

Lab4: Single Cycle CPU

实验简介

这个实验要求你实现基本的单周期CPU，架构为RISC-V，指令集是RV32I的子集

RISC-V, YES!

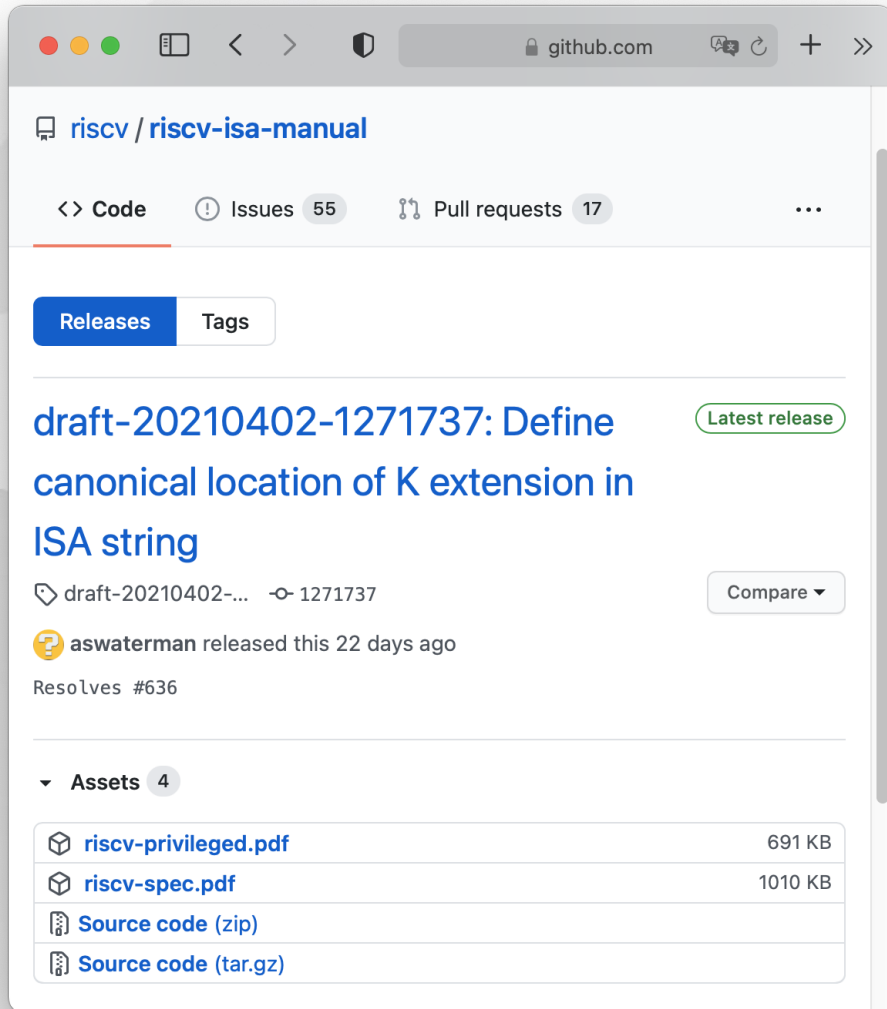
我们的CPU使用的架构是RISC-V。RISC-V诞生于最近十年，比起其他“老牌、庞大”的架构，RISC-V是一个“年轻”的架构。它的特点是**开源、简单、模块化**，使用RISC-V架构大大简化了硬件设计的流程。

如果对RISC-V有更大的兴趣，可以浏览RISC-V官网: <https://riscv.org>



永远是最好的武器


RISC-V获取官方ISA的成本很低，对开发者来说减少了踩坑的可能



一些其他的相关资料

- [偏实战向的基本RISCV编程](#)
- [中文RISC-V spec](#)
 - 里面有 ≥ 2 处错误，但我们的实验应该碰不到

RISC-V Emulator

- RISCV有非常多的Emulator版本可以选用。
 - GitHub搜索关键词[riscv emulator](#)
 - 有一些emulator需要一定的学习成本
- TA偏爱[Venus](#)
 - 部署在网页端，不需要本地支持
 - : “Venus还有VSCode插件的支持”

实验要求实现的指令(39条)

- lui, auipc, jal, jalr, lw, sw
- beq, bne, blt, bge, bltu, bgeu, mret
- addi, slti, sltiu, xori, ori, andi, slli, srli, srai
- add, sub, sll, slt, sltu, xor, srl, sra, or, and
- csrrw, csrrs, csrrc, csrrwi, cssrrsi, csrrci, ecall

🌟实验总共有3个部分，从最基本的SCPU到支持几乎全部RV32I指令的SCPU

🌟部分指令重合度很高，因此难度不大

中断和异常

Interruption & Exception

- 什么是中断 / 异常
- 为什么要中断 / 异常
- 在RISC-V中怎么处理中断 / 异常（以RV32I为例）
 - Control Status Register (CSR)
 - CSR指令
 - 如何写一段中断处理程序

什么是中断

- 中断是CPU内部或外部发生的一些事件
- 这些事件需要CPU特殊处理
- 例:
 - 按键中断
 - 时钟中断

什么是异常

- 异常是CPU在执行指令时发生的错误
- 这些错误也需要CPU处理
- 例:
 - 非法指令
 - PC地址不对齐
 - 读写地址错误(大名鼎鼎的*Segmentation Fault*)

为什么需要中断

- 你的操作系统依靠中断驱动
 - 双击鼠标 → **OS处理中断** → 打开了应用程序
 - 敲下键盘 → **OS处理中断** → 你敲下了一个字符
 - 插上USB → **OS处理中断** → 你连接了Sword板子

为什么需要异常

- 错误的程序不能扰乱正确的执行流程
 - 你的读写地址出错了: **Segmentation Fault**
- 如果没有异常
 - 錄板簪滸?拷涔塚禄鍊櫟紕鋁斤拷鋁斤拷鋁斤拷鋁剿达拷鋁斤拷鋁?16鋁秸慙拷幕 鋁斤拷鋁斤拷录冶鋁斤拷鋁斤拷?鋁斤拷鋁鋁斤拷鋁?03谢涛tql15鋁拷[鋁斤拷协十鋁斤拷斤拷鋁紬协鋁斤拷鋁节拷街?爸?鋁斤拷鋁鋁斤拷鋁?03鋁斤拷1拷鋁斤拷鋁斤拷鋁斤拷鋁拷鋁

在RISC-V中 怎么处理中断 / 异常

- Control Status Register (CSR)
- CSR指令
- 如何写一段中断处理程序

Control Status Register

- CSR是一组特殊寄存器，主要用来完成中断 / 异常的处理
- 在RISC-V中，最多可以定义4096个CSR ($2^{12} = 4096$)

riscv-spec.pdf – 页码: 75/236

31	20 19	15 14	12 11	7 6	0
csr	rs1	funct3	rd	opcode	
12	5	3	5	7	
source/dest	source	CSRRW	dest	SYSTEM	
source/dest	source	CSRRS	dest	SYSTEM	
source/dest	source	CSRRC	dest	SYSTEM	
source/dest	uimm[4:0]	CSRRWI	dest	SYSTEM	
source/dest	uimm[4:0]	CSRRSI	dest	SYSTEM	
source/dest	uimm[4:0]	CSRRCI	dest	SYSTEM	

Control Status Register

- 但不是每一个CSR都起作用，大部分都留空为了支持客制化
- 以RV32I-Machine模式为例，要支持中断，我们需要
 - **mtvec**: Machine Trap-Vector Base-Address Register
 - **mcause**: Machine Cause Register
 - **mepc**: Machine Exception Program Counter
 - **mstatus**: Machine Status Register

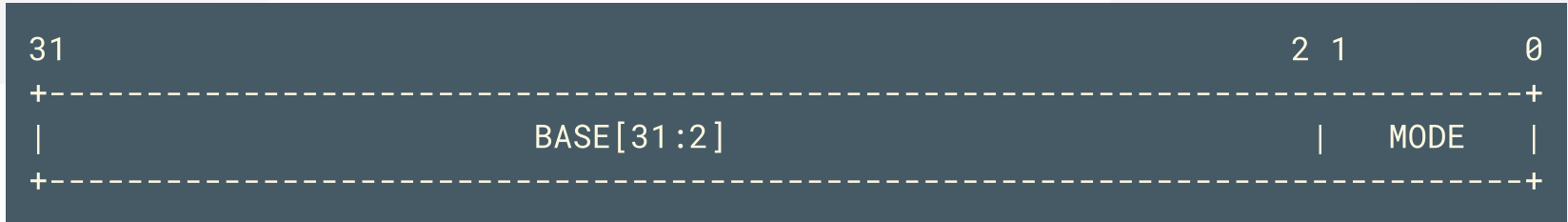
Control Status Register

- 此外，其他有关的寄存器包括：
 - **mie**: Machine Interrupt-Enable
 - **mip**: Machine Interrupt-Pending Register
 - **mtval**: Machine Trap Value Register

注: 下面介绍的都是机器行为

这些行为都是由CPU自动完成的，而不在写的汇编程序中完成。

mtvec



- 存放中断处理程序的入口地址
- 当中断 / 异常发生时，按照MODE字段的内容选择不同模式解析BASE字段，并将解析得到的地址加载进PC
 - mode=0: $pc \leftarrow BASE$
 - mode=1是Vector模式，比较复杂

mcause



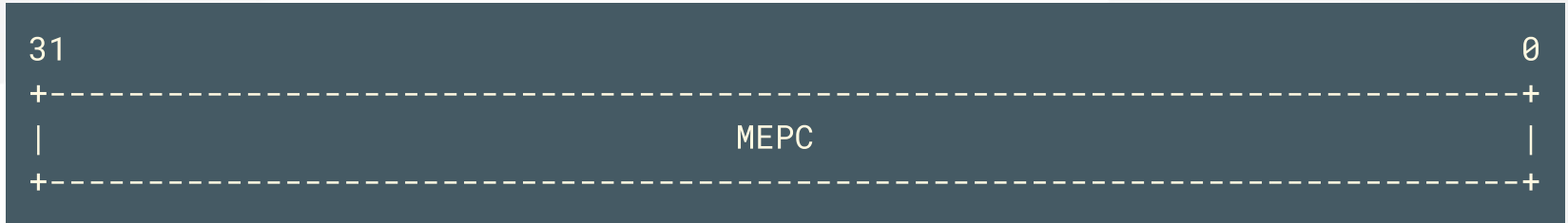
- Interrupt字段指明记录的mcause是中断还是异常
- Exception Code记录中断 / 异常号
 - 虽然叫'Exception Code', 但也会记录中断的代码
- 详情请参考ISA

mstatus



- Machine **P**revious Interrupt Enable & Machine Interrupt Enable
 - 与mie寄存器不一样，mstatus中的MIE是全局的中断使能
- 当发生中断/异常时， $\text{mstatus}[7] \leftarrow \text{mstatus}[3]$
- MRET时， $\text{mstatus}[3] \leq \text{mstatus}[7]$

mepc



- 记录发生异常/中断时的PC地址
- 当发生异常 / 中断时, $mepc \leftarrow$ 发生中断 / 异常时的pc
 - 例外: 对于pc非对齐异常由跳转指令引起的情况, 需要将跳转指令的目标地址送入mepc中
- 当MRET时, $pc \leftarrow mepc$

mepc

- 在中断处理程序中会检查mcause寄存器
 - 如果是异常， $mepc += 4$
 - 如果是中断，不变
 - （但不同的异常和中断有特殊处理的情况，我们的实验中只做最基本的处理）
- 🌟 换言之，对mepc的精细化的操作，是由软件完成的（这部分不属于机器的行为）

mtval

一个用来记录附加的异常信息的寄存器

- 当发生非法指令异常时，记录下这条指令的值
- 当发生PC非对齐异常时，记录下PC的值
- 当发生L/S地址非对齐异常时，记录下L/S的地址

mie



- mie标记各种中断使能
 - 如果关闭了某个bit，则无法响应对应的中断

mip





- mip记录某中断已经发生(pending)
 - 上述的mip中的位是只读的，不能由指令写入
 - mip.MSIP由memory-mapped方式写入

CSR及相关指令

- csrrw, csrrs, csrrc, csrrwi, cssrrsi, csrrci
 - 大部分都是与一个GPR交换，并做运算改变CSR
 - CSR指令具体的行为请阅读ISA
- mret
 - 用来从中断处理程序中返回

中断 / 异常处理程序

- 从易到难实现一个较完整的中断处理程序
- 大量使用了伪指令和gcc提供的语法糖
 - 考试时不能用 
-  这里中断处理程序的情景稍有不同。发生中断/异常时，机器把 $pc + 4$ 加载进mepc中（即发生中断/异常时的下一条指令）

中断处理程序-V1

在中断/异常到来时，跳到中断处理程序中

```
start:
    li      a0, 1 << 3           # mstatus.MIE = 1
    csrw    mstatus, a0          # 写入mstatus打开全局中断使能
    la      a0, handler          # 加载handler的地址
    csrrw    zero, mtvec, a0      # 写入mtvec中
    li      a0, 0x88             # mie.MEIE = 1 mie.MTIE = 1
    csrw    mie, a0
    ...                          # 等待中断

handler:    # 中断处理程序的入口(啥也不干)
    mret    # 直接返回
```

中断处理程序-V2

关闭中断使能，防止嵌套中断（中断/异常处理程序中发生中断）

```
handler:
    li      a0, 1<<3          # mstatus.MIE = 0
    csrc    mstatus, a0        # 清除mstatus.MIE位
    mret                    # 返回时，会自动恢复mstatus
```

中断处理程序-V3

保护寄存器，防止原有寄存器被中断 / 异常破坏

```
handler:
    addi    sp, sp, -124    # 开辟栈空间
    sw      x1, 0(sp)      # 压栈
    ...
    li      a0, 1<<3      # mstatus.MIE = 0
    csrc    mstatus, a0    # 清除mstatus.MIE位

    lw      x1, 0(sp)      # 弹栈
    ...
    addi    sp, sp, 124    # 调整栈指针
    mret
```

中断处理程序-V4

检测是中断或者异常

```
handler:
    addi    sp, sp, -124    # 开辟栈空间
    sw      x1, 0(sp)      # 压栈
    ...

    li      a0, 1<<3       # mstatus.MIE = 0
    csrrc   mstatus, a0    # 清除mstatus.MIE位

    csrr    a0, mcause      # 交换a0和mcause
    blt     a0, zero, handle_interruption # 根据最高位为符号位的性质判定
    b       handle_exception
```

中断处理程序-V4

中断对epc的特殊处理

```
handle_interruption:
    # 中断需要恢复
    csrr      a0, mepc
    addi      a0, a0, -4      # 所以epc需要-4
    csrw      mepc, a0

    b finish_handler
```


中断处理程序-V4

完成处理

```
handle_exception:
    ...
    b finish_handler

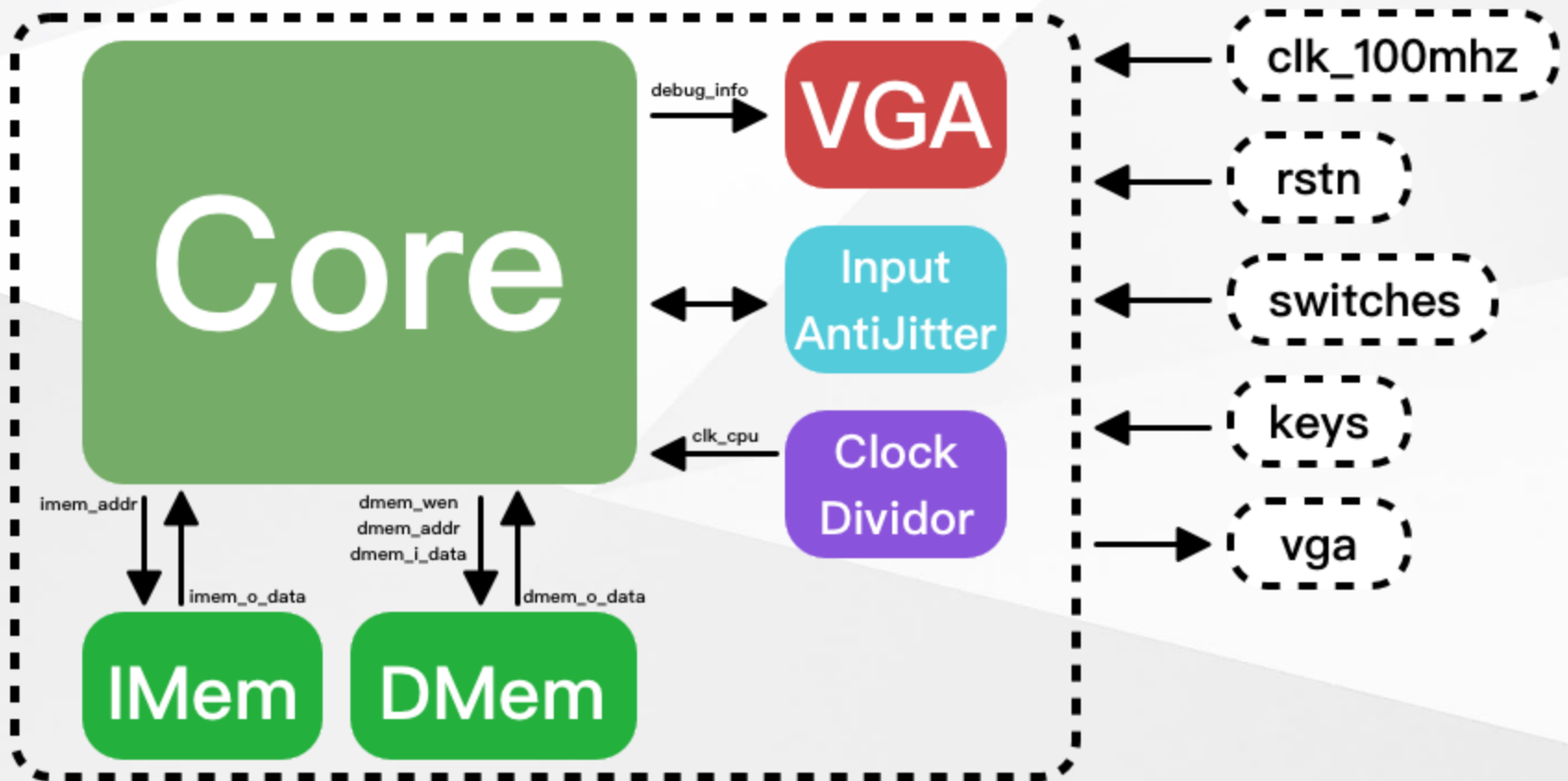
finish_handler:
    lw      x1, 0(sp)      # 弹栈
    ...
    addi    sp, sp, 124    # 调整栈指针
    mret
```

Core的最外层模块

根据DataPath的不同，Core有多种设计。

- 书上将IMem和DMem放在DataPath里，因此最外层模块只有一个Core
- TA的设计将IMem和DMem放在Core之外（参考Lab2课件）
- 采用任何一种都行，这两种方式只有代码上的些许不同，原理完全一致

Lab0中的顶层通路图



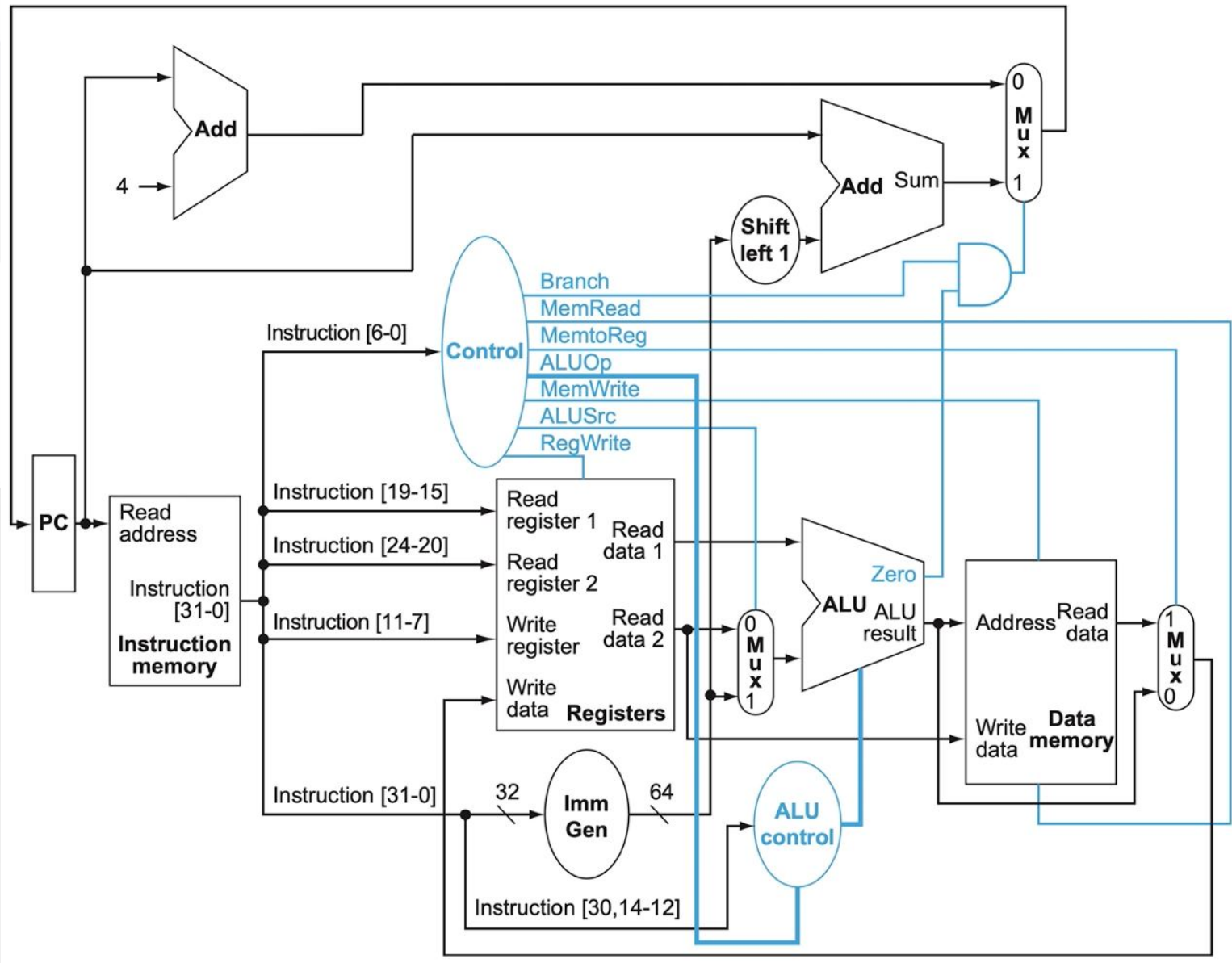
PART 1: 基本的SCPU

基本的SCPU

- 在这个版本的CPU中，我们只需要跟课本或课件上类似的SCPU即可。
- 需要支持以下指令：
 - lw, sw
 - beq, bne, blt, bge, bltu, bgeu
 - addi, slti, sltiu, xori, ori, andi, slli, srli, srai
 - add, sub, sll, slt, sltu, xor, srl, sra, or, and

设计DataPath

- 可以参考课本第257页和课件Chapter4
- 可以使用行为描述来实现简单的部件
 - 如使用三目运算实现MUX
 - 如使用Wire Bundle(`{A, B}`)实现ImmGen





设计DataPath

- 我们已经有的部件:
 - ALU
 - Comparator
 - RegFile
- 我们还需要部件:
 - DMem & IMem
 - Control Unit
 - 其他各类小元件

设计IMem和DMem

可以任选以下一种方式生成IMem和DMem:

- 使用Verilog语法糖（参考Lab3中的 `DataProvider`）
- 使用IP核（参考马德老师班的课件）
-  注意资源用量！ 2^{32} 是一个很大的资源占用量，板上没有如此多的资源
-  2^{12} 左右就足够使用了

设计Control Unit(Decoder)

- 是个组合逻辑部件
- Control Unit解码指令并根据指令产生多个控制信号
- 输入是当前的指令，输出是所有需要的控制信号

接入VGA信号

为了方便调试和观察CPU运行的结果，我们需要将寄存器内部的信号输出给VGA

在Core的模块定义中加入以下的宏声明

```
module Core(  
    `VGA_DBG_Core_Outputs /* 宏声明接口  
    input clk,  
    input rst,  
    ...  
);
```

顶层模块

```
`VGA_DBG_Core_Declaration
`VGA_DBG_RegFile_Declaration
`VGA_DBG_Csr_Declaration          // 增加CSR相关的接口

Core core(
    `VGA_DBG_Core_Arguments
    ...
);

VGA vga(
    `VGA_DBG_VgaDebugger_Arguments
    ...
);
```

VGA EDIF核可能遇到的问题

✦ 由于Vivado的编译器会对一些置空的线做优化，会造成EDIF核无法布线，因此需要保证EDIF核的接口不能置空

✦ 解决方案：对于暂时没有用到的VGA信号，可以直接用0置位。如：（在Core内部）

```
assign dbg_csr_wen = 1'b0;
```

上板测试

在实验材料中准备好了一组测试用例: test1, 其中包含了可以直接用于系统函数的 `.mem` 文件。

- `/test1/test_scpu1.s` 是汇编代码, 寄存器行为已经放在了注释中, 用于验证结果
- `/test1/imem_data.mem` 用于IMem
- `/test1/dmem_data.mem` 用于DMem

★ 注意更新了 `.mem` 后要手动重新运行一遍 Synthesis和Implementation

上板测试

这一步的测试结果与Lab1、Lab2中的也应当一致。

PART 2: 指令扩展

支持更多指令的SCPU

在这一步，我们考虑给之前已经完成的SCPU增加更多的指令支持(4条):

- lui
- auipc
- jal
- jalr

增加一条指令的基本思路

- 新增指令: 阅读ISA, 明白这条指令的行为
- DataPath: 能否复用目前的设计, 如果不行, 增加部件
- Control Unit: 是否需要增加更多的控制信号
- 测试: 新增测试样例, 看是否能够跑通测试

PART 3: 中断和异常

给SCPU加入CSR

- 只需要支持M模式的特权级别
- 新增6个CSR指令
- 新增7个CSR寄存器
- 新增7个异常/中断

新增的指令

- csrrw
- csrrs
- csrrc
- csrrwi
- cssrrsi
- csrrci
- ecall

🌟具体指令的行为参考ISA

CSR寄存器

需要支持的寄存器有


- mcause
- mepc
- mstatus
- mtval
- mtvec
- mie
- mip

需要支持的异常

- illegal instruction (不合法的指令)
- instruction address misalign (PC不是以4对齐)
- load address misalign (lw地址不是以4对齐)
- store address misalign (sw地址不是以4对齐)
- ecall from M (M模式下的ecall)

★ 异常不可恢复，从异常处理程序返回时要跳过发生异常的指令

需要支持的中断

- external interrupt (外部中断)
- timer interrupt (时钟中断)
-  中断需要恢复，从中断处理程序返回时要重新执行被中断的指令
- 这两处中断发生时，机器自动给 `mip` 的相应位置位

给Core增加两个中断信号的接口

external interrupt和timer interrupt的信号来源都在Core的外部，因此需要为Core增加两个外部输入

```
module Core(  
    `VGA_DBG_Core_Outputs  
    ...  
    input ext_int,           // 接key_x[1]  
    input tim_int            // 接clk_div[31]  
);
```

测试

✨ 除了官方的riscv-tools，还没有支持对CSR指令做汇编的工具，目前用来测试的CSR指令是由TA手动汇编的

🌟 如果自己写测试用例，需要在测试文件中添加中断和异常处理程序

`/test2/`里的测试样例是最终验收时的测试样例

验收

截止日期: 2021-06-07 23:59:59

```
Lab4_319010xxxx.zip
```

```
|— Lab4.bit      // 生成的bitstream二进制文件
|— Source       // 源代码
|   |— ...
|— Project      // 工程目录
|   |— ...
|— README       // 需要补充的说明（如果没有需要特殊说明则不需要此文件）
```

报告

截止日期: 2021-06-01 23:59:59

Lab4_319010xxxx.pdf // 单个pdf文件

Q & A