CS202 课程项目报告

本报告为 2025 年春季南方科技大学计算机系课程 CS202 计算机组成原理的课程项目报告

团队分工

> Lab: 周三78节 王薇老师

学号	姓名	分工	贡献比
12313023	黄思诚	pipeline实现及相关汇编	33%
12312030	魏国新	单周期cpu实现以及差分测试	37%
12313519	李思陈	上板测试以及测试场景汇编编写	30%

计划日程安排和实施情况

时间	任务
2025-4-23	project启动
2025-4-23	project整体规划和分工
2025-4-26	完成寄存器模块、立即数生成模块
2025-5-2	完成常量文件Constants的编写
2025-5-3	完成 ALU , BRU模块，pipeline完成EX模块
2025-5-5	完成Controller模块，pipeline完成ID模块
2025-5-11	pipeline完成WB模块代码
2025-5-14	进行中期答辩
2025-5-18	完成Memory模块，pipeline完成IF, MEM模块代码
2025-5-21	pipeline完成IF_ID, ID_EX等中间模块及hazard处理相关模块
2025-5-25	pipeline完成缓存相关模块
2025-5-31	pipeline完成CPU模块
2025-6-2	完成上板测试
2025-6-6	完成报告

CPU架构设计说明

- **寄存器**
  - 采用32位寄存器
  - 使用32个寄存器

    - 寄存器采用5bit地址
- **时钟频率:**
    - 单周期CPU
        - CPU: 最高可支持 10MHz
        - MEM: 50MHz
        - VGA: 25MHz
        - CPI 为1
    - Pipeline
        - **五级流水线** CPU，CPI 约为 2.1
        - CPU: 最高可支持 50MHz
        - MEM: 与 CPU 同频
        - VGA: 25MHz
- **寻址空间设计**
    - **冯诺依曼架构**
    - 寻址单位：寻址单位为 32 bit（4字节）
    - 存储空间：内存深度为 `16384`（索引范围 `0~16383`），每个存储单元是 32 位（4字节），总大小为 16384 × 4字节 = 64KB。
    - 栈空间基地址: **0x00007ffc**

ISA

参考 RISC-V 基本指令集 (RV32I) 及乘除法拓展 (RV32M)

指令	指令类型	执行操作
`add rd, rs1, rs2`	R	rd = rs1 + rs2
`sub rd, rs1, rs2`	R	rd = rs1 - rs2
`xor rd, rs1, rs2`	R	rd = rs1 ^ rs2
`or rd, rs1, rs2`	R	rd = rs1 | rs2
`and rd, rs1, rs2`	R	rd = rs1 & rs2
`sll rd, rs1, rs2`	R	rd = rs1 << rs2
`srl rd, rs1, rs2`	R	rd = rs1 >> rs2
`sra rd, rs1, rs2`	R	rd = rs1 >> rs2 (sign-extend)
`slt rd, rs1, rs2`	R	rd = (rs1 < rs2) ? 1 : 0
`sltu rd, rs1, rs2`	R	rd = ((u)rs1 < (u)rs2) ? 1 : 0
`addi rd, rs1, rs2`	I	rd = rs1 + imm
`xori rd, rs1, rs2`	I	rd = rs1 ^ imm
`ori rd, rs1, rs2`	I	rd = rs1 | imm
`andi rd, rs1, rs2`	I	rd = rs1 & imm
`slli rd, rs1, rs2`	I	rd = rs1 << imm[4:0]
`srli rd, rs1, rs2`	I	rd = rs1 >> imm[4:0]
`srai rd, rs1, rs2`	I	rd = rs1 >> imm[4:0] (sign-extend)
`slti rd, rs1, rs2`	I	rd = (rs1 < imm) ? 1 : 0
`sltiu rd, rs1, rs2`	I	rd = ((u)rs1 < (u)imm) ? 1 : 0
`lb rd, imm(rs1)`	I	读取 1 byte 并做符号位扩展
`lh rd, imm(rs1)`	I	读取 1 half-word (2 bytes) 并做符号位扩展
`lw rd, imm(rs1)`	I	读取 1 word (4 bytes)
`lbu rd, imm(rs1)`	I	读取 1 byte 并做 0 扩展
`lhu rd, imm(rs1)`	I	读取 2 byte 并做 0 扩展
`sb rd, imm(rs1)`	S	存入 1 byte
`sh rd, imm(rs1)`	S	存入 1 half-word (2 bytes)
`sw rd, imm(rs1)`	S	存入 1 word (4 bytes)
`beq rs1, rs2, label`	B	if (rs1 == rs2) pc += (imm << 1)
`bne rs1, rs2, label`	B	if (rs1 != rs2) pc += (imm << 1)

指令	指令类型	执行操作
`blt rs1, rs2, label`	B	if (rs1 < rs2) pc += (imm << 1)
`bge rs1, rs2, label`	B	if (rs1 >= rs2) pc += (imm << 1)
`bltu rs1, rs2, label`	B	if ((u)rs1 < (u)rs2) pc += (imm << 1)
`bgeu rs1, rs2, label`	B	if ((u)rs1 >= (u)rs2) pc += (imm << 1)
`jal rd, label`	J	rd = pc + 4; pc += (imm << 1)
`jalr rd, rs1, imm`	I	rd = pc + 4; pc = rs1 + imm
`lui rd, imm`	U	rd = imm << 12
`auipc rd, imm`	U	rd = pc + (imm << 12)
`mul rd, rs1, rs2` *	R	rd = (rs1 * rs2)[31:0]
`mulh rd, rs1, rs2` *	R	rd = (rs1 * rs2)[63:32]
`mulhsu rd, rs1, rs2` *	R	rd = (rs1 * (u)rs2)[63:32]
`mulhu rd, rs1, rs2` *	R	rd = ((u)rs1 * (u)rs2)[63:32]
`div rd, rs1, rs2` *	R	rd = rs1 / rs2
`rem rd, rs1, rs2` *	R	rd = rs1 % rs2

IO

- 使用 **MMIO** (Memory Mapping IO，内存映射) 进行 IO 操作并支持 UART
- 输入 (Input)
  - 1个按钮进行确认
  - 16个拨码开关（一组进行场景切换，一组输入测试数据）
- 输出 (Output)
  - 支持 16 个 LED 灯, 其中 8 个用于显示 CPU 状态
  - 8位数码管显示输出结果
  - VGA显示拨码开关读入数据

MMIO对应地址

地址	读/写	映射内容	取值范围 (省略前导0)
0xFFFFFF00	R	第 1 组拨码开关 (8 个)	0x00 - 0xFF
0xFFFFFF04	R	第 2 组拨码开关 (8 个)	0x00 - 0xFF
0xFFFFFF08	R	确认按钮	0x00 - 0x01
0xFFFFFF0C	W	第 1 组 LED (8 个)	0x00 - 0xFF
0xFFFFFF10	W	第 2 组 LED (8 个)	0x00 - 0xFF
0xFFFFFF14	W	数码管	0x00000000 - 0xFFFFFFFF
0xFFFFFF18	W	VGA	0x00 - 0xFF

CPU顶层接口

```
module Top (
    // clk -> cpuclk, memclk, vgaclk
    input                   fpga_clk, reset_n,
    // uart related
    input  logic            rx,
    // interact with devices
    input  logic [`SWCH_WIDTH ] switches1, switches2,// switches3,
    input  logic            Button_Mid,
    input  logic            Button_Up,
    //input  logic [`KBPIN_WIDTH] kp,
    output logic [`LED_WIDTH  ] led1_out, led2_out,// led3_out,
    //output logic [`LED_WIDTH  ] seg_en, seg_out,
    output logic [7:0] seg1,
                 logic [7:0] seg2,
                 logic [3:0] sel1,
                 logic [3:0] sel2,
    // vga interface
    output logic            hsync,          // line synchronization signal
    output logic            vsync,          // vertical synchronization
signal
    output logic [`COLOR_WIDTH] red,
    output logic [`COLOR_WIDTH] green,
    output logic [`COLOR_WIDTH] blue
);
```

单周期CPU 的输入信号包含：

- 一个 100MHz 的晶振时钟
- 高电平复位信号 rst
- UART 通信串口 rx
- 2 组拨码开关
- 1 个按钮（用来进行数据确认）

输出信号包含：

- 2 组 led
- 数码管信号
- VGA信号

CPU内部结构

ALU

```
module ALU(
    input logic [`DATA_WIDTH] InputA,           // First operand
    input logic [`DATA_WIDTH] InputB,           // Second operand
    input logic [`ALUOP_WIDTH] AluOperation,        // ALU operation control
    output logic [`DATA_WIDTH] Result     // Result of operation
);
```

ALU模块进行数据运算，输入两个32bit的数据，根据AluOperation控制信号选择运算操作，输出结果也是32位的运算结果。

Branch unit

```
module BRUSin(
    input logic [`DATA_WIDTH] Source1, Source2,  // Two source operands for
comparison
    input logic [`DATA_WIDTH] Pc,               // Current instruction PC
    input logic [`DATA_WIDTH] Immediate,                // Immediate value
(offset for branch)
    input logic [`BRUOP_WIDTH] BRUOperation,              // Branch operation
code (controls comparison type)
    input logic              Jalr,                // Jalr instruction flag
(whether it is a register jump)
    input logic reset,
    output logic [`DATA_WIDTH ] NewPc               // Directly output the next
instruction PC (no old PC needed for single-cycle)
);
```

用于处理分支指令的结果，在计算的下一条地址和Pc+4进行选择，可以理解为NextPc处理模块。

Constants

`Constants.vh` 存储常数便于同意管理

Controller

```
module Controller(
    /* verilator lint_off UNUSEDSIGNAL */
    input  logic [`DATA_WIDTH] Instruction,        // 32-bit Instructionuction
input
    /* verilator lint_on UNUSEDSIGNAL */
    output logic JumpLink,
    output logic [`BRUOP_WIDTH] BranchOperation,  // Branch unit control
    output logic [`ALUOP_WIDTH] AluOperation,     // ALU control code
    // output logic [`ALUSRC_WIDTH] AluSourceSelector,       // ALU operand2
selector    //0 reg, 1 imm, 3 pc+imm, 4 special
    output logic AluSrc1, //0 rs1 1 pc
    output logic AluSrc2, //0 rs2 1 imm
    output logic [`LDST_WIDTH] MemoryOperation,     // Memory access type
    output logic MemWriteEn,                  // Memory write enable
    output logic MemReadEn,                  // Memory read enable
    output logic RegWriteEn,                  // Register write enable
    output logic [1:0] Mem2Reg,                 // Data memory to register flag
```

```
    input logic CpuStart
);
```

控制信号生成单元。输入 32 bit 的指令，输出若干控制信号，分别控制CPU各个模块的正常工作。

ImmGenerator

```
module ImmGenerator(
    input logic [`DATA_WIDTH] Instruction,  // 32-bit input instruction
    output logic [`DATA_WIDTH] ImmData      // 32-bit generated immediate value
);
```

立即数生成模块，输入 32 bit 指令，根据指令类型生成符号扩展的 32 bit 立即数。

DataCache

```
module DataCache (
    input  logic             clk, reset,
    // cpu interface
    input  logic [`DATA_WIDTH] Address,
    input  logic [`DATA_WIDTH] WriteData,
    input  logic [`LDST_WIDTH] MemOperation,
    input  logic             MemRead,
    input  logic             MemWrite,
    output logic [`DATA_WIDTH] DataOut,
    // mem interface
    input  logic [`DATA_WIDTH] MemData,
    output logic [`DATA_WIDTH] MemAddress,
    output logic [`DATA_WIDTH] MemWriteData,
    output logic             MemWriteEnB
);
```

此模块并未实现缓存功能，只是直接从内存读写数据，同时在此模块处理了L型指令对于不同位宽数据的读写（根据内存现有数据和需要读写的数据将输出数据扩展成32位的数据）

```
module InstCache (
    input  logic             clk, reset,
    // cpu interface
    /* verilator lint_off UNUSEDSIGNAL */
    input  logic [`DATA_WIDTH] Address,
    /* verilator lint_on UNUSEDSIGNAL */
    output logic [`DATA_WIDTH] Instruction,
    // mem interface
    input  logic [`DATA_WIDTH] MemInstruction
    // output logic [`DATA_WIDTH] MemPc,
    // output logic uncached,
    // output logic hit
);
```

此模块并未实现缓存功能，只是直接从内存读取指令。

Memory

```
module Memory(
    input logic reset,                          // Global reset signal (active
high)
    input logic clkA, clkB,                     // Clock signals for port A and
port B
    input logic [`DATA_WIDTH] AddressA, AddressB, // Address buses for port A and
port B
    input logic [`DATA_WIDTH] WriteData,        // Write data for port B
    input logic EnableWriteB,                   // Write enable for port B
(active high)
    input logic [`SWCH_WIDTH] Switch1, Switch2,  // Input switches (MMIO)
    input logic Button_Confirm,
    //input logic [`VGA_ADDRESS] VgaAddress,          // VGA display address
(MMIO)
    output logic [`DATA_WIDTH] Seg1Out,          // 7-segment display output
(MMIO)
    output logic [`LED_WIDTH] Led1Out, Led2Out,  // LED outputs (MMIO)
    output logic [`LED_WIDTH] VGA1Out,
    output logic [`DATA_WIDTH] ReadDataA,        // Read data output for port A
    output logic [`DATA_WIDTH] ReadDataB         // Read data output for port B
    // output logic IsMMIO,                     // Indicates if the address is
MMIO
    // output logic IsExcept                     // Indicates if the address is
an exception
);
```

内存模块，输入输出包含内存相关信号，IO 相关信号。 IO 采用轮询的 MMIO 模式。

PC

```
module ProgramCounter(
    input clk,
    input reset,
    input logic [`DATA_WIDTH] PcInput,  // Input value of the program counter
    output logic [`DATA_WIDTH] PcOutput // Output value of the program counter
    );
```

Pc生成模块，根据branch unit模块产生的Pc值生成当前的Pc值。

RegisterFile

```
module RegisterFile(
    input logic clk,//neg
    input logic reset,
    input logic [4:0] ReadRegAddr1,ReadRegAddr2,// support read from two
Registers
    input logic [4:0] WriteRegAddr,// support write to one register
    input logic [`DATA_WIDTH] WriteData,// data to be written
    input logic RegWrite,
    output logic [`DATA_WIDTH] ReadData1,ReadData2// data read from Registers
    );
```

寄存器模块，输入包含时钟和复位信号，寄存器写入信号，和5 bit 位宽的读取寄存器的 rd1，rd2，写入寄存器的 WriteRegAddr。输出包含 32 bit 的从寄存器读取到的数据。

CPU_Clk

```
module CPU_Clk (
    input clk,
    output clk0
);
```

对时钟信号进行手动分频，可以产生的时钟信号的范围更大（ IP Clocking Wizard 能够生成的时钟范围仅为 6MHz 到 80MHz）。

复位信号

reset信号采用高电平有效，由按键复位信号和Uart数据传输信号协同控制，只有没有使用复位信号且Uart传输数据完成后CPU才会开始运行。

```
    logic RealReset ;
    assign RealReset = reset | ~UartOver;
```

方案分析说明

浮点数运算

一、硬件方案与软件方案定义

- **硬件方案**：通过专用电路（如扩展ALU、协处理器）直接实现功能，运算逻辑由硬件门电路完成。
- **软件方案**：通过汇编指令间接实现功能，依赖CPU现有指令集。

二、浮点数运算的典型差异分析

以**单精度浮点数加法**为例，对比两种方案的功能与性能：

1. **功能完整性**

- **硬件方案**：可直接支持IEEE 754标准的完整浮点运算（包括舍入、溢出处理等），功能原生且准确。
- **软件方案**：需通过整数运算模拟浮点操作（如拆分符号位、指数位、尾数位分别计算），易受限于CPU整数指令的精度和逻辑，可能无法完全覆盖所有边界情况（如非规格化数、无穷大）。

2. **性能（执行时间）**

- **硬件方案**：专用电路可并行处理浮点运算的各阶段（如指数对齐、尾数相加、结果规格化），单周期内完成运算。
- **软件方案**：需多条整数指令配合（如移位、加减、比较），完成一次浮点加法可能需要数十甚至上百个时钟周期，性能显著低于硬件。

3. **资源占用**

- **硬件方案**：需额外设计浮点运算单元（如浮点ALU），占用FPGA/ASIC的逻辑资源（如LUT、寄存器），增加芯片面积和功耗。
- **软件方案**：无需额外硬件，仅依赖现有CPU指令和内存，资源占用几乎为0（仅需存储库函数代码）。

4. **灵活性**

- **硬件方案**：功能固定，修改运算逻辑需重新设计电路（如调整舍入模式），灵活性差。

- **软件方案**：可通过修改库函数代码调整运算逻辑（如支持不同精度），灵活性高。

三、实验验证方法

实验目标：测量1000次单精度浮点数加法的总执行时间与资源占用。

实验步骤：

1. **硬件方案实现**：在单周期CPU中扩展浮点ALU，直接支持`fadd`指令，编写测试程序调用该指令完成1000次加法。
2. **软件方案实现**：用整数指令模拟浮点加法（拆分符号位、指数位、尾数，完成对齐、相加、规格化），编写测试程序调用该软件函数完成1000次加法。
3. **指标测量**：
   - 时间：通过CPU的`Pc_test`信号或仿真工具统计总时钟周期数。
   - 资源：通过FPGA综合工具统计硬件方案占用的LUT、寄存器数量；软件方案仅统计代码占用的指令内存大小。

预期结果：

- 硬件方案总周期数约为1000-2000周期（单周期/次），资源占用约500-1000个LUT。
- 软件方案总周期数约为100000-200000周期（约100周期/次），资源占用仅需存储几十条指令（约1KB代码）。

四、最终方案选择

结合单周期CPU的设计场景，**使用软件方案**：

- **选择软件方案**：若目标系统资源有限（如FPGA逻辑单元不足）或需支持灵活的浮点运算调整（如自定义舍入模式），软件方案更合适

总结：硬件方案以资源换性能，适合对速度要求极高的场景；软件方案以时间换灵活性，适合资源受限或需动态调整功能的场景。实际设计中需根据具体需求权衡。

乘法运算（RV32M扩展）的硬件实现 vs 软件实现

功能场景

单周期CPU需支持乘法指令（如`mul rd, rs1, rs2`），属于RISC-V的RV32M扩展。

硬件方案

- **实现方式**：在ALU中集成专用乘法器电路（如Booth算法乘法器），直接通过组合逻辑或流水线完成乘法运算。
- 功能特点：
  - 支持全精度32位乘法（结果为64位，取低32位）。
  - 运算结果符合RV32M标准（如符号处理、溢出忽略）。
- **性能**：单周期完成乘法（1个时钟周期），延迟极低。
- **资源占用**：需额外逻辑单元（如LUT、触发器）实现乘法器。

软件方案

- **实现方式**：通过软件库函数用加法和移位指令模拟乘法（如循环累加`rs1`共`rs2`次）。
- 功能特点：
  - 仅支持无符号乘法（或需额外逻辑处理符号位），可能无法覆盖所有边界情况（如大数相乘）。
  - 依赖CPU现有整数指令（`add`, `slli`等）。

- **性能**：完成一次32位乘法需32个时钟周期（每次移位加1次加法），总延迟约32周期。
- **资源占用**：无需额外硬件，仅需存储乘法函数的代码（约10-20条指令，占用指令内存）。

实验比较

- **实验方法**：编写测试程序计算1000次`mul t0, t1, t2`，分别用硬件乘法器和软件函数实现，统计总时钟周期数。
- **预期结果**：硬件方案总周期约1000周期，软件方案约32000周期（32周期/次）。

选择

- 采用硬件方案，所占有的逻辑资源并没有显出提高，但是显著降低了软件实现难度。

使用说明

详细使用步骤如下：

1. **创建 Vivado 项目**：通过 Vivado 创建一个 RTL Project，Project device 选择 ego1(与lab课相同)，Target Language设置为 VHDL，设置完毕并创建项目后，将所有Verilog 文件作为设计文件导入，再将 cpu_constr.xdc 作为约束文件导入
2. **创建 IP 核**
    - 创建 Clocking Wizard
        - 将组件名称改为 clockGen
        - 选择 PLL 时钟
        - 将 clk_in1 的 Source 修改为 Global buffer
        - 将 clk_out1 的频率设置为 25MHz 并取消 reset 信号和 locked 信号
    - 创建 Block Memory Generator
        - 将组件名称改为 Mem
        - Memory Type 选择 True Dual Port RAM
        - Port A 的 Write Width 修改为 32，Write Depth 修改为 16384 (Read Width, Read Depth 和 Port B 的相关参数会自动修改)
        - 取消勾选`Primitives Output Register`
3. **在 Vivado 中依次 Synthesis -> Implementation -> Generate Bitstream，将生成比特流文件 (.bit) 烧写进 FPGA**
4. **获取执行代码的机器码文件**：使用 RARS 打开需要执行的汇编代码，点击运行，再点击左上角 File，选择 Dump Memory，Dump Format 选择 Hexadecimal Text，点击 Dump To File... 并输入文件名后保存
5. **获取 UART 串口传输的文件**：将上一步得到的文件放在指令转换脚本 inst2txt.py 同一目录下，打开 inst2txt.py 将第 4 行的`filename`改为上一步所得的文件的名称，运行脚本，得到一个 .txt 文件 (如 test.txt)，这是要通过 UART 串口传输给 CPU 运行的指令
6. **通过 UART 加载程序并运行**：打开串口工具 UARTAssist.exe，串口号选择 COM6 (一般来说直接选能选的最后一个)，波特率设置为 115200，打开连接，发送选项选择 “按十六进制发送” 并 “启用文件数据源...”，选择上一步得到的 .txt 文件并确定，然后点击发送，发送完毕后 CPU 将会自动开始运行

测试说明

本项目使用了差分测试，采用verilator仿真+Unicorn模拟risc-vCPU进行测试。同时我们采用了risc-gnu-toolchain进行汇编指令的二进制代码的生成。

同时使用了shell脚本和make工具对于测试进行集成和部署。

测试方法	测试类型	测试用例描述	测试结果
Verilator仿真	单元测试/ALU	使用Verilartor仿真器，使用STL随机数生成器生成ALU的两个操作数，再随机生成ALUOp测试指令类型。同时将操作数和ALUOp输入到ALU单元和C++代码，生成两个结果并对比，C++的输出作为参考值。若不相同，输出输入数据和答案对比。	通过
Verilator仿真	单元测试/ImmGenerator	使用Verilartor仿真器。用C++编写ImmGenerator模块，其输出作为参考值。将数据同时输入到C++和模块中，得到并对比输出。若不相同，输出输入数据和答案。	通过
Verilator仿真	单元测试/Memory	使用Verilartor仿真器。该测试对象是Cache之前的Memory，包括IP RAM和sb/sh/sw/lb/lbu/lh/lhu/lw的逻辑正确性。由于Verilator不支持Vivado的IP核仿真，于是编写一个模拟IP RAM的模块。	通过
Verilator仿真	集成测试/CPU	使用Verilartor仿真器测试汇编代码。导入二进制文件并模拟UART行为写入内存。然后同时运行CPU和Unicorn模拟器，使用VPI读取寄存器并对比二者寄存器的值，相同则通过测试。	通过
上板测试	单元测试/UART	测试UART写入IP RAM的功能，并把数据输出到数码管上显示。	通过
上板测试	集成测试/Testcase 1	测试CPU功能与Testcase测试场景1汇编代码的正确性。	通过
上板测试	集成测试/Testcase 2	测试Testcase测试场景2汇编代码的正确性。	通过

测试结论

测试全部通过。先仿真确保行为逻辑正确，再进行上板调试时序问题，尽量减少上板测试的次数。测试在开发中占大部分比例，因此减少测试开销是十分必要的。

Bonus 相关说明

Pipeline

Pipeline采用**五级流水线**的结构，对hazard进行了处理同时引入了指令缓存 (ICache) 和数据缓存 (DCache)，涉及的模块如下

- Stage_IF, Stage_ID, Stage_EX, Stage_MEM, Stage_WB 为pipeline的五个阶段的模块
- IF_ID, ID_EX, EX_MEM, MEM_WB为中间寄存模块
- Hazard, Forward 为解决数据冒险的控制流水线前递和停顿的模块
- Branch_Predictor 为解决控制冒险的分支预测模块
- ICache、DCache 分别为指令和数据缓存，为提升与内存交互的效率的缓存模块

Hazard

```
module Hazard (
    input  logic [`REGS_WID] IF_ID_rs1, IF_ID_rs2, ID_EX_rd,
    input  logic             ID_EX_MemRead,
    output logic             stall, IF_ID_Write, PC_Write  // 1 stall, 0 not
stall
);
```

Hazard 模块用于判断是否存在因与内存交互而需停顿一个时钟周期的数据冒险，

判断逻辑为如果 `ID_EX_MemRead & ((ID_EX_rd == IF_ID_rs1) | (ID_EX_rd == IF_ID_rs2))` 成立，则需要停顿，其余情况无需停顿。

Forward

```
module Forward (
    input  logic [`REGS_WID] ID_EX_rs1, ID_EX_rs2, EX_MEM_rd, MEM_WB_rd,
    input  logic             EX_MEM_RegWrite, MEM_WB_RegWrite,
    output logic [`FW_WID  ] fwA, fwB // 00: no fwd, 01: from MEM/WB, 10: from
EX/MEM
);
```

前递模块，输入相关信号，输出是否前递的控制信号. 其中fwA和fwB分别控制ALU的src1和src2。以 ALU 的 src1 输入数据为例，src1 数据的前递选择器逻辑为，若 `EX_MEM_RegWrite & (EX_MEM_rd != 0) & (EX_MEM_rd == ID_EX_rs1)` 成立，那么将从 MEM 阶段前递数据，若 `MEM_WB_RegWrite & (MEM_WB_rd != 0) & ~(EX_MEM_RegWrite & (EX_MEM_rd != 0) & (EX_MEM_rd == ID_EX_rs1)) & (MEM_WB_rd == ID_EX_rs1)` 成立，将从 WB 阶段前递数据，其余情况均不需要前递。

Hazard处理部分测试代码如下:

```
.text
    # Data hazard
    li t0, 300
    li t1, 200
    addi t2, t1, 100
    # Control hazard
    beq  t2, t0, branch

not_branch:
    add  t3, t1, t2
    j    display

branch:
    addi t3, t2, 5

display:
    li   t4, 0xFFFFFF14
    sw   t3, 0(t4)
```

预期的正确输出应为0x00000131,即十进制下的305

ICache

```
module ICache #(
    parameter CACHE_WID = 4
)(
    input  logic             clk, rst,
    // cpu interface
    input  logic [`DATA_WID] addr,
    input  logic             predict_fail,
    output logic [`DATA_WID] inst,
    output logic             icache_stall,
    // mem interface
    input  logic [`DATA_WID] mem_inst,
    output logic [`DATA_WID] mem_pc
);
```

指令缓存模块用于管理指令的读取，缓存格式如下。 考虑到访问时延大，采用直接映射缓存，每个Block存储4 Words。如果 `tag` 不一样则缓存未命中(Cache Miss)，那么将流水线停顿并从内存中读取数据，输出并存入缓存。ICache停顿不会阻塞后续阶段的执行。

DCache

```
module DCache #(
    parameter CACHE_WID = 4
)(
    input  logic             clk, rst,
    // cpu interface
    input  logic [`DATA_WID] addr,
    input  logic [`DATA_WID] write_data,
    input  logic [`LDST_WID] MEMOp,
    input  logic             MemRead,
    input  logic             MemWrite,
    output logic [`DATA_WID] data_out,
    output logic             dcache_stall,
    // mem interface
    input  logic [`DATA_WID] mem_data,
    output logic [`DATA_WID] mem_addr,
    output logic [`DATA_WID] mem_write_data,
    output logic             mem_web
);
```

数据缓存需注意一下 3 点：

- 数据缓存原理与指令缓存相似，但与内存的交互不仅限于读取，还包括写入
- 如果缓存未命中，则停顿整个流水线
- 若地址为 MMIO 相关，将无需经过缓存判断命中与停顿，直接与内存进行读取和写入
- 同时，由于寻址单位为 32 bit，`lb`, `lh`, `sb`, `sh` 等指令需对数据进行切片，数据缓存中也需要相应的处理这些情况

对于比较pipeline和单周期cpu的性能，采用了循环的方式，提供比较得到最终结果所需要的时间来评价性能，并根据输出结果来判断是否正确运行，测试代码如下：

```
.text
    li   t0, 100000
    li   t1, 0

loop:
    add  t1, t1, t0
    addi t0, t0, -1
    bnez t0, loop

    li   t2, 0xFFFFFF14
    sw   t1, 0(t2)#0x2a06b550
```

auipc

auipc类型的指令首先在ID阶段被解码，如何生成控制信号供后续模块使用和处理

```
assign rd = inst[11:7];
```

```
`AUIPC_OPERATION: begin
                ALUOp    = `ALU_ADD;
                BRUOp    = `BRU_NOP;
                Jalr     = 0;
                ALUSrc   = 3;
                MemWrite = 0;
                MemRead  = 0;
                RegWrite = 1;
                MemtoReg = 0;
                MEMOp    = 0;
            end
```

然后EX阶段根据ALUSrc，将pc作为src1，imm作为src2，进入ALU进行加法操作，在后续将结果写入寄存器

```
assign src1 = ALU_src[1] ? pc : src1_mux;

    // source of ALU and write address
    always_comb begin
        unique case (ALU_src)
            2'b00: src2 = src2_mux;
            2'b01: src2 = imm;
            2'b10: src2 = 4;
            2'b11: src2 = imm;
        endcase
    end
```

测试场景为执行auipc并观察是否能够得到正确的结果。测试代码如下：

```
.text

    auipc t0, 1
    addi  t0, t0, 32

    li   t1, 0xFFFFFF14
    sw   t0, 0(t1)
```

UART

出于兼容性的考虑，本项目并未使用课程提供的IP核而是选择自己编写。本项目的 UART 波特率为 115200，参数为 8 bit 数据配合 1 bit 停止位。 由于寻址单位为 32 bit，因此需要一个队列来实现接受并计数 4 byte 数据后统一存进内存。

```
module Queue (
    input  logic        clk, reset,
    input  logic [7:0]  DataIn,
    input  logic        Ready,
    output logic [`DATA_WIDTH] DataOut,
    output logic [`DATA_WIDTH] Address
);
```

队列能够将 4 个 8 bit 的数据拼接并计算需要存入内存的内存地址。

```
module Uart (
    input                   clk, reset, rx,
    output logic [`DATA_WIDTH] DataOut,
    output logic [`DATA_WIDTH] Address,
    output logic            Done
);
```

UART 负责接收端口 `rx` 传入的信息，借助队列解析成需要的数据并存入内存，同时有一个代表是否传输完毕的信号。

VGA实现

VGA 采用字符映射的方式实现，即全屏共可以显示 96 × 32 个字符，每个字符有两种状态(亮/暗)。 在内存中，与其他 IO 一样，VGA 也采用 MMIO，字符编号对应的地址为 0xFFFFEXXX，颜色对应的地址为 0xFFFFDXXX，在 Memory 中被称为缓冲区。 可以通过汇编往缓冲区中写入，从而将字符显示在屏幕上。

```
module VGADisplay (
    input wire clk,
    input wire reset_n,
    input wire [7:0] data, // datain
    output hsync,   // line synchronization signal
    output vsync,   // vertical synchronization signal
    output reg [3:0] red,
    output reg [3:0] green,
    output reg [3:0] blue
);
```

开源及AI的启发和帮助

在缓存设计时参考了网络中的缓存设计，这些代码为代码实现提供了参考和思路；在搭建CPU模块时使用AI根据子模块生成CPU模块，提高了效率。

但AI对信号间在各个模块间的传输存在错误，需要追踪信号并进行手动修改。

Bonus 部分的问题及解决方案

- 指令 `auipc` 的实现。
  - **原因**: 指令 `auipc` 需要进行 pc 相关的计算，而 ALU 没有相关数据的输入
  - **解决方案**: 在 ALU 输入 rs1 的端口前添加选择器，对 pc 和 rs1_data 进行选择，同时拓宽控制信号 ALUSrc
- 缓存的实现。
  - **原因**: 只侧重于基本的缓存操作和状态机实现，缓存结构简单难以满足需求
  - **解决方案**: 使用脏位（dirty bit）来区分修改过的数据；针对特定地址（如 mmio）的未缓存访问做出了判断处理，确保对于特殊外设映射的访问不经过缓存；增加状态数进行更精细的管理

总结

在开发过程中，我们遇到了一些问题，并对这些问题进行了思考和总结。以下是我们的经验和教训：

1. **仿真与实际硬件的差异**：仿真环境与实际硬件之间存在差异，即使仿真结果正确，上板后也可能遇到各种问题。这可能是因为硬件调试需要考虑更多的实际因素，如信号完整性、电源稳定性等。因此，我们应该合理安排时间，留出充足的调试时间。
2. **硬件连接和信号定义**：上板后如果出现问题，首先检查顶层模块的接线是否正确，信号的输入输出方向和位宽是否符合预期。此外，还需要检查是否存在隐性的多驱动器问题，以及时钟频率是否设置得过快，因为仿真通常不考虑延迟。
3. **时序逻辑和复位信号**：在设计时序逻辑时，需要明确是在时钟的上升沿还是下降沿更新状态。同时，需要确认复位信号是高电平触发还是低电平触发，并与板子的用户手册相匹配。
4. **枚举性工作的细致性**：在实现CPU的过程中，有很多枚举性的工作，如ALU操作、控制信号等。这些部分需要非常仔细地设计和检查，因为一旦出现问题，调试将会非常困难。
5. **团队合作的重要性**：在整个项目过程中，从设计和架构到细节实现，再到测试和上板，团队成员之间的充分交流和沟通至关重要。有效的沟通可以确保项目的顺利进行，并减少错误和返工。
6. **工具的选择和使用**：Vivado工具可能在某些方面不够直观，可以考虑使用其他工具如Verilator仿真器和Unicorn进行CPU的差分测试，以提高开发效率。
7. **全面考虑设计**：在设计阶段，尽可能全面地考虑各种情况，尽管这可能会花费更多的时间。例如，我们单周期CPU项目最初考虑使用缓存，但后来由于实现难度问题取消了缓存功能，这导致了架构上的较大修改，既麻烦又容易出错。

通过这些经验，我们认识到在开发过程中，细致的工作、充分的准备和团队合作是成功的关键。同时，选择合适的工具和方法可以大大提高开发效率和质量。

本文档编写参考MineCPU的Report并由ai提供了帮助。