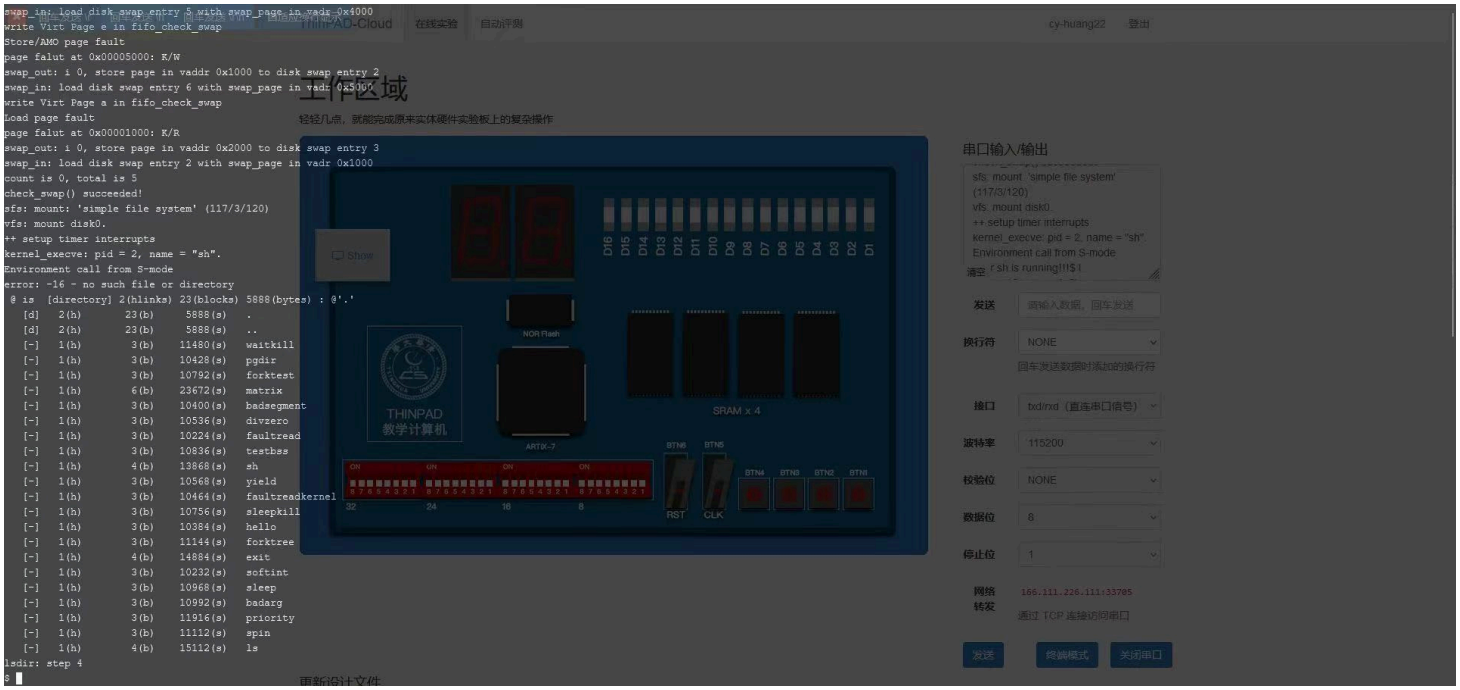
五级流水线 RISC-V 处理器设计与实现

> 计 23 张瀚宸，计 23 黄宸宇，计 23 秦晨阳

项目介绍

本项目中，我们基于 SpinalHDL 和 Verilator 工具链，设计并实现了五级流水线 RISC-V 处理器，能够支持 uCore 操作系统。为了方便起见，我们将 `counter` 单独作为一级流水，因此严格意义上我们实现的 CPU 是六级流水线 CPU。



环境

- Scala 2.13.14
- SpinalHDL 1.10.2a
- Verilator 5.024
- vivado 2019.2

仓库

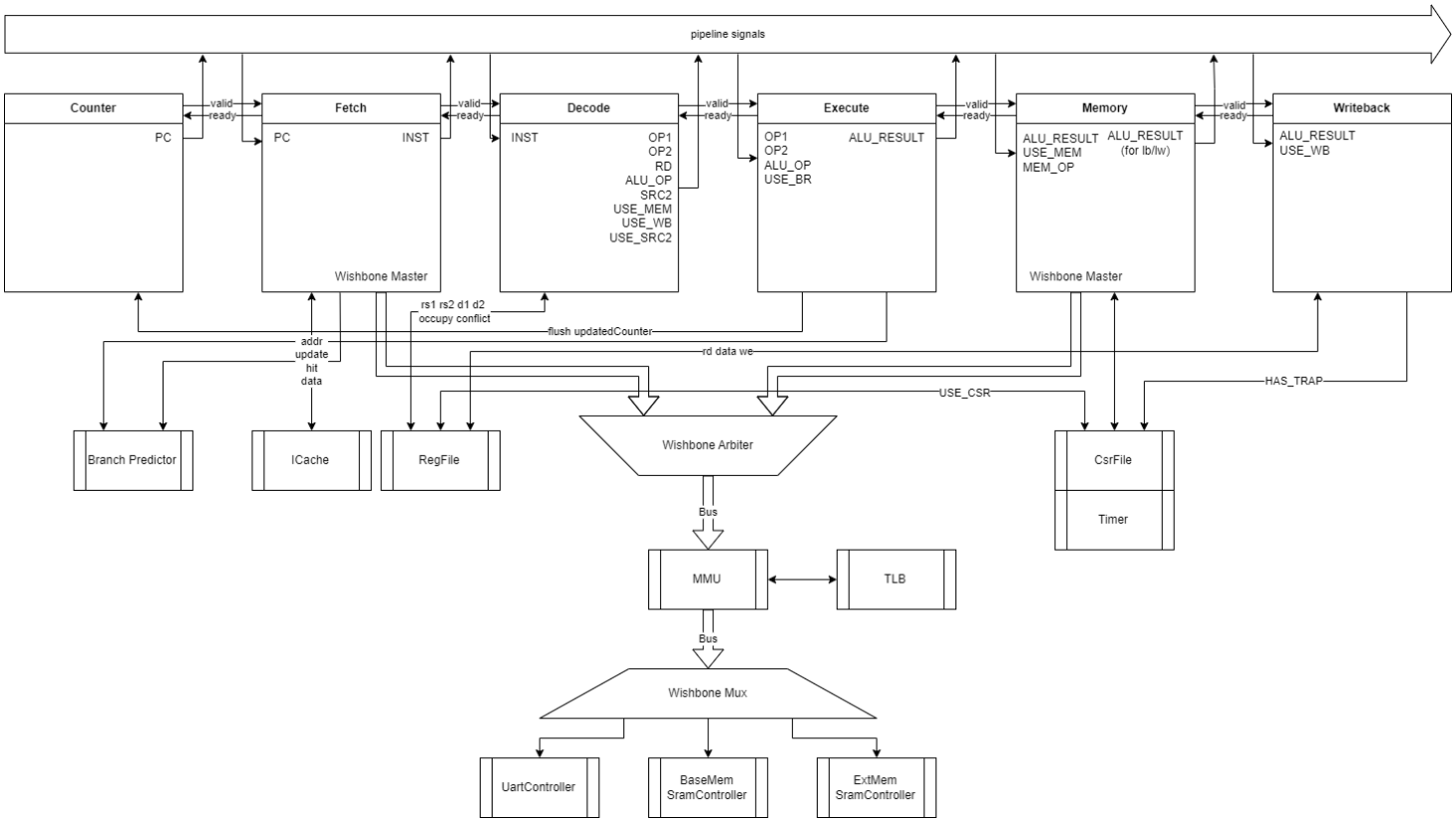
- SystemVerilog 仓库
- SpinalHDL 仓库

使用

```
sbt.bat
runMain pipeline.SimpleCPUSim # 运行仿真
runMain pipeline.SimpleCPUVerilator # 生成 Verilog 代码
```

将生成文件 `hw/gen/SimpleCPU.v` 放入 vivado 工程中即可。

CPU 结构图



功能实现

1. 除 FENCE 外完整的 RV32I 指令集
2. Zicsr 和 Zifencei 扩展指令集
3. M/S/U 特权态及中断异常处理
4. 虚拟内存支持
5. 计时器
6. 分支预测、iCache 和 TLB 等性能优化
7. B 扩展指令集中的 SBSET、XNOR 和 PCNT 三条指令

模块功能和实现

ALU

简单 ALU 实现，支持

`ADD, SUB, AND, OR, XOR, SLL, SRL, SRA, LT, LTU, GE, GEU, EQ, NEQ, SBSET, XNOR, PCNT` 运算

RegFile

一个 32 位的 32 个寄存器堆，读写冲突时优先进行写入，读取短路。

1. 使用 `occupied` 寄存器记录每个通用寄存器的占用情况，当一个指令需要写入某个寄存器时,如果该寄存器正被占用,就会产生冲突。
2. 在写入寄存器时,先检查 `occupy` 信号是否和要写入的寄存器 rd 相等，如果相等，说明这次写入正在占用该寄存器，可以直接将占用计数减 1。
3. 如果写入的寄存器 `rd` 正被其他指令占用(即 `occupied(rd) > 0`)且不是当前指令占用的寄存器，则会产生冲突。
4. 当发生冲突时，会将 `conflict1` 和 `conflict2` 标志置为 `true` 。在后续的指令执行中，可以根据这两个标志来判断是否需要暂停或者重新执行指令。
5. 在发生异常时， `clearOccupied` 信号会被置位,此时会将所有寄存器的占用状态清零，解决所有可能存在的冲突。

CsrFile

用于管理和操作控制和状态寄存器。

1. CSR 寄存器管理:

- 定义了 RISC-V 中所有的 CSR 寄存器，并提供了相应的读写接口。
- 每个 CSR 寄存器都是通过 BaseReg 类实现的，可以支持不同的读写操作(RW, RS, RC)。

2. 特权级管理:

- 维护当前的特权级(User, Supervisor, Machine)，并提供相应的读写接口。
- 根据特权级的不同，决定对某些 CSR 寄存器的访问权限。

3. 中断管理:

- 包含了 mtime 和 mtimecmp 寄存器，用于实现定时器中断。
- 管理并更新中断状态寄存器(mip)，当发生中断时自动更新相应的位。

4. 地址映射:

- 对于某些 CSR 寄存器(如 mstatus, mip, mie)，提供了一个 getTrueAddr 函数，将访问的地址映射到真实的 CSR 寄存器地址。

5. 委托管理:

- 提供了 getDelegation 函数，根据异常原因决定是否将异常委托给更低特权级处理。

6. 读写访问:

- 提供了一系列读写 CSR 寄存器的函数，如 readXMtvec、writeXcause 等，封装了具体的读写逻辑。
- 在写入 mstatus 寄存器时，根据特权级的不同，自动更新相应的位(如 xIE、xPIE)。

iCache

指令缓存功能， 32 位、32 行、2 路组相联，采用 LRU 替换算法，通过 ages 寄存器维护每个数据块的使用时间信息。

1. 缓存结构:

- 使用 indexWidth 位来索引缓存行,每个索引存储 waySize 个数据块。
- 每个数据块存储 dataWidth 位的数据,以及相应的标记(tag)。
- 还保存了每个数据块的有效位(valid)和替换算法(LRU)所需的年龄位(age)。

2. 读取缓存:

- 根据输入地址(addr)查找缓存,如果命中(hit)则返回缓存数据。
- 同时更新相应数据块的年龄位。

3. 写入缓存:

- 如果缓存可写(writable)且有效,则根据输入数据(dataIn)更新缓存数据和脏位(dirty)。但是为了过测试方便我们最后没用D-Cache。

4. 更新缓存:

- 可以通过 `update` 端口更新指定地址的缓存数据和标记。
- 更新时同时更新年龄位。

5. 刷新缓存:

- 支持通过 `flushAll` 信号全部清空缓存有效位。

MMU

- 功能:
  - 提供地址转换功能,将虚拟地址转换为物理地址。
  - 处理页面错误(page fault)异常。
  - 可选择性地启用 TLB (Translation Lookaside Buffer) 以提高地址转换效率。
- 实现:
  - 创建一个 PageTableVisitor 组件实例,用于完成实际的页表遍历和地址转换过程。
  - 通过 Wishbone 总线接口与外部设备进行交互。
  - 当收到有效的地址转换请求时(io.translate)，将虚拟地址传递给 PageTableVisitor，并根据转换结果更新物理地址。
  - 如果发生页面错误，将错误标志(io.pageFault)传递给请求方。
  - 在 ready 标志的控制下，在适当时机将请求从输入总线连接到输出总线。

1. PageTableVisitor

- 功能:
  - 实现页表遍历算法，将虚拟地址转换为物理地址。
  - 检查页表项的访问权限，并在发生页面错误时设置相应标志。
  - 可选择性地集成 TLB 以缓存页表转换信息，提高地址转换效率。
- 实现:
  - 定义 Sv32VirtualAddress 和 Sv32PhysicalAddress 两个 Bundle，分别表示 32 位 RISC-V 架构下的虚拟地址和物理地址。
  - 定义 Sv32PageTableEntry 结构，表示页表项的各个字段。
  - 使用多级页表结构(2 级)遍历页表，获取最终的物理地址。
  - 在遍历过程中，检查页表项的有效性和访问权限，并在发生页面错误时设置 io.pageFault 标志。
  - 如果启用了 TLB,则在遍历页表的同时检查 TLB 缓存，命中时直接返回物理地址；未命中时，在访问内存获取页表项后更新 TLB。
  - 通过 Wishbone 总线接口与外部存储器进行页表读取操作。
  - 在遍历完成后，更新 io.pa 输出端口,表示最终的物理地址。

2. TLB (Translation Lookaside Buffer)

- 功能:
  - 作为 MMU 的子模块,缓存页表转换信息,提高地址转换的效率。
- 实现:
  - 创建两个 Cache 组件,分别用于缓存一级页表和二级页表的转换信息。
  - 在页表访问过程中,同时检查 TLB 缓存:
    - 如果 TLB 命中,直接从 TLB 中获取物理地址信息,无需经过完整的页表查找。

        - 如果 TLB 未命中,则需要访问内存获取页表条目,并将其更新到 TLB 中。
    - 当收到 io.flushTLB 信号时,调用 Cache 组件的 flushAll 方法清空 TLB 中的所有缓存内容。 两个模块共同实现了基于分页机制的虚拟地址到物理地址的转换功能，并支持页面错误处理和 TLB 缓存优化。MMU 模块负责与外部设备的交互，而 PageTableVisitor 模块负责实际的页表遍历和地址转换过程。

Branch Predictor

使用两个主要的数据结构:

1. 分支历史表(Branch History Table, BHT):

- 大小为 BHTSize(2^PCRange.length)，每个条目存储 HistorySize 位的分支历史信息。
- 用于记录最近执行的分支指令的历史信息。

2. 模式历史表(Pattern History Table, PHT):

- 大小为 BHTSize(2^PCRange.length)，每个条目是一个大小为 2^HistorySize的数组。
- 每个数组元素存储一个 2 位的状态机计数器，用于保存对应历史信息的分支预测状态。 该分支预测器的功能如下:

1. 预测分支结果:

- 根据当前程序计数器 (PC) 的指定位 (PCRange) 查找 BHT 中的历史信息。
- 使用历史信息索引 PHT，得到相应的状态机计数器。
- 状态机的最高位即为分支预测结果(prediction)。

2. 更新预测表:

- 当收到分支更新信息(io.update.valid)时,根据更新 PC 更新 BHT 中的历史信息。
- 同时更新 PHT 中对应的状态机计数器,根据实际分支结果(update.taken)增加或减少计数器值。

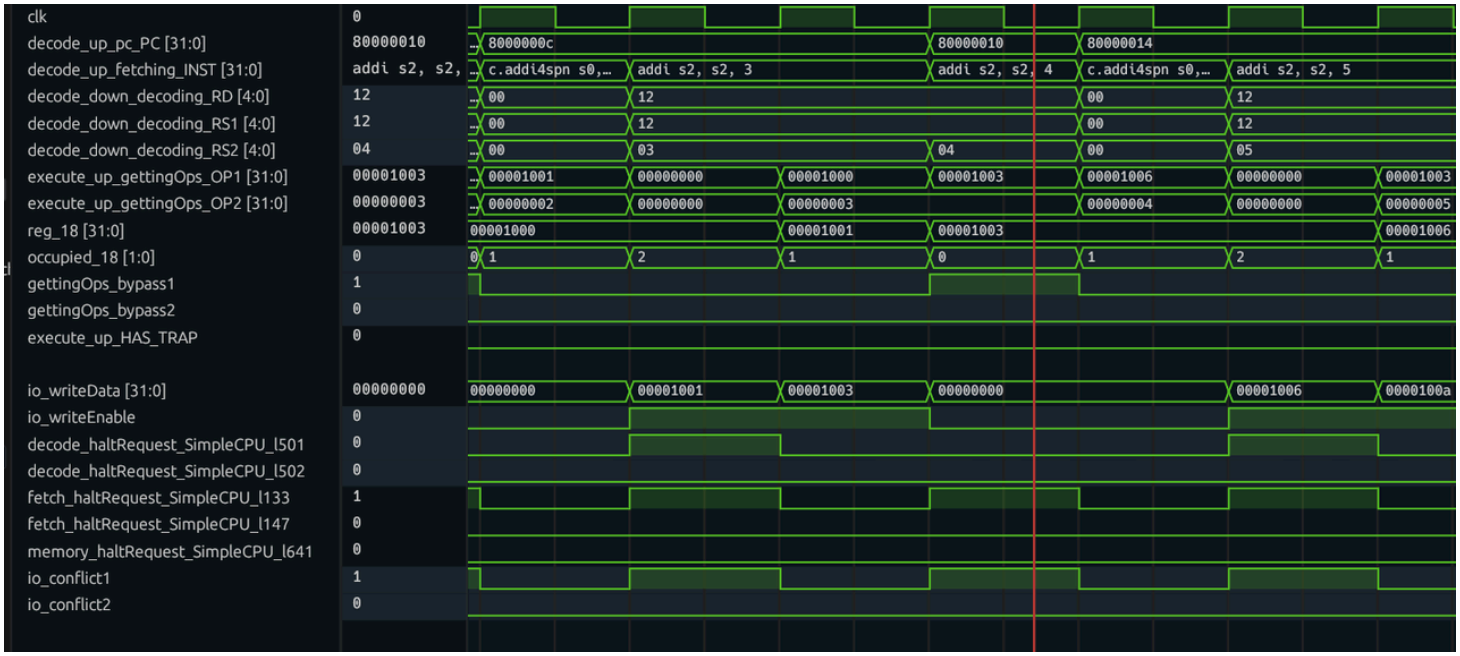
数据前传

实现逻辑

当当前位于EXE段的a指令的ALU_result就是要写入寄存器的值，且a的目标寄存器是当前位于ID阶段b指令的其中一个源寄存器时，触发数据前传，将结果直接作为b指令EXE阶段的操作数，不需等待a指令写回后再执行b。 数据冲突例子

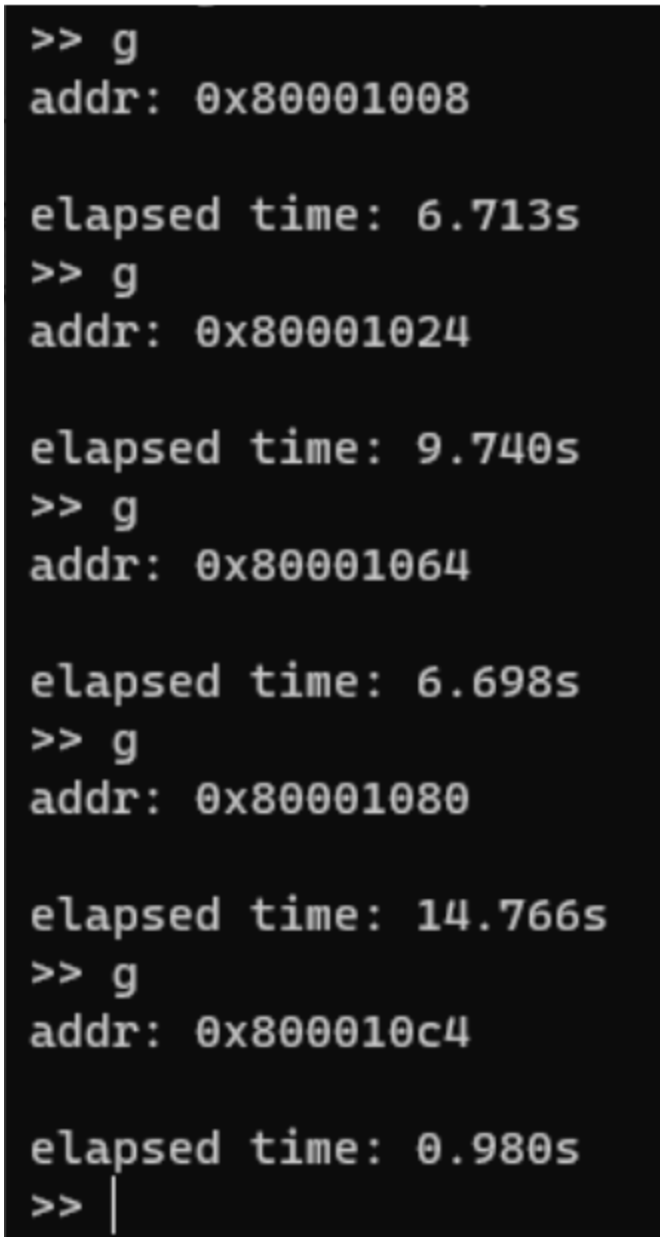
```
lui s2,1
addi s2,s2,1
addi s2,s2,2
addi s2,s2,3
addi s2,s2,4
addi s2,s2,5
```

在第4条指令结束后，s2寄存器的值应为0x1006,。 图中红线所示周期，第4条指令处于 `EXE` 段，第5条指令处于 `ID` 段，4的RD等于5的RS1，bypass1为True，触发了数据前传，所以下一个周期5直接获得了操作数0x1006。



性能

启用所有功能的性能测试结果如下：



I-Cache

版本	PTB	DCT	CCT	MDCT	CRYPTONIGHT
有 iCache	6.713	9.74	6.698	14.766	0.98
无 iCache	20.133	11.082	16.161	21.463	1.163

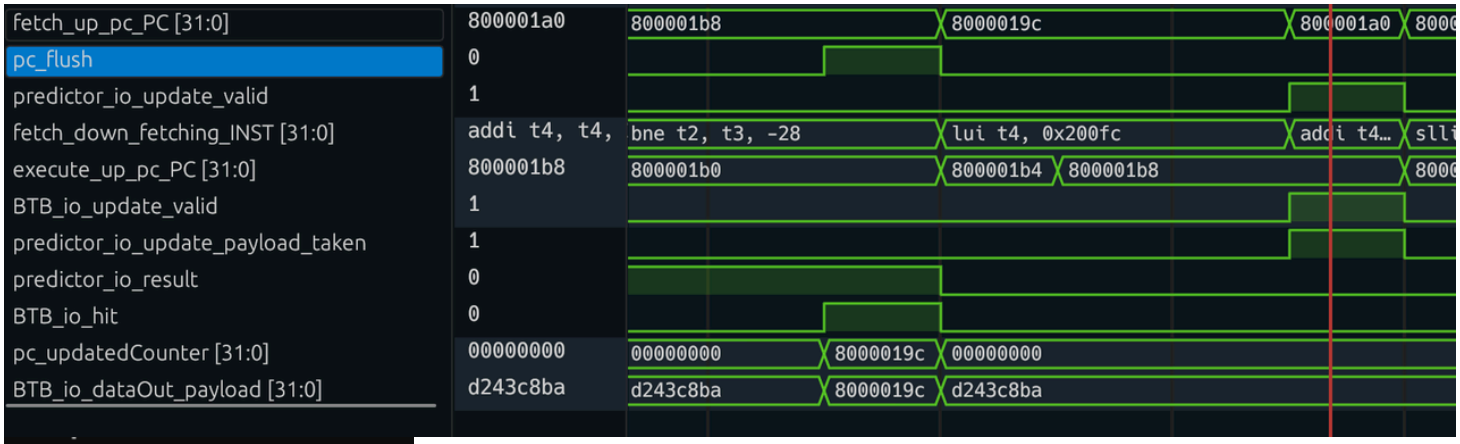

```
>> g
addr: 0x80001008

elapsed time: 20.133s
>> g
addr: 0x80001024

elapsed time: 11.082s
>> g
addr: 0x80001064

elapsed time: 16.161s
>> g
addr: 0x80001080

elapsed time: 21.463s
>> g
addr: 0x800010c4

elapsed time: 1.163s
>>
```

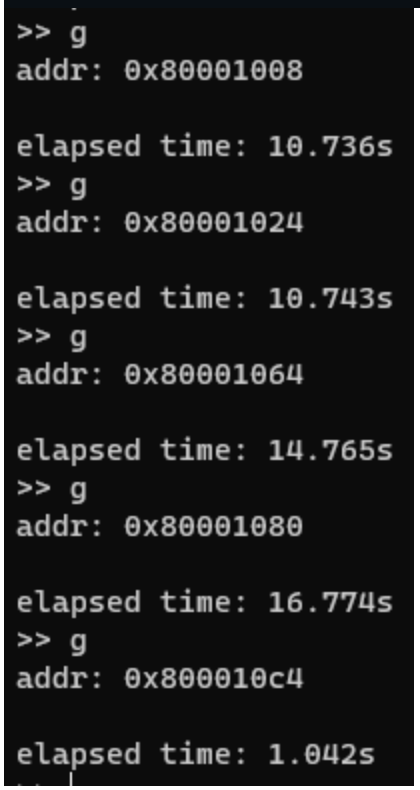
如图所示是一个分支预测的信号示例。在没有命中时，如0x80000a04位置，fetching阶段就要去使用wishbone去内存里取数据，当取到以后，就可以用这个取到的值去更新icache，也就是如图所示update_valid信号为高的时刻。红线标出的时刻是icache命中的情况，此时icache的hit位是高，代表命中，icache可以在单周期内完成读取，此时fetching阶段就不再使用wishbone了，数据由icache_io_dataout给出。使不使用wishbone这个决策也是在同一个周期内完成的，可以由useWb信号看出。使用icache以后原本的两周期访存变成单周期。

分支预测

版本	PTB	DCT	CCT	MDCT	CRYPTONIGHT
有分支预测	6.713	9.74	6.698	14.766	0.98

版本	PTB	DCT	CCT	MDCT	CRYPTONIGHT
无分支预测	10.736	10.743	14.765	16.774	1.042





如图所示是一次分支预测并且预测正确的示例。在fetch段读到bne指令以后，根据BTB查询结果，发现此处命中，代表确实是一条分支指令，同时BTB给出了如果分支为taken时的地址。此时结合predictor给出的结果，也就是高，代表predictor认为此处会跳转，那么此时fetch段就发出跳转信号给counter段，pc_flush被拉高，下一个时刻fetch段拿到的pc就是跳转后的了。同时这条跳转指令还需要继续往下流，当流到exe段的时候，alu计算得出此时确实应该跳转，BTB和predictor的update都被拉高，本条指令的实际结果被更新进去，同时由于预测正确，此时不再跳转，如果发现预测不正确，哪此时流水线将被冲刷并且发生跳转。

TLB

版本	PTB	DCT	CCT	MDCT	CRYPTONIGHT
有 TLB	6.713	9.74	6.698	14.766	0.98
无 TLB	6.711	9.717	6.716	14.765	0.979


```
>> g
addr: 0x80001008

elapsed time: 6.711s
>> g
addr: 0x80001024

elapsed time: 9.717s
>> g
addr: 0x80001064

elapsed time: 6.716s
>> g
addr: 0x80001080

elapsed time: 14.765s
>> g
addr: 0x800010c4

elapsed time: 0.979s
```

如图所示是一个TLB的例子。图中io_input是mmu输入的wishbone，也就是连接CPU方向，io_output是输出的wishbone，也就是连接内存方向。第一次访存时，虚拟地址0x80001000对应的页表没有被查过，因此TLB里没有相应记录。也就是TLB1和TLB0的hit全是低。mmu必须进行三次总共六周期访存，其中前两次为查询页表。当对应级页表被查到以后，对应TLB的update也被置为高，TLB被更新。在第二次访问0x80001004地址的时候，由于两级页表都已经遇到过，所以TLB都命中，地址翻译直接根据TLB的结果在组合逻辑内完成，仅在一个周期内地址翻译就被完成了。此时mmu总共只需要进行三周期处理，效率提升了一倍。

中断异常处理

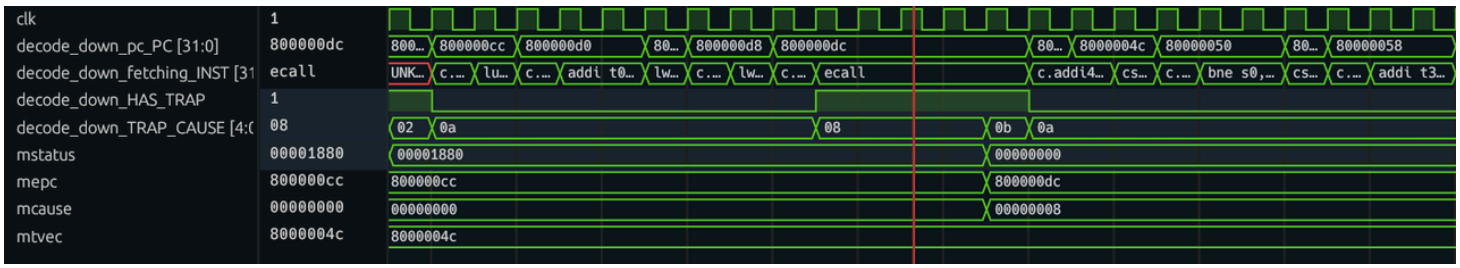
中断异常处理部分，我们实现了U，S，M三态，以及运行ucore所需的CSR寄存器。

在IF，ID，MEM阶段都可能发生异常，同时，我们选择在ID阶段判断时钟中断是否发生。
每条指令携带两个相关的payload，标志是否有中断异常以及类型。当流水线上某条指令有中断异常时，跳过其在后续流水线阶段的执行，以及后续指令的执行。

在MEM阶段进行CSR寄存器的读写，这样同一时刻只有一条指令进行CSR寄存器的读写，不用额外进行RAW冲突的处理。
对于mtime和mtimecmp寄存器的读写，我们选择在访问wisbone总线前特判，重定向对这两个寄存器的读写请求。 值得注意的是，S态寄存器sstatus，sip，sie分别是对应m态寄存器的子集，对应位的值需要相等，所以我们同样重定向了对这三个s态寄存器的读写到m态寄存器，保证了一致性。

在WB阶段进行中断异常的处理，因为此时流水线上只有这一条有效指令，可以方便地清除寄存器写的占用。

进行异常处理时，根据文档对CSR寄存器进行读写即可。



如图，在 `ecall` 指令的 ID 段触发了异常，随后在其 WB 段相应 csr 寄存器被写入，PC 也正常跳转至 `mtvec` 中储存的地址。

思考题

1. 流水线 CPU 设计与多周期 CPU 设计的异同？插入等待周期（气泡）和数据旁路在处理数据冲突的性能上有什么差异。

- 相同之处：都将一条指令划分为5段顺序执行
- 不同之处：多周期cpu同一时刻只有一条指令在执行。而流水线cpu同一时刻可能有多条指令一起执行，需要处理冲突，因而更加复杂，但也带来了性能的提升。
- 性能差异：数据旁路使得遇到数据冲突时不必暂停流水线数个周期，因此性能更高。

2. 如何使用 Flash 作为外存，如果要求 CPU 在启动时，能够将存放在 Flash 上固定位置的监控程序读入内存，CPU 应当做什么样的改动?

reset后需要将PC的初始值设为Flash监控程序初始位置。

3. 如何将 DVI 作为系统的输出设备，从而在屏幕上显示文字?

与VGA类似，为DVI分配一块独立的显存，将显存控制器接入到CPU中。CPU按照DVI时序将数据输出。

4. （分支预测）对于性能测试中的 3CCT 测例，计算一下你设计的分支预测在理论上的准确率和性能提升效果，和实际测试结果对比一下是否相符。

```
80001064 <UTEST_3CCT>:
    /*  控制指令冲突测试(3)
     *  这段程序有大量控制冲突。
     *  执行需要至少 256M 指令。
     */
UTEST_3CCT:
    lui t0, %hi(TESTLOOP64)         // 装入64M
80001064:   040002b7            lui t0,0x4000
.LC2_0:
    bne t0, zero, .LC2_1
80001068:   00029463            bnez    t0,80001070 <UTEST_3CCT+0xc>
    jr ra
8000106c:   00008067            ret
.LC2_1:
    j .LC2_2
80001070:   0040006f            j   80001074 <UTEST_3CCT+0x10>
.LC2_2:
    addi t0, t0, -1
80001074:   fff28293            addi    t0,t0,-1 # 3ffffff <INITLOCATE-0x7c000001>
    j .LC2_0
80001078:   ff1ff06f            j   80001068 <UTEST_3CCT+0x4>
    addi t0, t0, -1
8000107c:   fff28293            addi    t0,t0,-1
```

对于bne指令(0x08: bne t0, zero, .LC2_1)，由于t0寄存器不断递减,理论上这个分支只会预测错误3次(从1降到0)。对于两个j指令(0x10: j .LC2_2和0x18: j .LC2_0)，它们的跳转目标地址是固定的,理论上只会预测错误2次。所以总共的预测错误次数是3+2+2=7次。这个测例总共执行6410241024次迭代。所以分支预测准确率 = $1 - 7/6410241024 = 99.99$。因此对于有分支预测的情况，所有指令可以近似看作只执行一次，没有的话，每条指令则会产生2个气泡，因此根据指令行数来看，性能提升约为 $1 - \frac{4}{10} = \frac{6}{10}$。 实际测试中，性能提升为 $1 - \frac{6.698}{14.765} = 0.546$，符合计算预期结果。

5. （缓存）对于性能测试中的 4MDCT 测例，计算一下你设计的缓存在理论上的命中率和性能提升效果，和实际测试结果对比一下是否相符。

```
80001080 <UTEST_4MDCT>:
    /*  访存相关数据冲突测试(4)
     *  这段程序反复对内存进行有数据冲突的读写。
     *  需要至少执行 192M 指令。
     */
UTEST_4MDCT:
    lui t0, %hi(TESTLOOP32)          // 装入32M
80001080:   020002b7            lui t0,0x2000
    addi sp, sp, -4
80001084:   ffc10113            addi    sp,sp,-4
.LC3:
    sw t0, 0(sp)
80001088:   00512023            sw  t0,0(sp)
    lw t1, 0(sp)
8000108c:   00012303            lw  t1,0(sp)
    addi t1, t1, -1
80001090:   fff30313            addi    t1,t1,-1
    sw t1, 0(sp)
80001094:   00612023            sw  t1,0(sp)
    lw t0, 0(sp)
80001098:   00012283            lw  t0,0(sp)
    bne t0, zero, .LC3
8000109c:   fe0296e3            bnez    t0,80001088 <UTEST_4MDCT+0x8>
    addi sp, sp, 4
800010a0:   00410113            addi    sp,sp,4
    jr ra
800010a4:   00008067            ret
```

理论上未缓存时需要 30 个周期，启用 iCache 后可以降低至 14 个周期，加速比为 $1 - \frac{14}{30} = 0.533$。

- 实际测试中，性能提升为 $1 - \frac{14.766}{21.463} = 0.312$。

6. （虚拟内存）考虑支持虚拟内存的监控程序。如果要初始化完成后用 G 命令运行起始物理地址 0x80100000 处的用户程序，可以输入哪些地址？分别描述一下输入这些地址时的地址翻译流程。

可以输入0x0，0x80100000，流程分别如下：

- 查根页表得到二级页表，通过二级页表映射 [0x00000000, 0x002FFFFF] -> [0x80100000, 0x803FFFFF] 翻译得到 0x80100000

- 查根页表得到二级页表，通过二级页表映射 [0x80100000, 0x80100FFF] -> [0x80100000, 0x803FFFFF] 翻译得到 0x80100000

7. （异常与中断）假设第 a 个周期在 ID 阶段发生了 Illegal Instruction 异常，你的 CPU 会在周期 b 从中断处理函数的入口开始取指令执行，在你的设计中，b - a 的值为?

我们在WB阶段处理异常，此时为a+3 周期，则下一个周期会从中断处理函数的入口处开始取指令执行。b-a的值为4

分工

姓名	任务
张瀚宸	五级流水线框架，分支预测，iCache，TLB，页表
黄宸宇	RV32I指令，额外指令
秦晨阳	异常中断