哈爾濱工業大學

课程报告

课程名称：计算机组成原理

报告题目：给定指令系统的处理器设计

所在院系：未来技术学院

所在专业：人工智能（视听觉信息处理）

学生姓名：刘天瑞

学生学号：7203610121

选课时间：2022 年秋季学期

评阅成绩：

目录

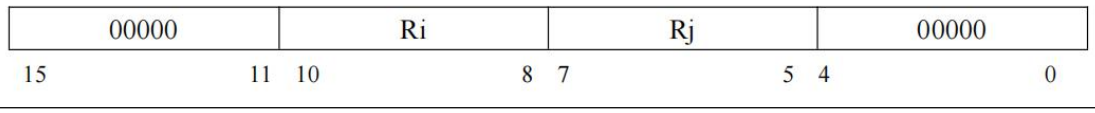
一、 指令格式设计

处理器所支持的指令包括 LDA，STA，MOV，MVI，ADD，SUB，JZ，JMP，IN，OUT。其中仅有 LDA 和 STA 是访存指令，所有的存储器访问都通过这两条指令完成；ADD 和 SUB 是运算指令，MOV 和 MVI 是传数指令，他们都在处理器内部完成；JZ 是跳转指令，根据寄存器的内容进行绝对跳转；JMP 是无条件转移指令；IN 和 OUT 是输入输出指令，所有 I/O 端口与 CPU 之间的通信都由 IN 和 OUT 指令完成，其中寄存器位数位 3，X 为 8 位。

1) 非访存指令

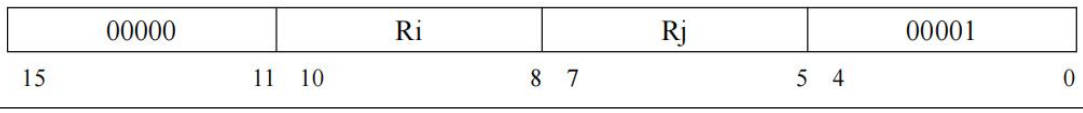
（1）加法指令 ADD　Ri, Rj

该指令在执行阶段，需完成两个寄存器内容相加，结果送回寄存器的操作，具体为：Ri + Rj → Ri，设置指令序列为 00000 XXX XXXXXXXX；

指令	指令格式设计
加法指令 Ri，Rj	

（2）减法指令 SUB　Ri, Rj

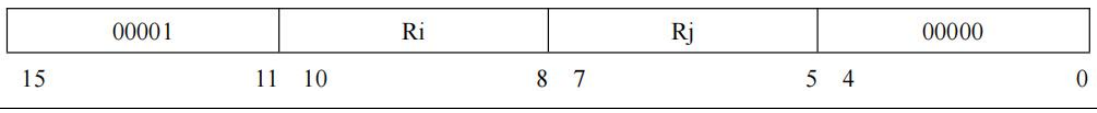
该指令在执行阶段需完成两个寄存器内容相减，结果送回寄存器的操作，具体为：Ri -Rj → Ri，设置指令序列为 00010 XXX XXXXXXXX；

指令	指令格式设计
减法指令 Ri，Rj	

（3）寄存器传送指令 MOV　Ri, Rj

该指令在执行阶段只完成数据信息从寄存器 Rj 向寄存器 Ri 传送的操作，具体为：Rj → Ri，设置指令序列为 00100 XXX 00000 XXX；

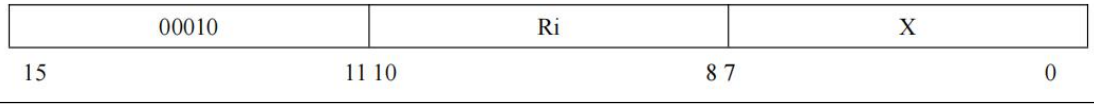
指令格式设计：MOV 指令为双字节长指令，第二字节是要存放的间接地址，含义是将 Rj 中的内容存储到间接地址的单元内存中。

指令	指令格式设计
寄存器传送指令 Ri，Rj	

（4）立即数传送指令 MVI　Ri, X

该指令在执行阶段只完成指令中的 8 位立即数 X 向寄存器 Ri 传送的操作，具体为： X → Ri，设置指令序列为 00110 XXX XXXXXXXX；

指令格式设计：MVI 为双字节长指令，第二字节是要存放的间接地址，含义是将 8 位立即数 X 中的内容存储到间接地址的单元内存中。

指令	指令格式设计
立即数传送指令 Ri，X	

2)访存指令

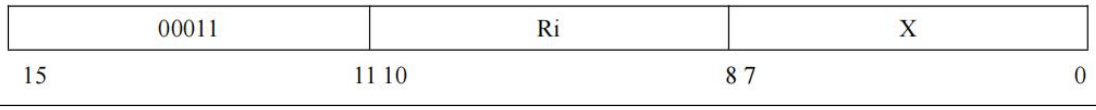
采用扩充寻址的方式支持访存。

扩充寻址的定义：

（1）存数指令 STA　Ri, X

该指令在执行阶段需将寄存器 Ri 的内容存于主存单元中，对应的地址由 8 位形式地址 X 经扩充寻址生成，R7 充当扩充寻址寄存器，即主存实际地址记为 R7//X，具体操作为：Ri → [R7//X（7 down to 0）]，设置指令序列为 01000 XXX XXXXXXXX；

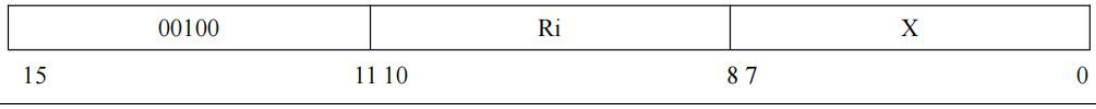
指令格式设计：STA 为双字节长指令，第二个字节是要存放的地址，含义是将 X 中的内容存储到地址的单元内存中。

指令	指令格式设计
存数指令 Ri，X	

（2）取数指令 LDA　Ri, X

该指令在执行阶段需将主存单元中的内容存于寄存器 Ri，对应的地址由 8 位形式地址 X 经扩充寻址生成，R7 充当扩充寻址寄存器，即主存实际地址记为 R7//X，具体操作为：[R7//X（7 down to 0）] → Ri，设置指令序列为 01010 XXX XXXXXXXX；

指令格式设计：LDA 为双字节指令，含义是将内存单元的地址存储于 Ri 中。

指令	指令格式设计
取数指令 Ri，X	

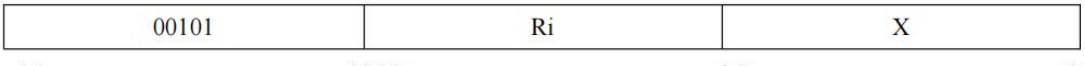
3．转移类指令

（1）条件转移（零则转）指令 JZ　Ri, X

设置指令序列为 01100 XXX XXXXXXXX；

指令格式设计：该指令根据寄存器 Ri 的内容决定下一条指令的地址，若寄存器内容为零，则 8 位形式地址 X 经寄存器 R7 扩充寻址后形成有效地址 R7//X，送至 PC，否则程序

按原顺序执行。具体操作为：if (Ri = 0) then [R7//X（7 down to 0）] → PC

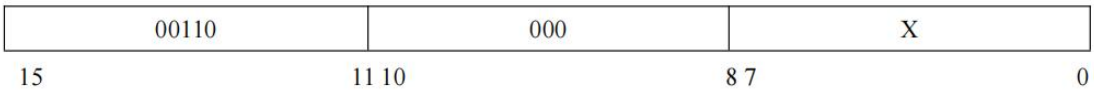
指令	指令格式设计
条件转移指令 Ri，X	

（2）无条件转移指令 JMP　X

设置指令序列为 01110 XXX XXXXXXXX；

该指令改变下一条指令的地址，指令码中的 8 位形式地址 X 经寄存器 R7 扩充寻址后形成有效地址 R7//X，送至 PC，记为：[R7//X（7 down to 0）] → PC

指令格式设计：JMP 为双字节指令，含义是使程序跳转到指定的地址执行。

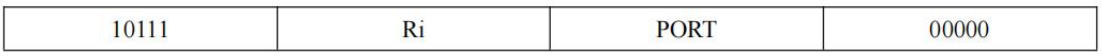
指令	指令格式设计
无条件转移指令 X	

4．I/O 指令（选做，加分项 1）

（1）输入指令 IN　Ri, PORT

该指令完成从 I/O 端口到 CPU 的信息传送，指令码中的端口号 PORT 为端口地址，传送的是端口中的信息，送至 Ri，记为：[PORT] → Ri，设置指令序列为 10000 XXX 000000 XX；

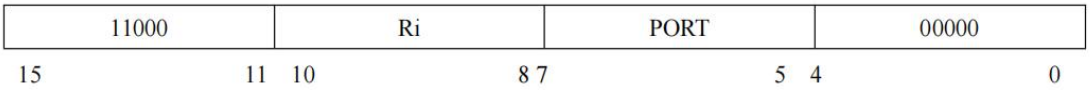
指令格式设计：IN 为单字节长指令，含义是是将输入设备输入的数据放入 Ri 中。

指令	指令格式设计
输入指令 Ri，PORT	

（2）输出指令 OUT　Ri, PORT

该指令完成从 CPU 到 I/O 端口的信息传送，指令码中的端口号 PORT 为端口地址，记为：Ri → [PORT]，设置指令序列为 10010 XXX 000000 XX；

指令格式设计：OUT 为单字节长指令，含义是根据指令提供的地址，将内存中的数据取出由数码管进行显示。

指令	指令格式设计
输出指令 Ri，PORT	

二、 微操作的定义及节拍的划分

在不考虑间接寻址以及中断的情况下，如下所示我分别按照取指阶段以及执行阶段来罗列写出对应机器指令相关的微操作序列：

（i）取指阶段微操作的定义：

T0：
PC->MAR, 1->R
T1：
M(MAR)->MDR, (PC)+1->PC
T2：
MDR->IR, OP(IR)->微地址所形成的部件（编码器：指令码->微地址）（该步骤为组合逻辑，自动完成，所以并不需要控制信号）

（ii）执行阶段微操作的定义：

执行阶段的微操作由操作码的性质决定，同时也需要考虑内存地址的形成问题。，如下所示：

1） 非访存指令

加法指令 ADD Ri，Rj

T0：

Reg(Ad1(IR)) + Reg(Ad2(IR))-> Reg(Ad1(IR))

减法指令 SUB Ri，Rj

T0：

Reg(Ad1(IR)) - Reg(Ad2(IR))-> Reg(Ad1(IR))

寄存器传送指令 MOV Ri，Rj

T0：

Reg(Ad2(IR))-> Reg(Ad1(IR))

立即数传送指令 MVI Ri，Rj

T0：

X -> Reg(Ad1(IR))

2） 访存指令

存数指令 STA Ri，X

T0：

Reg(R7)//Ad(IR) -> MAR;1->W;

T1：

Reg(Ad1(IR)) -> MDR;
T2：
MDR -> M(MAR);
取数指令 LDA Ri，X
T0：
Reg(R7)//Ad(IR) -> MAR;1->W;
T1：
M(MAR) -> MDR;
T2：
MDR -> Reg(Ad1(IR));

3） 转移类指令
条件转移（零则转）指令 JZ Ri，X
T0：
[Zero(Reg(Ad1(IR)))*Reg(R7)//Ad(IR) + Nzero(Reg(Ad1(IR)))] + PC -> PC

无条件转移指令 JMP X
T0：
Reg(R7)//Ad(IR) + PC -> PC

4） I/O 指令
输入指令 IN Ri，PORT
[PORT] -> Reg(Ad1(IR))
输出指令 OUT Ri，PORT
Reg(Ad1(IR)) -> [PORT]

（iii）节拍的划分安排

每个指令周期包含 4 个机器周期（译码部分包含在取指里）分别为取指、运算、访存以及回写周期，每个机器周期包含一个节拍。

1） 取指周期（包括译码周期）：
M(PC) -> IR
1 -> R
PC + 1 -> PC

2） 运算周期：
T1：操作数准备
Reg(Ad1(IR)) -> A，Reg(Ad1(IR)) -> B
Reg(R7)//Ad(IR) -> Addr
T2：运算
ADD：Reg(Ad1(IR)) + Reg(Ad2(IR)) ->Reg(Ad1(IR))
SUB：Reg(Ad1(IR)) - Reg(Ad2(IR)) -> Reg(Ad1(IR))
MVI：Ad(IR) -> Reg(Ad1(IR))

MOV：Reg(Ad2(IR)) -> Reg(Ad1(IR))
JZ：Zero(Reg(Ad1(IR)))*M(Reg7)//Ad(IR) + Nzero(Reg(Ad1(IR)))*PC -> PC
STA：1 -> W;
Reg(Ad1(IR)) -> M((Reg7)//Ad(IR)
LDA：M((Reg7)//Ad(IR)) -> Reg(Ad1(IR))
JMP：M((Reg7)//Ad(IR)) -> PC
IN：1 -> $R_{I/O}$;
[PORT] -> Reg(Ad1(IR))
OUT：1 -> $W_{I/O}$;
Reg(Ad1(IR)) -> [PORT]

3） 访存周期：
取数：M(Addr) -> Rtemp，1 -> R
存数：ALUOUT -> M(Addr)，1 -> W

4） 回写周期：
LDA：Rtemp -> Reg(A)
ADD，SUB，MVI，MOV：ALUOUT -> Reg(A)
JMP：Addr -> PC
JZ：if A = 0 then Addr -> PC

三、 处理器结构

3.1 设计框图

RISC 处理器结构设计框图如下图所示：

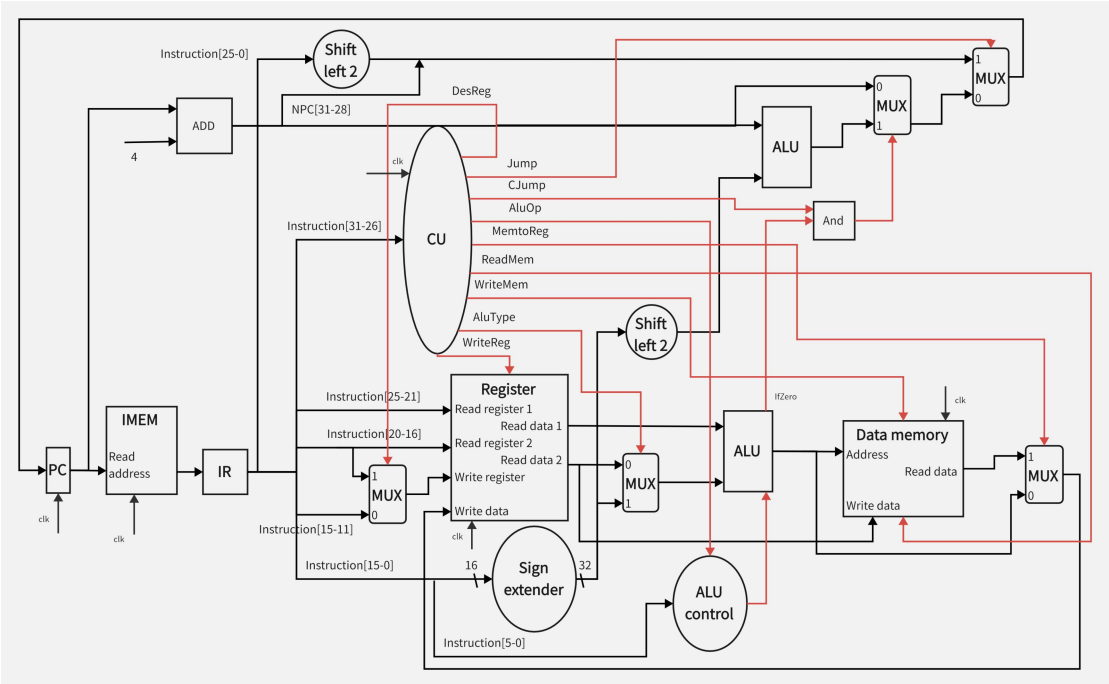


图 1 RISC 处理器结构设计框图

3.2 功能描述

各机器指令功能描述如下所示：

1. ADD：取指，前五位读入 CU，访问到主存中对应的操作微程序，从 Ri 和 Rj 中读数据，进入 ALU，ALU control 选择加法，得到结果，写回 Ri；
2. SUB：取指，前五位读入 CU，访问到主存中对应的操作微程序，从 Ri 和 Rj 中读数据，进入 ALU，ALU control 选择减法，得到结果，写回 Ri；
3. MOV：取指，前五位读入 CU，访问到主存中对应的操作微程序，Rj 读数据，Rj 数据进入 ALU，选择加法，输出 Rj 数据写回 Ri；
4. MVI：取指，前五位读入 CU，访问到主存中对应的操作微程序，读入立即数数据进 ALU，输出写回 Ri；
5. STA：取指，前五位读入 CU，访问到主存中对应的操作微程序，立即数和左移八位的 Reg7 进入 ALU，加法得到地址，Ri 数据作为写入数据；
6. LDA：取指，前五位读入 CU，访问到主存中对应的操作微程序，立即数和左移八位的 Reg7 进入 ALU，加法得到地址，读地址数据，写回 Ri；
7. JZ：取指，前五位读入 CU，访问到主存中对应的操作微程序，Ri 进入下部 ALU，立即数和左移八位的 Reg7 进入 PC 部分的 ALU，只有在判断为 0 后输出信号，PC 得到 ALU 算出数据的结果，跳转；
8. JMP：取指，前五位读入 CU，访问到主存中对应的操作微程序号，立即数和左移八位的 Reg7 进入 PC 部分的 ALU，PC 得到输出结果，跳转；
9. IN：取指，前五位读入 CU，访问到主存中对应的操作微程序，PORT 进入 ALU，访问 I/O 读取得到[PORT]，输出写回 Ri；
10. OUT：取指，前五位读入 CU，访问到主存中对应的操作微程序，PORT 进入 ALU，访问 I/O PORT 地址，写入 Ri 数据。

各功能模块的功能描述如下所示：

1. PC: 用于存放下一条将要执行的指令的地址；
2. IMEM: 为指令缓冲存储器, 用于根据地址寻找出将要执行的指令；
3. IR: 存储此时正要执行的指令；
4. Register（Reg）: 用于读入 rd, rs, rt 等参数,为寄存器组文件；
5. Sign Extend：16 位扩展到 32 位；
6. MUX：数据选择器；
7. ALU：算数运算逻辑单元, 执行和指令相关的一些计算；
8. ALUcontrol：控制 ALU，读低 6 位，让其选择执行哪种操作；
9. DMEM：数据缓冲存储器；
10. Shift Left2：左移两位；
11. Control Unit（CU）：接收反馈信号并根据时钟信号来发出控制信号。

各功能模块输入输出接口信号的具体定义（以表格形式给出，表格内容包括信号名称、位数、方向、来源/去向以及信号意义）：

1. PC: 用于存放下一条将要执行的指令的地址：

信号名称	位数	方向	来源/去向	意义
newAddress_pc	32	input	MUX	新的 pc 地址
clk	1	input	CPU clock	时钟信号
output_pc	32	output	IMEM	输出当前执行的指令地址
resetn	1	input	CPU	重启，初始化

2. IMEM: 为指令缓冲存储器, 用于根据地址寻找出将要执行的指令：

信号名称	位数	方向	来源/去向	意义
inputfrom_pc	32	Input	PC	查找当前指令地址
output_instruction	32	Output	IR	输出当前指令
Clk	1	Input	CPU clock	时钟信号

3. IR: 存储此时正要执行的指令：

信号名称	位数	方向	来源/去向	意义
Inputform_IMEM	32	input	IMEM	存储当前指令
Output_instruction	32	output	CU\Register\MUX\ShiftLeft2\SignExtend	解析输出当前指令

4. Registers: 用于读入 rd, rs, rt 等参数,为寄存器组文件：

信号名称	位数	方向	来源/去向	意义
R1	5	input	IR	指令中的 rs
R2	5	input	IR	指令中的 rt
writeback	32	input	MUX	要写回的数据
WriteRegister	5	Input	Mux	确定写回寄存器地址
output1	32	output	ALU	对应寄存器中的值
output2	32	output	MUX	对应寄存器中的值
Clk	1	Input	CPU	时钟信号

5. Sign Extend：16 位扩展到 32 位：

信号名称	位数	方向	来源/去向	意义
Inputinsturction_IR	16	input	IR	指令低 16 位扩展
Output_extend	32	output	MUX	输出扩展后 32 位

6. MUX：数据选择器：

信号名称	位数	方向	来源/去向	意义
input_data1	32	input	Data_holder	输入的数据
input_data2	32	input	Data_holder	输入的数据
RegDst	1	input	CU	选择 rt 还是 rd，区分两种指令
output	32	output	ALU/Registers	被选择的数据
Jump	1	Input	CU	Jump 指令
ALUsrc	1	Input	CU	选择 I 类指令或 R 类指令

7. ALU：算数运算逻辑单元, 执行和指令相关的一些计算：

信号名称	位数	方向	来源/去向	意义
op	1	input	ALUcontrol	执行哪种运算
input_data1	32	input	MUX	操作数
input_data2	32	input	MUX	操作数
output	32	output	Register\DMEM\MUX	运算结果
Zero	1	Output	与门	0 标志位

8. DMEM：数据缓冲存储器：

信号名称	位数	方向	来源/去向	意义
Address	32	input	ALU	要访问或者写入的地址
clk	1	input	CPU clock	时钟信号
WriteData	32	input	Register	寄存器的值
MEMread	1	Input	CU	是否执行写存
MEMwrite	1	Input	CU	是否执行访存
ReadData	32	Output	MUX	输出数据

9. ALUcontrol：控制 ALU，读低 6 位，让其选择执行哪种操作：

信号名称	位数	方向	来源/去向	意义
ALUOp	1	input	CU	低 6 位分析，选择运算
OutputALUcontrol	1	Output	ALU	根据低六位的操作码

10. Shift Left2：左移两位：

信号名称	位数	方向	来源/去向	意义
Input	26\32	input	IR\SignExtend	左移 2 位
Output	32	output	MUX\ALU	解析输出当前指令

11. Control Unit：接收反馈信号并根据时钟信号发出控制信号：

信号名称	位数	方向	来源/去向	意义

Opcode	6	input	IR	操作码
clk	1	input	CPU clock	时钟信号
RegDst	1	output	MUX	选择 rt 或者 rd 寄存器为目标
Jump	1	output	MUX	跳转指令
Branch	1	output	与门	条件分支，相等跳转使用
MemRead	1	output	DMEM	取数指令使用
MemtoReg	1	output	MUX	取数指令使用，选择读取数据
ALUOp	1	output	ALU control	ALU 选择进行何种操作
MemWrite	1	output	DMEM	存指令使用
ALUSrc	1	output	MUX	选择执行 sw|lw 或者其他类型指令（地址第 16 位在 ALU 中的调用）
RegWrite	1	output	Registers	寄存器是否允许写回

四、 采用微程序设计控制单元

4.1 写出每条机器指令对应的微指令序列

微程序设计控制单元的主要任务是编写对应各条机器指令的微程序，具体步骤是首先写出对应机器指令的全部微操作及节拍安排（前期已经完成），然后确定微指令格式，最后编写出每条微指令的二进制代码（称为微指令码点）。而每条机器指令对应的微指令序列便在编写微指令码点时体现出来。

4.2 确定微指令字长

按照实验要求，需要前提假设模型机存储字长和所设计的 RISC 处理器指令字长均为 16 位，地址字长为 8 位。可以采用 16 位定长指令，操作码为前 5 位,中间 3 位保留，地址码为后 8 位。

指令系统如下表所示：

助记符	操作数	指令码	长度
LDA	[*]	03H	2：取数
STA	[*]	06H	2：存数
MVI		09H	2：立即数传送
JZ	*	0CH	2：条件转移
JMP	*	0DH	2：无条件转移
IN		0EH	2: 输入
OUT		10H	2：输出
MOV		12H	2：寄存器传送
ADD	[*]	13H	2：加法
SUB	[*]	16H	2：减法

4.3 确定微指令格式

微指令格式包括微指令的编码方式、后续微指令下地址的形成方式以及微指令的指令字长（该部分在上一小节已经确定不再赘述）等总共 3 个方面。

（i）微指令的编码方式：

由于上述微操作数不多，可采用直接编码方式，即由微指令控制字段的某一位置直接控制一个微操作。具体方式如下罗列所示：

其中，

第 0 位表示控制 1 -> R；

第 1 位表示控制 M(PC) -> IR；

第 2 位表示控制 (PC) + 2 -> PC；

第 3 位表示控制 Reg(Ad1(IR)) <- M(Reg7)//Ad(IR)；

第 4 位表示控制 Reg(Ad1(IR)) -> M(Reg7)//Ad(IR)；

第 5 位表示控制 1 -> W；

第 6 位表示控制 Reg(Ad1(IR)) <- Ad(IR)；

第 7 位表示控制 1 -> $R_{I/O}$；

第 8 位表示控制 1 -> $W_{I/O}$；

第 9 位表示控制 Reg(Ad1(IR)) <- [PORT]；

第 10 位表示控制 Reg(Ad1(IR)) -> [PORT]；

第 11 位表示控制 Reg(Ad1(IR)) <- Reg(Ad2(IR))；

第 12 位表示控制 Reg(Ad1(IR)) <- Reg(Ad1(IR)) + Reg(Ad2(IR))；

第 13 位表示控制 Reg(Ad1(IR)) <- Reg(Ad1(IR)) - Reg(Ad2(IR))；

第 14 位表示控制 PC <- Zero(Reg(Ad1(IR)))*M((Reg7)//Ad(IR)) + Nzero(Reg(Ad1(IR)))*PC；

第 15 位表示控制 PC <- M((Reg7)//Ad(IR))；

（ii）后续微指令下地址的形成方式：

可以采用微指令下地址的字段和指令的操作码编码两种形成方式。

首先设置一控制位，0 表示前一种方式，1 表示后一种方式，来作为二路数据选择器的控制端输入。其中第 16 位表示控制下地址形成方式：0 表示选择顺序控制字段；1 则表示选择操作码编码结果。

4.4 编写微指令码点

如下表所示为编写好的 10 条机器指令所对应的微指令码点：

微程序名称	微指令地址(八进制)	微指令（二进制代码）																				
		操作控制字段															顺序控制字段				
		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
取指	00H	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
	01H	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0
	02H	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	X	X	X	X	X
LDA	03H	1	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	1	0	1	1	1
	04H	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0
	05H	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

STA	06H	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	1	0	0	1	1
	07H	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	0	0
	08H	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
MVI	09H	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
JZ	0CH	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0
JMP	0DH	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0
IN	0EH	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	1	0	1	1
	0FH	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0
OUT	10H	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	1	1	0	1
	11H	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0
MOV	12H	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0
ADD	13H	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	1	1	1	1
	14H	0	0	1	0	0	0	0	0	0	0	0	1	0	0	0	0	1	0	0	0	0
	15H	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
SUB	16H	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0

五、 用 Verilog 实现该 CPU，并仿真验证其功能

通过代码调试，利用波形进行错误分析，同时也在程序中添加相应的测试变量来查看某些中间变量的具体变化。而关于 PC 的更新：若 PC 在取指模块进行自动加一，跳转指令通过回写来改变 PC，会造成在回写模块难以判断 PC-update 信号值的问题，所以将 PC 加一的操作转移至回写模块进行，PC-update 一直保持有效，如果是跳转类指令，就让 PC 回写跳转地址，其余情况则回写 PC + 1。本次设计中采用自顶向下的硬件编程方法，将原本较为复杂的整体逐步拆分为小模块依次实现，使原本困难的问题得以解决。以下为用 Verilog 语言实现设计的简单指令 16 位 RISC 处理器的具体代码：

（i） 微地址形成部件：

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity uaddr_gen is
   Port(op:in STD_LOGIC_VECTOR(4 downto 0);
        op_add:out STD_LOGIC_VECTOR(4 downto 0));
end uaddr_gen;

architecture Behavioral of uaddr_gen is
begin
process(op) begin
                op_add <= op(3 downto 0)&'0';
end process;
```

（ii） 5 位 2 选 1 多路选择器:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL,IEEE.NUMERIC_STD.ALL;

entity MUX is
PORT(
        mode                    :IN STD_LOGIC; -- 输入控制信号
        next_add    :IN STD_LOGIC_VECTOR(4 DOWNTO 0); --5 位下地址数据端
        op_addr                 :IN STD_LOGIC_VECTOR(4 DOWNTO 0); --5 位微程序入口地址数据端
        out_add     :OUT STD_LOGIC_VECTOR(4 DOWNTO 0); --5 位下地址输出
end MUX;

architecture Behavioral of MUX is
signal tmp              :STD_LOGIC;
begin
tmp <= mode;
process(tmp,next_add,op_addr)begin
        case tmp is
                when'0'=>out_add<=next_add;
                when others=>out_add<=op_addr;
        end case;
end process;
end Behavioral;
```

（iii） 控制存储器:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.std_logic_unsigned.ALL;            --CONV_INTEGER

entity ROM is
   Port(add:in STD_LOGIC_VECTOR(4 downto 0);
      data_out:out STD_LOGIC_VECTOR(0 to 23));
end ROM;

architecture Behavioral of ROM is
        type microcode_array is array(22 downto 0) of std_logic_vector(0 to 21);
        constant code    : microcode_array:=(
    0=> "10000000000000000001",
    1=> "01000000000000000010",
    2=> "00100000000000000UUUUU",
    3=> "10000000010000010111",
```

```
            4=> ”00010000000000011000”,
            5=> ”00100000000000000000”,
            6=> ”00001000000000010011”,
            7=> ”00010000000000010100”,
            8=> ”00000100000000000000”,
            9=> ”00000010000000000000”,
            12=> ”00000000000000100000”,
            13=> ”00000000000000010000”,
            14=> ”00000001000000001011”,
            15=> ”00000000100000000000”,
            16=> ”00000001000000001101”,
            17=> ”00000000010000000000”,
            18=> ”00000000001000000000”,
            19=> ”01000000000100001111”,
            20=> ”00100000001000010000”,
            21=> ”00000000000100000000”,
            22=> ”00000000000010000000”,
                 others=> ”00000000000000000000”);
begin
                data_out <= code(conv_integer(add));
end Behavioral;
```

（iv） CMDR:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity CMDR is
GENERIC(n:Positive:=21); -size of counter/shifter
  Port(
            clock :IN Std_LOGIC; -serial inputs
            u_op :in STD_LOGIC_VECTOR(0 TO (n-1));
  control :out STD_LOGIC_VECTOR(0 TO 14);
            mode_sel: out STD_LOGIC;
  next_add: out STD_LOGIC_VECTOR(4 DOWNTO 0));
end CMDR;

architecture Behavioral of CMDR is
SIGNAL int_reg:Std_logic_vector(0 TO 20);

BEGIN
  main_proc:PROCESS
  BEGIN
  WAIT UNTIL rising_edge(clock);
  int_reg <= u_op;
```

```
 END PROCESS;
 --connect internal register to dataout port
 control <= int_reg(0 TO 14);
 mode_sel<= int_reg(15);
 next_add <= int_reg(16 TO 20);

end Behavioral;
```

（v） 总 CU:

```
library IEEE；
use IEEE.STD_LOGIC_1164.ALL;

entity CU is
   Port(clk:in STD-LOGIC:
        op_code:in STD_LOGIC_VECTOR(4 downto 0);
        ctrl_signal:out STD_LOGIC_VECTOR(14 downto 0));
end CU;

architecture Benavioral of CU is

component CMAR
 GENERIC(n:Positive:=21); -size of counter/shifter
 Port(
              clock :IN Std_LOGIC; -serial inputs
              u_op :in STD_LOGIC_VECTOR(0 TO (n-1));
 control :out STD_LOGIC_VECTOR(0 TO 14);
              mode_sel: out STD_LOGIC;
 next_add: out STD_LOGIC_VECTOR(4 DOWNTO 0));
end component;

component uaddr_gen
   Port(op:in STD_LOGIC_VECTOR(4 downto 0);
        op_add:out STD_LOGIC_VECTOR(4 downto 0));
end component;

component MUX
PORT(
              mode                    :IN STD_LOGIC; --  输入控制信号
              next_add     :IN STD_LOGIC_VECTOR(4 DOWNTO 0); --5  位下地址数据
端
              op_addr                 :IN STD_LOGIC_VECTOR(4 DOWNTO 0); --5
位微程序入口地址数据端
              out_add      :OUT STD_LOGIC_VECTOR(4 DOWNTO 0); --5  位下地址输
```

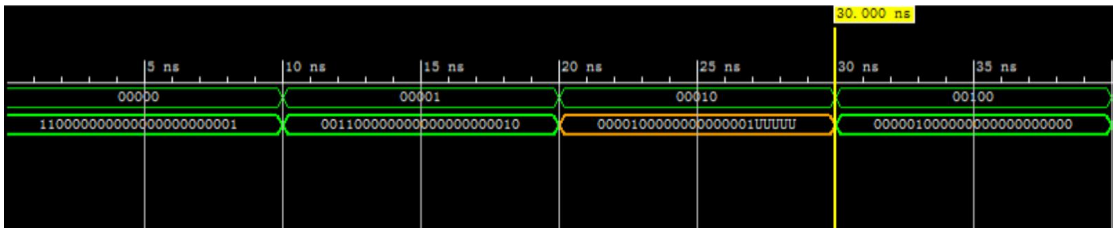
出
end component;

component ROM
  Port(add:in STD_LOGIC_VECTOR(4 downto 0);
    data_out:out STD_LOGIC_VECTOR(0 to 20));
end component;

signal op_add_MUX:std_logic_vector(a downto 0);
signal mode_MUX :std_logic;
signal next_add_MUX:std_logic_vector(4 downto 0);
signal MUX_CM :std_logic_vector(4 downto 0);
signal CM_CMAR :std_logic_vector(0 to 20);

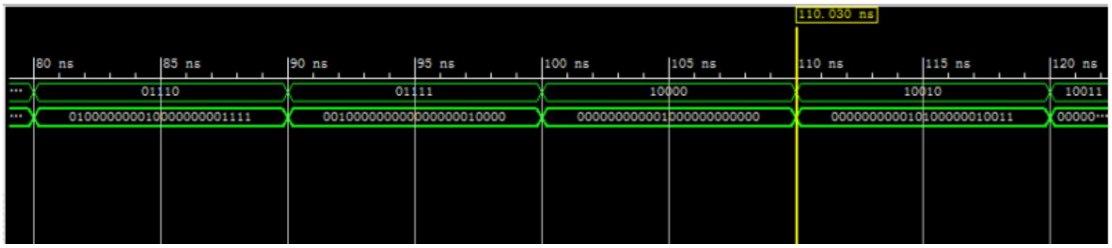
begin
u1:uaddr_gen port map(op_code,op_add_MUX);
u2:MUX port map(mode_MUX,next_add_MUX,op_add_MUX,MUX_CM);
u3:ROM port map(MUX_CM,CM_CMAR);
u4:CMAR port map(clk,CM_CMAR,ctrl_signal,mode_MUX,next_add_MUX);
end Behavioral;

通过仿真波形来验证设计好的处理器各机器指令的功能是否正常，如下所示按步骤一一输出各条微指令：

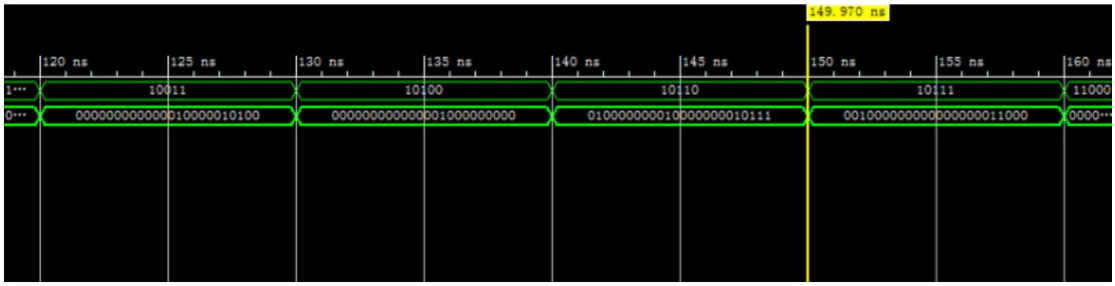


00H 01H 02H LDA

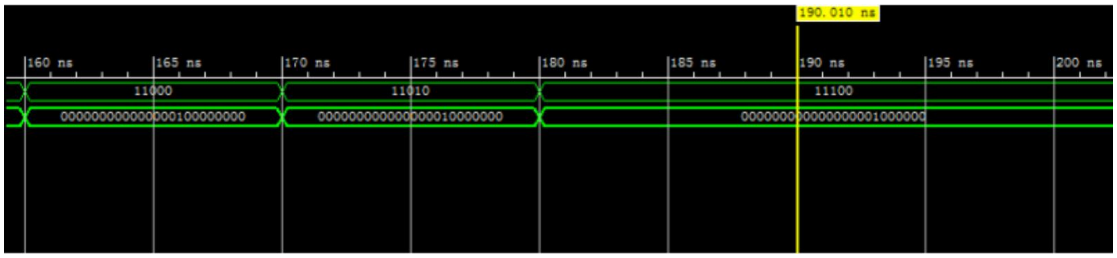
STA



MVI JZ



JMP IN OUT



MOV ADD SUB