计算机组成原理大作业 Group 75

组员：尹绍沣、陈煜翔、濮成风

CPU 时钟主频率为 `50 MHz`。

（一）实验目标

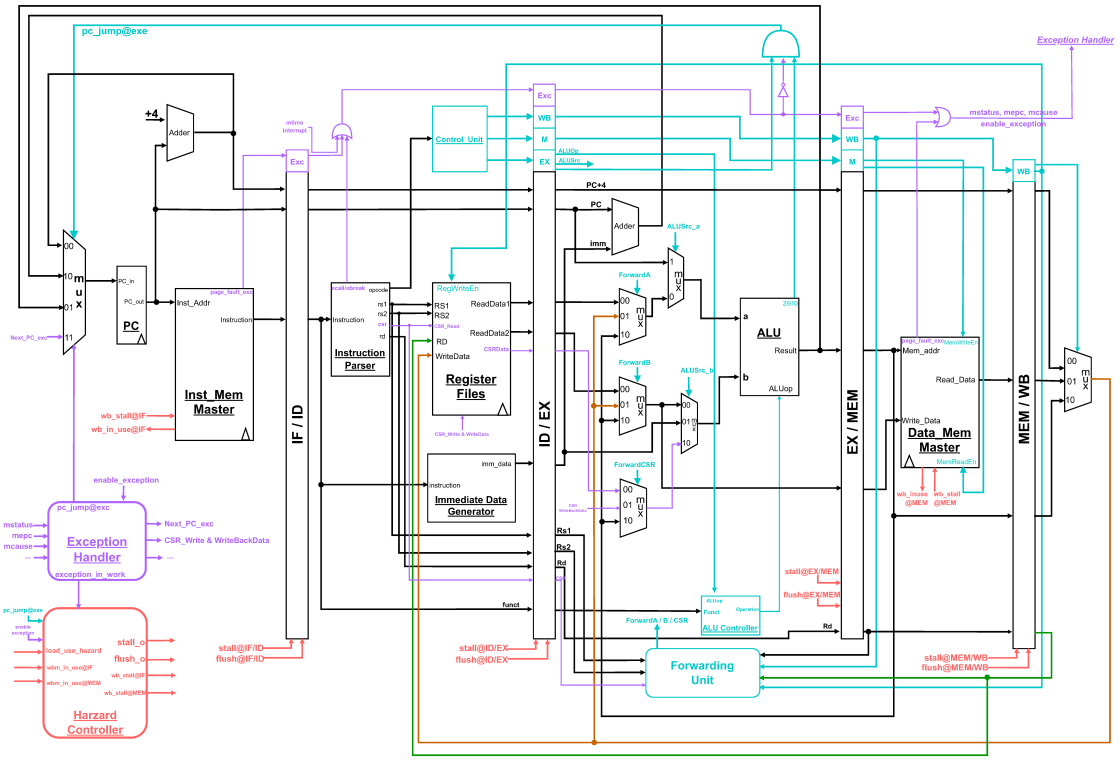
完成功能：

- 功能类：**五级流水线** RISC-V 32 处理器；**中断、异常**处理；**页表**（用户态虚拟地址翻译）
- 性能类：**I-Cache、D-Cache**（Writeback）、**页表的 TLB**
- 外设扩展：**VGA、Flash** 等

（二）实验内容

0. CPU 结构图和数据通路图

流水线部分数据通路图



信号含义：

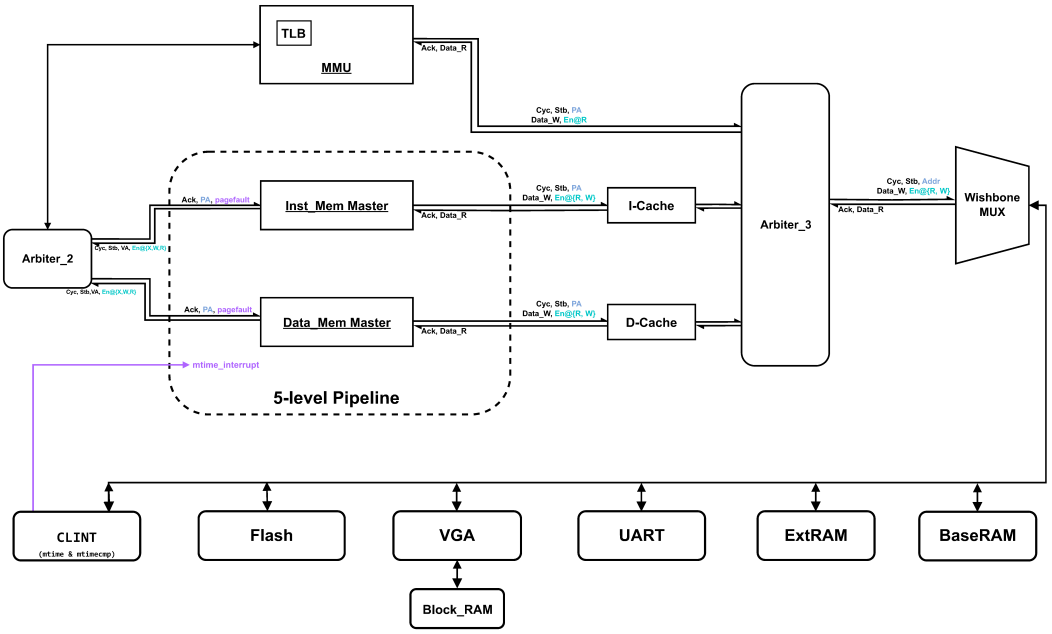
- 黑色的信号代表数据，例如指令、地址、读/写的数据、ALU 的运算
- 蓝色的信号代表控制信号，例如读/写使能、数据前传的选择
- 绿色、棕色的信号分别是 Writeback 的地址、数据
- 红色的信号代表冲突的处理信号，控制段寄存器的 Flush / Stall
- 紫色的信号代表中断异常的处理信号，包括 CSR 寄存器的读写，以及中断、异常信息

模块含义：

- Inst_Mem_Master 和 Data_Mem_Master 分别负责取指、读写内存（操作外设），它们会与 MMU 和 Wishbone 总线连接，完成数据通信。

- Forwarding Unit 控制数据前传，解决 Load-Use 之外的数据冲突
- Exception Handler 负责接收、生成异常信号，并生成相应的 CSR 寄存器读、写信号
- Harzard Controller 负责解决控制冲突（包括异常引发的跳转）、Load-Use 数据冲突、以及访问 Wishbone 引发的流水线暂停
- 其余模块的含义如图中标注所示

CPU 整体连线图



- MMU 负责翻译虚拟地址，其中 TLB 是快表
- I-Cache、D-Cache 分别对应指令缓存、数据缓存
- 仲裁器解决结构冲突
- CLINT 是专用于实现时钟中断的两个 MMIO 寄存器（见监控程序的 README）
- 其余模块的含义如图中标注所示

1. 五级流水线和冲突处理

五个流水线段

流水线使用了经典的五级流水线设计，分为`IF`、`ID`、`EXE`、`MEM`、`WB`五个阶段

`IF`阶段从会根据当前pc从内存读取指令，并且在下一条的时候更新pc。

`ID`阶段将`IF`阶段的指令进行译码，得到`EXE`、`MEM`、`WB`阶段需要的值。同时如果遇到异常，会产生相应控制信后向后传递。这一阶段也会读CSR寄存器一些特定的字段进行异常判断。

`EXE`阶段根据`ID`阶段的操作数和操作码进行计算，同时将得到的数据进行前传解决数据冲突。`EXE`阶段若出现了条件跳转指令，会将pc传递给pc寄存器进行跳转，同时冲刷后面的指令。

`MEM`阶段会根据前面的结果进行读写内存操作，同时传递需要写回寄存的信号，将读出的数据进行前传。

`WB`阶段写回寄存器、进行数据的前传，同时也会写CSR寄存器。

冲突处理

`hazard`控制器会根据每个阶段是否有指令、流水线中是否出现异常、是否出现跳转，给每个阶段暂停或冲刷。

数据前传会前传`MEM`阶段的alu计算结果、前传`WB`的写回寄存器结果到`EXE`阶段。其逻辑为：若`EXE`阶段需要`rs1_exe`寄存器的数据，如果当前在`MEM`阶段的指令需要写回寄存器，并且其写回寄存器的编号`rd_mem`等于`rs1_exe`，那么将数据前传给`EXE`；否则，如果当前在`WB`阶段的指令需要写回寄存器，并且其写回寄存器的编号`rd_wb`等于`rs1_exe`，那么将数据前传给`EXE`；否则不前传。对`rs2_exe`进行同样的处理。

而对于load_use情形，如果`rs1_exe`等于`rd_mem`或者`rs2_exe`等于`rd_mem`并且是读操作，那么会在`MEM`阶段塞入一个气泡，让数据到达`EXE`阶段再继续执行指令。

数据冲突的例子

对于如下三条指令:

`add x3, x1, x2`

`add x4, x1, x2`

`add x1, x3, x4`

这三条指令在cpu上执行时，第一条指令在`WB`阶段有`rd_wb`等于`rs1_exe`，于是会进行一次数据前传。第二条指令在`MEM`阶段有需要写回的寄存器`rs2_exe`，也会进行前传。**我们用表格表示不同阶段的波形，具体信号如下**:

阶段	指令	rd	rs1	rs2	mem_en	wb_en	wb_value
exe	add x1, x3, x4	x1	x3	x4	0	1	-
mem	add x4, x1, x2	x4	x1	x2	0	1	mem_res
wb	add x3, x1, x2	x3	x1	x2	0	1	wb_res

此时`(wb_men_en & (wb_rd==exe_rs1)) == 0`，但是`(wb_wb_en & (wb_rd==exe_rs1)) == 1`，有alu的第一个操作数被赋值为`wb_res`。

此时`(mem_men_en & (wb_rd==exe_rs2)) == 1`，有alu的第二个操作数被赋值为`mem_res`。

2. 异常处理模块

异常处理会根据当前的特权级、`mie`来决定是否需要处理中断信号。当启用中断时，流水线在`mem`阶段会根据异常信号进行如下处理:

1. 将当前pc保存到`mepc`中
2. 将异常原因写到`mcause`中
3. 将`mstatus.mpp`设置为当前的特权（`M`）
4. 根据`mtvec`，写`pc`，跳转到异常处理程序处
5. 禁用全局中断

之后异常返回时，会将`mstatus.mpp`设置为`U`态，根据`mepc`跳转到相应的`pc`处。

3. 页表模块（MMU & TLB）

为了实现虚拟地址翻译，我们需要将 IF 和 MEM 段的翻译请求通过仲裁器仲裁，发送到 MMU（Page Table Walker）。

MMU 收到请求后，依次进行如下操作（TLB miss 的情况）：

- 检查是否需要翻译
- 通过页表基址和 vpn 计算出页表项地址，通过 Wishbone 总线查询内存中的页表项（两级页表需要查两次）
- 根据 R/W/X 位和用户模式判断是否操作合法，不合法则触发 Page Fault
- 确认得到一个合法的页表项，加入 TLB；并通过查询得到的 ppn 和虚拟地址的 offset 计算物理地址
- 将物理地址发送回仲裁器，让 IF / MEM 段获得物理地址

其中，TLB 采用了直接映射，储存了 32 个快表项（将 vpn 的低 5 位作为 `tlbi`），如果 TLB hit，可以将上述过程简化，直接通过 TLB 储存的 ppn 计算出需要返回的物理地址，将 MMU 部分的翻译过程缩短到 1 周期。

具体实现如下：

MMU的状态包括`MMU_IDLE`、`MMU_READ`、`MMU_READ_PAUSE`、`MMU_DONE`、`MMU_WAIT`。其状态转移如下（若没写出转移条件，默认为保持不变）：

如果为`MMU_IDLE`，并且`need_translate==1`，那么此时需要进入翻译虚拟地址状态，并且需要从内存中读取页表项，有：

信号	条件	下一周期信号
state	tlb_hit	`MMU_IDLE`
state	!tlb_hit	`MMU_READ`
layer_idx	!tlb_hit	1
wb_stb_o	!tlb_hit	1
wb_adr_o	!tlb_hit	ptbr+vpn[1]*4
wb_sel_o	!tlb_hit	4'b1

其中`ptbr=={satp[19:0], 12'b0}`，`vpn[1]==vir_adr[31:22]`，`vpn[0]==vir_adr[21:12]`。

如果为`MMU_READ`，并且`wb_ack_i==1`，会依次进行以下判断:

1. `!pte_v | (!pte_r & pte_w)`：此时会产生页表异常，`state`更新为`MMU_DONE`，`page_fault_exc`设置为`1`。
2. `pte_r | pte_x`：
    1. 检查页表的权限：若写/读/执行权限不正确，产生页表异常。
    2. 检查是否对齐：若`layer_idx>0 && pte[19:10] != 0`，产生页表异常。
    3. 进行翻译：`ppn[1]<=pte[29:20]`，若`layer_idx>0`，则`ppn[0]<=vpn[0]`，否则`ppn[0]<=pte[19:10]`
    4. `state`更新为`MMU_DONE`
3. 否则，需要查询下一级，若`layer_idx>0`，则`layer_idx<=layer_idx-1`，`state`变为`MMU_READ_PAUSE`，否则产生页表异常。

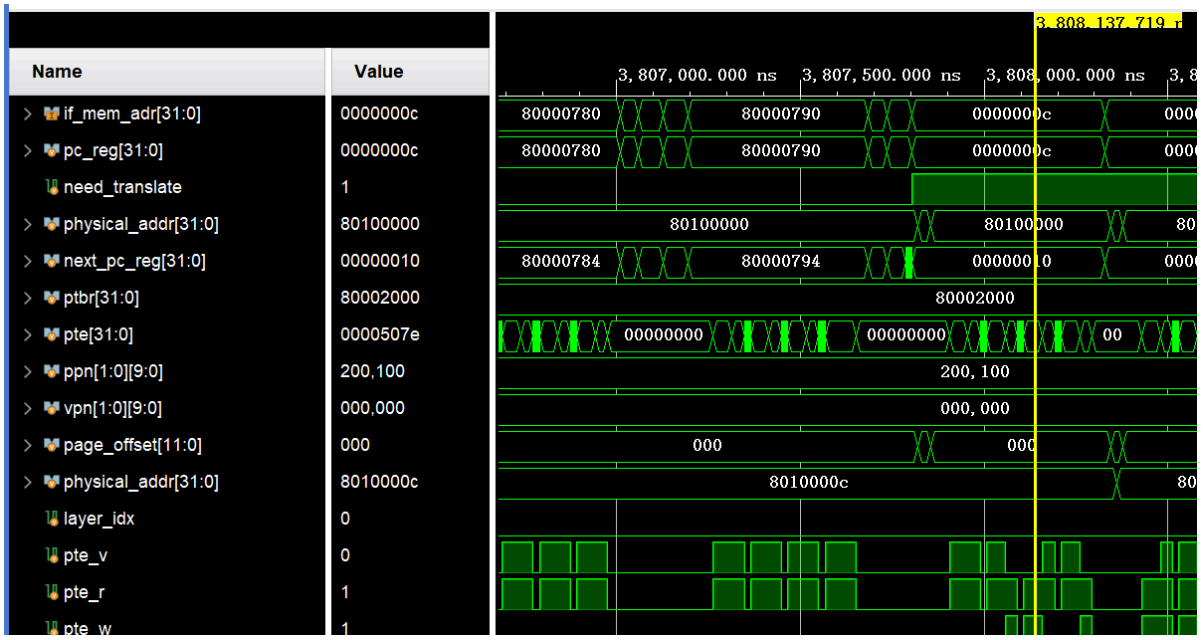
如果为`MMU_READ_PAUSE`，信号更新为：

信号	更新为
state	MMU_READ
wb_adr_o	{pte[29:10], 12'0} + vpn[layer_idx] * 4

如果为`MMU_DONE`，信号更新为：

信号	条件	更新为
state	-	MMU_WAIT
wb_stb_o	-	0
tlb_arrays[tlb_index]	!tlb_clear	{1'b1, ppn[1], ppn[0]}
tlb_arrays_tag[tlb_index]	!tlb_clear	tlb_tag

其部分波形图如图所示:



4. 缓存模块（I-Cache & D-Cache）

I-Cache

采用了两路组相联结构，缓存行数据大小 128 bits，设置16个缓存行，使用 **LRU 替换策略。**

采用如下缓存行结构：

TAG_BITS	VALID_BIT	DATA_BITS	COUNTER_BIT
24	1	128	1

其中VALID_BIT标记缓存行是否有效，高电位表示有效。

COUNTER_BIT用于指示LRU进行替换，低电位表示最近用过。

从主存取数据

向wishbone连续发送4个请求，读取连续四个字。

替换操作

由于是两路组相联，LRU的实现十分简单：

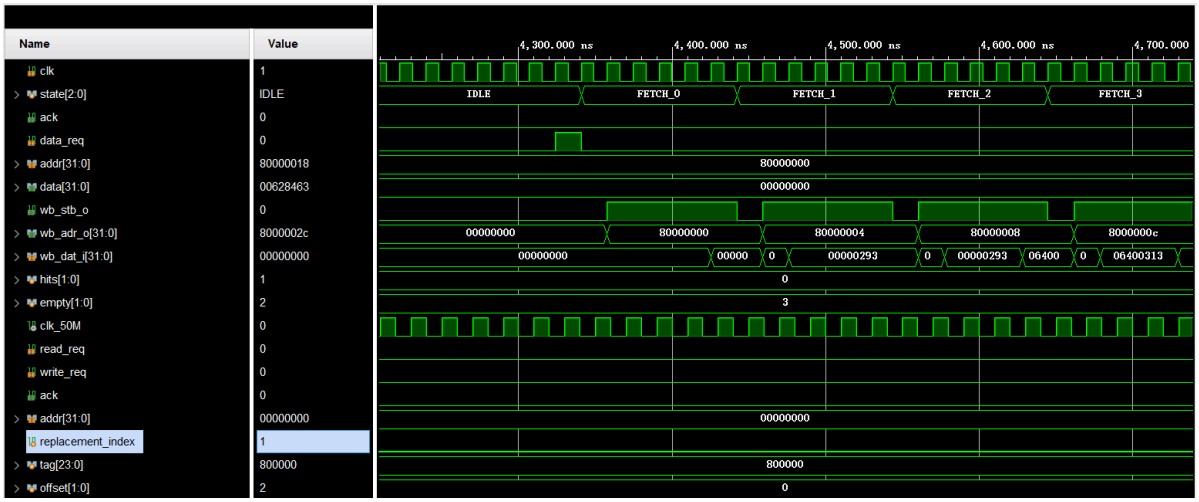
1. 每次写入或者读取有效数据时，将对应的counter置为0，另一路对应counter置为1。
2. 替换时，如果无空行，使用counter大的进行替换。

查询是否命中

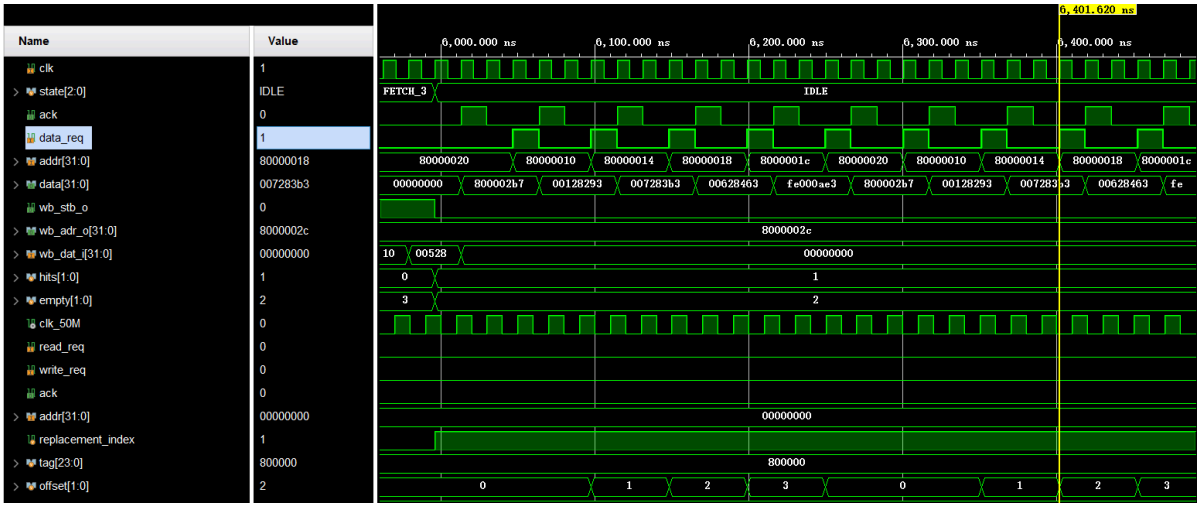
与tag进行比较同时确认valid位，如果同时满足，则命中，否则不命中。

关键信号波形

主存取数据



缓存命中



D-Cache

采用了两路组相联结构，实现Write-Back cache，缓存行数据大小 32 bits，设置32个缓存行，使用 **LRU 替换策略。**

采用如下缓存行结构：

TAG_BITS	DIRTY_BIT	VALID_BIT	DATA_BITS	COUNTER_BIT
25	1	1	32	1

读操作

如果缓存没有命中，区别于I-Cache，只向wishbone发送1个请求。

替换操作

基本与I-Cache相同。增加对DIRTY_BIT逻辑的处理，如果DIRTY，替换时要将此行写回。

写操作

从主存中读取相应位置一个字的数据，修改其中字段，直到被替换时才写入主存。

查询是否命中

与I-Cache相同。

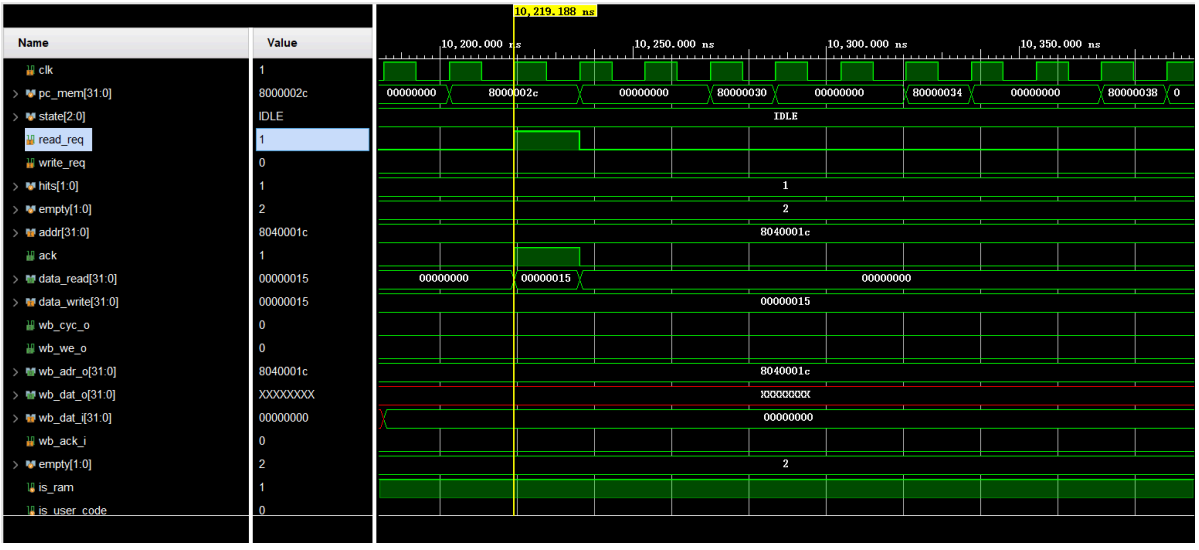
其他

为确保功能测试可以通过，特判是否为用户程序或者外设地址，如果是，则不缓存，直接写入主存。

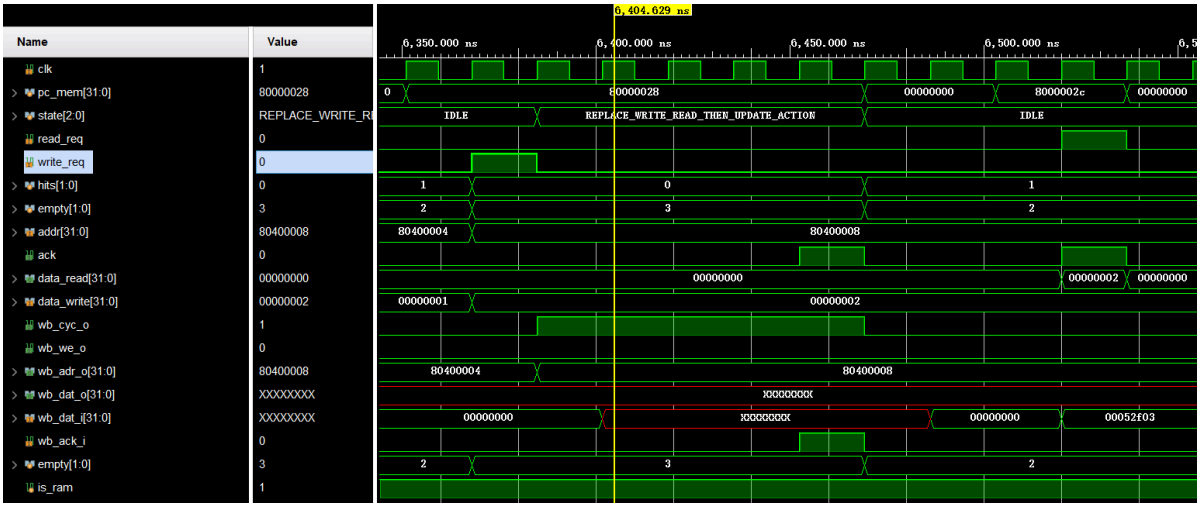
关键信号波形

判断为外设信号，直接读取/写入

缓存命中



缓存写入



5. 外设模块（Wishbone Slaves）

BaseRAM、ExtRAM、UART 均采用小作业已实现的 Controller 作为 Wishbone Slave。

5.1 VGA

VGA包括vga_controller和bram两个模块。vga_controller使用了MMIO，用户可以根据其地址写入VGA上对应的像素。vga_controller会通过wishbone得到请求的写数据，并将其写入到对应的bram地址。而vga使用`50M`的时钟，会依次读取bram中的数据，设置相应像素的RGB值。

bram深度为8bit，大小为75KB，能够显示800*600大小的黑白位图。

VGA 使用说明（**见仓库中的 `tests/3_VGA` 目录**）：

- 利用我们仓库中的 `tests/3_VGA/convert.py` 将 png 转换为 bin 文件
- 上传脚本转换好的 bin 文件，写入 Flash
- 运行监控程序、写入用户程序 `tests/3_VGA/test_vga.s`、运行用户程序

VGA大小为800*600像素，每个像素为黑色或者白色。其信息被储存在了75KB的BRAM中，每一个byte中的8个bit表示了一行中连续的8个像素的黑白情况（0表示黑，1表示白）。

VGA会以50MHz的频率刷新，其读BRAM时序如下：

信号	下一周期信号
hdraw	hdata < HSIZE ? hdata : HSIZE - 1
vdraw	vdata < VSIZE ? vdata : VSIZE - 1
hdata	hdata == HMAX-1 ? 0 : hdata + 1
vdata	(hdata == HMAX-1 ? (vdata == VMAX-1 ? 0 : vdata+1)) : vdata
bram_addr	(hdraw + vdraw * HSIZE)/8

其中，`HSIZE==800`，`VSIZE==600`，`HMAX=1040`，`VMAX==600`，。

由于BRAM可以同时读写，读使能被设置成了1。对于写VGA，其地址被MMIO到了0x03000000~0x030fffff中，并且接到了wishbone总线上。VGA的写时序如下：

VGA有状态：`STATE_IDLE`和`STATE_DONE`

如果为`STATE_IDLE`，转移为：

信号	条件	新的信号
state	wb_cyc_i & wb_stb_i	`STATE_DONE`
state	!(wb_cyc_i & wb_stb_i)	state
bram_wen	wb_cyc_i & wb_stb_i	1
bram_wen	!(wb_cyc_i & wb_stb_i)	0
bram_addr	wb_cyc_i & wb_stb_i	(hdata + vdata * HSIZE)/8
bram_addr	!(wb_cyc_i & wb_stb_i)	-

其中，`hdata`等于wishbone总线上的`data[19:10]`，`vdata`等于wishbone总线上的`data[9:0]`。

如果为`STATE_DONE`，转移为：

信号	条件	新的信号
state	-	`STATE_IDLE`
bram_wen	-	0

5.2 Flash

分配一段地址空间映射到 Flash（我们用了 `0x0400_0000 ~ 0x047F_FFFF`），然后让 Flash Controller 作为 Wishbone Slave。

然后 Flash Controller 实现了只读逻辑。由于 Flash 需要的读延迟较高，所以不能在较少周期数完成数据的读，故我们采用一个计数器来记录当前延迟的周期，保证两次信号变化之间的时间达到要求，来保证不发生 Flash 的时序违例。

我们采用了 16 位版本的 Flash。为了和 BaseRAM / ExtRAM 的读行为一致，要读一个字（32 位）的时候就分成两次读 16 位的操作。

5.3 CLINT

实现了时钟中断（见监控程序的 README）。

实现了两个 MMIO 寄存器的读写：mtime（64 位，可读写，表示当前时间）和 mtimecmp（64 位，可读写。表示下次时钟中断时间）。

除了读写寄存器操作，每个周期会让 mtime 加 1，并将其与 mtimecmp 比较，若 `mtime >= mtimecmp`，则触发时钟中断，向 CPU 的异常处理逻辑中直接发送 mtime_interrupt 信号。

（三）效果展示

1. 监控程序运行

以下程序通过 `ecall` 进行系统调用，输出字符 `O`，并且杀死了超时的死循环：

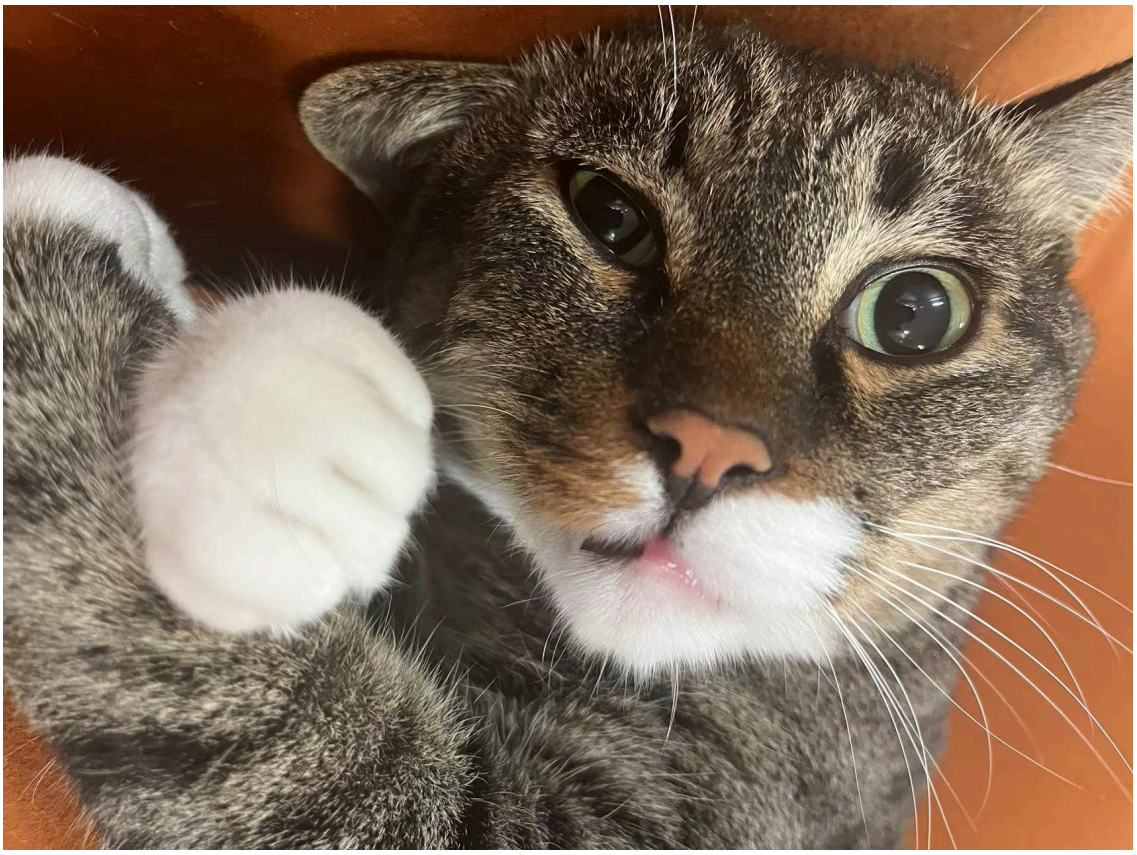
以下程序试图直接把数据写到无法访问的地址，产生了页表错误：

```
>> a
addr: 0x80100000
one instruction per line, empty line to end.
[0x80100000] li x1, 0x80400000
[0x80100004] li x2, 0x123
[0x80100008] sw x2, 0(x1)
[0x8010000c] ret
[0x80100010]
>> g
addr: 0x0
supervisor reported an exception during execution
  mepc: 0x00000008
  mcause: 0x0000000d
  mtval: 0x00000000
>> dd
Invalid command
Usage:  R: print registers
        D: display memory
        A: write assembly in command line and put at specified address
        F: load assembly from file and put at specified address
        U: read data and disassemble
        G: run user code
        T: print page table
>> d
addr: 0x80400000
num: 4
0x80400000: 0x19260817
```

2. VGA 展示

运行程序时，用户先将数据写入到FLASH中，再通过汇编将图片写入到VGA中。演示效果如下：

原图为



3. 效率优化结果

I-Cache && D-Cache

- I-Cache
- D-Cache（Writeback），且不缓存用户代码，因此可以通过功能测试
- 使用第一版监控程序（不带页表）进行测试

	1PTB	2DCT	3CCT	4MDCT	CRYPTONIGHT
No Cache	64.424s	32.210s	75.160s	46.976s	2.966s
I-Cache	24.190s	12.060s	28.174s	31.575s	1.276s
D-Cache	64.424s	32.212s	75.156s	37.585s	3.044s
I-Cache & D-Cache	24.159s	12.080s	28.186s	16.777s	1.363s

可以看到，相比于没有 Cache，我们能够做到 2~3 倍的性能提升。

TLB vs no_TLB

使用**第三版监控程序**（支持页表、中断异常）进行性能测试。

注：为了运行性能测试，我们修改了监控程序，确保不会因为程序超时而中断。

	1PTB	2DCT	3CCT	4MDCT	CRYPTONIGHT
without TLB	144.956s	72.477s	169.116s	124.885s	7.151s
with TLB	48.320s	24.160s	56.403s	38.929s	2.780s

可以看到，相比于没有 TLB，我们能够做到将近 3 倍的性能提升。

（四）思考题

1.

同：每个阶段都负责处理一件事情。异：流水线可以同时有多条指令运行，多周期CPU只能运行一条指令；流水线容易拓展，多周期不容易拓展。

在发现数据冲突时，插入气泡的方法会插入气泡，等待一个或多个等待周期，使得流水线停顿，直到冲突解决。性能损失比较大。

数据旁路的方法能立刻将已经处理好的数据前传到需要的部分，不需要暂停流水线，能够解决一部分冲突问题，性能损失很小。

2.

如何使用 Flash 作为外存：

分配一段地址空间映射到 Flash（例如我们就用了 `0x0400_0000 ~ 0x047F_FFFF`），然后让 Flash Controller 作为 Wishbone Slave 加到当前的 Wishbone MUX 后面，就可以通过地址来读 Flash 了（就像读 BaseRAM 一样，只不过读 Flash 需要用的周期数更多）

从 Flash 读入监控程序：

可以在 CPU reset 的时候，设计一个循环（或者状态机），依次读取 Flash 的前若干字（根据监控程序的长度决定），然后写到 BaseRAM 里面。

3.

可以创建一个bram用于储存图像的信息，并且增加一个controller用于读写其数据，controller的读写地址通过MMIO接到wishbone总线上。这样显示文字可以通过软件来实现。

5.

阅读程序可知，绝大多数情况下，发生了cache hit

加入cache前，循环需要的周期数为：

12+12+7+12+12+16=71

加入cache后，循环所需周期数（全部hit情况下）：

4+4+3+4+4+6=25

理论加速比为2.84，实际加速比为2.8

符合理论预期，因为实际中cache读入也需要时间，加速比实际值会比上述估计值略低。

6.

可以通过下面的虚拟地址访问：

- `0x8010_0000`：通过页表映射 `va[0x80100000, 0x80100FFF] = pa[0x80100000, 0x80100FFF]` 直接映射到 `0x8010_0000`
- `0x0`：通过页表映射 `va[0x00000000, 0x002FFFFF] = pa[0x80100000, 0x803FFFFF]` 映射到 `0x8010_0000`

流程：上面两个虚拟地址 va 都要通过 MMU 查询地址翻译到物理地址 pa

- 流水线 IF 段把 va 发送给 MMU 去翻译
- MMU 通过 ptbr 和 vpn 到内存中查询页表（页表有两级，要查询两次），得到页表项中的 ppn
- MMU 将 ppn 和 offset 拼接得到物理地址 pa，将 pa 发送回流水线 IF 段

7.

a周期的指令会从ID阶段到EXE阶段、到MEM阶段，接着会等待WB阶段的指令完成写回，然后会写MEPC、写MCAUSE、写MSTATUS、读新异常处理程序的PC、跳到PC处，总共8个周期，因此b-a=8。

（五）分工表

尹绍沣：共同完成基础流水线框架；编写 I-Cache、D-Cache

陈煜翔：共同完成基础流水线框架；编写时钟中断、页表、TLB、Flash

濮成风：共同完成基础流水线框架；编写中断异常处理、VGA

（六）遇到的困难与解决方案

例如，加上 I-Cache 后，取指部分的时序逻辑出 bug

- 这个问题在使用 I-Cache 后才出现，使用 I-Cache 之前取指太慢所以没有出现
- 发现是 Flush 逻辑没有编写完备

（七）实验心得体会

收获了一些调试 bug 的方法：

- 遇到上板结果与仿真不一致，可以先检查 warning
- 可以从错误的结果不断倒推，通过查看仿真信号，找到问题的来源