山东大学
SHANDONG UNIVERSITY

Shandong University

School of Computer Science and Technology

计算机组成原理课程设计

Author:
刘建东 Jiandong Liu
Student Number & Class:
201700130011 & Elite Class

Supervisor:
韩芳溪教授 Prof. Fangxi Han
Teaching Assistant:
潘云刚 Yungang Pan
李双良 Shuangliang Li

The Experimental Report of The Composition and Design of Computer

2020 年 7 月 3 日

摘　要

本课程设计一共分为五个阶段，分别是架构熟悉、设计指令系统、设计总体结构与数据通路、微程序实现控制器、硬布线实现控制器，其中架构熟悉分为两个实验，分别是微程序控制的运算器设计与微程序控制的存储器读写系统设计。

指令系统一共包含 16 类指令，根据寻址方式以及寄存器的不同，一共扩展至 29 个指令。并且为了支持条件语句、循环语句、递归语句，指令集中引入了条件跳转指令 BEQ 与 BNE、无条件跳转指令 JR、压/弹栈指令 PUSH 与 POP。

总体结构主要包含一个 ALU、两个通用寄存器、指令寄存器、地址寄存器、栈指针寄存器、两个可读写存储器、控制器，其中两个可读写存储器 RAM、SRAM 分别用于存放指令数据与充当递归栈。

微程序实现控制器需要将每一个指令拆分为对应微操作，并对各个微操作进行编码，以此实现其对于受控门状态的控制。另外，硬布线实现控制器则在微操作的基础上，利用组合逻辑电路的方式，求出每一个受控门所对应的组合逻辑电路以此来实现对受控门的控制。

最终模型机系统将依据指令系统、总体结构与数据通路、控制器这三部分进行设计，实现基于本系统指令集代码的程序运行。

关键词：模型机设计，指令系统，总体结构与数据通路，控制器，微程序，硬布线

Abstract

The design of this course is divided into five stages, which are architecture familiarity, design instruction system, design overall structure and data path, micro-program implementation controller, and hard-wired implementation controller. Among them, architecture familiarization is divided into two experiments, namely micro Program-controlled arithmetic unit design and micro-program-controlled memory read-write system design.

The instruction system contains a total of 16 types of instructions. According to the different addressing modes and registers, a total of 29 instructions are expanded. In order to support conditional statements, loop statements, and recursive statements, the instruction set introduces conditional jump instructions BEQ and BNE, unconditional jump instructions JR, and stack instructions PUSH and POP.

The overall structure mainly includes an ALU, two general registers, instruction registers, address registers, stack pointer registers, two readable and writable memories, and a controller, of which two readable and writable memories RAM and SRAM are used to store instruction data and act as Recursive stack.

The micro-program realization controller needs to split each instruction into corresponding micro-operations and encode each micro-operation to achieve its control of the state of the controlled gate. In addition, the hard-wired implementation controller uses the combinational logic circuit to find the combinational logic circuit corresponding to each controlled gate based on micro-operations to control the controlled gate.

The final model machine system will be designed according to the three parts of the instruction system, the overall structure and the data path, and the controller to implement the program operation based on the instruction set code of the system.

Key Words: Model machine design, instruction system, overall structure and data path, controller, microprogram, hard wiring

目录

1 课程设计概述

1.1 设计目的

- 运用计算机原理所学知识，理论联系实际，设计一台模型机
  - 确定数据通路
  - 确定机器指令的格式，并定义相应的指令系统
  - 分别采用微程序与硬布线方式
- 巩固课堂知识
- 深化学习内容

1.2 设计要求

- 课程设计的时间为 32 学时
- 独立完成设计任务，充分发挥个人特长，设计出具有一定特色的模型计算机
- 设计工具
  - 计算机组成原理课程设计平台：康芯 KX-CDS 平台
  - QuartusII 软件的使用

1.3 设计过程

- 第一阶段：两个分解实验
  - 实验一：微程序控制的运算器设计
  - 实验二：微程序控制的存储器读写系统设计
- 第二阶段：模型机内核设计
  - 实验三：微程序实现的模型机内核
  - 实验四：硬布线实现的模型机内核

1.4 设计步骤

完整设计流程一共分为五个步骤，分别是指令系统、总体结构和数据通路、逻辑设计、CU 设计、调试。其中 CU 设计有两种方法，分别是微程序与硬布线，具体流程图如图1所示。

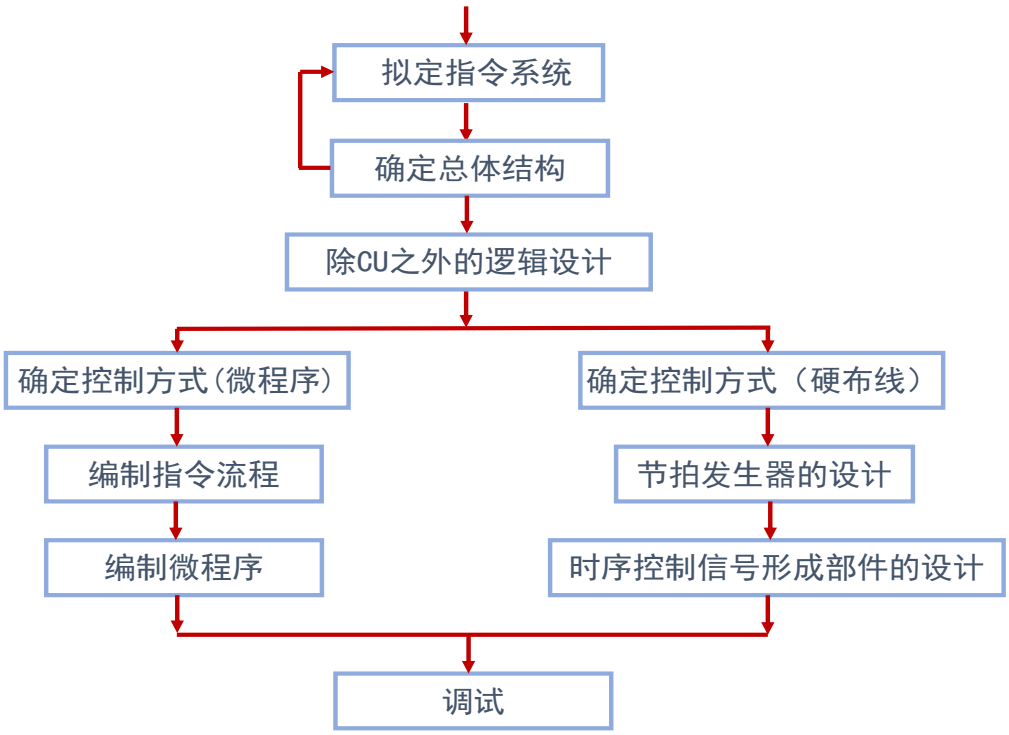

图 1: 模型机设计步骤

1.4.1 指令系统

指令系统主要涉及指令字长、指令格式、指令的编码、指令种类、寻址方式。

1.4.2 总体结构和数据通路

- 总体结构设计包含确定各部件设置以及它们之间的数据通路结构。
- 数据通路不同，执行指令所需要的操作就不同，计算机的结构也就不一样。

1.4.3 逻辑设计

总体结构确定之后，便开始总体结构中各部件的逻辑设计并连接部件。

1.4.4 微程序实现控制器

- **确定微程序控制方式：**需要确定微程序控制器的结构、下地址形成方式、微程序控制器的时序、微指令格式。其中微指令格式包括微指令字段定义、微命令形成逻辑、后继微地址产生逻辑。
- **编制指令执行流程：**根据指令的复杂程度，确定每条指令所需要的机器周期数，对于微程序控制的计算机，根据总线结构，需考虑哪些微操作可以安排在同一条微指

令中，哪些微操作不能安排在同一条微指令中。

- **微指令代码化：**根据后续微地址的形成方法，确定每个微程序地址及分支转移地址。
- **调试：**用单步微指令方式执行机器指令的微程序流程图。

1.4.5 硬布线实现控制器

- 所有指令执行步骤划分和功能确定
- 节拍发生器的设计
- 根据指令执行流程和数据通路，设计各控制信号的列表
- 根据列表，设计各控制信号的逻辑表达式
- 设计完整电路，下载
- 调试

2 微程序控制的运算器设计

2.1 设计概述

2.1.1 设计目的

- 熟悉简单运算器的结构
- 熟悉微命令的产生和时序
- 熟悉运算器功能测试

2.1.2 设计简述

- 设计一个八位算法逻辑运算单元 ALU
- 两操作数由两个八位寄存器 R_0、R_1 提供，其结果放入 R_2 中。具体执行的操作可由微命令任意设定。

2.2 架构设计

在正式开始设计 CPU 之前，我们需要设计两个架构来熟悉一下设计环境以及某些功能部件的设计逻辑。这两个架构分别对应着三、四部分的内容，即：

- 微程序控制的运算器设计
- 微程序控制的存储器读写系统设计

本部分内容主要探讨第一个架构的设计方案，如图3所示。

由于本部分内容主要是熟悉功能部件的设计，因此我们暂时用图2所示的微指令格式进行设计。

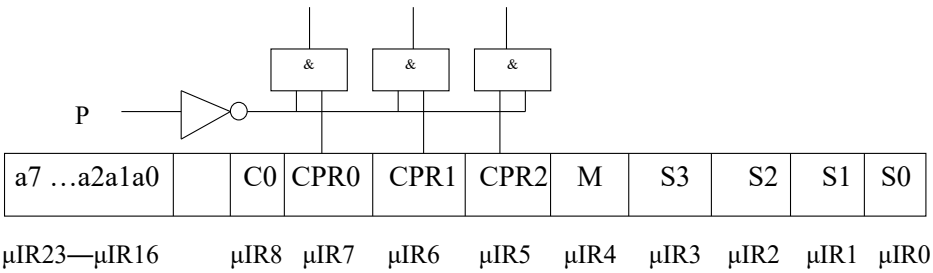

图 2: 微程序控制的运算器微指令格式

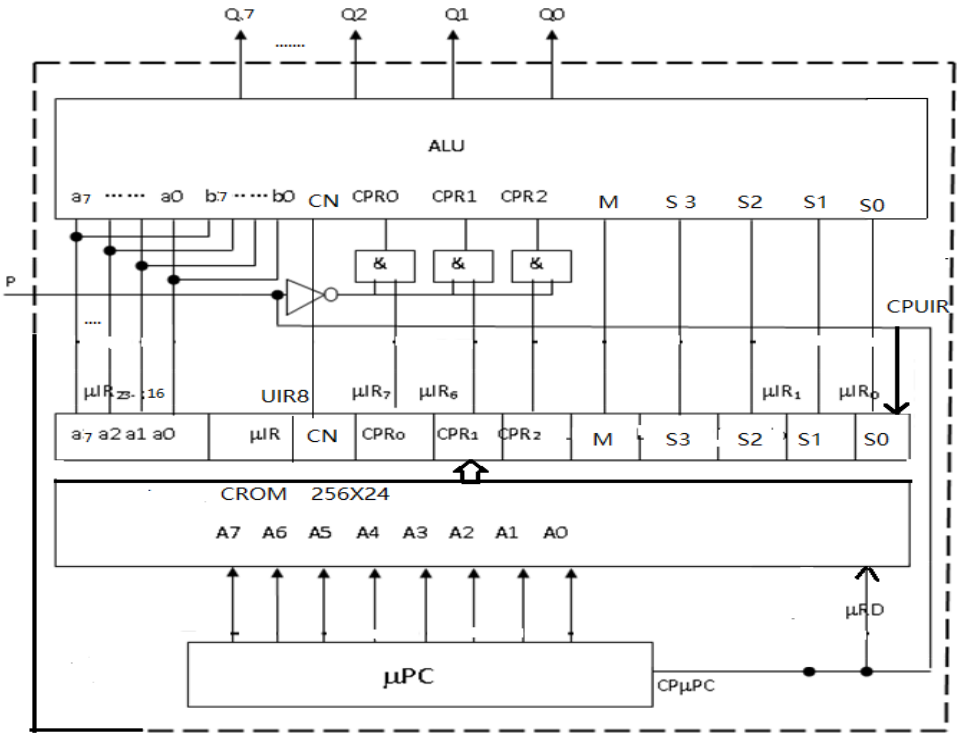

图 3: 微程序控制的运算器架构设计

根据上述架构，可以总结其大体逻辑如下：

1. 指令预先存在 CROM 中
2. uPC 从 CROM 中取出指令
3. CROM 将指令传给 CPUIR，即指令寄存器中
4. 指令寄存器再将具体的指令内容对应传给 ALU
5. ALU 执行指令，输出结果

弄明白设计逻辑之后，我们将元件一一分离进行实现。不难发现，一共有 4 个元件，列举如下：

- uPC：计数器
- CROM：存储器
- CPUIR：寄存器
- ALU：运算器

2.3 各元件设计

2.3.1 计数器 uPC

由于架构设计中 CROM 大小为 256*24，因此我们需要设计 8 位的计数器，可用两个 74161 元件组合而成。

该元件设计并不复杂，具体设计内容如图4所示。

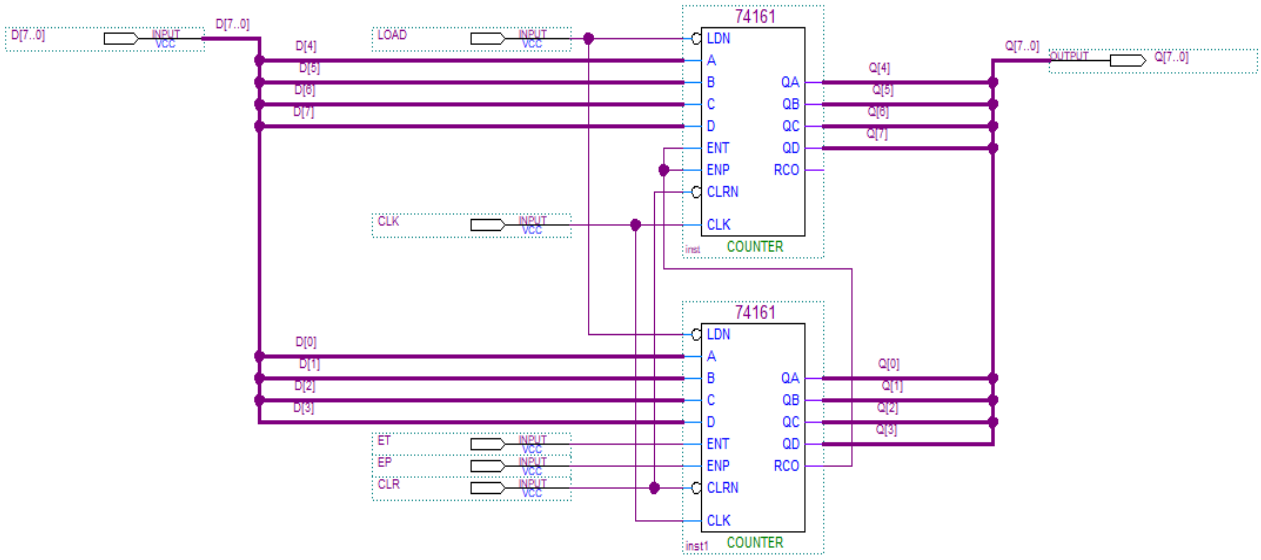

图 4: 微程序控制的运算器架构中计数器设计

设计完该元件后，我们将其封装，封装后的元件如图5所示。

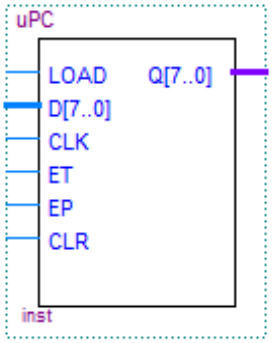

图 5: 微程序控制的运算器架构中计数器封装

2.3.2 存储器 CROM

点击图6中的按钮，根据流程即可设计 1-port 的 ROM，该元件的设计为 Quartus 应用的操作，按指示来即可。

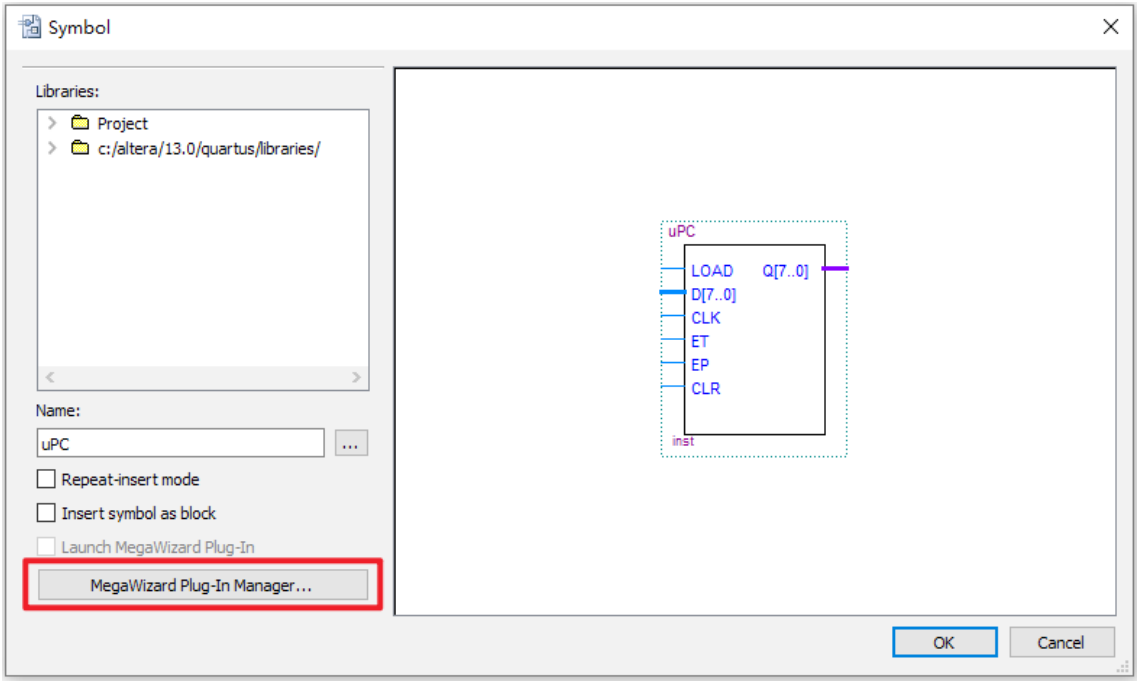

图 6: 微程序控制的运算器架构中存储器设计

最后的设计成果如图7所示。

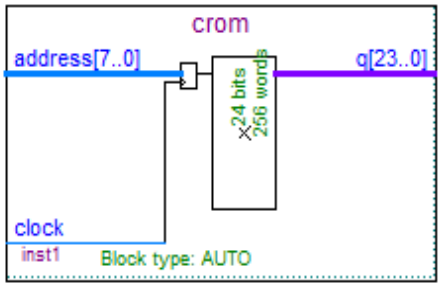

图 7: 微程序控制的运算器架构中存储器封装

2.3.3 寄存器 CPUIR

CPUIR 是 24 位寄存器，但 QuartusII 中提供的 74273 元件为 8 位寄存器，因此我们需要将 3 个 74272 元件并行连接成我们所需要的 CPUIR，具体电路结构如图8所示。

设计完后，我们将其封装，封装后的元件如图9所示。

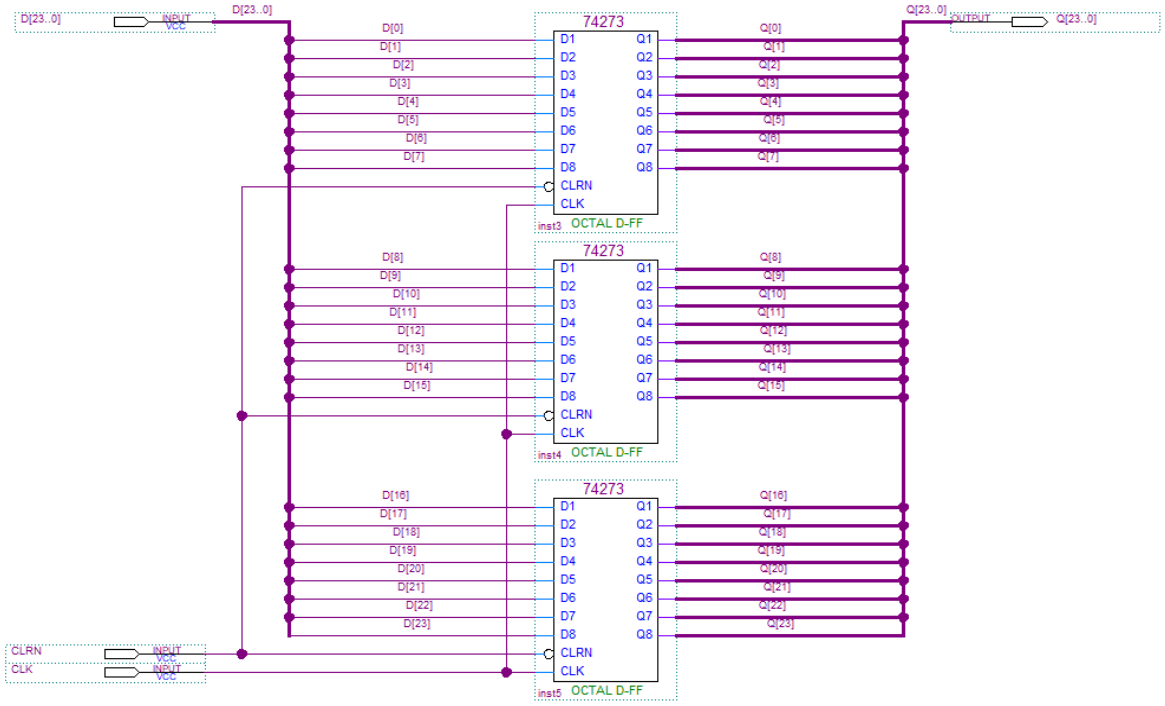

图 8: 微程序控制的运算器架构中存储器设计

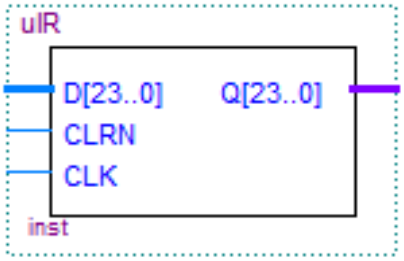

图 9: 微程序控制的运算器架构中存储器封装

2.3.4 运算器 ALU

在运算器的架构设计中，输入、输出均需要连接寄存器，但原有的 74181 元件中没有提供输入、输出寄存器的部分，因此我们需要手动在原有的 ALU 元件中加入输入、输出寄存器的部分。

另外原有的 74181 为 4 位运算器，因此我们需要拼接两个 74181 并加入 74182 组成超前进位并行加法器，具体结构如图10所示。

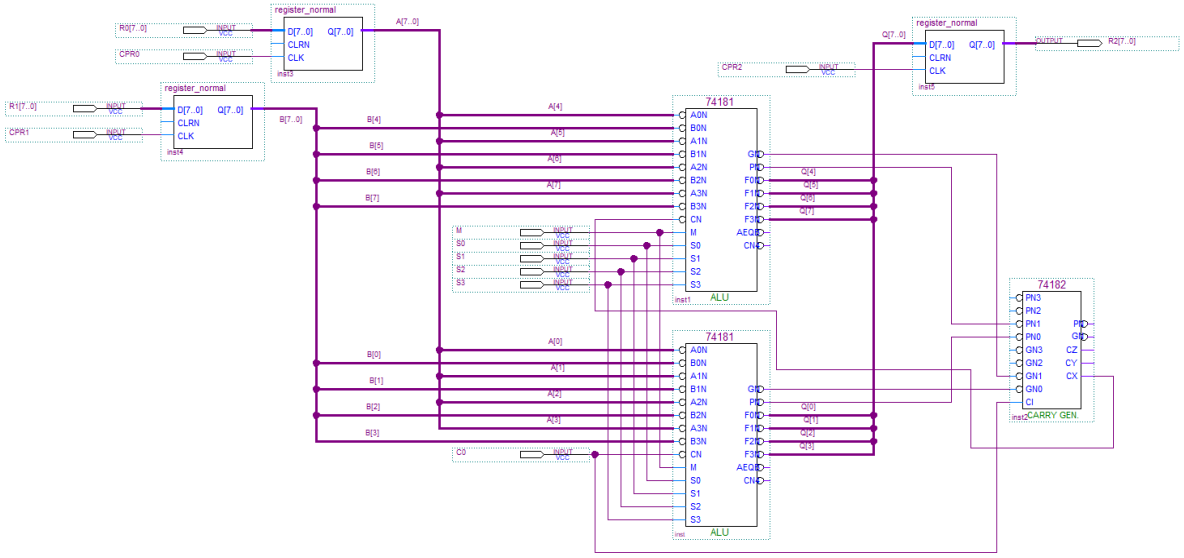

图 10: 微程序控制的运算器架构中运算器设计

其中 register_normal 元件是使用 74273 封装的 8 位寄存器，主要提供 ALU 的输入、输出端。最后，我们将 ALU 元件封装，结果如图11所示。

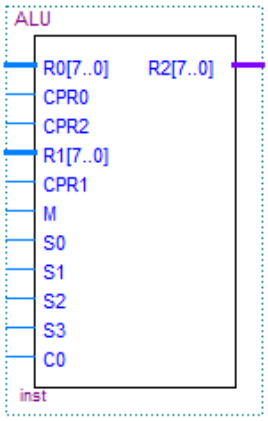

图 11: 微程序控制的运算器架构中运算器封装

2.4 完整电路设计

设计完上述的 4 个元件之后，我们便可以较方便地搭建出我们所要设计的“微程序控制的运算器”架构，具体电路设计如图12所示。

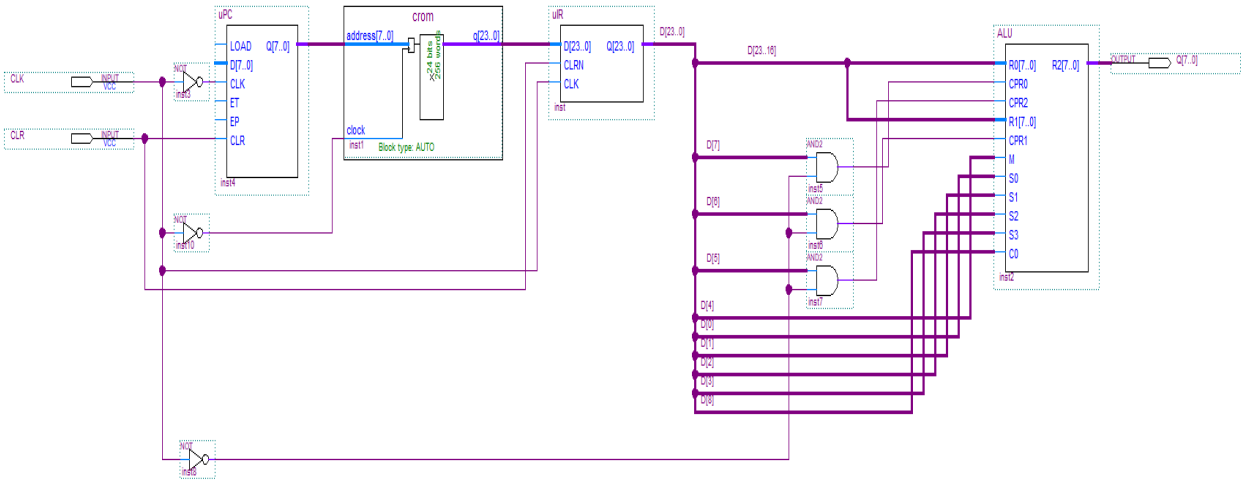

图 12: 微程序控制的运算器架构中运算器完整电路

注意 uPC、CROM、uIR 的 CLK 均取反，其中 uIR 取反是为了和 uPC 保持一致，CROM 取反是因为 ROM 是下降沿打入。

而 uPC 取反的原因是，让 CROM 比 uPC 先打入，才可以读出 CROM 中第一个单元的数据。若 uPC 和 CROM 同时打入，则 uPC 的输出值会立马变成 01H，即无法输出 00H，因此我们需要令 CROM 比 uPC 先打入。

除了上述时钟的控制以外，其余地方没有太大难度。至此，我们实现了微程序控制的运算器设计。

2.5 仿真测试

首先是内存设置，此处一共存入两条指令，分别执行 A+B 和 A+B+1 两个命令，内存存储如图13 所示。

Addr	+0	+1	+2	+3	+4	+5	+6	+7
000	2601E9	000000	000000	000000	2600E9	000000	000000	000000
008	000000	000000	000000	000000	000000	000000	000000	000000

图 13: 微程序控制的运算器架构内存设置

运行仿真，输入 CLK 的波形，即可得到输出结果，如图14 所示。可以发现输出结果恰好为 A+B 和 A+B+1 的答案，因此电路验证正确。

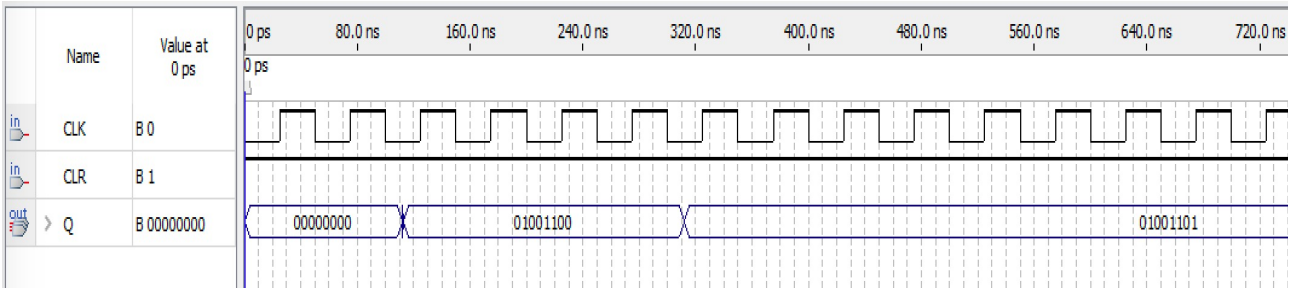

图 14: 微程序控制的运算器架构仿真测试

3 微程序控制的存储器读写系统设计

3.1 设计概述

3.1.1 设计目的

- 熟悉随机存储器读写系统结构设计
- 熟悉随机存储器的读写时序
- 熟悉随机存储器的读写操作的微程序实现
- 熟悉随机存储器的功能测试

3.1.2 设计简述

- 设计容量为 256*8 的随机存储器 RAM 和容量为 256*24 的控制存储器 CROM
- 在 RAM 和 CROM 的基础上，设计相应的外围电路和时序对随机存储器进行读写操作

3.2 架构设计

在设计完第一个练习架构——“微程序控制的运算器设计”之后，我们将开始第二个练习架构的设计过程——“微程序控制的存储器读写系统设计”。

首先我们需要了解整体的架构设计，如图15所示。

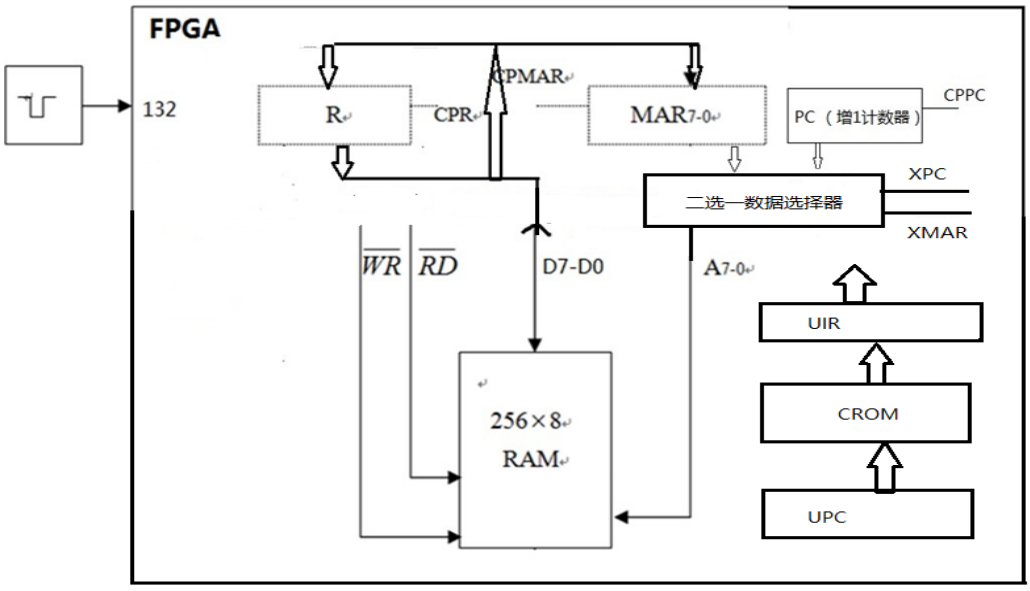

图 15: 微程序控制的存储器读写系统架构设计

根据上述架构，我们暂用图16所示指令格式进行设计。

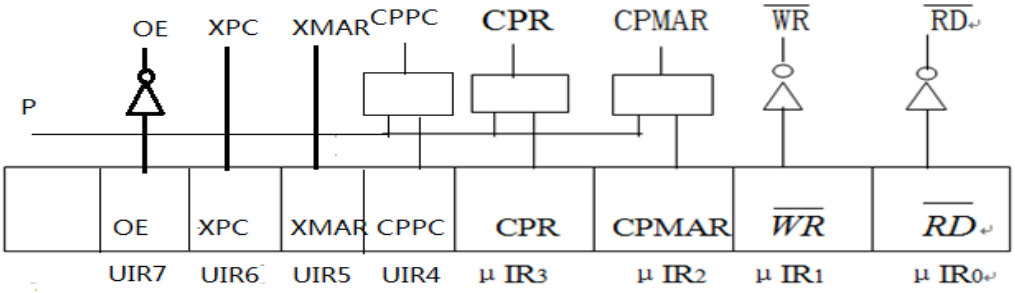

图 16: 微程序控制的存储器读写系统指令格式

上述架构的整体逻辑是，uPC+CROM+uIR 选出一个指令，然后根据 XPC 与 XMAR 两个信号从数据选择器中选出 PC 中或 MAR 中的地址，然后根据选出的地址对 RAM 进行读取或写入。

其中 R 寄存器用于存数，MAR 用于存地址，因此我们需要利用上述这个架构实现下述 4 个指令：

- 从 0 号单元取出 Ad1 放入 MAR
- 将 MAR 地址传入 RAM 中，取出对应数据，存入 R 寄存器
- 从 1 号单元中取出 Ad2 放入 MAR
- 将 R 寄存器中数据存入 MAR 中对应的地址单元

接下来我们再次将上述电路中的元件分离，逐一实现 R、MAR、PC、RAM、数据选择器。

3.3 各元件设计

3.3.1 R 寄存器

我们通过封装锁存器 74374 的输入输出来实现 R 寄存器，该寄存器的主要特点在于 OEN 的三态允许控制端，比普通的寄存器更适合应用在 CPU 设计中，具体电路如图17所示。

封装一下，即可得到如图18所示的 R 寄存器。

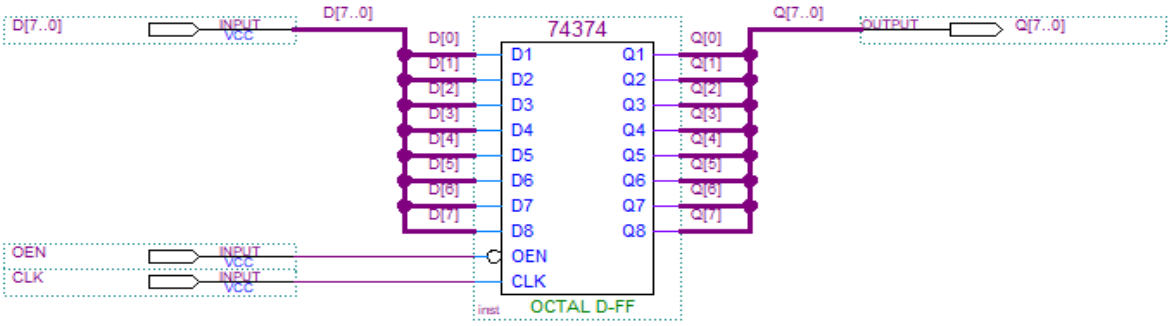

图 17: 微程序控制的存储器读写系统中 R 寄存器设计

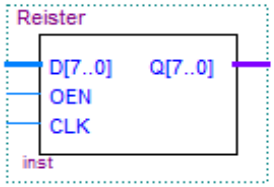

图 18: 微程序控制的存储器读写系统中 R 寄存器封装

3.3.2 地址寄存器 MAR

对比 R 寄存器使用了锁存器，MAR 寄存器则可以直接使用普通的寄存器完成，即直接调用 74273 进行实现，主要目的就是封装输入、输出端，具体电路如图19所示。

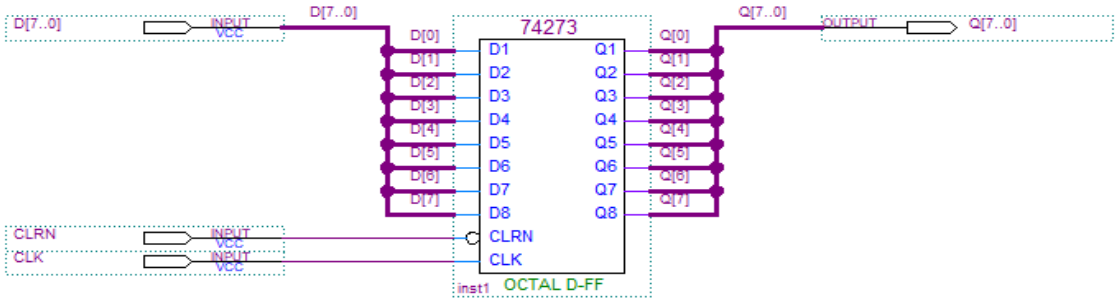

图 19: 微程序控制的存储器读写系统中 MAR 设计

封装一下，即可得到如图20所示的地址寄存器 MAR。

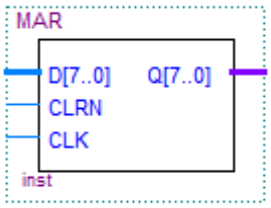

图 20: 微程序控制的存储器读写系统中 MAR 封装

3.3.3 指令寄存器 PC

PC 指令寄存器与第二篇文章中所实现的 uPC 没有较大差别，因此我们直接对 74161 进行封装即可，主要目的仍然是封装输入、输出端，具体电路如图21所示。

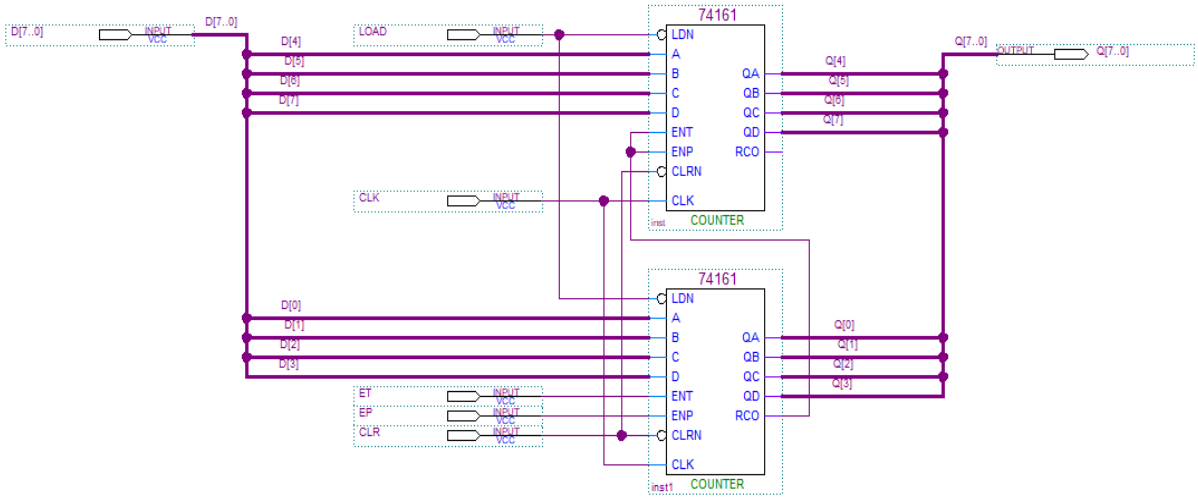

图 21: 微程序控制的存储器读写系统中 PC 设计

封装一下，即可得到如图22所示的指令寄存器 PC。

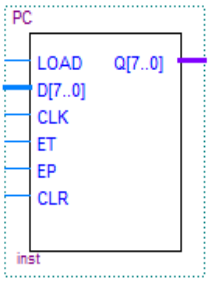

图 22: 微程序控制的存储器读写系统中 PC 封装

3.3.4 RAM

RAM 的设计过程与 ROM 一致，点击图23中的按钮，根据流程即可设计 1-port 的 RAM，此处设计部分为 QuartusII 应用的操作，按软件指示来即可。

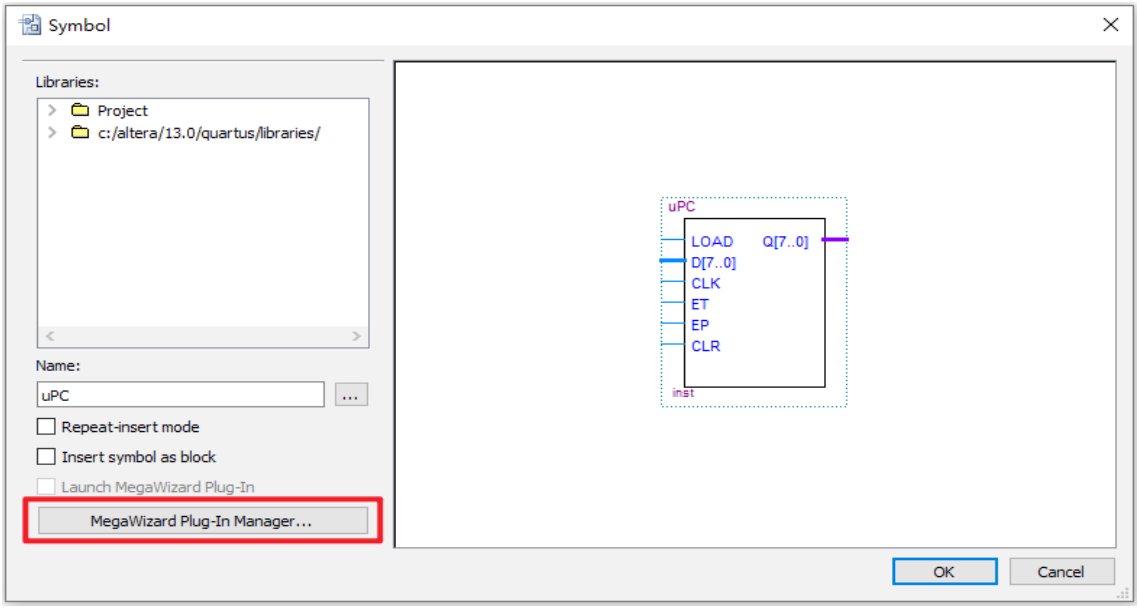

图 23: 微程序控制的存储器读写系统中 RAM 设计

最后的设计成果如图24所示。

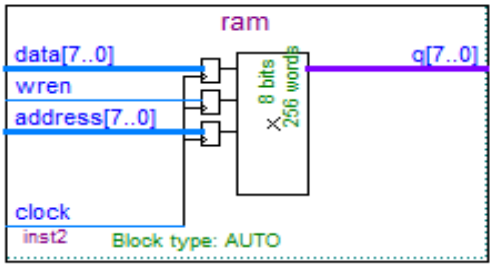

图 24: 微程序控制的存储器读写系统中 RAM 封装

此处有两个地方需要注意一下：

1. wren 低电平时读允许，高电平时写允许
2. CLK 下降沿打入

3.3.5 数据选择器

最后我们还需要实现二选一的数据选择器，即按照每一位进行与运算，从而选出我们想要的数据，实现的思路较为简单，具体电路如图25 所示。

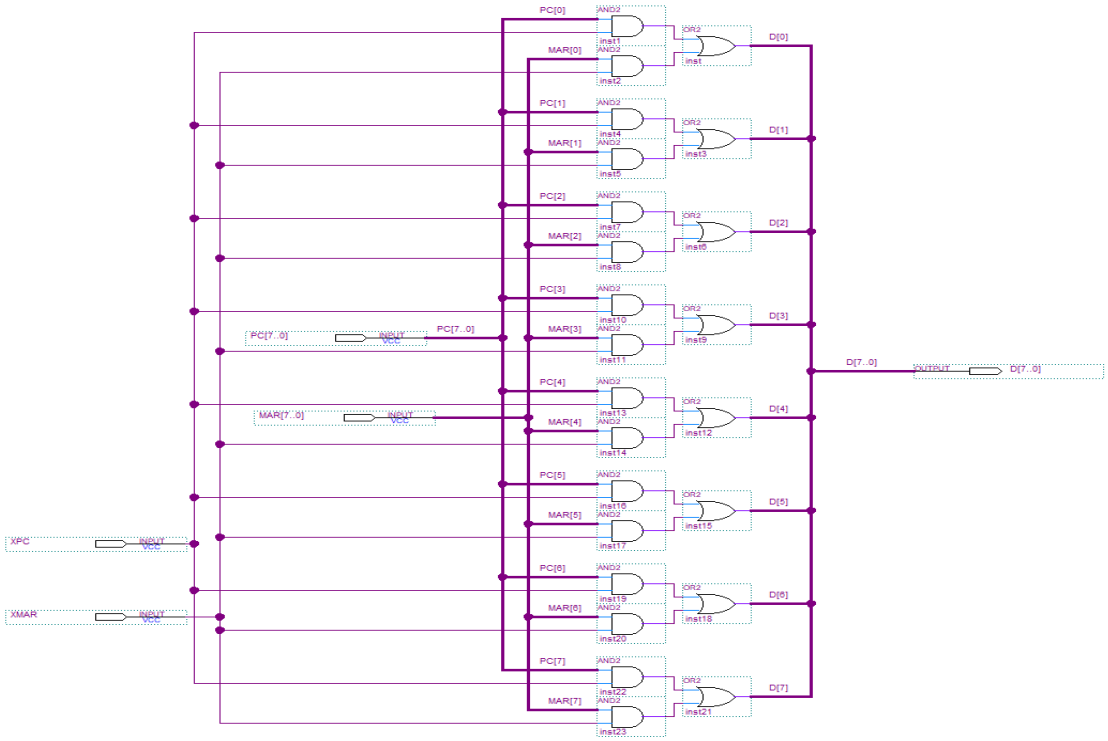

图 25: 微程序控制的存储器读写系统中数据选择器设计

封装一下，即可得到对应的二选一数据选择器，如图26所示。

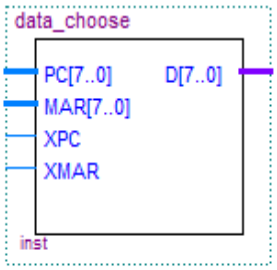

图 26: 微程序控制的存储器读写系统中数据选择器封装

3.4 完整电路设计

设计完上述的 5 个元件之后，再根据最开始给出的架构设计方案，即可搭建出我们所要设计的架构——微程序控制的存储器读写系统，具体电路如图27所示。

不难发现，指令中的 WR 和 RD 被合并成了一个 bit，其实 XPC 和 XMAR 也可以合并成一个 bit，但为了设计方便，此处没有进行合并。

除此之外，需要仔细关注三个时钟输入端。在一开始的设计中，CLK1 和 CLK2 其实是设计成为一个输入端的，但是由于延迟的存在，可能会发生 IR 的信号还没传过来，CLK2 的上升沿就到了，导致传递出去的信号仍然是旧的信号，从而导致错误。因此在

最终电路中将 CLK1、CLK2、CLK_ram 三个时钟端进行了分离，才仿真成功！

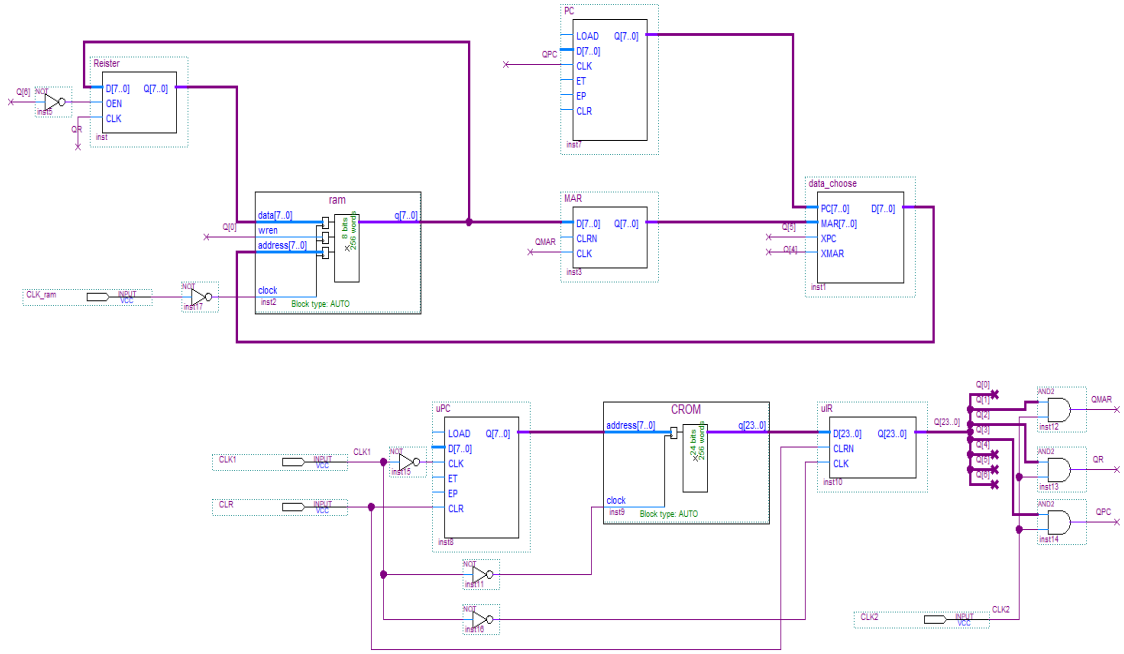

图 27: 微程序控制的存储器读写系统完整电路

3.5 仿真测试

首先是 CROM 指令预存，具体数据如图28所示。

Addr	+0	+1	+2	+3	+4	+5	+6	+7
000	000022	00001C	000022	000051	000000	000000	000000	000000
008	000000	000000	000000	000000	000000	000000	000000	000000

图 28: 微程序控制的存储器读写系统 CROM 设置

CROM 中一共有 4 条指令，其功能分别是：

1. 从 0 号单元中取出 Ad1 放入 MAR
2. 将 MAR 地址传入 RAM 中，取出对应数据，存入 R 寄存器
3. 从 1 号单元中取出 Ad2 放入 MAR
4. 将 R 寄存器中数据存入 MAR 中对应的地址单元

接下来是 RAM 中的数据预存，具体数据如图29所示。

Addr	+0	+1	+2	+3	+4	+5	+6	+7
000	09	10	00	00	00	00	00	00
008	00	20	00	00	00	00	00	00

图 29: 微程序控制的存储器读写系统 CROM 设置

将上述 CROM、RAM 数据存入对应存储器中，再按下述波形调整 CLK1、CLK2、CLK_ram 三个波形，即每一次先执行 CLK1 传入指令，再执行 CLK_ram 输出结果，最后执行 CLK2 将数据传入对应寄存器中。

观察 PC_out、mar_out、R_out、ran_out，不难发现上述指令均执行成功!

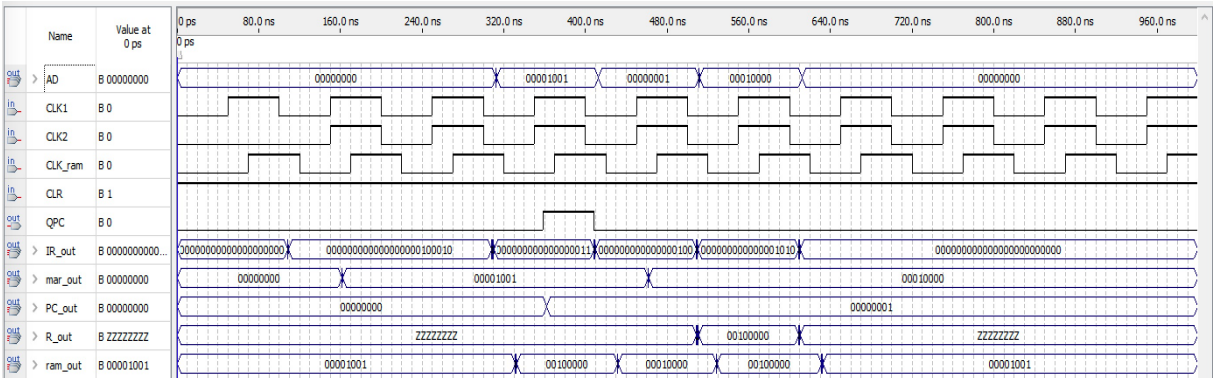

图 30: 微程序控制的存储器读写系统仿真测试

4 指令系统

4.1 指令格式

- 存储器 RAM 容量为 256*8，基本字长定为 8 位
- 指令格式可有单字长指令和双字长指令两种
- 在双字长格式中，第二子节一般定义为操作数或操作数地址

4.2 指令集合

4.2.1 指令编码

指令前 4 位决定指令类别，后 4 位决定指令操作数类型，一共有 29 类指令。

指令类别	操作码	指令缩写	完整编码
取指	0000	SELECT	00H
LW	0001	LW0	10H
		LW1	14H
		LW2	18H
		LW3	1CH
SW	0010	SW0	20H
		SW1	28H
PUSH	0011	PUSH	30H
POP	0100	POP	40H
JUMP	0101	BEQ	50H
		BNE	58H
JR	0110	JR	60H
ADD	0111	ADD0	70H
		ADD1	78H
SUB	1000	SUB0	80H
		SUB1	88H
MUL	1001	MUL0	90H
		MUL1	98H
DIV	1010	DIV0	A0H
		DIV1	A8H
MOD	1011	MOD0	B0H
		MOD1	B8H
AND	1100	AND0	C0H
		AND1	C8H
OR	1101	OR0	D0H
		OR1	D8H
XOR	1110	XOR0	E0H
		XOR1	E8H
HALT	1111	HALT	F0H

图 31: 指令编码

4.2.2 指令解析

表 1: 指令功能解析

指令缩写	指令编码	指令功能
SELECT	00H	从 RAM 中取出指令存入 IR
LW0	10H	立即数为数据，存入 R0
LW1	14H	立即数为数据，存入 R1
LW2	18H	立即数为地址，其内存中的数据存入 R0
LW3	1CH	立即数为地址，其内存中的数据存入 R1
SW0	20H	立即数为地址，R0 中数据存入其对应内存单元
SW1	28H	立即数为地址，R1 中数据存入其对应内存单元
PUSH	30H	将 R0、R1、PC 值存入栈中，PC 转到跳转地址
POP	40H	恢复现场，恢复 R0、R1、PC 数据
BEQ	50H	R0 为 0 则 PC 转到跳转地址，否则 PC 加 1
BNE	58H	R0 不为 0 则 PC 转到跳转地址，否则 PC 加 1
JR	60H	PC 无条件转到跳转地址
ADD0	70H	R0 加立即数，得到的数据存回 R0
ADD1	78H	R1 加立即数，得到的数据存回 R1
SUB0	80H	R0 减立即数，得到的数据存回 R0
SUB1	88H	R1 减立即数，得到的数据存回 R1
MUL0	90H	R0 乘立即数，得到的数据存回 R0
MUL1	98H	R1 乘立即数，得到的数据存回 R1
DIV0	A0H	R0 除立即数，得到的数据存回 R0
DIV1	A8H	R1 除立即数，得到的数据存回 R1
MOD0	B0H	R0 对立即数取余，得到的数据存回 R0
MOD1	B8H	R1 对立即数取余，得到的数据存回 R1
AND0	C0H	R0 对立即数进行与操作，得到的数据存回 R0
AND1	B8H	R1 对立即数进行与操作，得到的数据存回 R1
OR0	D0H	R0 对立即数进行或操作，得到的数据存回 R0
OR1	D8H	R1 对立即数进行或操作，得到的数据存回 R1
XOR0	E0H	R0 对立即数进行异或操作，得到的数据存回 R0
XOR1	E8H	R1 对立即数进行异或操作，得到的数据存回 R1
HALT	F0H	停机

5 总体结构与数据通路

5.1 总体结构与数据通路框图

模型机在 RAM 基础上引入了 SRAM，主要目的是作为栈空间，支持指令集中 PUSH、POP 指令的具体微操作。另外，MAR 为 RAM 提供地址，SP 为 SRAM 提供地址，各寄存器数据总线为 RAM 和 SRAM 提供数据。

各寄存器数据总线由 LA、LB、ALU 在 3-1 选择器中选择产生，其中 LB 有一个输入端为 01H，主要目的是用于 SP=SP+1 与 SP=SP-1 操作。

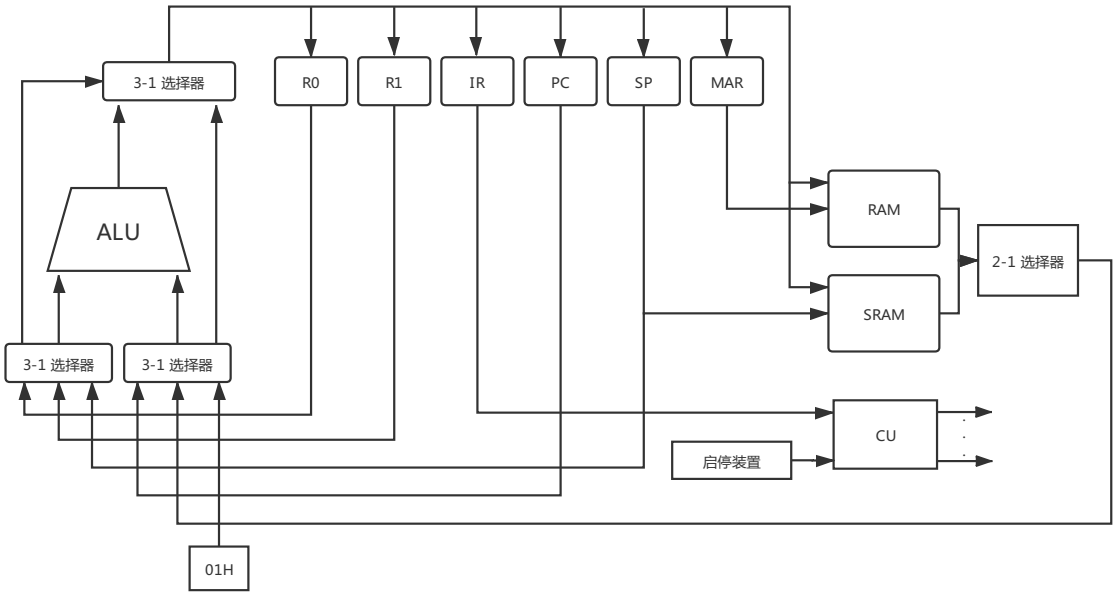

图 32: 总体结构与数据通路框图

5.2 各部件功能

表2即为本模型机总体结构中的主要部件，其对应功能决定了该模型机的能力。

5.3 各部件结构

5.3.1 8 位寄存器

8 位寄存器可直接在 74273 的基础上封装输入输出端口得到，具体结构如图33所示。

5.3.2 8 位计数器

8 位计数器可由两个 4 位计数器 74161 拼接而成，具体结构如图34所示。

表 2: 各部件功能

部件名	功能
R0	8 位通用寄存器
R1	8 位通用寄存器
IR	8 位指令寄存器
MAR	8 位地址寄存器
SP	8 位栈指针寄存器
PC	带置数的 8 位指令计数器
ALU	运算器，包含加减乘除、取余、与、或、异或等功能
RAM	可读写存储器，用于存放指令和数据
SRAM	可读写存储器，用于栈内数据存储
CU	控制器，根据指令类型输出受控门状态
3 选 1 选择器	8 位 3 选 1 数据选择器，从 3 个输入中选取一个作为输出
启停装置	控制整个程序启动与停止

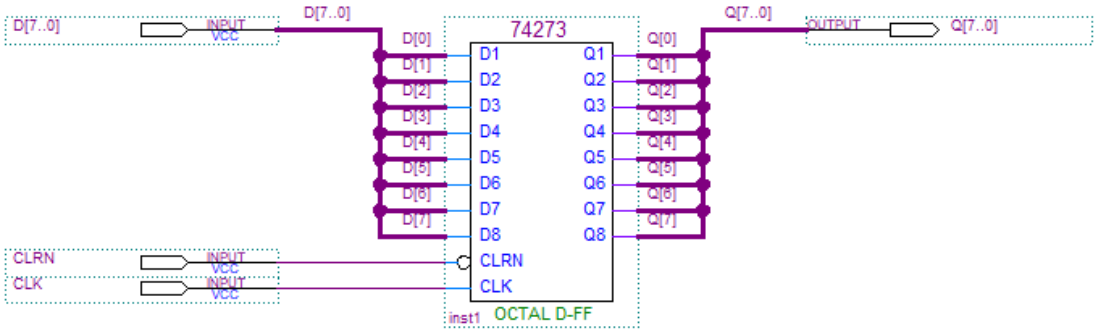

图 33: 8 位寄存器

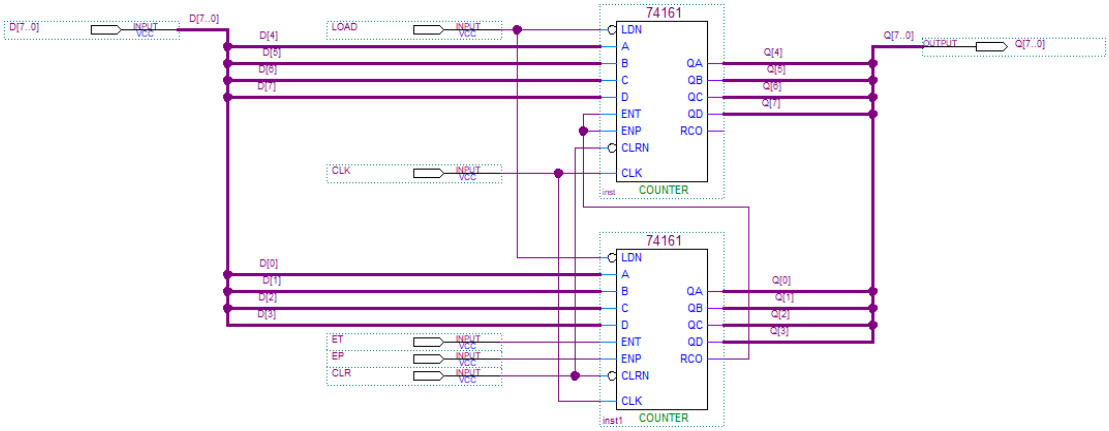

图 34: 8 位计数器

5.3.3 3 选 1 数据选择器

3-1 数据选择器即对于三个输入数据，选择一个作为输出。因此我们需要对三个输入数据的每一位进行与运算，从而选出我们想要的数据，具体结构如图35所示。

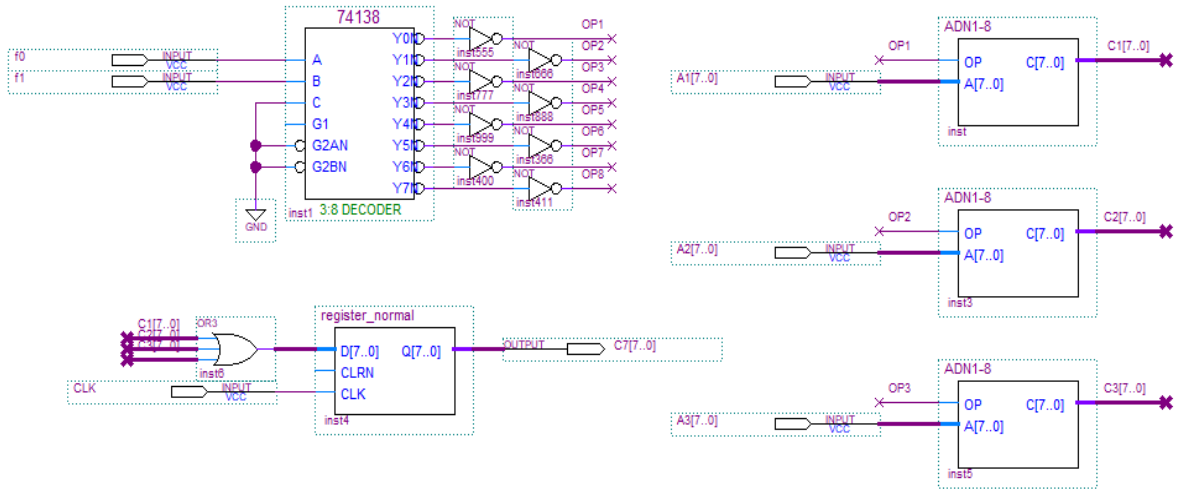

图 35: 3 选 1 数据选择器

5.3.4 ALU

最终模型机的 ALU 是最初架构一 ALU 的扩充版，引入了乘、除、与、或、异或等运算，具体结构如图36所示。

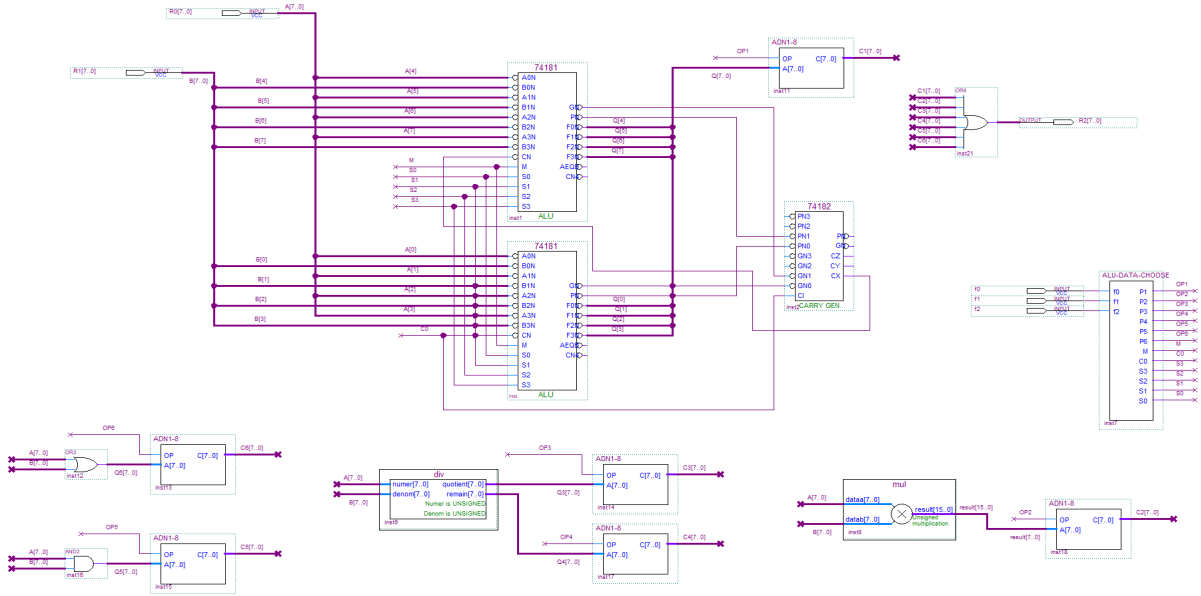

图 36: ALU

5.3.5 启停装置

启停装置主要用于控制整个模型机的启动与停止，其结构由 74161 计数器与 74138 译码器拼接而成，为整个模型机周期性提供 T0-T5 这六个时钟信号，具体电路如图37所示。

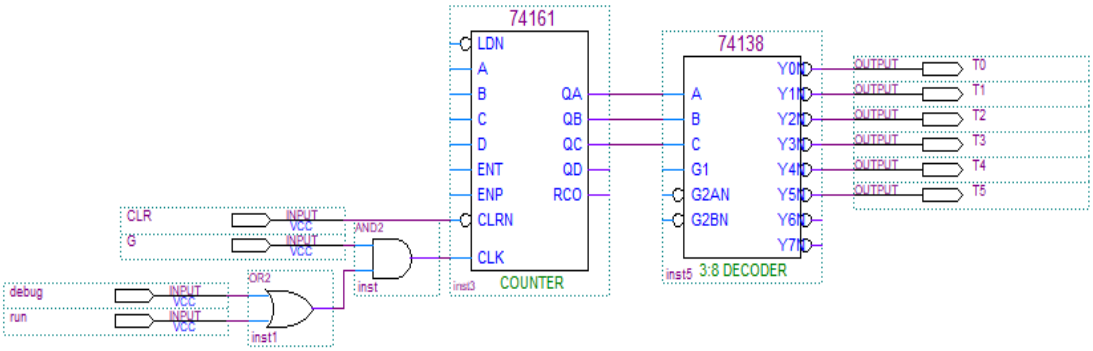

图 37: 启停装置

6 微程序实现的模型机内核

6.1 总体设计

微程序实现的模型机内核主要由控制存储器 CROM 与微程序计数器 uPC 组成，其中 uPC 识别指令操作码，输出对应微指令在 CROM 中的地址，CROM 则输出对应地址的数据，即各受控门状态。另外，启停装置为 CU 模块提供 T0 时钟，为其余模块提供 T1-T5 时钟。

微程序实现的模型机一共有 24 个受控门，分别对应 ins[0]-ins[23]，因此 CROM 容量为 8*24。

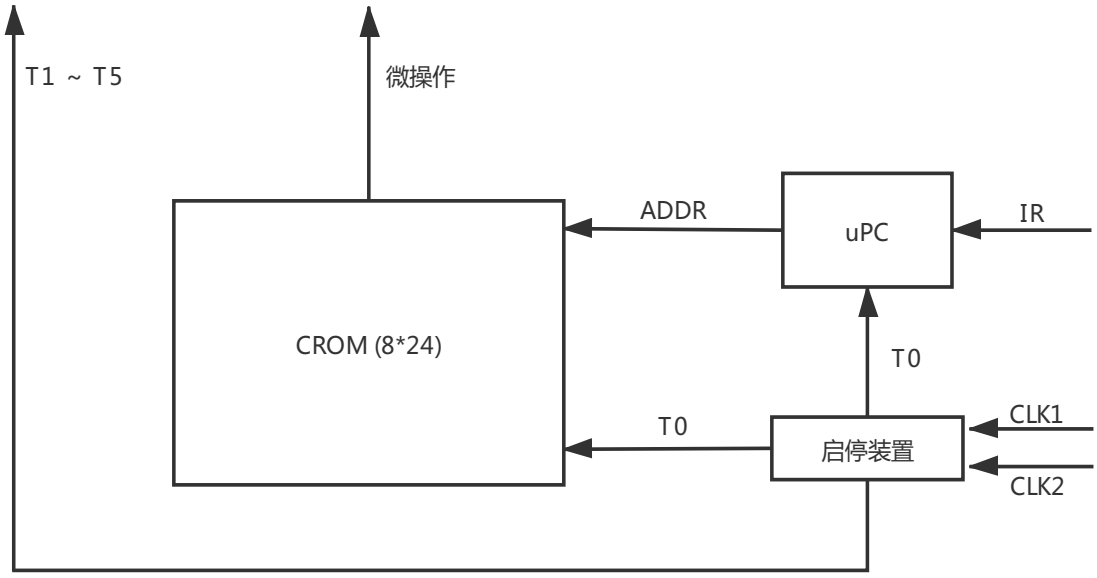

图 38: 微程序实现的控制器总框图

6.2 微指令

6.2.1 取指指令 SELECT

取指指令 SELECT 一共有 2 个微操作，第二个微操作同时完成 3 个微命令，其中微命令 JP1 表示将 IR 的值赋给 uPC，具体微操作及代码如表3所示。

表 3: 取指指令 SELECT 的微操作

指令缩写	指令地址	指令微操作	微操作代码
SELECT	00H	PC → MAR	088000H
		M(MAR) → IR	248201H
		PC = PC + 1	
		JP1	

6.2.2 取数指令 LW

取数指令 LW 根据其寻址方式、通用寄存器的不同，一共分为 4 类，分别是 LW0、LW1、LW2、LW3，具体微操作及代码如表4所示。

微命令 JP0 表示清空 uPC，即当前指令执行结束，重新开始取指。

表 4: 取数指令 LW 的微操作

指令缩写	指令地址	指令微操作	微操作代码
LW0	10H	PC → MAR	088000H
		M(MAR) → R0	418201H
		PC = PC + 1	
		JP0	
LW1	14H	PC → MAR	088000H
		M(MAR) → R1	428201H
		PC = PC + 1	
		JP0	
LW2	18H	PC → MAR	088000H
		M(MAR) → MAR	088200H
		M(MAR) → R0	418201H
		PC = PC + 1	
		JP0	
LW3	1CH	PC → MAR	088000H
		M(MAR) → MAR	088200H
		M(MAR) → R1	428201H
		PC = PC + 1	
		JP0	

6.2.3 存数指令 SW

存数指令 SW 根据其通用寄存器的不同，一共分为 2 类，分别是 SW0、SW1，具体微操作及代码如表5所示。。

6.2.4 压栈指令 PUSH

压栈指令 PUSH 作为指令集中最复杂的指令，一共有 9 个微操作，主要功能是将 R0、R1、PC 存入栈中，并执行 PC 跳转，具体微操作及代码如表6所示。

另外，SM(SP) 表示栈存储器 SRAM 中栈顶指针 SP 所对应的数据，SP = SP + 1 即为压栈操作。

表 5: 存数指令 SW 的微操作

指令缩写	指令地址	指令微操作	微操作代码
SW0	20H	PC → MAR	088000H
		M(MAR) → MAR	088200H
		R0 → M(MAR)	404011H
		PC = PC + 1	
		JP0	
SW1	28H	PC → MAR	088000H
		M(MAR) → MAR	088200H
		R1 → M(MAR)	404091H
		PC = PC + 1	
		JP0	

表 6: 压栈指令 PUSH 的微操作

指令缩写	指令地址	指令微操作	微操作代码
PUSH	30H	R0 → SM(SP)	004020H
		SP = SP + 1	100500H
		R1 → SM(SP)	0040A0H
		SP = SP + 1	100500H
		PC → MAR	088000H
		M(MAR) → R0	018201H
		PC = PC + 1	
		PC → SM(SP)	008020H
		SP = SP + 1	100500H
		R0 → PC	404009H
		JP0	

6.2.5 弹栈指令 POP

弹栈指令 POP 作为压栈指令 PUSH 的对应指令，一共有 6 个微操作，主要功能是恢复 R0、R1、PC，并弹栈，具体微操作及代码如表7所示。

另外，SM(SP) 表示栈存储器 SRAM 中栈顶指针 SP 所对应的数据，SP = SP - 1 即为弹栈操作。

表 7: 弹栈指令 POP 的微操作

指令缩写	指令地址	指令微操作	微操作代码
POP	40H	SP = SP - 1	100D00H
		SM(SP) → PC	008249H
		SP = SP - 1	100D00H
		SM(SP) → R1	028240H
		SP = SP - 1	100D00H
		SM(SP) → R0	418240H
		JP0	

6.2.6 有条件跳转指令 JUMP

有条件跳转指令 JUMP 根据其条件的不同，一共分为 2 类，分别是 BEQ、BNE，即相等跳转与不等跳转，具体微操作及代码如表8所示。

该指令的实现依赖于对 PC 计数器置数端口的电路组合控制。

表 8: 有条件跳转指令 JUMP 的微操作

指令缩写	指令地址	指令微操作	微操作代码
BEQ	50H	PC → MAR	088000H
		M(MAR) → R1	028200H
		R0 == 0 ? R1 → PC : PC = PC + 1	404085H
		JP0	
BNE	58H	PC → MAR	088000H
		M(MAR) → R1	028200H
		R0 != 0 ? R1 → PC : PC = PC +1	404083H
		JP0	

6.2.7 无条件跳转指令 JR

无条件跳转指令 JR 一共有 2 个微操作，第二个微操作同时完成 2 个微命令，具体微操作及代码如表9所示。

表 9: 无条件跳转指令 JR 的微操作

指令缩写	指令地址	指令微操作	微操作代码
JR	60H	PC → MAR	088000H
		M(MAR) → PC	400209H
		JP0	

6.2.8 加法指令 ADD

加法指令 ADD 根据其通用寄存器的不同，一共分为 2 类，分别是 ADD0、ADD1，具体微操作及代码如表10所示。

表 10: 加法指令 ADD 的微操作

指令缩写	指令地址	指令微操作	微操作代码
ADD0	70H	PC → MAR	088000H
		M(MAR) → MAR	088200H
		R0 + M(MAR) → R0	410201H
		PC = PC + 1	
		JP0	
ADD1	78H	PC → MAR	088000H
		M(MAR) → MAR	088200H
		R1 + M(MAR) → R1	420281H
		PC = PC + 1	
		JP0	

6.2.9 减法指令 SUB

减法指令 SUB 根据其通用寄存器的不同，一共分为 2 类，分别是 SUB0、SUB1，具体微操作及代码如表11所示。

表 11: 减法指令 SUB 的微操作

指令缩写	指令地址	指令微操作	微操作代码
SUB0	80H	PC → MAR	088000H
		M(MAR) → MAR	088200H
		R0 - M(MAR) → R0	410A01H
		PC = PC + 1	
		JP0	
SUB1	88H	PC → MAR	088000H
		M(MAR) → MAR	088200H
		R1 - M(MAR) → R1	420A81H
		PC = PC + 1	
		JP0	

6.2.10 乘法指令 MUL

乘法指令 MUL 根据其通用寄存器的不同，一共分为 2 类，分别是 MUL0、MUL1，具体微操作及代码如表12所示。

表 12: 乘法指令 MUL 的微操作

指令缩写	指令地址	指令微操作	微操作代码
MUL0	90H	PC → MAR	088000H
		M(MAR) → MAR	088200H
		R0 * M(MAR) → R0	411201H
		PC = PC + 1	
		JP0	
MUL1	98H	PC → MAR	088000H
		M(MAR) → MAR	088200H
		R1 * M(MAR) → R1	421281H
		PC = PC + 1	
		JP0	

6.2.11 除法指令 DIV

除法指令 DIV 根据其通用寄存器的不同，一共分为 2 类，分别是 DIV0、DIV1，具体微操作及代码如表13所示。

表 13: 除法指令 DIV 的微操作

指令缩写	指令地址	指令微操作	微操作代码
DIV0	A0H	PC → MAR	088000H
		M(MAR) → MAR	088200H
		R0 / M(MAR) → R0	411A01H
		PC = PC + 1	
		JP0	
DIV1	A8H	PC → MAR	088000H
		M(MAR) → MAR	088200H
		R1 / M(MAR) → R1	421A81H
		PC = PC + 1	
		JP0	

6.2.12 取余指令 MOD

取余指令 MOD 根据其通用寄存器的不同，一共分为 2 类，分别是 MOD0、MOD1，具体微操作及代码如表14所示。

表 14: 取余指令 MOD 的微操作

指令缩写	指令地址	指令微操作	微操作代码
MOD0	B0H	PC → MAR	088000H
		M(MAR) → MAR	088200H
		R0 % M(MAR) → R0	412201H
		PC = PC + 1	
		JP0	
MOD1	B8H	PC → MAR	088000H
		M(MAR) → MAR	088200H
		R1 % M(MAR) → R1	422281H
		PC = PC + 1	
		JP0	

6.2.13 与运算指令 AND

与运算指令 AND 根据其通用寄存器的不同，一共分为 2 类，分别是 AND0、AND1，具体微操作及代码如表15所示。

表 15: 与运算指令 AND 的微操作

指令缩写	指令地址	指令微操作	微操作代码
AND0	C0H	PC → MAR	088000H
		M(MAR) → MAR	088200H
		R0 & M(MAR) → R0	412A01H
		PC = PC + 1	
		JP0	
AND1	C8H	PC → MAR	088000H
		M(MAR) → MAR	088200H
		R1 & M(MAR) → R1	422A81H
		PC = PC + 1	
		JP0	

6.2.14 或运算指令 OR

或运算指令 OR 根据其通用寄存器的不同，一共分为 2 类，分别是 OR0、OR1，具体微操作及代码如表16所示。

表 16: 或运算指令 OR 的微操作

指令缩写	指令地址	指令微操作	微操作代码
OR0	D0H	PC → MAR	088000H
		M(MAR) → MAR	088200H
		R0 | M(MAR) → R0	413201H
		PC = PC + 1	
		JP0	
OR1	D8H	PC → MAR	088000H
		M(MAR) → MAR	088200H
		R1 | M(MAR) → R1	423281H
		PC = PC + 1	
		JP0	

6.2.15 异或运算指令 XOR

异或运算指令 XOR 根据其通用寄存器的不同，一共分为 2 类，分别是 XOR0、XOR1，具体微操作及代码如表17所示。

表 17: 异或运算指令 XOR 的微操作

指令缩写	指令地址	指令微操作	微操作代码
XOR0	E0H	PC → MAR	088000H
		M(MAR) → MAR	088200H
		R0 xor M(MAR) → R0	413A01H
		PC = PC + 1	
		JP0	
XOR1	E8H	PC → MAR	088000H
		M(MAR) → MAR	088200H
		R1 xor M(MAR) → R1	423A81H
		PC = PC + 1	
		JP0	

6.2.16 停机指令 HALT

停机指令 HALT 只有 1 个微操作，主要功能是停止模型机的运行，具体微操作及代码如表18所示。

表 18: 停机指令 HALT 的微操作

指令缩写	指令地址	指令微操作	微操作代码
HALT	F0H	$0 \rightarrow G$	800000H

6.3 电路结构

6.3.1 微程序计数器 uPC

微程序计数器 uPC 为两个 74161 元件拼接而成的 8 位计数器，具体电路如图39 所示。

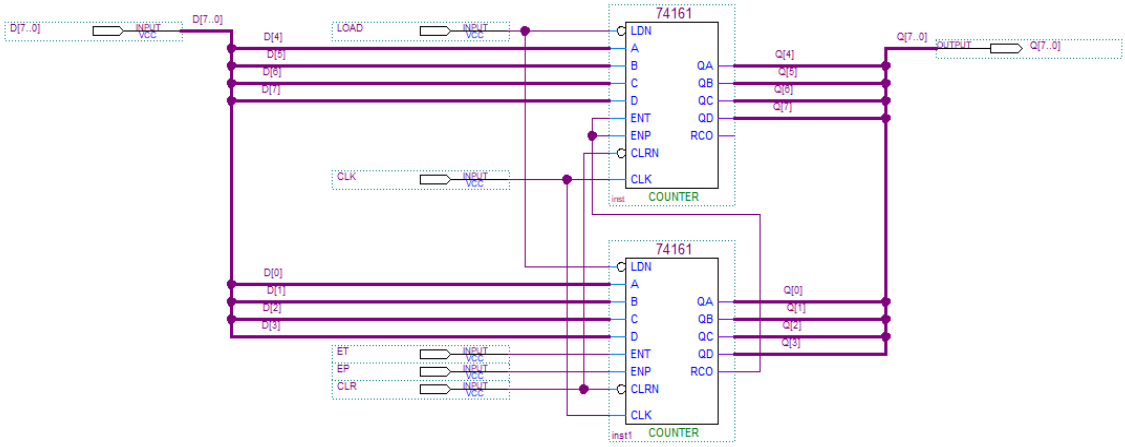

图 39: 微程序实现的控制器中的 uPC

6.3.2 完整电路

CROM 与之前的 ROM 元件没有区别，因此此处不再进行描述。将 uPC 与 CROM 拼接起来组成 CU，即可搭建出之前所述的总体结构，具体电路如图40所示。

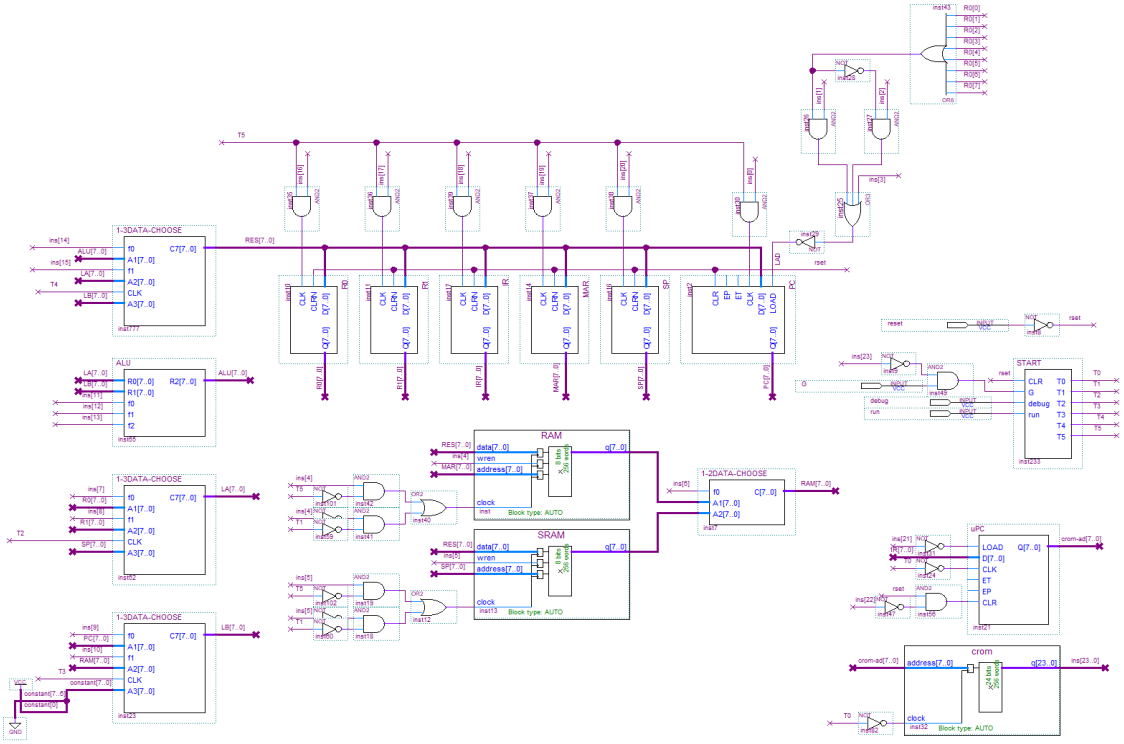

图 40: 基于微程序设计的模型机

6.4 测试程序

由于本模型机指令集中包含存取语句、运算语句、跳转语句、压/弹栈语句，因此我们编写一个递归版求斐波那契数列数的程序，用于模型机指令测试。

我们给出 C++ 版本的代码，以及本模型机支持的指令代码，用于进一步展示本模型机的功能。

C++ 代码如下：

```
int Fib(int x) {
    if(x == 0)
        return 0;
    if(x == 1)
        return 1;
    return Fib(x-1) + Fib (x-2);
}

int main() {
    Fib(10);
    return 0;
}
```

模型机对应指令如下：

```
    LW0 10#, R0
    LW1 0#, R1
```

```
3         SW0  R0, (addr0#)
4         PUSH  Fib#
5         LW1  (addr1#), R1
6         HALT
7   Fib:  LW2  (addr0#), R0
8         BNE  L1#
9         LW1  0#, R1
10        SW1  R1, (addr1#)
11        POP
12  L1:   SUB  1#, R0
13        BNE  L2#
14        LW1  1#, R1
15        SW1  R1, (addr1#)
16        POP
17  L2:   SW0  R0, (addr0#)
18        PUSH  Fib#
19        ADD1  (addr1#), R1
20        SUB0  1#, R0
21        SW0  R0, (addr0#)
22        PUSH  Fib#
23        ADD1  (addr1#), R1
24        SW1  R1, (addr1#)
25        POP
```

在上述指令中，x 存在 R0 中，最终结果存在 R1 中，地址 addr0、addr1 则用于保存 R0、R1 的临时结果。

7 硬布线实现的模型机内核

硬布线实现与微程序实现，既有相同点，又有不同点。相同点在于其指令集、指令编码、指令对应微操作、微操作对应受控门状态均一致，且测试程序也是相同的。不同点在于总体设计、实现方式不相同。因此本模块我们主要介绍不同点，对于相同点（指令集、指令编码、测试程序）则不再进行介绍。

7.1 总体设计

硬布线实现的模型机内核主要由节拍发生器、指令识别器、控制器组成，其中节拍发生器、指令识别器分别用于控制器的执行周期节拍、指令编码的输入，控制器最终生成 24 个受控门的状态。

另外需要注意的是，由于指令执行都需要进行取值，因此此处直接将取指周期与执行周期合并，统一为执行周期，但其本身的效率不会变化，具体实现见节拍发生器电路。

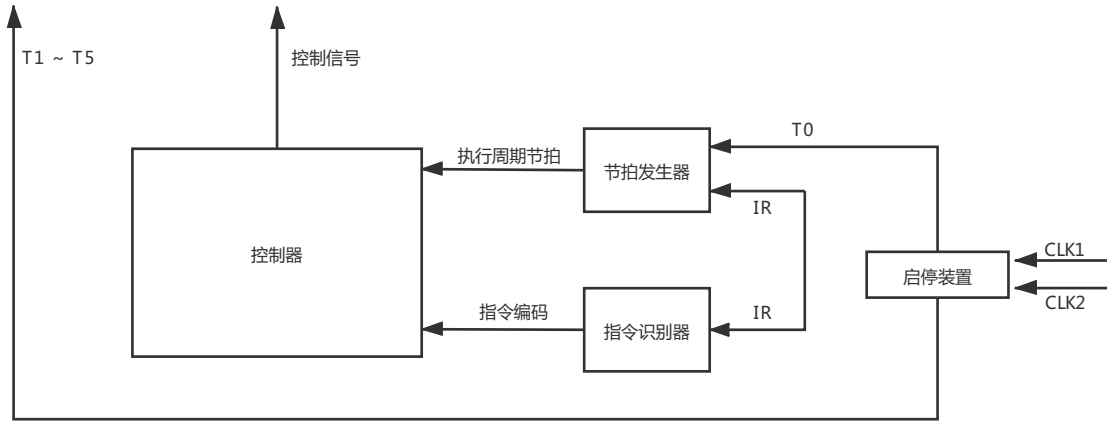

图 41: 硬布线实现的控制器总框图

7.2 各元件设计

根据图41所示的总框图，不难发现硬布线实现的控制器依赖于两个输入，一是执行周期节拍，二是具体指令编码，因此我们主要针对这两个输入进行具体元件设计的介绍。

7.2.1 指令识别器

指令识别器主要是对于 IR 的输入，将其识别成对应的指令输出。例如对于输入 18H，需要将其识别为 LW2 指令，并将 LW2 对应的输出端置 1。

具体实现过程是先使用译码器 74138 对于 IR 的高四位进行识别，然后再对于不同的高四位进行分类识别，具体电路实现如图42所示。

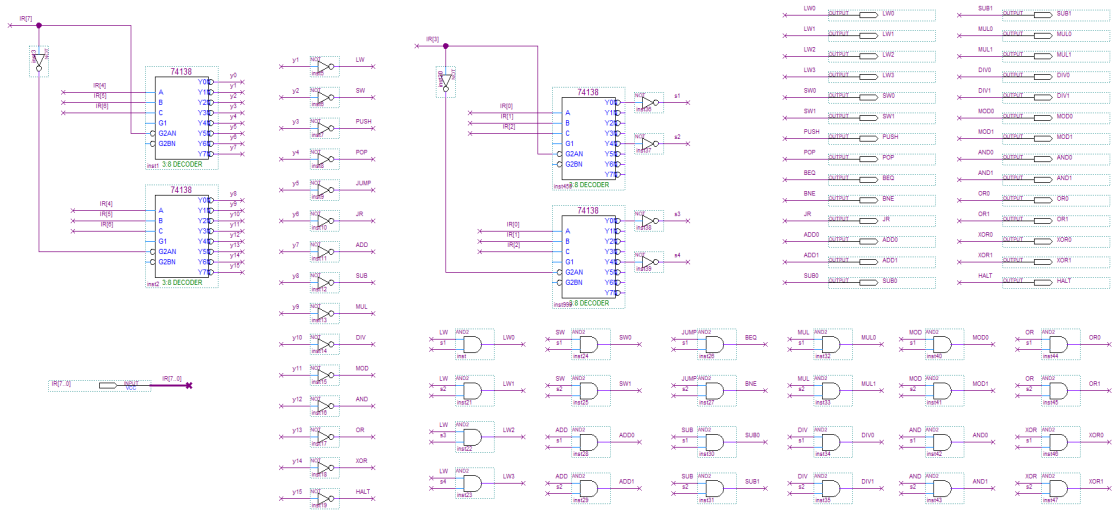

图 42: 硬布线实现的控制器中的指令识别器

7.2.2 节拍发生器

节拍发生器主要是为控制器提供执行周期节拍，简单说就是告诉控制器，不同的指令应该执行多少个周期。

本模型机中的指令，PUSH 指令一共有九个微操作，POP 指令一共有六个微操作，取指需要两个微操作，而其余指令微操作数量则均小于等于三。另外，所有指令执行都需要先取指再执行。因此本模型机令 s1、s2 为取指操作的周期，对于 POP 指令还需要执行 s3 ～ s8，PUSH 指令还需要执行 s3 ～ s11，而其余指令只需要执行 s3 ～ s5，因此我们可以得到节拍控制器置零端的组合逻辑（低位清空）。

$$clear = \overline{s12 \cdot PUSH + s9 \cdot POP + s5 \cdot (\overline{PUSH + POP})}$$

由此我们可以搭建出节拍发生器元件，具体电路实现如图43所示。

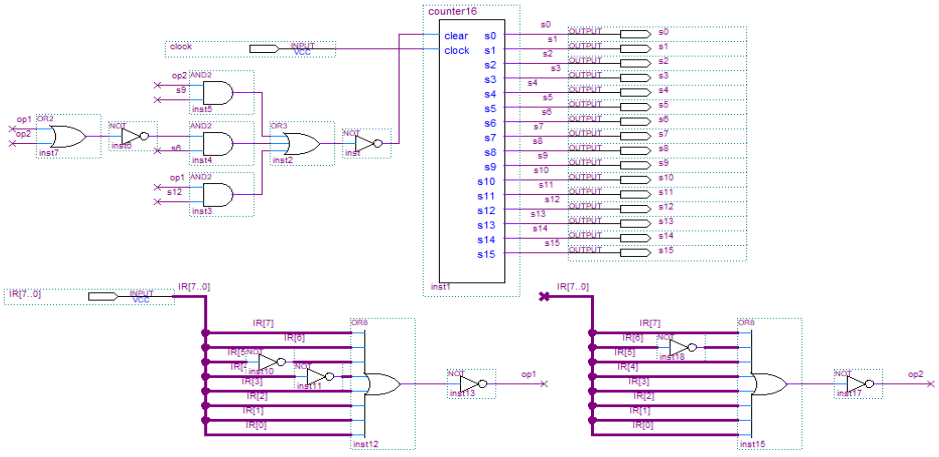

图 43: 硬布线实现的控制器中的节拍发生器

7.3 硬布线编码

根据指令识别器与节拍发生器，我们可以得到执行周期节拍与指令编码的信息，接下来我们需要根据每个指令微操作所对应的受控门状态来进行硬布线编码，即对于每个受控门设计出对应的组合逻辑电路。

表 19: 硬布线组合逻辑设计

受控门	组合逻辑
ins[0]	s2 + LW0 · s4 + LW1 · s4 + LW2 · s5 + LW3 · s5 + SW0 · s5 + SW1 · s5 + PUSH(s8 + s11) + POP · s4 + BEQ · s5 + BNE · s5 + JR · s4 + ADD0 · s5 + ADD1 · s5 + SUB0 · s5 + SUB1 · s5 + MUL0 · s5 + MUL1 · s5 + DIV0 · s5 + DIV1 · s5 + MOD0 · s5 + MOD1 · s5 + AND0 · s5 + AND1 · s5 + OR0 · s5 + OR1 · s5 + XOR0 · s5 + XOR1 · s5
ins[1]	BNE · s5
ins[2]	BEQ · s5
ins[3]	PUSH · s11 + POP · s4 + JR · s4
ins[4]	SW0 · s5 + SW1 · s5
ins[5]	PUSH(s3 + s5 + s9)
ins[6]	POP(s4 + s6 + s8)
ins[7]	SW1 · s5 + PUSH · s5 + BEQ · s5 + BNE · s5 + ADD1 · s5 + SUB1 · s5 + MUL1 · s5 + DIV1 · s5 + MOD1 · s5 + AND1 · s5 + OR1 · s5 + XOR1 · s5
ins[8]	PUSH(s4 + s6 + s10) + POP(s3 + s5 + s7)
ins[9]	s2 + LW0 · s4 + LW1 · s4 + LW2(s4 + s5) + LW3(s4 + s5) + SW0 · s4 + SW1 · s4 + PUSH · s8 + POP(s4 + s6 + s8) + BEQ · s4 + BNE · s4 + JR · s4 + ADD0(s4 + s5) + ADD1(s4 + s5) + SUB0(s4 + s5) + SUB1(s4 + s5) + MUL0(s4 + s5) + MUL1(s4 + s5) + DIV0(s4 + s5) + DIV1(s4 + s5) + MOD0(s4 + s5) + MOD1(s4 + s5) + AND0(s4 + s5) + AND1(s4 + s5) + OR0(s4 + s5) + OR1(s4 + s5) + XOR0(s4 + s5) + XOR1(s4 + s5)
ins[10]	PUSH(s4 + s6 + s10) + POP(s3 + s5 + s7)
ins[11]	POP(s3 + s5 + s7) + SUB0 · s5 + SUB1 · s5 + DIV0 · s5 + DIV1 · s5 + AND0 · s5 + AND1 · s5 + XOR0 · s5 + XOR1 · s5
ins[12]	MUL0 · s5 + MUL1 · s5 + DIV0 · s5 + DIV1 · s5 + OR0 · s5 + OR1 · s5 + XOR0 · s5 + XOR1 · s5

ins[13]	MOD0 · s5 + MOD1 · s5 + AND0 · s5 + AND1 · s5 + OR0 · s5 + OR1 · s5 + XOR0 · s5 + XOR1 · s5
ins[14]	SW0 · s5 + SW1 · s5 + PUSH(s3 + s5 + s11) + BEQ · s5 + BNE · s5
ins[15]	s1 + s2 + LW0(s3 + s4) + LW1(s3 + s4) + LW2(s3 + s4) + LW2 · s5 + LW3(s3 + s4 + s5) + SW0(s3 + s4) + SW1(s3 + s4) + PUSH(s7 + s8 + s9) + POP(s4 + s6 + s8) + BEQ(s3 + s4) + BNE(s3 + s4) + JR · s3 + ADD0(s3 + s4) + ADD1(s3 + s4) + SUB0(s3 + s4) + SUB1(s3 + s4) + MUL0(s3 + s4) + MUL1(s3 + s4) + DIV0(s3 + s4) + DIV1(s3 + s4) + MOD0(s3 + s4) + MOD1(s3 + s4) + AND0(s3 + s4) + AND1(s3 + s4) + OR0(s3 + s4) + OR1(s3 + s4) + XOR0(s3 + s4) + XOR1(s3 + s4)
ins[16]	LW0 · s4 + LW2 · s5 + PUSH · s8 + POP · s8 + ADD0 · s5 + SUB0 · s5 + MUL0 · s5 + DIV0 · s5 + MOD0 · s5 + AND0 · s5 + OR0 · s5 + XOR0 · s5
ins[17]	LW1 · s4 + LW3 · s5 + POP · s6 + BEQ · s4 + BNE · s4 + ADD1 · s5 + SUB1 · s5 + MUL1 · s5 + DIV1 · s5 + MOD1 · s5 + AND1 · s5 + OR1 · s5 + XOR1 · s5
ins[18]	s2
ins[19]	s1 + LW0 · s3 + LW1 · s3 + LW2(s3 + s4) + LW3(s3 + s4) + SW0(s3 + s4) + SW1(s3 + s4) + PUSH · s7 + BEQ · s3 + BNE · s3 + JR · s3 + ADD0(s3 + s4) + ADD1(s3 + s4) + SUB0(s3 + s4) + SUB1(s3 + s4) + MUL0(s3 + s4) + MUL1(s3 + s4) + DIV0(s3 + s4) + DIV1(s3 + s4) + MOD0(s3 + s4) + MOD1(s3 + s4) + AND0(s3 + s4) + AND1(s3 + s4) + OR0(s3 + s4) + OR1(s3 + s4) + XOR0(s3 + s4) + XOR1(s3 + s4)
ins[20]	PUSH(s4 + s6 + s10) + POP(s3 + s5 + s7)
ins[21]	0
ins[22]	0
ins[23]	HALT · s3

由于 ins[21]、ins[22] 在微程序设计中是 uPC 的输入受控门，因此在硬布线设计中使用不到，即置零。

7.4 电路结构

7.4.1 控制器

由于控制器包含了 24 个受控门的组合逻辑电路，因此我们进行分批展示。

(1) ins[0] ∼ ins[7]

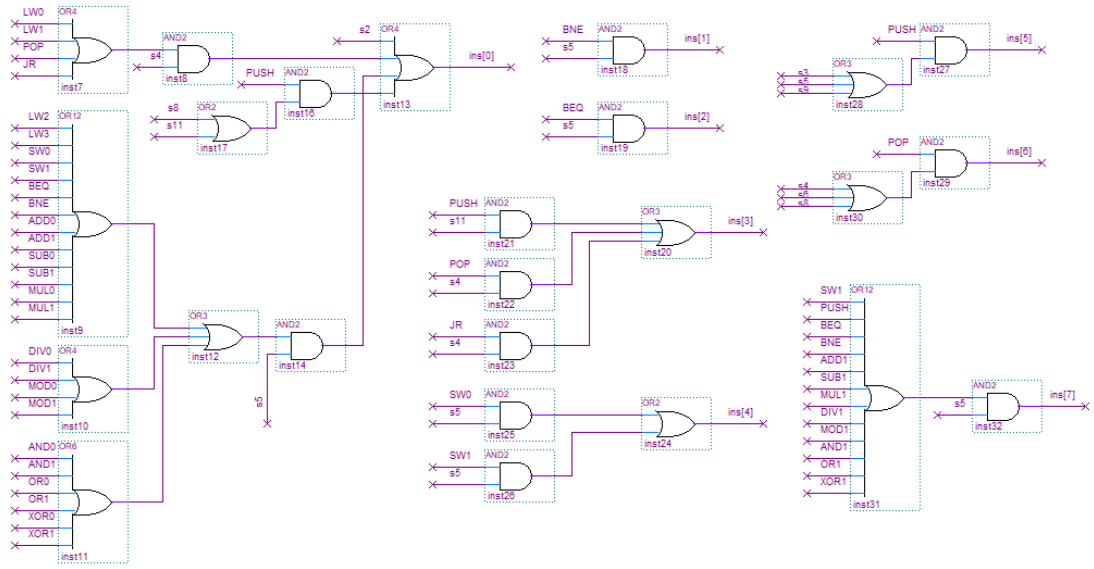

图 44: 硬布线实现的控制器中 ins[0] ∼ ins[7] 组合逻辑电路

(2) ins[8] ∼ ins[15]

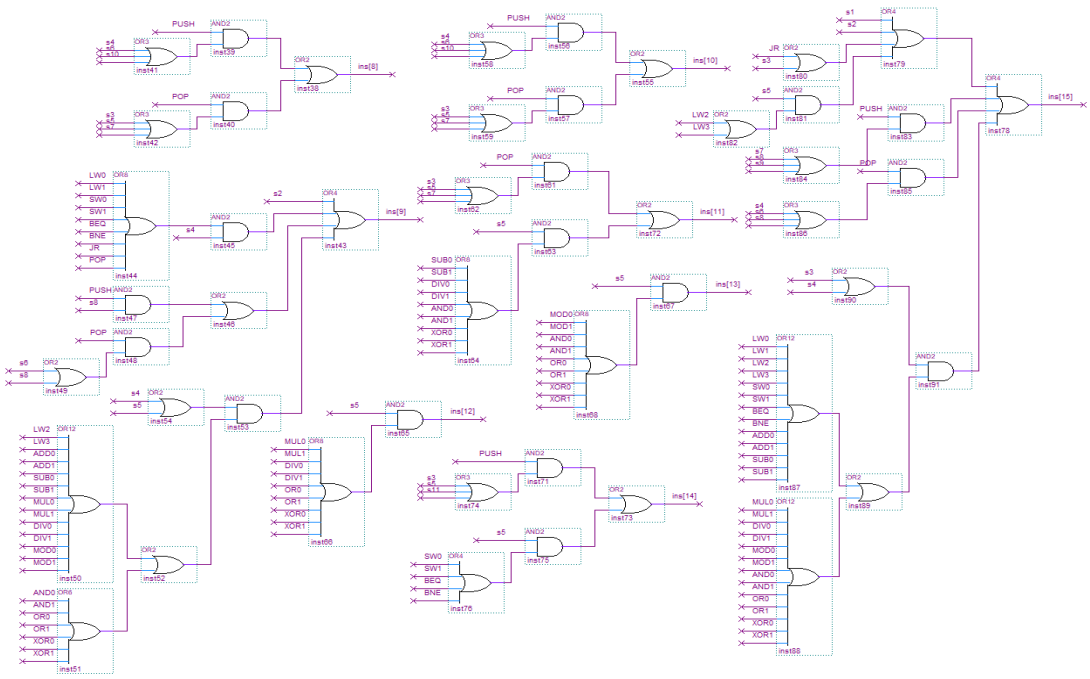

图 45: 硬布线实现的控制器中 ins[0] ∼ ins[7] 组合逻辑电路

(3) ins[16] ∼ ins[23]

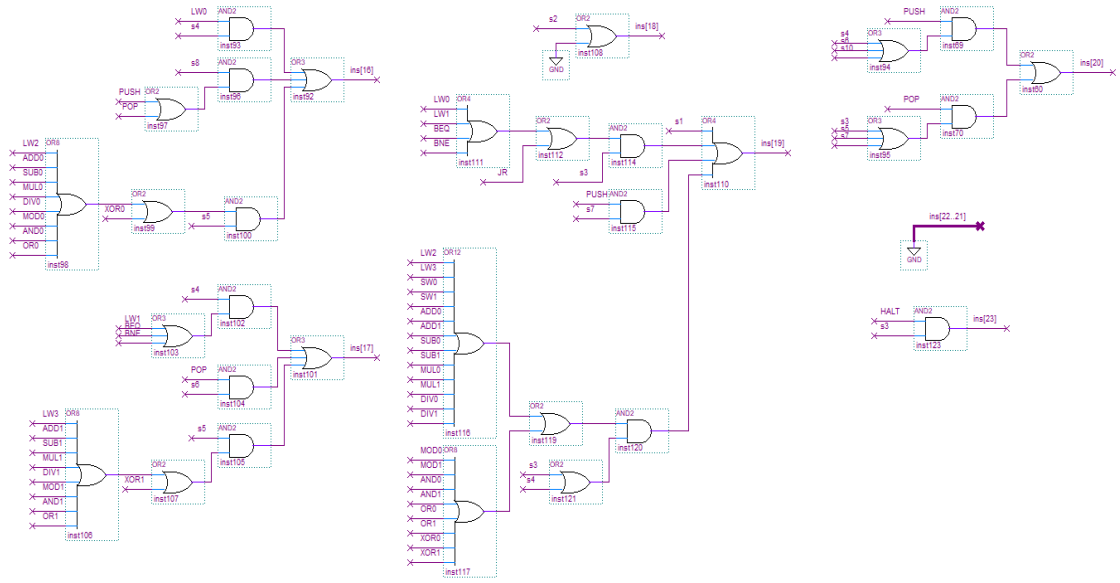

图 46: 硬布线实现的控制器中 ins[0] ∼ ins[7] 组合逻辑电路

(4) 控制器总电路

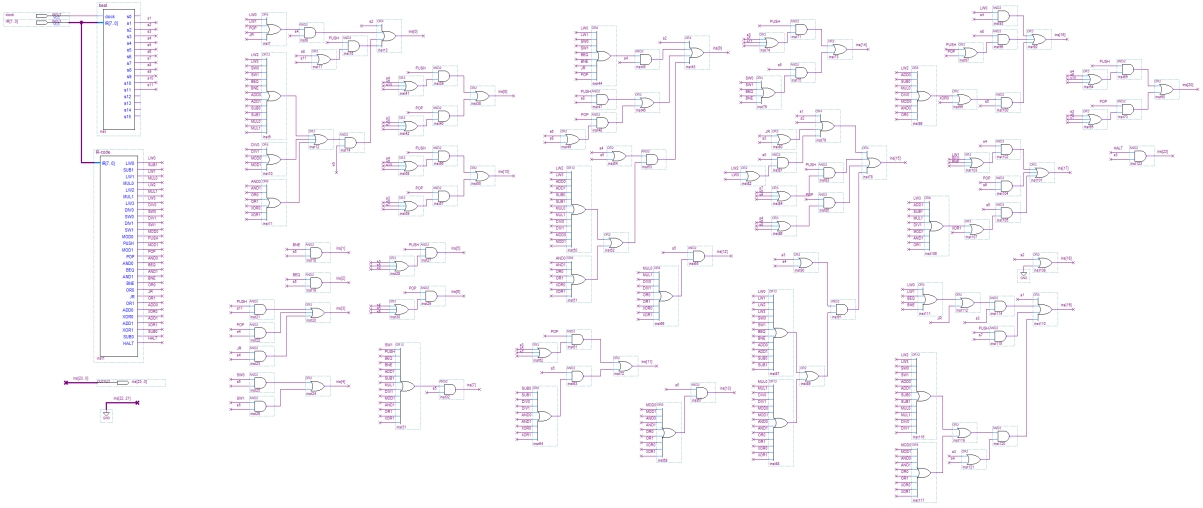

图 47: 硬布线实现的控制器总电路

7.4.2 完整电路

硬布线实现的模型机与微程序实现的模型机的唯一区别在于控制器 CU，因此我们只需用上述硬布线实现的控制器 CU 替换掉原先微程序电路中的 CU，即可构建出基于硬布线实现的模型机，具体电路如图48所示。

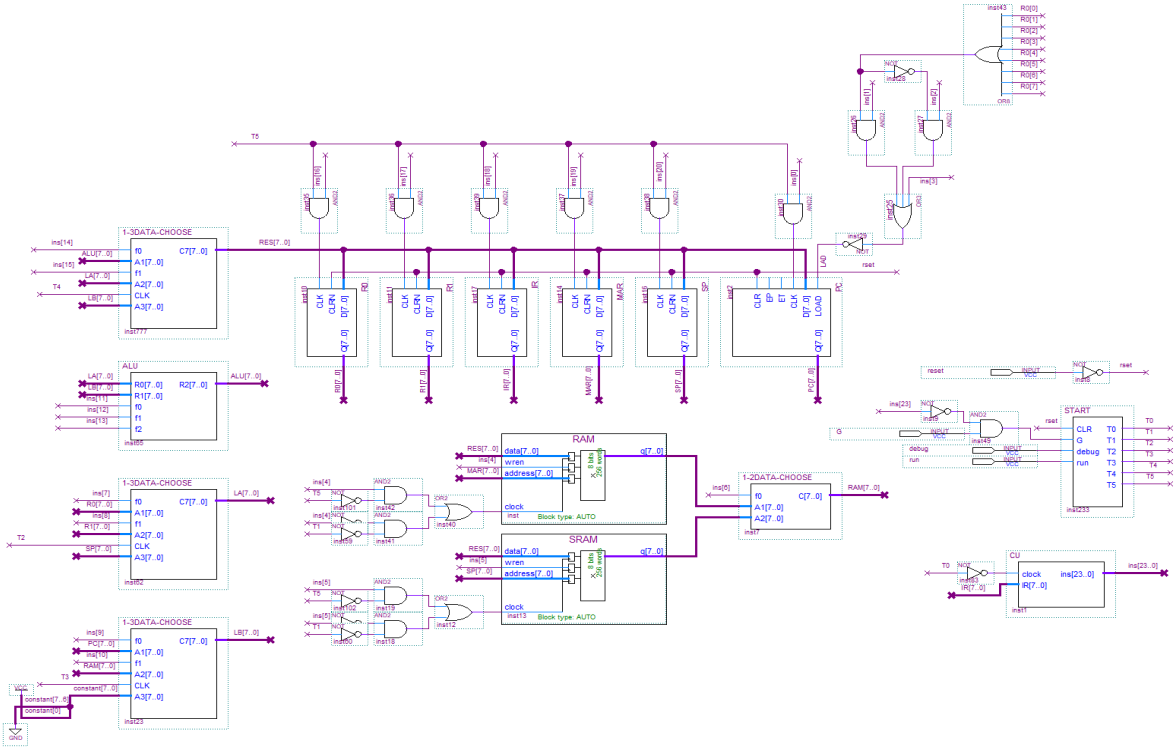

图 48: 基于硬布线设计的模型机

8 致谢与展望

感谢之前计算机组成原理课程中吕知辛老师和本次计算机组成原理课程设计中韩芳溪老师对我们的教导与指点，这些计算机组成原理知识以及设计方法在本次课设的实践过程中起到了很大的帮助，为最终模型机的设计实现打下了坚实的基础；感谢助教潘云刚、李双良学长解答我们在模型机设计中的疑问并为我们规划设计流程，这对我们后续准时完成本次模型机的设计起到了很大的帮助；感谢我的同学们在实践过程中与我一同讨论模型机设计的实现方案，分析设计可行性，以及帮助我考虑到了很多未曾想到的漏洞并为我提供了很多更高效实现方案的思路，这对整个模型机系统设计的创新性、正确性起到了重要作用。

在模型机的设计过程中，我遇到了大量未在课堂中出现的问题，并最终通过思考、资料搜寻以及同学讨论进行解决。还记得在设计指令集时，我发现原有的设计仅支持顺序逻辑执行的代码，即不包含条件、循环、递归语句，因此考虑能否增加指令以支持跳转与递归。从而进一步考虑到跳转指令本质上是对指令寄存器 PC 的修改，因此只要对 PC 的清空端进行组合逻辑设计，即可实现跳转指令。另外对于递归指令，很明显其内核在于递归栈的构建以及每次递归时的保存现场与返回时的恢复现场，由此在原有设计上引入了 BEQ、BNE、JR、PUSH、POP 等指令用于支持各类执行逻辑的代码。当然，目前的模型机设计方案仍然是粗糙与简陋的，例如指令集依然不够丰富、系统效率不够高、没有引入流水线等，望后来者能够在当前基础上进一步创新强化，以期获得一个适用范围更广、能力更强、效率更高的模型机系统。

最后，这次从零开始的模型机系统设计与搭建，极大地加深了我对于计算机组成原理这门课程的理解，希望在未来的研究过程中，我能学会以一种更加底层的、硬件实现的角度来审视不同的算法与代码，发现创新点，改进可优化点，以期在计算机这条漫漫长路上乐此不疲地持续前行！

9 参考文献

[1] 唐朔飞. 计算机组成原理 [M]. 高等教育出版社: 北京市,2008:1.