

版本

更新记录	文档名	实验指导书_Lab2		
	版本号	0.2		
	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	2017/10/18	吕昱峰	0.1	初版，单周期CPU取指译码实验
2	2018/10/18	吕昱峰	0.2	重新补充译码的理解细节。

文档错误反馈: lvyufeng@cqu.edu.cn

1. 实验二 单周期 CPU 取指译码实验

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，你需要具备以下基础能力：

- 1) 熟悉 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. 掌握单周期 CPU 控制器的工作原理及其设计方法。
2. 掌握单周期 CPU 各个控制信号的作用和生成过程。
3. 理解单周期 CPU 执行指令的过程。
4. 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(2018.1 版本)。

1.3 实验任务

1.3.1 实验要求

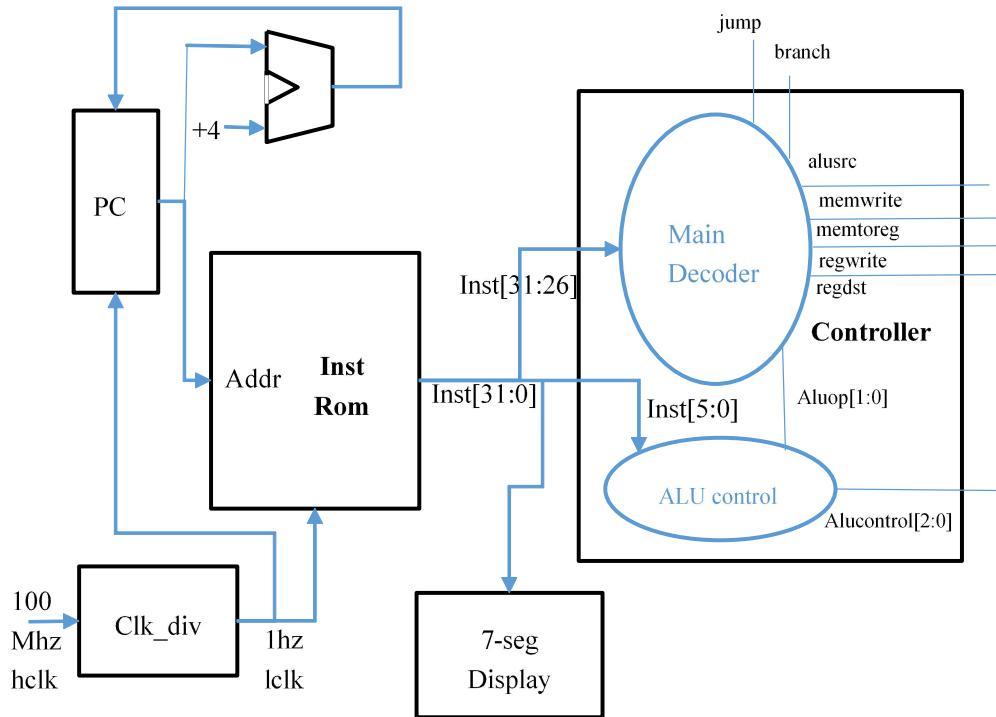


图 1.1 为本次实验所需完成内容的原理图，依据取指、译码阶段的需求，分别需要实现以下模块：

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 *addr*, *ena* 端口。其中 *addr* 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，具体参照指导书第二部分进行设计；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。(参考实验一)
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

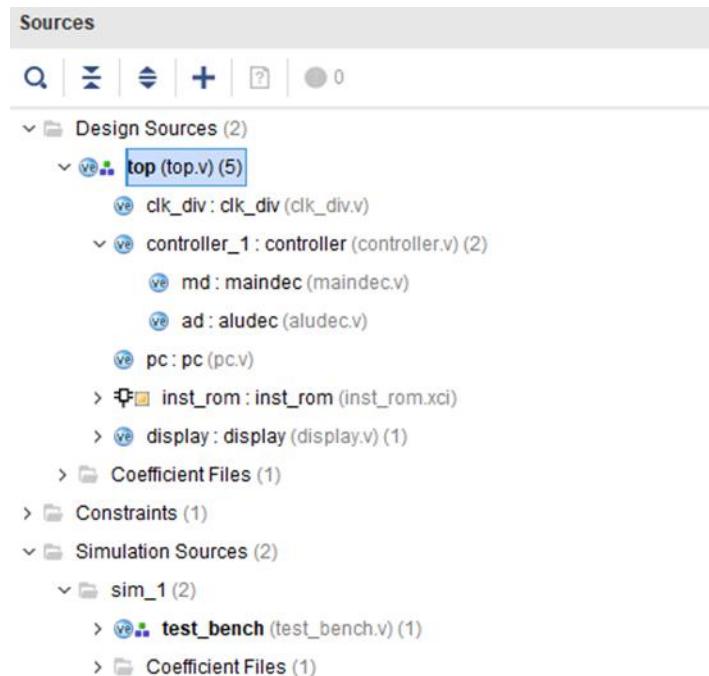
1. 从实验一、数字逻辑课程实验中，导入 Display、clk_div 模块
2. 创建 PC 模块
3. 创建 main_decode, alu_decode 模块
4. 创建 Controller，调用 main_decode, alu_decode
5. 使用 Block Memory，导入 coe 文件
6. 自定义顶层文件，连接相关模块
7. Controller 输出信号与 led 管脚对应关系如下表：

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]

1.4 实验环境

--top.v	设计顶层文件，参照图 1.1 将各模块连接。
-----clk_div.v	时钟分频模块，需将 100MHz 的时钟频率降低至 1Hz
-----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。

实验完成可得到与下图类似的目录结构



2. 实验原理

2.1 取指阶段原理

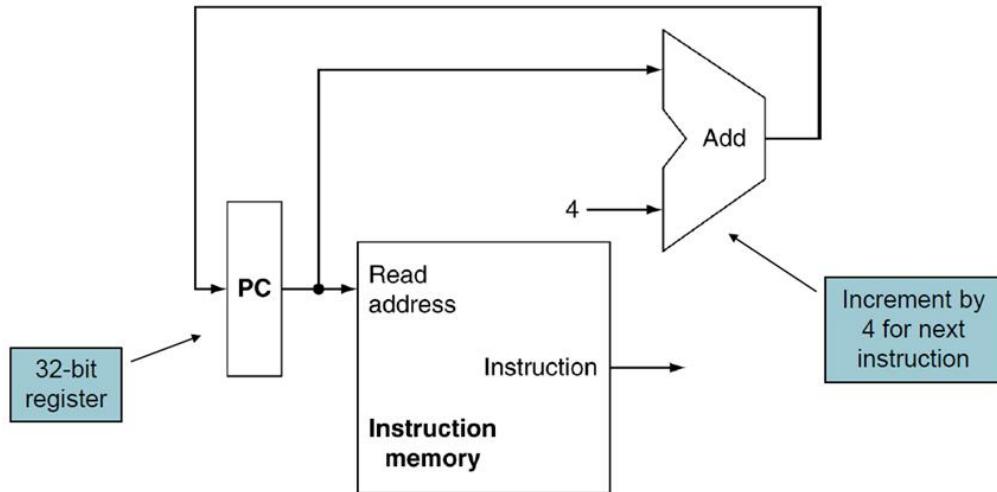


图 2.2

如图 2.2 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， $1 \text{ 32bit word} = 4 \text{ byte}$ ，故需要地址+4 来获取下一条指令。

2.2 指令译码原理

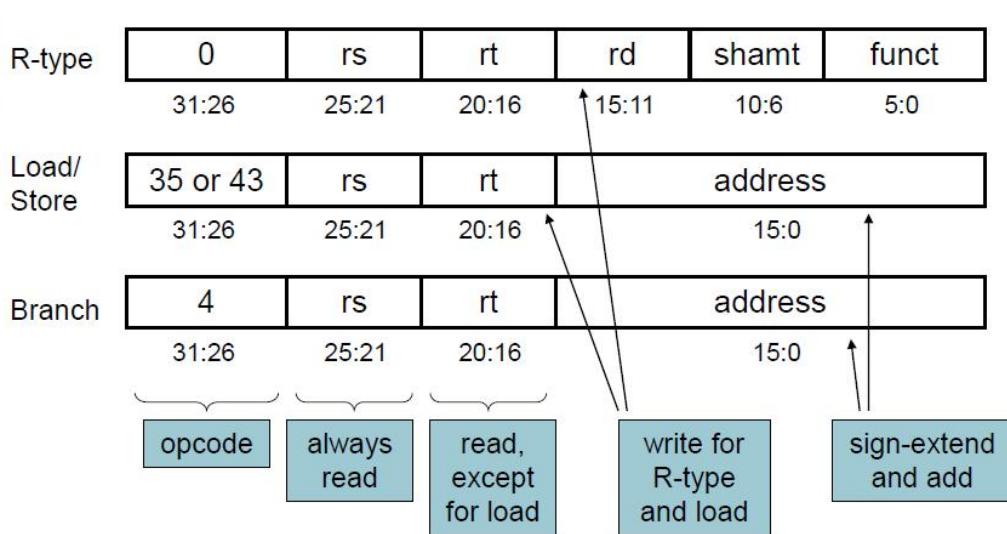


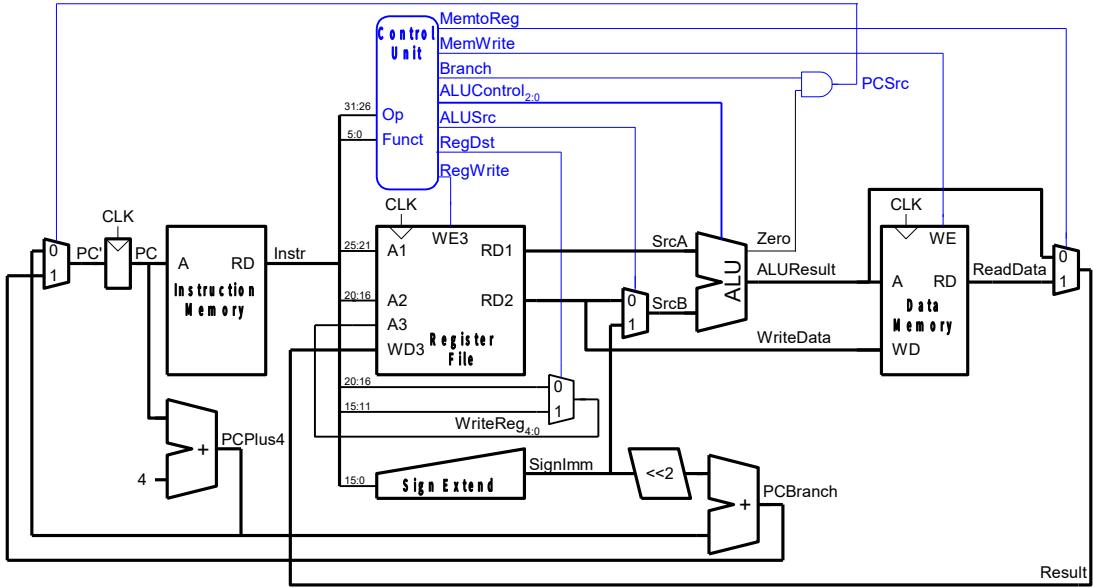
图 2.3

如图 2.3 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。下表为 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。

表 2.2

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	Subtract	110
R-type	10	Add	100000	Add	010
		Subtract	100010	Subtract	110
		And	100100	And	000
		Or	100101	Or	001
		Set-on-less-than	101010	SLT	111

2.3 控制器实现原理



由上图可知，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，参照下表对各个控制信号的含义进行理解。

表 2.3

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

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	Op _{5:0}	RegWrite	RegDst	AluSrc	Branch	MemWrite	MemtoReg	ALUOp _{1:0}
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	000100	0	X	X	X	0	X	XX

3. 附录 A

实验所附的 coe 文件中所有指令均包含于下表中，可供查询 opcode 及 funct 所代表的具体指令。

表 3.1 MIPS 的 31 种指令

助记符	指令格式						示例	示例含义	操作及解释
BIT #	31..26	25..21	20..16	15..11	10..6	5..0			
R-类型	op	rs	rt	rd	shamt	func			
add	000000	rs	rt	rd	00000	100000	add \$1,\$2,\$3	\$1=\$2+S3	(rd)←(rs)+(rt); rs=\$2,rt=\$3,rd=\$1
addu	000000	rs	rt	rd	00000	100001	addu \$1,\$2,\$3	\$1=\$2+S3	(rd)←(rs)+(rt); rs=\$2,rt=\$3,rd=\$1,无符号数
sub	000000	rs	rt	rd	00000	100010	sub \$1,\$2,\$3	\$1=\$2-S3	(rd)←(rs)-(rt); rs=\$2,rt=\$3,rd=\$1
subu	000000	rs	rt	rd	00000	100011	subu \$1,\$2,\$3	\$1=\$2-S3	(rd)←(rs)-(rt); rs=\$2,rt=\$3,rd=\$1,无符号数
and	000000	rs	rt	rd	00000	100100	and \$1,\$2,\$3	\$1=\$2&S3	(rd)←(rs)&(rt); rs=\$2,rt=\$3,rd=\$1
or	000000	rs	rt	rd	00000	100101	or \$1,\$2,\$3	\$1=\$2 S3	(rd)←(rs) (rt); rs=\$2,rt=\$3,rd=\$1
xor	000000	rs	rt	rd	00000	100110	xor \$1,\$2,\$3	\$1=\$2^S3	(rd)←(rs)^ (rt); rs=\$2,rt=\$3,rd=\$1
nor	000000	rs	rt	rd	00000	100111	nor \$1,\$2,\$3	\$1= ~(\$2 S3)	(rd)←~((rs) (rt)); rs=\$2,rt=\$3,rd=\$1
slt	000000	rs	rt	rd	00000	101010	slt \$1,\$2,\$3	if(\$2<\$3) \$1=1 else \$1=0	if (rs< rt) rd=1 else rd=0;rs=\$2, rt=\$3, rd=\$1
sltu	000000	rs	rt	rd	00000	101011	sltu \$1,\$2,\$3	if(\$2<\$3) \$1=1 else \$1=0	if (rs< rt) rd=1 else rd=0;rs=\$2, rt=\$3, rd=\$1,无符号数
sll	000000	00000	rt	rd	shamt	000000	sll \$1,\$2,10	\$1=\$2<<10	(rd)←(rt)<<shamt,rt=\$2,rd=\$1,shamt=10
srl	000000	00000	rt	rd	shamt	000010	srl \$1,\$2,10	\$1=\$2>>10	(rd)←(rt)>>shamt, rt=\$2, rd=\$1, shamt=10,(逻辑右移)
sra	000000	00000	rt	rd	shamt	000011	sra \$1,\$2,10	\$1=\$2>>10	(rd)←(rt)>>shamt, rt=\$2, rd=\$1, shamt=10,(算术右移, 注意符号位保留)
sllv	000000	rs	rt	rd	00000	000100	sllv \$1,\$2,\$3	\$1=\$2<<\$3	(rd)←(rt)<<(rs), rs=\$3,rt=\$2,rd=\$1
srlv	000000	rs	rt	rd	00000	000110	srlv \$1,\$2,\$3	\$1=\$2>>\$3	(rd)←(rt)>>(rs), rs=\$3,rt=\$2,rd=\$1,(逻辑右移)
srav	000000	rs	rt	rd	00000	000111	srav \$1,\$2,\$3	\$1=\$2>>\$3	(rd)←(rt)>>(rs), rs=\$3,rt=\$2,rd=\$1,(算术右移,注意符号位保留)
jr	000000	rs	00000	00000	00000	001000	jr \$31	goto \$31	(PC)←(rs)
I-类型	op	rs	rt	immediate					
addi	001000	rs	rt	immediate			addi \$1,\$2,10	\$1=\$2+10	(rt)←(rs)+(sign-extend)immediate,rt=\$1,rs=\$2
addiu	001001	rs	rt	immediate			addiu \$1,\$2,10	\$1=\$2+10	(rt)←(rs)+(sign-extend)immediate,rt=\$1,rs=\$2
andi	001100	rs	rt	immediate			andi \$1,\$2,10	\$1=\$2&10	(rt)←(rs)&(zero-extend)immediate,rt=\$1,rs=\$2
ori	001101	rs	rt	immediate			ori \$1,\$2,10	\$1=\$2 10	(rt)←(rs) (zero-extend)immediate,rt=\$1,rs=\$2
xori	001110	rs	rt	immediate			xori \$1,\$2,10	\$1=\$2^10	(rt)←(rs)^ (zero-extend)immediate,rt=\$1,rs=\$2
lui	001111	00000	rt	immediate			lui \$1,10	\$1=10*65536	(rt)←immediate<<16 & 0FFFF0000H, 将 16 位立即数放到目的寄存器高 16 位, 目的寄存器的低 16 位填 0
lw	100011	rs	rt	offset			lw \$1,10(\$2)	\$1=Memory[\$2+10]	(rt)←Memory[(rs)+(sign_extend)offset],rt=\$1,rs=\$2
sw	101011	rs	rt	offset			sw \$1,10(\$2)	Memory[\$2+10]=\$1	Memory[(rs)+(sign_extend)offset]←(rt),rt=\$1,rs=\$2
beq	000100	rs	rt	offset			beq \$1,\$2,40	if(\$1==\$2) goto PC+4+40	if ((rt)==(rs)) then (PC)←(PC)+4+(Sign-Extend) offset<<2), rs=\$1, rt=\$2

bne	000101	rs	rt	offset	bne \$1,\$2,40 goto PC+4+40	if(\$1≠\$2) \$1=1 else \$1=0	if ((rt)≠(rs)) then (PC)←(PC)+4+((Sign-Extend) offset<<2) , rs=\$1, rt=\$2
slti	001010	rs	rt	immediate	slti \$1,\$2,10	if(\$2<10) \$1=1 else \$1=0	if ((rs)<(Sign-Extend)immediate) then (rt)←1; else (rt)←0, rs=\$2, rt=\$1
sltiu	001011	rs	rt	immediate	sltiu \$1,\$2,10	if(\$2<10) \$1=1 else \$1=0	if ((rs)<(Zero-Extend)immediate) then (rt)←1; else (rt)←0, rs=\$2, rt=\$1
J-类型	op	address					
j	000010	address		j 10000	goto 10000	(PC)←((Zero-Extend) address<<2), address=10000/4	
jal	000011	address		jal 10000	\$31=PC+4 goto 10000	(\$31)←(PC)+4; (PC)←((Zero-Extend) address<<2), address=10000/4	