简明多周期MIPS54设计指南

> 这篇指南的目的在于帮助那些不知如何入手的人找到一条明确的设计途径，只要跟着下面所说的步骤走基本上不会遇到太大的问题。当然，数据通路和控制信号的设计方法因人而异，如果你有更好的想法，或是单纯只想把这篇指南作为参考，都非常欢迎。

这次要设计的是基于MIPS架构的多周期**非流水**线54条指令CPU，其中CP0部分不采用MIPS标准。

说到底，总体的流程就是：

1. 画数据通路
2. 安排控制信号
3. 写代码
4. 调试

话不多说，让我们马上开始。

准备工作

在开始之前，你需要满足以下条件：

1. Vivado的使用经验，包括IP核的使用（写过MIPS31就没问题）
2. 完成了之前的乘法器、除法器和CP0的实验
3. 在理论上对多周期的工作流程有一定了解（上过课就行）
4. 有充裕的时间

同时你需要准备好以下文档以供参考：

1. MIPS Architecture MIPS32 InstructionSet（官方的指令集手册）
2. MIPS31条指令介绍 和 MIPS23条扩展指令介绍（可选，帮助理解官方手册）
3. 课件中关于8条指令多周期CPU设计的部分（最好事先浏览一遍，本次设计的CPU就是以此为基础扩展而来的）

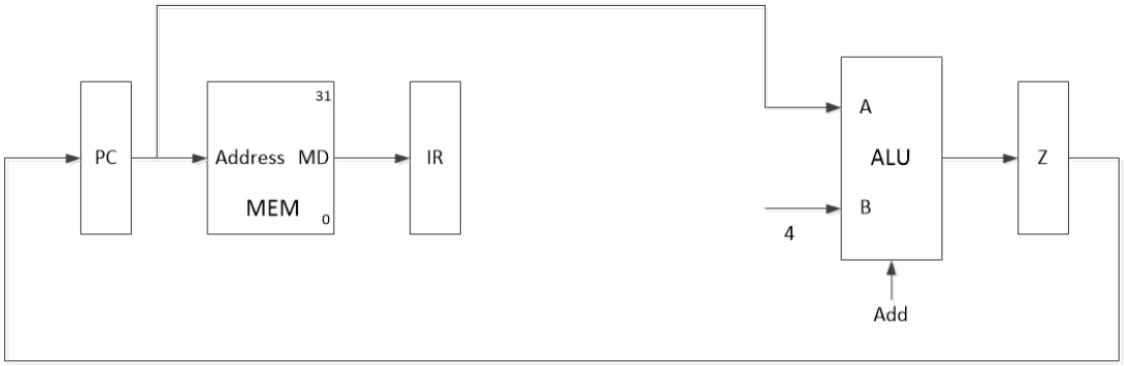
第一部分：数据通路

万事开头难，从零设计CPU更是如此。不过我们可以在现有的框架上进行扩展，正好课件中的8条指令CPU就提供了这样的一个框架，所以我们可以把那8个数据通路直接拿来用。具体用到哪些部件也不太容易一次性想清楚，我们可以边做边加。

这部分的目标是得到一张数据通路表，表上列出所有部件以及每条指令下这些部件的数据来源。

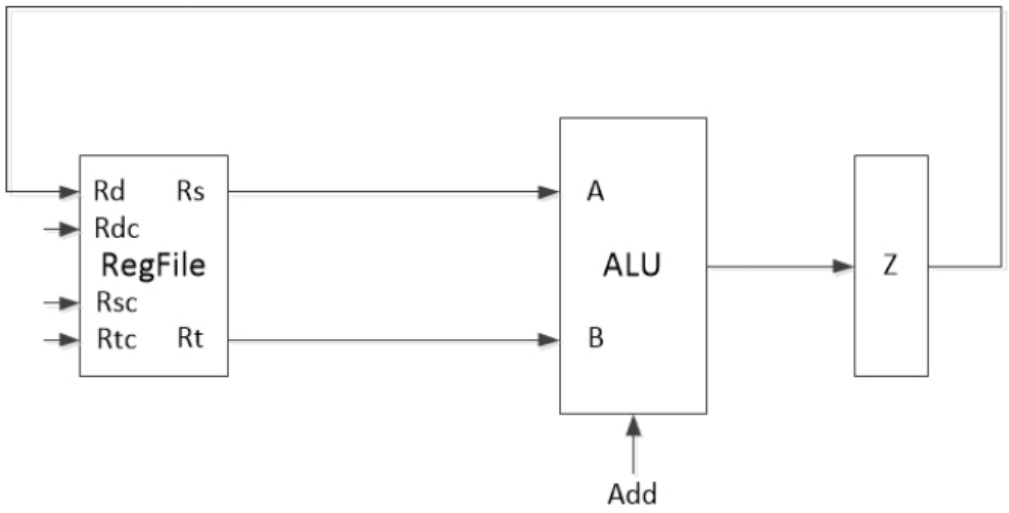
设计数据通路的时候，最好自己跟着画一遍，到时候画总体通路图和设计控制信号的时候也方便参考。做的时候多看看官方的指令手册。

FETCH



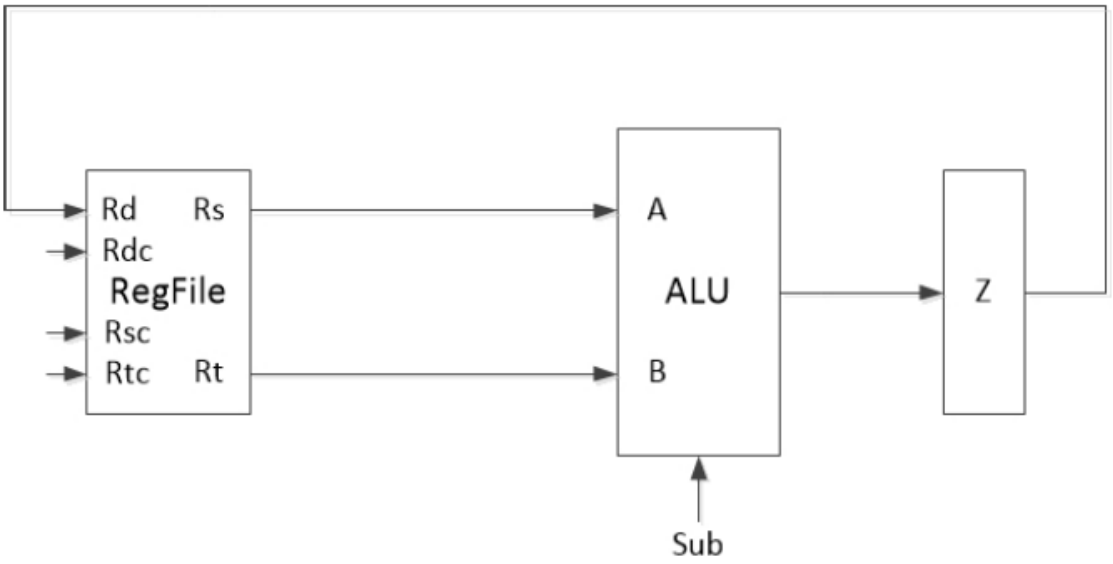
严格来说这不是一个指令，而是每条指令执行前都要进行的操作。跟单周期的数据通路对比，多周期少了一个NPC，多了一个Z寄存器。同一个部件可以在不同的时钟周期接受不同的信号，因此我们就可以将某些功能同质化的部件合并，NPC就是一个例子。Z寄存器的作用是保存ALU运算结果，在下一个时钟周期将数据送到目标部件。

ADDU



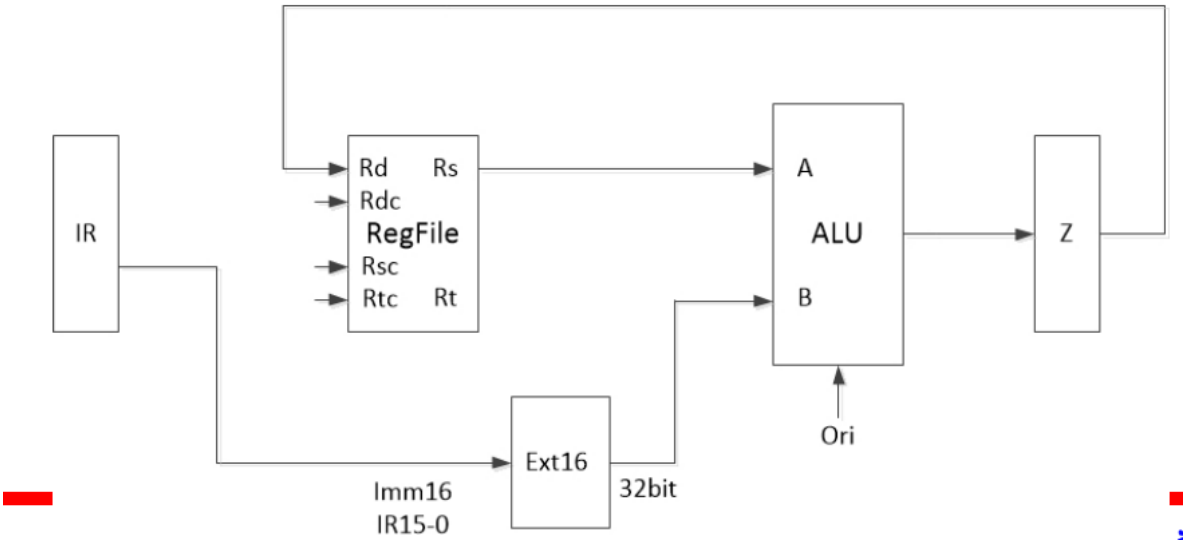
多周期的加法运算需要两个时钟周期来完成，第一个时钟周期去除相加数送至ALU，ALU将结果送至Z，第二个时钟周期Z将结果再送回寄存器堆。

SUBU



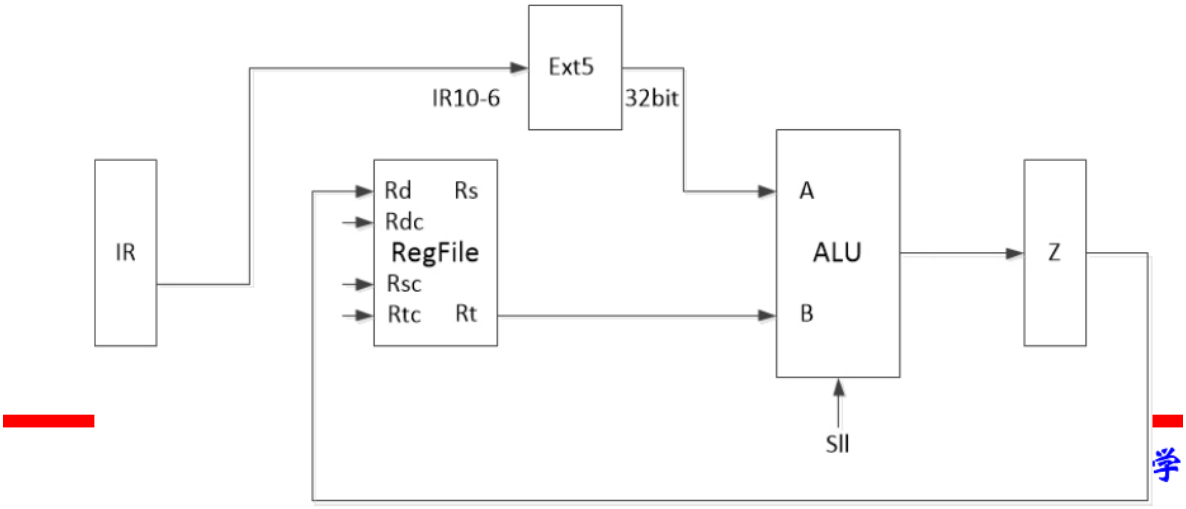
SUBU与ADDU类似。

ORI



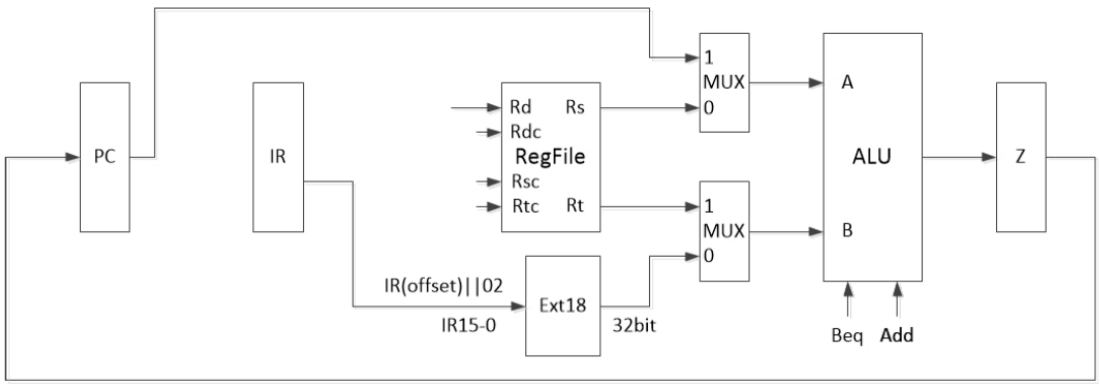
ORI的部分数据是存在指令之中的，处理方法跟MIPS31一样，用16位扩展把它送到ALU。

SLL



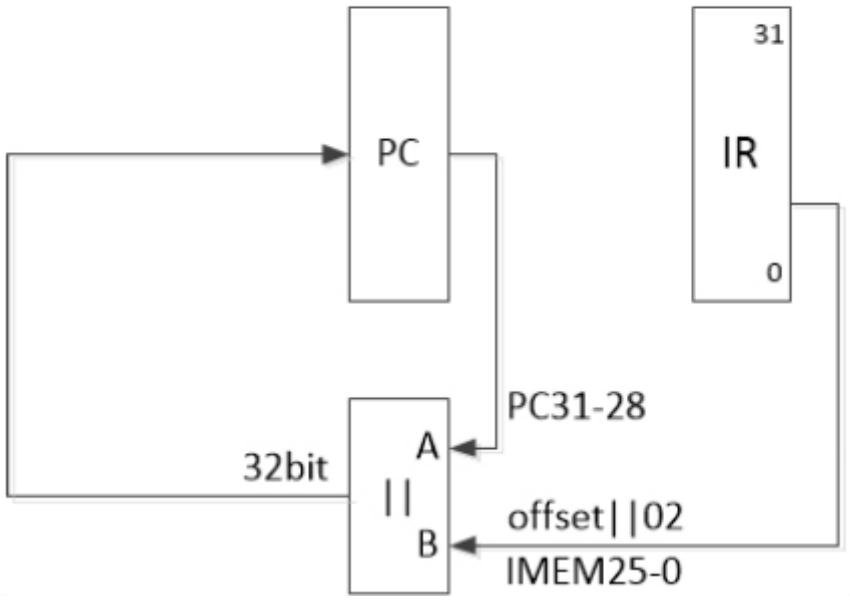
SLL的通路也是与MIPS31类似。

BEQ



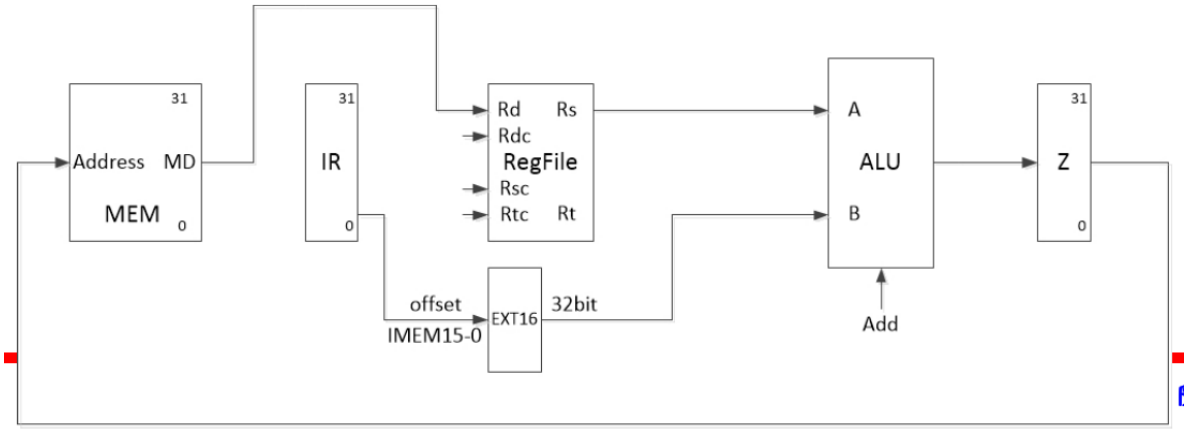
首先要把寄存器堆中的两个数拿出来比较，如果相等，再把PC的数据拿出来加上偏移量再送回PC。这里ALU被使用了两次，所以需要多路选择器。

J

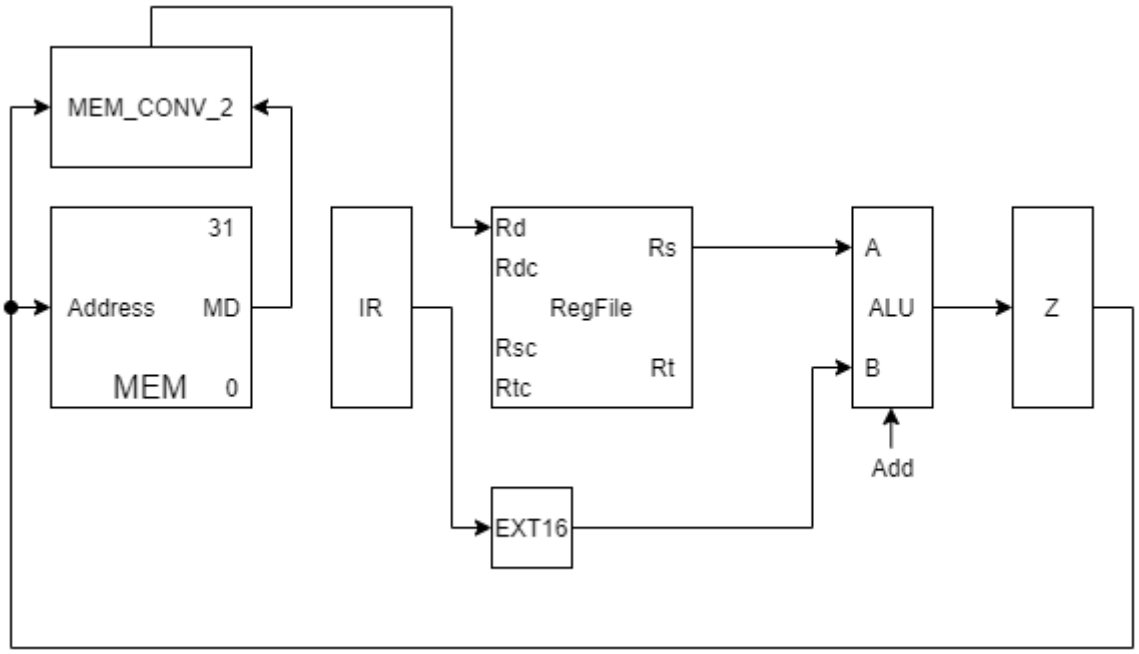


J这条指令很简单，所以数据通路也很简单。

LW



这里要注意，**课件上的框架采用的是冯诺依曼结构，指令和数据是存放在一起的**，读取指令的时候要求一次性读出32位数据，而下面需要实现的LB、SB、LH、SH等指令要求一次性读写8/16位数据。据我所知，**IP核RAM是不能支持读写多种位宽的数据的**，因此如果你想只用一个RAM，还必须设计专门的数据转换器，如果你觉得麻烦，可以自行设计哈佛结构的数据通路。



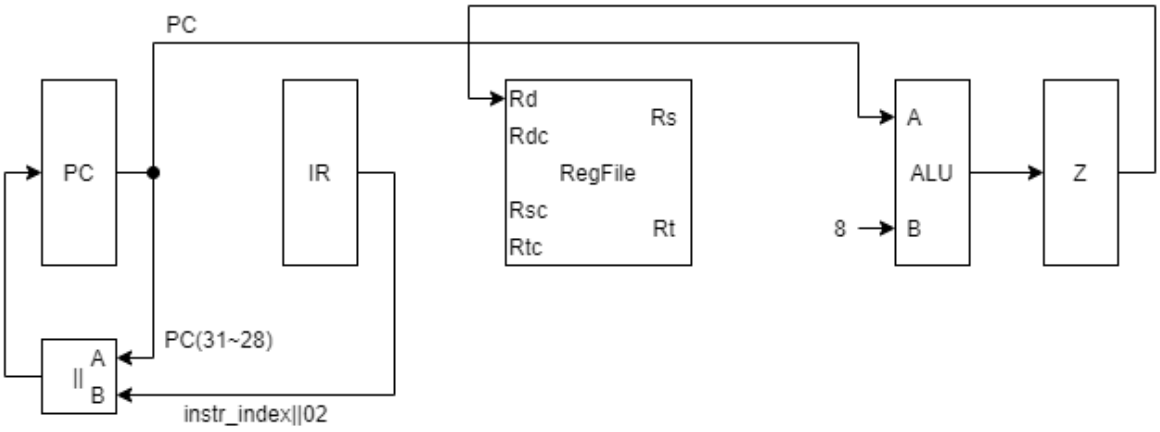
加上数据转换器大概是这个样子。LW要求从存储器一次性读出32位数据，所以我们存储器的字长至少需要32位，同时抛弃地址数据的最低两位。但是这会在之后遇到问题，比如LB指令，从存储器中读出8位数据，你怎么能知道这8位是32位中的哪一部分？这是地址数据最低两位提供的信息。所以我们需要在存储器和CPU之间添加一个部件，这个部件输入存储器读出的包括目标数据的32位数据，它再根据指令类型和地址的低两位将最终的数据传给CPU。

接下来...

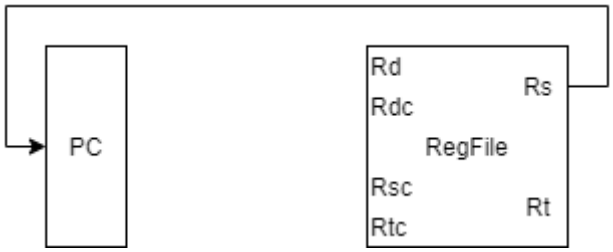
接下来的数据通路就要靠我们自己设计了，不过估计你也摸索出一些规律来了。比如，所有运算类的指令都需要两个周期（不包括取指），第一个时钟周期把数据送到ALU，ALU将结果送到Z，第二个时钟周期Z再把数据送到目标部件。光是这条规律就能够帮我们对付一半以上的指令。不要有顾虑，大胆的去设计，出错了可以再改。

接下来我只会给出一些典型的数据通路，剩下的同质化通路可以类推。

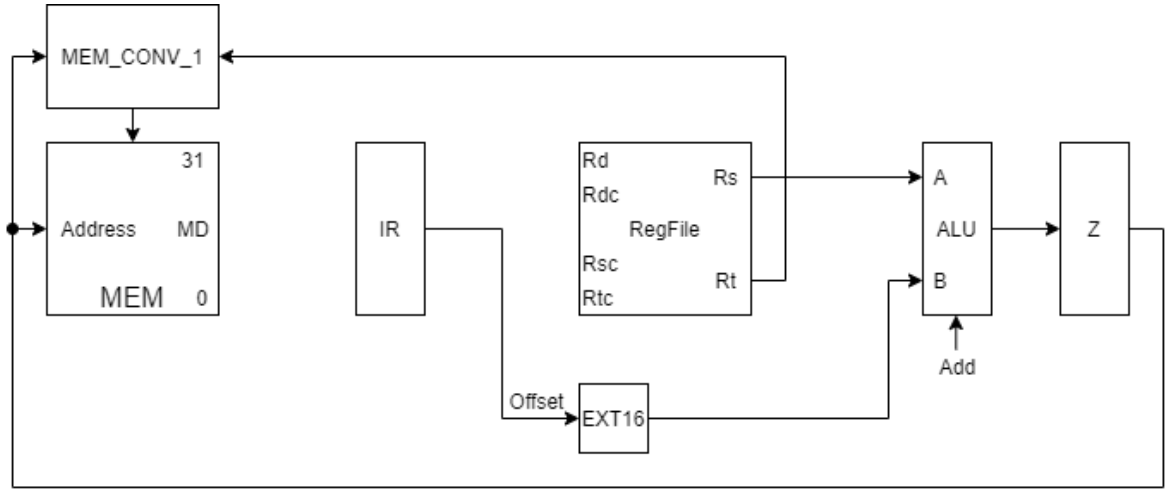
JAL



JR

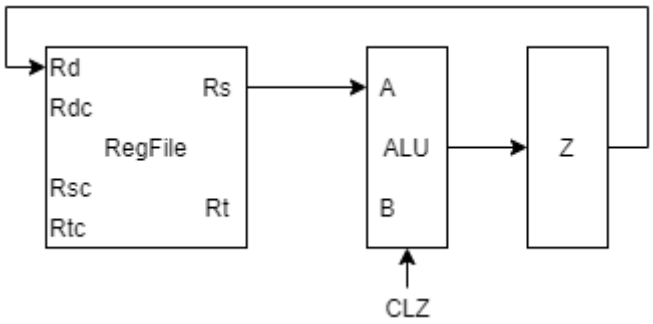


SW



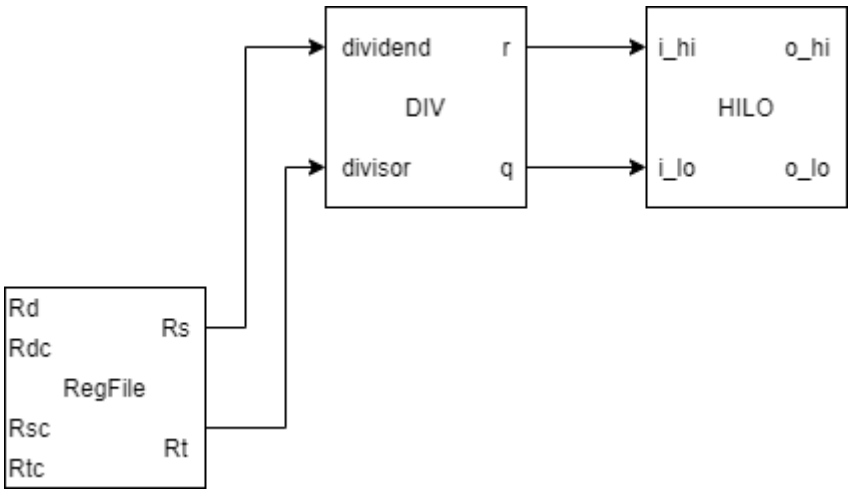
与LW相反，SW要求存储器一次性存储32位数据。还是那个问题，SB和SH要求一次性存储8/16位数据，而我们存储器的字长是32位的，地址的最低两位被抛弃了。我们同样需要一个数据转换器，这个转换器输入来自CPU的32/16/8位数据，根据指令类型和地址低两位判断输送给存储器的数据到底是哪些，结合存储器在这个字中原来的32位数据，形成新的32位数据，传给存储器存储。

CLZ



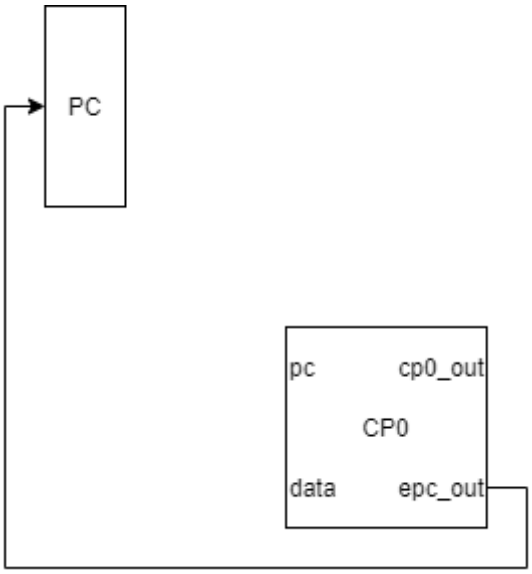
这是新加入的指令，同样属于运算类指令，记得在ALU中加入相应的功能。

DIVU

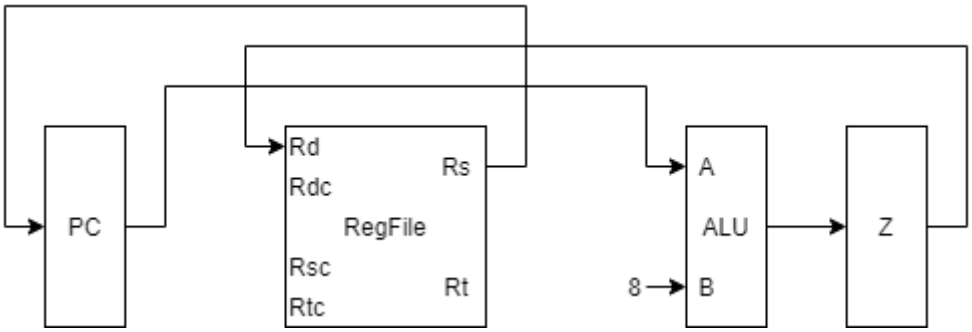


数据通路很简单，难点在于控制信号的设计

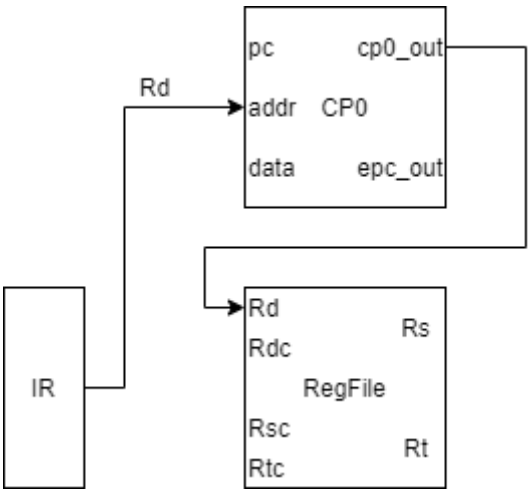
ERET



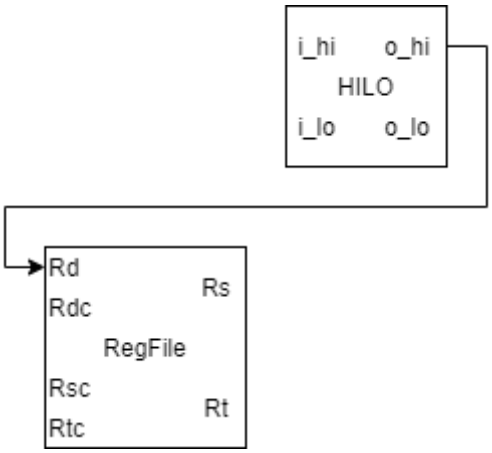
JALR



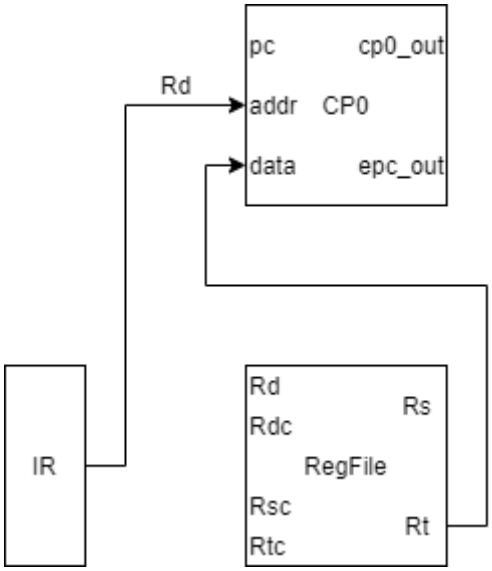
MFC0



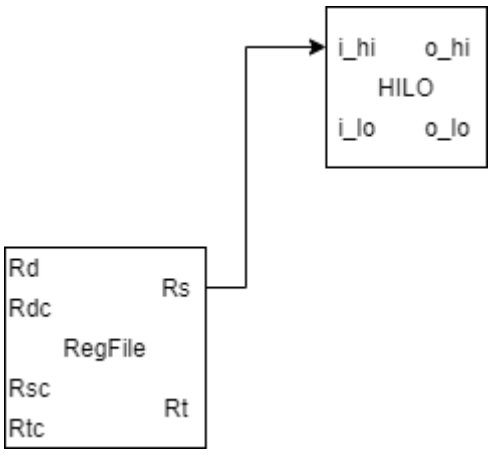
MFHI



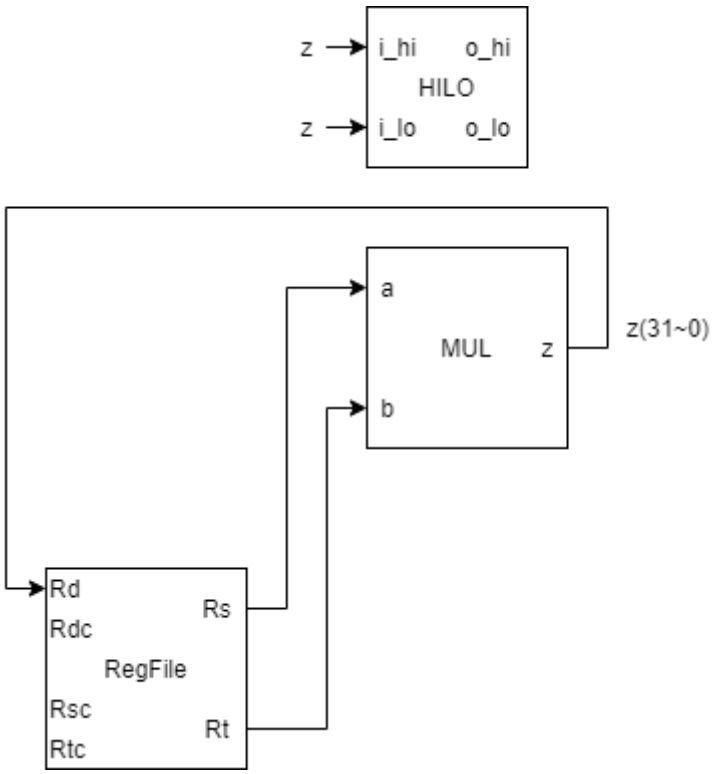
MTC0



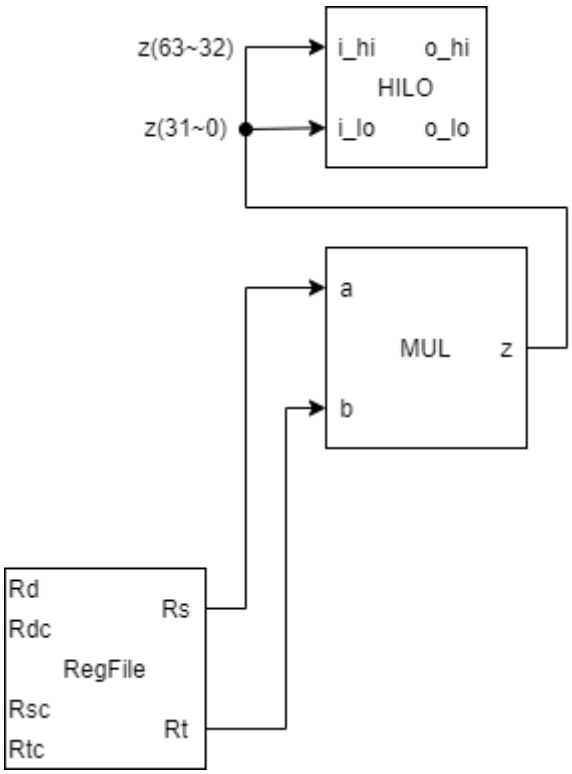
MTHI


i_hi
o_hi
HILO
i_lo
o_lo
Rd
Rdc
RegFile
Rsc
Rtc
Rs
Rt

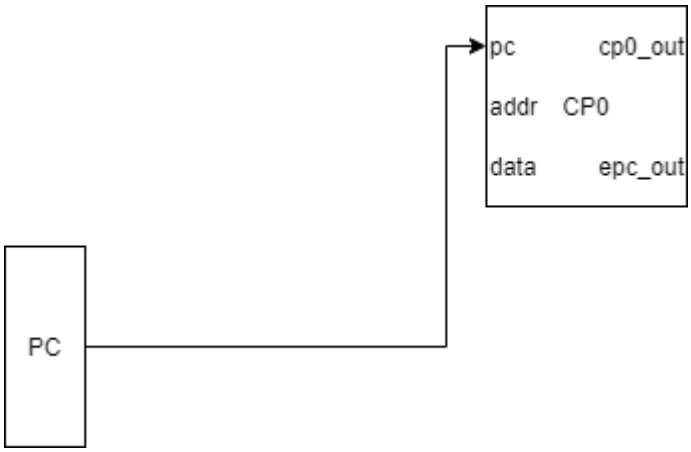
MUL


z
z
i_hi
o_hi
HILO
i_lo
o_lo
a
b
MUL
z
z(31~0)
Rd
Rdc
RegFile
Rsc
Rtc
Rs
Rt

MULTU

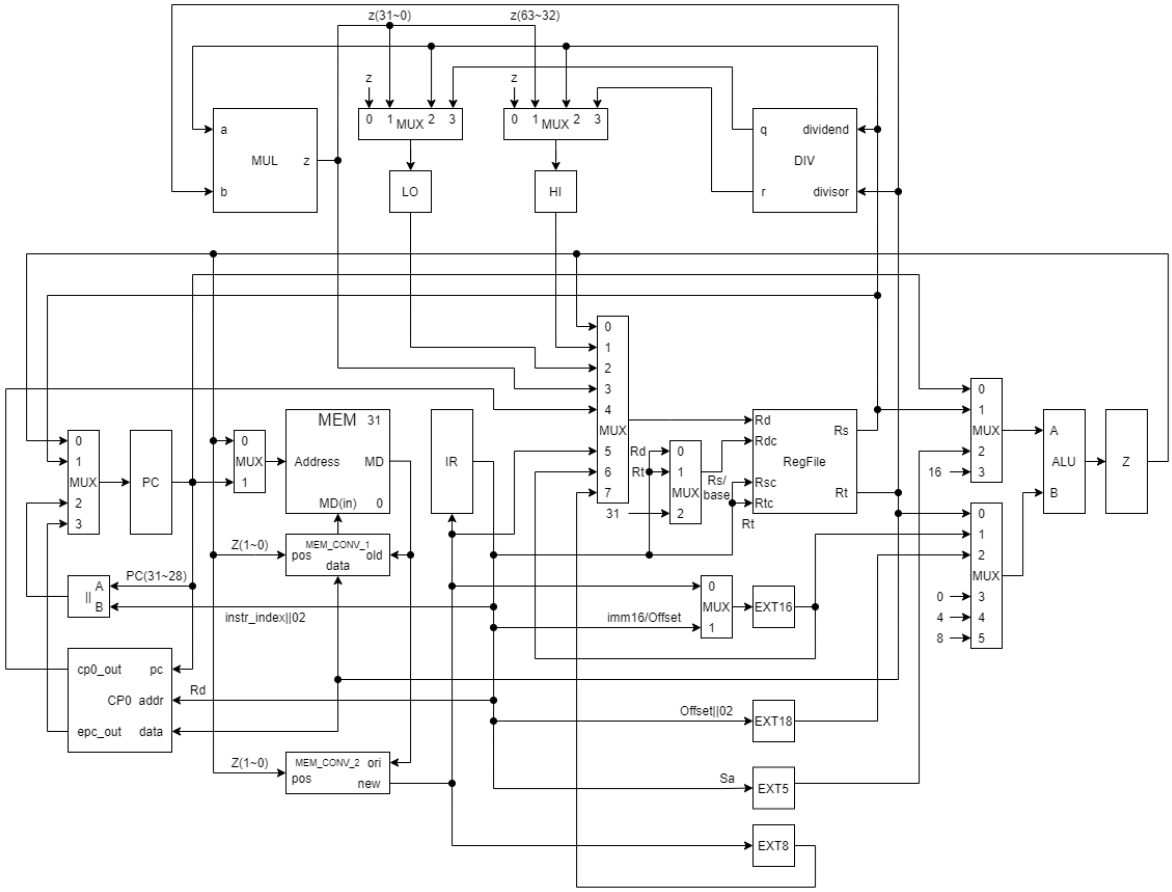


SYSCALL



由于不采用MIPS标准，我们所要做的就是把PC的数据送给CP0，这里也不用将PC的值设为中断程序入口地址。

形成完整数据通路



最后，将54条指令的通路图合并起来，可以得到这样一张总体数据通路图，以及数据通路表。这是设计控制信号的重要参考。

完整的数据通路图和表格可以在这里找到：https://github.com/4x10msv/MIPS54MC

第二部分：控制信号

一句话概括，控制信号就是把数据在正确的时间送到正确的地方。这要怎么实现？就是安排哪些地方开，哪些地方关，哪些地方选哪些数据通过，仅此而已。

> 这里控制信号的设计是根据第一部分的数据通路得到的，如果你的数据通路设计不同，这部分的内容就没多大参考价值。

首先我们观察一下整体数据通路图，看看每个部件都有哪些信号。

- PCin、PCout：控制PC寄存器的输入输出
- MEMr、MEMw：控制存储器的读写
- IRin：控制指令寄存器的输入，不需要输出控制信号是因为IR时刻保持输出状态
- RFrse、RFrte、RFrde：控制寄存器堆的三个端口是否有效
- ALUc：控制ALU运算类型
- Zin、Zout：控制Z寄存器的输入输出
- EXTs：位扩展模块是否进行符号扩展
- EXT_MEMtype：给数据转换模块，表示数据是32/16/8位数据
- DIVStart、DIVs：除法器开始信号和符号运算信号
- MULs：乘法器符号运算信号
- CP0mfc0、CP0mtc0、CP0exception、CP0eret、CP0cause：CP0控制信号
- HIin、HIout、LOin、LOout：控制HILO的输入输出
- 以及各个多路选择器的控制信号

同时，控制器也接受从其他部件来的信号用于判断：

- IRdata：最重要的，指令内容
- ALUz、ALUn：ALU运算结果
- DIVbusy：除法器是否正在进行运算
- CP0status：CP0的status寄存器，用来检查中断屏蔽位

明确了有哪些信号之后，就可以开始着手安排这些信号在每个指令每个时钟周期下的状态了。创建一个控制信号表，横轴列出控制信号，纵轴列出指令，开始填这张表。比如第一条指令ADDI：

		PCin	PCout	MEMr	MEMw	IRin	RFrse	RFrte	RFrde	ALUc	Zin	Zout	EXTs	...
ADDI	T1	0	1	1	0	1	0	0	0	ADD	1	0		
	T2	1	0	0	0	0	0	0	0		0	1		
	T3	0	0	0	0	0	1	0	0	ADD	1	0	1	
	T4	0	0	0	0	0	0	0	1		0	1		
	T5													

不填表示默认值，所有指令的T1和T2都是相同的取指操作。

如果你对自己设计的控制信号不放心，可以同步列出每个指令在每个时钟周期所做的具体操作。就像这样：

	T1	T2	T3	T4	T5
ADDI	PC->MEM, PC+4->Z	Z->PC	Rs+sext(imm16)->Z	Z->Rd	

下面给出几条典型指令的操作安排。

BEQ

	T1	T2	T3	T4	T5
BEQ	PC->MEM, PC+4->Z	Z->PC	Rs-Rt->Z	ALUz=1: PC+sext(offest||02)->Z	ALUz=1: Z->PC

BEQ是一个分支跳转指令，这就需要控制器在T3时根据ALU的结果决定是否转入T4。

J

	T1	T2	T3	T4	T5
J	PC->MEM, PC+4->Z	Z->PC	PC(31~28)||instr_index||02->PC		

JAL

	T1	T2	T3	T4	T5
JAL	PC->MEM, PC+4->Z	Z->PC	PC->Z	Z->Rd	PC(31~28)||instr_index||02->PC

JR

	T1	T2	T3	T4	T5
JR	PC->MEM, PC+4->Z	Z->PC	Rs->PC		

LW

	T1	T2	T3	T4	T5
LW	PC->MEM, PC+4->Z	Z->PC	base+sext(offset)->Z	Z->MEM,MD->MEM_CONV_2->Rd	

SW

	T1	T2	T3	T4	T5
SW	PC->MEM, PC+4->Z	Z->PC	base+sext(offset)->Z	Z->MEM,Rt->MEM_CONV_1->MD	

DIVU

	T1	T2	T3	T4	T5
DIVU	PC->MEM, PC+4->Z	Z->PC	Rs,Rt->DIV	dividend/divisor,busy=0: DIV->(HI,LO)	

由于除法完成运算需要数十个时钟周期，因此在除法器完成运算之前我们只能让控制器保持在相同状态而不进行其他操作，这可以根据DIVbusy信号进行判断。

ERET

	T1	T2	T3	T4	T5
ERET	PC->MEM, PC+4->Z	Z->PC	CP0(status>>5),CP0(epc)->PC		

JALR

	T1	T2	T3	T4	T5
JALR	PC->MEM, PC+4->Z	Z->PC	PC+0->Z	Z->Rd	

MFC0

	T1	T2	T3	T4	T5
MFC0	PC->MEM, PC+4->Z	Z->PC	IR(Rd)->CP0(addr),CP0(cp0_out)->Rt		

MFHI

	T1	T2	T3	T4	T5
MFHI	PC->MEM, PC+4->Z	Z->PC	HI->Rd		

MTC0

	T1	T2	T3	T4	T5
MTC0	PC->MEM, PC+4->Z	Z->PC	IR(Rd)->CP0(addr),Rt->CP0(data)		

MTHI

	T1	T2	T3	T4	T5
MTHI	PC->MEM, PC+4->Z	Z->PC	Rs->HI		

MUL

	T1	T2	T3	T4	T5
MUL	PC->MEM, PC+4->Z	Z->PC	(Rs*Rt)(31~0)->Rd		

MULTU

	T1	T2	T3	T4	T5
MULTU	PC->MEM, PC+4->Z	Z->PC	Rs*Rt->(HI,LO)		

SYSCALL

	T1	T2	T3	T4	T5
SYSCALL	PC->MEM, PC+4->Z	Z->PC	exception=1: CP0(status<<5),PC->CP0(epc),cause->CP0(cause)		

完整的控制信号表可以在这里获取：https://github.com/4x10msv/MIPS54MC

第三部分：代码&调试

有了数据通路和控制信号，也就成功了一半，接下来要做的就是根据上面得到的设计方案转化为代码。我推荐的顺序是：先实现各个模块，再实现控制器，最后将各个部件连接在一起。

宏的应用

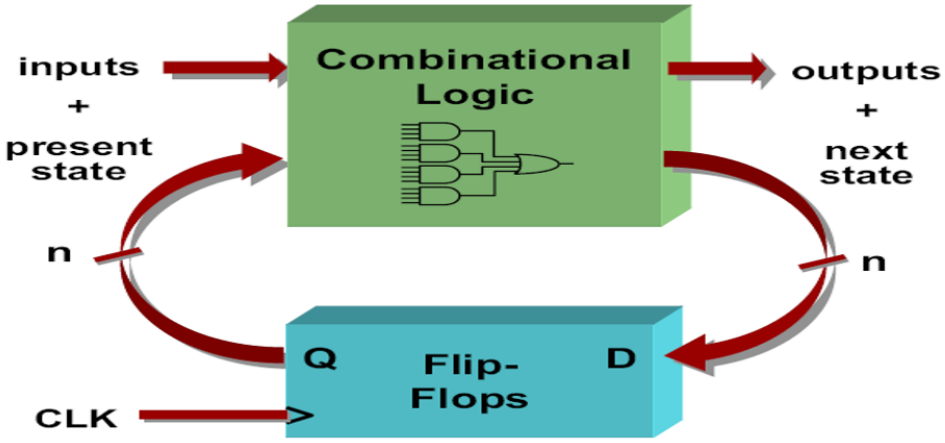
关于代码实现，这边提一个建议，就是善用宏来提高代码的可读性和可维护性。这里的宏，就跟C++里的宏差不多。先新建一个"defines**.vh**"文件，也就类似于C++里的头文件。我是这样组织里面的内容的：

```
`ifndef _defines_vh_
`define _defines_vh_    //防止重复编译
```

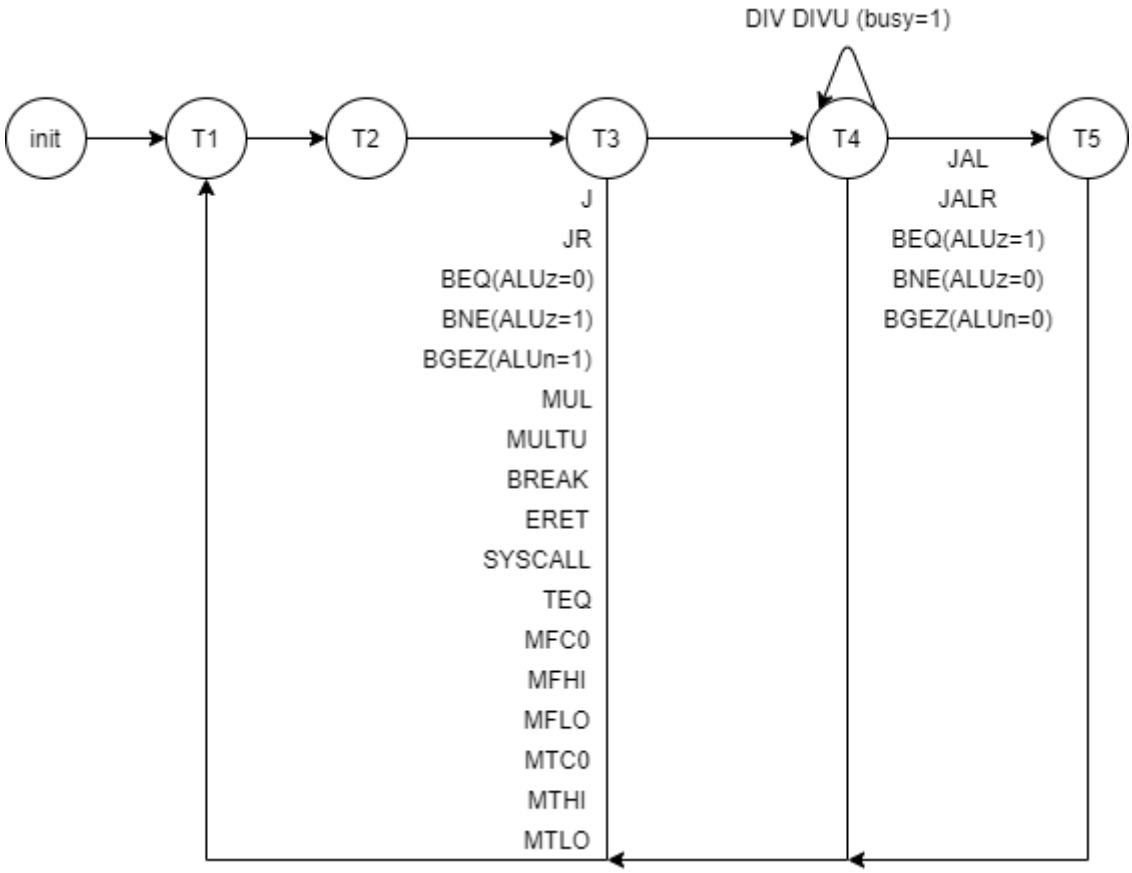
```
//----------Operation-----------
`define OP_ADDI          6'b001000
//...
`define FUNCT_ADD        6'b100000
//...

//-----------CtrlUnit-----------
`define CTRL_STATE_T1     3'b001
///...

//-------------ALU--------------
`define ALU_OP_ADDU          4'b0000
//...

//-------------MUX--------------
`define MUX_PC_Z             4'b0001      //这边MUX的控制信号采用独热码是为了方便修改
`define MUX_PC_RS            4'b0010
`define MUX_PC_CON           4'b0100
`define MUX_PC_EPCOUT        4'b1000
//...

//------------CP0---------------
`define CP0_CAUSE_SYSCALL    32'b1000
`define CP0_CAUSE_BREAK      32'b1001
`define CP0_CAUSE_TEQ        32'b1101
//...

`endif
```

模块文件想引用这些宏，只需要在模块定义前加上一行`` `include "defines.vh" ``即可。与C++不同的是，引用这些宏需要在宏名称之前加“`“，比如“`OP_ADDI“。

控制器的设计模式



多周期CPU的控制器就是一个自动机。回顾自动机的工作原理图，自动机由两部分组成，一部分是表示当前状态的寄存器组，另一部分是组合逻辑，输入外部信号和当前状态， 输出给外部的信号和下一状态。我们要设计的控制器也应是如此，T1~T5就是它的状态，组合逻辑部分输入来自其他模块的信号，输出控制信号和下一状态。



控制器的状态转移图可以根据先前的指令流程表得到。

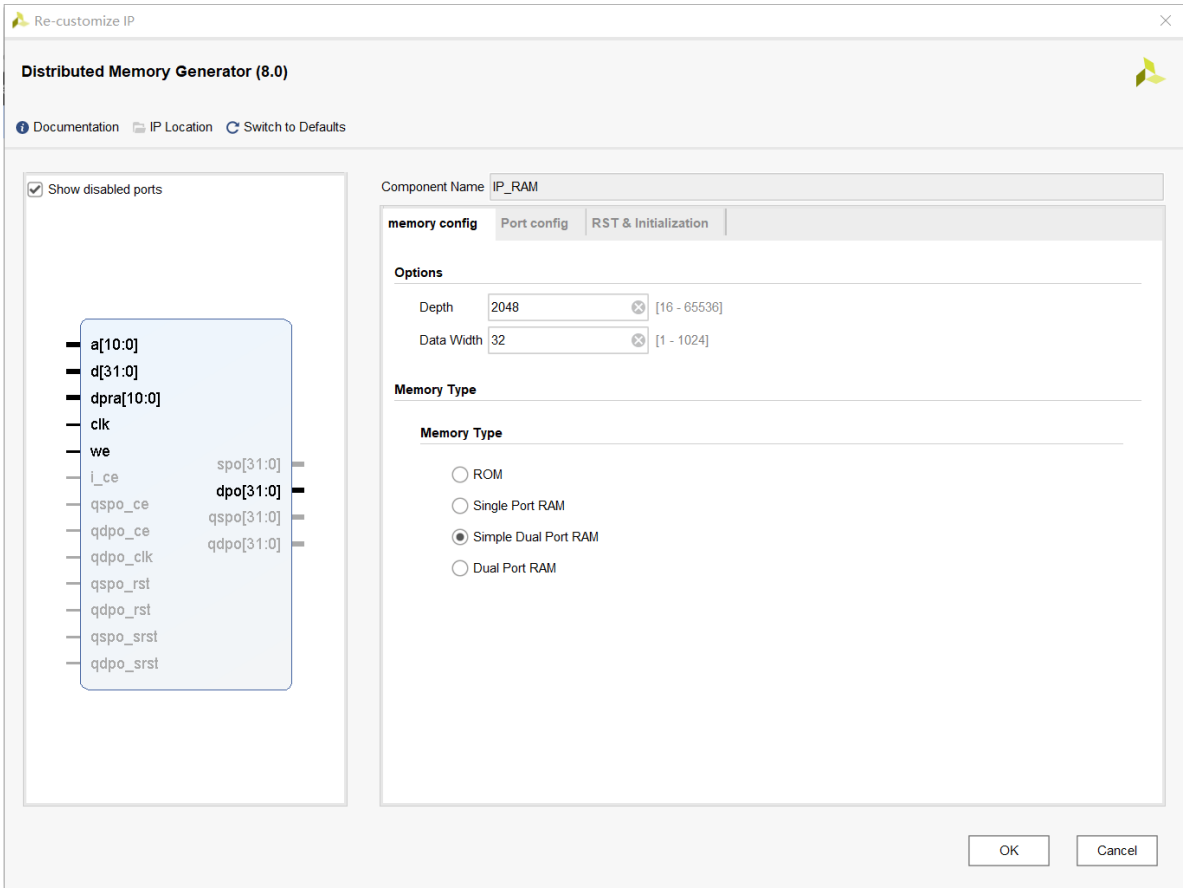
在代码实现上，推荐"3always"写法：

```
// 第一个always确定现态
always @ (posedge clk or posedge rst) begin
    if(rst) begin
        curState <= `CTRL_STATE_INIT;
    end
    else begin
        curState <= nextState;
    end
end

// 第二个always确定次态（组合逻辑）
always @ (*) begin
    case(curState)
        `CTRL_STATE_INIT: begin
            nextState <= `CTRL_STATE_T1;
        end
        `CTRL_STATE_T1: begin
            nextState <= `CTRL_STATE_T2;
        end
        `CTRL_STATE_T2: begin
            nextState <= `CTRL_STATE_T3;
        end
        `CTRL_STATE_T3: begin
            if(
                (opBeq && !aluZ) // || ...
            ) begin
                nextState <= `CTRL_STATE_T1;
            end
            else begin
```

```
                nextState <= `CTRL_STATE_T4;
            end
        end
        `CTRL_STATE_T4: begin
            if(
                opJal // || ...
            ) begin
                nextState <= `CTRL_STATE_T5;
            end
            else if((opDiv || opDivu) && divBusy) begin
                nextState <= `CTRL_STATE_T4;
            end
            else begin
                nextState <= `CTRL_STATE_T1;
            end
        end
        `CTRL_STATE_T5: begin
            nextState <= `CTRL_STATE_T1;
        end
        default: begin
            nextState <= `CTRL_STATE_T1;
        end
    endcase
end

//第三个always确定控制信号（组合逻辑）
always @ (*) begin
    case(curState)
        `CTRL_STATE_T1: begin
            //...
        end
        `CTRL_STATE_T2: begin
            //...
        end
        `CTRL_STATE_T3: begin
            if(ADDI) begin
                rRfRsE = 1;
                rAluC = `ALU_OP_ADD;
                rZIn = 1;
                rExtS = 1;
                rMuxAluaC = `MUX_ALUA_RS;
                rMuxAlubC = `MUX_ALUB_EXT16;
                rMuxExt16C = `MUX_EXT16_IMMOFF;
            end
            // else if ...
        end
        `CTRL_STATE_T4: begin
            //...
        end
        `CTRL_STATE_T5: begin
            //...
        end
    endcase
end
```

这样写虽然比较繁琐，但是有利于接下来调试的时候修改代码。

IP核



如果你采用的是冯诺依曼结构，为了配合数据转换模块，存储器需要支持同时读写，因此需要将IP核设置为简单双口RAM，如图所示。

调试的建议

调试必然是漫长而繁琐的过程，下面给出几点建议：

- 静态查错：写完代码后不要急着运行，先回顾一遍，能够发现不少错误，修改这些错误的代价远比你调试的时候低。
- 善用波形图：波形图只显示在顶层tb模块定义的信号，把需要查看的信号提升至顶层的tb模块，模块之间的层级关系可以用“.”表示。