《计算机组成原理》RISC-V CPU 实验报告

38 组

计 84 刘泓尊 2018011446

计 83 刘云昊 2018011335

计 84 张瀚天 2018011360

1. 实验目标概述

在 THINPAD 教学计算机实验平台上，设计并实现一个完整的支持 32 位 RSICV 指令集的 CPU，支持精确异常处理，实现 TLB 模块，并且使得其可以运行教学计算机的监控程序。

(1) 深入理解计算机指令的执行方式，掌握处理器的基本设计方法；

(2) 深入理解计算机的各部件组成及内部工作原理；

(3) 加深对于 RISCV32I 指令集的理解；

(4) 掌握计算机外部输入输出的设计；

(5) 提高硬件设计和调试的能力；

2. 代码链接

https://git.tsinghua.edu.cn/cod-ta/cod20-grp38/tree/paging_exception

3. 小组分工

代码编写与仿真测试（基础、扩展指令、异常处理、页表、TLB）：刘泓尊

文档阅读与汇编测试代码：刘云昊

PPT：张瀚天

实验报告：刘泓尊、刘云昊、张瀚天

4. 主要模块设计与实现

我们最终实现了可以支持异常中断、页表 TLB 的多周期 CPU，由 IF、ID、EX、MEM、WB 五个标准阶段构成。

取址（IF）阶段

首先对 PC 值进行判断是否出现地址越界或者地址不对齐的异常，如果出现则直接进入异常处理模块。如果未出现异常，则判断是否使用页表，若处于用户态且 Satp 寄存器有效，则使用页表；否则，按照 PC 正常取址。

使用页表时，根据 PC 在 TLB 中进行查询，若命中且 TLB 有效，则前 20 位用 TLB 对应的实地址，后 12 位用 PC 本身的 offset 进行内存读取;若未命中，则 TLB miss，将 PC 的虚地址通过页表查询映射成对应的实地址，进行内存读取，页表中的页表项无效则报指令页错误(instruction page fault),有效则将实地址存入 TLB 中。

译码（ID）阶段

ID 阶段实现指令译码，识别并解析 IF 阶段从内存读出来的指令，从而产生控制信号给 EX 阶段，是一个复杂的组合逻辑电路，根据对输入指令的解析，提供了 PC 跳转、写寄存器、读寄存器、Alu 数据来源、译码是否异常等输出信号，对于未定义的指令，则报非法指令的异常，传入 EX 阶段进行处理。

执行（EX）阶段

EX 阶段实现算数逻辑运算，根据 ID 译码模块提供的 Alu 数据来源的信号，将对应的数据传入 Alu 进行运算。Alu 是一个复杂的组合逻辑电路，其中的运算主要为对于运算操作的运算和对于访存操作的运算，分为逻辑运算和算数运算。其中，逻辑运算主要指 AND、OR、XOR、SLL、SRL、CTZ、SBCLR,算数运算主要指 ADD,SUB,JALR,MIN。运算结果在多周期中继续传递，在 WB 阶段写回到寄存器堆中，对于访存指令，需要在 EX 阶段根据给出的基地址二号偏移量，计算出访存的具体地址（虚地址），之后传给 MEM 阶段进行访存处理。

访存（MEM）阶段

MEM 阶段实现访存操作，主要工作是识别访存指令，并根据相应的访存指令产生相应的读写片选信号，如果处于用户态且 Satp 寄存器有效则 EX 阶段传过来的是虚拟地址，使用 MMU 与 TLB；否则为实际地址，直接进行访存。传过来的是虚拟地址时，先在 TLB 中进行查询，查到则直接进行虚实地址转换；未查到，则通过页表进行查询，查到则存入 TLB，根据页表得到的实际地址进行访存。如果页表中的页表项无效，则 load 报 load page fault, store 报 store page fault。

写回（WB）阶段

WB 阶段分为写回寄存器和更新 PC 两部分，根据指令的不同选择将 PC+4 或者 EX 阶段的结果写回到指定的寄存器或异常寄存器中；更新 PC 则若跳转，则将 PC 更新为 EX 阶段的结果，未跳转则更新为 PC+4，然后跳转到取址（IF）模块。

异常与中断

异常信息的种类：

Exception Code	Description
0	指令地址不对齐
1	指令地址越界
2	非法指令

3	断点
4	Load 地址不对齐
5	Load 地址越界
6	Store 地址不对齐
7	Store 地址越界
8	用户模式 Ecall
10	机器模式 Ecall
12	指令页错误
13	Load 页错误
15	Store 页错误

指令不对齐是判断地址的最后两位不为 0；
非法指令是译码阶段指令未定义；
地址越界是不在相应模式下的地址范围；
页错误是页表中对应的页表项无效；
Ecall 由 Ecall 指令触发；
断点由 Ebreak 指令触发；

在每次在其他阶段检测到发生异常时，都会跳转到异常处理阶段，将对应的异常码写入 mcause 寄存器，将当前 PC 存入 mepc 寄存器，将当前状态记录到 mstatus 寄存器的[12:11]，然后将 PC 设置为 mtvec 中记录的异常处理程序的地址，转变当前状态为机器态。

遇到 MRET 指令，则进行异常状态恢复，将 PC 赋值为 mepc 中记录的发生异常时的 PC,状态恢复为 mstatus 寄存器的[12:11]中记录的发生异常前的状态。

内存管理：MMU+TLB

虚拟地址与实际地址的映射关系如下：

- va[0x00000000, 0x002FFFFF] = pa[0x80100000, 0x803FFFFF] DAGUX-RV 用户态代码
- va[0x7FC10000, 0x7FFFFFFF] = pa[0x80400000, 0x807EFFFF] DAGU-WRV 用户态数据
- va[0x80000000, 0x80000FFF] = pa[0x80000000, 0x80000FFF] DAGUX-RV 用于返回内核态
- va[0x80100000, 0x80100FFF] = pa[0x80100000, 0x80100FFF] DAGUX-RV 方便测试

通过页表将虚拟地址转化为实际地址的过程经过了三次读取内存：

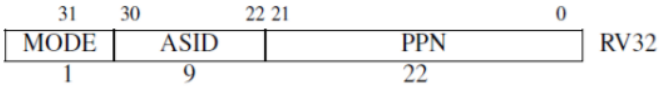
1. satp.PPN 给出了一级页表的基址，VA[31:22]给出了一级页号，因此处理器会读取位于地址(satp. PPN × 4096 + VA[31: 22] × 4)的页表项。

2. 该 PTE 包含二级页表的基址，VA[21:12]给出了二级页号，因此处理器读取位于地址(PTE. PPN × 4096 + VA[21: 12] × 4)的叶节点页表项。

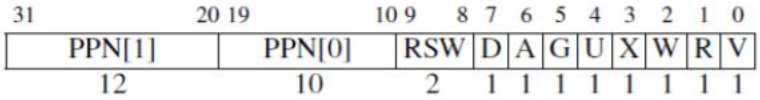
3. 叶节点页表项的 PPN 字段和页内偏移（原始虚址的最低 12 个有效

位）组成了最终结果：物理地址就是(LeafPTE. PPN × 4096 + VA[11: 0])

satp 的结构如下：



页表项的结构如下：



TLB 模块负责 128 条全相联的 TLB 表项，其中记录虚拟地址的前缀、实际地址以及是否有效，以虚拟地址的后 7 位作为 index 访问 TLB，匹配虚拟地址的前缀，若一致，则 TLB 命中，可以直接使用相应的实际地址，否则需要查询页表。

SRAM

Thinpad 上一共提供了两块 SRAM，分别为 baseRam 和 extRam，分别映射到物理地址的 0x00000000 - 0x003FFFFF 和 0x00400000 - 0x007FFFFF。

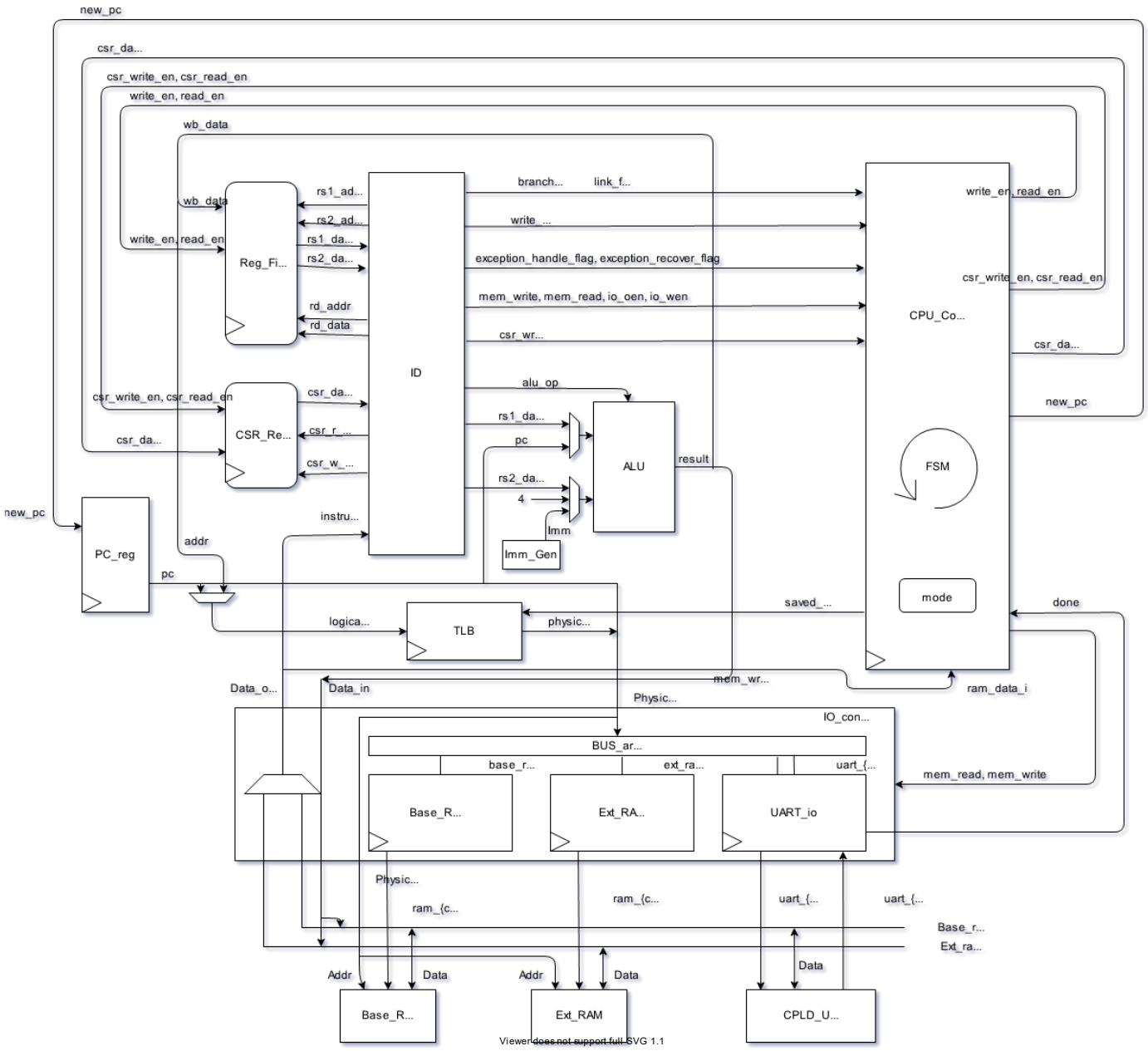
使用时，我们按照物理地址拉低对应 SRAM 片的 ce 和 oe/we 信号。读操作会先置 data 为高阻态，等待内存数据流入；写操作会直接将 data 置为 CPU 需要写入的内容，等待内存接受数据。

串口

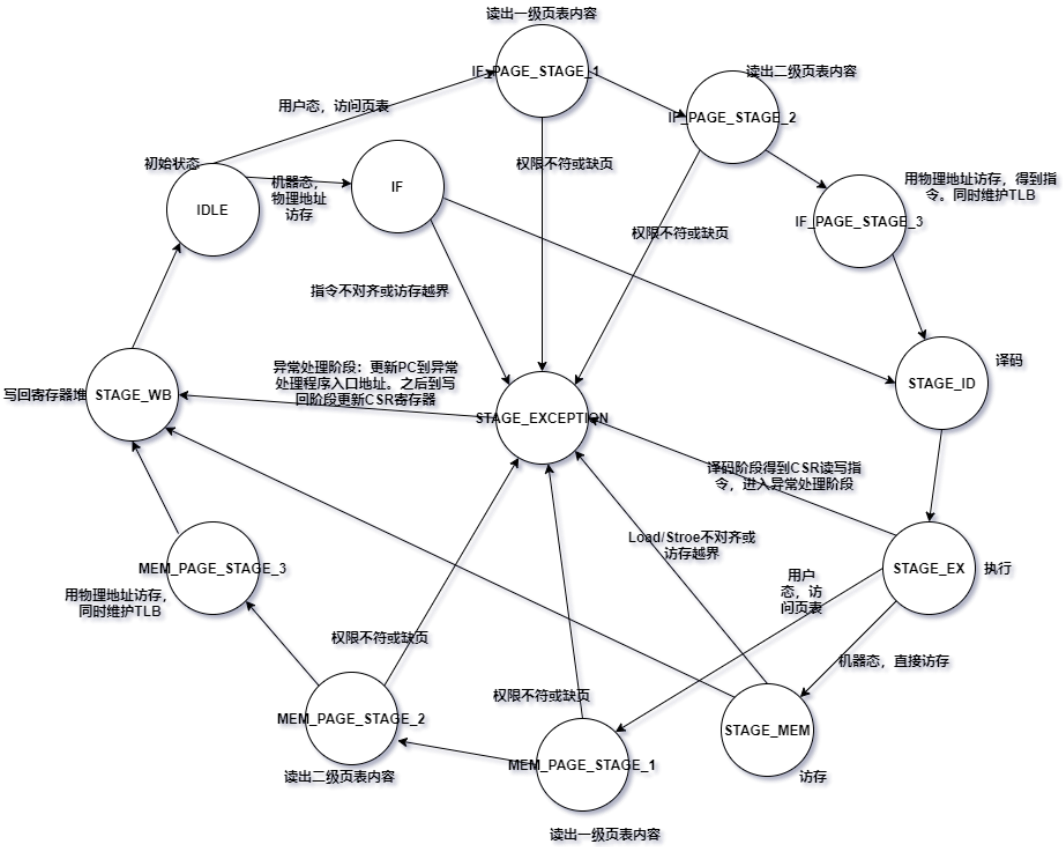
读串口时通过忙等待等待 uart_dataready 信号为 1，即数据准备完毕，从总线读出数据，将 uart_rdn 赋值为 1，关闭读使能；

写串口时将 uart_wrn 赋值为 0，打开写使能，等待若干空拍后，将 uart_wrn 赋值为 1，关闭写使能，忙等待 uart_tsre 为 1，即数据发送完毕后，写串口完成。

5. 数据通路


new_pc
csr_da...
csr_write_en, csr_read_en
write_en, read_en
wb_data
wb_data
write_en, read_en
Reg_Fi...
rs1_ad...
rs2_ad...
rs1_da...
rs2_da...
rd_addr
rd_data
ID
branch...
link_f...
write_...
exception_handle_flag, exception_recover_flag
mem_write, mem_read, io_oen, io_wen
csr_wr...
alu_op
rs1_da...
pc
rs2_da...
4
Imm
Imm_Gen
ALU
result
write_en, read_en
csr_write_en, csr_read_en
CPU_Co...
csr_da...
new_pc
FSM
mode
done
saved_...
csr_write_en, csr_read_en
csr_da...
CSR_Re...
csr_da...
csr_r_...
csr_w_...
csr_da...
new_pc
PC_reg
addr
pc
instru...
logica...
TLB
physic...
mem_wr...
ram_data_i
Data_o...
Data_in
Physic...
IO_con...
BUS_ar...
base_r...
ext_ra...
uart_{...
mem_read, mem_write
Base_R...
Ext_RA...
UART_io
Physic...
ram_{c...
ram_{c...
uart_{...
uart_{...
Base_r...
Ext_ra...
Data
Addr
Data
Addr
Data
Base_R...
Ext_RAM
CPLD_U...
Viewer does not support full SVG 1.1

6. CPU 主控状态机设计



7. 性能测试结果

1PTB: ~67s

```
>> A
addr: 0x80100000
one instruction per line, empty line to end.
[0x80100000] UTEST_1PTB:
[0x80100000]     li t0, 0x04000000
[0x80100004] .LC0:
[0x80100004]     addi t0, t0, -1
[0x80100008]     ori t1, zero, 0
[0x8010000c]     ori t2, zero, 1
[0x80100010]     ori t3, zero, 2
[0x80100014]     bne t0, zero, .LC0
[0x80100018]     jr ra
[0x8010001c]
>> U
addr: 0x80100000
num: 28
0x80100000:     040002b7        lui     t0,0x4000
0x80100004:     fff28293        addi    t0,t0,-1
0x80100008:     00006313        ori     t1,zero,0
0x8010000c:     00106393        ori     t2,zero,1
0x80100010:     00206e13        ori     t3,zero,2
0x80100014:     fe0298e3        bnez    t0,0x80100004
0x80100018:     00008067        ret
>> G
addr: 0x80100000

elapsed time: 67.108s
>>
```

2DCT: ~36s

```
>> A
addr: 0x80100000
one instruction per line, empty line to end.
[0x80100000] UTEST_2DCT:
[0x80100000]     lui t0, %hi(0x01000000)
[0x80100004]     ori t1, zero, 1
[0x80100008]     ori t2, zero, 2
[0x8010000c]     ori t3, zero, 3
[0x80100010] .LC1:
[0x80100010]     xor t2, t2, t1
[0x80100014]     xor t1, t1, t2
[0x80100018]     xor t2, t2, t1
[0x8010001c]     xor t3, t3, t2
[0x80100020]     xor t2, t2, t3
[0x80100024]     xor t3, t3, t2
[0x80100028]     xor t1, t1, t3
[0x8010002c]     xor t3, t3, t1
[0x80100030]     xor t1, t1, t3
[0x80100034]     addi t0, t0, -1
[0x80100038]     bne t0, zero, .LC1
[0x8010003c]     jr ra
[0x80100040]
>> G
addr: 0x80100000

elapsed time: 36.909s
>>
```

3CCT: ~59s

```
>> A
addr: 0x80100000
one instruction per line, empty line to end.
[0x80100000] UTEST_3CCT:
[0x80100000]     lui t0, %hi(0x04000000)
[0x80100004] .LC2_0:
[0x80100004]     bne t0, zero, .LC2_1
[0x8010000c]     jr ra
[0x80100010] .LC2_1:
[0x80100010]     j .LC2_2
[0x80100010] .LC2_2:
[0x80100010]     addi t0, t0, -1
[0x80100014]     j .LC2_0
[0x80100018]     addi t0, t0, -1
[0x8010001c]
>> G
addr: 0x80100000

elapsed time: 59.076s
```

4MDCT: ~55s

```
>> A
addr: 0x80100000
one instruction per line, empty line to end.
[0x80100000] UTEST_4MDCT:
[0x80100000]     lui a0, 0x7fc10 # virtual addr!
[0x80100004]     addi a0, a0, 4
[0x80100008]     lui t0, %hi(0x02000000)
[0x8010000c] .LC3:
[0x8010000c]     sw t0, 0(a0)
[0x80100010]     lw t1, 0(a0)
[0x80100014]     addi t1, t1, -1
[0x80100018]     sw t1, 0(a0)
[0x8010001c]     lw t0, 0(a0)
[0x80100020]     bne t0, zero, .LC3
[0x80100024]     addi a0, a0, 4
[0x80100028]     jr ra
[0x8010002c]
>> G
addr: 0x80100000

elapsed time: 55.034s
```

CRYPTONIGHT: ~3.67s

```
[0x8010006c]     xor s0, s0, s1
[0x80100070]     slli s1, s0, 5
[0x80100074]     xor s0, s0, s1
[0x80100078]     # calculate a valid addr from new rand n
[0x80100078]     and t1, s0, a4
[0x8010007c]     add t1, a0, t1
[0x80100080]     # write t0 to this addr
[0x80100080]     sw t0, 0(t1)
[0x80100084]     # save t0 for next iteration
[0x80100084]     mv t1, t0
[0x80100088]     # get new rand number from xorshift lfsr
[0x80100088]     slli s1, s0, 13
[0x8010008c]     xor s0, s0, s1
[0x80100090]     srli s1, s0, 17
[0x80100094]     xor s0, s0, s1
[0x80100098]     slli s1, s0, 5
[0x8010009c]     xor s0, s0, s1
[0x801000a0]     add a2, a2, 1
[0x801000a4]     bne a2, a3, .MAIN_LOOP
[0x801000a8]     jr ra
[0x801000ac]
>> G
addr: 0x80100000

elapsed time: 3.672s
>> A
```

8. 附加功能展示

异常部分：

实现了“异常与中断”部分全部的异常检测。下面列举几例进行展示。

ecall:

```
>> A
addr: 0x80100000
one instruction per line, empty line to end.
[0x80100000] UTEST_PUTC:
[0x80100000]      li s0, 30
[0x80100004]      li a0, 0x4F              # 'O'
[0x80100008]      ecall
[0x8010000c]      li a0, 0x4B              # 'K'
[0x80100010]      ecall
[0x80100014]      jr ra
[0x80100018]
>> G
addr: 0x80100000
OK
elapsed time: 0.000s
>>
```

地址不对齐相关

load 地址不对齐

```
>> A
addr: 0x80100000
one instruction per line, empty line to end.
[0x80100000]  .section .text
[0x80100000] .globl _start
[0x80100000] _start:
[0x80100000]  addi t1, x0, 0x1
[0x80100004]  lui t4,0x7fc10
[0x80100008]  addi t4, t4, 0x1
[0x8010000c]  sw t1,(t4)
[0x80100010]
>> G
addr: 0x80100000
supervisor reported an exception during execution
  mepc: 0x8010000c
  mcause: 0x00000006
  mtval: 0x7fc10001
>>
```

store 地址不对齐

```
>> A
addr: 0x80100000
one instruction per line, empty line to end.
[0x80100000]  .section .text
[0x80100000] .globl _start
[0x80100000] _start:
[0x80100000]  addi t1, x0, 0x1
[0x80100004]  lui t4,0x7fc10
[0x80100008]  addi t4, t4, 0x1
[0x8010000c]  lw t1,(t4)
[0x80100010]  ret
[0x80100014]
>> G
addr: 0x80100000
supervisor reported an exception during execution
  mepc: 0x8010000c
  mcause: 0x00000004
  mtval: 0x7fc10001
>>
```

取指地址不对齐

```
>> A
addr: 0x80100000
one instruction per line, empty line to end.
[0x80100000] 0x363
[0x80100004] li s0, 30
[0x80100008] li a0, 0x4F
[0x8010000c] ecall
[0x80100010] li a0, 0x4B
[0x80100014] ecall
[0x80100018] ret
[0x8010001c]
>> G
addr: 0x80100000
supervisor reported an exception during execution
  mepc: 0x80100000
  mcause: 0x00000000
  mtval: 0x00000363
>>
```

地址越界相关

load 地址越界

```
>> A
addr: 0x80100000
one instruction per line, empty line to end.
[0x80100000] UTEST_4MDCT:
[0x80100000]     lui a0, 0x7fc10
[0x80100004]     #addi a0, a0, 4
[0x80100004]     lui t0, %hi(0x02000000)          # 装入32M
[0x80100008]     addi a0, a0, -4
[0x8010000c] .LC3:
[0x8010000c]     sw t0, 0(a0)
[0x80100010]     lw t1, 0(a0)
[0x80100014]     addi t1, t1, -1
[0x80100018]     sw t1, 0(a0)
[0x8010001c]     lw t0, 0(a0)
[0x80100020]     bne t0, zero, .LC3
[0x80100024]     addi a0, a0, 4
[0x80100028]     jr ra
[0x8010002c]
>> G
addr: 0x80100000
supervisor reported an exception during execution
  mepc: 0x8010000c
  mcause: 0x00000007
  mtval: 0x7fc0fffc
>>
```

store 地址越界

```
>> A
addr: 0x80100000
one instruction per line, empty line to end.
[0x80100000] UTEST_4MDCT:
[0x80100000]     lui a0, 0x7fc10
[0x80100004]     lui t0, %hi(0x02000000)          # 装入32M
[0x80100008] .LC3:
[0x80100008]     sw t0, 0(a0)
[0x8010000c]     addi a0, a0, -4
[0x80100010]     lw t1, 0(a0)
[0x80100014]     addi t1, t1, -1
[0x80100018]     sw t1, 0(a0)
[0x8010001c]     lw t0, 0(a0)
[0x80100020]     bne t0, zero, .LC3
[0x80100024]     addi a0, a0, 4
[0x80100028]     jr ra
[0x8010002c]
>> G
addr: 0x80100000
supervisor reported an exception during execution
  mepc: 0x80100010
  mcause: 0x00000005
  mtval: 0x7fc0fffc
>>
```

非法指令

```
>> A
addr: 0x80100000
one instruction per line, empty line to end.
[0x80100000] .section .text
[0x80100000] .globl _start
[0x80100000] _start:
[0x80100000]  addi a0, x0, 0x0
[0x80100004]  addi a1, x0, 0x0
[0x80100008]  addi a2, x0, 0x1
[0x8010000c]  addi a3, x0, 0x0
[0x80100010]  addi t3, x0, 0x3c
[0x80100014]  lui t4,0x7fc10
[0x80100018] loop:
[0x80100018]  add t1, x0, a2
[0x8010001c]  add t2, x0, a3
[0x80100020]  add a5,x0, a2
[0x80100024]  add a2, a2, a0
[0x80100028]  sltu a5, a2, a5
[0x8010002c]  add a3,a1, a3
[0x80100030]  add a3,a5, a3
[0x80100034]  add a0,x0, t1
[0x80100038]  add a1,x0, t2
[0x8010003c]  add t3, t3, -1
[0x80100040]  bne t3, x0, loop
[0x80100044]  sw a0,(t4)
[0x80100048]  addi t4,t4,0x4
[0x8010004c]  sw a1,(t4)
[0x80100050]  jr ra
[0x80100054]
>> G
addr: 0x80100000
supervisor reported an exception during execution
  mepc: 0x80100028
  mcause: 0x00000002
  mtval: 0x00f637b3
>>
```

PageFault

Store Page Fault

```
>> A
addr: 0x80100000
one instruction per line, empty line to end.
[0x80100000] .section .text
[0x80100000] .globl _start
[0x80100000] _start:
[0x80100000]  addi t1, x0, 0x1
[0x80100004]  lui t4,0x00000
[0x80100008]  addi t4, t4, 0x4
[0x8010000c]  sw t1,(t4)
[0x80100010]
>> G
addr: 0x80100000
supervisor reported an exception during execution
  mepc: 0x8010000c
  mcause: 0x0000000f
  mtval: 0x00000004
>>
```

页表和 TLB：

上面的性能测试程序均使用虚拟地址(0x7fc10…)，已经充分展示了页表的功能。因为引入页表之后需要 3 次访存，程序性能会下降。但是加上 TLB 之后性能有大幅提升，和加入页表之前的性能相近。下面是运行 T 命令的结果，可以看到页表已经被正确配置.

```
>> T
Page table at 80002000
     Virtual Address      |      Physical Address      | D | A | G | U | X | W | R | V
    00000000-00000fff          80100000-80100fff        1   1   1   1   1   0   1   1
    00001000-00001fff          80101000-80101fff        1   1   1   1   1   0   1   1
    00002000-00002fff          80102000-80102fff        1   1   1   1   1   0   1   1
    00003000-00003fff          80103000-80103fff        1   1   1   1   1   0   1   1
    00004000-00004fff          80104000-80104fff        1   1   1   1   1   0   1   1
    00005000-00005fff          80105000-80105fff        1   1   1   1   1   0   1   1
    00006000-00006fff          80106000-80106fff        1   1   1   1   1   0   1   1
    00007000-00007fff          80107000-80107fff        1   1   1   1   1   0   1   1
    00008000-00008fff          80108000-80108fff        1   1   1   1   1   0   1   1
    00009000-00009fff          80109000-80109fff        1   1   1   1   1   0   1   1
    0000a000-0000afff          8010a000-8010afff        1   1   1   1   1   0   1   1
    0000b000-0000bfff          8010b000-8010bfff        1   1   1   1   1   0   1   1
    0000c000-0000cfff          8010c000-8010cfff        1   1   1   1   1   0   1   1
    0000d000-0000dfff          8010d000-8010dfff        1   1   1   1   1   0   1   1
    0000e000-0000efff          8010e000-8010efff        1   1   1   1   1   0   1   1
    0000f000-0000ffff          8010f000-8010ffff        1   1   1   1   1   0   1   1
    00010000-00010fff          80110000-80110fff        1   1   1   1   1   0   1   1
    00011000-00011fff          80111000-80111fff        1   1   1   1   1   0   1   1
    00012000-00012fff          80112000-80112fff        1   1   1   1   1   0   1   1
    00013000-00013fff          80113000-80113fff        1   1   1   1   1   0   1   1
    00014000-00014fff          80114000-80114fff        1   1   1   1   1   0   1   1
    00015000-00015fff          80115000-80115fff        1   1   1   1   1   0   1   1
    00016000-00016fff          80116000-80116fff        1   1   1   1   1   0   1   1
```

9. 指令流程表

10. 指令流程表

指令	ID	EX	MEM
ADD	获取两个源寄存器的值，获取目的寄存器的地址	计算结果	
AND	获取两个源寄存器的值，获取目的寄存器的地址	计算结果	
OR	获取两个源寄存器的值，获取目的寄存器的地址	计算结果	
XOR	获取两个源寄存器的值，获取目的寄存器的地址	计算结果	
MIN	获取两个源寄存器的值，获取目的寄存器的地址	计算结果	
ADDI	获取一个源寄存器的值，一个立即数的值，获取 目的寄存器的地 址	计算结果	
ANDI	获取一个源寄存器的值，一个立即数的值，获取 目的寄存器的地 址	计算结果	
ORI	获取一个源寄存器的值，一个立即数的值，获取 目的寄存器的地 址	计算结果	
SLLI	获取一个源寄存器的值，一个立即数的值，获取 目的寄存器的地 址	计算结果	
SRLI	获取一个源寄存器的值，一个立即数的值，获取 目的寄存器的地 址	计算结果	

CTZ	获取一个源寄存器的值，获取目的寄存器的地址	计算结果	
SB	获取两个源寄存器的值，获取目的寄存器的地址	获取 offset，计算地址	内存存储
SW	获取两个源寄存器的值，获取目的寄存器的地址	获取 offset，计算地址	内存存储
LB	获取一个源寄存器的值，获取目的寄存器的地址	获取 offset，计算地址	内存读取
LW	获取一个源寄存器的值，获取目的寄存器的地址	获取 offset，计算地址	内存读取
LUI	获取目的寄存器的地址		
JAL	获取目的寄存器的地址，计算跳转地址，开始暂停流水线进行跳转		
JALR	获取目的寄存器的地址，计算跳转地址，开始暂停流水线进行跳转		
AUIPC	获取一个立即数 的值，获取目的 寄存器地址	计算 PC 与立即 数的和	
CSRRW	获取一个源寄存器的值	读 CSR 原值并计算	将新值写入 CSR
CSRRC	获取一个源寄存器的值	读 CSR 原值并计算	将新值写入 CSR
CSRRS	获取一个源寄存器的值	读 CSR 原值并计算	将新值写入 CSR
ECALL	解析指令	生成 mepc 和 mcause 的值	输出信号给 control，之后由 control 控制跳转 到异常处理程序地址处，同时连接输出信号给 csr 寄存器模块，修改相应值并按照指令功能 更改 mode，并将当前 PC 保存
EBREAK	解析指令	生成 mepc 和 mcause 的值	输出信号给 control，之后由 control 控制跳转 到异常处理程序地址

			处，同时连接输出信号给 csr 寄存器模块，修改相应值并按照指令功能 更改 mode，并将当前 PC 保存
MRET	解析指令		输出信号给 control，之后由 control 控制跳转 到用户程序地址处，同时连接输出信号给 csr 寄存器模块，修改相应值并按照指令功能更改 mode
sfence.vma	解析指令，输出 信号清空 tlb		
SBCLR	获取两个源寄存器的值，获取目的寄存器的地址	计算结果	

11.实验心得和体会

刘泓尊：我负责了本次实验的 Verilog 代码编写和仿真测试。因为有了我的小实验 5 较好的封装和模块化设计，我选择继续在其上开发，节省了很多时间。实际上，我们的基础指令和扩展指令部分，基本上是在扩展译码模块和 ALU 模块，没有太大的难度。到了后面的异常处理阶段，因为计划支持文档中全部的异常码，所以逻辑变得极其复杂，包含大量的逻辑判断，出现了众多时序冲突问题，我在优化时序上花了很大的时间。之后是页表和 TLB 的实现，在多周期框架下较为简单，将之前单状态访存改成 3 次访存即可。但实际上逻辑复杂之后又会带来时序的混乱。总结来看，优化大量的逻辑和复杂度是我本次实验花费之间最多的。其次，仿真是及其重要的，我在前期花了一些时间写好了 testbench, 为后面 debug 的顺利进行做足了准备。在编写代码中出现的“手残”错误和逻辑错误，几乎全部是通过仿真发现的。此外，本次实验培养了我基础的硬件思维，做到了“写代码时心中有电路”，也对课上的知识有了深刻的理解。感谢老师们和助教团队一学期的辛苦付出和悉心指导！

刘云昊：①软件思维到硬件思维的转变很重要，我们写出的很多代码并不是

按照我们预想的次序执行的，很多时候可能因为一些延迟达不到预期的效果，比如我们串口插了很多空拍，又比如我们修改 CPU 的时钟频率 50M 到 100M 或 11M 都会导致代码运行失败，时序在代码编写的过程中真的很重要；②通过这次实验，我学会了 verilog 仿真，并从仿真中发现了很多问题，感受到了硬件中仿真的重要性；③从这次实验中，我对 CPU 的各个阶段有了更深的了解，感谢老师和助教一个学期的指导与帮助！

张瀚天：经过此次造机实验，我对陈康老师所说的心中要有电路有了一些自己的理解，硬件编程与软件编程有很大的不同，硬件编程在代码执行的次序上是并发的而不是平常我所熟悉的按序执行的，在实验中踩了不少坑也逐渐理解了一些。另外，在造机实验中也学会了前几次小实验中并没有用到的仿真功能，也理解了仿真驱动开发的必要性。总的来说，经过此次实验，我对多周期 CPU 有了更深的了解，感谢老师与助教们的指导与帮助。