同济大学

TONGJI UNIVERSITY

《计算机系统实验》

实验报告

实验名称	CPU 改造实验
实验成员	李闯（2253214）
日　　期	二零二五年 三月 二十八日

1、实验目的

- 回顾 MIPS 五阶段流水线 CPU 的基本结构
- 学习教学用 CPU 各模块的编写以及模块之间的连接结构
- 加深对 Verilog 语言的理解以及对硬件开发的理解
- 为之后移植操作系统做准备工作

2、实验内容

此次实验中实现的 MIPS CPU 为五级流水线 CPU，如图 1 所示，分别有取指、译码、执行、访存、 回写五个阶段：

- 取指阶段：从指令存储器读出指令，同时确定下一条指令地址。
- 译码阶段：对指令进行译码，从通用寄存器中读出要使用的寄存器的值，如果指令中含有立即数，那么还要将立即数进行符号扩展或无符号扩展。如果是转移指令，并且满足转移条件，那么给出转移目标，作为新的指令地址。
- 执行阶段：按照译码阶段给出的操作数、运算类型，进行运算，给出运算结果。如果是 Load/Store 指令，那么还会计算 Load/Store 的目标地址。
- 访存阶段：如果是 Load/Store 指令，那么在此阶段会访问数据存储器，反之，只是将执行阶段的结果向下传递到回写阶段。同时,在此阶段还要判断是否有异常需要处理，如果有，那么会清除流水线，然后转移到异常处理例程入口地址处继续执行。
- 回写阶段：将运算结果保存到目标寄存器。

图 1 五级流水线示意图

该处理器按照字节进行寻址，并且采用大端模式，在这种模式下，数据的高位存储在存储器的低地址中，而数据的地位保存在存储器的高地址中。此外，处理器采用哈佛结构，即指令存储器和数据存储器是分开的。

图 2 例：0x00001122 在存储器中的存储形式(大端模式)

改造后 CPU 支持的 89 条指令分别为：

- 逻辑操作指令：and、andi、or、ori、xor、xori、nor、lui
- 移位操作指令：sll、sllv、sra、srav、srl、srlv
- 移动操作指令：movn、movz、mfhi、mthi、mflo、mtlo
- 算术操作指令：add、addi、addiu、addu、sub、subu、clo、clz、slt、slti、sltiu、sltu、mulmult、multu、madd、maddu、msub、msubu、div、divu
- 转移指令：jr、jalr、j、jal、b、bal、beg、bgez、bgezal、bgtz、blez、bltz、bltzal、bne
- 加载存储指令：lb、lbu、lh、lhu、1l、lw、lwl、lwr、sb、sc、sh、sw、swl、swr
- 协处理器访问指令：mtc0、mfc0
- 异常相关指令：teq、tge、tgeu、tlt、tltu、tne、teqi、tgei、tgeiu、tlti、tltiu、tnei、syscall、eret
- 其他指令：nop、ssnop、sync、pref

装 订 线

3、实验步骤

3.1. 软件环境

开发环境：Vivado v2024

仿真环境：ModelSim PE 10.4c

测试环境：MARS 4.5

文档管理：office、notepad++等

3.2. 硬件环境

计算机：Windows 11 家庭中文版

开发板：NEXYS 4 DDR Atrix-7

4、实验结果

4.1 指令设计和数据通路设计

1. and、or、xor、nor 指令

31 26	25 21	20 16	15 11	10 6	5 0	
SPECIAL 000000	rs	rt	rd	00000	AND 100100	AND 指令
SPECIAL 000000	rs	rt	rd	00000	OR 100101	OR 指令
SPECIAL 000000	rs	rt	rd	00000	XOR 1001110	XOR 指令
SPECIAL 000000	rs	rt	rd	00000	NOR 100111	NOR 指令

- **and 指令**

指令格式：and rd, rs, rt.

指令功能：rd <- rs AND rt，将地址为 rs 的通用寄存器的值与地址为 rt 的通用寄存器的值进行逻辑“与”运算，运算结果保存到地址为 rd 的通用寄存器中。

- **or 指令**

指令格式：or rd,rs,rt。

指令功能：rd <- rs OR rt，将地址为 rs 的通用寄存器的值与地址为 rt 的通用寄存器的值进行逻辑“或”运算，运算结果保存到地址为 rd 的通用寄存器中。

- **xor 指令**

指令格式：xor rd,rs,rt。

指令功能：rd <- rs XOR rt，将地址为 rs 的通用寄存器的值与地址为 rt 的通用寄存器的值进行逻辑“异或”运算，运算结果保存到地址为 rd 的通用寄存器中。

- **nor 指令**

指令格式：nor rd,rs,rt。

指令功能：rd <- rs NOR rt，将地址为 rs 的通用寄存器的值，与地址为 rt 的通用寄存器的值进行逻辑“或非”运算，运算结果保存到地址为 rd 的通用寄存器中。

2. andi、xori、lui 指令

31 26	25 21	20 16	15 0	
ANDI 001100	rs	rt	immediate	ANDI 指令
XORI 001110	rs	rt	immediate	XORI 指令
LUI 001111	00000	rt	immediate	LUI 指令

- **andi 指令**

指令格式：andi rt,rs,immediate。

指令功能：rt <- rs AND zero extended(immediate)，将地址为 rs 的通用寄存器的值与指令中立即数进行零扩展后的值进行逻辑“与”运算，运算结果保存到地址为 rt 的通用寄存器中。

- **xori 指令**

指令格式：xori rt,rs,immediate。

指令功能：rt <- rs XOR zero extended(immediate)，将地址为 rs 的通用寄存器的值与指令中立即数进行零扩展后的值进行逻辑“异或”运算，运算结果保存到地址为 rt 的通用寄存器中。

- **lui 指令**

指令格式：lui rt,immediate。

指令功能：rt <- immediatel || 0^{16}，将指令中的 16bit 立即数保存到地址为 rt 的通用寄存器的高 16 位。另外，地址为 rt 的通用寄存器的低 16 位使用 0 填充。

3. sll、sllv、sra、srav、srl、srlv 指令

31 26	25 21	20 16	15 11	10 6	5 0

SPECIAL 000000	00000	rt	rd	sa	SLL 000000	SLL 指令
SPECIAL 000000	00000	rt	rd	sa	SRL 000010	SRL 指令
SPECIAL 000000	00000	rt	rd	sa	SRA 000011	SRA 指令
SPECIAL 000000	rs	rt	rd	00000	SLLV 000100	SLLV 指令
SPECIAL 000000	rs	rt	rd	00000	SRLV 000110	SRLV 指令
SPECIAL 000000	rs	rt	rd	00000	SRAV 000111	SRAV 指令

- **sll 指令**

指令格式：sll rd,rt,sa。

指令功能：rd <- rt << sa(logic)，将地址为 rt 的通用寄存器的值向左移 sa 位，空出来的位置使用 0 填充，结果保存到地址为 rd 的通用寄存器中。

- **srl 指令**

指令格式：srl rd,rt,sa。

指令功能：rd <- rt >> sa(logic)，将地址为 rt 的通用寄存器的值向右移 sa 位，空出来的位置使用 0 填充，结果保存到地址为 rd 的通用寄存器中。

- **sra 指令**

指令格式：sra rd,rt,sa。

指令功能：rd <- rt >> sa(arithmetic)，将地址为 rt 的通用寄存器的值向右移 sa 位，空出来的位置使用 rt[31]的值填充，结果保存到地址为 rd 的通用寄存器中。

- **sllv 指令**

指令格式：sllv rd,rt,rs。

指令功能：rd <- rt << rs[4:0](logic)，将地址为 rt 的通用寄存器的值向左移位，空出来的位置使用 0 填充，结果保存到地址为 rd 的通用寄存器中。移位位数由地址为 rs 的寄存器值的第 0~4bit 确定。

- srlv 指令

指令格式：srlv rd,rt,rs。

指令功能：rd <- rt >> rs[4:0](logic)，将地址为 rt 的通用寄存器的值向右移位，空出来的位置使用 0 填充，结果保存到地址为 rd 的通用寄存器中。移位位数由地址为 rs 的寄存器值的第 0~4bit 确定。

- srav 指令

指令格式：srav rd,rt,rs。

指令功能：rd <- rt >> rs[4:0](arithmetic)，将地址为 rt 的通用寄存器的值向右移位，空出来的位置使用 rt[31]填充，结果保存到地址为 rd 的通用寄存器中。移位位数由地址为 rs 的寄存器值的第 0~4bit 确定。

4. nop、ssnop、sync、pref 指令

31 26	25 21	20 16	15 11	10 6	5 0	
SPECIAL 000000	00000	00000	00000	00000	000000	NOP 指令
SPECIAL 000000	00000	00000	00001	00000	00000	SSNOP 指令
SPECIAL 000000	00000	00000	00000	00000	SYNC 001111	SYNC 指令
PREF 110011	base	hint	offset			PREF 指令

- nop 指令

指令格式：nop。

指令功能：不执行任何操作，通常用于占位或等待。该指令在执行时不会改变任何寄存器或内存的内容。

- **ssnop 指令**

指令格式：ssnop。

指令功能：与 NOP 指令类似，不执行任何操作，但这是一种特殊的 NOP 指令，可能在特定的处理器架构或系统中用于占位或同步。

- **sync 指令**

指令格式：sync。

指令功能：用于确保之前的指令完成并同步执行，避免指令乱序执行。它确保所有缓存和内存操作都完成，适用于多核处理器环境中的同步操作。

- **pref 指令**

指令格式：pref base, hint, offset。

指令功能：用于数据预取，将指定的内存数据预加载到缓存中，以便将来的访问可以更快。base 指定了数据的起始地址，hint 提供了预取的提示信息，offset 指定了数据的偏移量。

5. movn、movz、mfhi、mffo、mthi、mtlo 指令

31 26	25 21	20 16	15 11	10 6	5 0	
SPECIAL 000000	rs	rt	rd	00000	MOVN 001011	MOVN 指令
SPECIAL 000000	rs	rt	rd	00000	MOVZ 001010	MOVZ 指令
SPECIAL 000000	00000	00000	rd	00000	MFHI 010000	MFHI 指令
SPECIAL 000000	00000	00000	rd	00000	MFLO 010010	MFLO 指令
SPECIAL 000000	rs	00000	00000	00000	MTHI 010001	MTHI 指令

SPECIAL 000000	rs	00000	00000	00000	MTLO 010011	MTLO 指令

- **movn 指令**

指令格式：movn rd,rs,rt。

指令功能：if rt ≠ 0 then rd <- rs，判断地址为 rt 的通用寄存器的值。如果不为零，那么将地址为 rs 的通用寄存器的值赋给地址为 rd 的通用寄存器；反之，保持地址为 rd 的通用寄存器不变。movn 是 Move Conditional on Not Zero 的意思。

- **movz 指令**

指令格式：movz rd,rs,rt。

指令功能：if rt == 0 then rd <- rs，与上面 movn 指令的作用正好相反，判断地址为 rt 的通用寄存器的值。如果为零，那么将地址为 rs 的通用寄存器的值赋给地址为 rd 的通用寄存器；反之，保持地址为 rd 的通用寄存器不变。movz 是 Move Conditiona on Zero 的意思。

- **mfhi 指令**

指令格式：mfhi rd。

指令功能：rd <- hi，将特殊寄存器 HI 的值赋给地址为 rd 的通用寄存器。

- **mflo 指令**

指令格式：mflo rd。

指令功能：rd <- lo，将特殊寄存器 LO 的值赋给地址为 rd 的通用寄存器

- **mthi 指令**

指令格式：mthi rs。

指令功能：hi <- rs，将地址为 rs 的通用寄存器的值赋给特殊寄存器 HI。

- **mtlo 指令**

指令格式：mtlo rs。

指令功能：lo <- rs，将地址为 rs 的通用寄存器的值赋给特殊寄存器 LO。

6. add、addu、sub、subu、slt、sltu 指令

31 26 25 21 20 16 15 11 10 6 5 0

SPECIAL 000000	rs	rt	rd	00000	ADD 100000	ADD 指令
SPECIAL 000000	rs	rt	rd	00000	ADDU 100001	ADDU 指令
SPECIAL 000000	rs	rt	rd	00000	SUB 100010	SUB 指令
SPECIAL 000000	rs	rt	rd	00000	SUBU 100011	SUBU 指令
SPECIAL 000000	rs	rt	rd	00000	SLT1 01010	SLT1 指令
SPECIAL 000000	rs	rt	rd	00000	SLTU 101011	SLTU 指令

- **add 指令**

指令格式：add rd, rs, rt。

指令功能：rd <- rs + rt，将地址为 rs 的通用寄存器的值与地址为 rt 的通用寄存器的值进行加法运算，结果保存到地址为 rd 的通用寄存器中。如果加法运算溢出，会产生溢出异常，同时不保存结果。

- **addu 指令**

指令格式：addu rd, rs, rt。

指令功能：rd <- rs + rt，将地址为 rs 的通用寄存器的值与地址为 rt 的通用寄存器的值进行加法运算，结果保存到地址为 rd 的通用寄存器中。与 add 指令不同，addu 指令不进行溢出检查，始终将结果保存到目的寄存器。

- **sub 指令**

指令格式：sub rd, rs, rt。

指令功能：rd <- rs - rt，将地址为 rs 的通用寄存器的值与地址为 rt 的通用寄存器的值进行减法运算，结果保存到地址为 rd 的通用寄存器中。如果减法运算溢出，会产生溢出异常，同时不保存结果。

- **subu 指令**

指令格式：subu rd, rs, rt。

指令功能：rd <- rs - rt，将地址为 rs 的通用寄存器的值与地址为 rt 的通用寄存器的值进行减法运算，结果保存到地址为 rd 的通用寄存器中。与 sub 指令不同，subu 指令不进行溢出检查，始终将结果保存到目的寄存器。

- **slt 指令**

指令格式：slt rd, rs, rt。

指令功能：rd <- (rs < rt)，将地址为 rs 的通用寄存器的值与地址为 rt 的通用寄存器的值按照有符号数进行比较。如果 rs 小于 rt，则将 1 保存到地址为 rd 的通用寄存器中；否则，将 0 保存到地址为 rd 的通用寄存器中。

- **sltu 指令**

指令格式：sltu rd, rs, rt。

指令功能：rd <- (rs < rt)，将地址为 rs 的通用寄存器的值与地址为 rt 的通用寄存器的值按照无符号数进行比较。如果 rs 小于 rt，则将 1 保存到地址为 rd 的通用寄存器中；否则，将 0 保存到地址为 rd 的通用寄存器中。

7. addi、addiu、slti、sltiu 指令

31 26	25 21	20 16	15 0	
ADDI 001000	**rs**	**rt**	**immediate**	**ADDI 指令**
ADDIU 001001	**rs**	**rt**	**immediate**	**ADDIU 指令**
SLTI 001010	**rs**	**rt**	**immediate**	**SLTI 指令**
SLTIU 001011	**rs**	**rt**	**immediate**	**SLTIU 指令**

- **addi 指令**

指令格式：addi rt, rs, immediate。

指令功能：rt <- rs + (sign-extended immediate)，将指令中的 16 位立即数进行符号扩展，并与地址为 rs 的通用寄存器的值进行加法运算，结果保存到地址为 rt 的通用寄存器中。如果加法运算溢出，会产生溢出异常，同时不保存结果。

- **addiu 指令**

指令格式：addiu rt, rs, immediate。

指令功能：rt <- rs + (sign-extended immediate)，将指令中的 16 位立即数进行符号扩展，并与地址为 rs 的通用寄存器的值进行加法运算，结果保存到地址为 rt 的通用寄存器中。与 addi 指令不同，addiu 指令不进行溢出检查，始终将结果保存到目的寄存器。

- **slti 指令**

指令格式：slti rt, rs, immediate。

指令功能：rt <- (rs < (sign-extended immediate))，将指令中的 16 位立即数进行符号扩展，并与地址为 rs 的通用寄存器的值按照有符号数进行比较。如果 rs 小于扩展后的立即数，则将 1 保存到地址为 rt 的通用寄存器中；否则，将 0 保存到地址为 rt 的通用寄存器中。

- **sltiu 指令**

指令格式：sltiu rt, rs, immediate。

指令功能：rt <- (rs < (sign-extended immediate))，将指令中的 16 位立即数进行符号扩展，并与地址为 rs 的通用寄存器的值按照无符号数进行比较。如果 rs 小于扩展后的立即数，则将 1 保存到地址为 rt 的通用寄存器中；否则，将 0 保存到地址为 rt 的通用寄存器中。

8. clo、clz 指令

31 26	25 21	20 16	15 11	10 6	5 0	
SPECIAL2 011100	rs	rt	rd	00000	CLZ 100000	CLZ 指令
SPECIAL2 011100	rs	rt	rd	00000	CLO 100001	CLO 指令

- **clz 指令**

指令格式：clz rd, rs。

指令功能：rd <- count leading zeros(rs)，对地址为 rs 的通用寄存器的值，从其最高位开始向最低位方向检查，直到遇到值为 1 的位，将该位之前“0”的个数保存到地址为 rd 的通用寄存器中。如果地址为 rs 的通用寄存器的所有位都为 0（即 0x00000000），那么将 32 保存到地址为 rd 的通用寄存器中。

- clo 指令

指令格式：clo rd, rs。

指令功能：rd <- count leading ones(rs)，对地址为 rs 的通用寄存器的值，从其最高位开始向最低位方向检查，直到遇到值为 0 的位，将该位之前“1”的个数保存到地址为 rd 的通用寄存器中。如果地址为 rs 的通用寄存器的所有位都为 1（即 0xFFFFFFFF），那么将 32 保存到地址为 rd 的通用寄存器中。

9. multu、mult、mul 指令

31 26	25 21	20 16	15 11	10 6	5 0	
SPECIAL2 011100	rs	rt	rd	00000	MUL 000010	MUL 指令
SPECIAL 000000	rs	rt	00000	00000	MULT 011000	MULT 指令
SPECIAL 000000	rs	rt	00000	00000	MULTU 011001	MULTU 指令

- mul 指令

指令格式：mul rd, rs, rt。

指令功能：rd <- rs * rt，将地址为 rs 的通用寄存器的值与地址为 rt 的通用寄存器的值作为有符号数相乘，乘法结果的低 32 位保存到地址为 rd 的通用寄存器中。

- mult 指令

指令格式：mult rs, rt。

指令功能：{hi, lo} <- rs * rt，将地址为 rs 的通用寄存器的值与地址为 rt 的通用寄存器的值作为有符号数相乘，乘法结果的低 32 位保存到 LO 寄存器中，高 32 位保存到 HI 寄存器中。

- multu 指令

指令格式：multu rs, rt。

指令功能：{hi, lo} <- rs * rt，将地址为 rs 的通用寄存器的值与地址为 rt 的通用寄存器的值作为无符号数相乘，乘法结果的低 32 位保存到 LO 寄存器中，高 32 位保存到 HI 寄存器中。与 mult 指令不同，multu 指令将操作数作为无符号数进行乘法运算。

10. madd、maddu、msub、msubu 指令

31 26	25 21	20 16	15 11	10 6	5 0	
SPECIAL2 011100	rs	rt	00000	00000	MADD 000000	MADD 指令
SPECIAL2 011100	rs	rt	00000	00000	MADDU 000001	MADDU 指令
SPECIAL2 011100	rs	rt	00000	00000	MSUB 000100	MSUB 指令
SPECIAL2 011100	rs	rt	00000	00000	MSUBU 000101	MSUBU 指令

- **madd 指令**

指令格式：madd rs, rt。

指令功能：{HI, LO} <- {HI, LO} + (rs * rt)，将地址为 rs 的通用寄存器的值与地址为 rt 的通用寄存器的值作为有符号数进行乘法运算，运算结果与{HI, LO}相加。相加的结果保存到{HI, LO}中，其中{HI, LO}表示 HI 和 LO 寄存器连接形成的 64 位数，HI 是高 32 位，LO 是低 32 位。

- **maddu 指令**

指令格式：maddu rs, rt。

指令功能：{HI, LO} <- {HI, LO} + (rs * rt)，将地址为 rs 的通用寄存器的值与地址为 rt 的通用寄存器的值作为无符号数进行乘法运算，运算结果与{HI, LO}相加。相加的结果保存到{HI, LO}中。

- **msub 指令**

指令格式：msub rs, rt。

指令功能：{HI, LO} <- {HI, LO} - (rs * rt)，将地址为 rs 的通用寄存器的值与地址为

rt 的通用寄存器的值作为有符号数进行乘法运算，然后使用{HI, LO}减去乘法结果。相减的结果保存到{HI, LO}中。

- msubu 指令

指令格式：msubu rs, rt。

指令功能：{HI, LO} <- {HI, LO} - (rs * rt)，将地址为 rs 的通用寄存器的值与地址为 rt 的通用寄存器的值作为无符号数进行乘法运算，然后使用{HI, LO}减去乘法结果。相减的结果保存到{HI, LO}中。

11. div、divu 指令

31 26	25 21	20 16	15 11	10 6	5 0	
SPECIAL 000000	rs	rt	00000	00000	DIV 011010	DIV 指令
SPECIAL 000000	rs	rt	00000	00000	DIVU 011011	DIVU 指令

- div 指令

指令格式：div rs, rt。

指令功能：{HI, LO} <- rs / rt，将地址为 rs 的通用寄存器的值与地址为 rt 的通用寄存器的值作为有符号数进行除法运算。商保存到 LO 寄存器，余数保存到 HI 寄存器中。

- divu 指令

指令格式：divu rs, rt。

指令功能：{HI, LO} <- rs / rt，将地址为 rs 的通用寄存器的值与地址为 rt 的通用寄存器的值作为无符号数进行除法运算。商保存到 LO 寄存器，余数保存到 HI 寄存器中。

12. jr 、jalr 、j、jal 指令

31 26	25 21	20 16	15 11	10 6	5 0	
SPECIAL 000000	rs	00000	00000	00000	JR 001000	JR 指令

SPECIAL 000000	rs	00000	rd	00000	JALR 001001	JALR 指令
J 000010	instr_index					J 指令
JAL 000011	instr_index					JAL 指令

- **jr 指令**

指令格式：jr rs。

指令功能：PC <- rs，将地址为 rs 的通用寄存器的值赋给程序计数器 PC，作为新的指令地址，实现跳转。

- **jalr 指令**

指令格式：jalr rs; jalr rd, rs

指令功能：rd <- return address, PC <- rs，将地址为 rs 的通用寄存器的值赋给程序计数器 PC，作为新的指令地址，同时将跳转指令后面第 2 条指令的地址作为返回地址保存到地址为 rd 的通用寄存器。如果 rd 未指定，则默认将返回地址保存到寄存器$31。

- **j 指令**

指令格式：j target。

指令功能：PC <- (PC + 4)[31:28] | (target << 2)，跳转到新的指令地址，其中新指令地址的低 28 位由指令中的 target 左移两位得到，高 4 位由跳转指令后面延迟槽指令的地址高 4 位保持不变。。

- **jal 指令**

指令格式：jal target。

指令功能：PC <- (PC + 4)[31:28] | (target << 2), $31 <- return address，跳转到新的指令地址，计算方式与 j 指令相同。但 jal 指令还会将跳转指令后面第 2 条指令的地址作为返回地址保存到寄存器$31。

13. beq、b、bgtz、blez、bne、bltz、blzal、bgez、bgezal、bal 指令

31 26 25 21 20 16 15 0

BEQ 000100	**rs**	**00000**	**offset**	**BEQ 指令**
BEQ 000100	**00000**	**00000**	**offset**	**B 指令**
BGTZ 000111	**rs**	**00000**	**offset**	**BGTZ 指令**
BLEZ 000110	**rs**	**00000**	**offset**	**BLEZ 指令**
BNE 000101	**rs**	**rt**	**offset**	**BNE 指令**
REGIMM 000001	**rs**	**BLTZ 00000**	**offset**	**BLTZ 指令**
REGIMM 000001	**rs**	**BLTZAL 10000**	**offset**	**BLTZAL 指令**
REGIMM 000001	**rs**	**BGEZ 00001**	**offset**	**BGEZ 指令**
REGIMM 000001	**rs**	**BGEAZAL 10001**	**offset**	**BGEAZAL 指令**
REGIMM 000001	**00000**	**BAL 10001**	**offset**	**BAL 指令**

- **beq 指令**

指令格式：beq rs, rt, offset。

指令功能：if (rs == rt) then branch，将地址为 rs 的通用寄存器的值与地址为 rt 的通用寄存器的值进行比较，如果相等，则发生转移。

- **b 指令**

指令格式：b offset。

指令功能：无条件跳转，计算方式与 beq 相同。当 beq 指令的 rs、rt 都等于 0 时，即为 b 指令，因此在 OpenMIPS 实现时，不需要特意实现 b 指令，只需要实现 beq 指令即可。

- **bgtz 指令**

指令格式：bgtz rs, offset。

指令功能：if (rs > 0) then branch，如果 rs 寄存器的值大于零，则发生转移。

- **blez 指令**

指令格式：blez rs, offset。

指令功能：if (rs <= 0) then branch，如果 rs 寄存器的值小于等于零，则发生转移。

- **bne 指令**

指令格式：bne rs, rt, offset。

指令功能：if (rs != rt) then branch，如果 rs 寄存器的值不等于 rt 寄存器的值，则发生转移。

- **bltz 指令**

指令格式：bltz rs, offset。

指令功能：if (rs < 0) then branch，如果 rs 寄存器的值小于零，则发生转移。

- **bltzal 指令**

指令格式：bltzal rs, offset。

指令功能：if (rs < 0) then branch，如果 rs 寄存器的值小于零，则发生转移，并且将转移指令后面第 2 条指令的地址作为返回地址保存到通用寄存器$31。

- **bgez 指令**

指令格式：bgez rs, offset。

指令功能：if (rs >= 0) then branch，如果 rs 寄存器的值大于等于零，则发生转移。

- **bgezal 指令**

指令格式：bgezal rs, offset。

指令功能：if (rs >= 0) then branch，如果 rs 寄存器的值大于等于零，则发生转移，并且将转移指令后面第 2 条指令的地址作为返回地址保存到通用寄存器$31。

- **bal 指令**

指令格式：bal offset。

指令功能：无条件跳转，并且将转移指令后面第 2 条指令的地址作为返回地址保存到通用寄存器$31。由于 bal 指令是 bgezal 指令的特殊情况（当 bgezal 的 rs 为 0 时），所以在 OpenMIPS 实现时，不需要特意实现 bal 指令，只需实现 bgezal 指令即可。

14. lb、lbu、lh、lhu、lw、sb、sh、sw 指令

31 26	25 21	20 16	15 0	
LB 100000	base	rt	offset	LB 指令
LBU 100100	base	rt	offset	LBU 指令
LH 100001	base	rt	offset	LH 指令
LHU 100101	base	rt	offset	LHU 指令
LW 100011	base	rt	offset	LW 指令
SB 101000	base	rt	offset	SB 指令
SH 101001	base	rt	offset	SH 指令
SW 101011	base	rt	offset	SW 指令

- **lb 指令**

指令格式：lb rt, offset(base)。

指令功能：lb 指令用于从内存中指定的加载地址处读取一个 字节 (8 位)，然后进行 符号扩展，将其扩展为 32 位，并存入目标通用寄存器 rt。

- lbu 指令

指令格式：bal offset。

指令功能：lbu 指令从内存中指定的加载地址处读取 一个字节 (8 位)，但与 lb 不同，它执行的是 无符号扩展，将其扩展为 32 位，并存入目标通用寄存器 rt。

- lh 指令

指令格式：bal offset。

指令功能：lh 指令从内存中指定的加载地址处读取 一个半字 (16 位)，然后进行 符号扩展，将其扩展为 32 位，并存入目标通用寄存器 rt。该指令要求 地址必须为 2 字节对齐（最低位为 0）。

- lhu 指令

指令格式：bal offset。

指令功能：lhu 指令从内存中指定的加载地址处读取 一个半字 (16 位)，并进行 无符号扩展，将其扩展为 32 位，存入目标通用寄存器 rt。与 lh 一样，该指令要求 地址必须 2 字节对齐。

- lw 指令

指令格式：bal offset。

指令功能：lw 指令从内存中指定的加载地址处读取 一个字 (32 位)，并存入目标通用寄存器 rt。该指令要求 地址必须 4 字节对齐（最低两位必须为 00）。

- sb 指令

指令格式：sb rt, offset(base)。

指令功能：sb 指令用于将 通用寄存器 rt 的最低字节 (8 位) 存储到 内存中的指定地址。

- sh 指令

指令格式：sh rt, offset(base)。

指令功能：sh 指令用于将 通用寄存器 rt 的最低两个字节 (16 位) 存储到 内存中的指定地址。该指令要求 存储地址必须 2 字节对齐（最低位必须为 0）。

- sw 指令

指令格式：sw rt, offset(base)。

指令功能：sw 指令用于将 通用寄存器 rt 的整个 32 位值 存储到 内存中的指定地址。该指令要求 存储地址必须 4 字节对齐（最低两位必须为 00）。

15. lwl、lwr、swl、swr 指令

31 26	25 21	20 16	15 0	
LWL 100010	base	rt	offset	LWL 指令
LWR 100110	base	rt	offset	LWR 指令
SWL 101010	base	rt	offset	SWL 指令
SWR 101110	base	rt	offset	SWR 指令

- lwl 指令

指令格式：lwl rt, offset(base)。

指令功能：从 计算出的 loadaddr align 处加载一个完整字（4 字节），然后仅保留该字的 最低 4-n 个字节 并存入寄存器 rt 的 高位，寄存器 rt 的低位不变。该指令 不要求对齐地址，适用于非对齐数据加载。

- lwr 指令

指令格式：lwr rt, offset(base)。

指令功能：从 计算出的 loadaddr align 处加载一个完整字（4 字节），然后仅保留该字的 最高 n+1 个字节 并存入寄存器 rt 的 低位，寄存器 rt 的高位不变。该指令 不要求对齐地址，适用于非对齐数据加载。

- swl 指令

指令格式：swl rt, offset(base)。

指令功能：计算 存储地址 storeaddr align，将 rt 高位的 4-n 个字节 存入 storeaddr 处，存储地址无需对齐。在大端模式下，swl 存储 rt 的 高位部分 到 storeaddr 及其后续地址。

- swr 指令

指令格式：swr rt, offset(base)。

指令功能：计算 存储地址 storeaddr align，将 rt 低位的 n+1 个字节 存入 storeaddr 处，存储地址无需对齐。在大端模式下，swr 存储 rt 的 低位部分 到 storeaddr 及其前续地址。

16. ll、sc 指令

31 26	25 21	20 16	15 0	
LL 110000	base	rt	offset	LL 指令
SC 111000	base	rt	offset	SC 指令

- **ll 指令**

指令格式：ll rt, offset(base)。

指令功能：从 loadaddr 处读取 一个字（4 字节） 并 符号扩展至 32 位，存入 通用寄存器 rt。设置 LLbit 为 1，表示该加载操作已经链接，后续的 SC 操作可检查该状态。

- **sc 指令**

指令格式：sc rt, offset(base)。

指令功能：如果 RMW 序列没有受到干扰，也就是 LLbit 为 1，那么将地址为 rt 的通用寄存器的值保存到内存中指定的存储地址处，同时设置地址为的通用寄存器的值为 1，设置 LLbit 为 0。如果 RMW 序列受到了干扰，也就是 LLbit 为 0，那么不修改内存，同时设置地址为 rt 的通用寄存器的值为 0。

17. mtc0、mfc0 指令

31 26	25 21	20 16	15 11	10 3	2 0	
COP0 010000	MT 00100	rt	rd	00000000	sel	MTC0 指令
COP0 010000	MF 00000	rt	rd	00000000	sel	MFC0 指令

- **mtc0 指令**

指令格式：mtc0 rt, rd。

指令功能：将通用寄存器 rt 中的值移动到协处理器 CP0 中的 rd 寄存器。

- **mfc0 指令**

指令格式：mfc0 rt, rd。

指令功能：将协处理器 CP0 中 rd 寄存器的值读取到通用寄存器 rt。

18. teq、tge、tgeu、tlt、tltu、tne 指令

31 26	25 21	20 16	15 6	5 0	
SPECIAL 000000	rs	rt	code	TEQ 110100	TEQ 指令
SPECIAL 000000	rs	rt	code	TGE 110000	TGE 指令
SPECIAL 000000	rs	rt	code	TGEU 110001	TGEU 指令
SPECIAL 000000	rs	rt	code	TLT 110010	TLT 指令
SPECIAL 000000	rs	rt	code	TLTU 110011	TLTU 指令
SPECIAL 000000	rs	rt	code	TNE 110110	TNE 指令

- **teq 指令**

指令格式：teq rs, rt。

指令功能：将通用寄存器 rs 和 rt 的值进行比较，如果两者相等，则引发自陷异常。也就是如果 GPR[rs] = GPR[rt]，则触发异常。

- **tge 指令**

指令格式：tge rs, rt。

指令功能：将 rs 和 rt 中的值作为有符号数进行比较，如果 rs 的值大于或等于 rt 的值，则引发自陷异常。也就是如果 GPR[rs] ≥ GPR[rt]，则触发异常。

- **tgeu 指令**

指令格式：tgeu rs, rt。

指令功能：将 rs 和 rt 中的值作为无符号数进行比较，如果 rs 的值大于或等于 rt 的值，则引发自陷异常。也就是如果 GPR[rs] ≥ GPR[rt]（无符号），则触发异常。

- **tlt 指令**

指令格式：tlt rs, rt。

指令功能：将 rs 和 rt 中的值作为有符号数进行比较，如果 rs 的值小于 rt 的值，则引发自陷异常。也就是如果 GPR[rs] < GPR[rt]，则触发异常。

- **tltu 指令**

指令格式：tltu rs, rt。

指令功能：将 rs 和 rt 中的值作为无符号数进行比较，如果 rs 的值小于 rt 的值，则引发自陷异常。也就是如果 GPR[rs] < GPR[rt]（无符号），则触发异常。

- **tne 指令**

指令格式：tne rs, rt。

指令功能：将 rs 和 rt 中的值进行比较，如果两者不相等，则引发自陷异常。也就是如果 GPR[rs] ≠ GPR[rt]，则触发异常。

19. teqi、tgei、tgeiu、tlti、tltiu、tnei 指令

31 26	25 21	20 16	15 0	
REGIMM 000001	rs	TEQI 01100	immediate	TEQI 指令
REGIMM 000001	rs	TGEI 01000	immediate	TGEI 指令
REGIMM 000001	rs	TGEIU 01001	immediate	TGEIU 指令

REGIMM 000001	rs	TLTI 01010	immediate	TLTI 指令
REGIMM 000001	rs	TLTIU 01011	immediate	TLTIU 指令
REGIMM 000001	rs	TNEI 01110	immediate	TNEI 指令

- **teqi 指令**

指令格式：teqi rs, immediate。

指令功能：将通用寄存器 rs 的值与立即数符号扩展至 32 位后的值进行比较，如果两者相等，则引发自陷异常。也就是，如果 GPR[rs] = sign_extended(immediate)，则触发异常。

- **tgei 指令**

指令格式：tgei rs, immediate。

指令功能：将 rs 的值与立即数符号扩展后的值作为有符号数进行比较，如果 rs 的值大于或等于立即数的值，则引发自陷异常。也就是，如果 GPR[rs] ≥ sign_extended(immediate)，则触发异常。

- **tgeiu 指令**

指令格式：tgeiu rs, immediate。

指令功能：将 rs 的值与立即数符号扩展后的值作为无符号数进行比较，如果 rs 的值大于或等于立即数的值，则引发自陷异常。也就是，如果 GPR[rs] ≥ sign_extended(immediate)（无符号），则触发异常。

- **tlti 指令**

指令格式：tlti rs, immediate。

指令功能：将 rs 的值与立即数符号扩展后的值作为有符号数进行比较，如果 rs 的值小于立即数的值，则引发自陷异常。也就是，如果 GPR[rs] < sign_extended(immediate)，则触发异常。

- **tltiu 指令**

指令格式：tltiu rs, immediate。

指令功能：将 rs 的值与立即数符号扩展后的值作为无符号数进行比较，如果 rs 的值小于立即数的值，则引发自陷异常。也就是，如果 GPR[rs] < sign_extended(immediate)（无符号），则触发异常。

- **tnei 指令**

指令格式：tnei rs, immediate。

指令功能：将 rs 的值与立即数符号扩展后的值进行比较，如果两者不相等，则引发自陷异常。也就是，如果 GPR[rs] ≠ sign_extended(immediate)，则触发异常。

20. syscall 指令

31 26	25 6	5 0	
SPECIAL 000000	Code	SYSCALL 001100	SYSCALL 指令

- **syscall 指令**

指令格式：syscall。

指令功能：通过调用 syscall 指令来引发系统调用异常，进入异常处理例程，从而切换到内核模式。

21. eret 指令

31 26	25	24 6	5 0	
COP0 010000	CO 1	0000 0000 0000 0000 000	ERET 011000	ERET 指令

- **eret 指令**

指令格式：eret。

指令功能：从异常处理例程返回。

4.2 总体结构和数据通路设计

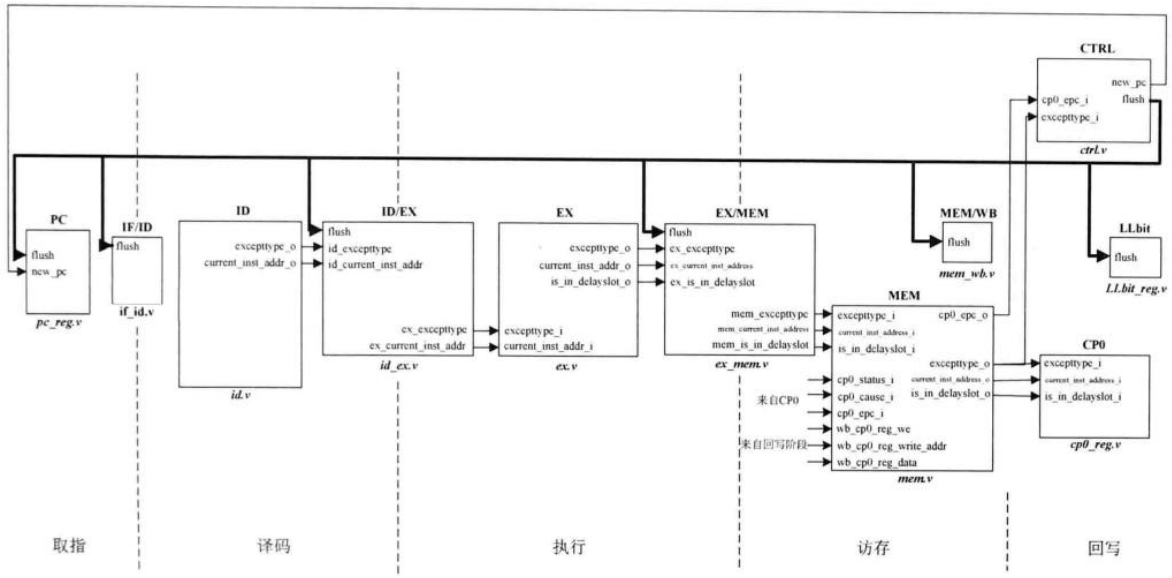


图 3 整体系统结构

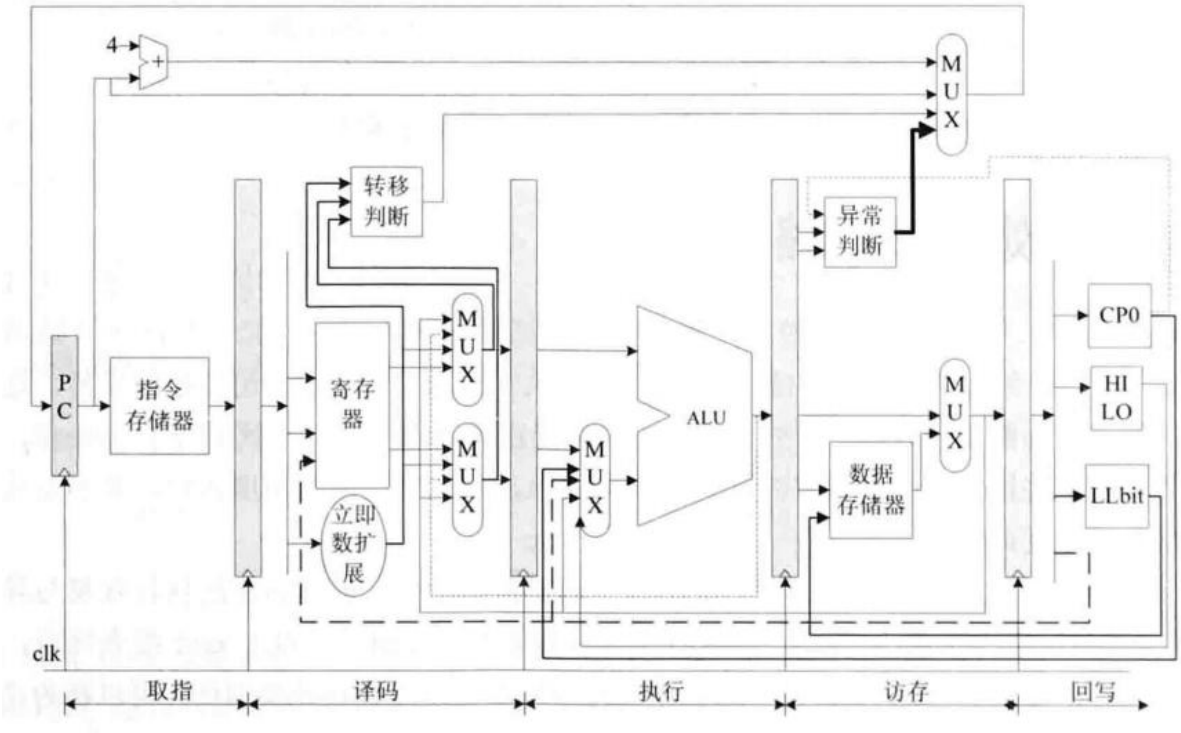


图 4 整体数据流图

4.3 模块设计

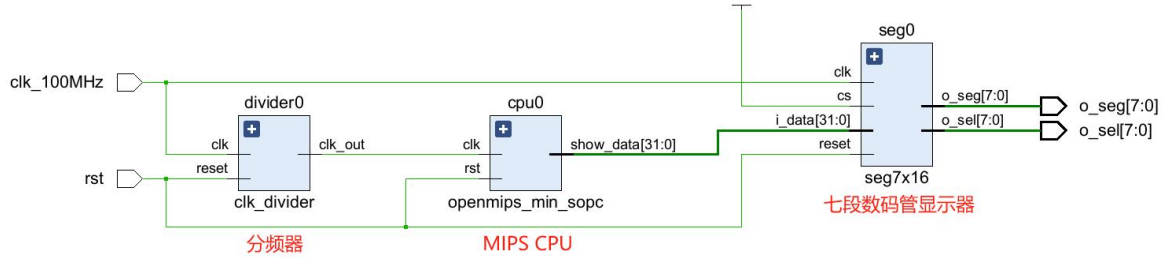


图 5 顶层模块(用于下板展示)

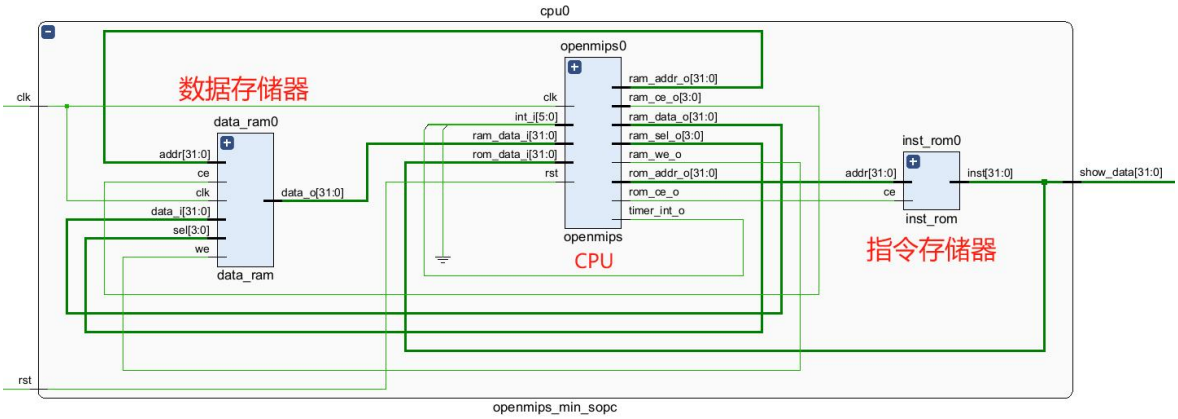


图 6 MIPS CPU 整体模块

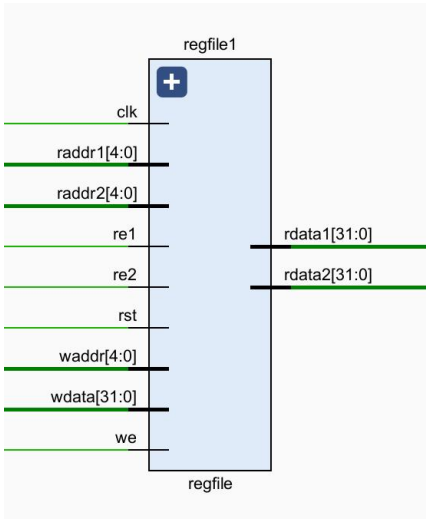


图 7 寄存器组 RegFile 模块

装 订 线

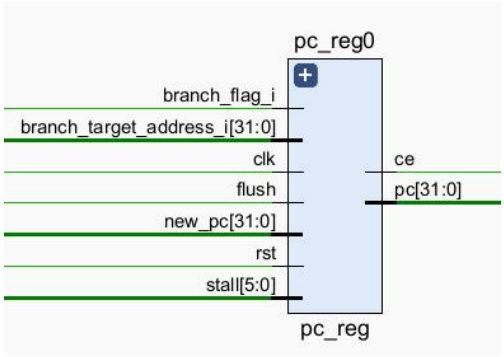


图 8 PC 寄存器模块

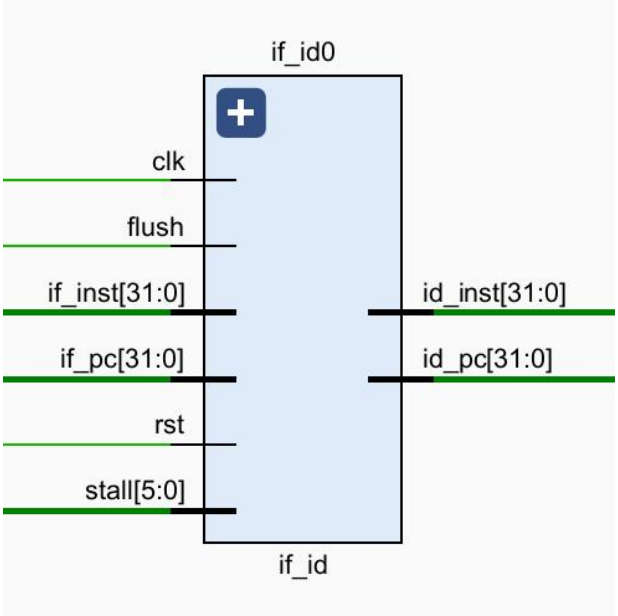


图 9 取指-译指寄存器

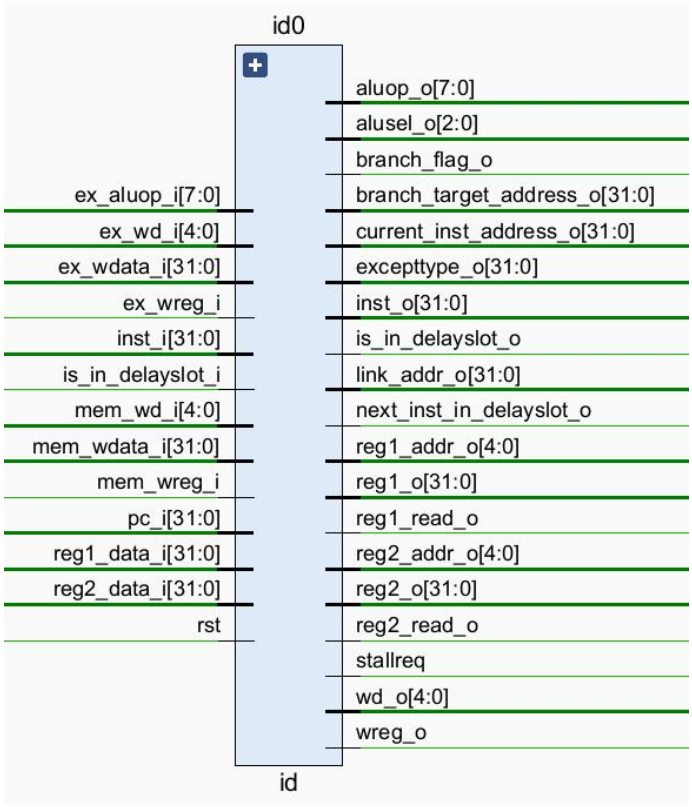


图 10 译指阶段模块

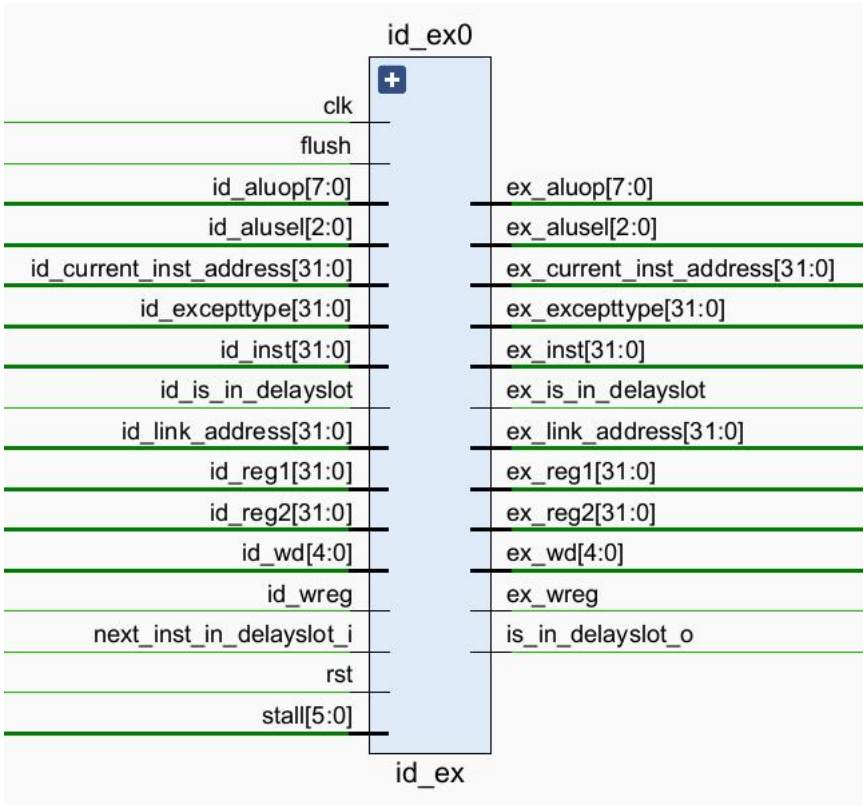


图 11 译指-执行寄存器

装 订 线

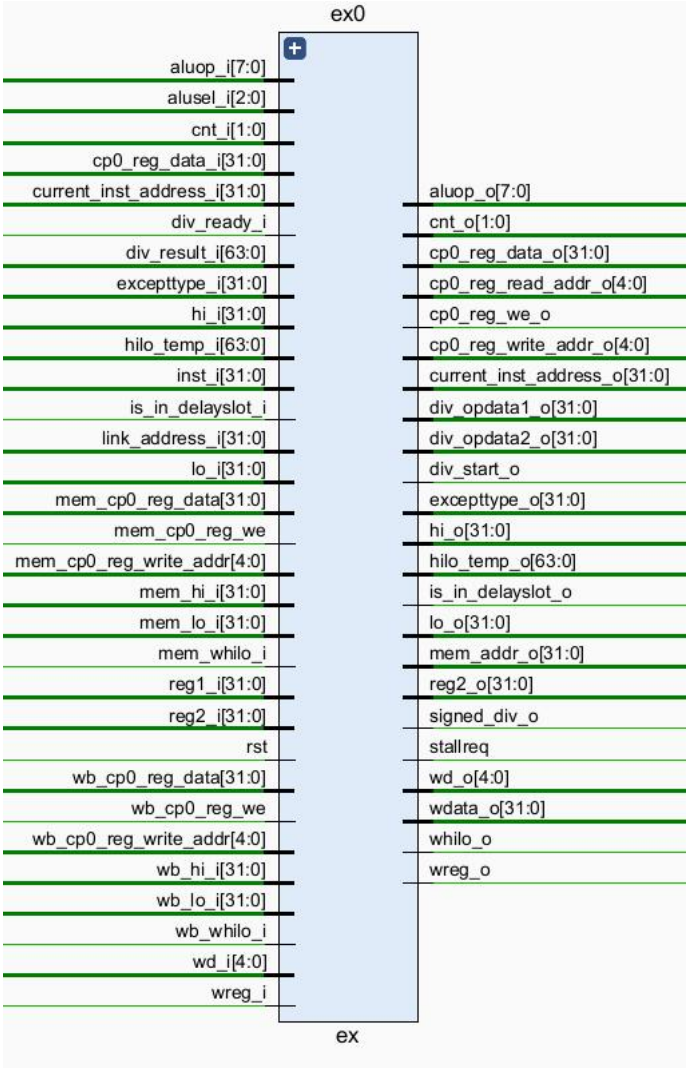


图 12 执行阶段模块

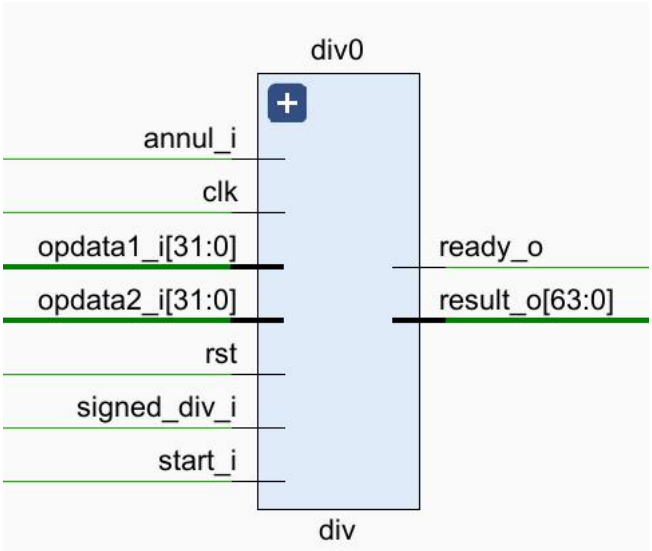


图 13 除法器

装 订 线

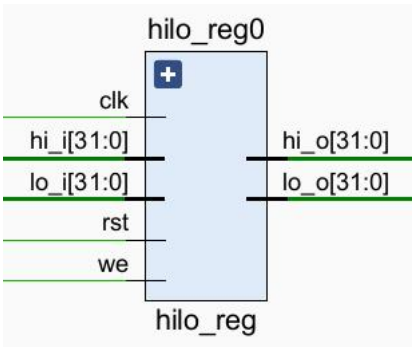


图 14 HI LO 寄存器

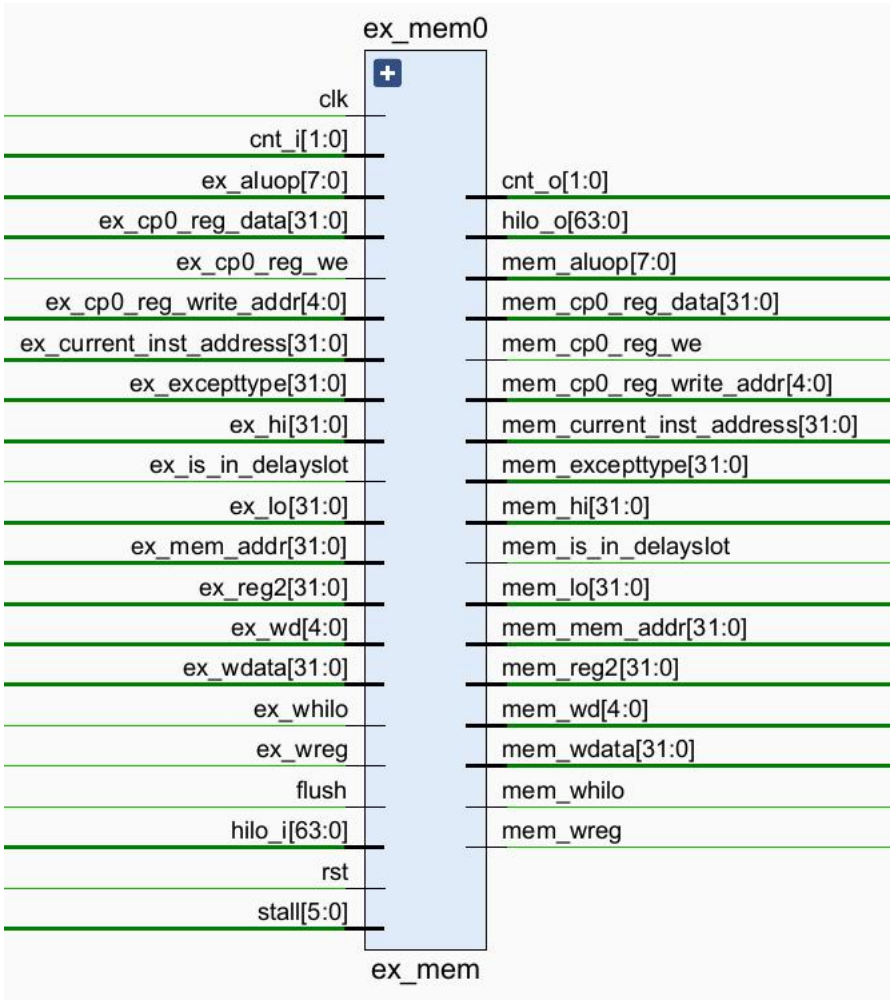


图 15 执行-存储寄存器

装 订 线

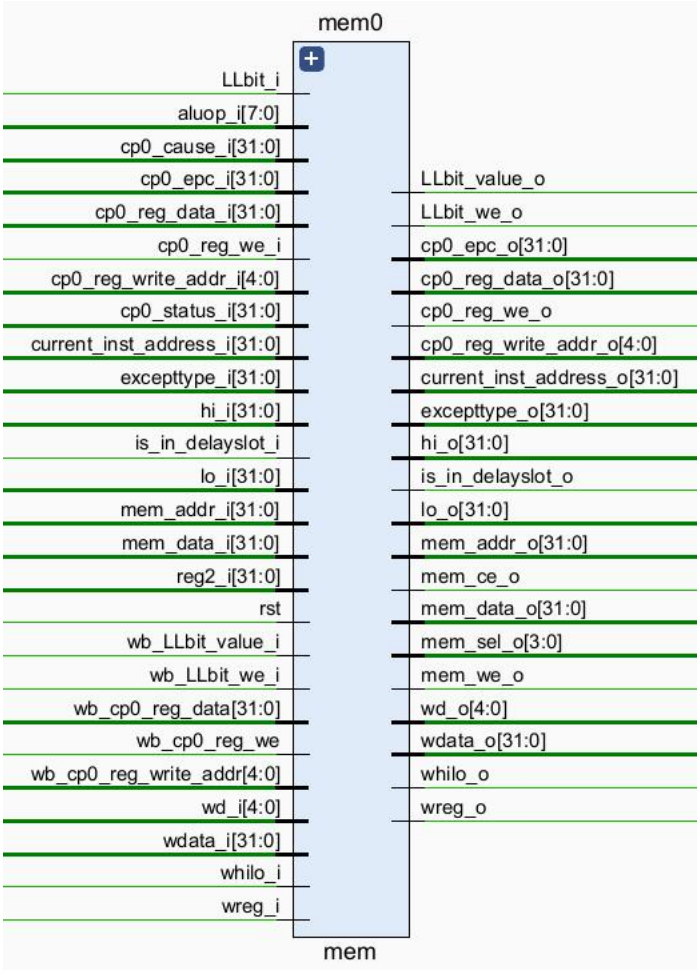


图 16 存储阶段模块

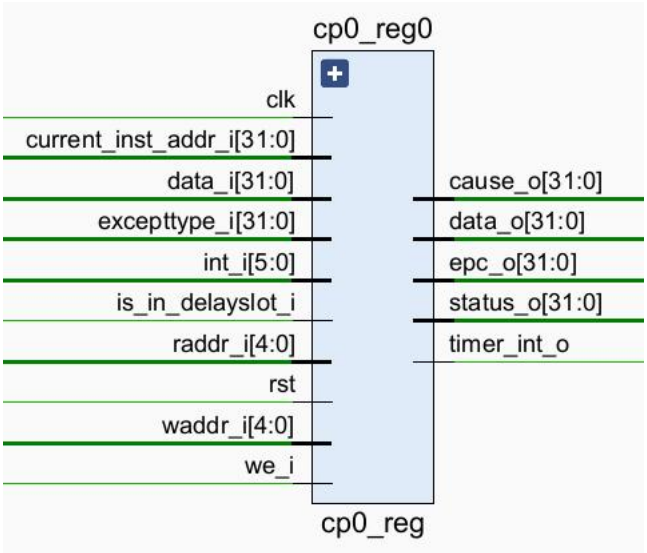


图 17 CPO 寄存器

装 订 线

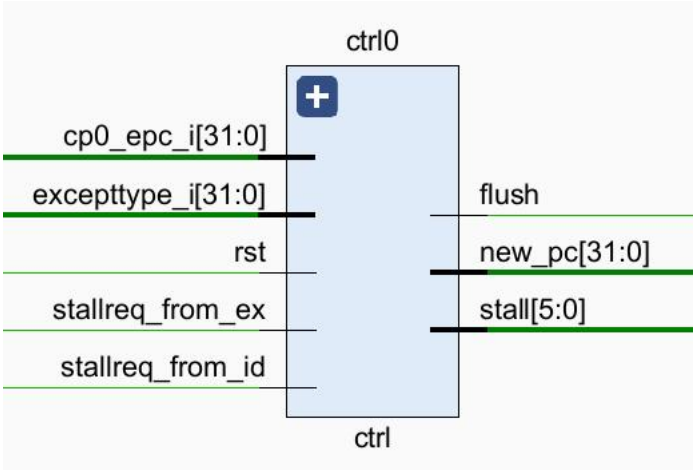


图 18 CTRL 模块

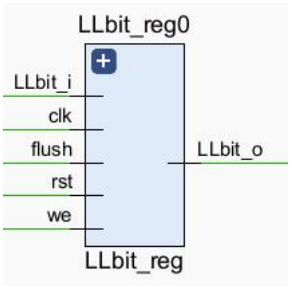


图 19 LLbit 模块

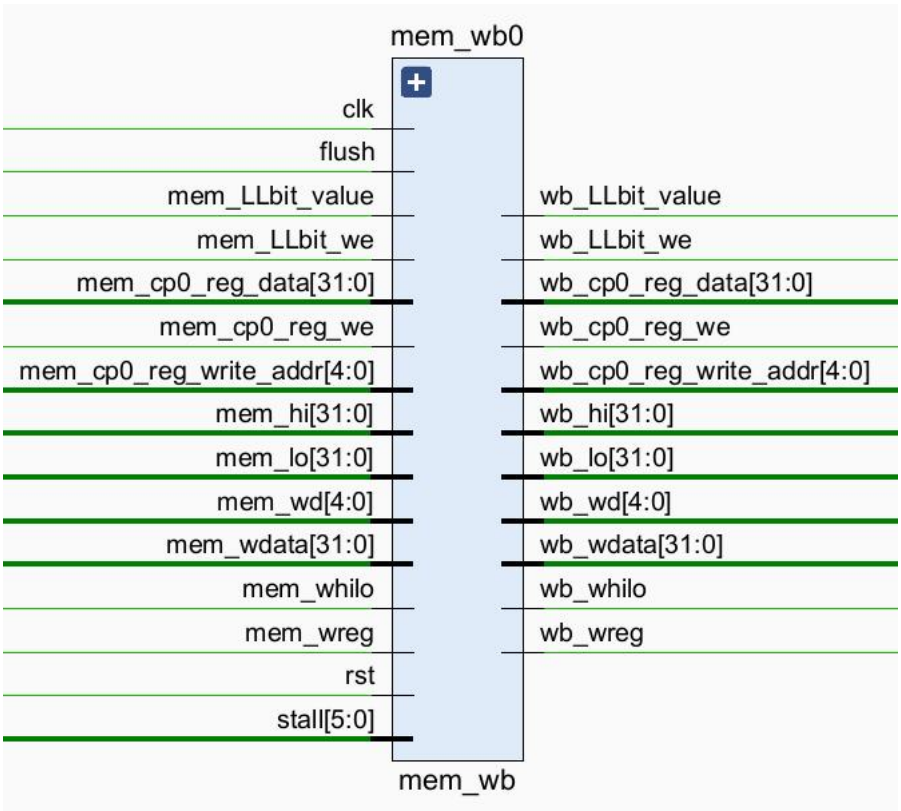


图 20 存储-写回寄存器

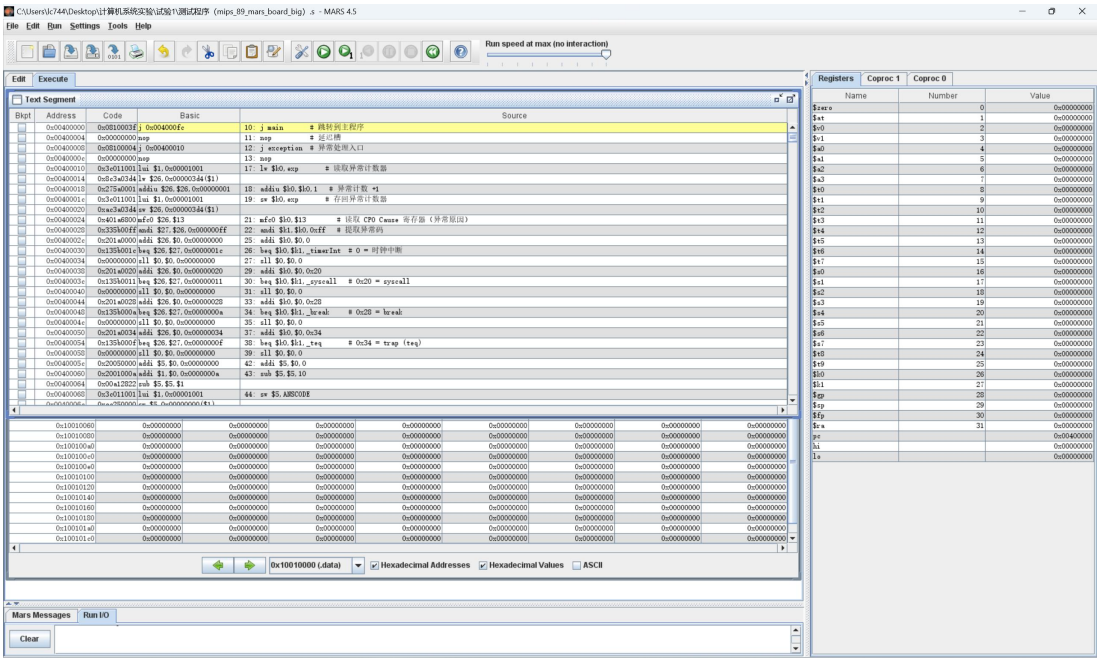
装 订 线

4.4 测试和调试过程与方法

对于 CPU 系统的测试和调试，主要使用的方法是利用 Vivado 自带的仿真工具进行仿真，观察相关的信号的变化，进行 Debug，最终测试和调试出正确的程序。

4.5 仿真过程与方法

① 使用 Mars4_5 将测试汇编程序“测试程序（mips_89_mars_board_big）.s”代码部分转换成 coe 文件。

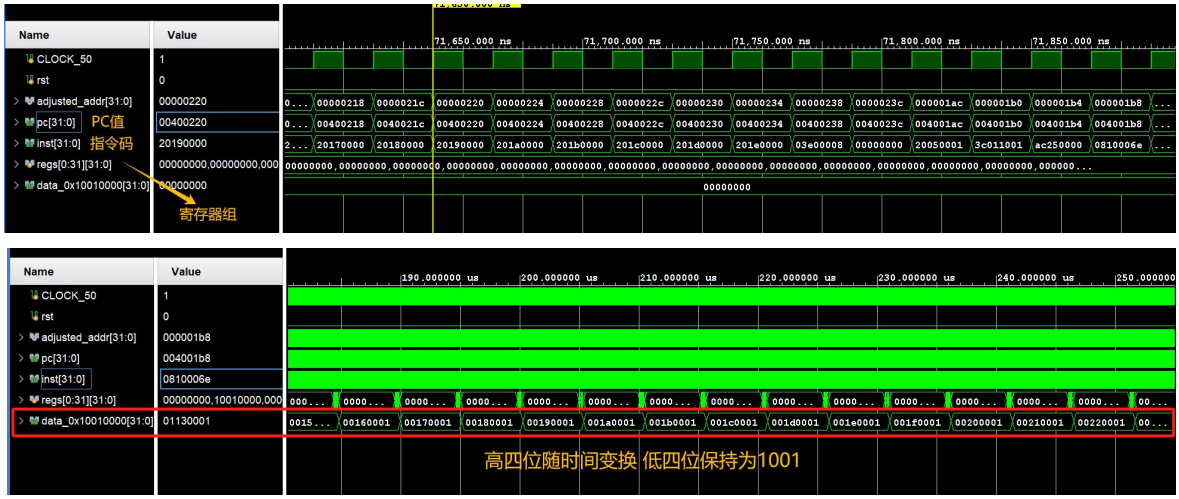


② 使用转换好的 coe 代码文件初始化 ROM IP 核做为指令寄存器。

③ 编写 testbench 文件。

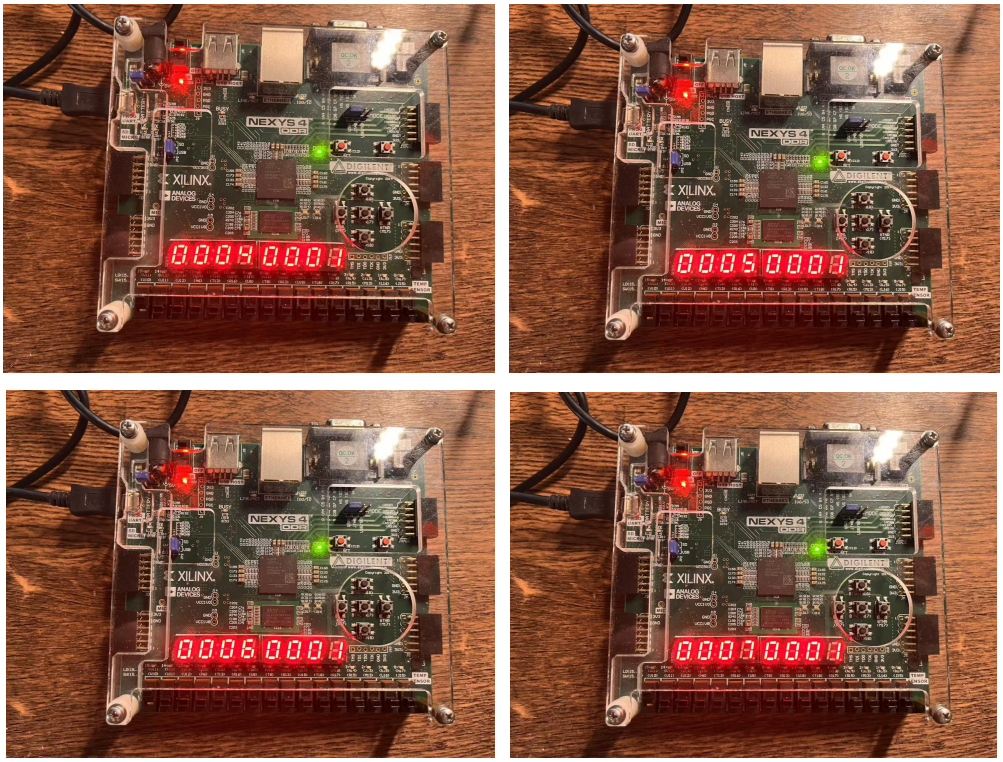
```
`timescale 1ns/1ps

module openmips_min_sopc_tb();

  reg     CLOCK_50;
  reg     rst;

  // 每隔10ns CLOCK_50信号翻转一次
  // 所以一个周期T=20ns 对应50MHz
  initial begin
    CLOCK_50 = 1'b0;
    forever #10 CLOCK_50 = ~CLOCK_50;
  end

  // 最初时刻 复位信号有效 在195ns时复位信号失效SOPC开始运行
  // 运行1000ns后，暂停仿真
  initial begin
    rst = `RstEnable;
    #195 rst= `RstDisable;
    #1000 $stop;
  end

  // 实例化SOPC
  openmips_min_sopc openmips_min_sopc0(
        .clk(CLOCK_50),
        .rst(rst)
  );

endmodule
```

④ 使用 Vivado 自带的仿真工具进行仿真调试。



4.6 下板验证过程与方法

配置约束文件，连接开发板，然后进行综合、生成 bit 文件，下板，可以看到数码管低半字显示 0x0001，高半字随着时钟中断而计数，验证结果正确。



5、实验总结

在这次 CPU 改造实验的过程中，通过深入研读和分析优秀的参考代码实现，我

对硬件开发的工程实践有了全新的认识和体会。这些精心设计的代码不仅功能完善，更在工程规范和组织架构上展现了极高的专业水准，给我带来了诸多启发。

首先，参考代码中宏定义的巧妙使用让我收获颇多。虽然我之前的实现也使用了宏定义，但仅局限于指令译码部分。参考代码则将各类常量、状态和位宽等参数都进行了宏定义封装，通过语义化的命名方式，使代码可读性得到质的飞跃。这种规范化的处理方式不仅便于后续维护，更让代码具备了良好的自解释性。当需要调整参数时，只需修改宏定义即可全局生效，完全避免了全文搜索替换的繁琐操作，大大提升了开发效率。

信号命名规范是另一个让我受益匪浅的方面。参考代码中采用的功能性命名方式，配合_i/_o 等方向标识，使得整个数据通路的走向一目了然。相比我之前随意使用的简写命名，这种规范的命名方式大大提升了代码的可读性和可维护性。

最让我感触深刻的是，参考代码展现出的工程美学。从整齐的代码排版到详尽的注释说明，处处体现着开发者对代码质量的极致追求。这让我意识到，真正的专业不仅体现在功能实现上，更体现在这些看似细微却至关重要的工程细节中。

通过这次实验，我不仅完成了 CPU 改造的任务，更重要的是从优秀的代码中学习到了很多宝贵的经验，这些经验将指引我在未来的开发工作中，以更高的标准要求自己，持续提升代码质量，培养良好的工程习惯。