****

**《计算组成原理实验》**

**实验报告**

**（实验一）**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **学院名称** | **：** | 计算机学院 | | | | | |
| **专业（班级）** | **：** | 21级行政1班 | | | | | |
| **学生姓名** | **：** | 黄鑫 | | | | | |
| **学 号** | **：** | 21307008 | | | | | |
| **时间** | **：** | 2023 | 年 | 1 | 月 | 11 | 日 |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  |  |  | **成绩** | **:** |  |
| **实验一** | **：** | **单周期CPU（YstCPU Risc-V）** | | | |

**目录**

[第一章 实验目的 3](#_Toc125056158)

[第二章 实验器材 3](#_Toc125056159)

[第三章 实验原理与分析 3](#_Toc125056160)

[1. 周期的概念 3](#_Toc125056161)

[2. 指令集 3](#_Toc125056162)

[3. 典型指令的具体细节 4](#_Toc125056163)

[4. 数据通路与控制线路 6](#_Toc125056164)

[5. 指令的执行流程 8](#_Toc125056165)

[第四章 实验过程与结果 10](#_Toc125056166)

[1．InstructionFetch 10](#_Toc125056167)

[2. InstructionDecode 11](#_Toc125056168)

[3. Execute 12](#_Toc125056169)

[4. CPU 12](#_Toc125056170)

[5. 标准测试 13](#_Toc125056171)

[6. 自定义测试 14](#_Toc125056172)

[第五章 实验心得 16](#_Toc125056173)

[1. 查手册 16](#_Toc125056174)

[2. 钻研指令执行流程 16](#_Toc125056175)

# 实验目的

1. 掌握单周期CPU的基本构造，包括数据通路、控制单元；

2. 掌握单周期CPU的代码实现方法；

3. 认识单周期CPU是如何执行各种指令的；

4. 掌握自定义应用测试方法；

# 实验器材

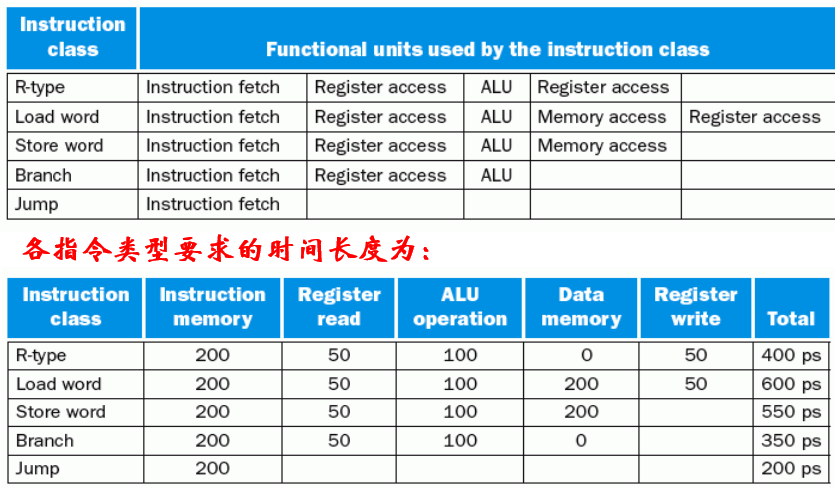
电脑一台，Intellij IDEA 软件一套。

# 实验原理与分析

实验提供了CPU的部分代码，在理解了单周期CPU的数据通路、控制单元与指令执行流程之后，将单周期CPU的实现代码补充完整。然后自行编写并编译自己的程序，在单周期CPU上运行。

## 周期的概念

指令周期是指一条指令从取出到执行完成的时间。不同指令的执行时间不同（如下图），在单周期CPU中，指令周期取所有指令的指令周期中的最长者，即600ps。由于时钟周期是固定的，所以执行所有指令都必须和执行最慢指令耗费一样的时间，这导致单周期 CPU 性能很差，但单周期CPU每次只运行一条指令，最容易实现。



一个指令周期由若干个时钟周期组成，后者是CPU操作的基本单位。在单周期CPU中，时钟周期等于指令周期。

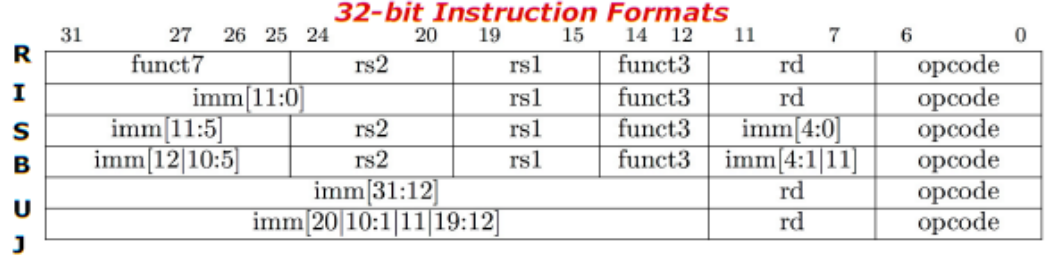
在单周期CPU中，在一个时钟周期中CPU便要完成指令五大阶段：取值IF、译码并访寄存器组ID、执行EX、访存MEM与写回寄存器组WB。

## 指令集

本实验支持的指令集是 [RISC-V](https://riscv.org/) 的 RV32I 整数运算指令集。

RV32I的六种基本指令类型：

|  |  |  |
| --- | --- | --- |
| R 类型指令 | 用于寄存器-寄存器操作 | sll/srl/sra/add/sub/xor/or/and/slt/sltu |
| I 类型指令 | 用于立即数和访存 load 操作 | slli/srli/srai/addi/xori/ori/andi/slti/  sltiu/jalr/fence/fence.i/ecall/ebreak/  csrrw/csrrs/csrrc/csrrwi/csrrsi/csrrci/  lb/lh/lbu/lhu/lw |
| S 类型指令 | 用于访存 store 操作 | sb/sh/sw |
| B 类型指令 | 用于条件跳转操作 | beq/bne/blt/bge/bltu/bgeu |
| U 类型指令 | 用于长立即数 | lui/auipc |
| J 类型指令 | 用于无条件跳转 | jal |



|  |  |
| --- | --- |
| opcode | 操作码 |
| rs | 源操作数寄存器地址，占5位（因为有32个寄存器） |
| rd | 目的操作数寄存器地址，占5位（因为有32个寄存器） |
| funct3、7 | 功能码，在指令中配合操作码使用，用于指定指令的功能 |
| imm | 立即数 |

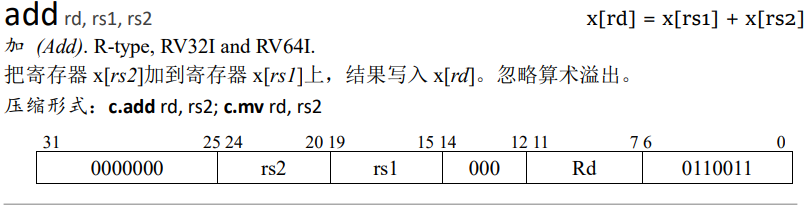
## 典型指令的具体细节

出于实现上的便利，本实验结合指令的功能与构造，将指令划分为以下：

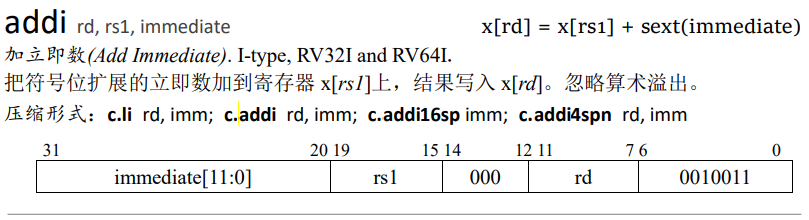
|  |  |
| --- | --- |
| L指令 | *lb/lh/lw/lbu/lhu* |
| I指令 | *add/slli/slti/sltiu/xori/sri/ori/andi* |
| S指令 | *sb/sh/sw* |
| R指令 | *add\_sub/sll/slt/sltu/sr(sra&srl)/xor/or/and* |
| B指令 | *beq/bne/blt/bge/bltu/bgeu* |
| CSR指令 | *csrrw/csrrs/csrrc/csrrwi/csrrsi/csrrci* |
| 杂项 | *nop/mret/ret/ecall/ebreak/csr/lui/auipc/fence/jal/jalr* |

下面展示一些典型例子，以便之后的程序分析（如果有若干个指令在控制信号译码、执行流程上相同，则只取其中一种）：

（1） R-type

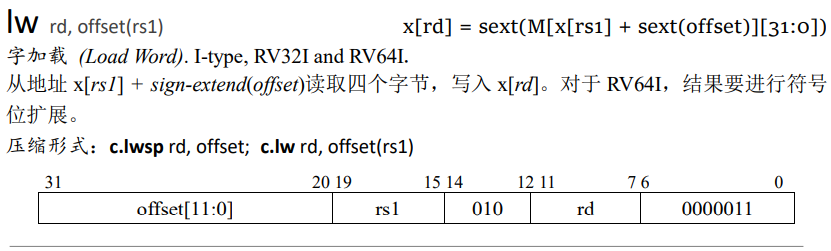


（2）I-type

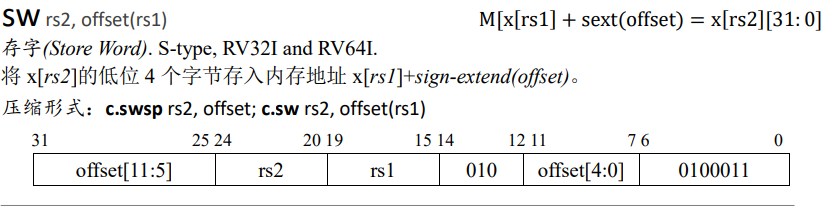


（3） L-type

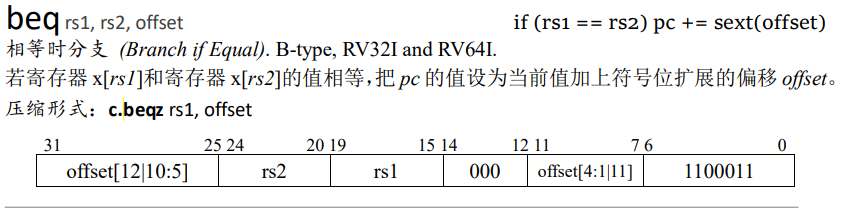
本实验未实现中断，因此没有实现CSR相关指令。



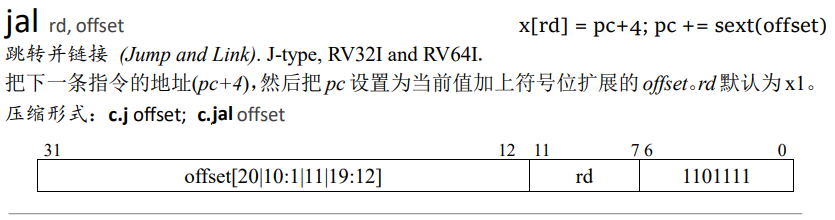
（4） S-type

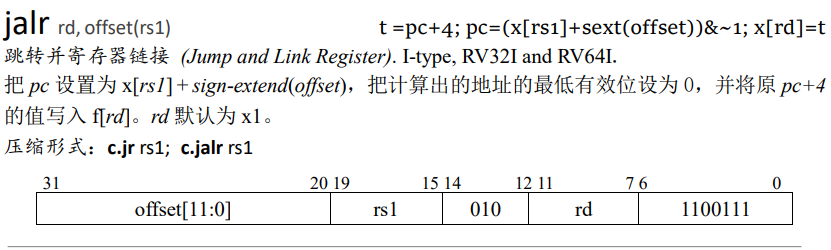


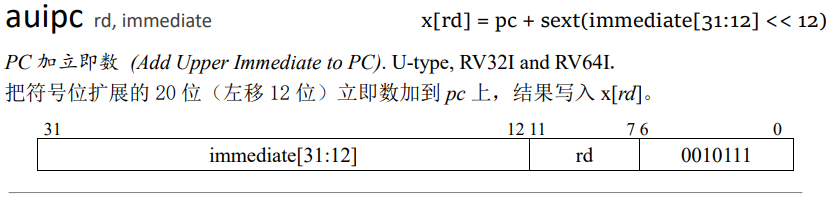
（5） B-type

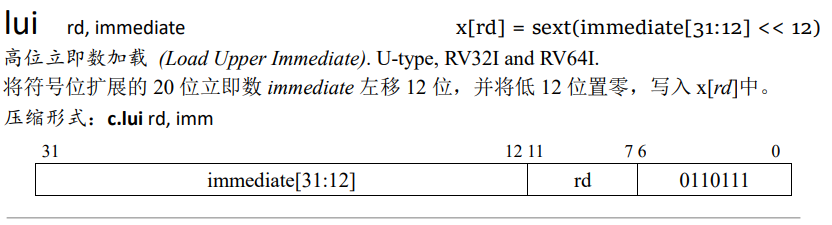


（6） 杂项

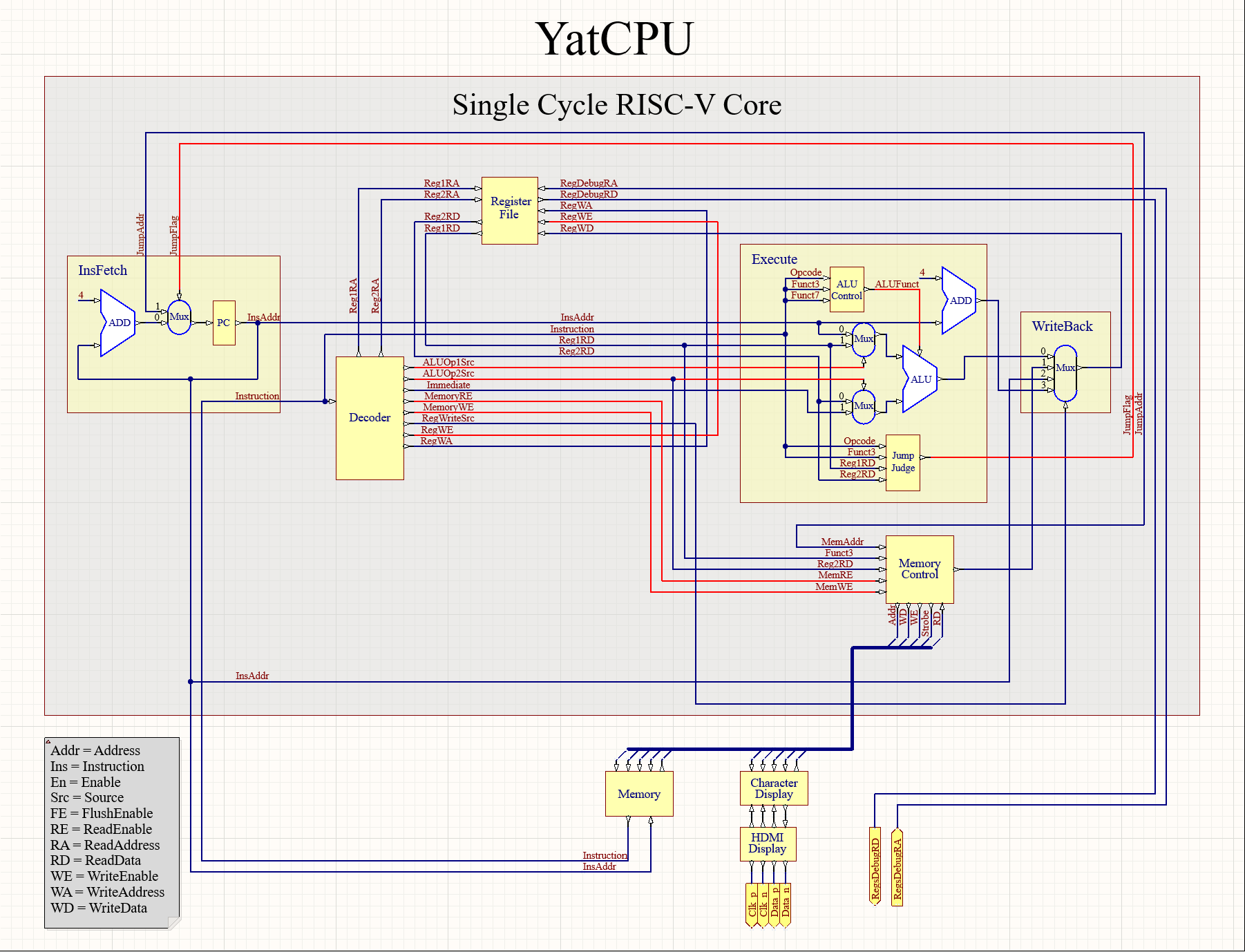








## 数据通路与控制线路



上图有一些接线错误，下面是更正：

1. InsFetch模块JumpAddr线的另一端应该连接Execute模块中的一个JumpAddr计算单元的输出，该单元在图中没有画出。
2. Memory Control模块的MemAddr线应该连接上ALU的输出。
3. Memory Control模块的Funct3应接在Instruction上。

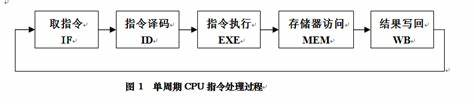
控制信号的解释

|  |  |
| --- | --- |
| 控制信号 | 含义 |
| JumpFlag | 是否跳转，控制PC的输入Mux：   1. 为真则PC获取跳转地址 2. 否则PC正常“+1”   （这里是32为指令字长且按字节编址，所以+4） |
| ALUOp1Src | 指示ALU操作数1的来源是当前指令地址还是源寄存器1的值 |
| ALUOp2Src | 指示ALU操作数1的来源是源寄存器2的值还是译出的立即数 |
| MemoryRE | 内存读使能，在Risc-v中是load指令的标志 |
| MemoryWE | 内存写使能，在Risc-v中是store指令的标志 |
| RegWriteSrc | 写寄存器的数据来源选择，控制一个四路Mux，选项为ALU的输出、读内存的输出、CSR的跳转地址（下一lab实现）、顺序下条指令地址（pc+4，来自jal或jalr） |
| RegWE | 寄存器写使能，出现在有目标寄存器的指令 |
| Opcode | 指示指令类型 |
| Funct3 | 指示指令类型 |
| Funct7 | 指示指令类型 |
| ALUFunct | 指示ALU的运算法 |

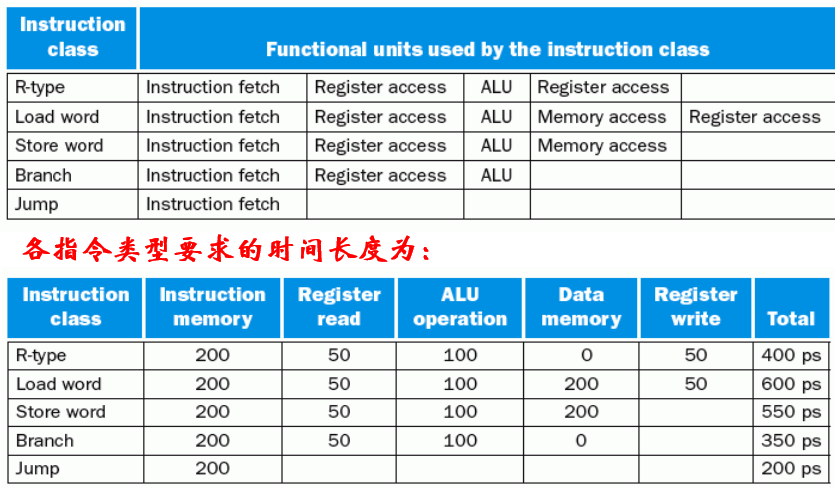
模块接口的解释

|  |  |
| --- | --- |
| InsAddr | 当前指令的地址 |
| Instruction | 当前指令 |
| Reg1RA | 源寄存器1地址 |
| Reg2RA | 源寄存器2地址 |
| Immediate | 从指令中译码出来的立即数 |
| Reg1RD | 源寄存器1的值 |
| Reg2RD | 源寄存器2的值 |
| RegDebugRA | 读寄存器组地址，用于测试程序获取寄存器组的情况 |
| RegDebugRD | 读寄存器组结果，用于测试程序获取寄存器组的情况 |
| RegWA | 寄存器写地址，出现在有目标寄存器的指令 |
| RegWD | 寄存器写输入，出现在有目标寄存器的指令 |
| JumpAddr | 跳转目标地址 |
| MemAddr | 访存地址 |

## 指令的执行流程



但是不同指令的执行流程不尽相同，有些指令不需要执行某些过程，见下图：



（1） InsFetch是公共取指模块，在IF阶段中负责将PC寄存器（program count）的值作为地址在内存中找到将要执行的指令。由于访存较慢，IF阶段用时平均约200ps。

（2）在InsDecode阶段中，译码器Decoder的输入为IF阶段的读内存结果，即一条具体的指令，随后译码器将指令译码，得到各种后续执行所需的控制信号。在单周期CPU中，这些控制信号将一直保持到该时钟周期结束。接着ID阶段还要完成读寄存器组的任务（如果该指令有源寄存器字段）。ID阶段用时约50ps。

（3） Execute模块执行EX阶段的任务，任务包括：使用指令中的Opcode、Funct3和Funct7字段与ALUControl单元译码确定ALU的运算法ALUFunct、用Decoder译码结果ALUopSrc1和2与两个二路选择器Mux确定ALU的两个操作数输入、输出ALU计算结果、判断是否跳转、计算跳转地址。EX阶段用时约100ps。

下面用四个表格说明这四项工作。

下面是两个ALU相关译码表。出于描述的易读性考虑，用常量名add、sll等代替ALUFunct、opcode、funct3具体的值。其中a表示op1，b表示op2。

ALUControl译码表

|  |  |  |  |
| --- | --- | --- | --- |
| Opcode | funct3 | funct7(5) | ALUFunct |
| I | addi | - | add |
| slli | - | sll |
| slti | - | slt |
| sltiu | - | sltu |
| xori | - | xor |
| ori | - | or |
| andi | - | and |
| sri | 1 | sra |
| 0 | srl |
| RM | add\_sub | 1 | sub |
| 0 | add |
| sll | - | sll |
| slt | - | slt |
| sltu | - | sltu |
| xor | - | xor |
| or | - | or |
| and | - | and |
| sr | 1 | sra |
| 0 | srl |
| B/L/S/jal/jalr/lui/auipc | - | - | add |

ALU功能表

|  |  |
| --- | --- |
| ALUFunct | ALU\_Result |
| add | a+b |
| sub | a-b |
| sll | a<<b(4,0) (逻辑左移) |
| slt | a<b (有符号数) |
| xor | a^b |
| or | a|b |
| and | a&b |
| srl | a>>b(4,0) (逻辑左移) |
| sra | a>>b(4,0) (算术右移) |
| sltu | a<b (无符号数) |

if\_jump\_flag跳转判断

|  |  |  |
| --- | --- | --- |
| 条件 | | if\_jump\_flag |
| opcode === jal || jalr | | true |
| opcode为B-type | funct3 === beq &&  reg1\_data === reg2\_data | true |
| funct3 === bne &&  reg1\_data =/= io.reg2\_data | true |
| funct3 === blt &&  reg1\_data < io.reg2\_data | true |
| funct3 === bge &&  reg1\_data >= io.reg2\_data | true |
| funct3 === bltu &&  reg1\_data < io.reg2\_data | true |
| funct3 === bgeu &&  reg1\_data >= io.reg2\_data | true |
| other | | false |

if\_jump\_address跳转地址计算（图中漏掉了）

|  |  |
| --- | --- |
| 条件 | if\_jump\_address |
| opcode === jalr | Imm + Reg1Data |
| other | Imm + InsAddr |

（4） MEM阶段的任务是，若MemWE为真，则此时为S指令，要根据 funct3选择写内存的数据来源是alu\_result还是Reg2Data；若MemRE为真，则此时为L指令，要读内存并输出读到的数据给WB阶段。MEM阶段由于需要访存，用时约200ps。

（5） WB阶段的任务很简单，就是根据ID阶段译出的RegWriteSrc 信号作为控制信号来选择四路Mux的输出，并将该输出作为寄存器组的写数据来写寄存器。WB阶段由于访寄存器组，用时约50ps。

# 实验过程与结果

有了上面的铺垫后（尤其是对[执行流程](#_5._指令的执行流程)的分析），补充代码实现单周期CPU就很容易了。

## 1．InstructionFetch

只需完成pc前的Mux选择，简单接线即可：



## 2. InstructionDecode

任务包括了AluOpSrc、MemRE、MemWE、RegWSrc的译码。列出译码表格：

aluop1\_source

|  |  |  |
| --- | --- | --- |
| Mux控制信号 | aluop1\_source | |
| 控制信号为真 | 控制信号为假 |
| opcode为auipc或B或jal | InstructionAddress | Register |

aluop2\_source

|  |  |  |
| --- | --- | --- |
| Mux控制信号 | aluop2\_source | |
| 控制信号为真 | 控制信号为假 |
| opcode为RM | Register | Immediate |

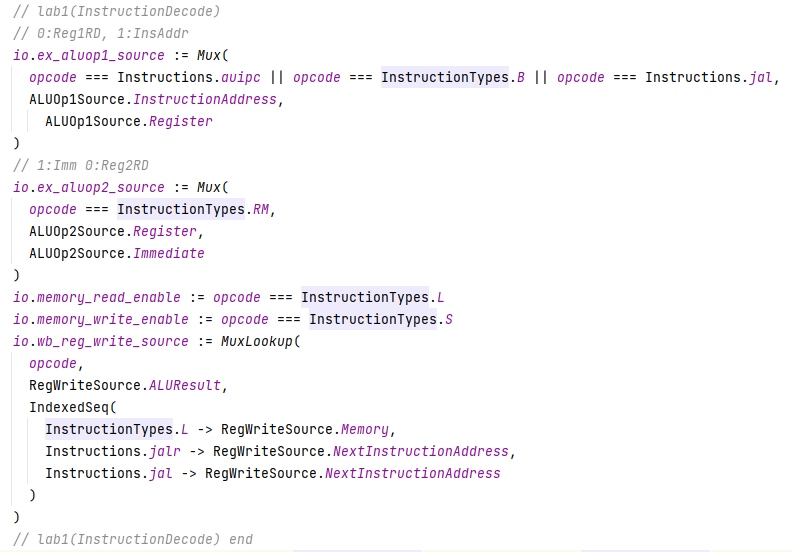
MemRE 即 opcode === L-type （为load指令则开内存读使能）

MemWE 即 opcode === S-type （为store指令则开内存写使能）

RegWSrc

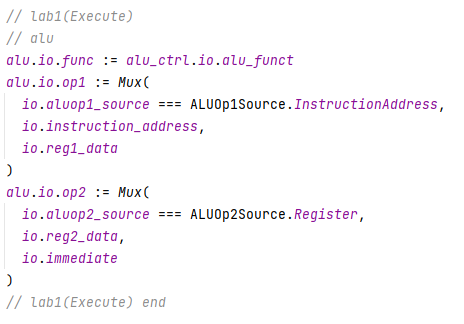
|  |  |
| --- | --- |
| 四路Mux控制信号为opcode | RegWSrc |
| L-type | MemRD |
| jalr | NextInsAddr |
| jal | NextInsAddr |
| other | Alu\_Result |

于是：



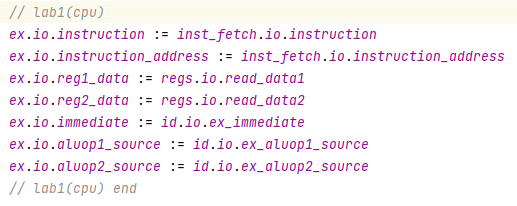
## Execute

我们在EX阶段的代码补充任务只有给ALU单元接线：ALUFunct与两个Mux的选择，经过前面的分析：



## CPU

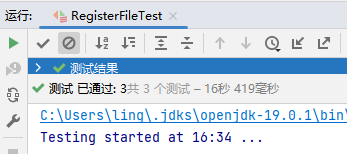
我们还需要将CPU的接线完成。只有EX模块需要补充，照着数据通路与控制线路图连即可：



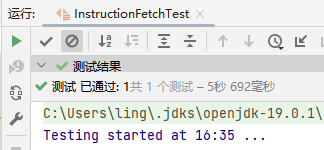
## 标准测试

全部通过：

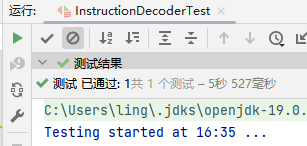
RegisterFileTest



InstructionFetchTest



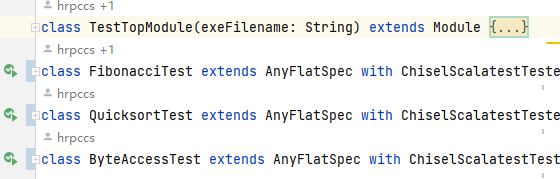
InstructionDecoderTest



ExecuteTest

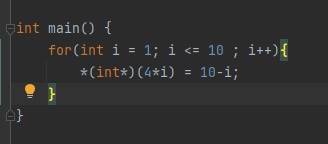


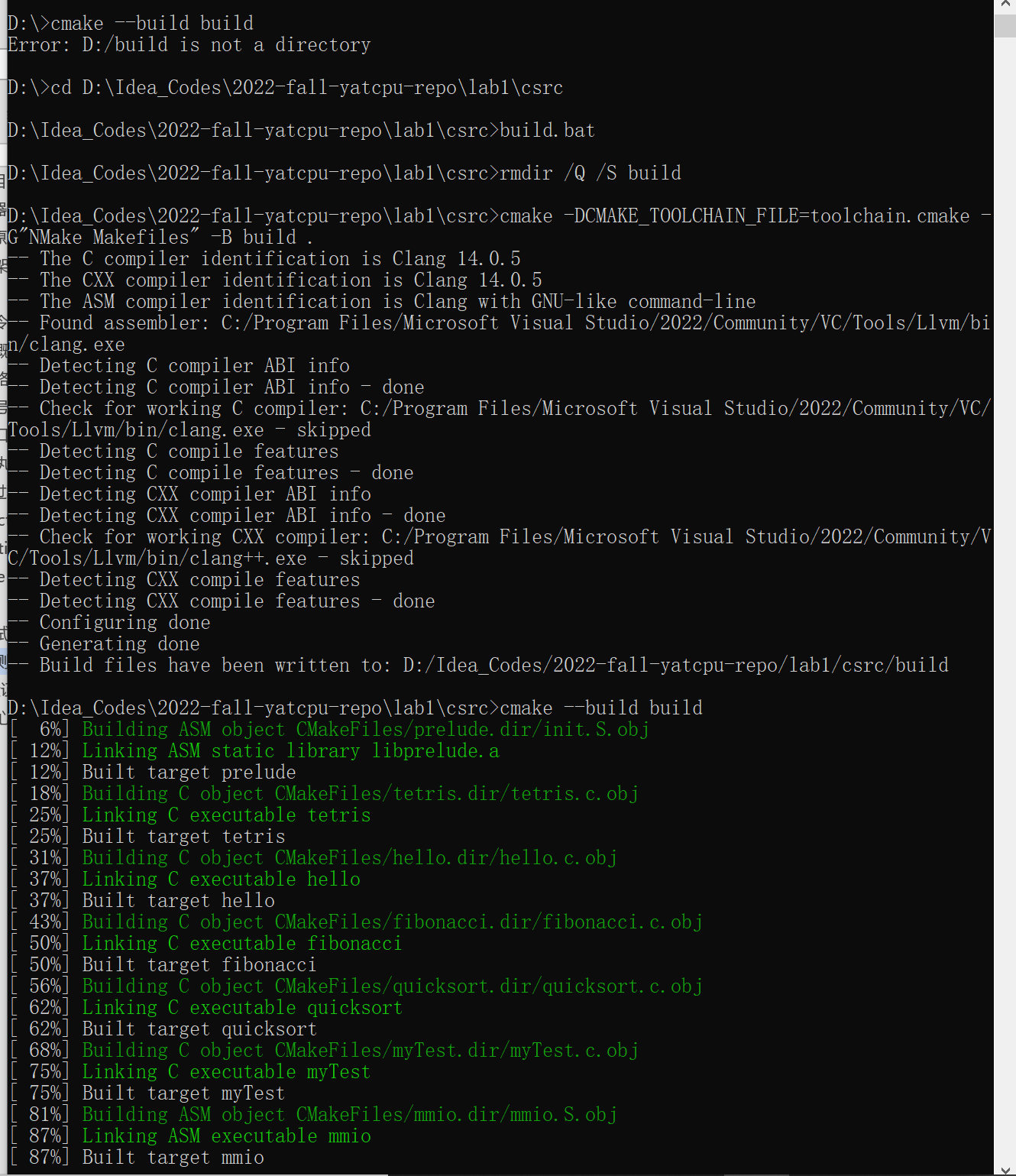
CPUTest



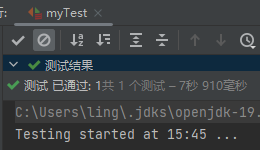
## 自定义测试

我的程序：

  
跑build.bat编译脚本：



测试：



# 实验心得

## 查手册

作为新手，我认为本实验的难点在于ID阶段的指令译码，要对指令译码就需要了解指令，要了解指令就需要去查询各种手册，而初读手册（有些还是全英的）。过了这道坎就获得了自行查询手册的能力，而且这个能力在后续的实验中还频繁用上，可谓非常实用。

## 钻研指令执行流程

在写计组实验报告时我一条一条指令细推指令的执行流程，事无巨细的描述，在其中我居然感受到了学习乐趣，尤其是在后面lab3多种指令组合的执行流程。今晚我终于想明白了这种学习方法为什么有效！这其实和我高中做数学错题本是一个道理：哪怕这题我看答案的，我只要一步一步纠原因，我也能得到大进步！