1 实验三　简易单周期 CPU 实验

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 ALU 设计并掌握了存储器 IP 的使用；实验二实现了单周期 CPU 的取指、译码阶段，完成了 PC、控制器的设计。在实验一与实验二的基础上，单周期 CPU 的设计的各模块已经具备，再引入数字逻辑课程中所实现的多路选择器、加法器等门级组件，通过对原理图的理解，分析单条（单类型）指令在数据通路中的执行路径，依次连接对应端口，即可完成单周期 CPU。

在进行本次实验前，你需要具备以下基础能力：

1. 熟悉 Vivado 的仿真功能 (行为仿真)
2. 理解数据通路、控制器的信号

1.1 实验目的

1. 掌握不同类型指令在数据通路中的执行路径。
2. 掌握 Vivado 仿真方式。

1.2 实验设备

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

注：本次实验为 CPU 软核实验，不涉及开发板外围设备，故不需要开发板

1.3 实验任务

1.3.1 实验要求

阅读实验原理实现以下模块：

(1) Datapath，其中主要包含 alu(实验一已完成)，PC(实验二已完成)，adder、mux2、signext、sl2(其中 adder、mux2 数字逻辑课程已实现，signext、sl2 参见实验原理)，
(2) Controller(实验二已完成)，其中包含两部分，分别为 main_decoder，alu_decoder。
(3) 指令存储器 inst_mem(Single Port Ram)，数据存储器 data_mem(Single Port Ram)；使用 Block Memory Generator IP 构造指令，注意考虑 PC 地址位数统一。(参考实验二)
(4) 参照实验原理，将上述模块依指令执行顺序连接。实验给出 top 文件，需兼容 top 文件端口设定。
(5) 实验给出仿真程序，最终以仿真输出结果判断是否成功实现要求指令。

1.3.2 实验步骤

1. 从实验一中，导入 alu 模块；
2. 从实验二中导入 PC、Controller 模块；
3. 从数字逻辑实验中导入多路选择器、加法器模块；
4. 使用 Block Memory，其中 inst_mem 导入 coe 文件；
5. 参考实验原理，连接各模块；
6. 导入顶层文件及仿真文件，运行仿真；

1.4 实验环境

|–top.v	设计顶层文件，已提供。
|—-mips.v	MIPS 软核顶层文件，将 Controller 与 Datapath 连接，已提供
|———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。
|———datapath.v	数据通路模块，自行实现
|———pc.v	PC 模块，使用实验二代码
|———alu.v	ALU 模块，使用实验一代码
|———sl2.v	移位模块，参考《其他组件实现.pdf》
|———signext.v	有符号扩展模块，参考《其他组件实现.pdf》
|———mux2.v	二选一选择器，自行实现
|———regfile.v	寄存器堆，已提供
|———adder.v	加法器，已提供
|—-*inst_ram.ip*	*RAM IP*，通过 *Block memory generator* 进行实例化
|—-*data_ram.ip*	*RAM IP*，通过 *Block memory generator* 进行实例化

表 1: 实验文件树

2 实验原理

2.1 总体框架及通路图

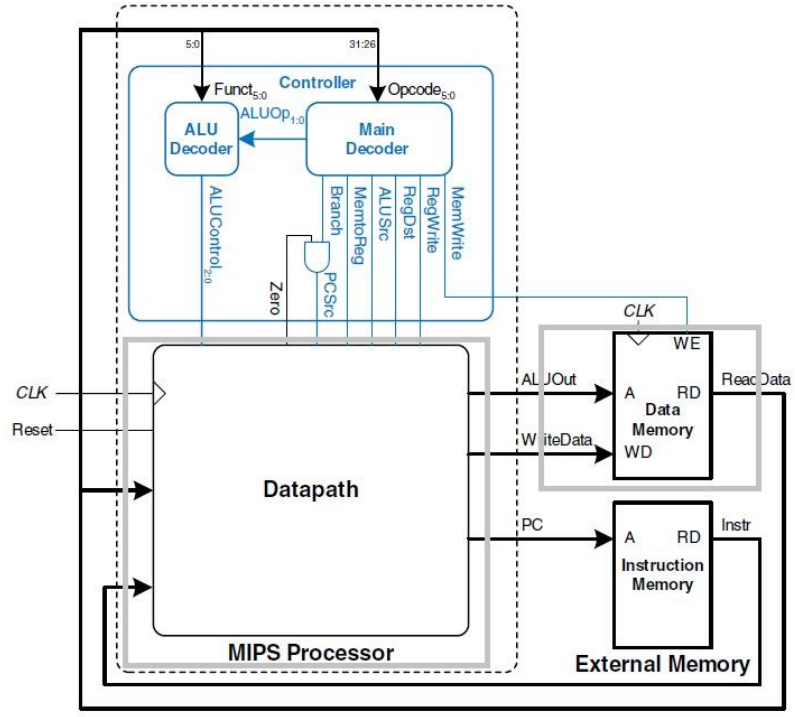

图 1: 单周期 CPU 框架图

如图 1，完整的单周期 CPU 实现框架图。其中灰色线框外的部分已在实验二中完成，继续完成 datapath，即可将单周期 MIPS 软核完整实现。

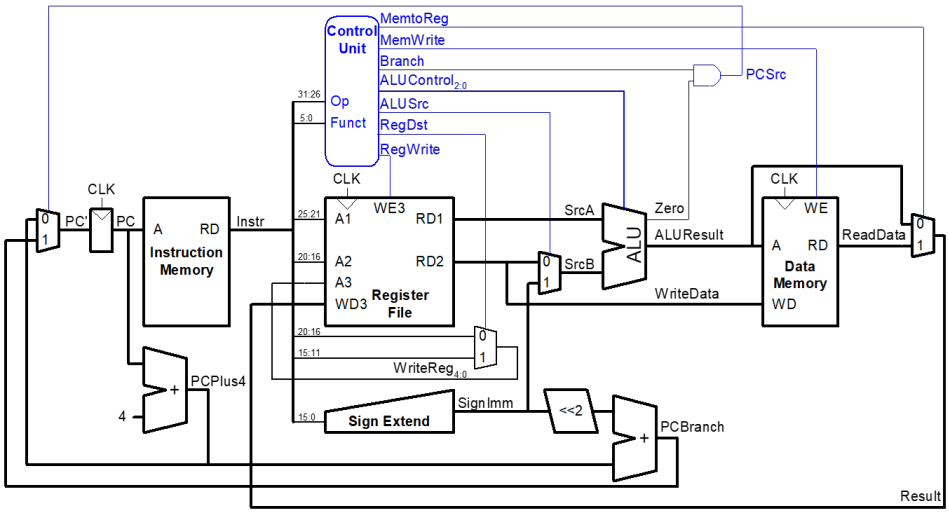

图 2: 单周期 CPU 框架图

图 2为单周期的完整通路图，涵盖 *add*、*and*、*sub*、*or*、*slt*、*beq*、*j*、*lw*、*sw*、*addi* 等指令。实验三仅实现上述指令，完整的 MIPS 指令集将与硬件综合设计中完善，此处以掌握数据通路分析为主要

目的。

2.2 控制器译码信号规范

2.2.1 ALU 控制码译码

由于控制器的设计在实验二中并不进行强制限制，因此 ALUOp 信号在部分设计中可能不会存在，只需按照表 2中其他信号即可

ALUOp[1:0]	Funct	ALUControl[2:0]
00	X(for lw)	010 (Add)
01	X(for beq)	110 (Subtract)
10	100000 (add)	010 (Add)
10	100010 (sub)	110 (Subtract)
10	100100 (and)	000 (And)
10	100101 (or)	001 (Or)
10	101010 (slt)	111 (SLT)

表 2: ALU 控制码译码信号表

这里需要注意，不按照表中的信号也可，只要保证不同运算的 alucontrol 信号不同即可。

2.2.2 信号控制码译码信号

信号控制码译码信号与实验二相同，在通路连接时分别接入到需要控制的端口。

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	000010	0	X	X	X	0	X	XX

表 3: 控制信号译码

2.3 数据通路连接

在进行数据通路连接前，除了三大基本器件，还有些许小器件需要实现，包括加法器、触发器、多路选择器、移位器、符号扩展器件等。这些器件多为简单的组合逻辑，在数字逻辑课程已有涉及，此处不再赘述，具体实现参见附录。

2.3.1 LW 指令

以 LW 指令为例构建基本的单周期 CPU 数据通路，需要的基本器件有 PC、Regfile、存储器，见图 3，其他小的组合逻辑部件根据需求进行添加。

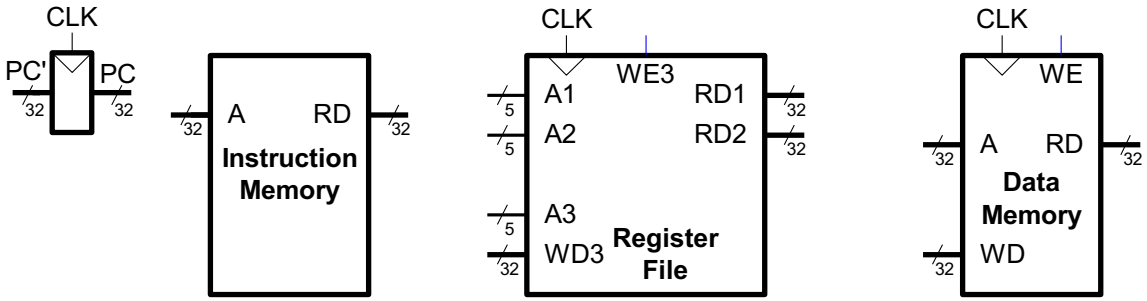

图 3

第一步为取值，将 PC(即指令地址) 输出至 Instruction Memory(指令存储器) 的 address 端口，如图 4。

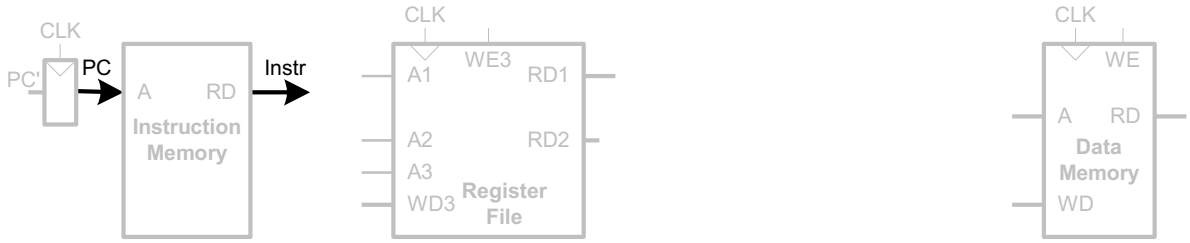

图 4

LW 指令的汇编格式为：LW rt,offset(base)，其中 base(基地址) 为指令 [25:21] 所指向的寄存器值，offset(地址偏移) 为指令 [15:0]。将 base 寄存器的值加上符号扩展后的立即数 offset 得到访存的地址，根据地址从存储器中读取 1 个字 (连续 4 个字节) 的值写入到 rt 寄存器中。

根据 LW 指令的定义，将指令存储器读出的指令 ([31:0]) 中 [25:21] 连接至 regfile 寄存器堆的第一个输入地址，即 Address1(A1)。

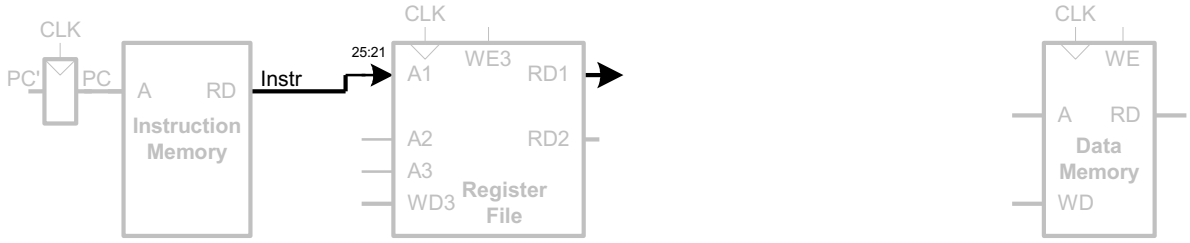

图 5

除此之外需要将 offset 进行扩展，因而将指令 [15:0] 传至有符号扩展模块，输出 32 位的符号扩展信号 (SignImm)。

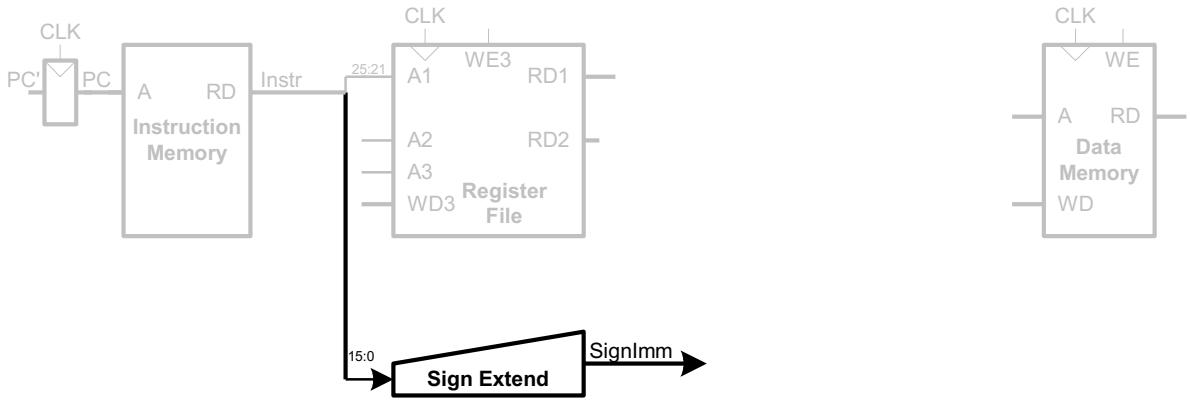

图 6

得到基地址 base 和 offset 后，需进行加法计算，其操作可以采用如下描述：Addr GPR[base] + sign_extend(offset)。加法计算使用 ALU 中设计实现的加法运算，因而图 7中，RD1 读出的 GPR[base] 和经过有符号扩展后的 sign_extend(offset) 分别作为 ALU 的两个输入 (SrcA、SrcB)，经过 ALU 进行加法运算后，得到 ALUResult 作为地址，输入到数据存储器 Data Memory 的 Address 端口。

这里需要注意的是，由于计算地址与进行加法运算相同，所以用于控制 ALU 运算类型的信号 ALUControl 与加法运算应该相同，如图 7中蓝色部分所示。AlUControl 信号由 Controller 译码得到，在此不再赘述。

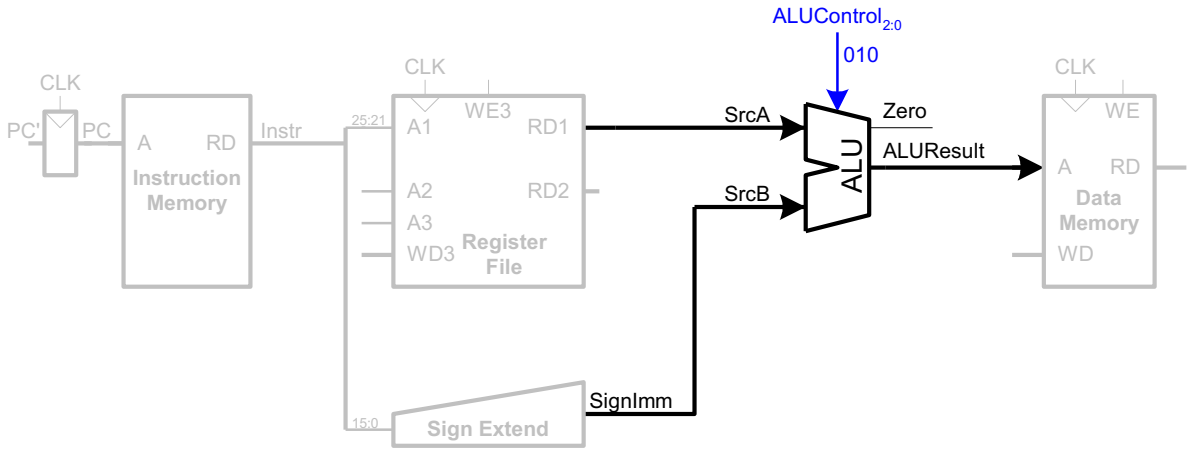

图 7

将地址输入后，将数据存储器读出的数据写回到 regfile 中，其地址为 rt，即指令的 [20:16]。如图 8，连接时需要将指令 [20:16] 连接至寄存器堆的 Address3(A3) 端口，对应的数据信号 ReadData 连接至 WriteData3(WD3) 端口。

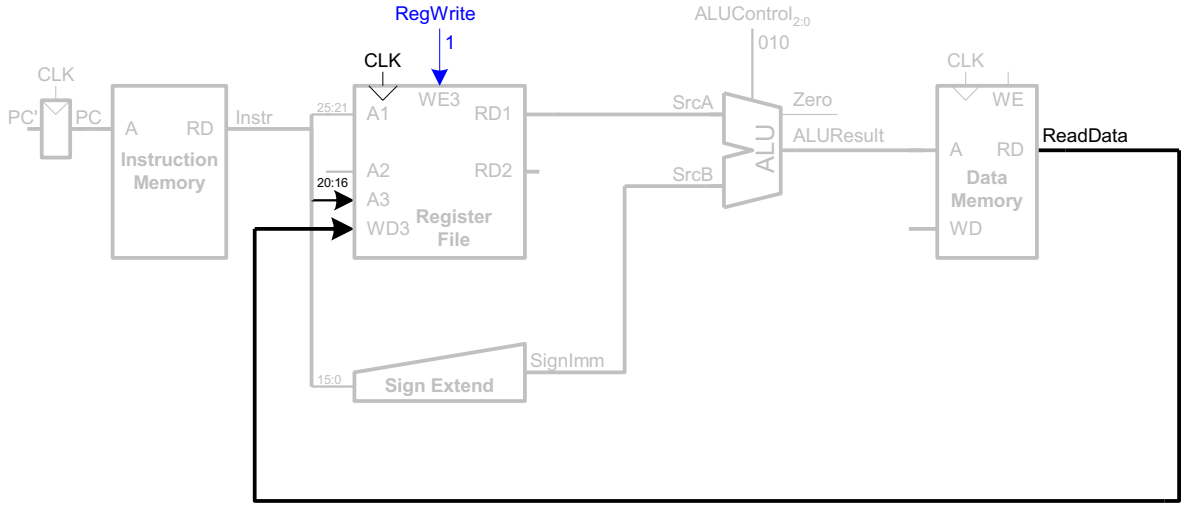

图 8

完成上述连接后，一条能够满足 LW 指令执行的数据通路即完成。LW 指令执行结束后，需开始下一条指令的执行，重复同样的执行过程。唯一的不同在于 PC 地址需要变为下一条指令所在地址。由于实现的是 32 位 MIPS 指令集，并且采用字节读写的方式，因为需要读取后续 4 个字节的数据，即 PC+4。将得到的 PC+4 信号写入 PC(D 触发器) 的输入端，即可实现每周期地址 +4 的操作。

注意：这里我们不采用很多参考书籍中复用 ALU 的方式计算加法，而是使用一个单独的加法器模块，如图 9，原因在于，ALU 在实现多种运算逻辑后，其组合逻辑更加复杂，复用 ALU 进行 PC+4 的运算需要控制信号、输入的多路选择，增加了设计的复杂性的同时，也不符合硬件电路设计的思维。在电路设计中的复用更多是出于简化设计、优化延迟、降低功耗等目的，而非单一功能完成后进行复用的软件设计思维。

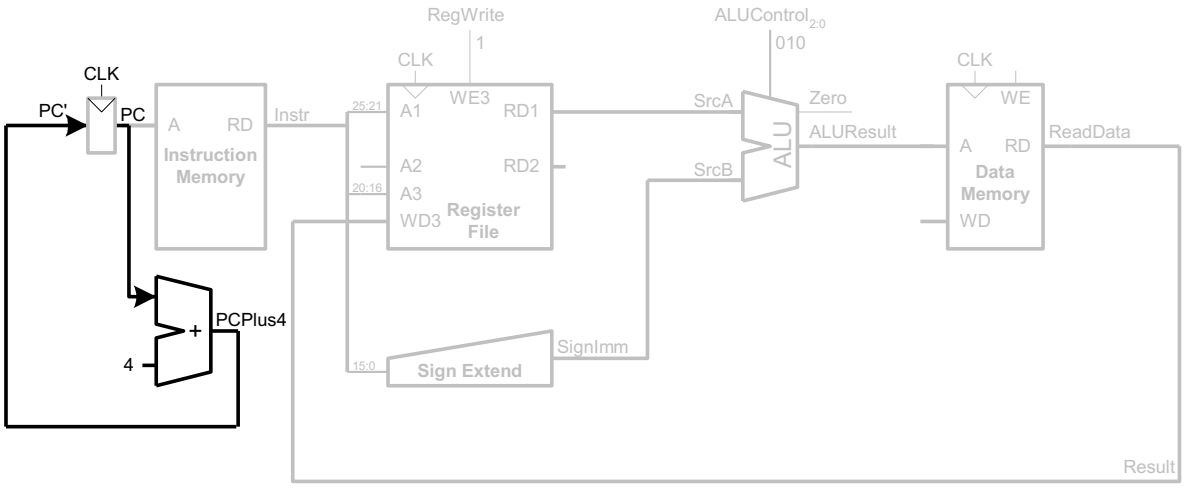

图 9

2.3.2 SW 指令

在完成 LW 指令的数据通路后，SW 指令仅需进行些许改动即可。首先来看 SW 指令的定义：SW rt,offset(base)。从指令格式和译码的数据域来看，与 LW 无异。其区别在于，LW 需要从数据存储器读取数据并写入 regfile，而 SW 仅需要将数据写入数据存储器。计算地址的 GPR[base]+sign_extend(offset) 不变，但 rt 变为读取寄存器，因而需要将其连接至 Address2(A2)，读出的数据 RD2 信号连接至数据存储器的 WD 端口。

这里涉及到几个控制信号的值，分别为 MemWrite 和 RegWrite，均连接至写使能端口 WE，值为 1 时使能，为 0 时不使能。如图 10所示，在进行存储器读写时，MemWrite 需要使能，置 1。虽然此时执行的是 sw 写操作，但是 ReadData 信号会是什么值？这取决于使用的 Memory 类型，假设会有随机值读出，A3(write address) 同时将指令的 [20:16] 写入，这时将会有错误的值写到寄存器堆中。为了避免这种情况，需要将 RegWrite 置为 0，此时无法对寄存器堆进行写操作。这里进行延伸，所有不对寄存器堆进行写操作的指令，都可能会有类似的情况发生，因此这种类型的指令都应该将 regfile 的写使能关闭。同理，所有不对指令寄存器进行读写的指令，也应当关闭其使能端口，避免意外情况的出现。

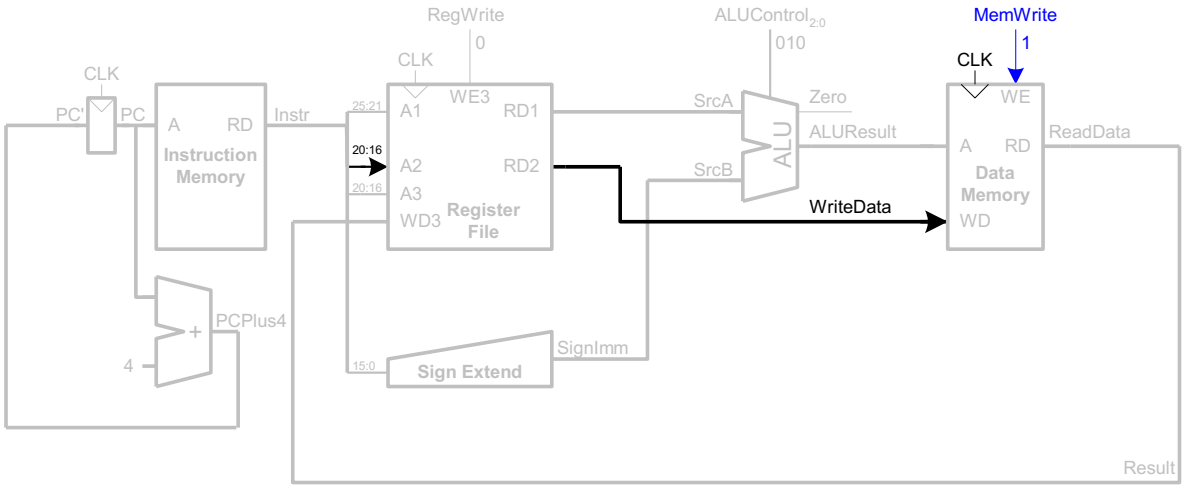

图 10

2.3.3 R-Type 指令

R-Type 是 MIPS 指令中一大类指令的合集，均为 XXX rs,rt,rd 形式的指令。除了指令进行的运算类型不同外，其操作均为将 rs,rt 所在寄存器的值进行相应运算后，存入 rd 中。因而，对已经满足 lw、sw 的数据通路进行一定改造即可。

通路改造方法为添加多路选择器。lw 指令写入 regfile 的地址为 rt，而 R-Type 为 rd，在此处加入一个多路选择器，输入分别为指令的 [20:16] 和 [15:11]，使用 RegDst(register destination) 信号控制，多路选择其输出信号命名为 WriteReg(write register)。ALU 输入为 rs,rt，分别对应 srcA,srcB，因此需要在 srcB 处加入多路选择器，选择来源为 RD2 或立即数 SignImm，控制信号为 ALUSrc(ALU source)。最后写回到 regfile 的值应该为 ALU 计算得到的值，为 ALUResult，加入多路选择器控制

Result 来源为 ALU 或数据存储器，控制信号为 MemtoReg(Memory to regfile)。

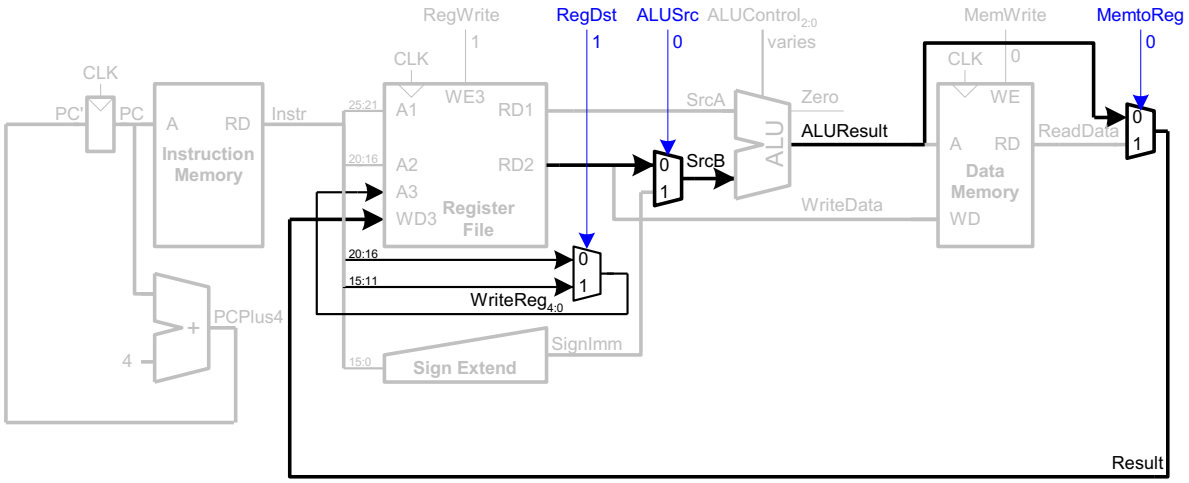

图 11

2.3.4 Branch 指令

接下来添加分支指令，即通过条件判断是否需要跳转。此处以 beq 指令为例。其定义为：BEQ rs, rt, offset。如果寄存器 rs 的值等于寄存器 rt 的值则转移，否则顺序执行。转移目标由立即数 offset 左移 2 位并进行有符号扩展的值加上该分支指令对应的延迟槽指令的 PC计算得到。

这里暂时无须考虑延迟槽的概念，只需要知道其地址为 PC+4，即当前指令的下一条指令所在的地址。

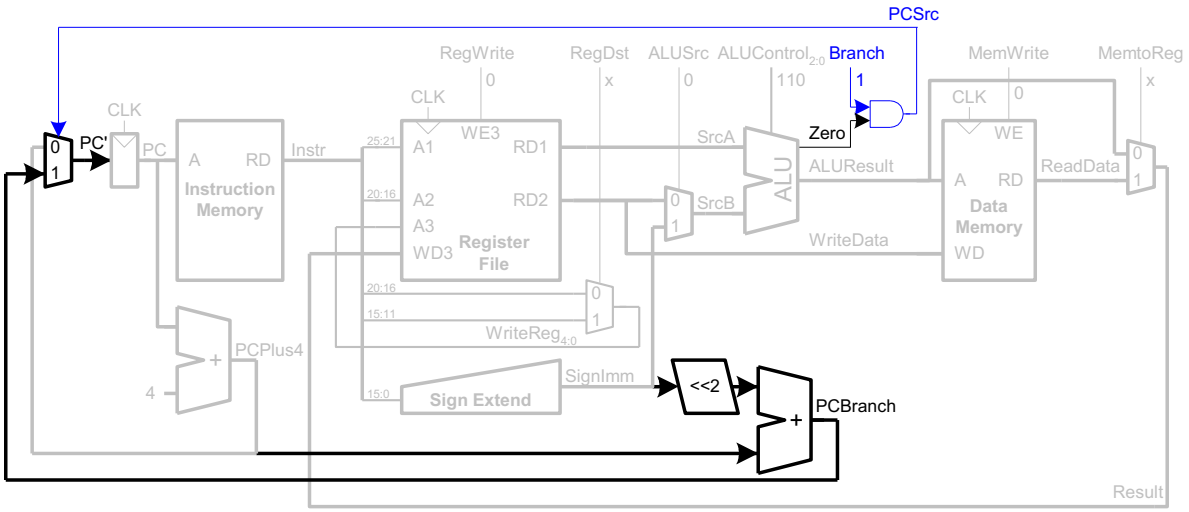

图 12

beq 需要三步操作，分别为条件判断，偏移计算，PC 转移，下面分别实现每步操作。条件判断需要判断 rs、rt 所在的寄存器值是否相等，可以使用 ALU 的减法进行计算，输出 zero 信号，并与译码得到的 Branch 信号 (判断是否为分支指令) 进行 and 操作，作为 PCSrc 信号。第二步偏移计

算公式为:

- target_offsetSign_extend(offset||0^2)
- PC←PC + target_offset

先将 offset 左移 2 位后, 再进行有符号扩展, 最后与 PC+4 相加。最后 PC 转移根据 PCSrc 信号进行控制, 满足条件时, PCSrc 为 1, 选择 PCBranch 作为下一条指令的地址。通路图修改见图 12。

2.3.5 J 指令

跳转指令实际为无条件跳转，不需要做任何判断，因此更加简单。只需要计算地址，更改 PC 即可。其跳转目标由该指令对应的延迟槽指令的 PC(PC+4) 的最高 4 位与立即数 instr_index(instr[25:0]) 左移 2 位后的值拼接得到。如图 13, instr[25:0] 左移 2 位, 拼接 pc[31:28], 而后通过多路选择器。多路选择器直接由 jump 进行控制。

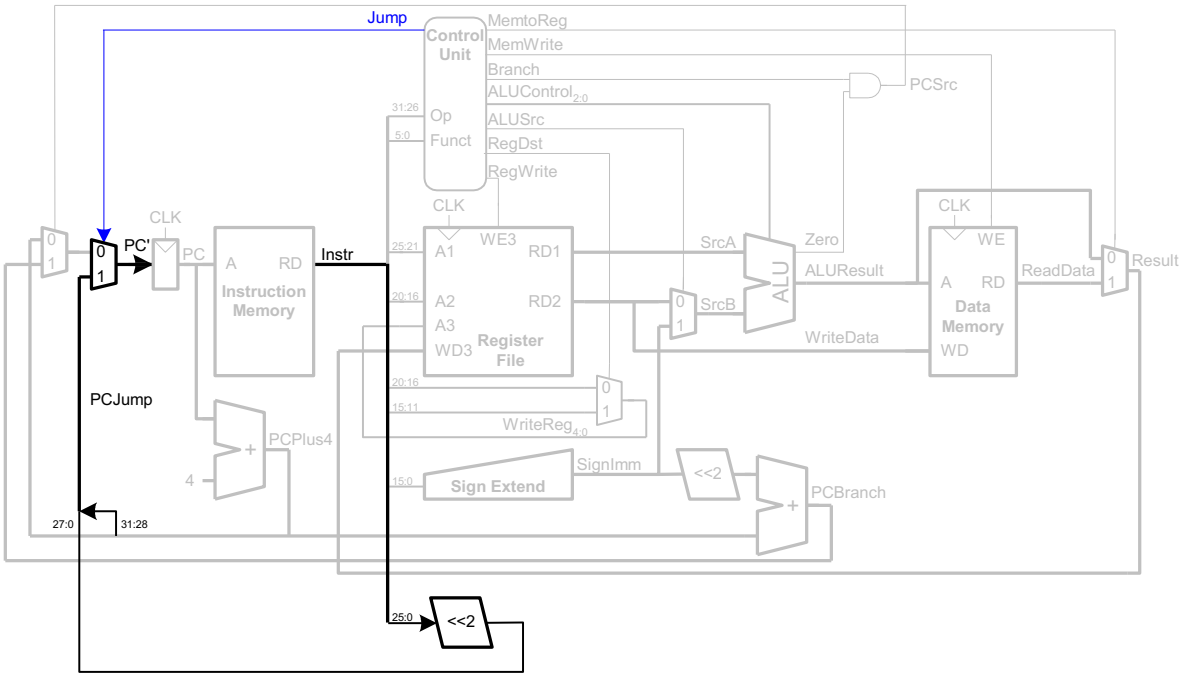

图 13