版本

更新记录	文档名		实验指导书 _lab2	
	版本号		0.3	
	创建人		计算机组成原理教学组	
	创建日期		2017/10/18	
更新历史				
序号	**更新日期**	**更新人**	**版本号**	**更新内容**
1	2017/10/8	吕昱峰	0.1	初版,单周期 CPU 取指译码实验
2	2018/10/8	吕昱峰	0.2	重新补充译码的理解细节。
3	2019/11/14	吕昱峰	0.3	调整存储器部分至实验 2

文档错误反馈: 本文档经多版本迭代,但由于作者精力能力有限,其中出现错误请联系:

lvyufeng@cqu.edu.cn

1 实验二　存储器与控制器实验(单周期 CPU 取指译码)

本次实验开始涉及 MIPS 架构 CPU 的设计,其中涵盖 CPU 在流水线设计中所分割的两个阶段,以下为实验概述:

MIPS 架构 CPU 的传统流程可分为取指、译码、执行、访存、回写 (Instruction Fetch, Decode, Execution, Memory Request, Write Back),五阶段。实验一完成了执行阶段的 ALU 部分,并进行了简单的访存实验,本实验将实现取指、译码两个阶段的功能。

在进行本次实验前,你需要具备以下实验环境及基础能力:

1. 了解 Xilinx Block Memory Generator IP 的使用
2. 了解数据通路、控制器的概念

1.1 实验目的

1. 了解随机存取存储器 RAM 的原理;
2. 掌握调用 Xilinx 库 IP(Block Memory Generator) 实例化 RAM 的方法;
3. 掌握单周期 CPU 各个控制信号的作用和生成过程。
4. 掌握单周期 CPU 控制器的工作原理及其设计方法。
5. 理解单周期 CPU 执行指令的过程。;
6. 掌握取指、译码阶段数据通路、控制器的执行过程。

1.2 实验设备

1. 计算机 1 台 (尽可能达到 8G 及以上内存);
2. Nexys4 DDR 实验开发板;
3. Xilinx Vivado 开发套件 (2019.1 版本)。

1.3 实验任务

1.3.1 实验要求

图 1为本次实验所需完成内容的原理图，依据取指、译码阶段的需求，分别需要实现以下模块:

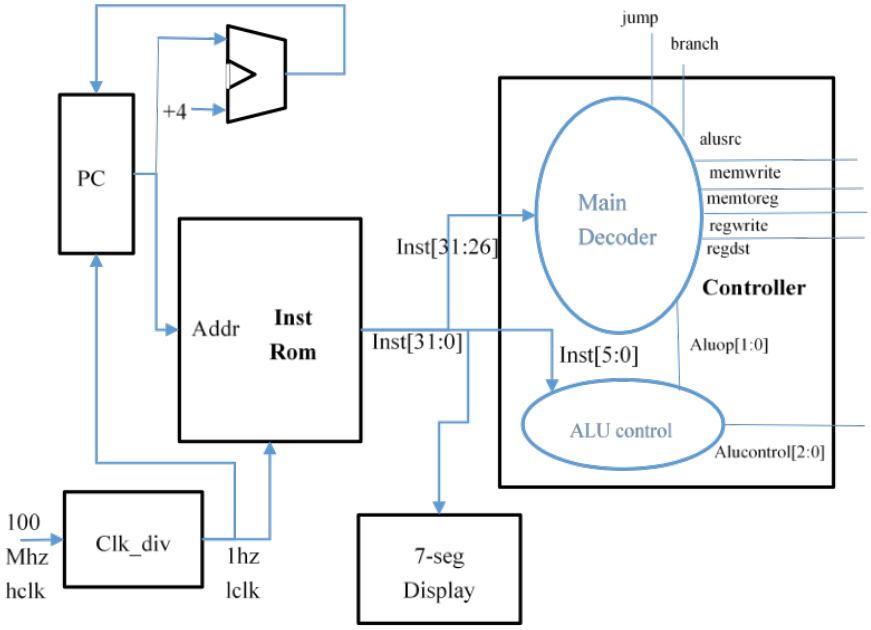

图 1: 取指译码原理图

1. PC。D 触发器结构，用于储存 PC(一个周期)。需实现 2 个输入，分别为 *clk, rst*, 分别连接时钟和复位信号；需实现 2 个输出，分别为 *pc, inst_ce*, 分别连接指令存储器的 *addra, ena* 端口。其中 *addra* 位数依据 coe 文件中指令数定义；
2. 加法器。用于计算下一条指令地址，需实现 2 个输入，1 个输出，输入值分别为当前指令地址 *PC*、*32'h4*；
3. Controller。其中包含两部分：
   (a). main_decoder。负责判断指令类型，并生成相应的控制信号。需实现 1 个输入，为指令 inst 的高 6 位 *op*，输出分为 2 部分，控制信号有多个，可作为多个输出，也作为一个多位输出，具体参照 3进行设计；*aluop*，传输至 alu_decoder，使 alu_decoder 配合 *inst* 低 6 位 *funct*，进行 ALU 模块控制信号的译码。
   (b). alu_decoder。负责 ALU 模块控制信号的译码。需实现 2 个输入，1 个输出，输入分别为 *funct, aluop*；输出位 *alucontrol* 信号。
   (c). 除上述两个组件，需设计 controller 文件调用两个 decoder，对应实现 *op*,*funct* 输入信号，并传入调用模块；对应实现控制信号及 *alucontrol*，并连接至调用模块相应端口。
4. 指令存储器。使用 Block Memory Generator IP 构造。(参考 2)
   注意：Basic 中 Generate address interface with 32 bits 选项不选中；PortA Options 中 Enable Port Type 选择为 Use ENA Pin

5. 时钟分频器。将板载 100Mhz 频率降低为 1hz，连接 PC、指令存储器时钟信号 clk。(参考数字逻辑实验)
   注意：Xilinx Clocking Wizard IP 可分的最低频率为 4.687Mhz，因而只能使用自实现分频模块进行分频

1.3.2 实验步骤

1. 创建 PC 模块
2. 创建 main_decde, alu_decode 模块
3. 创建 Controller，调用 main_decode, alu_decode
4. 使用 Block Memory，导入 coe 文件
5. 自定义顶层文件，连接相关模块
6. **自行编写 Testbench 仿真文件，时钟周期设为 10ns，每个时钟周期输入地址 Address + 4，捕获输出信号并打印在控制台。**
   打印格式:instruction: 32'h00000000, memtoreg: 1, memwrite: 0, ...

Controller 输出信号与 led 管脚对应关系如下表：

memtoreg	memwrite	pcsrc	alusrc	regdst	regwrite	jump	branch	alucontrol
[0:0]	[0:0]	[0:0]	[0:0]	[0:0]	[0:0]	[0:0]	[0:0]	[2:0]
led[0]	led[1]	led[2]	led[3]	led[4]	led[5]	led[6]	led[7]	led[8:10]

表 1: 输出信号与 led 管脚

1.4 实验环境

以下表格中红色部分需自行实现，黑色部分于实验发布包中提供。

|–top.v	设计顶层文件，参照图 1将各模块连接。
|—-pc.v	D 触发器结构。输入为下一条指令地址，输出为当前指令地址。
|—-ram.ip	RAM IP，通过 Block memory generator 进行实例化。
|—-controller.v	控制器模块，本次实验重点。
|——maindec.v	Main decoder 模块，负责译码得到各个组件的控制信号。
|——aludec.v	ALU Decoder 模块，负责译码得到 ALU 控制信号。
|—-display.v	七段数码管显示模块文件，已提供。
|——seg7.v	七段数码管显示模块组成文件，已提供。
|–constr.xdc	综合实现时，约束文件，已提供。

表 2: 实验文件树

2 调用 Xilinx 库 Block Memory Generator 方法

2.1 新建工程

新建一个工程 data_ram，参考文档“A04_Vivado 使用说明”:

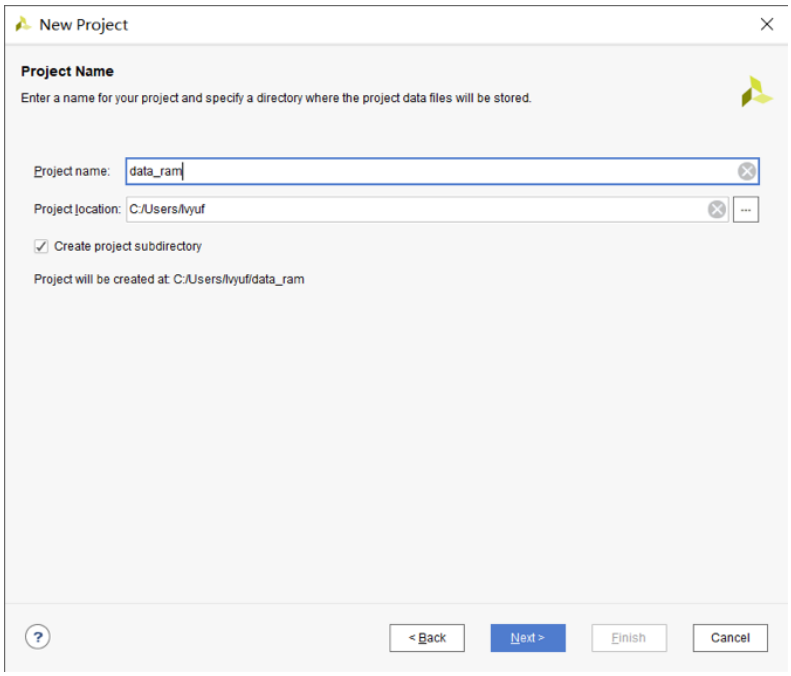

图 2: 新建工程

2.2 新建 IP

IP 核查找路径：Flow Navigator->IP Catalog->Vivado Repository ->Basic Elements->Memory Elements->Block Memory Generator

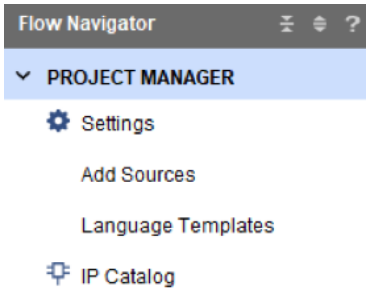

图 3: 查找 IP

或者 Flow Navigator->IP Catalog，在搜索栏直接搜索 Block Memory Generator

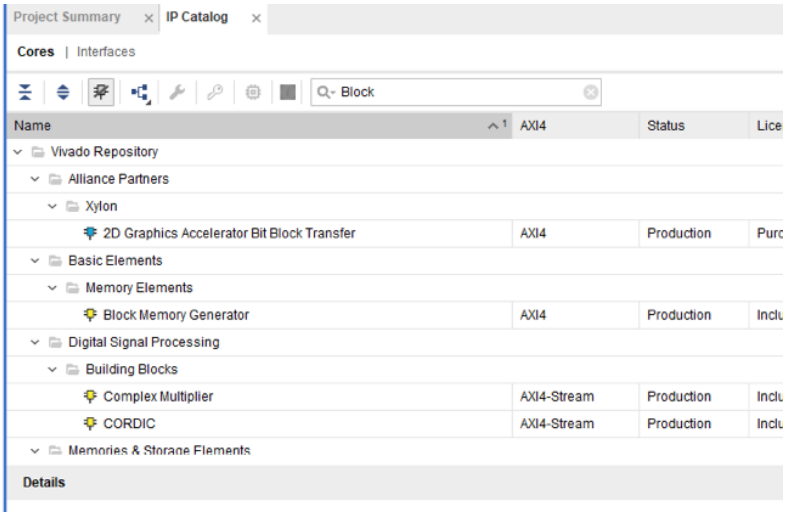

图 4: 搜索 Block Memory Generator

2.3 设置 RAM 参数

Block Memory Generator 共有四类设置，分别为 Basic、端口设置、其他设置、Summary：

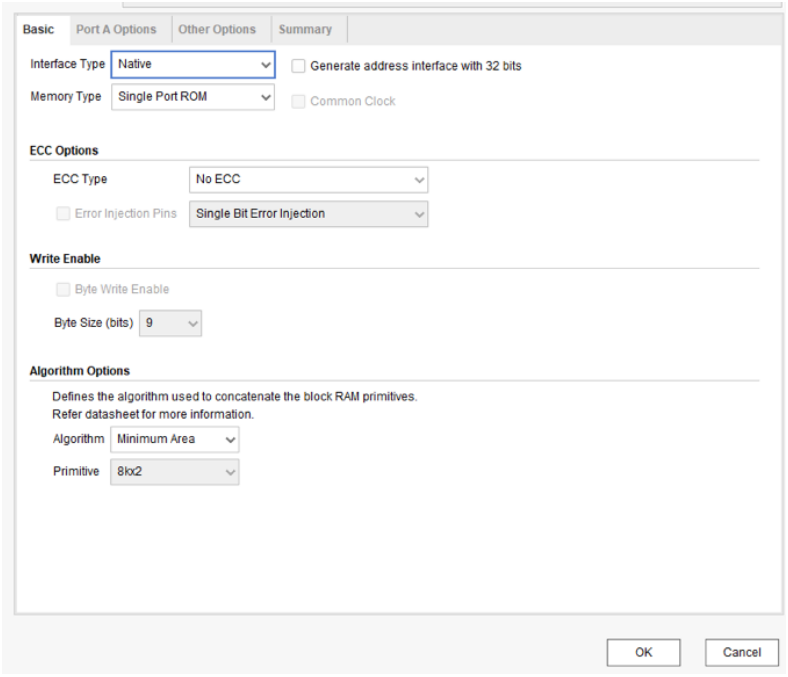

图 5: Basic

其中 Basic 需要设置存储器类型，Interface Type 需选择 Native，选中 Generate address interface with 32bits，将地址长度设置为 32 位，Memory Type 根据实验要求选择，其他选项无需设置。

端口设置需要设置数据字宽度及阵列深度，根据实验要求，字宽均为 32 位，阵列深度需根据需求自定义，但不可超过 155520 字。

写数据端口默认开启写使能，读数据端口默认不开启，可根据自己需求选择 Enable Port Type。

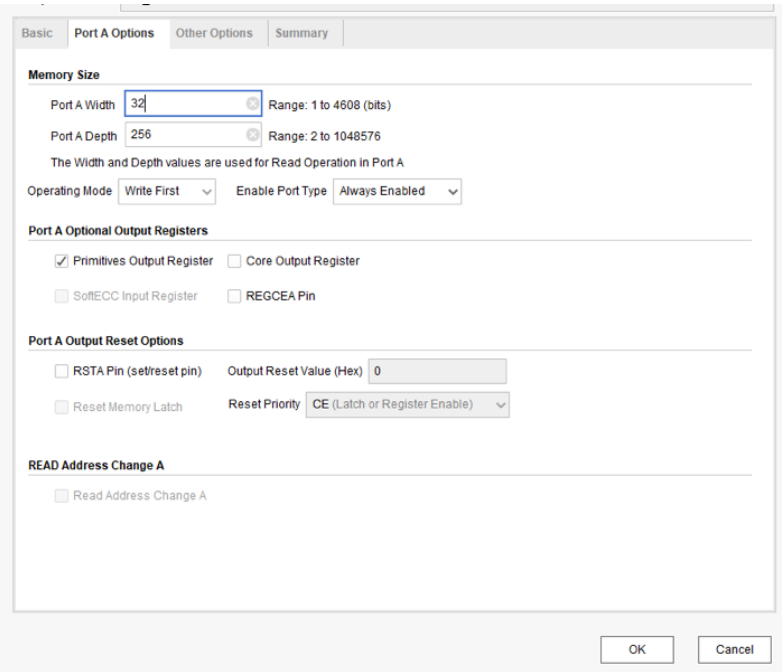

图 6: 端口设置

2.4 设置初始化数据

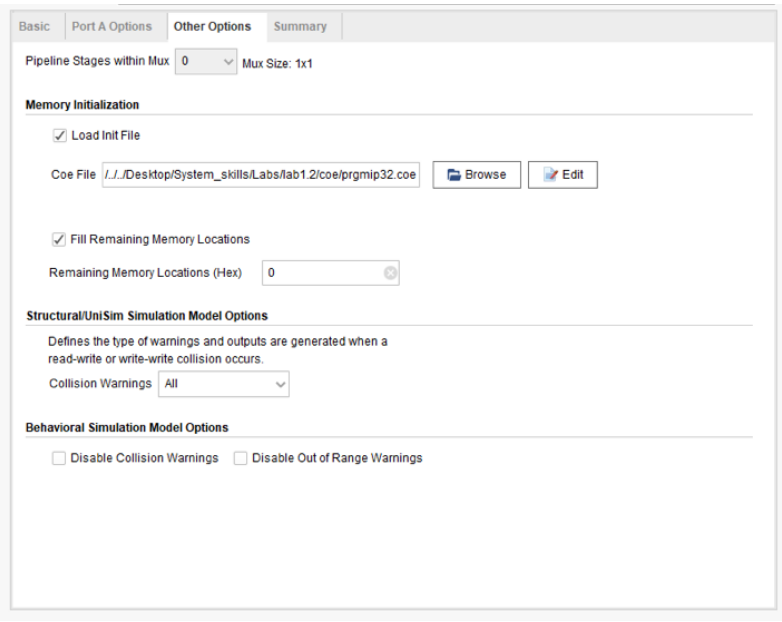

图 7: 设置初始化数据

其他设置主要用于加载 coe 文件，在图 7中，需要勾选“Load Init File”，并选中需要装载的初始化文件 (.coe 文件)。.coe 文件为 Vivado 中存储器初始化文件，其格式如下：

```
1  memory_initialization_radix = 16;
2  memory_initialization_vector =
3  24010001
4  00011100
```

5 ……

第一行指定了初始化数据格式，此处为 16 进制，也可以设置为 2 进制。第二行说明从第三行开始为初始化的数据向量，由于宽度为 32 位，故一个初始化向量为 32 位数据。初始化向量之间必须用空格或换行符隔开，此处使用换行符，故一行为一个初始化向量。初始化数据会从 RAM 中的 0 地址处开始依次填充。当初始化数据格式设置为 2 进制时，后续的初始化向量需要用二进制编写。

这里只需要注意一个问题，Fill Remaining Memory Locations 需要选中，以防读数据操作时，地址超过 coe 文件已有数据范围，导致异常。

3 实验原理

3.1 取指阶段原理

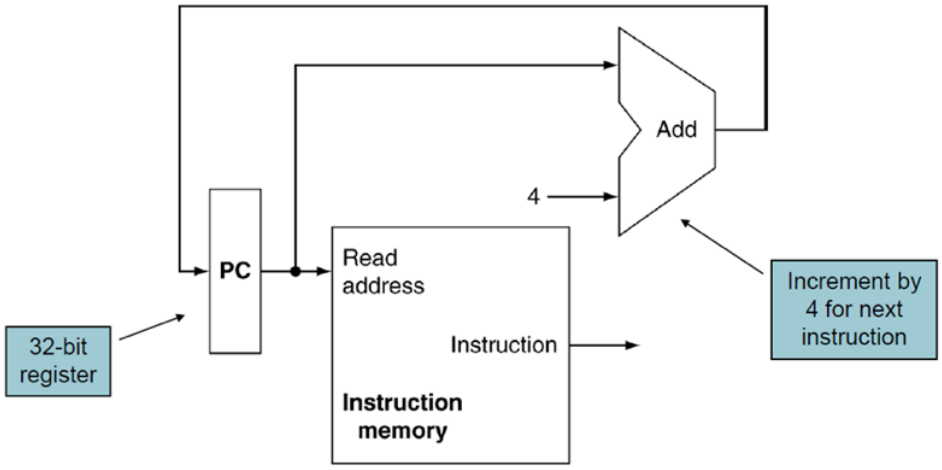

图 8: 取指示意图

如图 8所示，PC 为 32bit(1 word) 的寄存器，其存放指令地址，每条指令执行完毕后，增加 4，即为下一条指令存放地址。指令地址传入指令存储器，即可取出相应地址存放的指令。

需要注意的是，MIPS 架构中，采用字节读写，1 32bit word = 4 byte，故需要地址 +4 来获取下一条指令。

3.2 指令译码原理

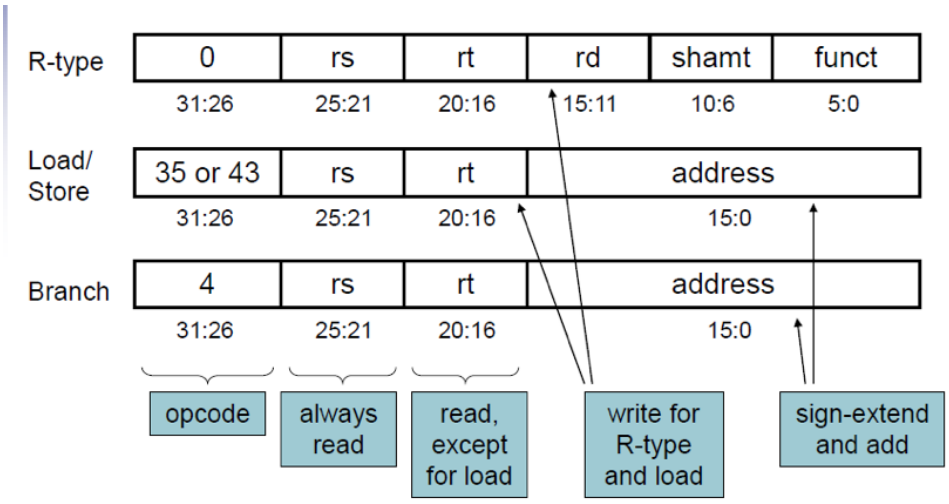

图 9: MIPS 指令原理图

如图 9所示，32 位 MIPS 指令在不同类型指令中分别有不同结构。但 [31:16] 表示的 opcode，

以及 [5:0] 表示的 funct，为译码阶段明确指令控制信号的主要字段。表 3为 opcode 及 funct 识别得到的部分信号，详细信号表参照课本及课堂 Slides。

opcode	aluop	operation	funct	alu function	alu control
lw	00	Load word	XXXXXX	Add	010
sw	00	Store word	XXXXXX	Add	010
beq	01	Branch equal	XXXXXX	Subtact	110
R-type	10	Add	100000	qdd	010
		subtract	100010	Subtract	110
		and	100100	And	000
		or	100101	Or	001
		set-on-less-than	101010	SLT	111

表 3: 译码控制信号

3.3 控制器实现原理

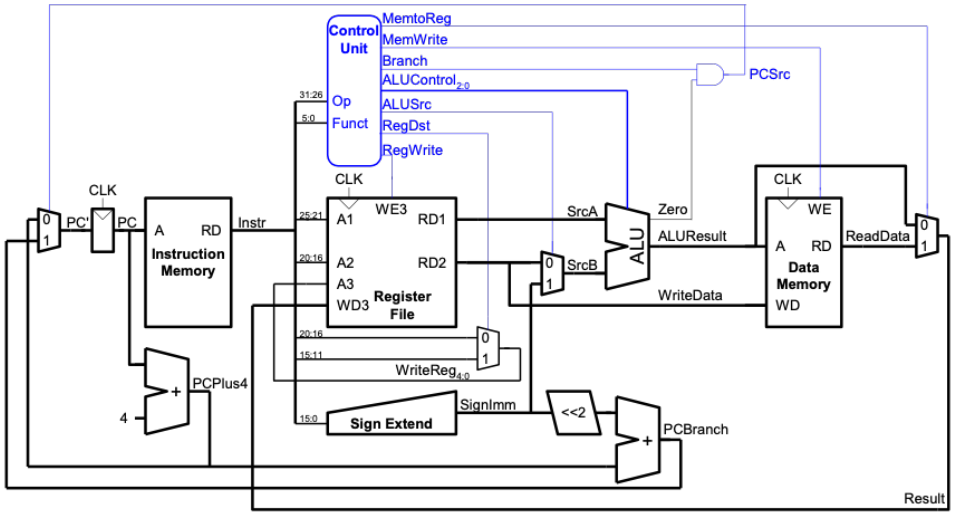

图 10: 单周期数据通路

由图 10可知，控制器输出的控制信号，用于控制器件的使能和多路选择器的选择，因此，根据不同指令的功能分析其所需要的路径，即可得到信号所对应的值。在此之前，参照表 4对各个控制信号的含义进行理解。

信号	含义
memtoreg	回写的数据来自于 ALU 计算的结果/存储器读取的数据
memwrite	是否需要写数据存储器
pcsrc	下一个 PC 值是 PC+4/跳转的新地址
alusrc	送入 ALU B 端口的值是立即数的 32 位扩展/寄存器堆读取的值
regdst	写入寄存器堆的地址是 rt 还是 rd,0 为 rt,1 为 rd
regwrite	是否需要写寄存器堆
branch	是否为 branch 指令,且满足 branch 的条件
jump	是否为 jump 指令
alucontrol	ALU 控制信号,代表不同的运算类型

表 4: 信号含义

分析数据通路图,判断指令是否需要写寄存器、访存等等操作,以产生相应的控制信号。下面给出参考信号表:

instruction	op5:0	regwrite	regdst	alusrc	branch	memWrite	memtoReg	aluop1:0
R-type	000000	1	1	0	0	0	0	10
lw	100011	1	0	1	0	0	1	00
sw	101011	0	X	1	0	1	X	00
beq	000100	0	X	0	1	0	X	01
addi	001000	1	0	1	0	0	0	00
j	000010	0	X	X	X	0	X	XX

表 5: 控制信号译码