继续执行；当读写数据内存未完成时，阻塞 pc，dreg,ereg,mreg，并清零 wreg，来避免多次写回。当两个都未完成时，交给仲裁。

根据这些情况，将 ireq.valid,iresp.data_ok,dreq.valid,dresp.data_ok 作为变量，画出真值表后化简，将化简后的值赋给 stallF,stallD,flushD,stallE,flushE,stallM,flushM,flushW，从而支持随机延迟。

此外，流水线冒险也会影响 stallF，stallD 等的值（比如控制冒险，load-alu 数据依赖），在支持随机延迟延迟的基础上，对相应的 stall,flush 等进行或运算后便可正常执行。

两个 part 测试通过的仿真和上板截图

阶段一：

```
[WARNING] difftest store queue overflow
Pass!
[src/cpu/cpu-exec.c:393,cpu_exec] nemu: HIT GOOD TRAP at pc = 0x0000000080003c90
[src/cpu/cpu-exec.c:394,cpu_exec] trap code:0
[src/cpu/cpu-exec.c:74,monitor_statistic] host time spent = 1699 us
[src/cpu/cpu-exec.c:76,monitor_statistic] total guest instructions = 3993
[src/cpu/cpu-exec.c:77,monitor_statistic] simulation frequency = 2350206 instr/s
sh: 1: spike-dasm: not found
```

```
Hello world!


BASYS3 GPIO/UART DEMO!
```

阶段二。

```
------------------------------------------------------------
Run conwaygame
Play Conway's life game for 200 rounds.
seed=9278977
```

```
                              **
                              **
** **
** **
[src/cpu/cpu-exec.c:393,cpu_exec] nemu: HIT GOOD TRAP at pc = 0x00000000800152c0
[src/cpu/cpu-exec.c:394,cpu_exec] trap code:0
```

```
Round 193/200
Round 194/200
Round 195/200
Round 196/200
Round 197/200
Round 198/200
Round 199/200


               **
               **


                         *
                        * *
                        * *
                         *


Exit with code = 0
```