

第三章 计算机体系结构实验

3.1. 微程序版 CPU 实验

3.1.1 实验概述

本实验的主要内容是掌握基于微程序控制器的 CPU 组成结构，了解 CPU 的中断工作机制。熟悉 CPU 微指令设计，掌握机器指令的微程序实现方法。

本实验将设计一个微程序版 CPU，包括微程序控制器、运算器、存储器、寄存器堆及外部 IO 接口。定义一套较完备的机器指令集，编写每条机器指令对应的微程序，在 CPU 电路上运行基于上述机器指令集的机器语言程序，并且用汇编助记符（语言）加以注释。

3.1.2 CPU 指令集

本书设计的微程序、硬连线和流水线版本 CPU 采用统一的 CPU 指令集，指令的 OP 码如下表 3-1 所示，其格式定义如下：指令 OP 码为指令第一字节的高四位，即指令寄存器 IR 的{I₇I₆I₅I₄}位。而 RA 和 RB 是指由 I₃I₂ 和 I₁I₀ 定义的逻辑寄存器，RA 或 RB 都可以选择四个物理寄存器（R0~R3）中的任何一个。

表 3-1 微程序版 CPU 指令集(OP 码表)

OP 码(I ₇ I ₆ I ₅ I ₄)	指令助记符	OP 码(I ₇ I ₆ I ₅ I ₄)	指令助记符
0111	IRET	1111	OR/ORI
0110	MOV	1110	AND/ANDI
0101	OUT/OUTA	1101	ADD/ADDI
0100	IN	1100	SUB/SUBI
0011	SET	1011	XOR/XORI
0010	SOP (INC/DEC/NOT/THR)	1010	SHT (RLC/ LLC/ RRC/ LRC)
0001	JMP/JMPR/Jx/JxR	1001	STO/PUSH
0000	NOP/HLT	1000	LAD/POP

上述指令集总共有 38 条机器指令，可以分成以下五个大类：

一、系统指令：

系统及中断指令包括三条单字节指令：空指令（NOP）、停机指令（HLT）和中断返回指令（IRET）。其中，NOP 指令主要用于精准延时（微程序/硬连线版 CPU 延时 4 个 T，流水线版 CPU 延时 1 个 T）；而 HLT 指令用作程序末尾 CPU 停机或设置“断点”：程序自动运行到 HLT 指令时刻停机，可以观察当时 CPU 寄存器，运算器标志位等信息。IRET 指令用于在中断处理子程序末尾返回主程序（即 BP_PC 保存的地址弹回 PC，BP_PSW 保存的标志位信息弹回 PSW），因此，**不允许在主程序使用 IRET 指令**，否则将导致程序错误跳转。

【注：x 在指令格式说明中表示此处的二进制数值可任意为 0 或 1】

汇编语言	功能	I ₇ I ₆ I ₅ I ₄	I ₃ I ₂	I ₁ I ₀
NOP;	无操作（延时 4 个 T）	0000	0/0	x/0
HLT;	停机（断点）	0000	0/0	x/1
IRET;	中断返回：BP_PC→PC; BP_PSW→PSW	0111	0/0	x/x

二、寄存器及 I/O 操作指令：

寄存器操作指令包括单字节的寄存器间数据传送指令（MOV）和双字节的寄存器赋值指令（SET）。SET 指令的第二字节是赋予寄存器 RA 的立即数 IMM。

例：“0110 0001;”表示把 R1 的内容赋值 R0；“0011 0000; 0000 0101;”表示把“05H”赋予 R0。

汇编语言	功能	I ₇ I ₆ I ₅ I ₄	I ₃ I ₂	I ₁ I ₀
MOV RA, RB;	(RB)→RA	0110	RA	RB
SET RA, IMM;	IMM→RA	0011	RA	x/x
		IMM		

I/O 操作指令包括三条单字节指令：输入指令（IN）、输出指令（OUT）及地址选择指令（OUTA）。OUTA 指令的功能是把寄存器的内容作为地址输出到 IO 端口的地址选择电路，选择所要操作的外部设备。OUT 指令选定操作的外设后，CPU 可以执行两种操作指令：IN 指令把外设的数据输入寄存器 RA，OUT 指令则是把寄存器 RA 的内容输出给外设。

【注：IN 指令可以选择 I₁I₀指定的四个输入端 PORT0~3 中的一个；

而 OUT/OUTA 指令只能选择 I₀指定的两个输出端 PORT0~1 中的一个。】

例：“0100 0001;”表示把端口 1 的输入数据传送到 R0。

“0101 0001;”表示把 R0 的内容作为数据，输出到端口 1。

“0101 0011;”则是表示把 R0 的内容作为地址，输出到端口 1。

汇编语言	功能	I ₇ I ₆ I ₅ I ₄	I ₃ I ₂	I ₁ I ₀
IN RA, PORTx;	(PORTx)→RA	0100	RA	PORTx
OUT RA, PORTx;	(RA)→PORTx	0101	RA	0/PORTx
OUTA RA, PORTx;	(RA)→PORTx_addr	0101	RA	1/PORTx

三、存储器及堆栈操作指令：

存储器操作指令包括两条双字节指令：取数指令（LAD）和存数指令（STO）。LAD 指令把数据从地址 ADDR（指令第二字节）的存储器单元取出，存入逻辑寄存器 RA；而 STO 指令把逻辑寄存器 RA 的数据取出，存入地址 ADDR（指令第二字节）的存储器单元。

堆栈操作指令包括两条单字节指令：出栈指令（POP）和入栈指令（PUSH）。此处提到的“堆栈”是基于存储器 ROM/RAM 的“软堆栈”，其指针就是逻辑寄存器 RB。出栈和入栈指令即是把寄存器 RB 存放的内容作为存储器地址，把该地址单元的数据弹出到逻辑寄存器 RA（POP 指令）或把逻辑寄存器 RA 的内容弹入到该地址单元（PUSH 指令）。

【注：因为 LAD 和 POP 指令共用 OP 码，STO 和 PUSH 指令也共用 OP 码；所以共用 OP 码的指令间区别在于 I₁I₀指定的内容。LAD 和 STO 指令的 I₁I₀=00，故 POP 和 PUSH 指令的 I₁I₀≠00，即其指定的逻辑寄存器 RB（指针）不能选择 R0】

例：“1000 0000; 0000 0101;”表示把存储器地址[05H]存放的数据弹出到寄存器 R0。

“1000 0001;”表示把堆栈指针 R1 指向的地址[R1]存放的数据弹出到寄存器 R0。

汇编语言	功能	I ₇ I ₆ I ₅ I ₄	I ₃ I ₂	I ₁ I ₀
LAD RA, [ADDR];	[ADDR]→RA	1000	RA	0/0
		ADDR		
POP RA, [RB];	[RB]→RA	1000	RA	RB
STO RA, [ADDR];	(RA)→[ADDR]	1001	RA	0/0
		ADDR		
PUSH RA, [RB];	(RA)→[RB]	1001	RA	RB

四、跳转系列指令：

无条件跳转指令 **JMP/JMPR** 的功能是程序必须跳转到目标地址执行，而有条件跳转指令 **Jx/JxR** 的功能则是程序是否跳转需要条件判断：当运算器结果标志位 **CF**（溢出）、**ZF**（零）或 **SF**（符号位）为 1 时，程序跳转到目标地址执行；反之，标志位为 0 则程序不跳转，继续顺序执行。根据判断标志位的不同，共有 **JC**、**JZ** 和 **JS** 三个有条件跳转指令。

跳转系列指令共用 **OP** 码“0001”，指令的 **I₃I₂** 位规定是无条件跳转指令（**JMP/JMPR**）还是有条件跳转指令（**Jx/JxR**）中的一种；而指令的 **I₁I₀** 位则指定目标地址是来源于寄存器 **RB**（单字节 **JMPR/JxR** 指令）还是来源于地址 **ADDR** 的存储器单元（双字节 **JMP/Jx** 指令）。

【注：双字节跳转指令 **I₁I₀**=00，故单字节跳转指令 **I₁I₀** 定义的逻辑寄存器 **RB** 不能选择 **R0**】
例：“0001 0100; 0000 0101;”表示当 **CF**=1 时，程序跳转到第二字节指定的目标地址 05H。

“0001 1001;”表示当 **ZF**=1 时，寄存器 **R1** 存放数据作为目标地址，程序跳转到该地址。

汇编语言	功能	I₇ I₆ I₅ I₄	I₃ I₂	I₁ I₀
JMP ADDR;	ADDR→PC	0001	0/0	0/0
		ADDR		
JMPR RB;	(RB)→PC	0001	0/0	RB
JC ADDR;	IF CF=1, ADDR→PC	0001	0/1	0/0
		ADDR		
JCR RB;	IF CF=1, (RB)→PC	0001	0/1	RB
JZ ADDR;	IF ZF=1, ADDR→PC	0001	1/0	0/0
		ADDR		
JZR RB;	IF ZF=1, (RB)→PC	0001	1/0	RB
JS ADDR;	IF SF=1, ADDR→PC	0001	1/1	0/0
		ADDR		
JSR RB;	IF SF=1, (RB)→PC	0001	1/1	RB

五、算术逻辑运算指令：

单字节移位指令 **SHT** 可以把逻辑寄存器 **RA** 中存放的数据向左或向右移动一个位(bit)，移入的位是 0（逻辑移位）或是数据另一端的位（循环移位）。四种 **SHT** 指令共用 **OP** 码“1010”，指令的 **I₀** 位指定逻辑移位还是循环移位，而指令的 **I₁** 位则是指定移位的方向。

【注：此处“右移位”指的是寄存器输出 **Q₃Q₂Q₁Q₀** 往小端移动，而“左移位”指的是寄存器输出 **Q₃Q₂Q₁Q₀** 往大端移动，跟时序发生器 74LS194 的“右移”和“左移”的定义相反】

汇编语言	功能	I₇ I₆ I₅ I₄	I₃ I₂	I₁ I₀
RLC RA;	(RA)右逻辑移位	1010	RA	0/0
LLC RA;	(RA)左逻辑移位	1010	RA	1/0
RRC RA;	(RA)右循环移位	1010	RA	0/1
LRC RA;	(RA)左循环移位	1010	RA	1/1

单字节单操作数运算指令 **SOP** 可以把寄存器 **RA** 递增(**INC**)、递减(**DEC**)、取反(**NOT**)和直通(**THR**)。如下表所示，四个 **SOP** 指令共用 **OP** 码“0010”，由指令 **I₁I₀** 位指定具体功能。

【注：**THR** 指令一般用于根据某个寄存器的数据判断 **ZF** 和 **SF** 标志位，从而决定是否跳转】

汇编语言	功能	I₇ I₆ I₅ I₄	I₃ I₂	I₁ I₀
INC RA;	(RA)+1→RA	0010	RA	0/0
DEC RA;	(RA)-1→RA	0010	RA	0/1
NOT RA;	#(RA)→RA	0010	RA	1/0
THR RA;	(RA)→RA	0010	RA	1/1

双字节双操作数运算指令可以把两个操作数进行算术运算：加法（ADD）、减法（SUB），以及逻辑运算：与（AND）、或（OR）、异或（XOR）。指令的 I_1I_0 位指定两个操作数全部来自寄存器或是操作数分别来源于寄存器和立即数 IMM（指令第二字节）。前者是单字节指令（ADD、SUB、AND、OR、XOR），后者是双字节指令（ADDI、SUBI、ANDI、ORI、XORI）。

【注：双字节指令的 $I_1I_0=00$ ，故单字节指令 I_1I_0 定义的逻辑寄存器 RB 不能选择 R0。】

例：“1101 0000; 0000 0101;”表示加法运算“ $R0=(R0)+05H$ ”。

“1101 0001;”表示加法运算“ $R0=(R0)+(R1)$ ”。

汇编语言格式	功能	$I_7 I_6 I_5 I_4$	$I_3 I_2$	$I_1 I_0$
ADD RA, RB;	$(RA) + (RB) \rightarrow RA$	1101	RA	RB
ADDI RA, IMM;	$(RA) + IMM \rightarrow RA$	1101	RA	0/0
		IMM		
SUB RA, RB;	$(RA) - (RB) \rightarrow RA$	1100	RA	RB
SUBI RA, IMM;	$(RA) - IMM \rightarrow RA$	1100	RA	0/0
		IMM		
AND RA, RB;	$(RA) \wedge (RB) \rightarrow RA$	1110	RA	RB
ANDI RA, IMM;	$(RA) \wedge IMM \rightarrow RA$	1110	RA	0/0
		IMM		
OR RA, RB;	$(RA) \vee (RB) \rightarrow RA$	1111	RA	RB
ORI RA, IMM;	$(RA) \vee IMM \rightarrow RA$	1111	RA	0/0
		IMM		
XOR RA, RB;	$(RA) \oplus (RB) \rightarrow RA$	1011	RA	RB
XORI RA, IMM;	$(RA) \oplus IMM \rightarrow RA$	1011	RA	0/0
		IMM		

3.1.3 微程序版 CPU 架构

如下图 3-1 所示，本实验的微程序版 CPU 由微程序控制器通路(CONTROLLER)、时序电路(CLOCK)及数据通路组成。数据通路包括：程序存储器 ROM、数据存储器 RAM 及通用寄存器 $R0 \sim R3$ ；IO 接口；算术逻辑运算器(74LS181)及附带的移位寄存器(74LS194)；程序计数器(PC)、ALU 运算结果标志位寄存器(PSW)及其断点寄存器(BP_PC、BP_PSW)。数据通路的所有部件都共同挂在一条 8 位系统总线 BUS 上。

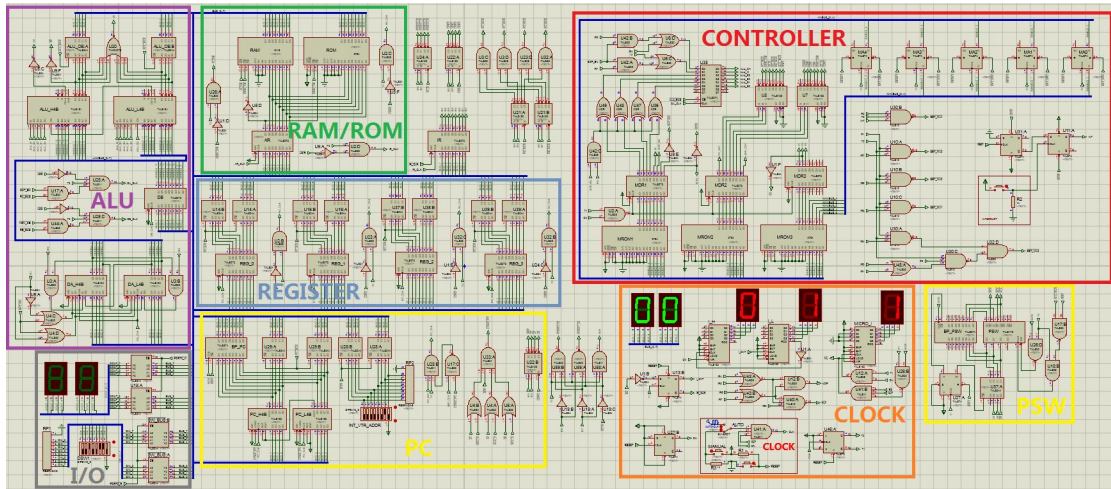
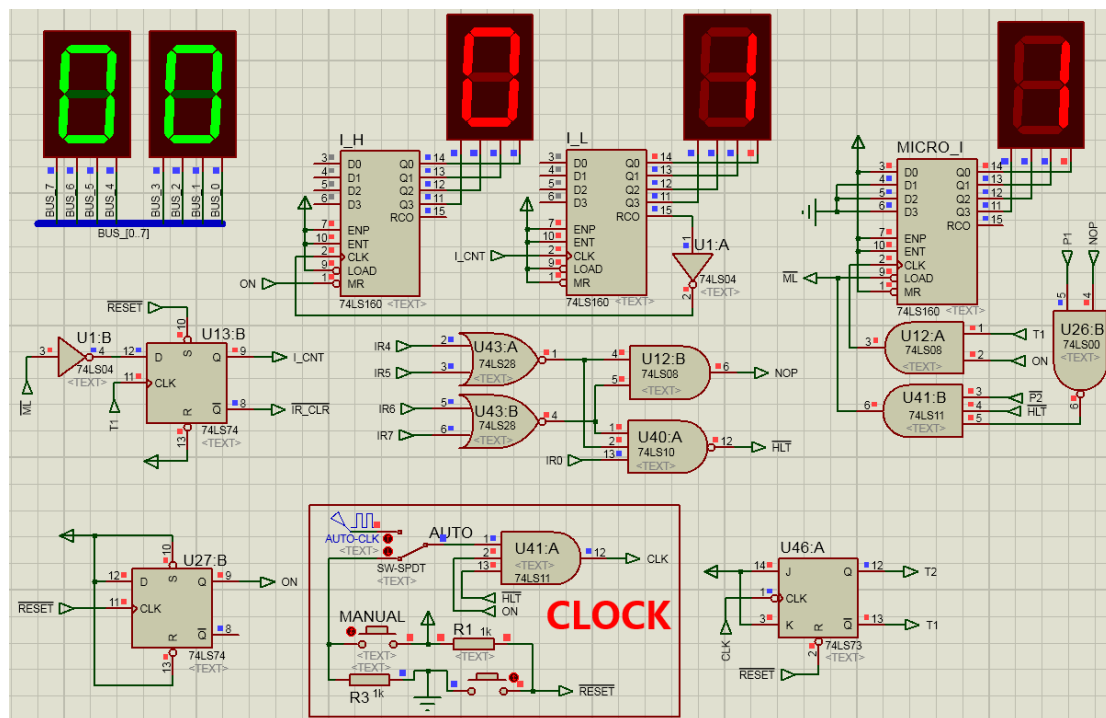


图 3-1 微程序版 CPU 电路图

[illegible]

微程序版 CPU 的时序电路如上图 3-3 所示：用红色边框围起来的 CLOCK 电路是 CPU 的基准时钟电路，系统时钟 CLK 可以由方波信号源 AUTO-CLK 提供（双击信号源可以自行选择方波信号频率）或者通过开关 MANUAL 手动步进。当初始化信号 ON=0 或停机指令信号 $\overline{HLT}=0$ ，时钟 CLK 阻塞（强制 CLK=0），CPU 停机。

CLOCK 电路右侧是一个 JK 触发器 74LS73 实现的微指令状态机。由于微程序控制器和数据通路相互独立，两者操作可以并行执行，如下表 3-2 所示。所以，微程序版 CPU 的微指令周期只需要 T1 和 T2 两个状态，时钟信号 CLK 驱动微指令状态机循环输出节拍序列 {T1,T2}，使状态顺序转移：T1→T2→T1→.....

表 3-2 微程序版 CPU 的微指令状态机

状态	微程序控制器通路	数据通路
T1	使能当前微指令的微操作信号有效	信息从源部件输出到总线 BUS
T2	微指令下址取址； 根据 OP 码决定微指令下址 [0I ₇ I ₆ I ₅ I ₄]（取指微指令）	信息从总线 BUS 打入目的部件； 程序计数器 PC+1（取指微指令）

CLOCK 电路左侧是初始化电路，手动按钮令复位信号 \overline{RESET} 上升沿跳变，可以使信号 ON=1。CPU 启动仿真后，初始化过程十分简单，如下所述：

- 1) 启动仿真后，时钟 CLK 选择从手动按钮 MANUAL 输入信号；
- 2) 手动按钮使信号 \overline{RESET} 跳变 “1→0→1”，令信号 ON=1，CLK 允许输出，过程结束。

CLOCK 电路上方是 NOP/HLT 指令电路：当指令寄存器 IR 的 OP 码 I₇I₆I₅I₄=0000 的时候，空指令信号 NOP=1，送往微指令计数器；OP 码 I₇I₆I₅I₄=0000 且 I₀=1 的时候，指令信号 $\overline{HLT}=0$ ，时钟 CLK 阻塞，CPU 停机（陷入“断点”）。跳出 HLT 指令“断点”的复位过程与上述初始化过程完全相同，区别在于初始化过程结束后，CPU 进入第一条指令的取指周期 T1 节拍；而复位过程结束后，CPU 进入 HLT 指令后续下一条指令的取指周期 T1 节拍。

如上图 3-3 所示，为了便于观测程序和微程序的运行，时序电路提供了双位的指令计数器 I 显示当前运行第几条机器指令，以及单位的微指令计数器 MICRO-I 显示当前运行指令计数器 I 所示指令中的第几条微指令。微指令计数器 MICRO-I 由十进制计数器 74LS160 构成，基于信号 ON（初始化过程）或 T1 节拍上升沿驱动递增，在指令周期末尾使能加载信号 $\overline{ML}=0$ ，在下一个指令周期开始时刻，重置 MICRO-I 的计数值为“1”，重新计数。当以下条件之一成立时，表示当前微指令是指令周期最后一条微指令，令 $\overline{ML}=0$ 。

- 1) 当前执行微指令中的判断位 P2=1（即 $\overline{P2}=0$ ）；（P 字段请参考“3.1.5 微程序控制器”）
- 2) 空指令信号 NOP=1 且判断位 P1=1（特殊情况：NOP 指令末尾）
- 3) 停机信号 $\overline{HLT}=0$ ；（特殊情况：HLT 指令末尾）

如上图 3-3 所示，当加载信号 $\overline{ML}=0$ ，下一个指令周期开始的 T1 节拍上升沿令指令计数信号 I_CNT=1，驱动指令计数器 I（由两个计数器 74LS160 级联构成）递增。与此同时，指令清除信号 $\overline{IR_CLR}=0$ ，即指令周期开始之际，清空指令寄存器 IR。

3.1.5 微程序控制器

微程序版 CPU 的微指令结构图如下图 3-4 所示，微指令字长 24 位，其中微指令的 1-5 位是下一条微指令地址，即下址字段[uA4, uA0]；微指令的 6-8 位是判断字段 P1~P3；微指

令的 9-24 位则是微命令字段，对应数据通路的所有微操作信号：其中置“1”的位表示执行相应的微操作；反之，置“0”的位则是不执行相应的微操作。

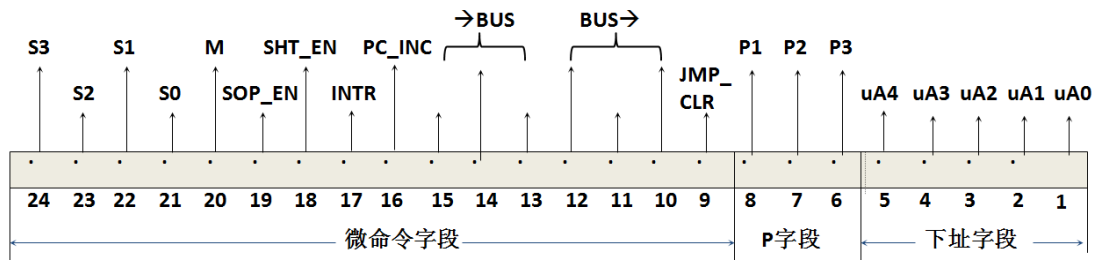


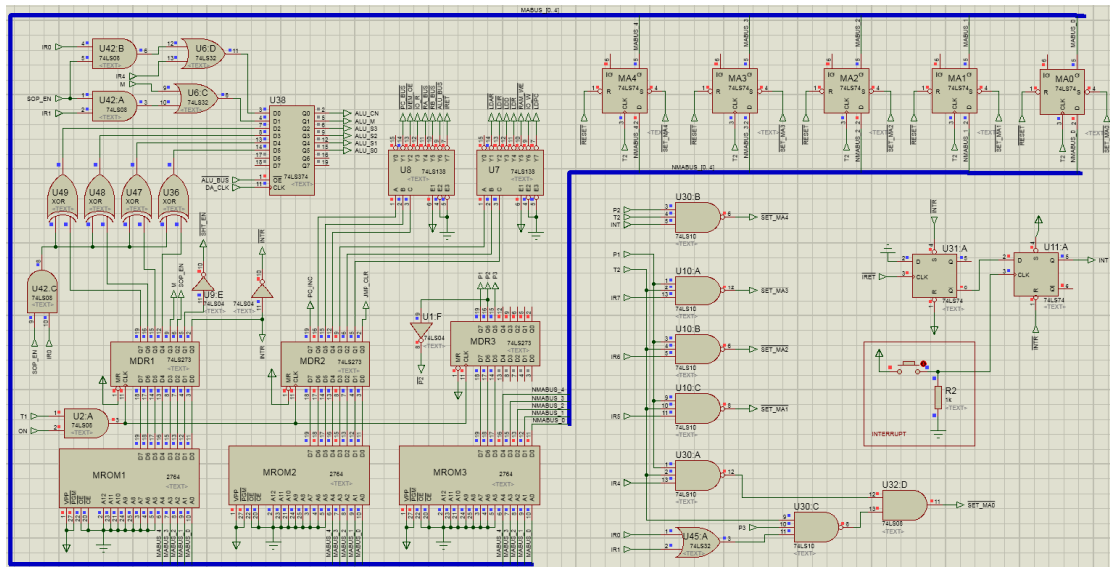
图 3-4 微程序版 CPU 的微指令结构图

此外，微指令的 15-13 位和 12-10 位采用字段编译法（3-8 译码），分别对应源部件输出到总线 BUS 和总线 BUS 打入目标部件的微操作信号，如下表 3-3 所示：

表 3-3 微指令字段编译列表

“→BUS” 字段				“BUS→” 字段			
15	14	13	微命令	12	11	10	微命令
0	0	0	\	0	0	0	\
1	0	0	PC_BUS	1	0	0	LDAR
0	1	0	MEM_OE	0	1	0	LDIR
1	1	0	IO_R	1	1	0	LDD
0	0	1	RA_BUS	0	0	1	LDR
1	0	1	RB_BUS	1	0	1	RAM_WE
0	1	1	ALU_BUS	0	1	1	IO_W
1	1	1	IRET	1	1	1	LDPC

基于上述微指令结构，本实验设计了如下图 3-5 所示的微程序控制器通路，包括三个 8 位 ROM 存储器 2764 组成的微指令存储器 MROM1~3，三个寄存器 74LS273 组成的微指令寄存器 MDR1~3，微指令译码电路，五位微地址寄存器 MA0~MA4 及微地址转移电路。



逻辑如下所示（可以对照后面具体指令流程图中“菱形框”的条件判断分支过程）：

P1 逻辑：若当前微指令是机器指令取指周期的最后一条微指令，则判断位 $P1=1$ ，从而根据指令寄存器 IR 的 $I_7I_6I_5I_4$ 位强制置位微地址寄存器的 $MA3-MA0$ ，修改微地址[$uA3, uA0$]位，转向该机器指令的执行周期序列的第一条微指令地址[$0I_7I_6I_5I_4$]。

P2 逻辑：若当前微指令是机器指令执行周期的最后一条微指令，则判断位 $P2=1$ ，此时若无中断发生，则返回取指周期第一条微指令地址[00000]；若有中断发生（ $INT=1$ ），则强制置位微地址寄存器的 $MA4$ ，转向中断处理过程第一条微指令地址[10000]。

P3 逻辑：在 CPU 指令集中部分单字节指令和双字节指令（LAD/POP、STO/PUSH、ALU 系列和 JMP 系列指令）共用 OP 码，其执行周期的微指令序列共用第一条微指令（判断位 $P3=1$ ），从第二条微指令开始分支，根据指令寄存器 IR 的 I_1I_0 位来决定不同微指令的分支走向：若 $I_1I_0=00$ ，微指令下址的 $MA0=0$ ，操作数分别来自寄存器和存储器（双字节指令）；若 $I_1I_0 \neq 00$ ，则微指令下址的 $MA0=1$ ，操作数全部来自寄存器（单字节指令）。

3.1.6 取指及中断处理过程

除了空指令（NOP）和停机指令（HLT）以外，所有的 CPU 指令都包括了取指周期和执行周期。因为 NOP 指令 OP 码为“0000”，所以取指周期末尾 $P1(0I_7I_6I_5I_4)$ 译码的时候，直接返回取指周期（取下一条指令），没有执行周期。而 HLT 指令与 NOP 指令完全相同，唯一不同是在取指周期后 CPU 硬件停机，需要手动 RESET“重启”才能跳出停机状态，进入下一条指令。此外，外部中断触发后，中断处理周期有专用的微指令使程序转向中断子程序。待到中断子程序末尾，最后一条指令必须是中断返回指令（IRET），才能返回主程序。

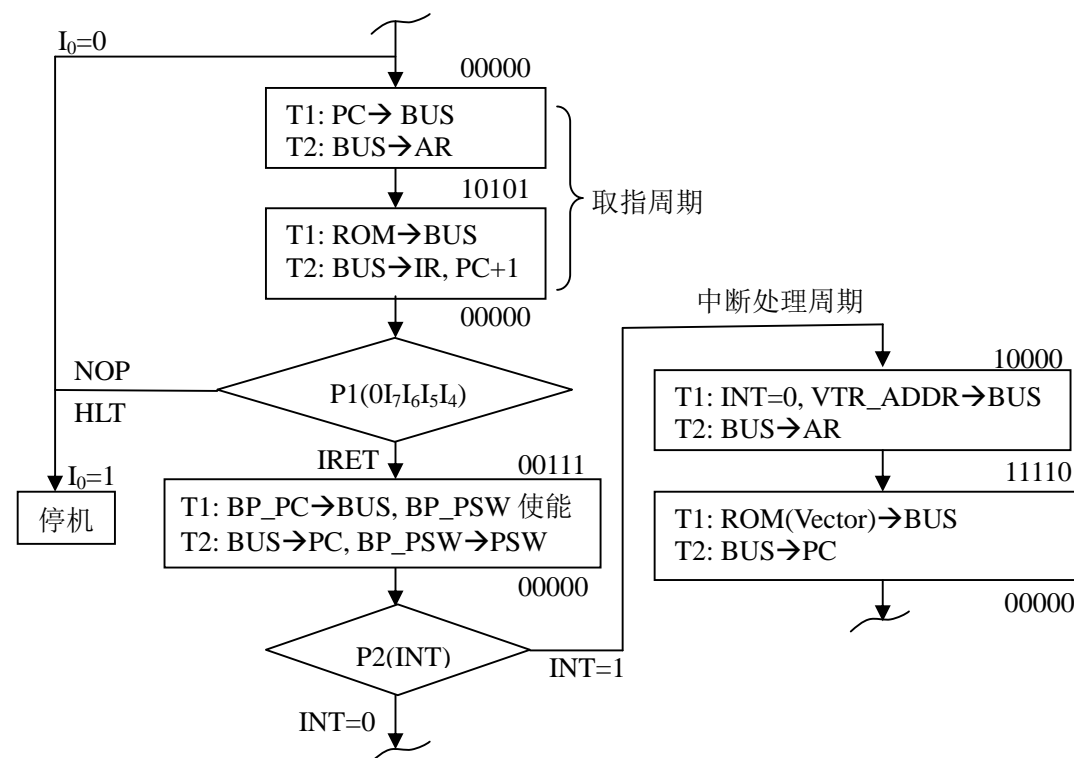


图 3-8 取指周期、中断处理周期及系统指令的微程序流程图

上图 3-8 所示是取指周期、中断处理周期及 NOP、HLT、IRET 指令的微程序流程图，

其中每个方框在时间上表示一个微指令周期，包括 T1 和 T2 两个节拍；在空间上表示数据从某个源部件经过总线 BUS 到达另一个目标部件的路径。每个方框的右上方是该微指令在控制存储器中的地址，右下方则是下一条微指令的地址。下表 3-4 则列出了上图 3-8 所对应的取指周期（即 NOP/HLT 指令）、中断处理周期及 IRET 指令的微指令代码：

表 3-4 微指令代码表（取指周期、中断处理周期及 IRET 指令）

Addr	S3	S2	S1	S0	M	SOP_EN	SHT_EN	INTR	PC_INC	→BUS	BUS→	JMP_CLR	P1	P2	P3	uA4	uA3	uA2	uA1	uA0
00000	0	0	0	0	0	0	0	0	0	100	100	0	0	0	0	1	0	1	0	1
10101	0	0	0	0	0	0	0	1	1	010	010	0	1	0	0	0	0	0	0	0
10000	0	0	0	0	0	0	0	1	0	000	100	0	0	0	0	1	1	1	1	0
11110	0	0	0	0	0	0	0	0	1	010	111	0	0	1	0	0	0	0	0	0
00111	0	0	0	0	0	0	0	0	1	111	111	0	0	1	0	0	0	0	0	0

本实验设计的存储器地址总线 8 位，地址空间 256 字节（00H~FFH）。分配其中低半区（00H~7FH）为 ROM 存储区(128 字节)，高半区（80H~FFH）为 RAM 存储区（128 字节），如下图 3-9 所示，存储器 ROM 和 RAM 共用一个地址寄存器 AR，两个存储器共用 $\overline{\text{MEM_OE}}$ 信号作为存储器读信号，由地址最高位 A7 来作为两个存储器的片选信号。RAM 存储器是可读写存储器，存放临时的数据。而 ROM 是只读存储器，存放程序和常量（采用堆栈操作指令访问）。因此，CPU 程序和常量的存储容量最大是 128 字节，若程序和常量的代码量超过了 128 字节，则会越界出错。同样的，因为只有存储器 RAM 允许写入，所以当存储器写信号 $\text{R_WE}=0$ 的时候，只有地址范围[80H,FFH]是允许写入操作的；对地址范围[00H,7FH]的存储器单元进行写入操作是非法的。值得注意的是，上述存储器不同地址范围的读写差异必须由通过软件（程序员或汇编器）来判别。

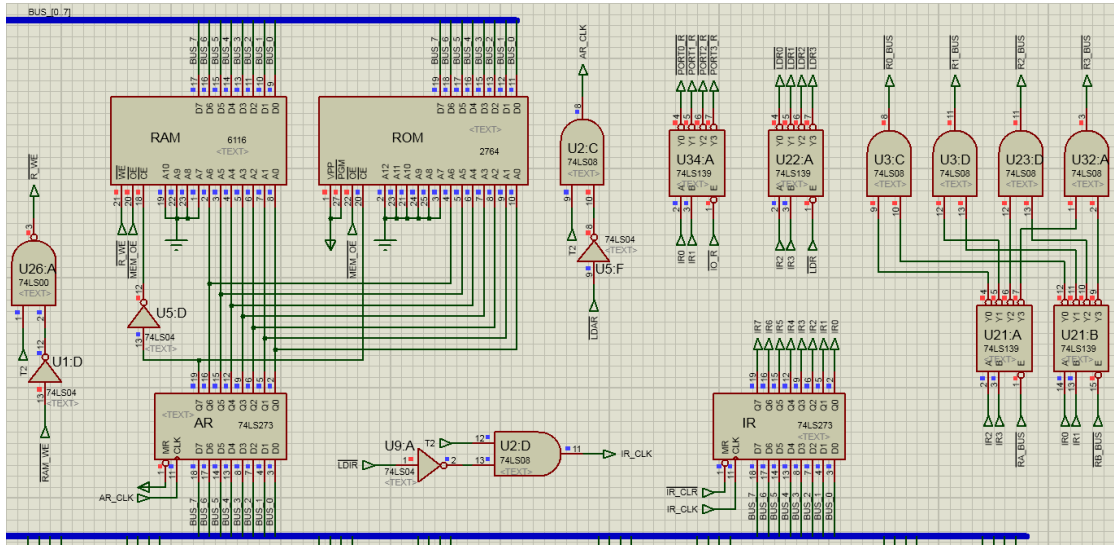


图 3-9 存储器（ROM/RAM）及指令寄存器 IR 电路

存储器 ROM（存放程序和常量）和 RAM（存放数据）共用地址寄存器 AR，而程序计数器 PC 和 AR 并联挂到总线 BUS。因此，如上图 3-9 和下图 3-10 所示，取指周期需要两条微指令（即两次路径）：第一条微指令[00000]的 T1 时刻，PC 输出当前指令地址到总线 BUS（ $\overline{\text{PC_BUS}}=0$ ），T2 时刻由 AR_CLK 上升沿打入存储器地址寄存器 AR；第二条微指令 T1 时刻，程序存储器 ROM 输出指令（ $\overline{\text{MEM_OE}}=0$ ）到总线 BUS，在 T2 时刻由 IR_CLK 上跳沿打入指令寄存器 IR；并且 PC+1（PC_CLK 上升沿）。

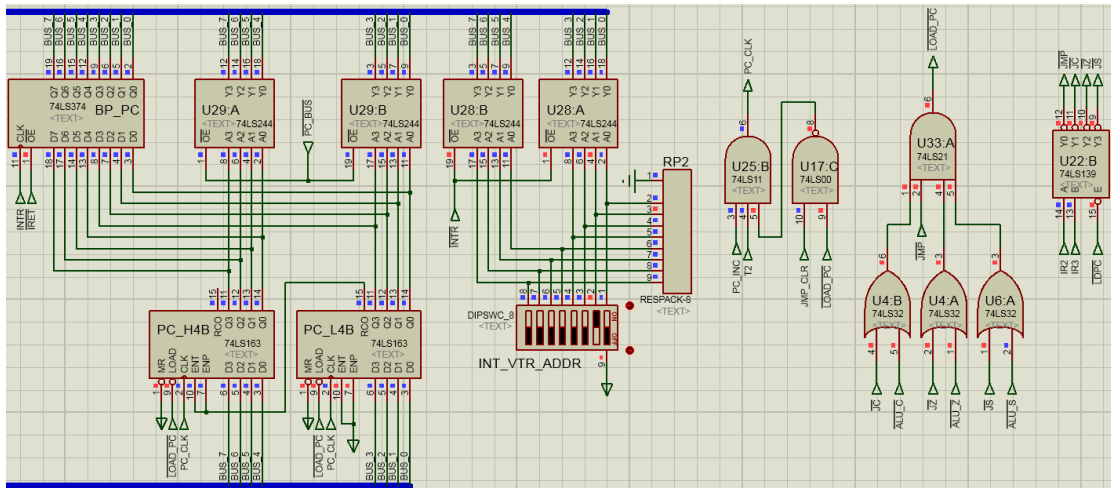


图 3-10 程序计数器 PC、断点 BP_PC 及中断向量地址电路

本实验的 CPU 中断电路采用单级中断机制，不允许中断嵌套；同时，CPU 采用中断向量的形式保存中断向量 **Vector**（即中断子程序入口地址）。如下图 3-11 所示，中断子程序的位置和长度随意设置，子程序的首地址（即中断向量 **Vector**）必须放在中断向量表中。中断发生时，CPU 通过二次寻址跳转到中断子程序执行。如上图 3-10 所示，进入中断处理周期的第一条微指令[10000]后，在 T1 时刻， $\text{INTR}=1$ 的上升沿跳变把 PC 的当前值保存到断点寄存器 BP_PC；同时， $\overline{\text{INTR}}=0$ 令拨码开关设置的中断向量地址 VTR_ADDR 输出到总线 BUS，并在 T2 时刻上升沿打入存储器地址寄存器 AR。在中断处理周期第二条微指令的 T1 时刻，存储器输出 **Vector** 到总线 BUS，并且在 T2 时刻打入 PC。

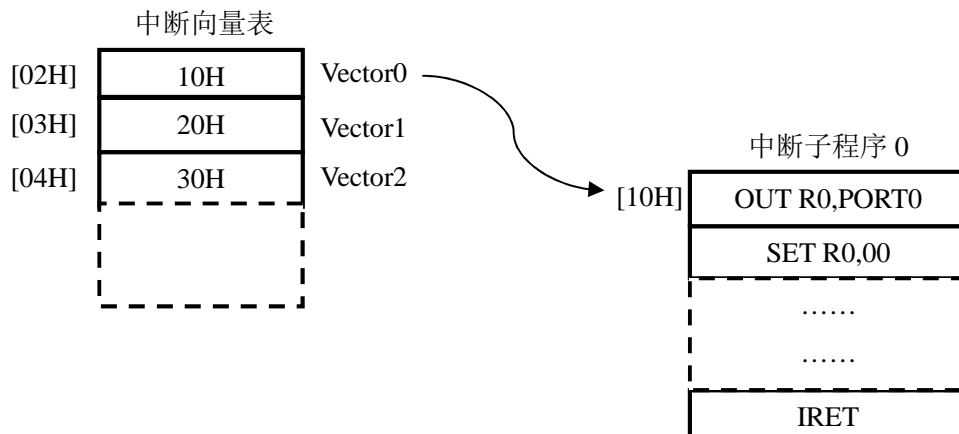


图 3-11 中断向量表示意图

如下图 3-12（右）所示，采用按钮来模拟 CPU 的外部中断，其按下的时候将产生一个上升沿跳变，输出 $\text{INT}=1$ 。所有指令的执行周期末尾必须执行 $\text{P2}(\text{INT})$ 判断：若 $\text{INT}=1$ ，表示在当前指令周期有中断触发，则 P2 置位 MA4=1，即微指令下址[uA4, uA0]=[10000]，进入中断处理周期；若 $\text{INT}=0$ ，表示没有中断，则返回取指周期（取下一条机器指令）。在中断处理周期的第一条微指令[10000]中， $\text{INTR}=1$ 清零 INT，并且把中断触发电路的输入（U31:A 反向输出端）锁死在低电平 0，即在中断子程序中不允许再次触发中断。如下图 3-12（左）所示， $\text{INTR}=1$ 上升沿跳变把标志位寄存器 PSW 的内容保存到 PSW 断点寄存器 BP_PSW 中，进而在 T2 时刻把寄存器 PSW 清零，即主程序标志位不影响中断子程序。

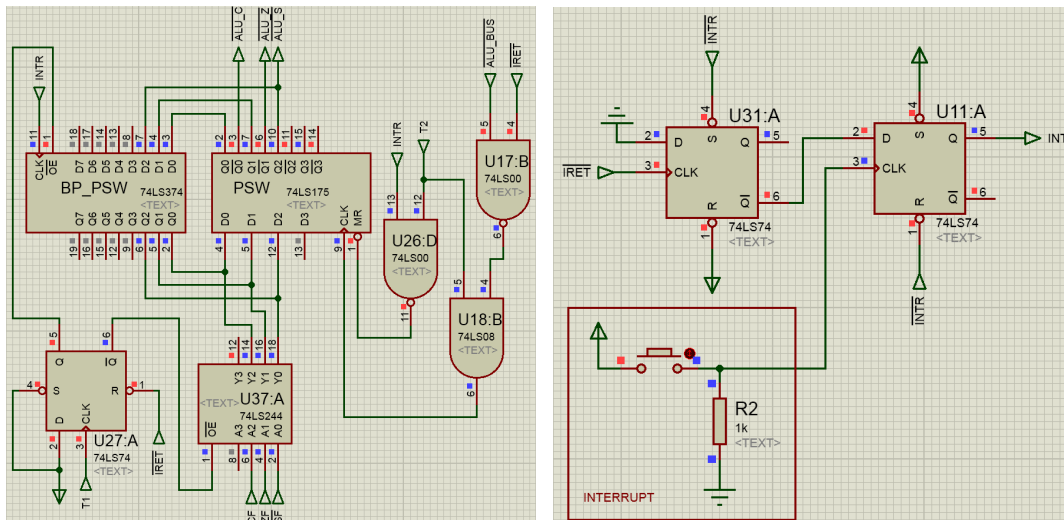


图 3-12 断点 BP_PSW 及中断触发电路

IRET 指令则是一个与中断处理周期相反的过程：如上图 3-10 所示， $\overline{\text{IRET}}=0$ 把断点寄存器 BP_PC 保存的“断点”输出到总线 BUS，然后在 T2 时刻打入 PC 中。类似的，如上图 3-12（左）所示，断点寄存器 BP_PSW 保存的“断点”输出，把数据通路的标志位输出缓冲器 U37:A 禁止，然后在 T2 时刻恢复到标志位寄存器 PSW 中（注：在主程序中是由运算结果输出信号 $\overline{\text{ALU_BUS}}$ 在 T2 时刻把标志位打入 PSW）。最后，如上图 3-12（右）所示，在 IRET 指令结束返回主程序后， $\overline{\text{IRET}}=1$ 产生的上升沿把中断触发电路输入（U31:A 的反向输出端）置为高电平 1，令中断触发电路恢复正常。

3.1.7 寄存器及 I/O 操作指令

寄存器操作指令包括一条单字节的寄存器间传送指令（MOV）和一条双字节的寄存器赋值指令（SET）；I/O 操作指令包括三条单字节指令：输入指令（IN）、数据输出指令（OUT）及地址输出指令（OUTA）。下图 3-13 是 MOV、SET 指令和 IN、OUT/OUTA 指令的微程序流程图，其中（P1 判断前）取指周期和若有中断触发的（P2 判断后）中断处理周期参见上图 3-8。此外，OUT 和 OUTA 指令的微指令序列完全相同，由硬件逻辑区分。

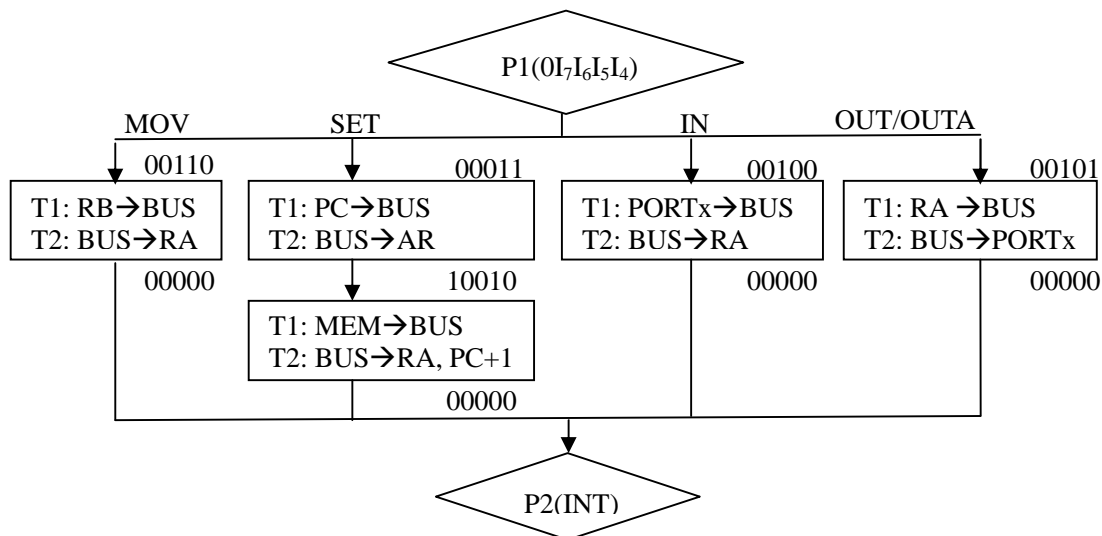


图 3-13 寄存器及 I/O 操作指令的微程序流程图

下表 3-5 列出了上图 3-13 对应的寄存器操作指令 MOV、SET 及 I/O 操作指令 IN、OUT/OUTA 的微指令代码。

表 3-5 微指令代码表（MOV/SET/IN/OUT 指令）

Addr	S3	S2	S1	S0	M	SOP_EN	SHT_EN	INTR	PC_INC	→BUS	BUS→	JMP_CLR	P1	P2	P3	uA4	uA3	uA2	uA1	uA0
00110	0	0	0	0	0	0	0	0	0	101	001	0	0	1	0	0	0	0	0	0
00011	0	0	0	0	0	0	0	0	0	100	100	0	0	0	0	1	0	0	1	0
10010	0	0	0	0	0	0	0	0	1	010	001	0	0	1	0	0	0	0	0	0
00100	0	0	0	0	0	0	0	0	0	110	001	0	0	1	0	0	0	0	0	0
00101	0	0	0	0	0	0	0	0	0	001	011	0	0	1	0	0	0	0	0	0

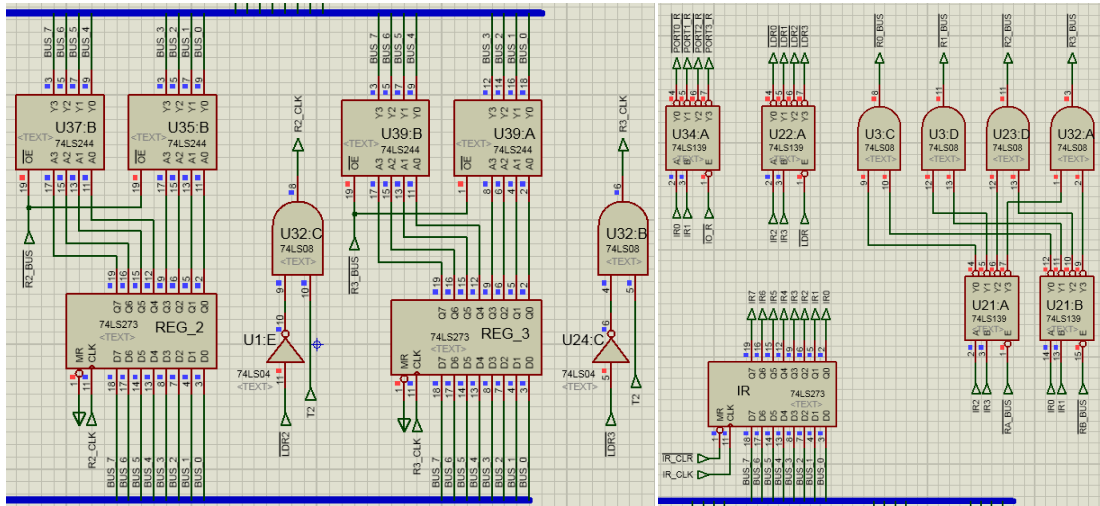


图 3-14 通用寄存器 R3 及寄存器选择电路图

如上图 3-14（左）所示，CPU 共有四个并行的通用寄存器 R0~R3，因为在上述 CPU 指令中，指令执行的操作数来源可能是逻辑寄存器 RA 或 RB 的输出，但是指令执行的结果只能是打入寄存器 RA。所以，在上图 3-14（右）中，微操作信号RA_BUS和RB_BUS分别根据指令的 I₃I₂ 位和 I₁I₀ 位选择通用寄存器（R0~R3）之一输出操作数（注：微操作信号RA_BUS和RB_BUS不允许出现在同一个微指令中，避免出现冲突）；而微操作信号LDR则直接根据指令的 I₃I₂ 位指定打入的通用寄存器。

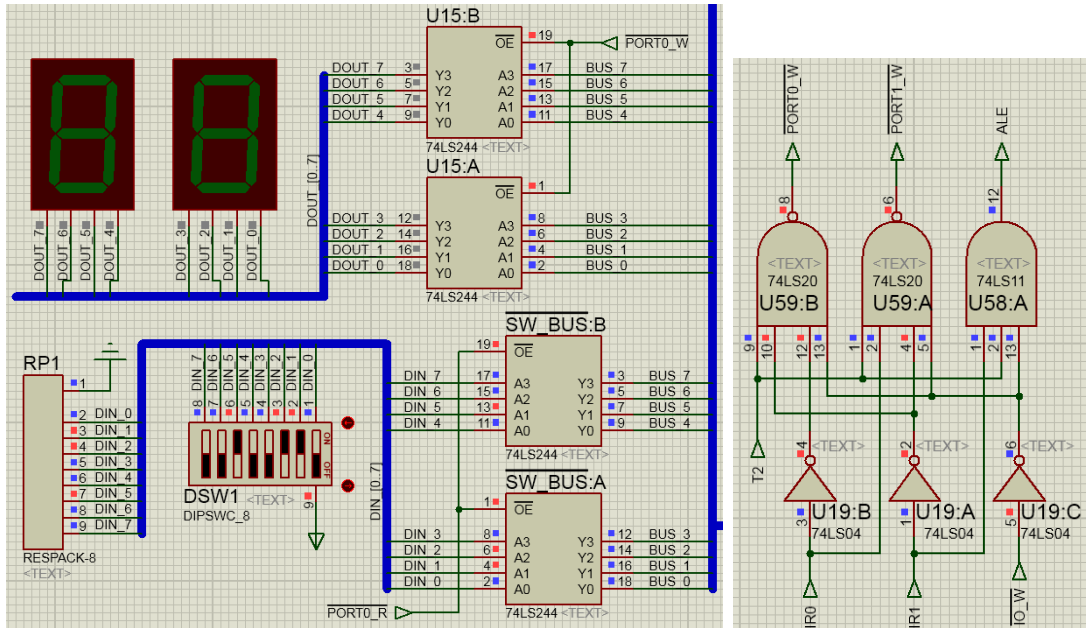


图 3-15 外围设备及 IO 接口电路图

上图 3-15 所示是 CPU 的 IO 接口外挂设备，采用拨码开关 DSW1 模拟输入设备，数码显示管模拟输出设备。若当前运行 IN 指令，则信号 $\overline{IO_R}=0$ ，外围设备输入数据到 BUS 总线。在上图 3-14（右）中，根据 IN 指令的 I_1I_0 位产生 IO 输入使能信号 $\overline{PORTx_R}$ ，可以指定 4 个输入设备，同样的，若当前运行 OUT/OUTA 指令，则信号 $\overline{IO_W}=0$ ，BUS 总线输出数据到 IO 接口外围设备，根据指令的 I_0 位产生 IO 输出使能信号 $\overline{PORTx_W}$ ，可以指定 2 个输出设备，而指令的 I_1 位作为地址锁存信号 ALE。若 ALE=1，则输出地址（OUTA 指令）；若 ALE=0，则输出数据（OUT 指令），如上图 3-15（右）所示。注意： $\overline{PORTx_W}$ 信号是打入目标部件的使能信号，时序须与其它打入信号保持一致，仅在 T2 周期有效。

3.1.8 存储器及堆栈操作指令

双字节存储器操作指令包括了取数指令 LAD 和存数指令 STO，而单字节堆栈操作指令包括了出栈指令 POP 和入栈指令 PUSH，下图 3-16 是存储器操作指令 LAD、STO 和堆栈操作指令 POP、PUSH 的微程序流程图。从图中可以看出，POP 指令只需要[11011]和[11101]两条微指令就够了，但是为了节省 OP 码，POP 和 LAD 指令共用 OP 码“1000”，即共用第一条微指令[01000]（即使 POP 指令其实并不需要微指令[01000]）。从而可以在第一条微指令的末尾采用 $P3(I_1I_0)$ 判断 LAD 和 POP 指令的不同路径：若 $I_1I_0=00$ ，执行直接根据第二字节目标地址 ADDR 从存储器取数的双字节 LAD 指令；若 $I_1I_0 \neq 00$ ，则执行根据逻辑寄存器 RB（R1~R3）内容指定的目标地址从存储器取数的单字节 POP 指令。STO 和 PUSH 指令的关系类似 LAD 和 POP 指令。

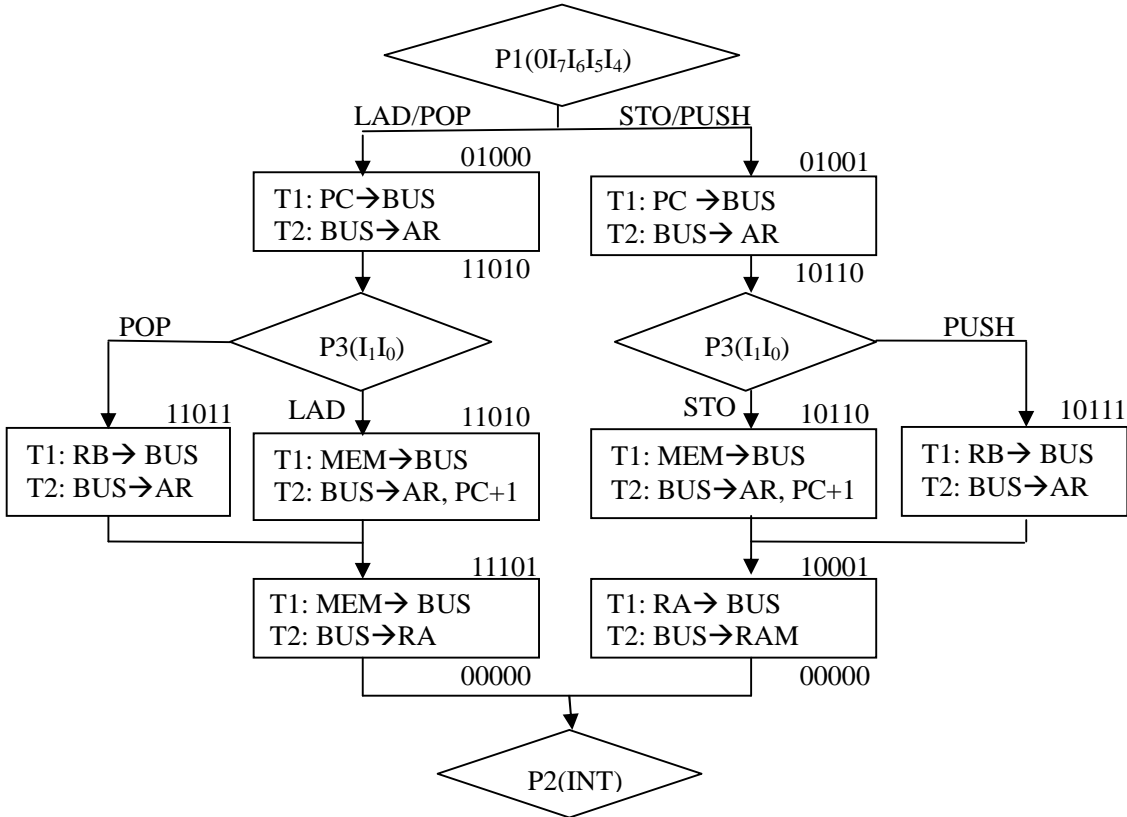


图 3-16 存储器及堆栈操作指令的微程序流程图

下表 3-6 列出的存储器指令 LAD/STO 和堆栈指令 POP/PUSH 的微指令代码：

表 3-6 微指令代码表（LAD/POP/STO/PUSH 指令）

Addr	S3	S2	S1	S0	M	SOP_EN	SHT_EN	INTR	PC_INC	→BUS	BUS→	JMP_CLR	P1	P2	P3	uA4	uA3	uA2	uA1	uA0
01000	0	0	0	0	0	0	0	0	0	100	100	0	0	0	1	1	1	0	1	0
11010	0	0	0	0	0	0	0	0	1	010	100	0	0	0	0	1	1	1	0	1
11011	0	0	0	0	0	0	0	0	0	101	100	0	0	0	0	1	1	1	0	1
11101	0	0	0	0	0	0	0	0	0	010	001	0	0	1	0	0	0	0	0	0
01001	0	0	0	0	0	0	0	0	0	100	100	0	0	0	1	1	0	1	1	0
10110	0	0	0	0	0	0	0	0	1	010	100	0	0	0	0	1	0	0	0	1
10111	0	0	0	0	0	0	0	0	0	101	100	0	0	0	0	1	0	0	0	1
10001	0	0	0	0	0	0	0	0	0	001	101	0	0	1	0	0	0	0	0	0

3.1.9 跳转系列指令

下图 3-17 是 JMPR/JxR 指令和 JMP/Jx 指令的微程序流程图。从图中可以看出，JMPR/JxR 指令只需要[11111]微指令就够了，但是为了节省 OP 码，两条跳转指令共用 OP 码“0001”，即共用第一条微指令[00001]（即使 JMPR/JxR 指令其实并不需要 [00001] 微指令）。从而可以在第一条微指令的末尾采用 P3(I₁I₀)区分两种跳转指令的不同路径：若 I₁I₀=00，执行直接根据第二字节目标地址 ADDR 跳转的双字节 JMP/Jx 指令；若 I₁I₀≠00，则执行根据逻辑寄存器 RB（R1~R3）内容指定的目标地址跳转的单字节 JMPR/JxR 指令。

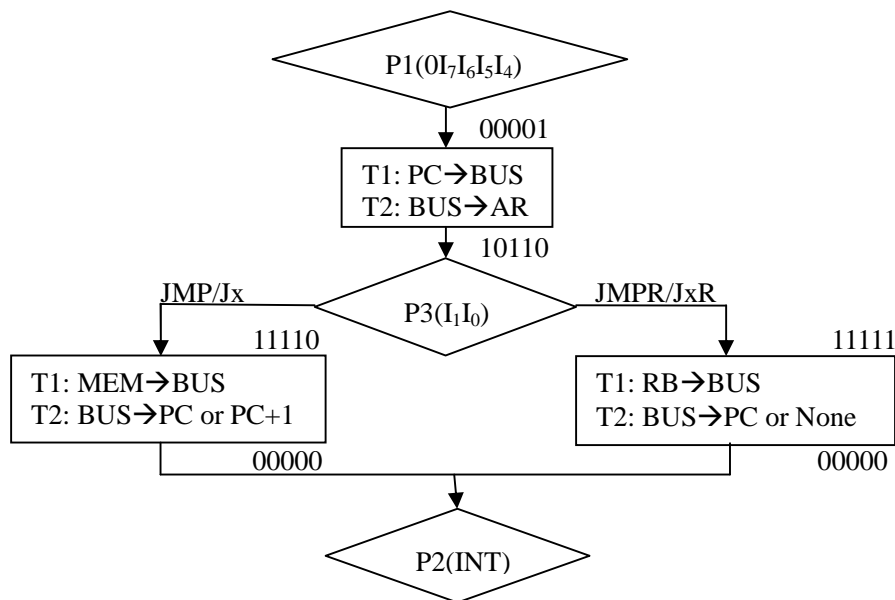


图 3-17 跳转系列指令的微程序流程图

下表 3-7 列出了单字节跳转指令 JMPR/JxR 和双字节跳转指令 JMP/Jx 的微指令代码。

表 3-7 微指令代码表（JMP/JMPR/Jx/JxR 指令）

Addr	S3	S2	S1	S0	M	SOP_EN	SHT_EN	INTR	PC_INC	→BUS	BUS→	JMP_CLR	P1	P2	P3	uA4	uA3	uA2	uA1	uA0
00001	0	0	0	0	0	0	0	0	0	100	100	0	0	0	1	1	1	1	1	0
11110	0	0	0	0	0	0	0	0	1	010	111	0	0	1	0	0	0	0	0	0
11111	0	0	0	0	0	0	0	0	1	101	111	1	0	1	0	0	0	0	0	0

下图 3-18（左）所示，跳转指令“0001”执行的时候，微操作信号 \overline{LDPC} 首先根据指令的 I₃I₂ 位译码，判断是执行无条件跳转指令（微操作信号 $\overline{JMP}=0$ ）还是有条件跳转指令 JC、JZ 和 JS（对应的微操作信号 Jx=0）。若执行有条件跳转指令，还需要根据标志位寄存器 PSW 保存的运算器标志位 CF/ZF/SF 来断定是否生成微操作信号 $\overline{LOAD_PC}$ ，使得跳转发生。

值得注意的是，在有条件跳转指令 JCR、JZR、JSR 的执行周期最后一条微指令[11111]处，倘若最后不跳转，则因为 JxR 是单字节指令。所以，此处不但需要不打入 PC，而且还

必须禁止 PC+1。因此，只有在地址[11111]的微指令执行的时候，使用微操作信号 $\overline{\text{JMP_CLR}}$ 和 $\overline{\text{LOAD_PC}}$ 的逻辑“与”来决定是否运行 $\text{PC_CLK}=1$ （即 PC+1 操作）。

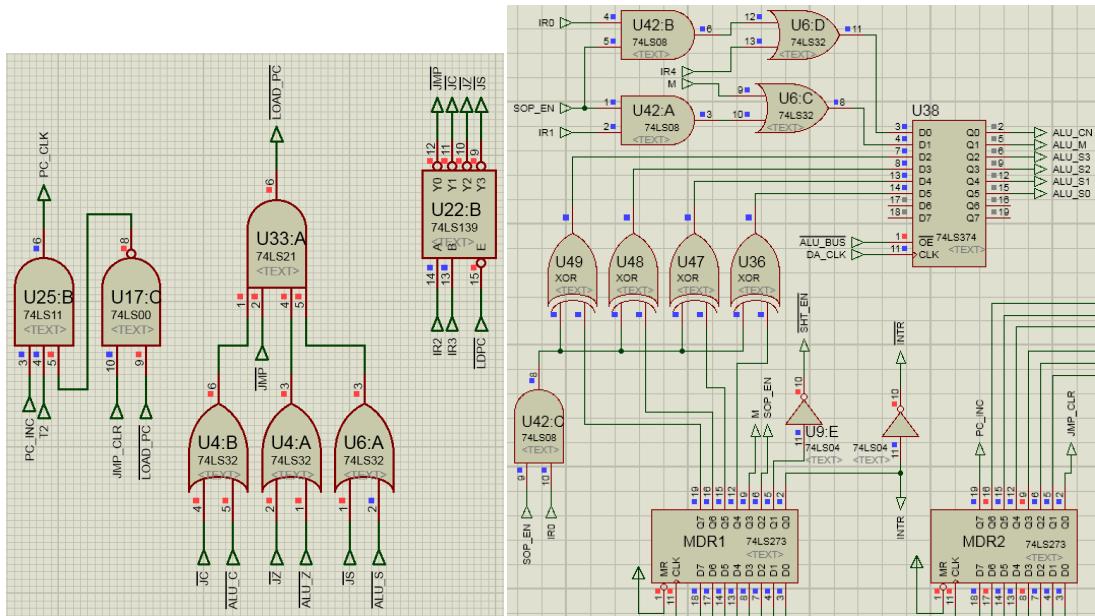


图 3-18 跳转系列指令和运算指令的硬件译码逻辑电路图

3.1.10 算术逻辑运算系列指令

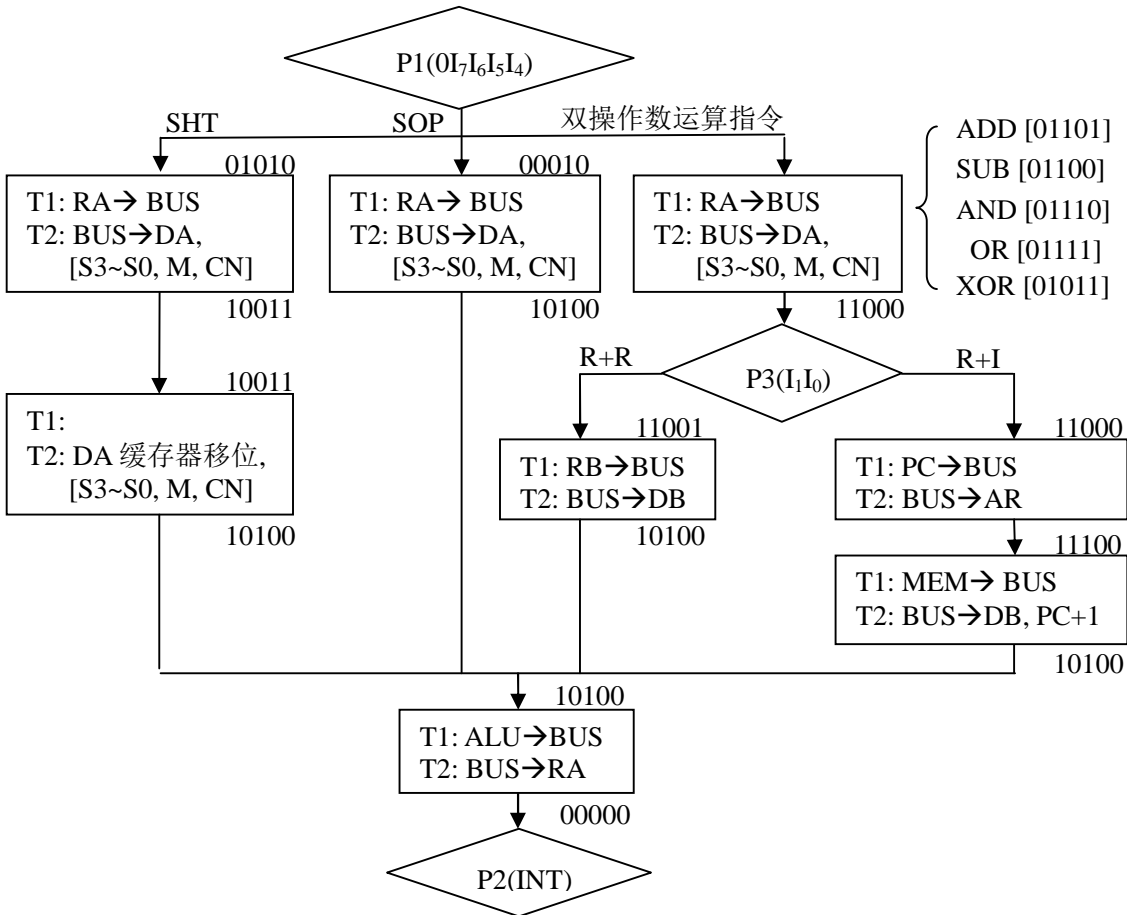


图 3-19 算术逻辑运算系列指令的微程序流程图

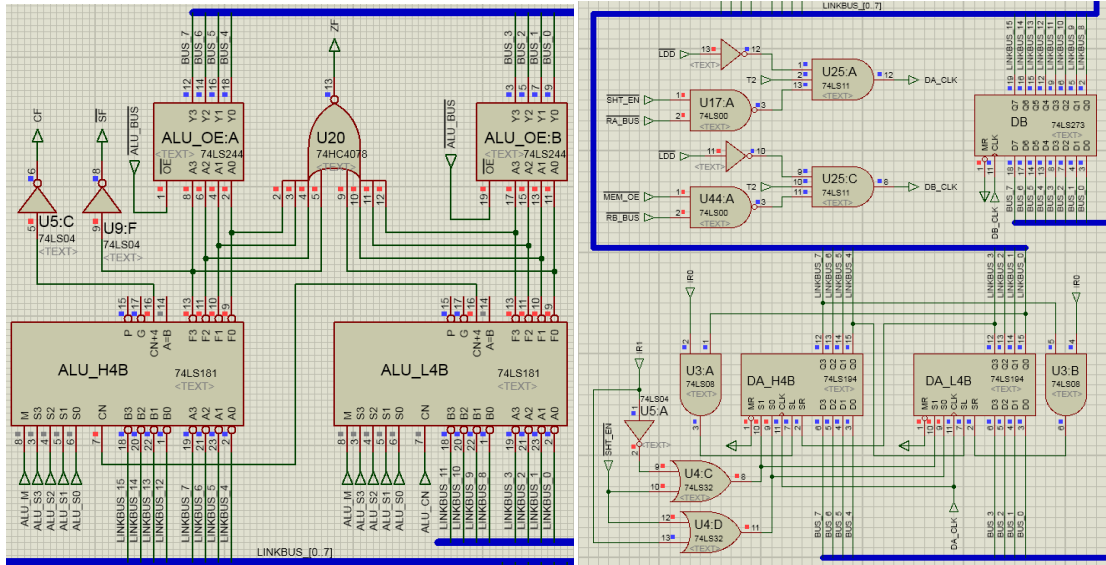
算术逻辑运算系列指令包括了单字节的移位指令 SHT，单字节单操作数运算指令 SOP，以及五条单字节的双操作数运算指令（ADD/SUB/AND/OR/XOR）和五条双字节的双操作数运算指令（ADDI/SUBI/ANDI/ORI/XORI），其微程序流程图如上图 3-19 所示。为了节省微指令，所有的运算指令都在第一条微指令期间锁存 74181 运算器的控制端逻辑 [S3,S2,S1,S0,M,CN]。此外，五种双操作数的运算指令都采取在第一条微指令的末尾采用 P3(I₁I₀)判断双字节和单字节指令的不同路径：若 I₁I₀=00，执行操作数分别来自逻辑寄存器 RA 和指令第二字节（立即数 IMM）的双字节指令；若 I₁I₀≠00，则执行操作数全部来自寄存器的单字节指令。上述算术逻辑运算指令的微指令代码表如下表 3-8 所示：

表 3-8 微指令代码表（SHT、SOP 及双操作数运算指令）

Addr	S3	S2	S1	S0	M	SOP_EN	SHT_EN	INTR	PC_INC	→BUS	BUS→	JMP_CLR	P1	P2	P3	uA4	uA3	uA2	uA1	uA0
01010	1	1	1	1	1	0	0	0	0	001	110	0	0	0	0	1	0	0	1	1
10011	1	1	1	1	1	0	1	0	0	000	110	0	0	0	0	1	0	1	0	0
00010	0	0	0	0	0	1	0	0	0	001	110	0	0	0	0	1	0	1	0	0
01101	1	0	0	1	0	0	0	0	0	001	110	0	0	0	1	1	1	0	0	0
01100	0	1	1	0	0	0	0	0	0	001	110	0	0	0	1	1	1	0	0	0
01110	1	0	1	1	1	0	0	0	0	001	110	0	0	0	1	1	1	0	0	0
01111	1	1	1	0	1	0	0	0	0	001	110	0	0	0	1	1	1	0	0	0
01011	0	1	1	0	1	0	0	0	0	001	110	0	0	0	1	1	1	0	0	0
11000	0	0	0	0	0	0	0	0	0	100	100	0	0	0	0	1	1	1	0	0
11100	0	0	0	0	0	0	0	0	1	010	110	0	0	0	0	1	0	1	0	0
11001	0	0	0	0	0	0	0	0	0	101	110	0	0	0	0	1	0	1	0	0
10100	0	0	0	0	0	0	0	0	0	011	001	0	0	1	0	0	0	0	0	0

值得注意的是，单操作数运算指令 SOP 只有一条微指令[00010]。必须由上图 3-18（右）所示的硬件逻辑电路根据 SOP 指令的 I₁I₀ 位修改运算器控制端[S3,S2,S1,S0,M,CN]，实现递增(INC)、递减(DEC)、取反(NOT)、直通(THR)四个功能。同时，OP 码的 I₄ 位则用来指定双操作数算术运算指令 ADD 和 SUB 的 CN 操作信号，如下所示：

CPU 指令		OP 码 (I ₇ I ₆ I ₅ I ₄)	S3 S2 S1 S0	M	CN	I ₁	I ₀	I ₄
SOP	INC	0010	0000	0	0	0	0	0
	DEC		1111	0	1	0	1	0
	NOT		0000	1	x	1	0	0
	THR		1111	1	x	1	1	0
ADD/ADDI		1101	1001	0	1	X	X	1
SUB/SUBI		1100	0110	0	0	X	X	0



上图 3-20（左）所示运算器 ALU 通路，运算器 74LS181 除了输出结果到总线 BUS，还输出运算结果的标志位 CF（溢出）、ZF（零）、SF（符号位）到标志位寄存器 PSW 保存。

上图 3-20（右）所示是运算器 ALU 的缓存器 DA 和 DB。其中 DB 采用寄存器 74LS273，而 DA 则采用移位寄存器 74LS194，兼有缓存和移位功能。当微操作信号 $\overline{\text{SHT_EN}}=1$ 的时候（非 SHT 指令），74LS194 的状态 $\{S_0, S_1\}=\{1, 1\}$ ，工作模式强制为送数，DA_CLK 上升沿跳变把 74LS194 输入端 $D_3D_2D_1D_0$ 保存到输出端 $Q_3Q_2Q_1Q_0$ ；当 $\overline{\text{SHT_EN}}=0$ 的时候（SHT 指令），移位寄存器 74LS194 的状态 $\{S_0, S_1\}$ 由指令 SHT 的 I_1 位决定。若 $I_1=0$ ， $\{S_0, S_1\}=\{0, 1\}$ ，则寄存器输出端 $Q_3Q_2Q_1Q_0$ 往 Q_0 端移动，即右移；若 $I_1=1$ ， $\{S_0, S_1\}=\{1, 0\}$ ，则寄存器输出端 $Q_3Q_2Q_1Q_0$ 往 Q_3 端移动，即左移。而 SHT 指令的 I_0 位则决定是逻辑移位还是循环移位：若 $I_0=0$ ，则 74LS194 的输入端 $S_L=S_R=0$ ，即逻辑移位；若 $I_0=1$ ，则 74LS194 的输入端 S_L 和 S_R 分别接 74LS194 的另一端，即循环移位。

缓存寄存器 DA 和 DB 的打入微操作信号 DA_CLK 和 DB_CLK 分别在三种情况下触发：

- 1) 双操作数运算指令：操作数全部来源于逻辑寄存器 RA 和 RB，DA_CLK 和 DB_CLK 分别由微操作信号 $\overline{\text{RA_BUS}}$ 和 $\overline{\text{RB_BUS}}$ 驱动；
- 2) 双操作数运算指令：操作数之一来自指令第二字节（立即数 IMM），则 DA_CLK 由微操作信号 $\overline{\text{RA_BUS}}$ 驱动，而微指令 $[11100]$ 期间，DB_CLK 则由微操作信号 $\overline{\text{MEM_OE}}$ 驱动；
- 3) 移位指令 SHT：在微指令 $[10011]$ 期间，DA 缓存器 74LS194 移位需要再次 DA_CLK 上升沿跳变才能实现，则 DA_CLK 由微操作信号 $\overline{\text{SHT_EN}}$ 驱动。

3.1.11 实验步骤

实验 1 顺序结构程序

● 在微程序版 CPU 项目工程的子文件夹 PROGRAMS 里，存放着全部机器指令的示例源程序（asm 文件）。除了 JS、SOP_JZ 和 INT_IRET 三个源程序外，其他源程序都是顺序结构的程序，例如以下 ADD 指令的示例程序 ADD.asm：


```
ORG    0000H
        DB    00110000B ; SET R0, 03H
        DB    00000011B ;
        DB    00110100B ; SET R1, 30H
        DB    00110000B ;

        DB    00111000B ; SET R2, F0H
        DB    11110000B ;
        DB    11010001B ; ADD R0,R1
        DB    11011001B ; ADD R2,R1

        DB    00000001B ; HLT
END
```

● 编译 ADD.asm 源程序，生成 HEX 文件烧写到存储器 ROM（编译和烧写 asm 文件的方法参见“2.3.3 ROM 批量导入数据的技巧”）。

● 启动仿真后，时钟 CLK 选择从手动开关 MANUAL 输入信号；手动按钮使复位信号 $\overline{\text{RESET}}$ 跳变“1→0→1”，令信号 ON=1，CLK 允许输出，初始化过程完成。

- 若手动执行程序，则直接手动 MANUAL 按钮，令时钟 CLK 输出“”信号，程序单步执行。对照 ADD 指令流程图及其微指令代码表，观察每次手动单步执行结果，记录寄存器 AR、IR、PC、通用寄存器 Rx 及总线 BUS 上的数据变化。

- 若自动运行程序，则把时钟 CLK 改接 AUTO-CLK 信号源（主频 10Hz），程序自动运行，直到 HLT 指令“断点”处暂停。暂停后，可以通过信号 CLK 改接手动按钮 MANUAL，然后手动按钮令复位信号 $\overline{\text{RESET}}$ 输出“1→0→1”变化，即可跳出“断点”，进入 HLT 指令后续下一条指令，然后时钟 CLK 再接 AUTO-CLK 信号源，程序即继续自动运行。

- 在程序自动运行过程中，可以采用 HLT 指令在程序需要调试的位置设置“断点”。观察“断点”暂停时刻，寄存器 AR、IR、PC、通用寄存器 Rx 及总线 BUS 上的数据（**注意：增加 HLT 指令“断点”会出现跳转指令的目标地址偏移问题。**）。

- 参照上述过程，编译、烧写、手动或自动运行文件夹 PROGRAMS 里其余机器指令（JS 和 SOP_JZ 除外）的示例程序。对照相应的指令流程图及其微指令代码表，观察自动运行或每次单步执行结果，记录寄存器 AR、IR、PC、PSW 及总线 BUS 上的数据变化。

实验 2 分支结构程序

- 条件跳转指令验证程序 JS 是典型的分支结构程序，其功能类似于汇编语言的“CMP”语句，实现了比较寄存器 R0 和 R1 所存数据的大小，并且输出较大的数据到 IO 端口外挂设备（数码显示管），具体代码如下所示：

```
ORG    0000H
        DB    00110000B ; SET R0, 04H
        DB    00000100B ;
        DB    00110100B ; SET R1, 03H
        DB    00000011B ;

        DB    00000001B ; HLT
        DB    11000001B ; SUB R0, R1
        DB    00011100B ; JS 0CH
        DB    00001100B ;

        DB    11010001B ; ADD R0, R1
        DB    01010000B ; OUT R0, PORT0
        DB    00010000B ; JMP 0DH
        DB    00001101B ;

        DB    01010100B ; OUT R1, PORT0
        DB    00000001B ; HLT
END
```

- 参照上述实验 1 的操作，编译、烧写、自动运行 JS 源程序。观察程序自动运行过程中两个“断点”的暂停时刻，寄存器 R0 和 R1 的数据变化。

- 修改上述 JS 源程序，把寄存器 R0 和 R1 保存的数据对调，程序运行过程和结果有什么不同？记录运行过程中寄存器 R0 和 R1 的数据变化，观察 IO 接口的数码管显示。

- 请问本程序中的 ADD 指令起什么作用？如果要求比较的过程不能改动 R0 和 R1 的

值，那 JS 源程序需要如何修改？

● 编译、执行如下所示的源程序 ADD0_SUB0.asm。试问 $0+0=0$ 且 $0-0=0$ ，为何两个运算后执行 JC 的结果不一致（一个跳转，另一个不跳转）？

```
ORG    0000H
      DB    00110000B ;SET R0,0
      DB    00000000B
      DB    00110100B ;SET R1,0
      DB    00000000B

      DB    11010001B ;ADD R0,R1
      DB    00010100B ;JC 0CH
      DB    00001100B ;
      DB    01010000B ;OUT R0,PORT0

      DB    11000001B ;SUB R0,R1
      DB    00010100B ;JC 0CH
      DB    00001100B ;
      DB    01010000B ;OUT R0,PORT0

      DB    00000001B ;HLT
END
```

实验 3：循环结构程序

● 单操作数运算指令验证程序 SOP_JZ 是典型的循环结构程序，其功能类似于汇编语言的“LOOP”语句，实现了“ $1+2+\dots+9+10$ ”的连续十次相加求和。具体代码如下所示：

```
ORG    0000H
      DB    00110000B ;SET R0,01H
      DB    00000001B ;
      DB    00110100B ;SET R1,02H
      DB    00000010B ;

      DB    00111000B ;SET R2,09H
      DB    00001001B ;
      DB    00000001B ;HLT
      DB    11010001B ;ADD R0,R1

      DB    00101001B ;DEC R2
      DB    00100100B ;INC R1
      DB    00101011B ;THR R2
      DB    00011000B ;JZ 0FH

      DB    00001111B ;
      DB    00010000B ;JMP 07H
      DB    00000111B ;
      DB    01010000B ;OUT R0,PORT0
```

```

        DB      00000001B ;HLT
END

```

● 参照上述实验 1 的操作，编译、烧写、自动运行 SOP_JZ 程序。观察自动运行过程中的“断点”暂停时刻，寄存器 R0、R1 和 R2 的数据变化。

● 请问 R0 和 R1 总共循环相加了几次？为何统计次数的 R2=09？最后 R0 输出的结果是多少？“THR R2”指令执行的意义是什么？能否只使用两个通用寄存器完成连续相加求和的任务？如果可以，程序要如何修改？

实验 4：中断程序

● INT_IRET 是基于中断向量二次跳转实现的单级中断程序，主程序功能是寄存器 R0 的数值累加，而中断子程序则是显示中断时刻 R0 数值并且清零。具体代码如下所示：

```

ORG      0000H
        DB      00010000B ; JMP 08H
        DB      00001000B ;
        DB      00000011B ; vector [02]=03      ;中断向量地址 02，中断向量 03
        DB      00000001B ; HLT                  ;sub 中断子程序入口

        DB      01010000B ; OUT R0, PORT0
        DB      00110000B ; SET R0, 0
        DB      00000000B ;
        DB      01110000B ; IRET

        DB      00110000B ; SET R0, 02H          ;main 主程序
        DB      00000010B ;
        DB      11010000B ; ADDI R0, 02H
        DB      00000010B ;

        DB      00010000B ; JMP 0AH
        DB      00001010B ;
        DB      00000001B ; HLT
END

```

● 参照上述实验 1 操作，编译、烧写、自动运行中断程序 INT_IRET，随机触发 INTERRUPT 按钮（模拟外部中断），观察 R0 的变化。

● 在主程序中设置的 HLT“断点”暂停时刻，信号 CLK 改用手动单步执行，触发 INTERRUPT 按钮，模拟外部中断，观测和记录中断处理过程中，寄存器 PC、BP_PC、PSW、BP_PSW 及总线 BUS 的数据变化。

● 在本实验中，中断出现会令寄存器 R0 清零，改变主程序的参数。因为中断是随机发生的，不确定中断发生时刻主程序运行的位置。所以，应该尽量使中断子程序和主程序的参数（主要是寄存器）互相独立。请问在寄存器资源有限情况下，可以采用什么方法实现？

3.1.12 思考题

1、中断返回指令 IRET 只能在中断子程序出现，请设计一个硬件电路的保护机制：若在主程序中出现 IRET 指令，则 CPU 不执行打入 PC 的操作，避免系统崩溃。

2、微程序版 CPU 的取指周期和中断处理周期目前尚需要两个 CPU 周期（两条微指令），可否修改硬件电路和微指令列表，只用一个 CPU 周期实现上述功能？

（提示：拆分独立的数据存储器 ROM/RAM 和程序存储器 PROGRAM，数据存储器的地址寄存器仍为 AR，程序计数器 PC 则作为程序存储器的地址寄存器，不再直连到总线 BUS。注意，因为取指周期只有一个 CPU 周期，需要谨慎考虑取指周期末尾 PC+1 的问题，以及取指周期开头把 IR 寄存器输出的 OP 码清零，从而避免影响微地址 P1 跳转的问题。）

3、在思考题 2 基础上，可否利用节省出来的空闲微指令地址安排新的微指令或增加指令功能？例如使 OUT/OUTA 指令既可以输出通用寄存器 Rx 内容（单字节指令），也可以输出立即数 IMM（双字节指令）。从而在 I/O 端口的操作中减少对寄存器资源的占用。