

北京科技大学

计算机组成原理 课程设计指导书

计算机通信与工程学院计算机实验室

张 磊

2019-11-11

目 录

序言.....	3
第一章 基本概念.....	4
1.1 流水线的基本原理	4
1.2 五级流水线 CPU 结构	5
1.3 MIPS 指令	5
第二章 构建一条流水线.....	7
2.0 准备工作	7
2.1 顶层模块 Core	9
2.2 存储模块	10
2.2.1 ROM	10
2.2.2 RAM	12
2.2.3 RegFile	15
2.3 取指阶段 IF	16
2.3.1 PC	16
2.3.2 IFID	18
2.4 译码阶段 ID	20
2.5 执行阶段 EX	26
2.6 访存阶段 MEM	27
2.7 写回阶段 WB	28
2.8 解决数据相关问题	29
第三章 流水线的改进.....	33
3.1 跳转指令的实现	33
3.1.1 延迟槽	33
3.1.2 PC 模块修改	33
3.1.3 ID 模块修改	34
3.2 访存指令的实现	35
3.3 流水线暂停的实现	37
第四章 TinyMIPS 处理器的测试.....	41
4.1 TinyMIPS 模块结构	41
4.2 功能仿真测试	41
4.3 上板功能测试	42
附录.....	42
A: 使用 GCC 工具链编译生成固件	42

A.1 WSL 的安装	42
A.2 VMware 下安装 Ubuntu.....	44
A.3 安装交叉编译软件包	45
A.4 编写 Makefile	45
A.5 FAQ.....	49
B: 使用 Vivado 的 Block Design 功能	50
B.1 新建 Block Design	50
B.2 绘制	50

序言

计算机组成原理课程设计是计算机专业、信息安全专业、物联网专业的专业必修课，共 32 学时，2 个学分。该课程在计算机组成原理课程之后的一个学期内开设。在计算机组成原理实验中，大家了解了单周期处理器的设计过程，并且通过功能仿真进行了单周期处理器的功能测试。在课程设计中，主要以流水线处理器设计实现为主，尚未涉及异常处理，亦较少涉及处理器性能的提高。在本书中，通过分模块的方式带大家实现了一个 22 条指令的支持流水线暂停机制的 MIPS 五级静态流水处理器 TinyMIPS，与本书配套的 TinyMIPS 工程也一并提供给大家。为了了解大家的实验过程中的情况，在 CG 平台设置了 20 个单元模块的在线评测。大部分评测题都可以使用工程中的代码完成，只有少部分题目需要大家补全代码后再进行在线评测。该处理器在编码风格和设计思路都是值得大家学习和思考的。在第四章中给出了 TinyMIPS 处理器整体的功能测试和在龙芯实验箱上进行功能点测试的过程，目的是学生能够对设计的处理器的测试方法有所了解。学有余力的同学可以在完成本设计内容的基础上根据喜好自行设计其余指令集手册中的指令完善 MIPS 处理器或者 RISC-V 指令集处理器。与本书配套的文档有 A01_MIPS32_QRC、A02_“系统能力培养大赛”MIPS 指令系统规范_v1.00、A03_计组课设指导书 2018、A04_MIPS_Vol2、A05_龙芯体系结构实验箱 (Artix-7) -实验系统使用手册_v1.00、A06_《自己动手写 CPU》等。

本书在撰写过程中得到了刘宏岚老师、张晓彤老师、何杰老师、齐悦老师的大力支持。其中的 TinyMIPS 处理器为以邢其正同学为队长的计算机系统能力培养大赛团队独立设计实现的。王硕、任亮、邓继堂、崔迪等同学也做了大量的工作。在此一并表示感谢！在计算机系统能力培养的道路上任重而道远，期待着更多的同学能够加入到计算机系统能力培养团队，提升我校计算机专业的人才培养质量。

编者

2019.11

第一章 基本概念

1.1 流水线的基本原理

该指导书主要介绍如何实现一个 5 级流水线结构的简单 CPU。除此之外，在这个世界上同样存在着许多其他 CPU 的实现方法，同学们可以自行上网查阅相关内容。

流水线是将计算机指令处理过程拆分为多个步骤，并通过多个硬件处理单元并行执行来加快指令的执行速度的一种结构。对于一个单周期 CPU 而言，CPU 需要在一个周期内执行从取指、译码到执行全部的功能；但进行拆分后，CPU 每个周期只需要执行一个功能，并且各模块之间又可以并行处理，从而大大的提高了 CPU 的工作效率。

此处借用各个参考书中都十分常见的五级流水线的工作结构进行说明。图 1-1 为一个标准的五级流水线的工作时序图。

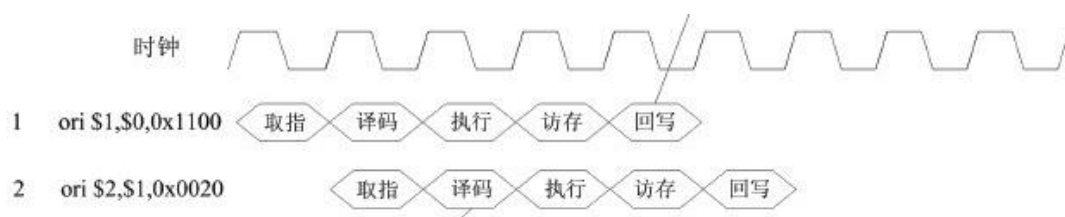


图 1-1 标准的五级流水线的工作时序图

可见，原本需要 10 个周期执行的两条指令，被缩短到 6 个周期，CPU 执行指令的速度大幅提升。在指令条数增多且没有任何冲突、异常及特殊操作的理想情况时，CPU 的各大模块不会暂停工作，从而更加合理地利用了硬件资源，减少了资源浪费。

在指导书的后半部分将介绍有关分支、跳转指令的执行方式、流水线冲突的处理。如果大家学有余力，也可以在 TinyMIPS 的基础上自行实现乘除法指令以及中断、异常的处理。

1.2 五级流水线 CPU 结构

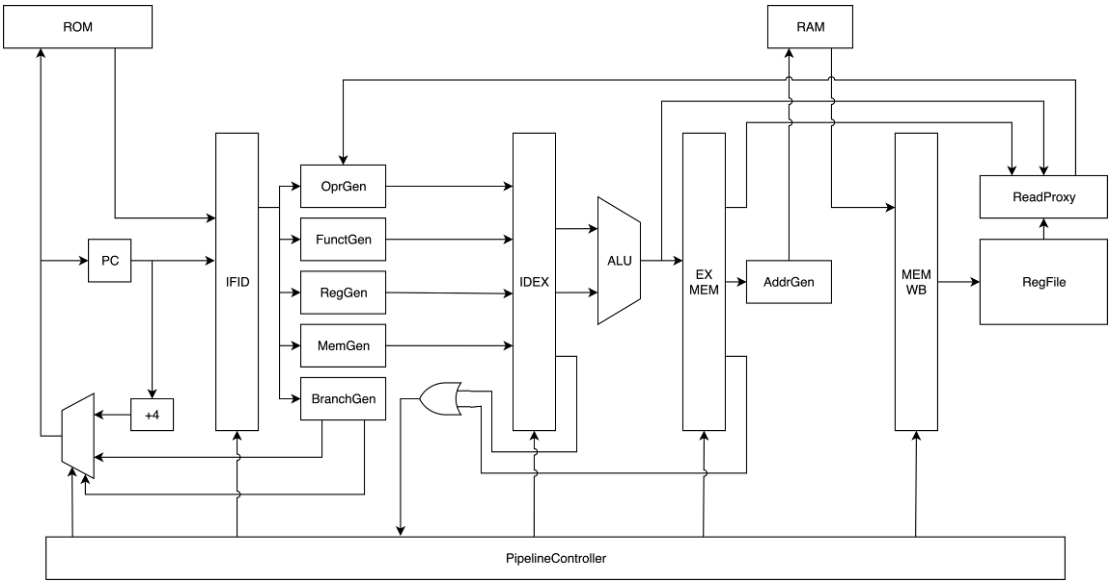


图 1-2 TinyMIPS 数据通路框图

图 1-2 为 TinyMIPS 流水线的数据通路框图。

TinyMIPS 的流水线共分为五级，对应五个功能模块，分别为 IF（取指令）、ID（译码）、EX（执行）、MEM（访存）、WB（写回）。而这五个流水级分别对应 CPU 处理指令时的五个步骤：IF 级负责从存储器（内存或缓存）中取出指令；ID 级负责将指令译码，并从寄存器堆取出指令的操作数；EX 级负责根据译码结果执行对应的 ALU 操作；MEM 级负责处理可能产生访存请求的指令，向存储器（内存或缓存）发送控制信号；WB 级负责将指令的执行结果写回寄存器堆。

1.3 MIPS 指令

本次 TinyMIPS 总共只需要实现 22 条指令，这 22 条指令足以支持一个简单且功能完备的 CPU 的运行。当然有能力的同学可以继续实现 MIPS 指令集架构中的其他指令(指令规范详见文档 A01_MIPS32_QRC 或 A02_“系统能力培养大赛” MIPS 指令系统规范_v1.00)。

TinyMIPS 支持的指令如下：

表 1-1 算术运算指令

指令助记格式	指令功能简述
ADDU rd,rs,rt	无符号加（无溢出异常）

ADDIU rt,rs,imm	无符号加立即数（无溢出异常）
SUBU rd,rs,rt	无符号减（无溢出异常）
SLT rd,rs,rt	有符号小于置 1
SLTU rd,rs,rt	无符号小于置 1

表 1-2 逻辑运算指令

指令助记格式	指令功能简述
AND rd,rs,rt	按位与
LUI rt,imm	寄存器高位置立即数
OR rd,rs,rt	按位或
XOR rd,rs,rt	按位异或

表 1-3 移位指令

指令助记格式	指令功能简述
SLL rd,rt,sa	立即数逻辑左移
SLLV rd,rs,rt	寄存器逻辑左移
SRAV rd,rs,rt	寄存器算术右移
SRLV rd,rt,sa	寄存器逻辑右移

表 1-4 分支跳转指令

指令助记格式	指令功能简述
BEQ rs,rt,offset	相等时分支转移
BNE rs,rt,offset	不等时分支转移
JAL target	无条件直接跳转，并保存返回地址
JALR rd,rs	无条件寄存器跳转，并保存返回地址

表 1-5 访存指令

指令助记格式	指令功能简述
LB rt,offset(base)	访存读字节（8 位），有符号扩展
LBU rt,offset(base)	访存读字节（8 位），无符号扩展
LW rt,offset(base)	访存读字（32 位）

SB rt,offset(base)	访存写字节（8 位）
SW rt,offset(base)	访存写字（32 位）

第二章 构建一条流水线

本章开始实现 TinyMIPS 的基本功能，并构建一条简单的流水线。为了便于进行测试，请大家在实现 CPU 的每一个模块时，按照我们所给的定义实现模块的接口信号。

2.0 准备工作

在我们编写的 CPU 中涉及多个具有固定宽度的总线，用于传递指令、数据、地址等等。为了方便对这些总线的宽度进行修改，加强代码可读性，我们可以将这些总线的宽度编写成宏定义的形式。这么做的同时也便于我们快速理解代码的含义。在具体实现时，我们可以新建一个文件 bus.v，在其中进行定义。形式如下：

bus.v 样例

```

`ifndef TINYMIPS_BUS_V_
`define TINYMIPS_BUS_V_

// address bus
`define ADDR_BUS          31:0
`define ADDR_BUS_WIDTH    32

// instruction bus
`define INST_BUS           31:0
`define INST_BUS_WIDTH    32

// data bus
`define DATA_BUS          31:0
`define DATA_BUS_WIDTH    32

`endif // TINYMIPS_BUS_V_

```

对于 MIPS 指令集，各指令的操作码、funct 字段、shamt 字段等都是固定值，因此我们也可以将这些值通过宏定义的方式进行定义，从而可以提升在译码阶段代码的可读性。在 TinyMIPS 中我们将这些宏定义放在不同文件当中，并在开发过程中不断完善这些文件的内

容，大家也可以根据自己的需求建立宏定义文件。文件组织如下：

宏定义文件列表

文件名	用途
funct.v	用于存放 funct 字段内容
opcode.v	用于存放 opcode 字段内容
segpos.v	用于存放指令当中各字段位置
pcdef.v	定义 PC（程序计数器）的初始值
sim.v	用于存放 ROM 与 RAM 中的相关宽度

以下是 opcode.v 的部分代码：

opcode.v 样例

```
// immediate
`define OP_LUI      6'b001111
// memory accessing
`define OP_LB       6'b100000
`define OP_LW       6'b100011
`define OP_LBU      6'b100100
`define OP_SB       6'b101000
`define OP_SW       6'b101011
```

在我们实现的五级流水当中存在 4 个流水线中间级：IF_ID、ID_EX、EX_MEM、MEM_WB。这些部件用于在时钟上升沿时将上一个流水级的输出锁存，并且传递给下一个流水级作为输入。它们的功能基本一致，内部可以由多个 D 触发器实现，这些触发器只在数据宽度上存在区别。因此我们可以使用 Verilog 的带参数模块来实现流水线中间级中的 D 触发器——这样能够显著提高代码复用程度，降低错误出现的可能。这里进行一下带参数模块的介绍，同学们可以自行实现：

带参数模块样例

```
module PipelineDeliver #(
    parameter WIDTH = 1
) (
    input                clk,
    input                rst,
    input                stall_current_stage,
```

```

input                stall_next_stage,
input [WIDTH - 1:0] in,
output reg [WIDTH - 1:0] out
);

```

这是一个定义的带参数模块，其中 **WIDTH** 就是这里的可变参数，我们可以在实例化模块的时候对这一参数进行重新的定义，从而使得我们的 D 触发器可以存储不同宽度的数据，其实例化过程如下：

```

PipelineDeliver #(`MEM_SEL_BUS_WIDTH) ff_mem_sel(
    clk, rst,
    stall_current_stage, stall_next_stage,
    mem_sel_in, mem_sel_out
);

```

这里可见我们在模块实例化的时候通过#后的数据大小，对 **WIDTH** 进行了重定义，使得 **WIDTH** 变为了我们宏定义中 **`MEM_SEL_BUS_WIDTH** 的大小。

2.1 顶层模块 Core

我们首先来创建一个顶层模块 **Core**，它有对 **ROM**、**RAM** 的控制信号，以及用于进行测试的 **debug** 信号，大家可以直接使用我们提供的 **Core** 模块。

具体信号如下：

Core.v 的信号定义

名称	类型	宽度	方向	用途
clk	wire	1	i	时钟信号
rst	wire	1	i	复位信号
rom_en	wire	1	o	rom 使能
rom_write_en	wire	4	o	rom 写使能
rom_addr	wire	32	o	rom 地址
rom_read_data	wire	32	i	rom 读数据
rom_write_data	wire	32	o	rom 写数据

ram_en	wire	1	o	ram 使能
ram_write_en	wire	4	o	ram 写使能
ram_addr	wire	32	o	ram 地址
ram_read_data	wire	32	i	ram 读数据
ram_write_data	wire	32	o	ram 写数据
debug_reg_write_en	wire	1	o	用于测试的信号
debug_reg_write_addr	wire	5	o	用于测试的信号
debug_reg_write_data	wire	32	o	用于测试的信号
debug_pc_addr	wire	32	o	用于测试的信号

2.2 存储模块

2.2.1 ROM

ROM 模块在流水线中的位置如图 2-1 蓝色部分所示：

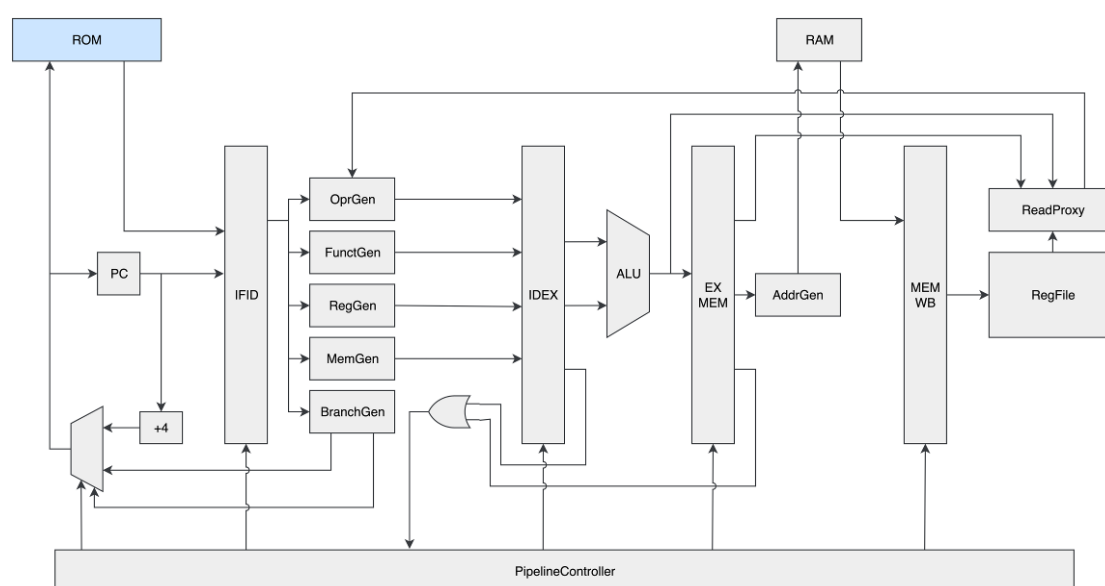


图 2-1 ROM 模块

ROM 作为只读存储，用于保存我们预先编译好的程序，在 CPU 上电运行后从 ROM 的 0 号地址读取第一条指令开始执行。

需要说明的是，该模块仅做仿真用途。而且在任何一个实际的 CPU 中，CPU 的核心内部并不会包含类似 ROM 和 RAM 的结构。即，该模块和之后实现的 RAM 模块不属于 CPU 的一部分。由于 ROM 和 RAM 属于存储资源而非控制逻辑，在芯片上很容易占用大量的面

积，所以 CPU 的核心只会对外暴露控制 ROM 和 RAM 的端口而不会包括任何相关实现。

一般情况下，对 ROM 和 RAM 的任何读写操作都会被反映到总线上，而为了提升访存的速度，市面上任何一款 CPU 都会在核心和总线之间放置 L1、L2 缓存。这些缓存对于核心的其他部分来说，实际上就充当了 ROM 和 RAM 的作用。但是考虑到实现的复杂性，我们的 CPU 中并没有任何缓存结构，取而代之的是更为简单的 ROM 和 RAM 控制接口。

接下来给出 ROM 模块的接口定义：

（注：模块中的写使能与写数据端口实际未被使用）

ROM 的信号定义

名称	类型	宽度	方向	用途
clk	wire	1	i	时钟信号
rom_en	wire	1	i	使能信号，1 为有效
rom_write_en	wire	4	i	写使能位
rom_addr	wire	32	i	指令地址
rom_write_data	wire	32	i	写数据
rom_read_data	reg	32	o	输出的指令

这里给出 ROM 的实现：

```
`timescale 1ns / 1ps

`include "pcdef.v"
`include "bus.v"
`include "sim.v"

module ROM(
    input                clk,
    input                rom_en,
    input                [`MEM_SEL_BUS] rom_write_en,
    input                [`ADDR_BUS] rom_addr,
    input                [`DATA_BUS] rom_write_data,
    output reg [`DATA_BUS] rom_read_data
);
    // `INST_MEM_BUS 1023:0
    // inst_mem 1KB
    reg[7:0] inst_mem[`INST_MEM_BUS];
```

```

// initialize with program
initial begin
    $readmemh("inst_rom.bin", inst_mem);
end

wire[`ADDR_BUS] addr = rom_addr - `INIT_PC;

always @(posedge clk) begin
    if (!rom_en) begin
        rom_read_data <= 0;
    end
    else begin
        rom_read_data <= {
            inst_mem[addr[`INST_MEM_ADDR_WIDTH - 1:0] + 0],
            inst_mem[addr[`INST_MEM_ADDR_WIDTH - 1:0] + 1],
            inst_mem[addr[`INST_MEM_ADDR_WIDTH - 1:0] + 2],
            inst_mem[addr[`INST_MEM_ADDR_WIDTH - 1:0] + 3]
        };
    end
end

endmodule // ROM

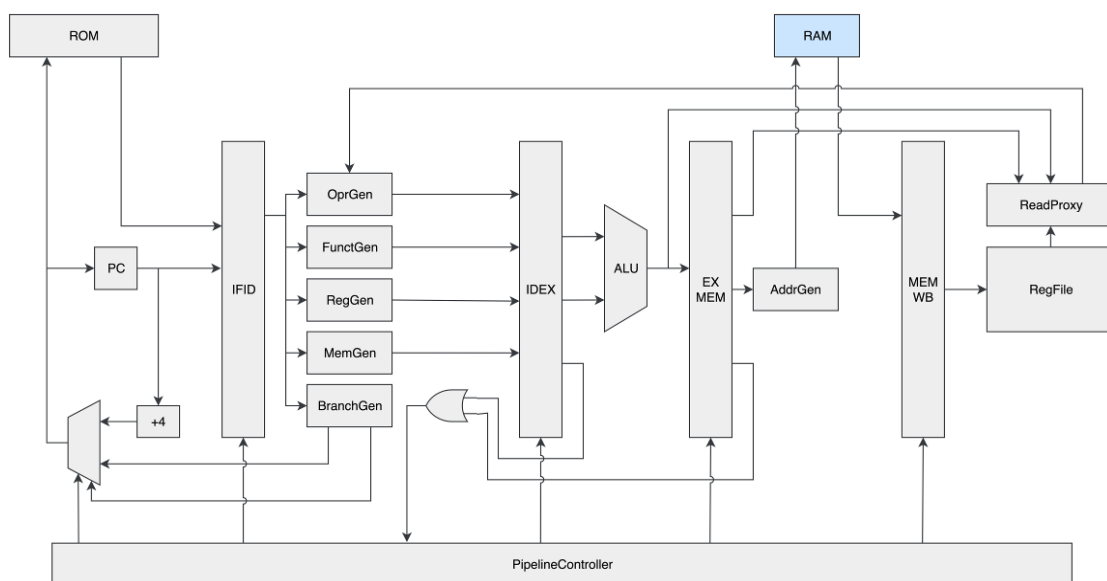
```

注意上面的 ROM 实现中，存储器的最小单元是 1 字节而不是 4 字节。这种设计是考虑到 GCC 的编译输出格式，它就是以 1 字节为最小单元。为了能够用上 GCC 为我们编译汇编程序，就把 ROM 的最小单元也变成 1 字节。

`readmemh` 函数中文件路径请大家自行修改成 `.bin` 文件所在的绝对路径，同时在输入路径时与 C 语言相同，“\”表示转义符，因此需要写作“\\”或“/”。

2.2.2 RAM

RAM 模块在流水线中的位置如图 2-2 蓝色部分所示：



RAM 用于保存数据。RAM 在设计当中，对读取是异步的，给出地址就能得到数据；写入是同步的，只当时钟上升沿的时候才会写入。因此 RAM 的设计会相对复杂。与 ROM 模块一致，该模块同样仅做仿真用途。

RAM 的信号定义

名称	类型	宽度	方向	用途
clk	wire	1	i	时钟信号
ram_em	wire	1	i	使能信号，1 为有效
ram_write_en	wire	4	i	位写使能
ram_addr	wire	32	i	指令地址
ram_write_data	wire	32	i	写数据
ram_read_data	wire	32	o	输出的指令

注意：

1. RAM 设计时大小设为 256KB 即可。
2. 对于一个 32 位的数据，我们按字分为 4 个部分，ram_wirte_en 用来控制写入 RAM 的位置。当 ram_wire_en=4'b1111 时，我们对 32 位进行写入；为 4'b0001 时，写入低 8 位；为 4'b1100 时，对高 16 位进行写入。

这里给出 ROM 的设计:

```
`timescale 1ns / 1ps
```

```

`include "bus.v"
`include "sim.v"

module RAM(
    input                clk,
    input                ram_en,
    input                [`MEM_SEL_BUS] ram_write_en,
    input                [`ADDR_BUS] ram_addr,
    input                [`DATA_BUS] ram_write_data,
    output reg [`DATA_BUS] ram_read_data
);

    reg[7:0] data_mem0[`DATA_MEM_BUS];
    reg[7:0] data_mem1[`DATA_MEM_BUS];
    reg[7:0] data_mem2[`DATA_MEM_BUS];
    reg[7:0] data_mem3[`DATA_MEM_BUS];

    // write operation
    always @(posedge clk) begin
        if (ram_en && |ram_write_en) begin
            if (ram_write_en[3]) data_mem3[ram_addr[`DATA_MEM_ADDR_WIDTH + 1:2]] <= ram_write_data[31:24];
            if (ram_write_en[2]) data_mem2[ram_addr[`DATA_MEM_ADDR_WIDTH + 1:2]] <= ram_write_data[23:16];
            if (ram_write_en[1]) data_mem1[ram_addr[`DATA_MEM_ADDR_WIDTH + 1:2]] <= ram_write_data[15:8];
            if (ram_write_en[0]) data_mem0[ram_addr[`DATA_MEM_ADDR_WIDTH + 1:2]] <= ram_write_data[7:0];
        end
    end

    // read operation
    always @(*) begin
        if (!ram_en || |ram_write_en) begin
            ram_read_data <= 0;
        end
        else begin
            ram_read_data <= {
                data_mem3[ram_addr[`DATA_MEM_ADDR_WIDTH + 1:2]],
                data_mem2[ram_addr[`DATA_MEM_ADDR_WIDTH + 1:2]],
                data_mem1[ram_addr[`DATA_MEM_ADDR_WIDTH + 1:2]],
                data_mem0[ram_addr[`DATA_MEM_ADDR_WIDTH + 1:2]]
            }
        end
    end
endmodule

```

```

    };
end
end

endmodule // RAM

```

2.2.3 RegFile

RegFile 在流水线中的位置如图 2-3 蓝色部分所示：

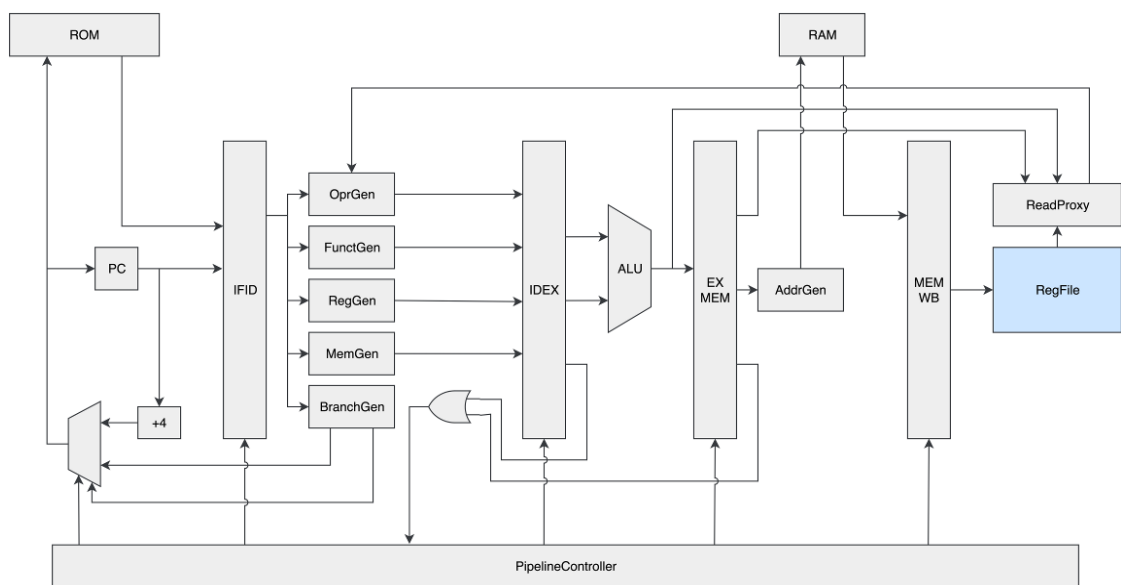


图 2-3 RegFile 模块

RegFile 模块表示 CPU 内部的寄存器堆，用于存放 MIPS 中使用的 32 个通用寄存器。考虑到大部分 MIPS 指令会同时读取两个寄存器中的数据，因此我们在 RgeFile 设计时需要设置两个读端口以及一个写端口。同样，寄存器堆对于读数据是异步的，写数据是同步（需要上升沿触发）的。其接口定义如下：

RegFile 的信号定义

名称	类型	宽度	方向	用途
clk	wire	1	i	时钟信号
rst	wire	1	i	复位信号
read_en_1	wire	1	i	读使能信号 1

read_addr_1	wire	5	i	读寄存器地址 1
read_data_1	reg	32	o	读出数据 1
read_en_2	wire	1	i	读使能信号 2
read_addr_2	wire	5	i	读寄存器地址 2
read_data_2	reg	32	o	读出数据 2
write_en	wire	1	i	写使能信号
write_addr	wire	5	i	写寄存器地址
write_data	wire	32	i	写入数据

注意：0 号寄存器不允许写入，任何试图写入 0 号寄存器的操作都应该被忽略。

【1】请完成寄存器堆模块 (RegFile.v)，并在 CG 平台完成 2-1 的自动评测。

2.3 取指阶段 IF

2.3.1 PC

本节实现的部分如图 2-4：

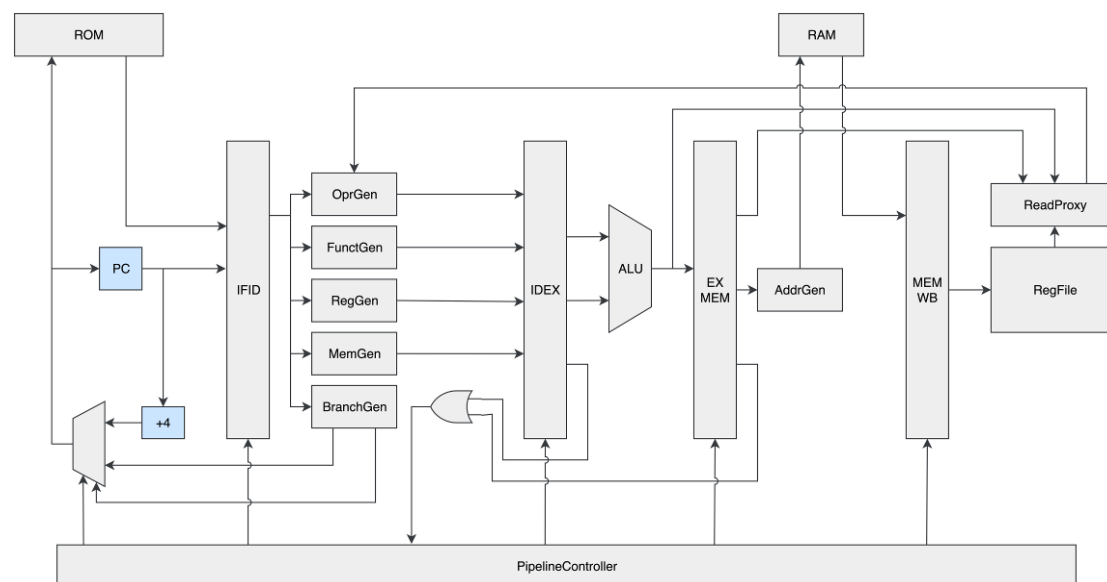


图 2-4 PC 模块

在取指阶段核心部件为 PC，其主要功能用于更新指令地址，并将指令地址传输给 ROM。

其信号定义如下：

PC 模块信号定义

名称	类型	宽度	方向	用途
clk	wire	1	i	时钟信号
rst	wire	1	i	复位信号
stall_pc	wire	1	i	用于暂停 pc
branch_flag	wire	1	i	分支跳转信号
branch_addr	wire	32	i	分支跳转地址
pc	reg	32	o	当前指令的 pc 值
rom_en	reg	1	o	rom 使能
rom_write_en	wire	4	o	rom 写使能
rom_addr	wire	32	o	rom 地址信号
rom_write_data	wire	32	o	rom 写数据

现阶段因为不考虑冒险与暂停，因此我们在此可以不进行分支跳转与暂停的实现。当 rst 信号为 1 时，且在时钟上升沿时进行复位，将 rom_en 使能置为 0；当 rst 不为 1 时，rom_en 置 1。若 rom_en 为 1 且时钟周期上升沿时对 PC 进行加 4，也就是获取下一条指令的地址。

我们可以通过在时钟上升沿检测 rom_en 来实现对 PC 的更新。当 rom_en 为 1 时，PC 自动取下一条指令，也就是将 PC 加 4。当 rom_en 为 0 时，我们将 PC 复位。**注意在时钟上升沿来临的时候，我们需要通过 rom_en 判断，而不是 rst，这是为了使 pc 能够从初始地址开始访问。**

接口定义中所有“rom”前缀的信号均为对 ROM 的控制信号。PC 级除了需要设置 rom_en 为 1 之外，还需要根据当前 PC 的值来控制从 ROM 取指令的地址，即 rom_addr。参考之前实现的 ROM 模块，我们很容易发现：**ROM 在收到读请求之后，会在一拍后返回读取的数据**。而目前的 PC 级会向之后的 ID 级同时传送当前指令以及指令的 PC，这步操作同样需要在一拍内完成。

试想一下：如果我们直接将 rom_addr 设为当前 PC 的值，那么 ROM 会在下一拍返回当前指令；与此同时，当前的 PC 已经在这一拍传送到了 ID 级，而 ID 级是不可能收到下一拍传来的对应指令的。也就是说，如果这样处理，ID 级收到的 PC 和指令的值永远会存在一拍的错位。这要怎么办呢？

其实解决方法很简单：在向 ROM 发送读请求时，为了保证 ID 级能收到之前一拍的指令，我们可以将 PC 级的 rom_addr 设为下一拍的 PC，而非当前的 PC，这样就可以抵消之前存在的一拍的错位了。为了方便编码，我们可以设置一个 wire 型变量 “next_pc”，将其值设为 PC+4。每次在时钟上升沿更新 PC 时，我们直接将 PC 设为 “next_pc”；同时，我们也需要直接将 rom_addr 赋值为此量。而为了保证 PC 从 ROM 中取到的指令为初始地址处的指令，我们在复位时应该将 PC 设为 “初始 PC” 减 4。在 TinyMIPS 当中，取指的初始地址应设为 32'hbfc00000。

上述内容可能比较难以理解，建议同学们分别尝试将 rom_addr 设为当前 PC 和下一 PC 的值，并将 PC 级、ROM 和之后实现的 IFID 级连接，然后观察从 IFID 级输出的指令及其 PC 的情况。你应该能从仿真波形中发现两种方法的差异。【代码分析报告中需要给出差异说明】

【2】 请完成流水线取指级模块 (PC.v)，并在 CG 平台完成 2-2 的自动评测。

2.3.2 IFID

本节实现的部分如图 2-5 所示：

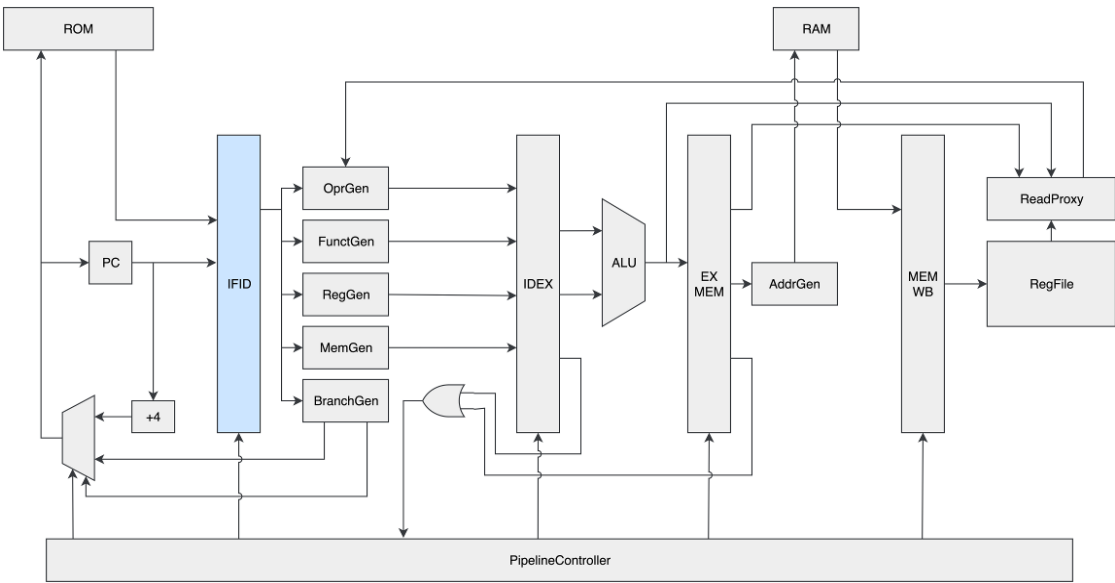


图 2-5 IFID 模块

IFID 是流水线 IF 级与 ID 级之间的时序逻辑部件。该模块接受 PC 级输出的地址与指令

作为输入，然后在上升沿时锁存模块的输入到输出。此处我们可以通过实例化两个 D 触发器对这些信号进行处理，也就是使用之前的带参数模块进行设计，简化代码量。

IFID 模块信号定义

名称	类型	宽度	方向	用途
clk	wire	1	i	时钟信号
rst	wire	1	i	复位信号
stall_current_stage	wire	1	i	暂停当前阶段信号
stall_next_stage	wire	1	i	暂停下一阶段信号
addr_in	wire	32	i	地址输入
inst_in	wire	32	i	指令输入
addr_out	wire	32	o	地址输出
inst_out	wire	32	o	指令输出

这里给出 IFID 的设计代码：

```
module IFID(  
    input          clk,  
    input          rst,  
    input          stall_current_stage,  
    input          stall_next_stage,  
    input  [`ADDR_BUS] addr_in,  
    input  [`INST_BUS] inst_in,  
    output [`ADDR_BUS] addr_out,  
    output [`INST_BUS] inst_out  
);  
  
    PipelineDeliver #(`ADDR_BUS_WIDTH) ff_addr(  
        clk, rst,  
        stall_current_stage, stall_next_stage,  
        addr_in, addr_out  
    );  
  
    PipelineDeliver #(`INST_BUS_WIDTH) ff_inst(  
        clk, rst,  
        stall_current_stage, stall_next_stage,  
        inst_in, inst_out  
    );  
;
```

```
endmodule // IFID
```

这里的 PipelineDeliver 模块就是我们设计的 D 触发器，“#”后的元素表示对 D 触发器中的线宽参数进行重定义。

【3】 请完成 PipelineDeliver 模块 (*PipelineDeliver.v*)，结合给出的流水线取指-译码中间级模块 (*IFID.v*) 在 CG 平台完成 2-3 的自动评测。

2.4 译码阶段 ID

本节实现的部分如图 2-6 所示：

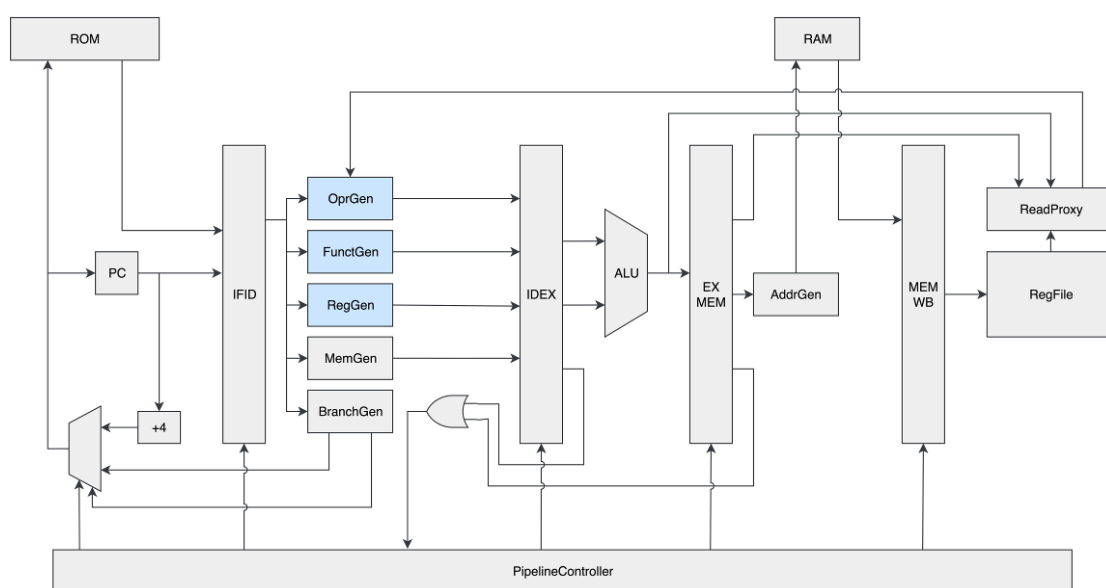


图 2-6 ID 模块

流水线译码阶段的主要工作是：

1. 将 IF 阶段传递过来的指令进行译码，生成各类控制信号，并将这些信号传递给流水线的其余部件（IF、EX、MEM、WB）；
2. 判断指令类型并得到指令的操作数：如果为 I 型（立即数型）指令，则从指令中读取其中的立即数；如果为 R 型（寄存器型）指令，则通过指令内的寄存器编号将数据从 RegFile 中取出；
3. 获取指令的目的寄存器号，并将此信息作为输出传递到 WB 级，**以便于将指令的执**

行结果写回 RegFile。

在 ID 中主要有三个部件：组合逻辑的流水级 ID、寄存器堆 RegFile、时序逻辑的中间级 IDEX。其中寄存器堆我们已经实现了，IDEX 同样可以通过 D 触发器进行实现，这里我们主要讨论 ID 的实现。


首先是 ID 模块的信号定义：

ID 模块信号定义


名称	类型	宽度	方向	用途
addr	wire	32	i	从 IF 得到的 pc 取指地址
inst	wire	32	i	从 IF 得到的指令内容
load_related_1	wire	1	i	EX 中 load 指令标志
load_related_2	wire	1	i	MEM 中 load 指令标志
reg_read_en_1	wire	1	o	给 RegFile 的使能信号 1
reg_addr_1	wire	32	o	给 RegFile 的地址 1
reg_data_1	wire	32	i	从 RegFile 得到的数据 1
reg_read_en_2	wire	1	o	给 RegFile 的使能信号 2
reg_addr_2	wire	32	o	给 RegFile 的地址 2
reg_data_2	wire	32	i	从 RegFile 得到的数据 2
stall_request	wire	1	o	暂停请求（暂不使用）
branch_flag	wire	1	o	跳转信号（暂不使用）
branch_addr	wire	32	o	跳转地址（暂不使用）
funct	wire	6	o	给 EX 的操作码
shamt	wire	5	o	给 EX 的位移数
operand_1	wire	32	o	给 EX 的操作数 1
operand_2	wire	32	o	给 EX 的操作数 2
mem_read_flag	wire	1	o	给 MEM 的读信号
mem_write_flag	wire	1	o	给 MEM 的写信号
mem_sign_ext_flag	wire	1	o	给 MEM 的符号拓展信号
mem_sel	wire	4	o	给 MEM 的写位置信号

mem_write_data	wire	32	o	给 MEM 的写数据
reg_write_en	wire	1	o	给 WB 的写寄存器信号
reg_write_addr	wire	5	o	给 WB 的写寄存器地址
current_pc_addr	wire	32	o	给下一级的该指令地址


注意：传输给 MEM 与 WB 的信号都不能直接传输给相应部件，只能够依次传输给前面的模块，最后再传输给接收模块，这是由于流水线本身结构所导致的。shamt 可以直接通过 inst 中 shamt 字段进行输出。

接下来就是实现 ID 了，这里我们以 ADDIU 与 ADDU 指令为例子进行讲解。通过查阅手册可以知道 ADDIU 指令 OP 为 6'b001001,因此我们在 opcode.v 添加该指令 OP_ADDIU 的宏定义。继续查阅 ADDU 指令 OP 为 6'b000000,表示 SPECIAL, 因此我们也在 opcode.v 添加 OP_SPECIAL 的宏。在 MIPS 指令当中存在很多以 OP_SPECIAL 为开头的指令，这些指令主要通过 funct 段进行区分。

同时我们可以查阅 MIPS 手册，我们发现 ADDIU 与 ADDU 指令除了操作数的来源不同（一个是寄存器，一个是立即数）和写入寄存器不同（一个是 rd，一个是 rt）以外，在 ALU 看来其实做的都是同一种运算（加法）。

通过 ID 传递给 WB 的写入寄存器地址信号和相关使能信号，我们可以控制 WB 模块写入寄存器的位置。对于操作数来说，我们需要获得指令中的两个操作数，然后将其赋值给输出信号 operand_1 与 operand_2 即可。这两组数据被发送到 EX 级，由 EX 级执行具体的运算操作。

除此之外，我们可以考虑：EX 级除了两个操作数外，还需要知道什么信息？答案显而易见：EX 级实际上就是一个 ALU，它负责根据操作码将两个操作数做具体的运算，然后再将输出传递到下一级。所以 ID 级还需要生成 EX 级具体执行的运算的操作码。就目前这个例子而言，EX 模块只需进行一个加法操作，而具体这个操作来自指令 ADDIU 还是 ADDU，模块本身并不关注。因此我们可以做一个归一化处理：对 ADDIU 和 ADDU 生成同样的操作码，这样做的好处是可以简化 EX 级的实现。归一化体现在 ID 级中，就是输出相同的 funct 信号。这里我们可以对两个指令都输出 ADDU 的 funct。

在 ID 模块的编写时，我们不推荐将全部的译码环节写进 ID 模块，这样会导致一个模块内的代码过多，可读性下降。此处我们推荐将 ID 模块分解为多个子模块进行设计：

BranchGen.v: 分支译码部件，主要用于判断分支和跳转指令，并产生相关信号。当前

章节暂不讨论。

FuncGen.v: 操作码 funct 生成模块，主要用于读取指令中的 funct 段并进行归一化，输出最终提交给 EX 的 funct。这里给出 FuncGen 的部分代码：

```
always @(*) begin
  case (op)
    `OP_SPECIAL: funct <= funct_in;
    `OP_ADDIU: funct <= `FUNCT_ADDU;
    default: funct <= `FUNCT_NOP;
  endcase
end
```

因为当 OP 段为 SPECIAL 时候的指令已经带有了 funct 字段，我们只需将指令自己的 funct 字段赋值给输出的 funct 即可。对于 ADDIU 指令，之前已经介绍过，其在 EX 阶段需执行的操作与 ADDU 相同，于是此时我们可以直接将输出的 funct 设为 ADDU 的 funct。

注意：我们建议大家在所有的 case 块的最后都加上 default。另外，这里的`FUNCT_NOP 并不是 NOP 指令的 funct 字段。为了表示“不执行任何操作”，我们随意选取了一个在 MIPS 指令集中无意义的值，此处使用的是 6'b111111。

RegGen.v: 生成所有与寄存器读写相关的控制信号。对所有 OP_SPECIAL 的指令（R 型指令）而言，我们需要同时读取 rs 与 rt 的内容，并将结果写入 rd 寄存器。而对于 OP_ADDIU 这条 I 型指令，只需读取 rs 寄存器的值（因为另一个操作数是立即数而非寄存器），并将结果写入 rt 寄存器。因此有如下代码：

```
always @(*) begin // generate read address
  case (op)
    `OP_ADDIU: begin
      reg_read_en_1 <= 1;
      reg_read_en_2 <= 0;
      reg_addr_1 <= rs;
      reg_addr_2 <= 0;
    end
    `OP_SPECIAL: begin
      reg_read_en_1 <= 1;
      reg_read_en_2 <= 1;
      reg_addr_1 <= rs;
      reg_addr_2 <= rt;
    end
    // 默认不读寄存器
  default: begin
```

```

        reg_read_en_1 <= 0;
        reg_read_en_2 <= 0;
        reg_addr_1 <= 0;
        reg_addr_2 <= 0;
    end
endcase
end
always @(*) begin // generate write address
    case (op)
        `OP_ADDIU: begin
            reg_write_en <= 1;
            reg_write_addr <= rt;
        end
        `OP_SPECIAL: begin
            reg_write_en <= 1;
            reg_write_addr <= rd;
        end
        default: begin
            reg_write_en <= 0;
            reg_write_addr <= 0;
        end
    endcase
end
end

```

OperandGen.v: 生成 EX 阶段的两个操作数。之前的 RegGen 模块只负责生成读 RegFile 的控制信号，并没有把数据读出来。而 OperandGen 模块要做的就是：判断指令操作数来自寄存器堆还是立即数，然后从 RegFile 或者指令本身中，读出寄存器或立即数的值，作为操作数的输出。

对所有类似 ADDU 的 R 型指令而言，我们需要直接读取 RegFile 的两个数据输出信号作为操作数；对类似 ADDIU 的 I 型指令而言，指令的一个操作数来自寄存器堆，另一个操作数来自指令内的立即数字段。所以，我们需要从 RegFile 读出第一个操作数，然后从指令中读出 16 位的立即数字段并进行符号扩展，作为第二个操作数。因为我们在之前已经向 RegFile 发出了读信号，而寄存器堆的读取是异步的，因此这里使用的 reg_data_1 与 reg_data_2 就是当前对应寄存器的值。此外，sign_ext_imm 进行了立即数字段的符号扩展：

```

// extract immediate from instruction
wire[`DATA_BUS] sign_ext_imm = {{16{imm[15]}}}, imm};
// generate operand_1

```

```

always @(*) begin
  case (op)
    // immediate
    `OP_ADDIU: begin
      operand_1 <= reg_data_1;
    end
    `OP_SPECIAL: begin
      operand_1 <= reg_data_1;
    end
    default: begin
      operand_1 <= 0;
    end
  endcase
end
// generate operand_2
always @(*) begin
  case (op)
    `OP_ADDIU: begin
      operand_2 <= sign_ext_imm;
    end
    `OP_SPECIAL: begin
      operand_2 <= reg_data_2;
    end
    default: begin
      operand_2 <= 0;
    end
  endcase
end
end

```

MemGen.v: 生成输出到 MEM 级的信号，即生成访存指令相关的控制信号。该模块的实现我们目前同样不需要关心。

完成了这 5 个模块，我们只需在 ID.v 下进行实例化并连线，即可完成 ID 模块的实现。当然大家也可以不使用这套方案，而是自行进行设计，只要保持最终 ID 模块接口的一致性以及功能的正确性即可。

【4】请完成流水线译码级模块 (*ID.v*)，结合操作码 *funct* 生成模块 (*OperandGen.v*)、寄存器控制信号生成模块 (*RegGen.v*)、执行阶段操作数生成模块 (*OperandGen.v*) 在 CG 平台完成 2-4 的自动评测。

2.5 执行阶段 EX

本节实现的部分如图 2-7 所示：

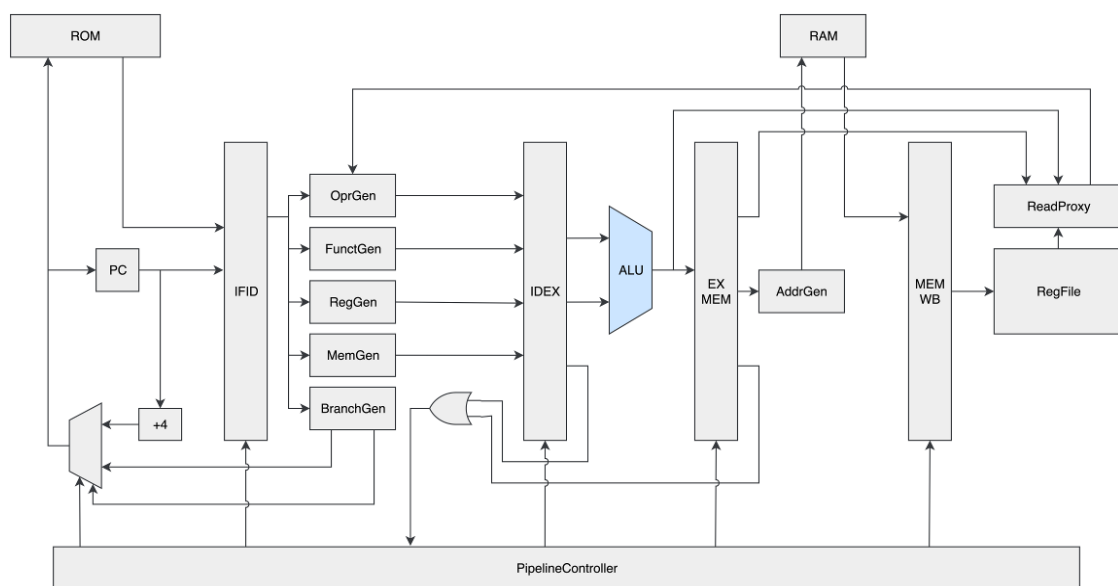


图 2-7 ALU 模块

EX 的功能之前也已经介绍过了，该模块主要对 ID 传入的两个操作数执行具体的运算操作，并将运算结果传递到流水线的下一级。此外，EX 级还会处理访存产生的控制信号，将相关信息交给 MEM 级，但是目前我们不必关心这部分的实现。此处给出 EX 模块的信号定义：

名称	类型	宽度	方向	用途
funct	wire	6	i	操作码
shamt	wire	5	i	位移数
operand_1	wire	32	i	操作数 1
operand_2	wire	32	i	操作数 2
mem_read_flag_in	wire	1	i	给 MEM 的读信号
mem_write_flag_in	wire	1	i	给 MEM 的写信号

mem_sign_ext_flag_in	wire	1	i	给 MEM 的符号拓展信号
mem_sel_in	wire	4	i	给 MEM 的写位置信号
mem_write_data_in	wire	32	i	给 MEM 的写数据
reg_write_en_in	wire	1	i	给 WB 的写寄存器信号
reg_write_addr_in	wire	5	i	给 WB 的写寄存器地址
current_pc_addr_in	wire	32	i	当前指令 PC 值
ex_load_flag	wire	1	o	发生 load 指令
mem_read_flag_out	wire	1	o	给 MEM 的读信号
mem_write_flag_out	wire	1	o	给 MEM 的写信号
mem_sign_ext_flag_out	wire	1	o	给 MEM 的符号拓展信号
mem_sel_out	wire	4	o	给 MEM 的写位置信号
mem_write_data_out	wire	32	o	给 MEM 的写数据
result	reg	32	o	对操作数计算后的结果
reg_write_en_out	wire	1	o	给 WB 的写寄存器信号
reg_write_addr_out	wire	5	o	给 WB 的写寄存器地址
current_pc_addr_out	wire	32	o	当前执行指令 PC 值

注意：这里传递给 MEM、WB 的信号（mem、reg 开头的信号）以及当前 PC 值并不需要进行处理，只需将其输入直接连接到输出即可。

这里我们只需要判断 ID 级输出的 funct 的具体类型，然后对 operand_1 和 operand_2 进行相应的运算操作即可，代码从略，同学们可以自行设计。同样，EX 流水级之后还需要加入 EXMEM 中间级，进行输入输出的锁存。



【5】请完成流水线执行级模块 (EX.V) 并在平台完成 2-5 的自动评测。

2.6 访存阶段 MEM

因为 ADDU 与 ADDIU 没有访存操作，因此在 MEM 模块中，我们暂时不需要处理访存操作，只将输入直接连接到输出即可。同时不要忘了在之后加入 MEMWB 中间级。

MEM 模块的信号定义如下：

名称	类型	宽度	方向	用途
mem_read_flag_in	wire	1	i	MEM 中内存的读信号
mem_write_flag_in	wire	1	i	MEM 中内存的写信号
mem_sign_ext_flag_in	wire	1	i	读取内存数据符号拓展信号
mem_sel_in	wire	4	i	MEM 的写位置信号
mem_write_data	wire	32	i	给 MEM 的写数据
result_in	wire	32	i	输入的计算结果
reg_write_en_in	wire	1	i	寄存器堆写使能
reg_write_addr_in	wire	5	i	寄存器堆写地址
current_pc_addr_in	wire	32	i	当前执行指令 PC 值
ram_en	wire	1	o	ram 使能
ram_write_en	wire	4	o	ram 写使能
ram_addr	wire	32	o	ram 地址
ram_write_data	reg	32	o	ram 写数据
mem_load_flag	wire	1	o	发生 load 指令
mem_read_flag_out	wire	1	o	MEM 中内存的读信号
mem_write_flag_out	wire	1	o	MEM 中内存的写信号
mem_sign_ext_flag_out	wire	1	o	读取内存数据符号拓展信号
mem_sel_out	wire	4	o	MEM 的写位置信号
result_out	wire	32	o	<u>将计算结果传给 WB</u>
reg_write_en_out	wire	1	o	寄存器堆写使能
reg_write_addr_out	wire	5	o	寄存器堆写地址
current_pc_addr_out	wire	32	o	当前执行指令 PC 值

【6】 请完成流水线访存级模块 (**MEM.v**)，并在 CG 平台完成 2-6 的自动评测。

2.7 写回阶段 WB

流水线的最后一个部分是 WB 级，该级会将 EX 级和 MEM 级产生的输出，根据 ID 产生的寄存器堆地址，写回到对应的寄存器中。

这里给出 WB 模块的信号定义：

名称	类型	宽度	方向	用途
ram_read_data	wire	32	i	从内存中读取的数据
mem_read_flag	wire	1	i	MEM 中内存的读信号
mem_write_flag	wire	1	i	MEM 中内存的写信号
mem_sign_ext_flag	wire	1	i	读取内存数据符号拓展信号
mem_sel	wire	4	i	内存的写位置信号
result_in	wire	32	i	EX 中的计算结果
reg_write_en_in	wire	1	i	寄存器堆写使能
reg_write_addr_in	wire	5	i	寄存器堆写地址
current_pc_addr_in	wire	32	i	当前执行指令 PC 值
result_out	reg	32	o	写入寄存器堆的数据
reg_write_en_out	wire	1	o	寄存器堆写使能
reg_write_addr_out	wire	5	o	寄存器堆写地址
debug_reg_write_en	wire	1	o	寄存器堆写使能（debug）
debug_pc_addr_out	wire	32	o	寄存器堆写地址（debug）

因为目前没有实现访存，因此我们只需要将 result、reg_write_en、reg_write_addr 三个信号直接接入 RegFile 即可。两个 debug 开头的信号用于我们对 CPU 进行功能测试，我们这里只需要将 current_pc_addr 与 reg_write_en 两个信号直接连接到对应的输出即可。

【7】 请完成流水线写回级模块 (WB.v)，并在 CG 平台完成 2-7 的自动评测。

2.8 解决数据相关问题

本节实现的部分如图 2-8 所示：

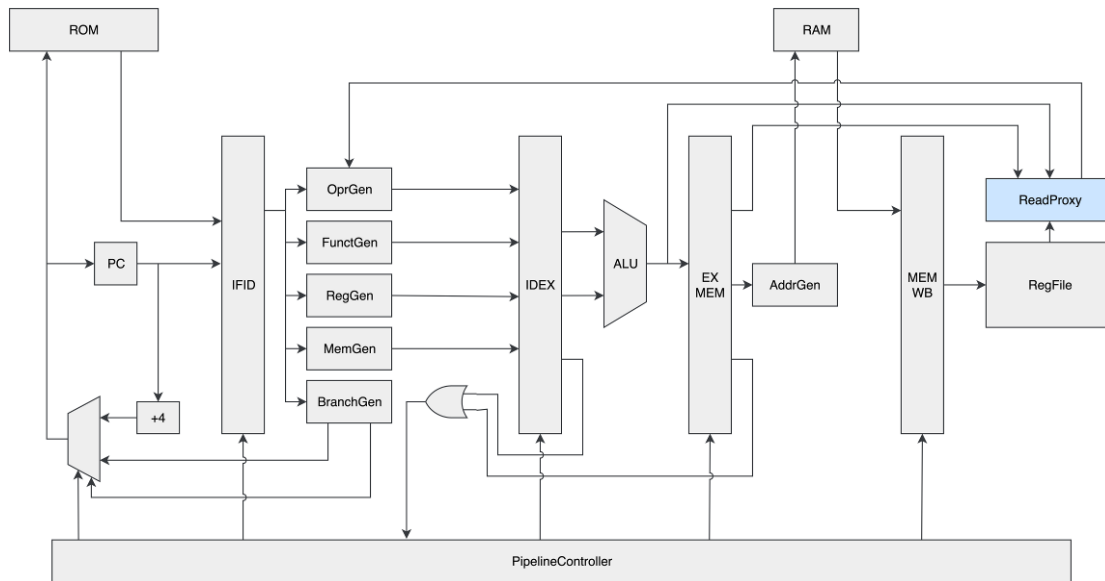


图 2-8 解决数据相关问题

目前我们已经完成了一个可以执行 `ADDU` 与 `ADDIU` 这两条指令的五级流水线 CPU，但是这个 CPU 目前依然会出现一些问题。比如有如下指令序列：

```
addiu $8, $0, 1      # 将 8 号寄存器赋值为 1
addiu $9, $0, 2      # 将 9 号寄存器赋值为 2
addu  $10, $8, $9    # 寄存器 8 和 9 的值相加，赋给 10 号寄存器
```

想象一下：第三条指令正在 ID 级执行译码操作，此时第二条指令刚刚执行到 EX 级，第一条指令还在 MEM 级逗留。在之前的实现中，指令的两个操作数会在 ID 级读出，而指令的执行结果直到 WB 级才能写回到寄存器堆。所以，目前第三条指令尝试读出的 8 号和 9 号寄存器的值并非预期中的 1 和 2，因为前两条修改寄存器的指令还没有执行到 WB 级，寄存器的值也并没有被修改。这就导致第三条指令的执行结果和我们的预期存在偏差。

这类问题被称为**数据相关问题**。CPU 中有三种数据相关的情况（这里假设两条指令 A、B，先执行 A 后执行 B）：

1. **Read After Write, RAW**。A 写入寄存器，B 读取相同的寄存器。由于流水线的特点，A 在 WB 阶段写入寄存器，B 在 ID 访问寄存器，若 A 写入与 B 读取的为同一个寄存器，那么 B 就有可能在 A 完成之前读出寄存器数据，那么该数据从逻辑上显然是不正确的。
2. **Write After Read, WAR**。A 读取寄存器，B 写入相同的寄存器。若 A 读取前 B 先写入了数据，那么 A 拿到的数据就可能有误。这种情况只可能出现在乱序执行的

CPU 中，而我们实现的五级流水线 CPU 保证指令的执行顺序不会被修改，所以不需要考虑这种数据相关。

3. ~~Write After Write~~, WAW。A、B 均写入同一个寄存器，若 B 先写入，则目的寄存器的值可能与预期不符。同上，在我们实现的 CPU 中，这种数据相关也不会出现。

因此通过我们的分析，我们的 CPU 只会出现 RAW 这种数据相关的情况。从 EX 得到结果或是 MEM 取出数据起，直到到 WB 级写回数据，中间还需要经历两个时钟周期——此时，如果有某条指令希望取得前两条指令写入的结果，RAW 相关就会发生。那么我们如何在数据未写回的时候就将其取出呢？这里我们使用数据前递的方式来解决 RAW 相关，也就是通过一个额外的部件将 EX、MEM 级的结果提前传递给 ID 模块。

我们创建一个新的模块 RegReadProxy，它在 RegFile 的读端口上做了一层代理。从 ID 模块来看，它和实际的 RegFile 没有区别。但是从它读取得到的寄存器值都是经过前推且正确的。需要被前推的数据应当是从 EX 级和 MEM 级的输出的结果数据，这些输出端口需要连接到 RegReadProxy。

```
module RegReadProxy(
    // input from ID stage
    input                read_en_1,
    input                read_en_2,
    input                [`REG_ADDR_BUS] read_addr_1,
    input                [`REG_ADDR_BUS] read_addr_2,
    // input from regfile
    input                [`DATA_BUS]    data_1_from_reg,
    input                [`DATA_BUS]    data_2_from_reg,
    // input from EX stage (solve data hazards)
    input                ex_load_flag,
    input                reg_write_en_from_ex,
    input                [`REG_ADDR_BUS] reg_write_addr_from_ex,
    input                [`DATA_BUS]    data_from_ex,
    // input from MEM stage (solve data hazards)
    input                mem_load_flag,
    input                reg_write_en_from_mem,
    input                [`REG_ADDR_BUS] reg_write_addr_from_mem,
    input                [`DATA_BUS]    data_from_mem,
    // load related signals
    output               load_related_1,
    output               load_related_2,
```

```
// reg data output (WB stage)
output reg [`DATA_BUS]    read_data_1,
output reg [`DATA_BUS]    read_data_2
);
```

观察该模块接口的设计代码，忽略所有和“load”相关的信号，我们可以发现模块的输入分为四个部分：第一部分用于接受 ID 级的寄存器读取请求；第二部分接管了 EX 级的寄存器写入，包括寄存器的写使能、写地址和写数据；第三部分接管了 MEM 级的寄存器写入；第四部分输出了前递后的最终数据。

前递的思路其实很简单：如果模块发现 ID 正在读取的数据正好是 EX 或是 MEM 模块要写入的，那么就要将输出设为 EX 或 MEM 级的写入数据；反之直接输出 RegFile 中读取的数据。

注意：这里我们需要先处理 EX 级前递的数据，再处理 MEM 级的前递。因为 EX 级离 ID 级更近，产生的数据相对更新。

【8】请完善寄存器读操作的数据前递 (*forwarding*) 模块 (*RegReadProxy.v*)，并在 CG 平台完成 2-8 的自动评测。

目前的流水线实现对数据冲突的处理尚不完善。在之后的章节中，我们会对其进行进一步修改。

第三章 流水线的改进

3.1 跳转指令的实现

3.1.1 延迟槽

对于使用了时序逻辑设计的 PC 模块，我们需要考虑延迟槽的存在，在执行了跳转指令后，CPU 的 PC 部分不能对指令立即跳转，而是要先执行跳转指令后面的一条指令后，再进行跳转，但是对于跳转指令后面的那条指令理论上不应该进行实行，那么延迟槽就自然的出现了。

对于我们设计的 CPU，当跳转指令到达 ID 阶段时，才会向 IF 发出跳转信号，但是此时下一条指令已经被 IF 模块取出。当上升沿再次来临时，跳转指令进入 EX，而它后面的一条指令则进入了 ID 被译码，而此时的 PC 才按照我们的跳转信号对 PC 进行更新，完成跳转的任务。

3.1.2 PC 模块修改

本节实现的部分如图 3-1 所示：

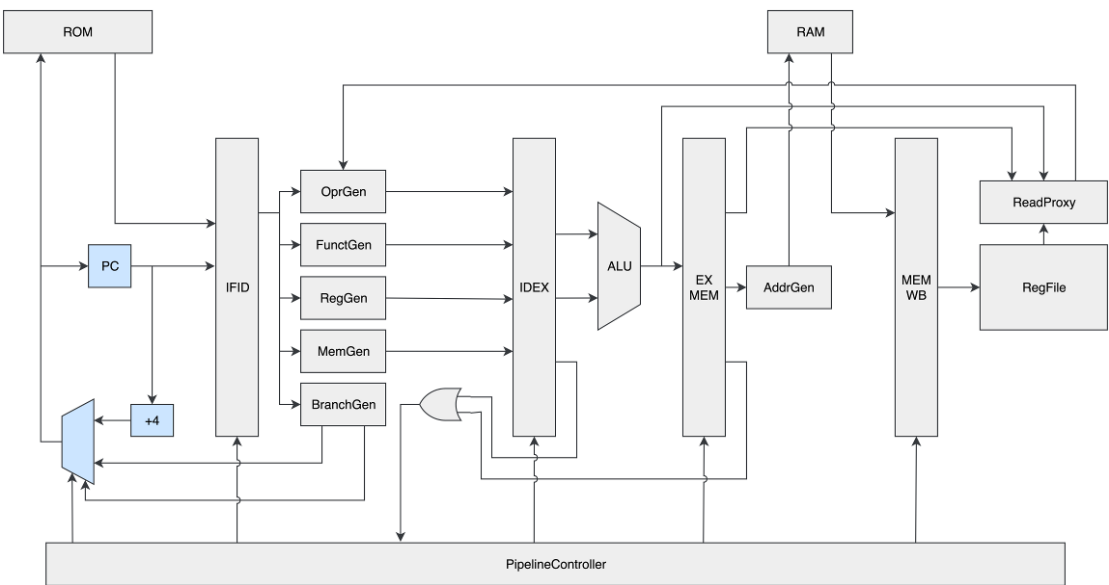


图 3-1 考虑分支后的 PC 模块修改

【10】修改流水线译码级模块 (ID.v)，使译码模块能够正确处理

BEQ, BNE, JAL 和 JALR 四种指令，并生成正确的分支控制信号和寄存器堆控制信号。在 CG 平台完成 3.1-2 的自动评测。

3.2 访存指令的实现

本节实现的部分如图 3-3 所示：

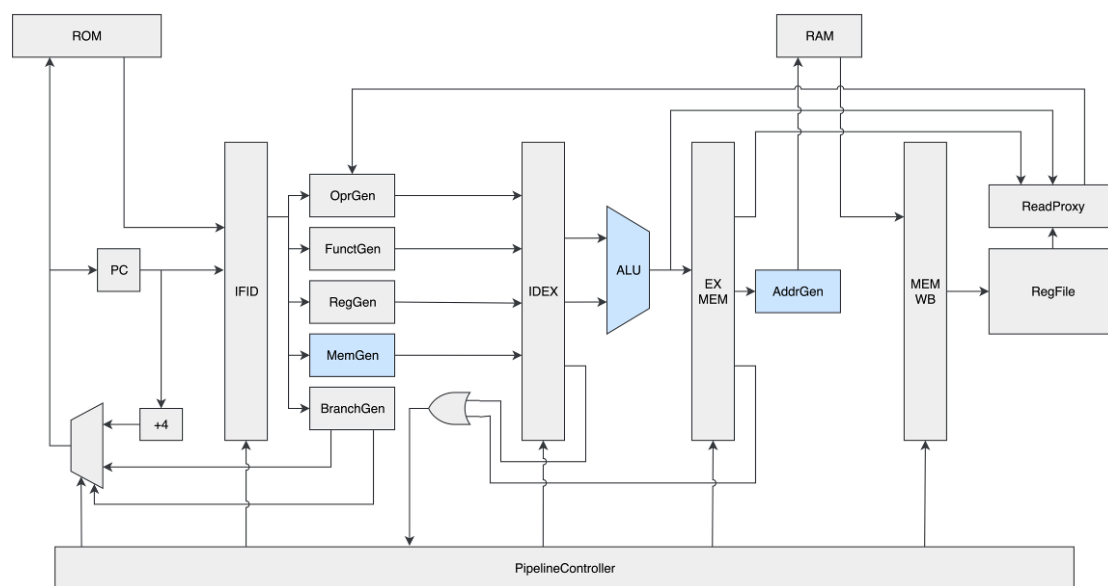


图 3-3 实现访存指令对模块的修改

到现在为止，我们的 CPU 还没有运用到 RAM。在 MIPS 中访存指令分为读取（load）与写入（save）两种。本次实现的指令当中访存指令有 LB、LBU、LW,写入指令有 SB、SW，这里我们以 LW 与 SW 为例实现访存指令。

通过查阅手册可以知道，LW 与 SW 都是 I 型指令，并且他们要访问的内存地址 $\text{addr} = \text{rs 寄存器值} + \text{对立即数进行符号扩展后的值}$ 。

那么我们为了实现访存指令，我们需要对 ID、EX 与 MEM 进行部分的修改：

对于 ID 模块，我们需要对已经定义了的信号进行具体的实现：mem_read_flag，表示是否读取内存；mem_write_flag，表示是否写内存；mem_sign_ext_flag，表示是否需要对读出的信号进行符号扩展；mem_sel，用来表示数据的长度，当 mem_sel 为 4'b1111 时表示读取/写入 4 个字节内容，当 mem_sel 为 4'b0001 时表示只读取/写入一个字节的內容（LB、LBU、SB 均是对一个字节内容进行操作）；mem_write_data，表示写入内存的数据。

因为在 TinyMIPS 当中所有访存指令，都需要通过寄存器的值与立即数符号拓展后进行加法来得到地址，因此 funct 字段我们在 ID 当中设定为 FUNCT_ADDU，从而让其在 EX 模块中进行加法操作。

【11】 修改流水线译码级模块 (ID.v)，使得译码模块能够正确处理 LB, LBU, LW, SB 和 SW 五种指令，并生成正确的寄存器堆控制信号、操作数和操作码 (funct)、访存控制信号。在 CG 平台完成 3.2-1 的自动评测。

对于 EX 模块，我们令访存指令的操作码为加法操作码，并且前面我们已经实现了加法的操作，因此这里我们不需要进行进一步的修改。

【12】 修改流水线执行级模块 (EX.v)，使得执行级模块能够正确处理访存控制信号（直接将输入作为模块端口的输出即可）。在 CG 平台完成 3.2-2 的自动评测。

对于 MEM 模块，因为该模块我们之前并没有进行设计，因此这里进行详细的介绍。

首先当 mem_read_flag 或 mem_write_flag 为 1 的时候，表示我们需要进行内存的操作，这时我们需要将 ram_en 设置为 1。

读取操作时，我们只需要输入地址就可以从 ram 当中得到我们需要的数据。**注意：**对于地址需要将后两位进行置 0 的操作，这时由于在内存当中都是以字对齐的方式进行访问的，因此即使我们只需要一个字节内容，也需要将这个字节所在位置的一个字全部读取出来。

当进行写操作时，首先判断 mem_write_flag 信号是否为 1。因为我们的 MIPS 指令当中存在只对一个字节进行写入的指令，因此我们要对 mem_sel_in 进行判断：当 mem_sel_in 为 4'b1111 时，只需要将 ram_write_sel 置为 4'b1111，表示正常的一次写入操作；当 mem_sel_in 为 4'b0001 时，表示对一个字节进行写入，这时我们要判断写入地址的后两位，这是用来判断我们需要对那一个字节进行写入的，并对输入数据进行判断，判断如下：

address[1:0]	ram_write_sel	ram_write_data
2'b00	4'b0001	mem_write_data
2'b01	4'b0010	mem_write_data<<8
2'b10	4'b0100	mem_write_data<<16
2'b11	4'b1000	mem_write_data<<24

注意：虽然我们可能只需要写入一个字节，但是对于 RAM 依然需要让地址从 {address[31:2],2'b0} 进行操作，这与 RAM 的结构有关。

【13】 修改流水线访存级模块 (**MEM.v**)，使得访存级模块能够根据访存控制信号，生成正确的 RAM 控制信号，以及交给写回级的访存控制信号。在 CG 平台完成 3.2-3 的自动评测。

对于 WB 模块，我们只需要在之前的设计上，对访存指令从内存读取的数据进行判断，当前指令为读取内存时，若 mem_sel_in 为 4'b1111 表示读取一个字的内容，此时将读取后的数据直接写入 RegFile 即可；若 mem_sel_in 为 4'b0001 表示只读取一个字节，与上面 MEM 模块类型，此时要进行地址后两位的判断，并对相应字节内容进行符号扩展后，再写入 RegFile。

【14】 修改流水线写回级模块 (**WB.v**)，使得写回级模块能够根据访存控制信号 在发生 RAM 读取时，生成正确的读取结果 (result)。在 CG 平台完成 3.2-4 的自动评测。

3.3 流水线暂停的实现

本节实现的部分如图 3-4 所示：

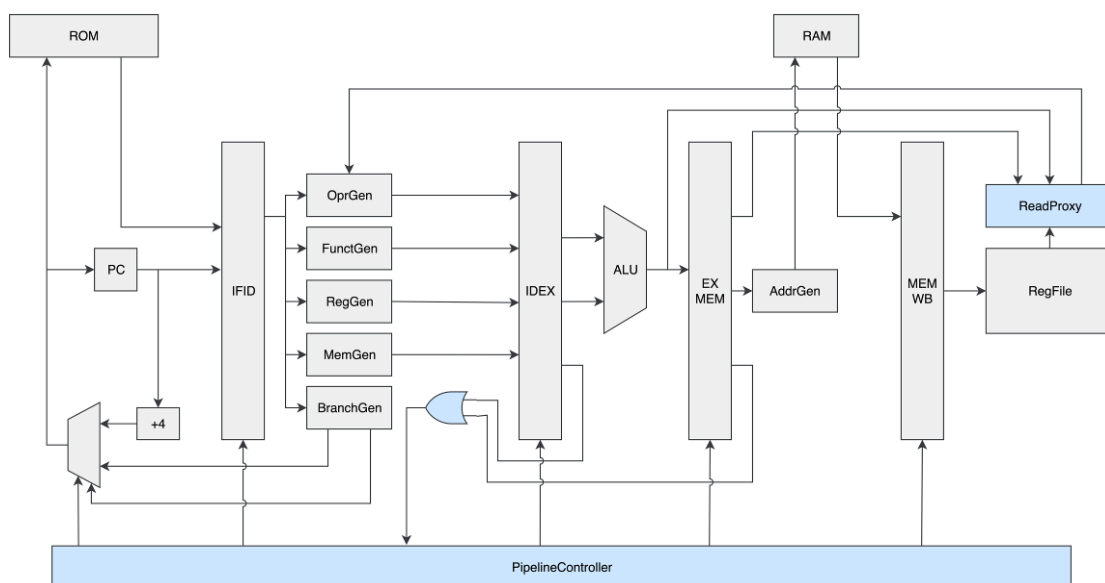


图 3-4 支持流水线暂停需要修改的模块

虽然我们已经完成了数据相关的处理，但是我们的访存指令实际上还存在 `load` 相关的问题：这是因为当 `load` 指令在 `MEM` 阶段才能够拿到所需要的数据，但是在这时 `load` 指令之后的一条指令 `A` 已经运行到了 `EX` 阶段，那么显然如果 `load` 指令此时写入的寄存器与指令 `A` 读取的是同一个寄存器，那么必定会导致 `A` 指令拿到错误的数

据。而我们之前所用到的数据前递其实并没有解决这里的问题，因为之前数据前递其实只能解决 `ID` 阶段读取 `EX` 阶段时已经计算出来的数据，而无法正确的提前得到 `MEM` 读取的数据。那么我们就引入流水线暂停的机制，如果 `ID` 模块检测到 `load` 相关的情况发生，那么我们就暂停 `ID` 之前的流水线，让后面的 `EX`、`MEM` 和 `WB` 先继续执行，而之前的信号继续维持，直到我们 `load` 指令到达 `WB` 阶段，数据已经从 `RAM` 取出后，我们再解除流水线的暂停，此时我们之前设计的数据相关模块 `RegReadProxy` 就可以正确的将数据进行前递，从而解决 `load` 指令的数据相关问题。接下来我们开始实现流水线的暂停：

首先我们要实现一个流水线暂停的控制模块，该模块用于将组合逻辑中的流水线暂停请求信号进行接收，并发出信号暂停的信号。我们将其定义为 `PipelineController`。它的接口如下：

名称	类型	宽度	方向	用途
<code>request_from_id</code>	wire	1	i	来自 id 的暂停请求
<code>stall_all</code>	wire	1	i	CPU 外部的暂停信号
<code>stall_pc</code>	wire	1	o	暂停 pc 阶段
<code>stall_if</code>	wire	1	o	暂停 if 阶段数据
<code>stall_id</code>	wire	1	o	暂停 id 阶段数据
<code>stall_ex</code>	wire	1	o	暂停 ex 阶段数据
<code>stall_mem</code>	wire	1	o	暂停 mem 阶段数据
<code>stall_wb</code>	wire	1	o	暂停 wb 阶段数据

这里我们主要是暂停流水线当中的时序模块，但是本次用于处理 `load` 指令中数据冲突问题时，我们只对 `pc`、`if` 和 `id` 的数据暂停输出操作，而其余的部件暂停用于处理其他的问题上，感兴趣的同学可以自行了解。

该部件是一个组合逻辑部件，实现十分简单，当 `CPU` 外部暂停信号为 1 时，我们将全部的暂停信号进行拉高，暂停全部模块；当 `request_from_id` 为 1 时，我们需要对 `pc`、`if` 和 `id` 的数据暂停输出操作，也就是将 `stall_pc`、`stall_if` 和 `stall_id` 置为 1，其余置为 0 即可；其

余条件下关闭这些暂停。

接下来我们看一下流水线暂停机制的实现过程。

首先我们来看一下 EX 与 MEM 中 load_flag 信号，该信号用于告诉 id 阶段，当前正在执行一个 load 指令，有可能会出现 load 相关的可能。对于该信号的实现也十分的简单，我们只需要将 mem_read_flag 信号连接到 load_flag，因为出现了读取内存的操作，那么当前指令就存在出现 load 数据相关的可能性。

接下来我们将分别来自 EX 与 MEM 的 load_flag 与 mem_load_flag 信号传送给我们的前递模块 RegReadProxy 进行判断，如果存在 load 相关，需要数据前递。这里的判断与数据相关判断相似，我们需要判断两个需要读取的寄存器地址是否与 EX 或是 MEM 中指令需要写入的寄存器相同，并且当前阶段是否存在 load 相关的可能，若写入寄存器相同且存在 load 相关可能，那么我们就将这个读取的寄存器的 load 相关信号 load_related 信号置为 1 传递给 ID 模块。

之后我们会在 ID 模块中接收到两个读取寄存器的 load_related 信号，当至少一个 load 相关信号为 1 时，我们触发流水线暂停请求。也就是产生了 request_from_id 信号。

这是 PipelineController 会接受到置为高电平的 request_from_id 信号，就会对 pc、if 和 id 的数据暂停输出操作，直到出现 load 相关的指令运行到 WB 阶段，这是没有 load 相关出现，request_from_id 置为 0，流水线再次开始运行。



【15】修改流水线取指级模块 (**PC.v**)，使得取指模块支持在顺序执行的基础上，能够根据暂停信号 (**stall_pc**) 暂停取指；取指模块能够在发生分支跳转时，根据暂停信号 (**stall_pc**) 暂停取指，并在恢复后完成正确的分支跳转操作。在 CG 平台完成 3.3-1 的自动评测。

【16】修改流水线译码级模块 (**ID.v**)，使得译码模块能够根据 EX 级和 MEM 级的访存相关控制信号 (**load_related_***)，生成正确的暂停请求 (**stall_request**)。在 CG 平台完成 3.3-2 的自动评测。

【17】修改流水线执行级模块 (**EX.v**)，使得执行级模块能够根据访存控制信号，在发生内存读时生成正确的访存相关控制信号。在 CG 平台完成 3.3-3 的自动评测。

【18】修改流水线访存级模块 (**MEM.v**)，使得访存模块能够根据访存控制信号，在发生内存读时生成正确的访存相关控制信号。在 CG 平台完成 3.3-4 的自动评测。

【19】修改寄存器读操作的数据前递 (**forwarding**) 模块 (**RegReadProxy.v**)，使得寄存器前递模块能够结合当前寄存器读取请求，根据 ID 级和 EX 级发出的访存相关控制信号 (***_load_flag**)，生成交给 ID 级的两个访存相关控制信号 (**load_related_***)。在 CG 平台完成 3.3-5 的自动评测。

【20】修改流水线流水线控制器模块 (**PipelineController.v**)，使得流水线控制器能够在发生以下两种情况时，暂停流水线的对应部分：ID 级发出暂停请求时 (**request_from_id**)；需要暂停整条流水线时 (**stall_all**)。在 CG 平台完成 3.3-6 的自动评测。

这里我们来补充一下 D 触发器在暂停机制下设计的原理。因为我们使用 D 触发器连接两个阶段，因此我们在 D 触发器设计中加入了两个暂停信号，分别为 **stall_current_stage** (暂停当前阶段) 和 **stall_next_stage** (暂停下一阶段)，当暂停当前阶段且不暂停下一阶段时，我们阻断输入的数据进行输出，将输出置为 0；当不暂停当前指令时，我们就可以直接对当前数据进行输出；当两个阶段都暂停时，我们只需要让该模块保持之前的输出即可。这里给出

大家我们D触发器的实现代码：

```
module PipelineDeliver #(
    parameter WIDTH = 1
) (
    input                clk,
    input                rst,
    input                stall_current_stage,
    input                stall_next_stage,
    input                [WIDTH - 1:0] in,
    output reg [WIDTH - 1:0] out
);

always @(posedge clk) begin
    if (rst) begin
        out <= 0;
    end
    else if (stall_current_stage && !stall_next_stage) begin
        out <= 0;
    end
    else if (!stall_current_stage) begin
        out <= in;
    end
end
endmodule // PipelineDeliver
```

第四章 TinyMIPS 处理器的测试

4.1 TinyMIPS 模块结构

请参考指导书上册。

4.2 功能仿真测试

请参考指导书上册。

4.3 上板功能测试

请参考指导书上册。

附录

A：使用 GCC 工具链编译生成固件

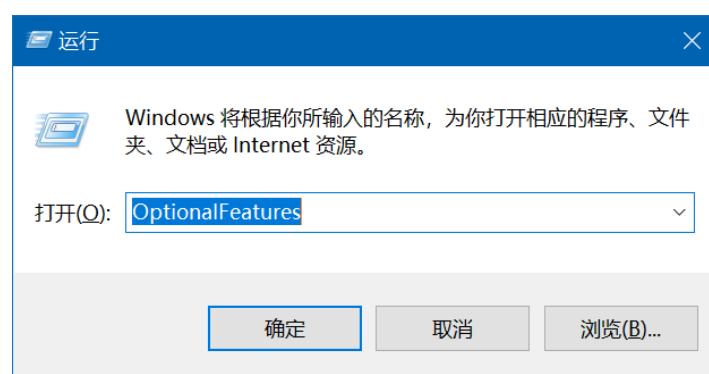
生成不同指令平台的可执行文件的过程称为**交叉编译**。在 Linux 上搭建交叉编译环境比较方便。这里以 Ubuntu 16.04 LTS 为例，讲解如何在 Linux 上生成 Verilog 支持的 ROM 初始化文件。

Ubuntu 操作系统可以在虚拟机中安装。下载 Ubuntu 16.04LTS 或者更高版本的安装镜像，在 VMware 或者 Virtual Box 软件中安装。在 VMware 下安装完毕后，在虚拟机中安装 VMware Tools 可以实现拖拽文件到宿主机的操作。

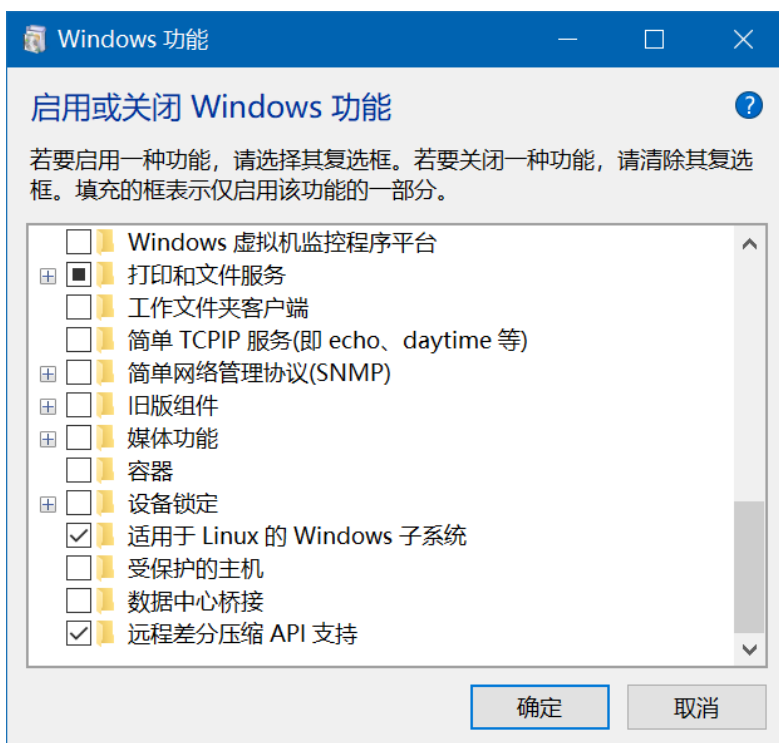
使用 Windows 10 Build 1607 及以上的同学还可以使用 **Windows Subsystem for Linux (WSL)**。这是由微软开发的一个工具，在 Windows 上模拟了大部分 Linux 的内核调用，使得我们能够在 Windows 环境下执行大部分的 Linux 命令。相比于虚拟机，这种方式更加简单。

A.1 WSL 的安装

首先需要启用 **WSL 功能**。在 Windows 下按下 **Win+R** 组合键，输入 “OptionalFeatures”，按下回车：



在弹出的窗口中找到“适用于 Linux 的 Windows 子系统”，勾选它，点击确定。



等待功能安装完毕。

然后打开 Windows 10 的开发者模式。打开 Windows 10 的**设置**，搜索“开发者选项”，然后选择“开发人员模式”即可。

开发者选项

使用开发人员功能

这些设置只用于开发。

[了解更多信息](#)

☐ Microsoft Store 应用

仅安装 Microsoft Store 的应用。

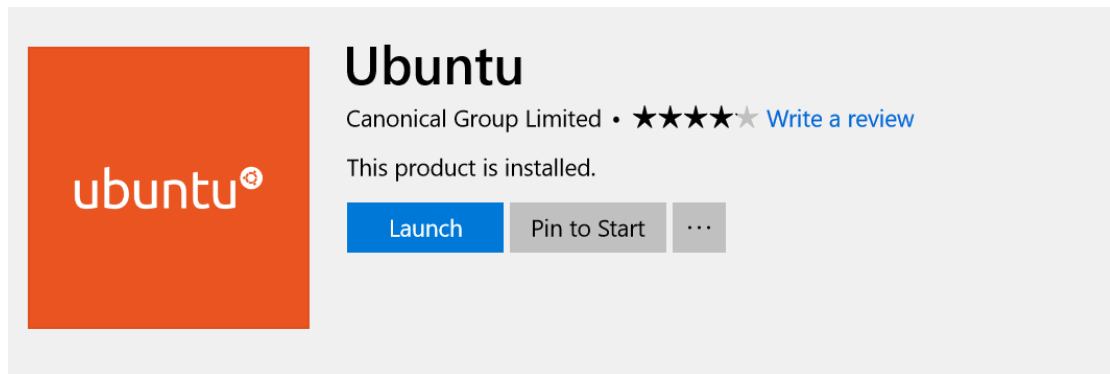
☐ 旁加载应用

从你信任的其他来源（例如工作区）安装应用。

☒ 开发人员模式

安装任何已签名的可信应用并使用高级开发功能。

第三步对于不同的版本的 Windows 10 有所区别。对于较新的版本，如 Windows 10 Build 1709 及以上版本，打开 Microsoft Store，搜索“Ubuntu”，找到 Ubuntu 点击安装：



对于较旧的 Windows 10，如 Build 1607 版本，你需要打开 powershell（**Win**+R 然后输入“powershell”），执行命令 bash:

```
Windows PowerShell
版权所有 (C) 2016 Microsoft Corporation。保留所有权利。

PS C:\Users\qrato> bash
-- Beta 版功能 --
这将在 Windows 上安装由 Canonical 分发的 Ubuntu，
根据其条款的授权参见此链接：
https://aka.ms/uowterms

键入“y”继续：
```

两种方法都能够安装 WSL 下的 Ubuntu。由于系统需要下载 Ubuntu 的镜像，整个过程可能需要较长时间。安装完毕后，命令行会提示你输入 Linux 的用户名和密码，请牢记。

A.2 VMware 下安装 Ubuntu

略。


```

ifndef CROSS_COMPILE
CROSS_COMPILE = mips-linux-gnu-
endif

CC = $(CROSS_COMPILE)as
LD = $(CROSS_COMPILE)ld

OBJCOPY = $(CROSS_COMPILE)objcopy
OBJDUMP = $(CROSS_COMPILE)objdump

OBJECTS = inst_rom.o
export CROSS_COMPILE

all: inst_rom.bin

%.o: %.S
$(CC) -mips32 $< -o $@

inst_rom.om: ram.ld $(OBJECTS)
$(LD) -T ram.ld $(OBJECTS) -o $@

inst_rom.bin: inst_rom.om
$(OBJCOPY) --remove-section .reginfo \
--remove-section .MIPS.abiflags -O verilog $< $@

clean:
rm -f *.o *.om *.data *.bin

```

我们还需要创建 ram.ld 文件，输入 “vi ram.ld” 然后输入下面内容：

```

MEMORY
{
    ram : ORIGIN = 0x00000000, LENGTH = 0x00010000
}

SECTIONS
{
    .text :
    {
        *(.text)
    } > ram
    .data :
    {
        *(.data)
    }
}

```

```

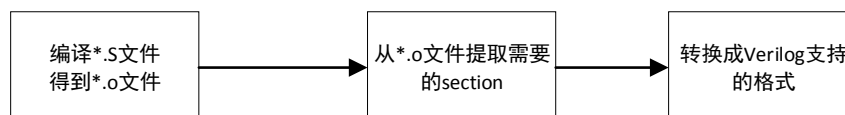
} > ram
.bss :
{
    *(.bss)
} > ram
}

ENTRY (_start)

```

这个文件的作用是指示链接器按照我们指定的格式生成二进制文件。

首先我们简单介绍一下 **make** 工具的原理。**make** 工具被用于自动化的构建过程，它把构建过程分为不同的**目标**，每个目标除了要执行的命令，还会依赖于其他目标。**make** 工具会分析这些目标的依赖关系，得到正确的编译步骤。对于我们生成固件的过程而言，编译步骤可以简单描述成下面的形式：



分析 **Makefile** 脚本中各个目标的依赖结构，可以知道它做了下面几件事：

1. 首先给编译器和链接器加上 “**mips-linux-gnu-**” 前缀表示交叉编译；
2. “**all**” 目标是输入 **make** 命令后的默认目标，它依赖于 **inst_rom.bin**，这个文件就是我们最终生成的固件；
3. 所有的 “**.o**” 目标文件都依赖于 “**.S**” 文件，因为我们工程的源代码都是汇编文件。
“**\$(CC)-mips32 \$< -o \$@**”表示使用编译器编译生成目标文件，指令集是 **MIPS32**；
4. **inst_rom.om** 目标依赖于目标文件和 **ram.ld**。**ram.ld** 文件我们已经给出，而目标文件在 “**.o**” 目标编译得到；
5. **inst_rom.bin** 目标的作用是移除 **inst_rom.om** 文件中不需要的段（**section**）并按照 **Verilog** 格式输出到文件 **inst_rom.bin**。到此我们就得到了一个可以用于初始化的固件。
6. **clean** 目标是清理命令，它会删除所有产出的文件。

下面我们编写一段 MIPS 汇编程序测试一下。在 Makefile 同一个目录下编辑文件 inst_rom.S，输入内容：

```
.org 0x0
.global _start
.set noat
_start:
    lui $1, 0x0101
    ori $1, $1, 0x0101
    ori $2, $1, 0x1100
    or $1, $1, $2
    andi $3, $1, 0x00fe
    and $1, $3, $1
    xori $4, $1, 0xff00
    xor $1, $4, $1
    nor $1, $4, $1
```

然后在此目录下执行 make 命令：

```
jasper@DESKTOP-62C17AB:~/rom_src$ make
mips-linux-gnu-as -mips32 inst_rom.S -o inst_rom.o
mips-linux-gnu-ld -T ram.ld inst_rom.o -o inst_rom.om
mips-linux-gnu-objcopy --remove-section .reginfo \
--remove-section .MIPS.abiflags -O verilog inst_rom.om inst_rom.bin
jasper@DESKTOP-62C17AB:~/rom_src$
```

如果 Makefile、rom.ld 或者 inst_rom.S 有任何的语法错误，编译过程就会失败。

打开 inst_rom.bin 文件，它的内容如下：

```
@00000000
3C 01 01 01 34 21 01 01 34 22 11 00 00 22 08 25
30 23 00 FE 00 61 08 24 38 24 FF 00 00 81 08 26
00 81 08 27 00 00 00 00 00 00 00 00 00 00 00 00
```

每 4 个数字是一条指令（32bit），后面的 0 用于对齐到 16 字节。非 0 的部分刚好是汇编程序中的 9 条指令。把这个文件复制到对应位置，然后在 Verilog 代码中输入这个文件的路径。

可以看到 GCC 工具链的 Verilog 格式输出都是以 8bit 为单位。为了能够利用 GCC 工具链自动编译固件，我们就不得不把程序中的 ROM 设计为以 **8bit 为最小单位**的形式而不是 32 位或者其他。

每当需要新的程序进行测试时，遵循下面的步骤编译生成新的固件：

1. 编写 `inst_rom.S`;
2. 执行 `make` 命令;
3. 如果 `make` 命令没有出错，那么复制生成的 `inst_rom.bin` 文件到对应位置；反之，检查 `inst_rom.S` 的语法错误

由于汇编程序比 C 语言等更加贴近硬件，导致很多错误不能在编译过程中发现。例如，写错了指令的参数顺序；计算错了 `beq` 指令的跳转步长；遗漏了 MIPS 中延迟槽的特性等等。这些情况下编译都不会给出错误或者警告。因而，当你的程序运行的结果和预期不一致时，除了检查 Verilog 部分的代码外，还需要注意是否是汇编程序本身存在错误。

A.5 FAQ

问：VMware 下怎么把编译好的文件复制到 Windows 下？

答：在虚拟机中安装 VMware Tools。重启虚拟机后，就可以直接拖拽文件到 Windows 下。

问：WSL 下怎么把编译好的文件复制到 Windows 下？

答：在 WSL 中，C 盘、D 盘等被挂载到 `/mnt` 目录下。这个目录下的 `c`、`d` 文件夹就是 Windows 中的磁盘。使用 `cp` 命令就可以完成文件复制。你还可以把工程目录放在这些文件夹下，这样编译的输出直接可以在 Windows 中访问。

问：执行 `make` 命令出错了怎么办？

答：仔细检查 `Makefile`、`ram.ld` 和汇编源程序。如果自己不能解决的请向助教求助。

问：使用非 Ubuntu 的 Linux 发行版可以吗？

答：理论上可以，但是具体的交叉编译器的命令名字可能有所区别，产出的中间文件可能也不一样（例如多出或缺失某些 `section`）。你需要修改 `Makefile` 并检查输出是否和我们需要的固件一致。某些发行版没有预编译的 MIPS 交叉编译环境，你可能需要下载对应的源码进行编译。

问：section 是什么东西？

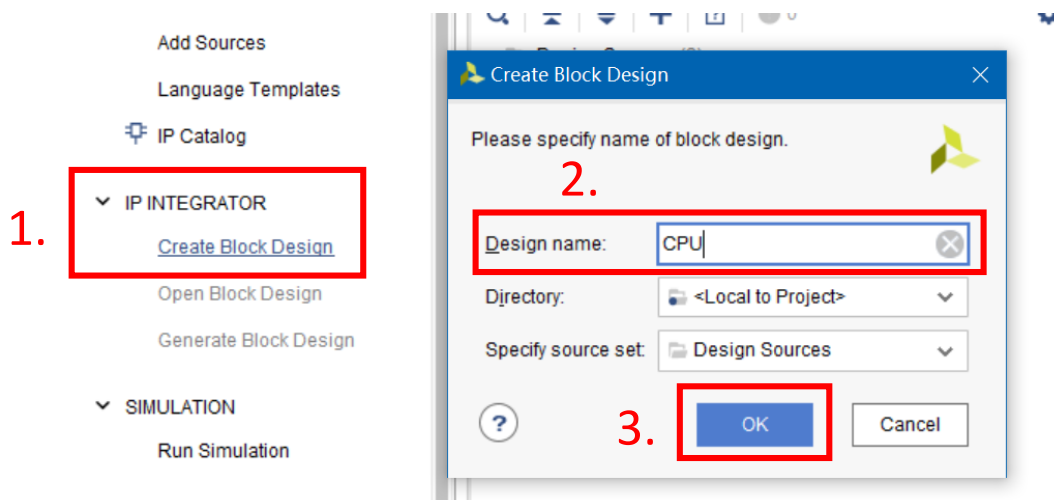
答：GCC 工具链生成的*.o 文件的格式是 ELF。ELF 是 Linux 下的可执行文件格式，和 Windows 下的*.exe 类似(*.exe 称为 PE 格式)。ELF 格式规定了它的内容由多个段(section) 构成。我们的汇编程序不需要其他的 section，所以编译过程中需要去掉多余的 section。

B：使用 Vivado 的 Block Design 功能

有时候某个模块的内容只是把其他模块连接起来封装在一起。这种模块可以使用 Vivado 的 Block Design 功能完成。

B.1 新建 Block Design

首先点击 Vivado 左侧的 Flow Navigator，选择“IP INTEGRATOR”下的“Create Block Design”，在弹出的窗口里输入模块的名字：



点击“OK”后，就会出现一个新的窗口“Diagram”。在这里可以对模块内容进行编辑。

B.2 绘制

首先，右击 Diagram 的空白处，查看这个菜单的功能：

1. Add IP: 添加 IP 核。这里的 IP 核一般是 Vivado 自带的。我们的设计中没有用到。
2. Add module: 添加模块。用 Verilog 编写好其他模块后，就可以用这个功能把它加到

我们的 Block Design 中。

3. **Create Port:** 创建接口，即 Verilog 中的接口定义部分。使用这个功能时一定要注意接口的 input/output 属性和宽度。

添加模块和创建接口之后，就可以点击拖拽对应的引脚实现连线功能。

注意事项:

1. Diagram 中用到的模块如果被修改，一定要点击窗口上方出现的“Refresh Changed Modules”。更新之后一般会出现警告，可以不予理会。
2. 在直接用 Verilog 编写模块时，执行仿真不需要编译。但是使用 Diagram 的情况下，每次仿真前你都需要 Run Synthesis。