

# GROUP26 大实验文档

---

## GROUP26 大实验文档

- 实验目标
- 设计框图
- 实验内容及效果展示
  - Controller
    - 数据通路设计
      - ALU@EXE/MEM->ALU
      - DM@MEM/WB->ALU
      - ALU@MEM/WB->ALU
    - 流水线气泡与冲刷
    - 结构冲突
    - 逻辑冲突
  - 指令缓存
    - 实现方式
    - 性能数据对比
  - 分支预测
    - 实现方式
    - 性能数据对比
  - 数据缓存
    - 实现方式
    - 性能数据对比
  - vga
    - 实现方式
    - 上板实验
  - Flash
    - 实现方式
    - 上板实验
  - 中断异常
    - 实现方式
    - 上板实验
- 实验心得体会
- 遇到的困难及解决方案
- 思考题
- 分工情况

## 实验目标

---

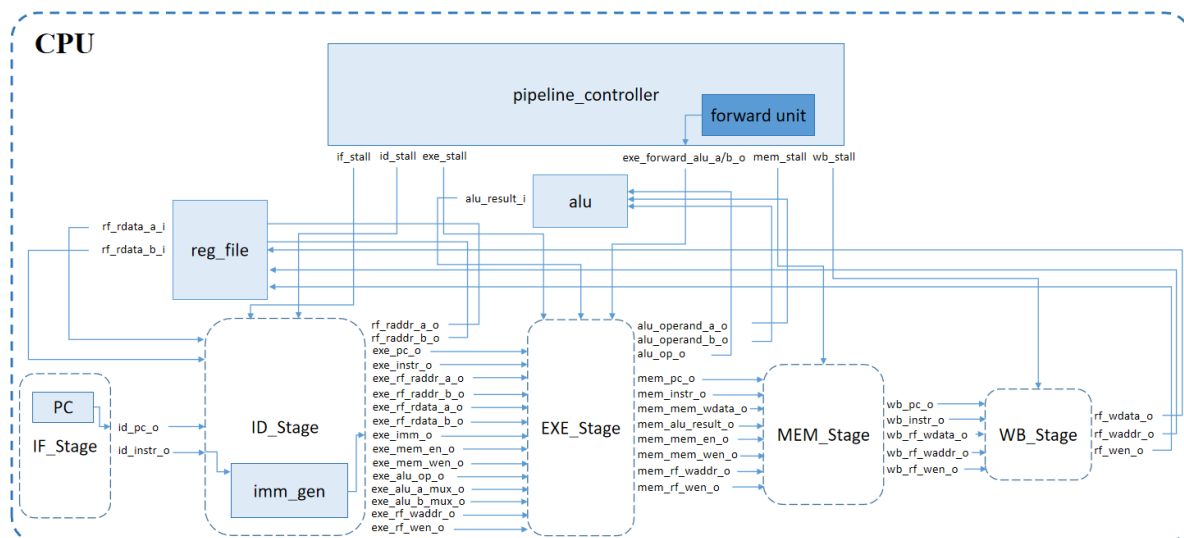
在 ThinPAD-Cloud 实验平台上实现一个完整的五级流水线CPU，支持运行 32 位监控程序。具体需要达到：

1. 能够支持监控程序基本版本用到的 RV32I 指令集。  
实现指定给本小组的额外三条指令。
2. 具有内存（SRAM）访问功能，能够满足监控程序数据与代码的存储需求。
3. 利用串口实现计算机的输入输出模块，能够支持监控程序与 PC 的相互通信。
4. 作为提高要求，实现中断处理机制，对串口产生的中断信号，运行中断处理程序接收数据。
5. 拓展任务：支持相关外设（VGA、Flash）和缓存、分支预测等拓展功能

## 设计框图

---

我们基础版本的流水线 CPU 设计如下：



扩展功能对设计框图进行的改动将在下面各个部分中进行介绍。

## 实验内容及效果展示

### Controller

`controller` 主要负责控制数据通路和流水线气泡与冲刷。

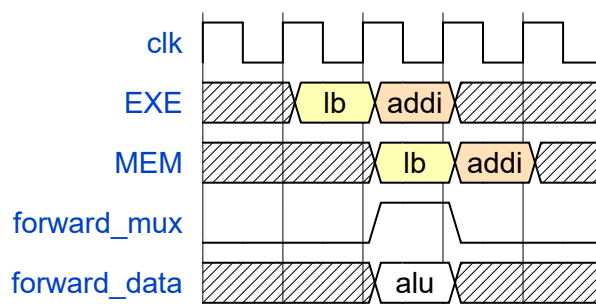
### 数据通路设计

数据通路主要有以下三条：

`ALU@EXE/MEM->ALU`

该数据通路主要解决相邻的、不涉及访存的数据冲突，例如：

```
li t0, 1
addi t1, t0, 1
```

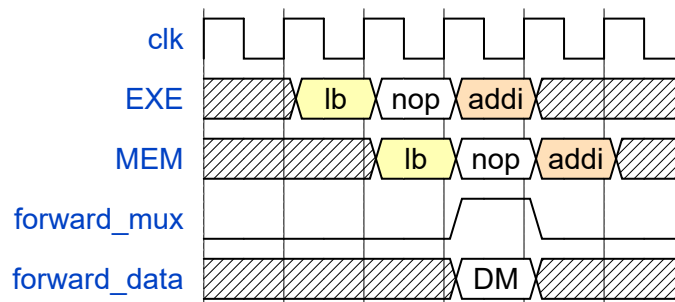


如图，当第二条指令 `addi` 执行到 EXE 阶段时，`controller` 判断 `addi` 的 `rs1`, `rs2` 寄存器与上一条指令 `li` 的在 EXE/MEM 中间寄存器的 `rd` 寄存器是否相同。如果相同说明发生数据冲突，此时 `forward_mux` 置1，`forward_data` 置为 `li` 指令保存在 EXE/MEM 中间寄存器的 `alu` 计算结果。这些 `forward` 信号作为 EXE\_STAGE 的输入信号

DM@MEM/WB->ALU

该数据通路主要解决 L 指令造成的数据冲突，例如：

```
lb t0, 0(sp)
addi t1, t0, 1
```

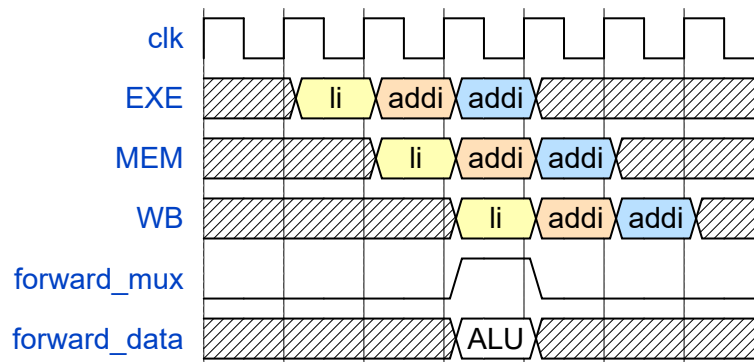


如图，当第二条指令 `addi` 执行到 EXE 阶段时（此前 `lb` 指令在流水线插入一个气泡），`controller` 判断 `addi` 的 `rs1`, `rs2` 寄存器与上一条指令 `lb` 的在 EXE/MEM 中间寄存器的 `rd` 寄存器是否相同。如果相同说明发生数据冲突，此时 `forward_mux` 置1，`forward_data` 置为 `lb` 指令保存在 MEM/WB 中间寄存器的 DM 访存结果。这些 `forward` 信号作为 EXE\_STAGE 的输入信号

ALU@MEM/WB->ALU

该数据通路主要解决相隔一条指令的数据冲突，例如：

```
li t0, 1
addi t2, t1, 1
addi t1, t0, 1
```



如图，当第三条指令 `addi` 执行到 EXE 阶段时，`controller` 判断 `addi` 的 `rs1`, `rs2` 寄存器与第一条指令 `li` 的在 MEM/WB 中间寄存器的 `rd` 寄存器是否相同。如果相同说明发生数据冲突，此时 `forward_mux` 置1，`forward_data` 置为 `li` 指令保存在 MEM/WB 中间寄存器的 ALU 结果（或 DM 访存结果）。这些 `forward` 信号作为 EXE\_STAGE 的输入信号

## 流水线气泡与冲刷

流水线需要插入气泡或冲刷主要有两类情况：

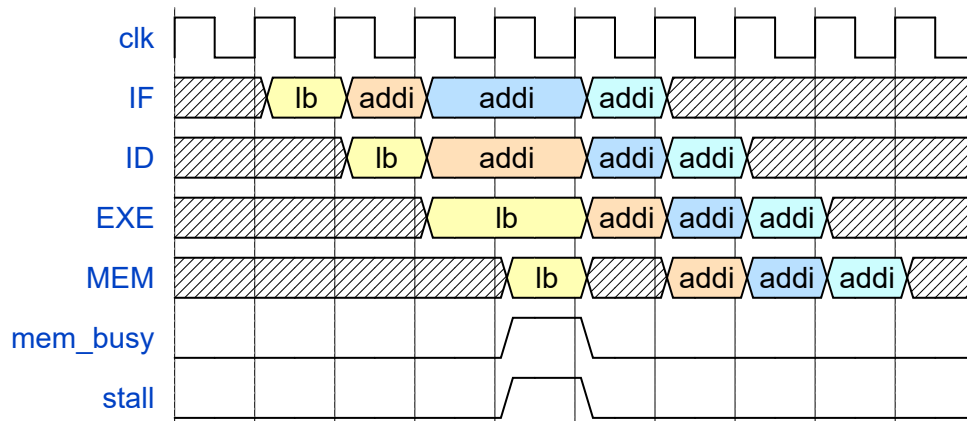
### 结构冲突

当 S 或 L 型指令需要访存时，会和 IF 阶段的读取指令访存冲突。此时 MEM 阶段拉高信号 `mem_busy`，`controller` 拉高 `if_stall`，`id_stall`，`exe_stall`，暂停 IF，ID，EXE 阶段，等待 MEM 阶段访存完成。例如：

```

lb t0, 0(sp)
addi t1, t0, 1
addi t1, t1, 1
addi t2, t1, 1

```



如图 MEM 阶段执行 lb 指令时，流水线其他阶段保持原状态不变，直到 MEM 访存结束，流水线继续运行。

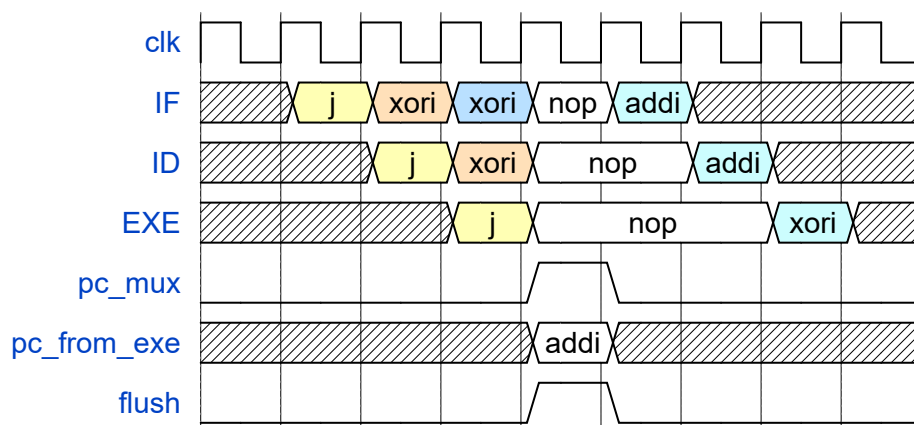
### 逻辑冲突

当 EXE 阶段的比较器判断需要执行跳转时，会拉高 pc\_mux 信号（信号连接到 IF 和 controller）。controller 默认拉高 if\_flush，id\_flush，冲刷流水线。等待跳转完成，IF 从新的地址开始执行，例如：

```

j .next
xori t0, t0, 1
xori t1, t0, 1
...
next:
addi t1, t1, 1
addi t2, t2, 1

```



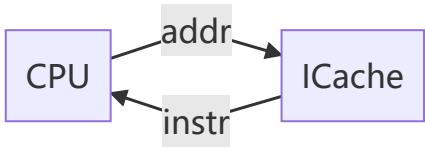
如图 EXE 阶段 j 指令执行完成，发现需要跳转，此时清空流水线 IF，ID 阶段寄存器，同时向 IF 阶段发送 pc\_mux 信号和跳转地址 pc\_from\_exe。IF 阶段收到信号后继续执行流水线。

# 指令缓存

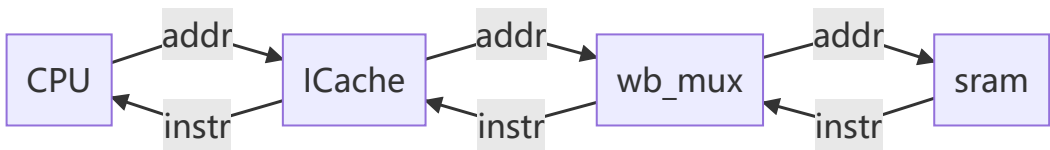
## 实现方式

使用直接映射的缓存映射机制。设置64条57位的缓存记录，分别对应1位有效位，24为标志位（32-2-6），以及32位数据位（存储32位指令数据）。具体性能对比见数据缓存部分表格。

当指令缓存命中（hit）时，取指令只需一个时钟周期：



当指令缓存未命中（!hit）时，取指令与原 IF 阶段相同，需要大约3个时钟周期：



## 性能数据对比

如下表所示：

icache	1PTB	2DCT	3CCT	4MDCT	CRYPTONIGHT
	-	22.146s	-	-	2.328s
✓	28.186s	11.073s	24.159s	-	1.867s

（其中第一行表示只支持基础版本监控程序的 CPU，— 表示测试超时）

# 分支预测

## 实现方式

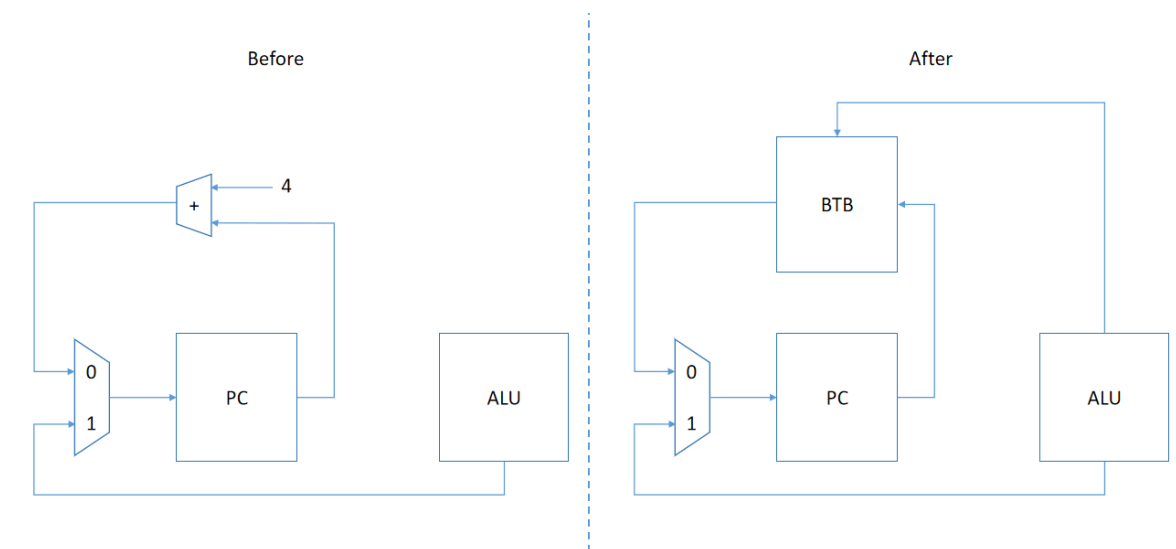
BTB 表采用直接映射的实现方式，表项数量设置为 64，每条表项共 67 位，如下图所示：



- 1 位有效位；
- 2 位 BHT 位；
- 32 位记录源 PC 地址；
- 32 位记录跳转 PC 地址。

我们通过源 PC 地址的 [7：2] 位确定表项的序号。

加入 BTB 表之后，CPU 结构改动如下：



在 IF 阶段，如果没有收到冲刷指令，则查找 BTB 表，根据 BHT 位选择跳转与否。

而在 EXE 阶段，我们不需要在解析出跳转 PC 地址后便冲刷流水线，而是查找 BTB 表，如果：

- BTB 表中不存在相应表项（有效位为 0），或表项中跳转 PC 地址与当前解析出的地址不同；
- 表项相同，但 BHT 位显示此时 IF 阶段的预测结果与实际结果不一致，

我们才会进行流水线的冲刷。

此外，在 EXE 阶段还要进行 BTB 表的更新。

## 性能数据对比

如下表所示：

icache	BTB	1PTB	2DCT	3CCT	4MDCT	CRYPTONIGHT
		-	22.146s	-	-	2.328s
✓		28.186s	11.073s	24.159s	-	1.867s
✓	✓	13.422s	7.382s	10.738s	-	1.688s

（其中第一行表示只支持基础版本监控程序的 CPU，— 表示测试超时）

## 数据缓存

### 实现方式

dcache 选择写直达（Write through）加写分配（Write allocate）的策略，并采用直接映射的实现方式，表项数量设置为 256，每条表项共 69 位，如下图所示：

valid	address	data	sel
1	32	32	4

- 1 位有效位；
- 32 位记录主存地址；
- 32 位记录主存数据；

- 4 位 sel 位。

我们通过主存地址的 [9 : 2] 位确定表项的序号。

加入 dcache 之后，CPU 在存储数据时同时修改 dcache 和对应的主存内容，在读取数据时首先访问 dcache，未命中时再访问主存。

### 性能数据对比

如下表所示：

icache	BTB	dcache	1PTB	2DCT	3CCT	4MDCT	CRYPTONIGHT
			-	22.146s	-	-	2.328s
✓			28.186s	11.073s	24.159s	-	1.867s
✓	✓		13.422s	7.382s	10.738s	-	1.688s
✓	✓	✓	13.422s	7.382s	10.738s	29.528s	1.688s

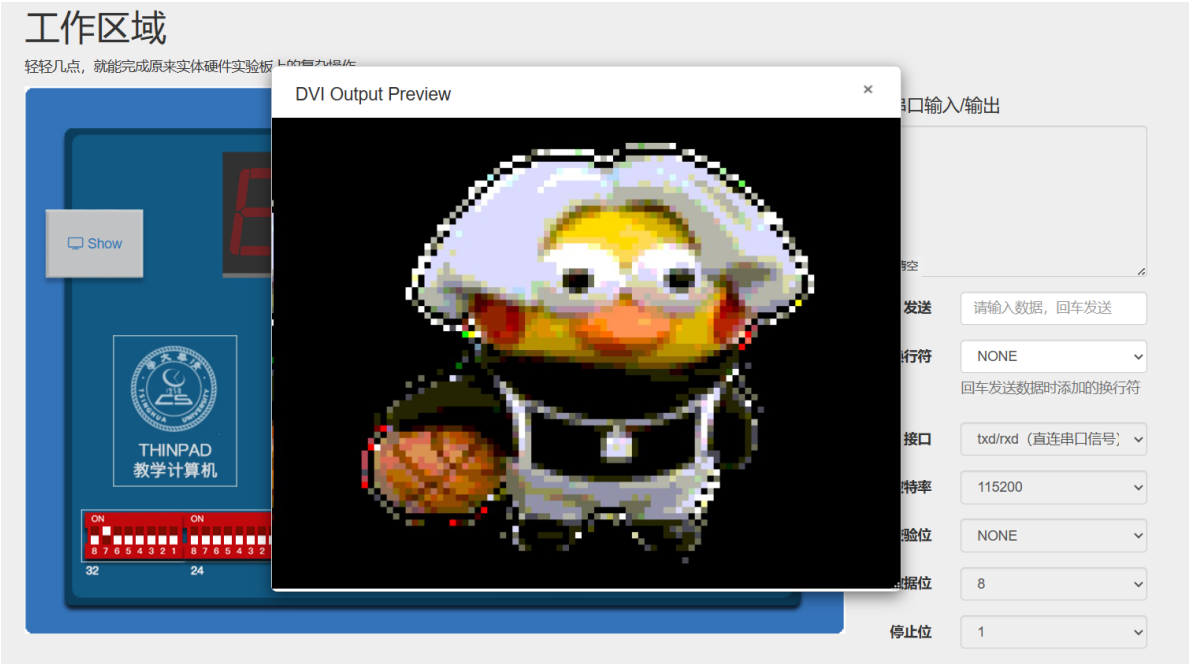
(其中第一行表示只支持基础版本监控程序的 CPU，－ 表示测试超时)

### vga

#### 实现方式

vga 使用 block memory 作为显存，每个字节中的 8 位，低 2 位代表蓝色通道；中间 3 位代表绿色通道；高 3 位代表红色通道。通过扫描遍历整个显存确定每个像素点的 RGB 值。vga 使用地址 0x6000\_0000 ~ 0x603F\_FFFF 作为显存地址，与总线相连。

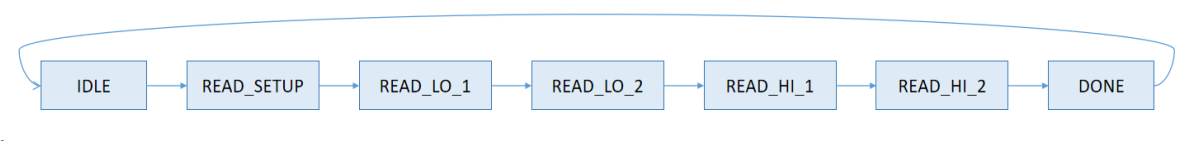
### 上板实验



# Flash

## 实现方式

Flash 采用 16 位模式，每次读取 4 个字节（32 位），Flash 控制器的状态机为：



读取低 16 位数据与高 16 位数据各使用两阶段。

## 上板实验

- 将一段文本装入 Flash 中：

Life is tale told by an idiot, full of sound and fury, signifying nothing.

Flash&RAM 读写 使用本功能将重置FPGA

存储选择

Flash

容量 8MB

起始地址

0x 0

(Byte)

起始地址应当按16位对齐（即为偶数）

读取数据

0x 读取长度

(Byte)

查看 下载

读取长度应当按16位对齐（即为偶数）

写入数据

选择文件 flash.bin

写入

写入长度为数据文件大小，按16位对齐（即为偶数）

✔操作成功完成

Hex Dump

00000000: 4c 69 66 65 20 69 73 20 74 61 6c 65 20 74 6f 6c |Life.is.tale.tol|

00000010: 64 20 62 79 20 61 6e 20 69 64 69 6f 74 2c 20 66 |d.by.an.idiot,.f|

00000020: 75 6c 6c 20 6f 66 20 73 6f 75 6e 64 20 61 6e 64 |ull.of.sound.and|

00000030: 20 66 75 72 79 2c 20 73 69 67 6e 69 66 79 69 6e |.fury,.signifyin|

00000040: 67 20 6e 6f 74 68 69 6e 67 2e df df df df df df |g.nothing.....|

00000050: df df df ff df df df df df df df df df df df df |.....|

00000060: df df df df df df df df df df df df df df df df |.....|

00000070: df df df df df df df df df df df df df df df df |.....|

00000080: df df df df df df df df df df df df df df df df |.....|

00000090: df df df df df df df df df df df df df df df df |.....|

000000a0: df df df df df df df df df df df df df df df df |.....|

000000b0: df df df df df df df df df df df df df df df df |.....|

000000c0: df df df df df df df ff df df df df df df df df |.....|

000000d0: df df df df df df df df df df df df df df df df |.....|

000000e0: df df df df df df df df df df df df df df df df |.....|

000000f0: df df df df df df df df df df df df df df df df |.....|

- 在 BaseRAM 中装入测试程序，该程序从 Flash 的起始地址出发，依次读取 200 位的数据并写入串口：



Flash&RAM 读写 使用本功能将重置FPGA

存储选择 BaseRAM 容量 4MB

起始地址 0x 0 (Byte)  
起始地址应当按16位对齐 (即为偶数)

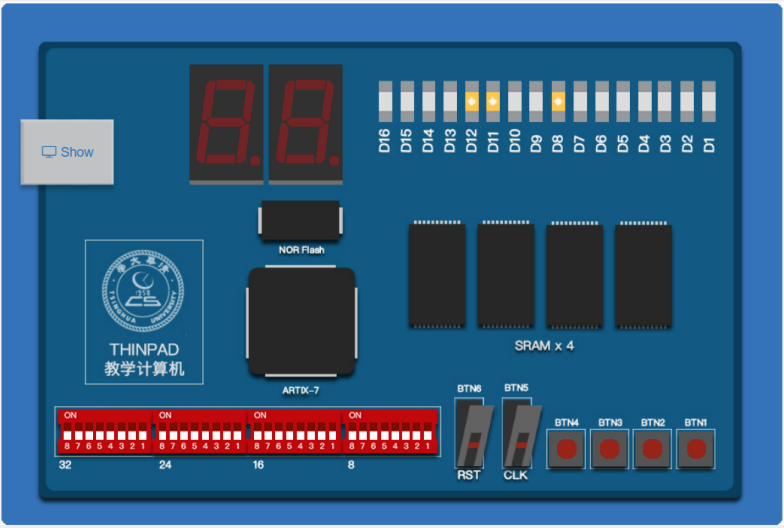
读取数据 0x 读取长度 (Byte) 查看 下载  
读取长度应当按16位对齐 (即为偶数)

写入数据 选择文件 test-flash.bin 写入  
写入长度为数据文件大小, 按16位对齐 (即为偶数)

✓操作成功完成

- 可以看到, 串口成功显示 Flash 中装入的文本:

轻轻几点, 就能完成原来实体硬件实验板上的复杂操作



串口输入/输出

Life is tale told by an idiot, full of sound and fury, signifying nothing.

清空

发送 请输入数据, 回车发送

换行符 NONE  
回车发送数据时添加的换行符

接口 txd/rxd (直连串口信号)

波特率 115200

校验位 NONE

数据位 8

停止位 1

网络转发 166.111.226.111:38927  
通过 TCP 连接访问串口

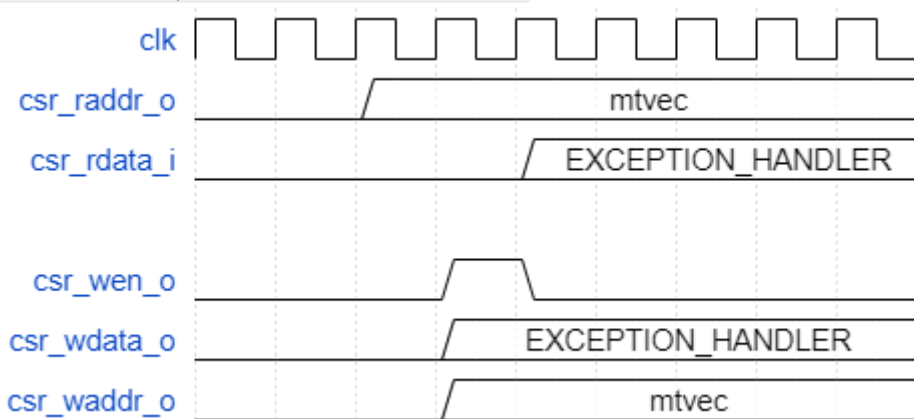
发送 终端模式 关闭串口

## 中断异常

### 实现方式

- CSR 寄存器的读写: ID 阶段读取, EXE 阶段写入(根据当前特权等级写入不同字段)

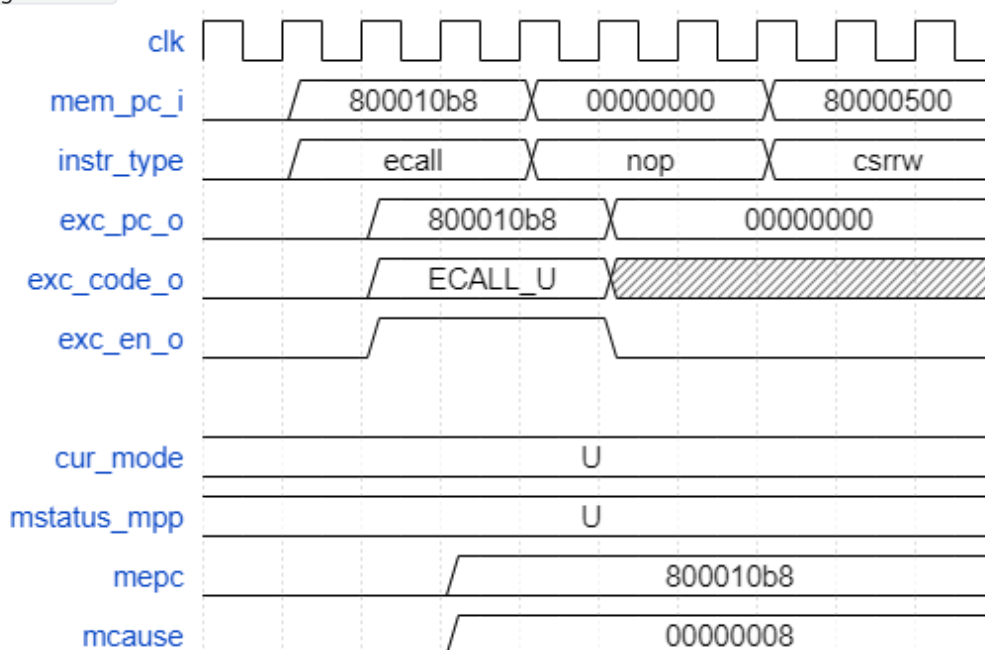
e.g. `scrw mtvec, s0 # EXCEPTION_HANDLER`



- 异常处理:
  - ID 阶段检测异常, 等待到 MEM 阶段进行异常处理, 将异常相关信号传递给 CSR 寄存器 (使能、异常原因、异常地址), 修改对应 CSR 寄存器, 将当前模式保存到 `mstatus_mpp`, 随后

冲刷ID、EXE阶段，跳转到异常处理地址;

- 时钟中断由独立的slave读取，`mtime > mtime_cmp` 信号会输出到 MEM (用于提供异常 PC)与 CSR 寄存器，CSR 寄存器拉高 `mip_mtip` 字段, 当 `mie_mtie`、`mip_mtip` 均拉高且当前特权状态为U模式时打开时钟中断;
- 异常恢复时从 `mstatus` 寄存器中恢复特权状态与mie  
e.g. `ecall`



## 上板实验

运行kernel内测试程序如下：

```
connecting to 166.111.226.111:41715...connected

MONITOR for RISC-V - initialized
running in 32bit, xlen = 4
>> g
addr: 0x800010a8
OK
elapsed time: 0.000s
>> g
addr: 0x800010c0
killed timeout program.

elapsed time: 0.200s
>> a
addr: 0x80100000
one instruction per line, empty line to end.
[0x80100000] addi a0, a0, 1
[0x80100004]
>> g
addr: 0x80100000
supervisor reported an exception during execution
  mepc: 0x80100000
  mcause: 0x00000002
  mtval: 0x00000000
>>
```

## 实验心得体会

---

学习到了硬件编程和调试技能 |\_-|

## 遇到的困难及解决方案

---

硬件编程的极其不适应；调试过程极其困难，仿真和上板存在诸多不一致现象。

## 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。
-

流水线 CPU 设计通过将指令的执行过程划分为多个阶段，并在每个时钟周期中并行地执行不同阶段的指令，从而实现指令级并行。这样可以提高指令的执行效率和吞吐量。插入等待周期（气泡）会导致流水线暂停，延迟指令的执行，从而降低了流水线的效率和吞吐量。数据旁路可以将计算结果直接传输给需要使用这些结果的指令，而无需等待中间结果的存储和加载操作。通过数据旁路，多周期 CPU 可以实现更高的吞吐量和效率，而无需插入等待周期来解决数据冲突。

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？

---

CPU 需要做以下改动来读取存放在 Flash 上固定位置的监控程序：

- 设置引导向量或合理配置系统启动流程
- 设置 Flash 的地址映射
- 编写或选择适当的引导程序，负责将监控程序从 Flash 加载到内存的固定位置。

3. 如何将 DVI 作为系统的输出设备，从而在屏幕上显示文字？

---

选择适用于 DVI 输出的硬件，再将显示器的 DVI 接口连接到 cpu 系统的 DVI 输出端口。使用脚本，将预设的文字映射为对应的矢量图或像素图，并将其按规定的编码全部装入显存中。最后，逐字节读取显存，解码为逐像素的 RGB 值，通过 DVI 接口发送到输出设备上。

4. 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。

---

测例内容如下：

```
UTEST_3CCT:
    lui t0, %hi (TESTLOOP64)          // 装入64M
.LC2_0:
    bne t0, zero, .LC2_1
    jr ra
.LC2_1:
    j .LC2_2
.LC2_2:
    addi t0, t0, -1
    j .LC2_0
    addi t0, t0, -1
```

前几次循环中遭遇跳转指令，IF 阶段均会选择 not 跳转，从而导致预测失败；但由于循环次数足够大，我们无需考虑前几次循环预测失败的情况，则对于使用了分支预测的版本来说，BTB 表中将存在如下表项：

源指令	跳转位置指令	跳转与否
bne t0, zero, .LC2_1	j .LC2_2	跳转
j .LC2_2	addi t0, t0, -1	跳转
j .LC2_0	bne t0, zero, .LC2_1	跳转

故对于使用了分支预测的版本，每次循环 IF 阶段都会进行正确的分支预测，一次循环将经过 4 条指令。而对于未使用分支预测的版本，每次 EXE 阶段解析出跳转指令后，都需要冲刷流水线，相当于额外经过了 2 条指令，于是，一次循环将经过 10 条指令。

因此，理论上分支预测的准确率可以达到近似 100%，且相较于未使用分支预测的版本，性能可以提升至原本的 10/4，即用时变为原本的 40%。

由测试结果：

icache	BTB	3CCT
✓		24.159s
✓	✓	10.738s

有：

$$\frac{10.738}{24.159} \approx 44.4\%$$

说明与分析大致相符。

5. 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。

测例内容如下：

```
UTEST_4MDCT:
    lui t0, %hi(TESTLOOP32)      // 装入32M
    addi sp, sp, -4
.LC3:
    sw t0, 0(sp)
    lw t1, 0(sp)
    addi t1, t1, -1
    sw t1, 0(sp)
    lw t0, 0(sp)
    bne t0, zero, .LC3

    addi sp, sp, 4
    jr ra
```

即程序会在同一个主存地址中反复进行写入与读取。

则对于使用了 dcache 的版本来说，首次写入将遭遇必然缺失，但之后的每次写入均会命中，而读取操作每次都会命中，其写入和读取过程大致为：

- 写入过程：同时写入 dcache 和主存；

- 读取过程：直接在 dcache 中进行读取。

而对于未实现 dcache 的版本，读取过程将会到主存中进行读取。

我们假定对主存进行操作将花费大约 3 个时钟周期，而对 dcache 进行操作只需 1 个时钟周期，则由于每次循环进行两个写入操作和读取操作，相较于未实现 dcache 的版本，性能将提升至原本的 6/4，即用时变为原本的约 60%。

由测试结果：

dcache	4MDCT
	49.534s
✓	29.528s

有：

$$\frac{29.528}{49.534} \approx 59.6\%$$

说明与我们的分析大致相符，dcache 的实现能够节省更多的时钟周期。

7. 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？

异常tag会经过ID、EXE、MEM阶段，随即中断异常控制器将mtvec传给IF阶段，因此b-a为3个周期

## 分工情况

汪晗阳：流水线 controller，指令缓存，vga

李想：流水线 IF、ID、WB，分支预测，数据缓存，flash

占海川：流水线 EXE、MEM，中断异常