



**计算机组成原理**

**课 程 设 计**

团队实验报告

|  |  |
| --- | --- |
| 学 院 | 计算机学院 |
| 专 业 | 计算机科学与技术 |
| 指导老师 | 王殿欣 |
| 组 长 | 杨昊 |
| 组 员 | 王欣宇、李昊阳 |
| 组长联系方式 | 13818794782 |

二O二一年八月

目录

[第一章 项目简述 1](#_Toc145175935)

[第二章 组员分工 1](#_Toc145175936)

[第三章 设计目的 1](#_Toc145175937)

[第四章 设计环境 1](#_Toc145175938)

[第五章 设计原理及内容 2](#_Toc145175939)

[5.1 单周期CPU实现 2](#_Toc145175940)

[5.1.1 指令集选取 2](#_Toc145175941)

[5.1.2 设计过程 2](#_Toc145175942)

[5.2 流水线CPU整体架构 9](#_Toc145175943)

[5.1.3 数据通路 9](#_Toc145175944)

[5.1.4 控制逻辑 10](#_Toc145175945)

[5.3 流水线冒险问题以及解决方案 12](#_Toc145175946)

[5.2.1 数据冒险（Data Hazard）： 12](#_Toc145175947)

[5.4 其他特色 12](#_Toc145175948)

[第六章 设计与实现 13](#_Toc145175949)

[第七章 测试 13](#_Toc145175950)

[7.1 仿真测试用例、结果以及分析 13](#_Toc145175951)

[第八章 问题及解决方法 13](#_Toc145175952)

[第九章 心得体会及总结 13](#_Toc145175953)

[第十章 参考文献有价值的资源推荐 13](#_Toc145175954)

[附录A：MIPS指令列表 13](#_Toc145175955)

# 项目简述

本项目实现了一种基于MIPS架构的多周期流水线CPU，实现了完备的移位与逻辑运算、不包括乘除在内的大部分算术运算，以及跳转指令、分支指令、加载与存储指令等45条指令。项目实现了经典的五级流水线CPU，解决了流水线中数据相关、结构相关和控制相关问题。对所有指令在vivado上进行行为仿真，通过测试。最终设计并编写汇编测试算法，结合汇编与接口设计实验，最终上板验证成功。

# 组员分工

小组分工见表2.1 小组分工。

|  |  |
| --- | --- |
| **组员** | **分工任务** |
| 组长：杨昊 | * 负责 |
| 组员1：王欣宇 |  |
| 组员2：李昊阳 |  |

表2.1 小组分工

# 设计目的

主要任务是以CPU作为设计核心，应用已经学过的专业知识，设计并实现一台简单的计算机系统，并测试通过。

# 设计环境

|  |  |
| --- | --- |
| 操作系统 | Windows x64 10 Pro 22H2 |
| 编程语言 | Verilog |
| EDA工具 | Vivado v2019.2 (64-bit) |
| 汇编语言 | MIPS32 |
| 汇编语言与Verilog代码编辑器 | Visual Studio Code 1.81.1 |
| 开发板 | xc7a35tcsg324-1 |

# 设计原理及内容

## 单周期CPU实现

### 指令集选取

本项目决定实现MIPS32的子指令集，包括了三种基本指令格式。R格式指令为纯寄存器指令，所有的操作数（除移位外）均保存在寄存器中。Op字段均为0，使用funct字段区分指令。I格式指令为带立即数的指令，最多使用两个寄存器，同时包括了load/store指令。使用Op字段区分指令。J格式指令为长跳转指令，仅有一个立即数操作数。使用Op字段区分指令。所有指令都是32位长，以简化指令编码。

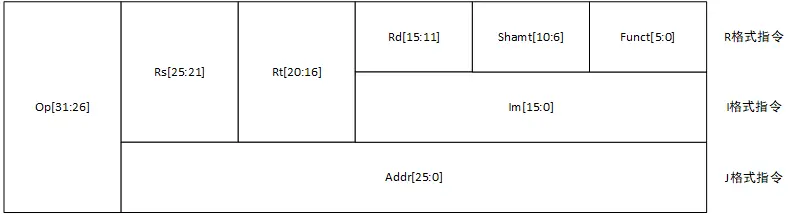


图5.1.1-1 MIPS指令格式

首先是R格式指令集，我们决定实现其中的18条指令，包括算术类、逻辑类、位移类与跳转类指令。指令有：ADDI、ADDIU、SLTIU、SLTI、ANDI、ORI、XORI、LUI、LW、SW、BEQ、BNE。指令格式与功能如下表所示。

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 算术类指令 | 指令 | op | rs | rt | rd | shamt | funct | 功能 |
| add | 0 | rs | rt | rd | 0 | 100000 | rd=rs+rt |
| addu | 0 | rs | rt | rd | 0 | 100001 | rd=rs+rt（无符号数） |
| sub | 0 | rs | rt | rd | 0 | 100010 | rd=rs-rt |
| subu | 0 | rs | rt | rd | 0 | 100011 | rd=rs+rt（无符号数） |
| slt | 0 | rs | rt | rd | 0 | 101010 | rd=(rs<rt)?1:0 |
| sltu | 0 | rs | rt | rd | 0 | 101011 | rd=(rs<rt)?1:0（无符号数） |
| 逻辑类指令 | **指令** | **op** | **rs** | **rt** | **rd** | **shamt** | **funct** | **功能** |
| and | 0 | rs | rt | rd | 0 | 100100 | rd=rs&rt |
| or | 0 | rs | rt | rd | 0 | 100101 | rd=rs|rt |
| xor | 0 | rs | rt | rd | 0 | 100110 | rd=rs xor rd |
| nor | 0 | rs | rt | rd | 0 | 100111 | rd=!(rs|rt) |
| 位移类指令 | **指令** | **op** | **rs** | **rt** | **rd** | **shamt** | **funct** | **功能** |
| sll | 0 | 0 | rt | rd | shamt | 0 | rd=rt<<shamt |
| srl | 0 | 0 | rt | rd | shamt | 10 | rd=rt>>shamt |
| sra | 0 | 0 | rt | rd | shamt | 11 | rd=rt>>shamt（符号位保留） |
| sllv | 0 | rs | rt | rd | 0 | 100 | rd=rt<<rs |
| srlv | 0 | rs | rt | rd | 0 | 110 | rd=rt>>rs |
| srav | 0 | rs | rt | rd | 0 | 111 | rd=rt>>rs（符号位保留） |
| 跳转指令 | **指令** | **op** | **rs** | **rt** | **rd** | **shamt** | **funct** | **功能** |
| jr | 0 | rs | 0 | 0 | 0 | 1000 | PC=rs |
| jalr | 0 | rs | 0 | 0 | 0 | 1001 | $31=PC+4;PC=rs |

图5.1.1-1 MIPS R型指令

以及I格式指令集，我们决定实现其中的12条指令，包括算术类、逻辑类、跳转类以及载入类指令。指令有：ADD、ADDU、SUB、SUBU、SLT、SLTU、AND、OR、XOR、NOR、SLL、SRL、SRA、SLLV、SRLV、SRAV、JR、JALR。

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 算数类指令 | 指令 | op | rs | rd | im | 功能 |
| addi | 1000 | rs | rd | im | rd=rs+im |
| addiu | 1001 | rs | rd | im | rd=rs+im（无符号数） |
| slti | 1010 | rs | rd | im | rd=(rs<im)?1:0 |
| sltiu | 1011 | rs | rd | im | rd=(rs<im)?1:0（无符号数） |
| 逻辑类指令 | **指令** | **op** | **rs** | **rd** | **im** | **功能** |
| andi | 1100 | rs | rd | im | rd=rs&im |
| ori | 1101 | rs | rd | im | rd=rs|im |
| xori | 1110 | rs | rd | im | rd=rs xor im |
| 载入类指令 | **指令** | **op** | **rs** | **rd** | **im** | **功能** |
| lui | 1111 | 0 | rd | im | rt=im\*65536 |
| lw | 100011 | rs | rd | im | rt=memory[rs+im] |
| sw | 101011 | rs | rd | im | memory[rs+im]=rt |
| 跳转类指令 | **指令** | **op** | **rs** | **rd** | **im** | **功能** |
| beq | 100 | rs | rd | im | PC=(rs==rt)?PC+4+im<<2:PC |
| bne | 101 | rs | rd | im | PC=(rs!=rt)?PC+4+im<<2:PC |

图5.1.1-2 MIPS I型指令

最后为J格式指令集，我们决定实现最基础的两条条指令：J、JAL。

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 跳转类指令 | 指令 | op | address | 功能 |
| j | 10 | addr | PC={(PC+4)[31,28],addr,00} |
| jal | 11 | addr | $31=PC+4;PC={(PC+4)[31,28],addr,00} |

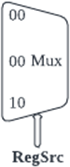
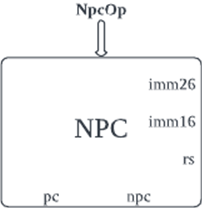
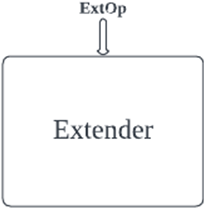
图5.1.1-3 MIPS J型指令

每条指令的详细信息：指令名称、操作数、寄存器传输级定义、指令格式类型、中文描述等详见附录A。

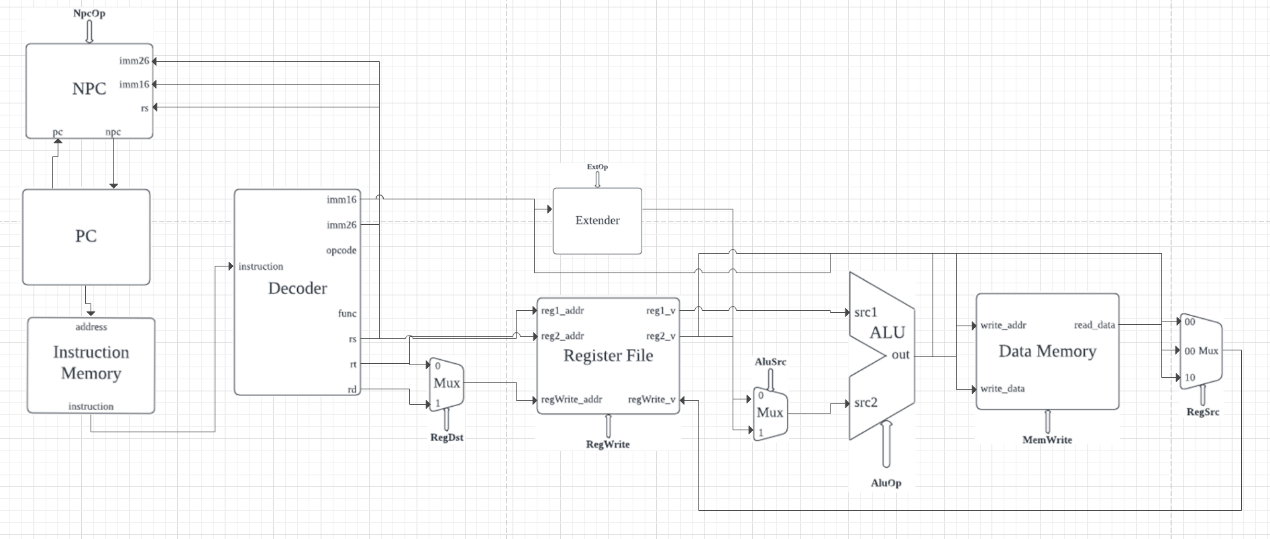
### 设计过程

首先根据小组讨论，我们利用图示软件确定了CPU各个模块的输入输出以及控制信号，然后画出了初步的CPU数据通路图。

模块图



数据通路图



接着，列出需要列出二进制指令集和控制信号表，新建definition文件定义操作码、控制信号和二进制编码的对应关系，指令操作码、控制信号和二进制编码的对应关系如下所示：

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 指令 | reg\_dst | reg\_src | alu\_src | alu\_op | en\_men\_write | reg\_src | extend | npc\_op |
| lui | 01 | 011 | - | - | 0 | imm | shift | next |
| addiu | 01 | 001 | 1 | add | 0 | alu | unsigned | next |
| addi | 01 | 001 | 1 | add | 0 | alu | signed | next |
| add | 11 | 001 | 0 | add | 0 | alu | - | next |
| sub | 11 | 001 | 0 | sub | 0 | alu | - | next |
| lw | 0 | 010 | 1 | add | 0 | mem | unsigned | next |
| sw | - | 010 | 1 | add | 1 | - | unsigned | next |
| beq | - | - | 0 | sub | 0 | - | - | offset |
| j | - | - | - | - | 0 | - | - | immediate |

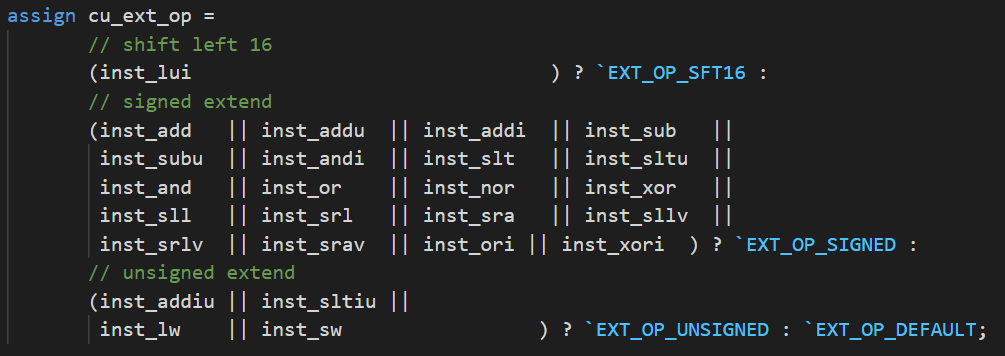
控制信号表

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 信号 | 作用模块 | 含义 | 编码 | 信号名 |
| en\_reg | register\_file | 写寄存器文件 | 1 |  |
|  |  | / | 0 |  |
| extend | extend | 不扩展 | 00 |  |
|  |  | LUI左移16位 | 01 |  |
|  |  | 符号扩展16->32 | 10 |  |
|  |  | 无符号扩展16->32 | 11 |  |
| npc\_op | npc | / | 000 |  |
|  |  | PC+4 | 001 |  |
|  |  | 26位立即数扩展 | 010 |  |
|  |  | PC+offset+4 | 011 |  |
|  |  | rs | 100 |  |
| alu\_src | alu\_src\_mux | register源 | 0 |  |
|  |  | immediate源 | 1 |  |
| alu\_op | alu | / | 0000 |  |
|  |  | ADD | 0001 |  |
|  |  | SUB | 0010 |  |
|  |  | SLT | 0011 |  |
|  |  | AND | 0100 |  |
|  |  | OR | 0101 |  |
|  |  | XOR | 0110 |  |
|  |  | NOR | 0111 |  |
|  |  | SLL | 1000 |  |
|  |  | SRL | 1001 |  |
|  |  | SRA | 1010 |  |
|  |  | SLLV | 1011 |  |
|  |  | SRLV | 1100 |  |
|  |  | SRAV | 1101 |  |
| reg\_src | reg\_src\_mux | / | 000 |  |
|  |  | 来自ALU | 001 |  |
|  |  | 来自data\_mam | 010 |  |
|  |  | 来自extend | 011 |  |
|  |  | 来自ALU | 100 |  |
| reg\_dst | reg\_dst\_mux | / | 00 |  |
|  |  | 写到rt | 01 |  |
|  |  | 写到rs | 10 |  |
|  |  | 写到寄存器文件中的某个 | 11 |  |

控制信号编码表

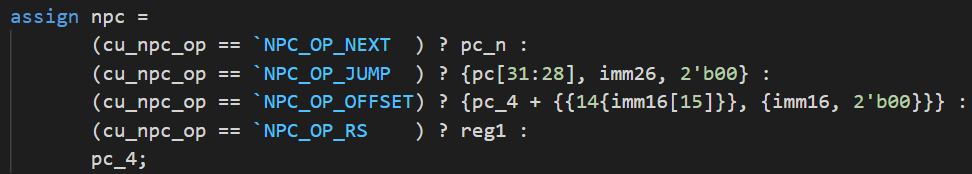
**控制单元（CU）**

控制模块CU负责根据译码结果产生控制信号，按照之前列出的控制信号表，确定控制信号，根据之前在definetion文件中定义的宏为不同的模块生成控制信号。以扩展模块extend的控制信号extop为例，根据表格lui指令应该使用特殊的左移16扩展信号，对于运算指令中含有字符‘u’的以及lw、sw指令应该进行无符号扩展，其余算数指令应该进行符号扩展，由此便可以写出extop控制信号的扩展表：



**NPC模块**

|  |  |  |
| --- | --- | --- |
| input | pc | 当前程序计数器值 |
| input | imm16 | 16位立即数译码 |
| input | imm26 | 26位立即数译码 |
| input | rs | rs寄存器值 |
| input | control\_npc | npc模块控制信号 |
| output | npc | 下一条指令地址 |

NPC模块负责根据控制信号control\_npc判定下一条指令地址npc。下一条指令的地址可能有以下几种情况：①当前指令地址+4（pc+4） ②无条件跳转地址（imm26）③条件跳转（imm16+4或rs寄存器），因此输入要同时传入imm16，imm26，rs的值。在npc模块中还需要对不同长度的立即数做扩展，直接跳转的扩展方式为pc[31:28], imm26, 2'b00，相对寻址的扩张方式为符号扩展pc\_n + {{14{imm16[15]}}, {imm16, 2'b00}},关键代码如下所示：

**PC模块**

|  |  |  |
| --- | --- | --- |
| input | wire clk | 时钟信号 |
| input | wire rst | 复位信号 |
| input [31:0] | wire npc | 下一条指令地址 |
| output | reg pc | 程序计数器值 |

PC模块接收NPC模块传入的下一条指令地址npc，储存并输出程序计数器的值。clk与rst分别代表时钟信号和复位信号，npc是由NPC模块传入的下一条指令地址。PC模块在时钟上升沿将npc的值更新到pc寄存器并输出，在rst信号上升沿刷新PC的值，代码中采用了异步刷新的方式。

**指令寄存器 （Instruction Memory）**

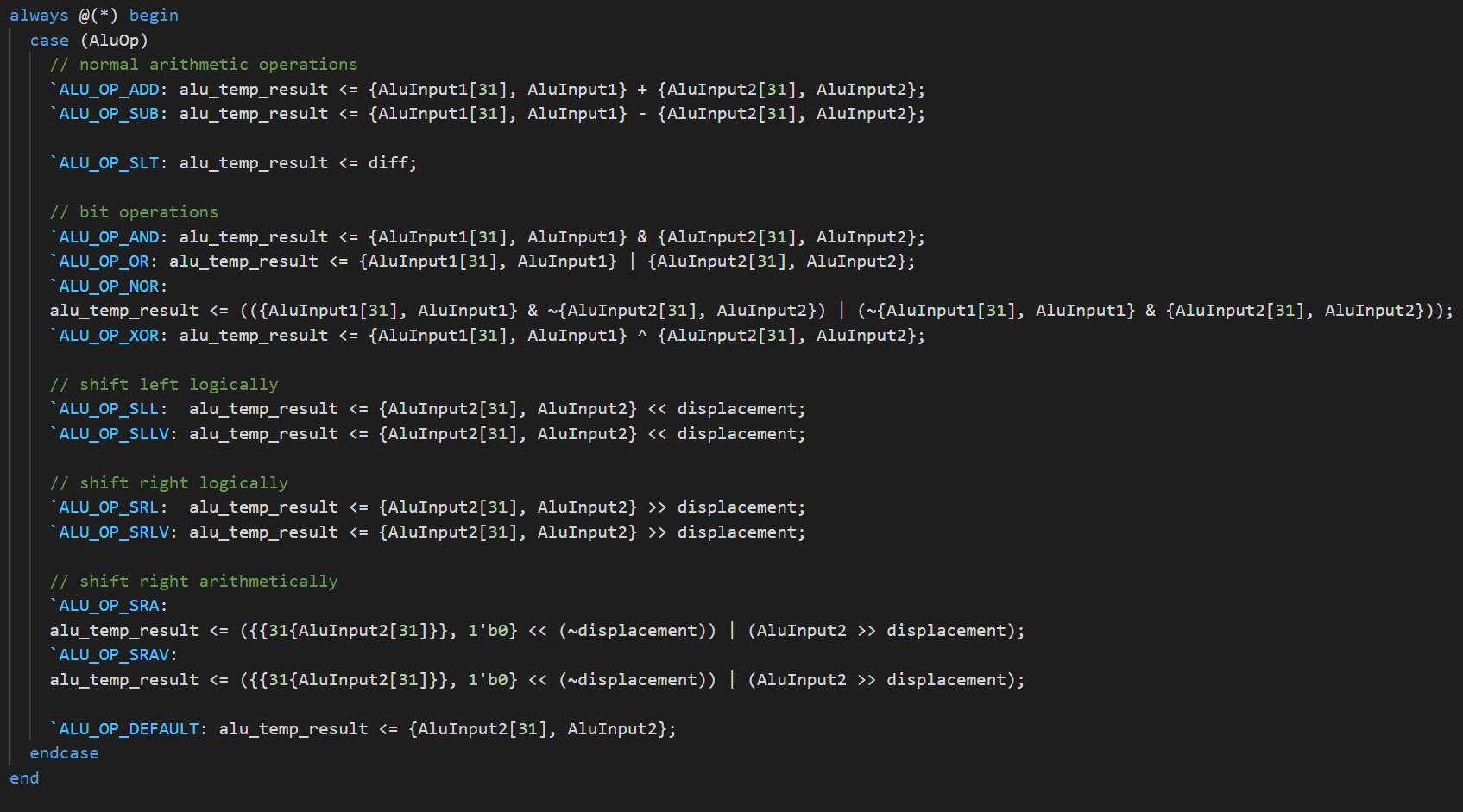
|  |  |  |
| --- | --- | --- |
| input | instruction\_addr | 指令地址 |
| output | [31:0] instruction | 指令内容 |

ins\_mem模块是存储指令的模块，根据传入指令地址输出32位指令信号，传入后续的译码阶段。此外，为了储存指令，建立instruction.txt以16进制的方式储存，之后在testbench中便可以通过$readmem函数进行读取。

**算数运算单元（ALU）**

|  |  |  |
| --- | --- | --- |
| input | input1 | Alu输入1 |
| input | input2 | Alu输入2 |
| input | AluOp | Alu控制信号 |
| output | AluOut | Alu运算结果 |
| output | overflow | 是否发生溢出 |

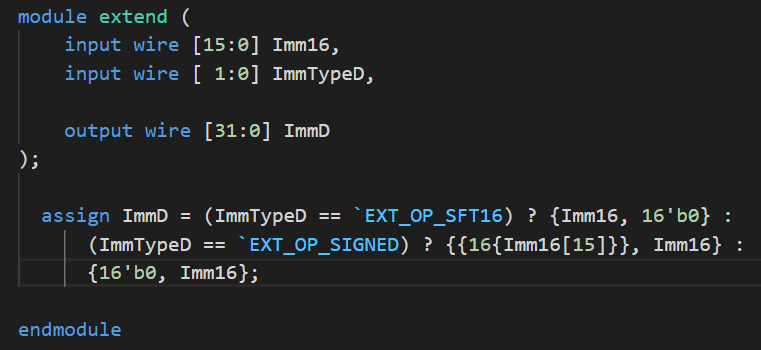
Alu依据控制信号AluOp对两个输入input1和input2做算数运算操作，根据之前准备实现的算数运算指令类型以及算数运算的特性，分情况写出对应的输出，具体代码如下所示



**扩展（Extend）**

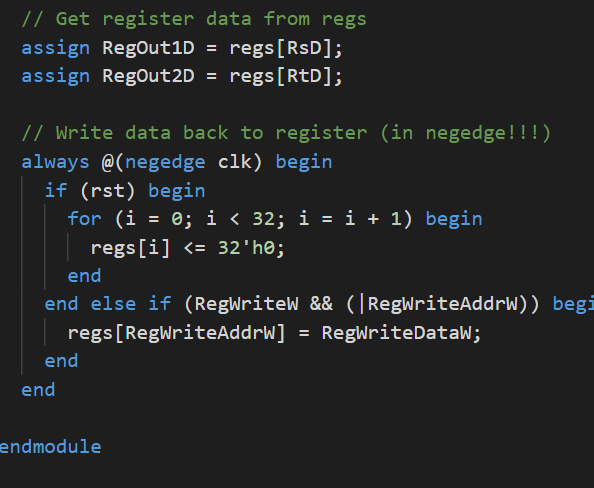
|  |  |  |
| --- | --- | --- |
| input | imm16 | 16位立即数输入 |
| input | ExtOp | 扩展控制信号 |
| output | out | 扩展结果 |

译码得到的16位立即数需要进行扩展，依据控制信号ExtOp将其扩展为32位，分情况讨论16位立即数的扩展操作。左移16的扩展情况下，将16位立即数放到高位，低16位置0；无符号和符号扩展分别将高16位置为0或符号位，具体代码如下。



**寄存器文件（Register File）**

寄存器文件的默认行为是在时钟下降沿（避免冲突）将根据输入的两个寄存器地址输出对应寄存器的值。此外寄存器写文件的控制信号为RegWrite，用来判断是否允许写寄存器，其从译码结果获取寄存器地址，从写回阶段获取写入值，并依据RegWrite判断是否改写寄存器文件，关键代码如下。

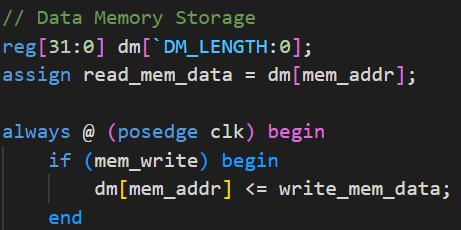
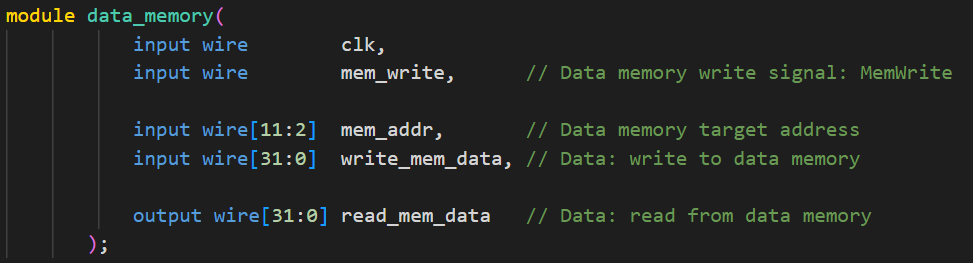
****

**数据存储器（Data Memory）**

|  |  |  |
| --- | --- | --- |
| input | mem\_write | 数据存储器的写使能信号 |
| input | mem\_addr | ALU输出的数据地址 |
| input | write\_mem\_data | 地址内应存的数据 |
| output | read\_mem\_data | 从地址读出的数据 |

数据存储器是存储数据的模块。存储器访问阶段，如果指令涉及读取、存储内存，则需要对存储器中相应地址进行读取或者写入。结果写回阶段，则将得到的数据（访问存储器或者修改其它寄存器的值获得）写回相应的寄存器。

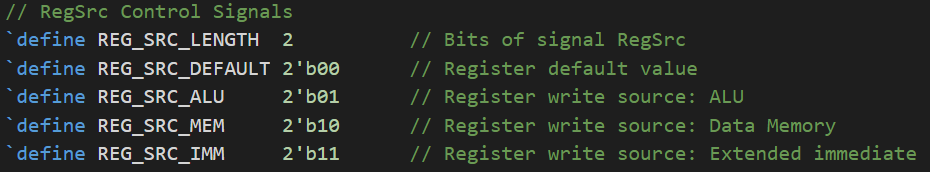
数据存储器关键代码实现如下：



对于RegSrc信号，需要选择将 ALU 计算结果、数据存储器输出或 Extend 模块输出写入寄存器，其设计如下：

| **RegSrc[1:0]** | **功能** | **指令** |
| --- | --- | --- |
| 00 | Default | 缺省值 |
| 01 | 来自 ALU | ADDIU、ADD、SUB |
| 10 | 来自 Data Memory | LW |
| 11 | 来自 Extend 模块输出 | LUI |

RegSrc控制信号的定义在instruction\_head.v中声明，具体如下图所示：



**顶层模块（top）**

在完成全体模块的编写之后，需要建立一个顶层模块将全部模块进行连接，实现模块的整体调用。单周期 CPU 的设计包括指令存储器：Instruction Memory、Data Memory、Register File、PC、NPC、CU、ALU、Extender、MUX这几个模块，顶层模块实例化各子模块使其接口互相连接。

## 流水线CPU整体架构

本流水线CPU覆盖31条指令(见表5.1.1-1/2)，下面将从数据通路和控制逻辑阐述设计思路。

### 数据通路

本项目的MIPS32流水线CPU采用经典的五级顺序流水线结构，包括取指（IF）、译码（ID）、执行（EX）、访存（MEM）、写回（WB）。支持32个32位整数型寄存器，并且遵循小端处理模式。

五个流水线阶段分别为：

取指阶段：从指令存储器中读取指令，并同时确定下一条指令的地址，为保持流水线的连续性做好准备。

译码阶段：对取出的指令进行译码，同时从通用寄存器中读取所需寄存器的值。如果指令包含立即数，会进行适当的符号或无符号扩展。对于转移指令，如果满足转移条件，确定转移目标，作为新的指令地址。

执行阶段：根据译码阶段提供的操作数和运算类型，执行计算并生成结果。针对 Load/Store 指令，还会计算出存储器访问的目标地址。

访存阶段：用于存储器访问，特别是 Load/Store 指令。在这一阶段，数据存储器会被访问，或者执行阶段的结果被传递到下一阶段。同时，在此阶段进行异常处理判断，如果出现异常，将清空流水线，然后跳转到异常处理入口地址继续执行。

写回阶段：将执行阶段计算得到的结果写回目标寄存器。这是流水线的最后一个阶段，将计算结果返回到通用寄存器中。

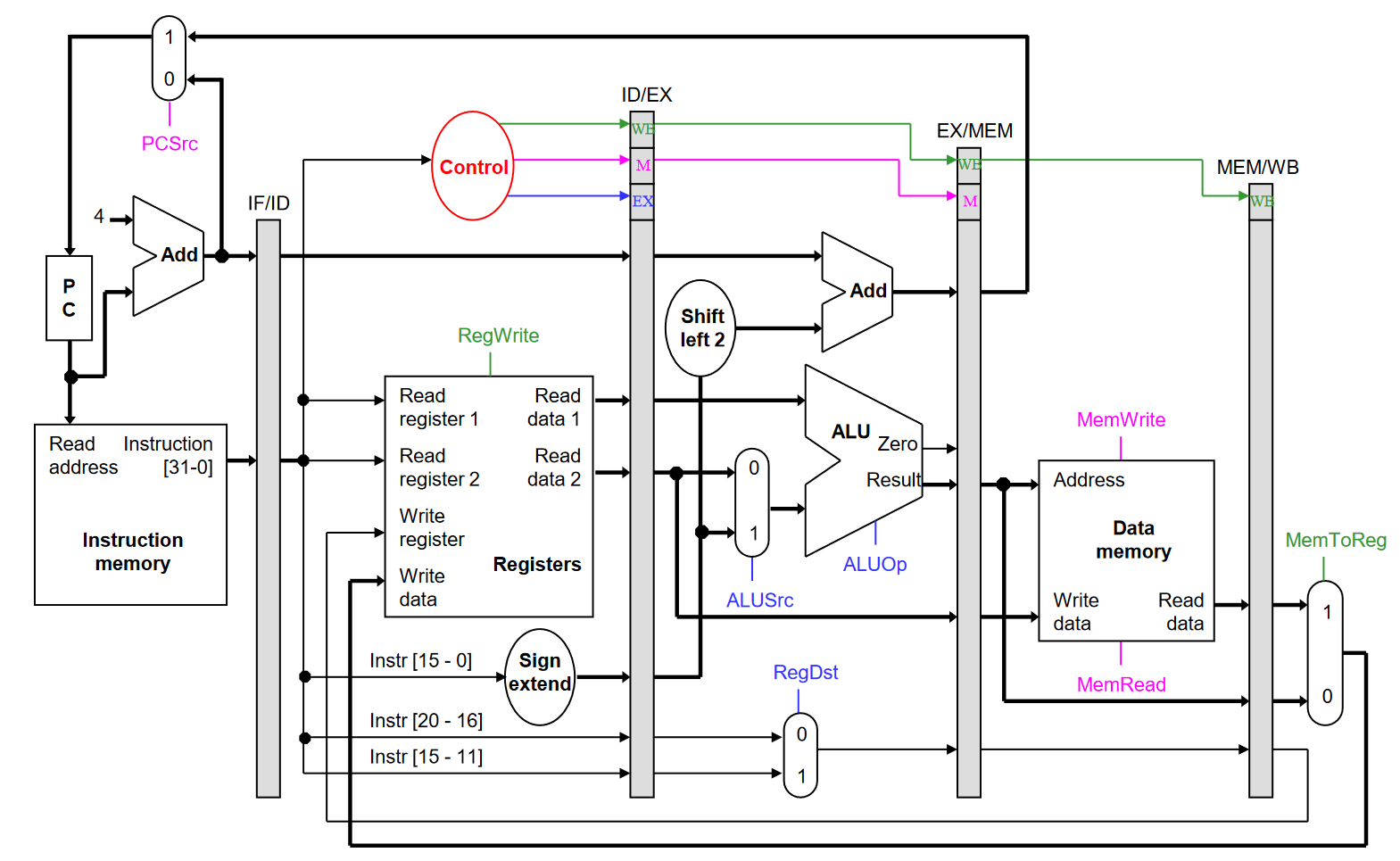
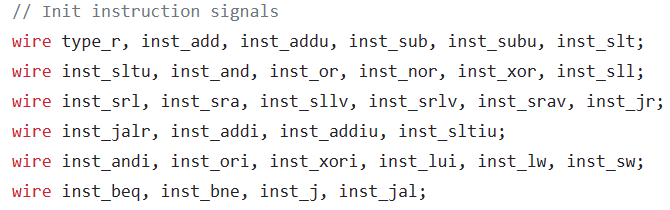


图5.1.2-1 5级流水线CPU数据通路

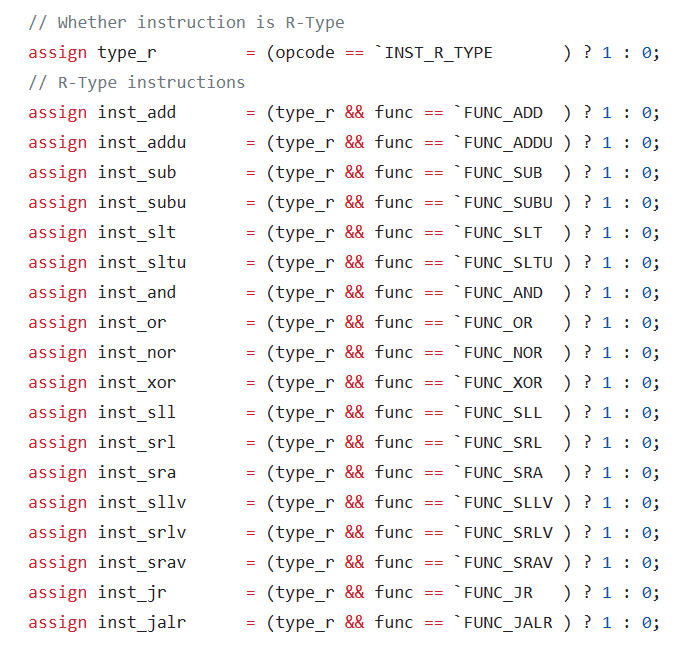
由于采用了流水线结构的设计，系统效率比多周期结构更高。但是，流水线仍具有一些劣势以及限制，因此处理器的主频不宜过高，以确保整个系统的正常运行。最终，我们决定使用 5MHz 的处理器主频。

### 控制逻辑

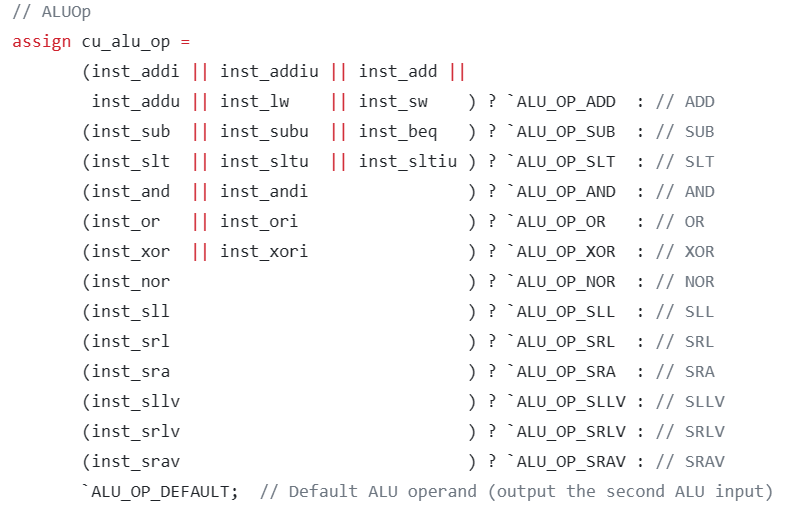
本项目以组合逻辑的方式实现控制逻辑，所有模块由cpu顶层模块组织并连接。而不同模块间的信号控制主要在control\_unit中生成，并在不同模块间传递。在control\_unit中，首先初始化多个控制信号：



根据MIPS指令格式，在指令译码后，根据不同指令字段的值，确定指令格式与类型：



再根据不同指令决定相应控制信号，有lw、ALUOp、RegDst、ALUSrc、RegWrite、MemWrite、RegSrc、ExtOp、NPCOp等控制信号。如ALUOp：



### 流水线寄存器

为了实现让多条指令同时执行，要将五个模块的执行过程进行组合，从时间的维度进行重叠，从而加速 CPU 指令执行的吞吐量，提高 CPU 执行指令的效率，因此增加流水线寄存器。

流水线寄存器用来控制整个数据通路中不同流水阶段里信号、数据的传递。五级流水分为 IF、ID、EX、MEM 和 WB 五个阶段，我们在段与段之间分别设置Reg IF/ID、Reg ID/EX、Reg EX/MEM、Reg MEM/WB 这四个流水线寄存器，用来连接流水的五个阶段。

流水线寄存器作用：① 将各段的工作隔开，使得它们不会互相干扰 ② 保存相应的处理结果 ③ 向后传递后面将要用到的数据或者控制信息，并共享给其他指令。

## 流水线冒险问题以及解决方案

### 5.2.1 数据冒险（Data Hazard）：

数据冒险是由于指令的数据依赖关系导致的流水线停顿。分为三种情况：

①读后写（Read After Write，RAW）冒险：当一条指令需要读取另一条指令刚刚写入的数据时，由于流水线中前一条指令还未完成写入，会导致后一条指令无法正确读取数据，从而需要停顿等待。

②写后写（Write After Write，WAW）冒险：当多条指令需要写入同一个寄存器时，由于后面的指令会覆盖前面指令的结果，会导致后面的指令结果错误。通常不会发生数据冒险，因为流水线中的写操作是顺序执行的。

③写后读（Write After Read，WAR）冒险：当一条指令需要写入一个寄存器，而另一条指令需要从同一个寄存器读取数据时，读操作在写操作之后执行，可能导致读取到错误的数据。

### 5.2.2 控制冒险（Control Hazard）

控制冒险是由于分支指令的跳转或者转移指令的结果尚未确定导致的流水线停顿。当流水线中的分支指令或转移指令执行时，下一条指令的地址可能会发生变化，因此流水线中的后续指令可能需要等待分支指令的结果确定后才能正确执行。为了减少这种停顿，可以采取预测策略，例如分支预测，根据历史信息预测分支的方向，从而减少流水线的停顿。

### 5.2.3 结构冒险（Structural Hazard）

结构冒险是由于硬件资源冲突导致的流水线停顿。这种冒险发生在多个指令同时需要访问同一个硬件单元的情况下，比如多个指令都需要访问内存或者某个寄存器。由于硬件资源有限，某个指令必须等待另一个指令完成后才能继续执行，从而导致流水线停顿。解决方法包括增加硬件资源，使其能够同时支持多个指令的访问，或者通过优化指令顺序减少资源冲突。

## 其他特色

# 设计与实现

# 测试

## 仿真测试用例、结果以及分析

功能性测试

性能测试

下板测试

# 问题及解决方法

# 心得体会及总结

# 参考文献有价值的资源推荐

# 附录A：MIPS指令列表

本附录列出了所选取的指令集的更详细信息，每个条目都包括指令名称、操作数、寄存器传输级定义、指令格式类型、中文描述，以及一张带有操作码的指令布局图。此外还给出了总的操作码映射，包括指令布局，操作码，格式类型和名称。

ADD:

