

# FPGA Classic Mips pipeline CPU

Vivado version 2017.4 Mars version 4.5 Code version 1.2

## 目录

- [项目介绍](#)
- [安装](#)
- [使用说明](#)
- [Insight](#)
- [示例](#)
- [相关仓库](#)
- [维护者](#)
- [如何贡献](#)
- [使用许可](#)

## 项目介绍

本项目是一个 **Classic Mips pipeline CPU**, 由 Verilog 语言编写, Mips 汇编实现场景测试。

## 项目架构

### CPU特性

- ISA: 支持的指令为Minisys-1 ISA, 支持指令集的所有指令; 内置32个可供汇编代码使用的寄存器, 位宽均为32bits; 保护0号寄存器禁止写入, 保护pc、hi、lo寄存器不可访问。
- 寻址空间设计: 基于[哈佛架构](#)设计; 寻址单位为Byte, 数据空间的大小为2048KB。
- 外设与IO支持: 硬件将24位拨码开关与25号寄存器绑定, 将七段数码管与24号寄存器绑定, 低8bit数据输入位显示在VGA显示器, 蜂鸣器播放电子音乐。
- CPI: 多周期5级流水线CPU, 解决冲突方式见后文。

### CPU接口

输入/输出	端口类型	位宽	端口名称	用途
input	wire	1	fpga_RST	CPU复位信号
input	wire	1	fpga_CLK	CPU时钟信号
input	wire	24	switch2N4	24位拨码开关
output	wire	24	led2N4	24位LED灯
input	wire	1	rx	uart写入
output	wire	1	tx	uart传出
output	wire	8	segment_led	7段数码管信号
output	wire	8	segment_en	7段数码管信号
output	wire	1	buzzer	蜂鸣器信号
output	wire	4	red	VGA信号
output	wire	4	green	VGA信号
output	wire	4	blue	VGA信号
output	wire	1	hsync	VGA信号
output	wire	1	vsync	VGA信号

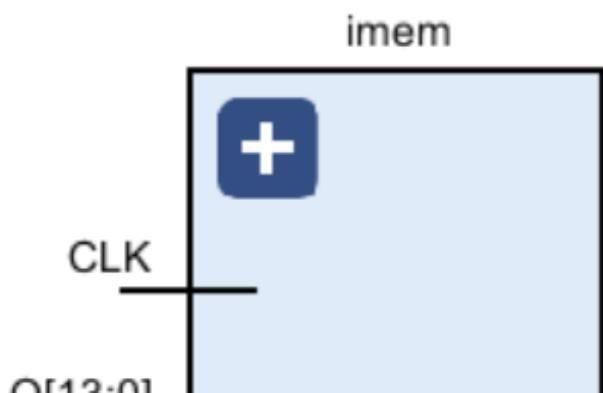
## 项目结构

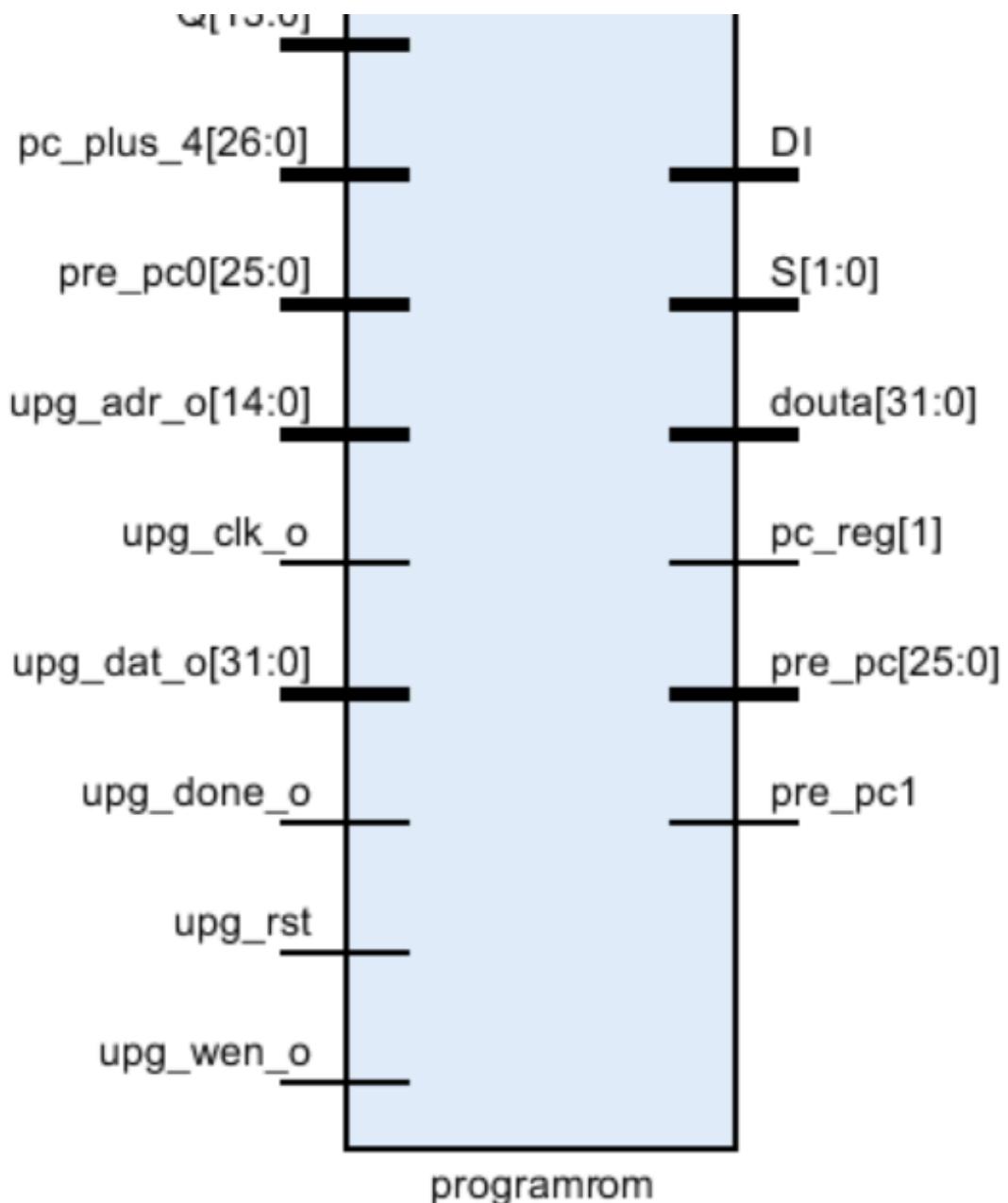
- 详见 [Schematic文件](#)

## 模块细节

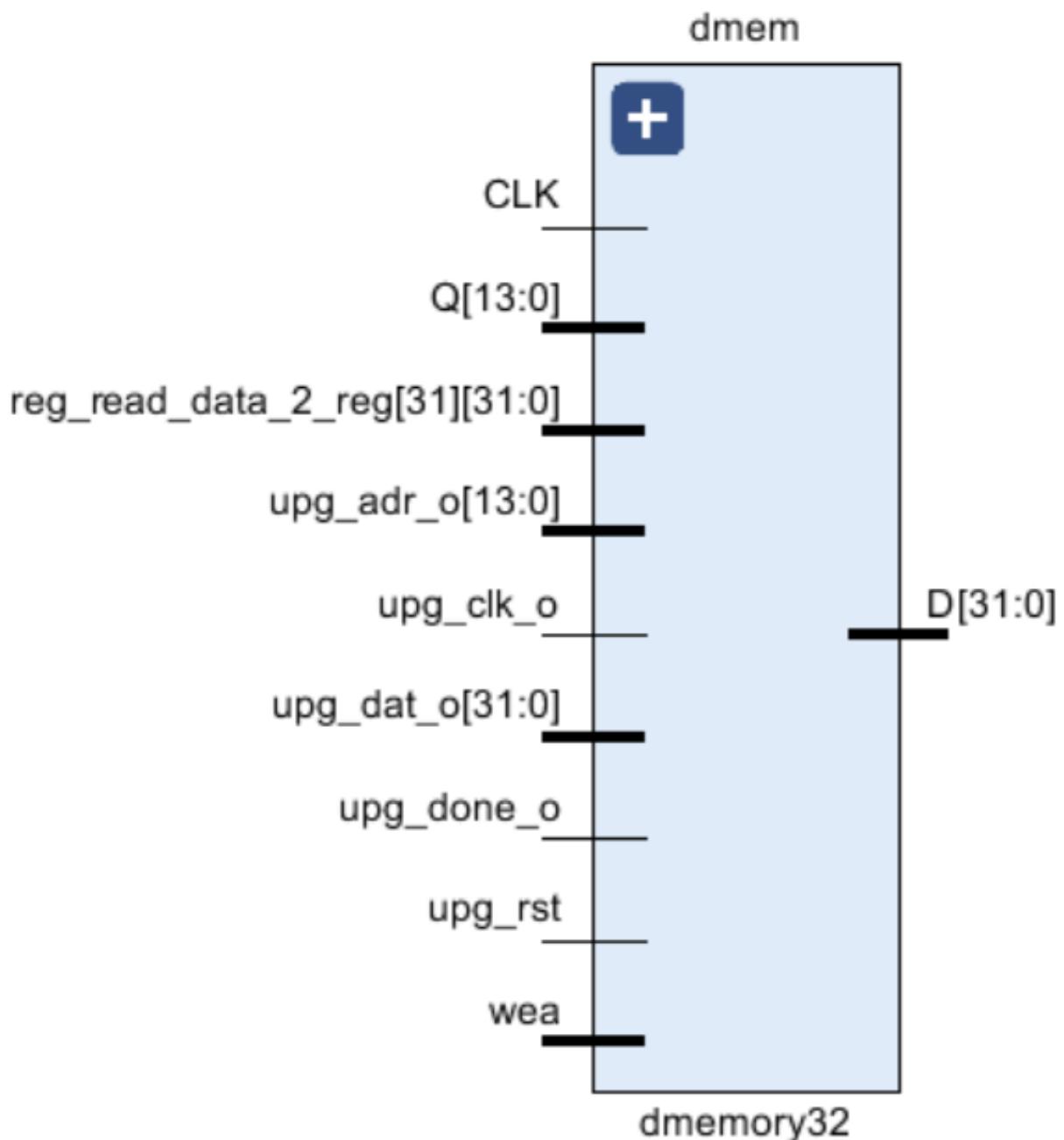
### 基本CPU模块

#### Instruction Memory





Data Memory



```

module dmemory32(
    input ram_clk_i, // from CPU top
    input ram_wen_i, // from controller
    input [13:0] ram_adr_i, // from alu_result of ALU
    input [31:0] ram_dat_i, // from read_data_2 of decoder
    output [31:0] ram_dat_o, // the data read from ram
    // UART Programmer Pinouts
    input upg_RST_i, // UPG reset (Active High)
    input upg_clk_i, // UPG ram_clk_i (10MHz)
    input upg_wen_i, // UPG write enable
);

```

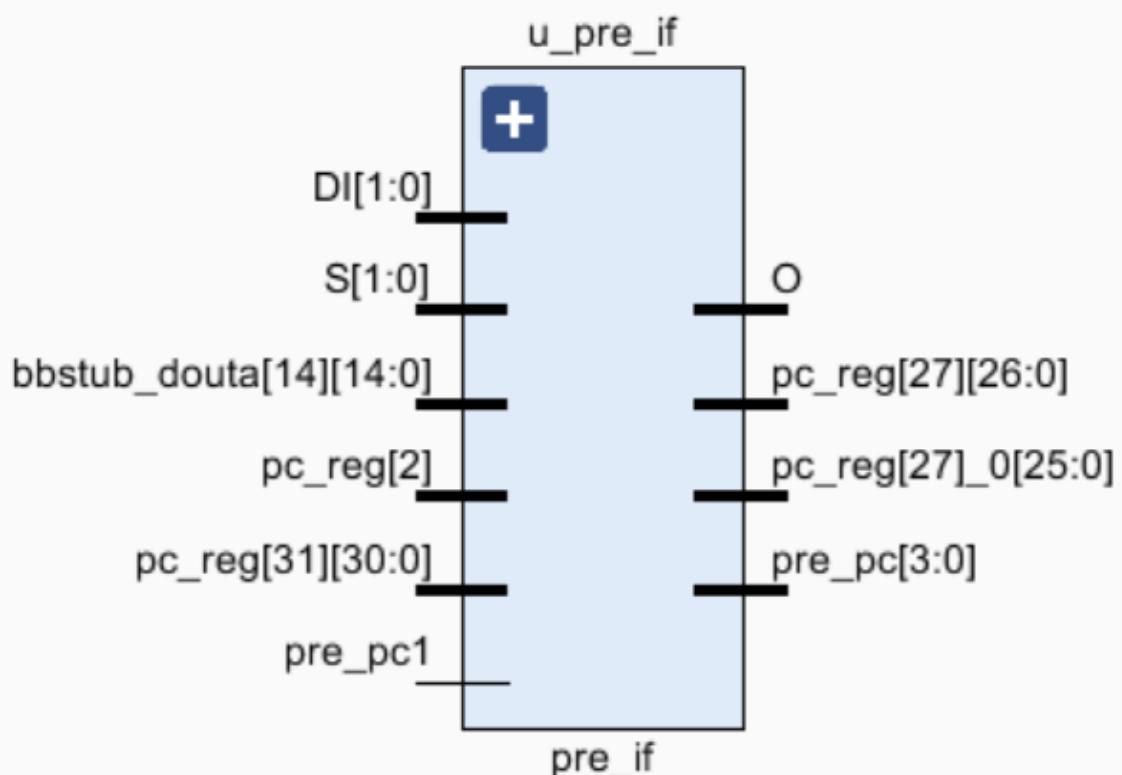
```

    input [13:0] upg_adr_i, // UPG write address
    input [31:0] upg_dat_i, // UPG write data
    input upg_done_i // 1 if programming is finished
);

```

## 流水线架构与设计

- pre\_if模块，用于预测除jal指令以外下一条指令地址。遇到beq和bne时按照跳转预测。



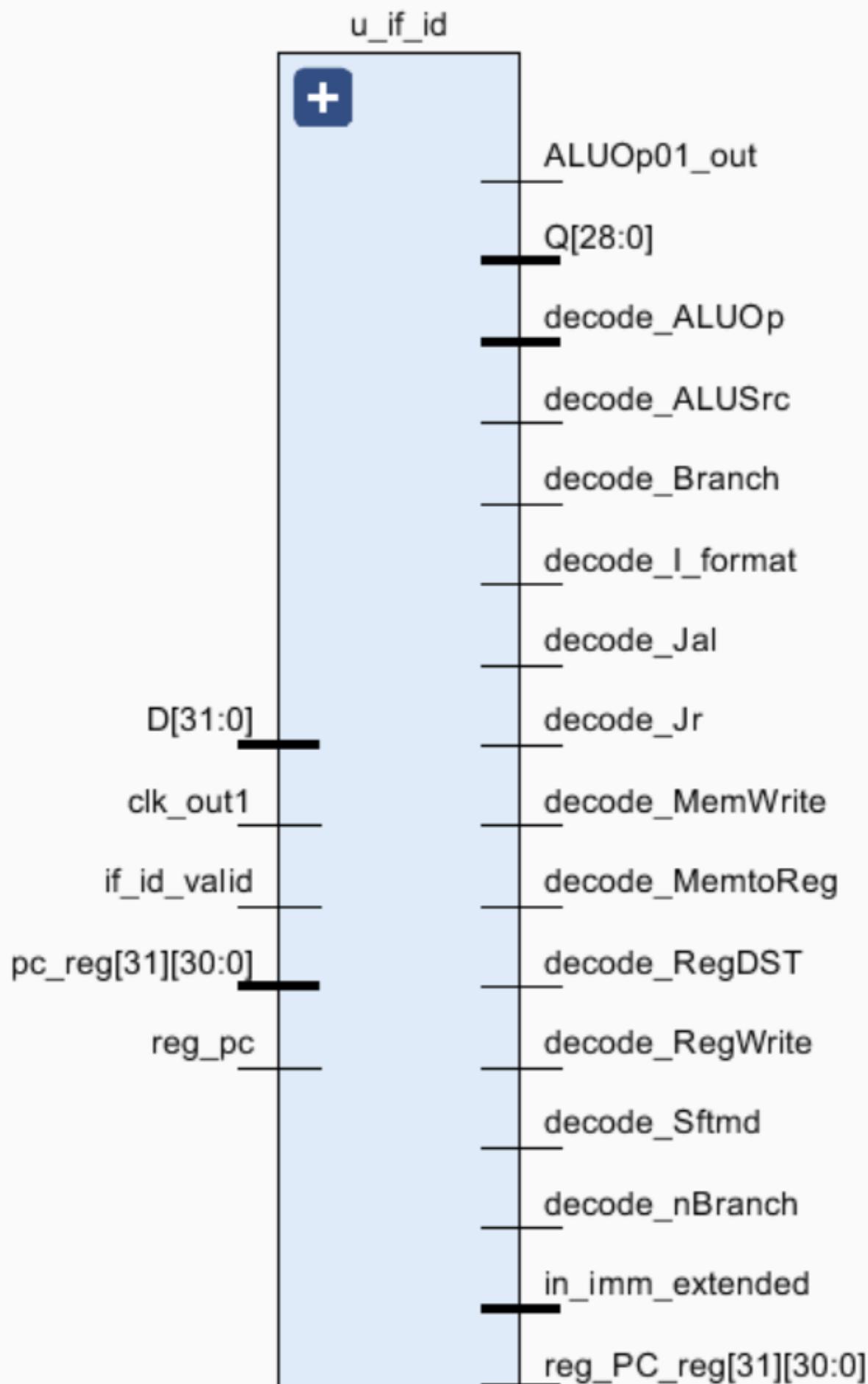
```

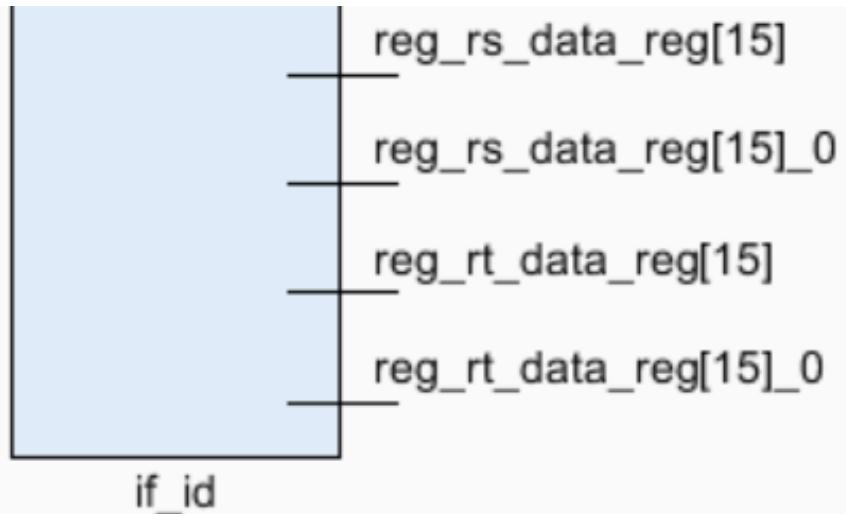
// 做下一条指令的预测
// this is pre_if for a minisys cpu
module pre_if (
    input [31:0] instr,
    input [31:0] pc,

    output [31:0] pre_pc
);

```

- if\_id模块，将instructiton从if传递到id



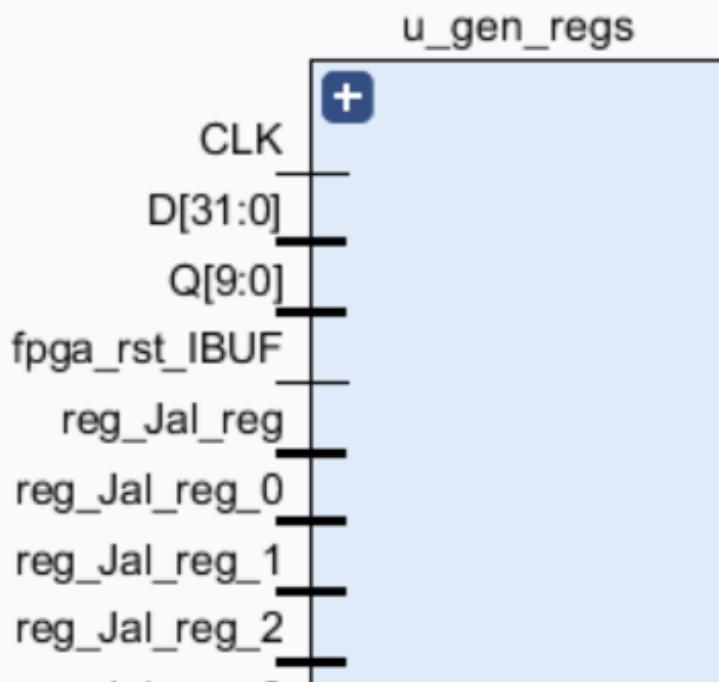


```

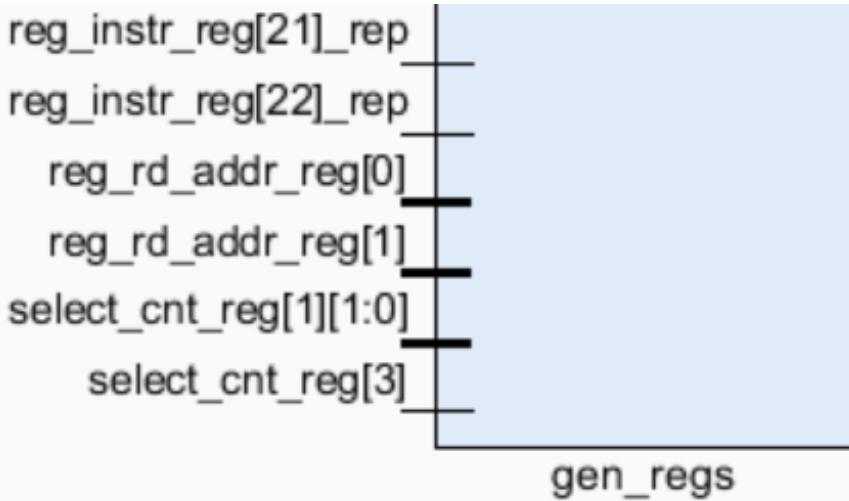
// IF_ID
// 通路模块 if_id 主要负责指令和 PC 的传递，以及流水线冲刷标志位的传递。
module if_id(
    input      clk,
    input      reset,
    input [31:0] in_instr,
    input [31:0] in_pc,
    input      flush,
    input      valid,
    output [31:0] out_instr,
    output [31:0] out_pc,
    output      out_noflush
);

```

- gen\_regs模块，寄存器模块，读写寄存器，将指定寄存器向外传到io中



reg_Jal_reg_3	
reg_Jal_reg_4	
reg_Jal_reg_5	
reg_Jal_reg_6	
reg_Jal_reg_7	
reg_Jal_reg_8	
reg_Jal_reg_9	
reg_Jal_reg_10	
reg_Jal_reg_11	
reg_Jal_reg_12	
reg_Jal_reg_13	
reg_Jal_reg_14	E
reg_Jal_reg_15	ram_reg_o[23:0]
reg_Jal_reg_16	regs_read_data_1[31:0]
reg_Jal_reg_17	regs_read_data_2[31:0]
reg_Jal_reg_18	segment_led_reg[6][6:0]
reg_Jal_reg_19	
reg_Jal_reg_20	
reg_Jal_reg_21	
reg_Jal_reg_22	
reg_Jal_reg_23	
reg_Jal_reg_24	
reg_Jal_reg_25[31:0]	
reg_Jal_reg_26	
reg_Jal_reg_27	
reg_Jal_reg_28	
reg_Jal_reg_29	
reg_instr_reg[16]_rep	
reg_instr_reg[17]_rep	



```

// gen_regs - 寄存器文件
// clk - 时钟信号
// reset - 异步复位信号
// wen - 写使能信号
// regRAddr1, regRAddr2 - 两个读端口的寄存器地址
// regWAddr - 写端口的寄存器地址
// regWData - 写入寄存器的数据
// regRData1, regRData2 - 两个读端口的寄存器暑假

module gen_regs (
    input  clock,
    input  reset,
    input [4:0] rs,
    input [4:0] rt,
    input [4:0] rd,
    input write,
    input [4:0] write_reg,
    input [31:0] write_data,
    input outer_input,
    input [31:0] outer_t9,
    output reg[31:0] ram_reg_o,
    output reg[31:0] ram_reg_o2,
    output [31:0] read_data_1,
    output [31:0] read_data_2
);

```

- decode模块, decoder模块, 从指令中读取出各种控制信号以及立即数等

```

module decode(
    input [31:0] instr,
    output reg Jr,// 1 indicates the instruction is "jr", otherwise it's not "jr"
    output reg Jmp,// 1 indicate the instruction is "j", otherwise it's not

```

```

    output reg Jal, // 1 indicate the instruction is "jal", otherwise it's not
    output reg Branch, // 1 indicate the instruction is "beq" , otherwise it's not
    output reg nBranch, // 1 indicate the instruction is "bne", otherwise it's not
    output reg RegDST,// 1 indicate destination register is "rd",otherwise it's "rt"
    output reg MemtoReg, // 1 indicate read data from memory and write it into register
    output reg RegWrite, // 1 indicate write register, otherwise it's not
    output reg MemWrite, // 1 indicate write data memory, otherwise it's not
    output reg ALUSrc, // 1 indicate the 2nd data is immidiate (except "beq","bne")
    output reg I_format, // 1 indicate the instruction is I-type but isn't
    "beq","bne","LW" or "SW"
    output reg Sftmd, // 1 indicate the instruction is shift instruction
    // if the instruction is R-type or I_format, ALUOp is 2'b10; if the instruction
    is"beq" or "bne", ALUOp is 2'b01??
    // if the instruction is"lw" or "sw", ALUOp is 2'b00??
    output reg [1:0] ALUOp,
    output[4:0] rs_addr,
    output[4:0] rt_addr,
    output[4:0] rd_addr,
    output [4:0] Shamt,
    output [31:0] imm_extended,
    output[5:0] Opcode,
    output [5:0] Function_opcode
);

```

- id\_ex模块，控制信号和数据从decoder传到execute的通路

详见 [Schematic文件](#)

```

// this module is the ID/EX pipeline register
// it takes the output from the ID stage and passes it to the EX stage

module id_ex(
    input clk,
    input reset,
    // these are pipeline signals
    input flush,
    input valid,
    // these are inputs from decode stage
    input in_Jr, // jump or not
    input in_Jmp,
    input in_Jal,
    input in_Branch,
    input in_nBranch,
    input in_RegDST,
    input in_MemtoReg, // lw
    input in_RegWrite,
    input in_MemWrite, // sw
    input in_ALUSrc,

```

```

    input in_I_format,
    input in_Sftmd,
    input[1:0] in_ALUOp,
    input[4:0] in_rs_addr,
    input[4:0] in_rt_addr,
    input[4:0] in_rd_addr,
    input[31:0] in_rs_data,
    input[31:0] in_rt_data,
    input[31:0] in_imm_extended,
    input[5:0] in_Opcode,
    input[5:0] in_Function_Opcode,
    input [4:0] in_Shamt,
    input[31:0] in_pc,
    // these are outputs to the execute stage
    output out_Jr,
    output out_Jmp,
    output out_Jal,
    output out_Branch,
    output out_nBranch,
    output out_RegDST,
    output out_MemtoReg,
    output out_RegWrite,
    output out_MemWrite,
    output out_ALUSrc,
    output out_I_format,
    output out_Sftmd,
    output[1:0] out_ALUOp,
    output[4:0] out_rs_addr,
    output[4:0] out_rt_addr,
    output[4:0] out_rd_addr,
    output[31:0] out_rs_data,
    output[31:0] out_rt_data,
    output[31:0] out_imm_extended,
    output[5:0] out_Opcode,
    output[5:0] out_Function_Opcode,
    output [4:0] out_Shamt,
    output[31:0] out_pc
);

```

- execute模块，得到控制信号和输入后计算出alu控制信号，并集成alu进行运算

```

// 设置时间精度为1纳秒/1皮秒
timescale 1ns / 1ps

// 定义模块execute，包含多个输入和输出端口
module execute (
    input [31:0] Read_data_1, // rs输入数据

```

```

    input [31:0] Read_data_2, // rt输入数据
    input [31:0] Imme_extend, // immediate扩展后的
    input [5:0] Function_opcode, // 指令[5:0]
    input [5:0] opcode, // 指令[31:26]
    input [4:0] Shamt, // 指令[10:6], 移位量
    input [31:0] PC, // PC, current PC
    input [1:0] ALUOp, // {(R_format || I_format), (Branch || nBranch)}
    input ALUSrc, // 1表示第二个操作数是立即数 (除了beq、bne)
    input I_format, // 1表示I型指令, 除了beq、bne、LW、SW
    input Sftmd, // 1表示这是一个移位指令?
    input Jr, // 1表示这是一个jr指令

    output Zero, // 1表示ALU计算结果为0, 否则为1
    output [31:0] ALU_Result, // ALU计算结果
    output [31:0] Addr_Result // 计算得到的指令地址, 用于branch指令
);

```

- ex\_mem模块, 控制信号和数据从execute传递到mem部分的通路

详见 [Schematic文件](#)

```

// this module is used to pass data from the EX stage to the MEM stage
// the data is related to MEM and WB
module ex_mem(
    input clk,
    input reset,
    input flush, // flush the pipeline
    // input
    input in_MemtoReg, // 是否把内存数据写入寄存器
    input in_MemWrite, // 1: 写入内存
    input in_RegWrite, // 1: 写入寄存器
    input in_RegDST, // 1: 写入哪个寄存器
    input in_Jal, // jump and link
    input in_Zero, // ALU计算结果是否为0
    input [4:0] in_rs_addr,
    input [4:0] in_rt_addr,
    input [4:0] in_rd_addr,
    input [31:0] in_ALUResult, // ALU计算结果
    input [31:0] in_Addr_Result, // 计算出的内存地址
    input [31:0] in_imm_extended, // 符号扩展后的立即数
    input [31:0] in_read_data_1,
    input [31:0] in_read_data_2,
    input [31:0] in_pc, // 传进来的pc值
    // output
    output out_MemtoReg,
    output out_MemWrite,
    output out_RegWrite,
    output out_RegDST,
);

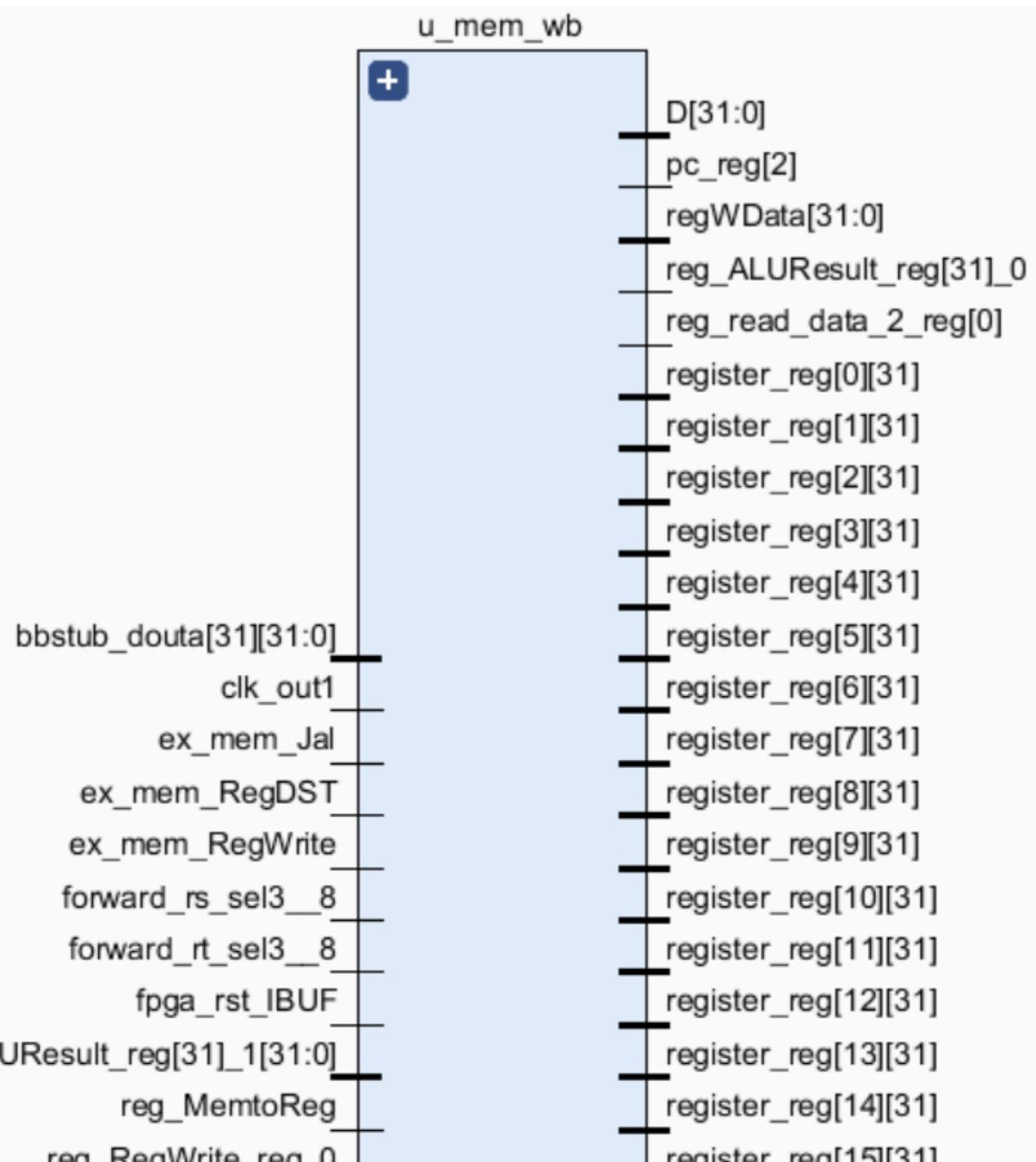
```

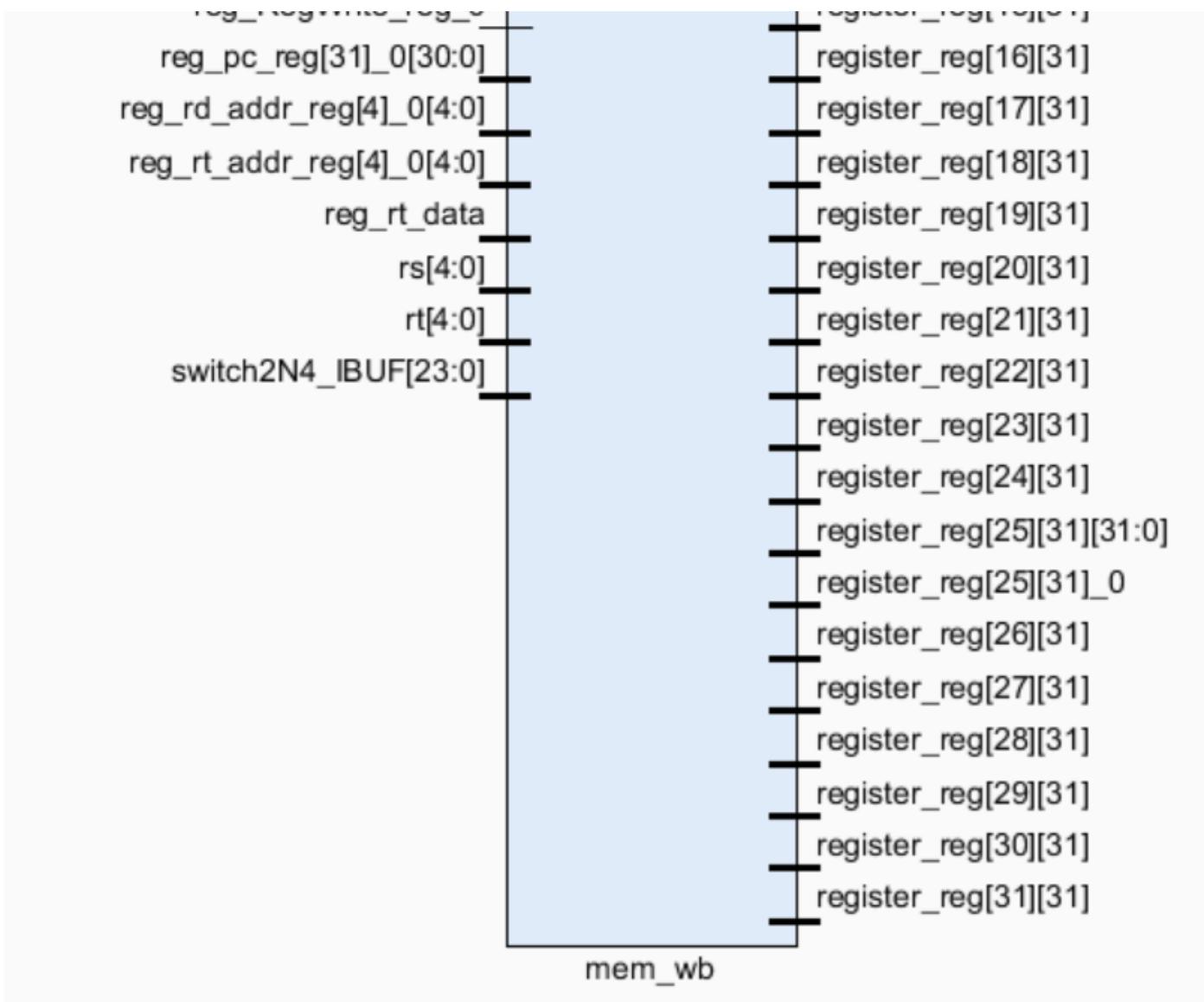
```

    output out_Jal,
    output out_Zero,
    output [4:0] out_rs_addr,
    output [4:0] out_rt_addr,
    output [4:0] out_rd_addr,
    output [31:0] out_ALUResult,
    output [31:0] out_Addr_Result,
    output [31:0] out_imm_extended,
    output [31:0] out_read_data_1,
    output [31:0] out_read_data_2,
    output [31:0] out_pc
);

```

- mem\_wb模块，控制信号和数据从mem传递到wb的通路





```
module mem_wb(
    input clk,
    input reset,
    // input
    input in_RegWrite,
    input in_Jal,
    input in_RegDST,
    input in_MemtoReg,
    input [31:0] in_ReadData, // data from memory
    input [31:0] in_ALUResult,
    input [4:0] in_rt_addr,
    input [4:0] in_rd_addr,
    input [31:0] in_pc,
    input in_zero,
    input [31:0] in_Addr_Result,
    // output
    output out_RegWrite,
    output out_Jal,
    output out_RegDST,
```

```

    output out_MemtoReg,
    output [31:0] out_ReadData,
    output [31:0] out_ALUResult,
    output [4:0] out_rt_addr,
    output [4:0] out_rd_addr,
    output [31:0] out_pc,
    output out_Zero,
    output [31:0] out_Addr_Result
);


```

- forwarding模块，根据控制信号和数据判断前递什么数据并且前递数据

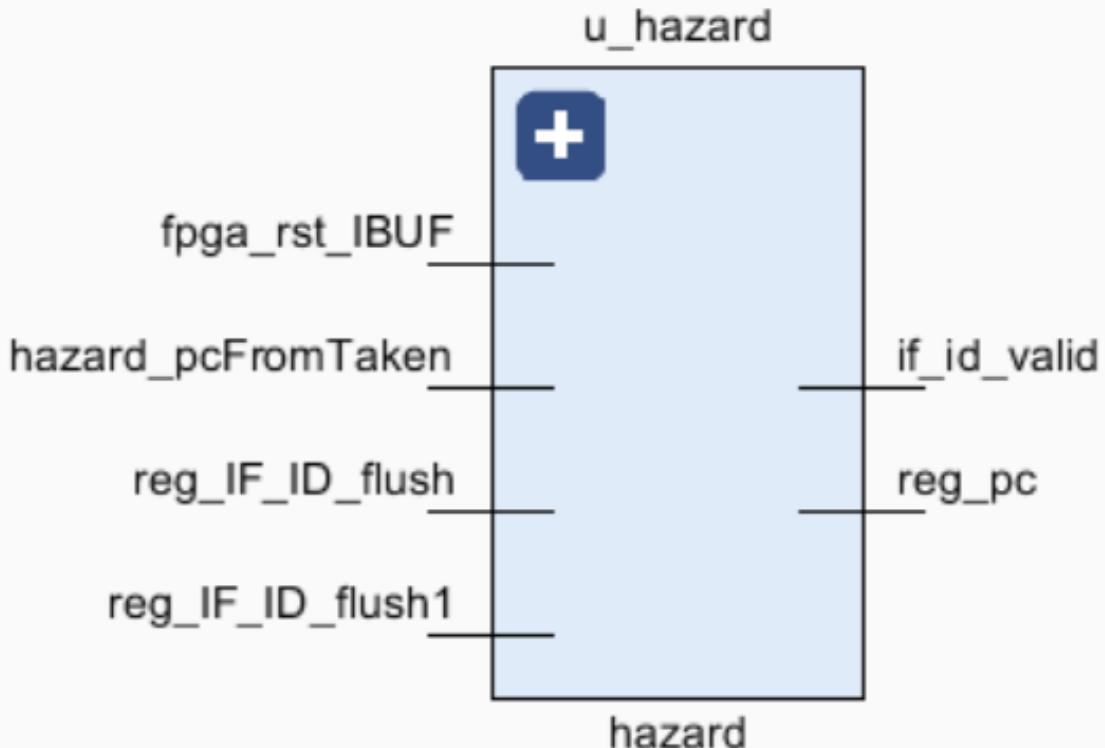
```

module forwarding (
    input [4:0] rs, // the address of rs register
    input [4:0] rt, // the address of rt register
    input [4:0] exMemRd, // 来自访存模块的对通用寄存器的访问地址
    input      exMemRw, // 来自访存模块的对通用寄存器的写使能信号
    input [4:0] memWBrd, // 来自写回模块的对通用寄存器的访问地址
    input      memWBrw, // 来自写回模块的对通用寄存器的写使能信号
    input      mem_wb_ctrl_data_toReg,
    input [31:0] mem_wb_readData,
    input [31:0] mem_wb_data_result,
    // 下面两个分别是不发生数据冒险时使用的信号
    input [31:0] id_ex_data_RegRData1, // data from rs register after decoder
    input [31:0] id_ex_data_RegRData2, // data from rt register after decoder
    input [31:0] ex_mem_data_result, // 访存阶段需要写到寄存器的数据

    output [31:0] forward_rs_data,
    output [31:0] forward_rt_data
);

```

- hazard模块，根据控制信号和数据判断冒险类型，并且根据冒险类型判断是否进行stall，重新计算pc值或者对流水行进行冲刷

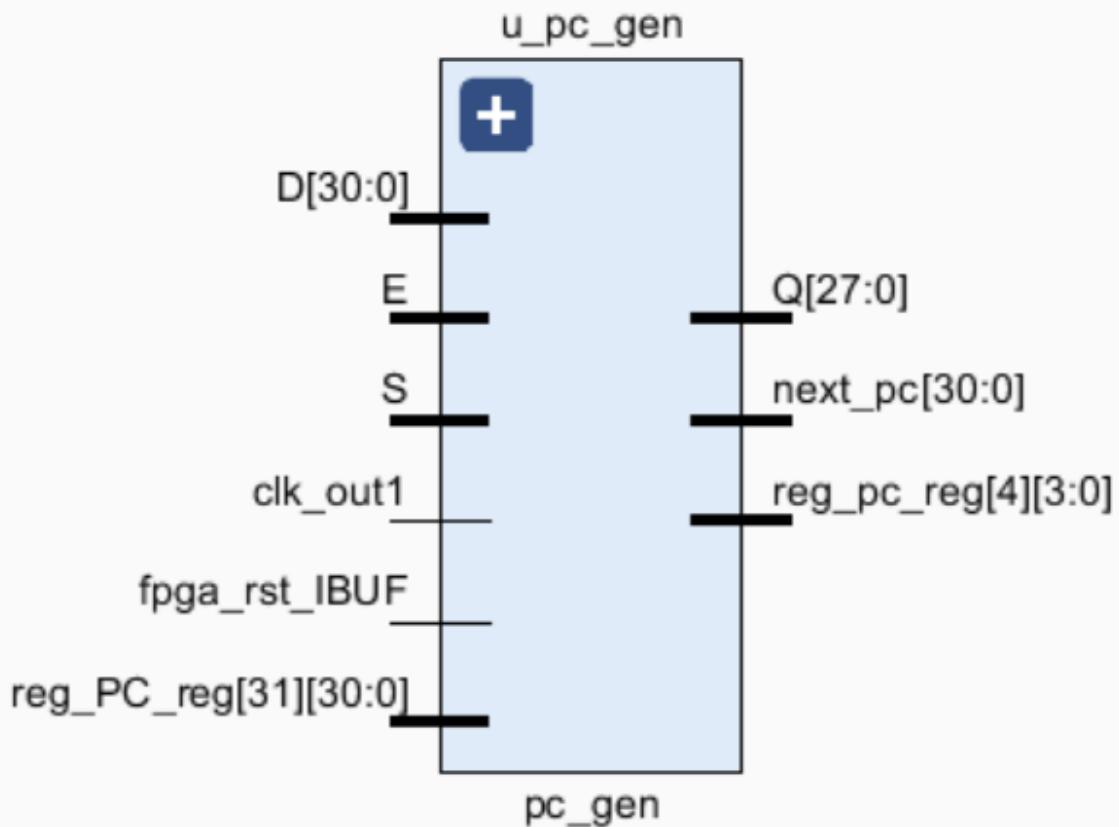


```

module hazard (
    input  [4:0] rs,           // 第一个寄存器编号
    input  [4:0] rt,           // 第二个寄存器编号
    input      id_ex_jr,       // jump and link 指令
    input      id_ex_Branch,   // 是否是分支指令 beq
    input      id_ex_nBranch,  // 是否是分支指令 bne
    input      ex_Zero,        // 两数比较, 结果是否为0
    input      id_ex_memRead,  // 是否从内存读取数据, decode里的MemToReg
    input      id_ex_memWrite, // 是否向内存写入数据
    input  [4:0] id_ex_rd,     // 要写入的寄存器编号
    input      ex_mem_memWrite, // 是否向内存写入数据
    output     pcFromTaken,   // 分支指令执行结果, 判断是否与预测方向相同
    output     pcStall,        // 程序计数器停止信号
    output     IF_ID_stall,    // 流水线IF_ID段停止信号
    output     ID_EX_stall,    // 流水线ID_EX段停止信号
    output     ID_EX_flush,    // 流水线ID_EX段清零信号
    output     EX_MEM_flush,   // 流水线EX_MEM段清零信号
    output     IF_ID_flush     // 流水线IF_ID段清零信号
);

```

- pc\_gen模块，根据hazard, pre\_if模块的结果及其他控制信号和数据判断下一条指令的pc，在这个模块中预测错误的pc会被推翻



```
module pc_gen(
    reset,           // 复位信号
    clk,             // 时钟信号
    Read_data_1,     // jr指令跳转的地址, PC=reg[rs]
    hazard_pcStall, // 是否停顿
    hazard_pcFromTaken, // 是否跳转
    id_ex_Jr,       // 是否是Jr 指令
    pre_pc,          // 前一条指令的 PC 地址
    pc_i,            // id_ex_pc
    pc_o             // 下一条指令的 PC 地址
);
```

- cpu模块，集合上述模块，传递信号和数据，比如传递到instruction memory和data memory的信号和数据，比如wb部分中传到寄存器的信号和数据

```
module cpu(
    input clk,
    input reset,
    output ram_wen_w, // when to write data to data memory
    output [31:0] ram_addr_i_w, // address of data memory
    output [31:0] ram_dat_i_w, // output data to data memory input
```

```
    input [31:0] ram_dat_o_w, // input data from data memory output
    input[31:0] imem_instr,
    output [31:0] imem_addr,
    output [31:0] ram_reg_o,
    output [31:0] ram_reg_o2,
    input  outer_input,
    input [31:0] outer_t9
);

```

## 其他附加模块

### VGA协议显示器显示模块

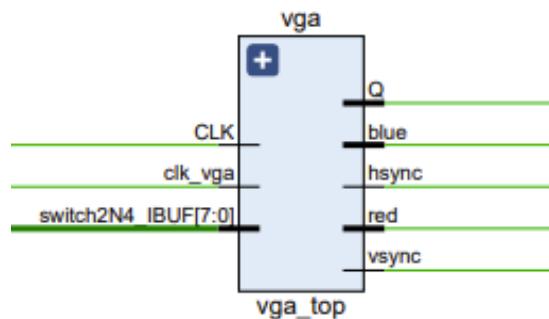
#### 模块说明

本项目中实现的VGA显示功能是：通过VGA接口，显示器可以显示当前拨码开关低8位输入的当前状态（0或1的数字显示），并且可以实时更新。

利用VGA进行显示，首先需要了解VGA的显示原理，在使用VGA时，开发板通过hsync, vsync, red, green, blue四个输入实现与显示器的信息传递。在编写实现VGA显示的代码时，要注意由于VGA显示信号传输的原理，需要确定好对应的时钟频率，同时要注意需要使用时序逻辑。



## 模块结构



## 引脚说明

输入/输出	端口类型	位宽	端口名称
input	wire	1	clk
input	wire	8	switch2N4
output	reg	1	hsync
output	reg	1	vsync
output	reg	4	red
output	reg	4	green
output	reg	4	Blue

## 蜂鸣器音乐模块

### 设计思路

要输出某个音符，我们需要知道它的频率，您可以在 [MixButton](#) 获取到更多信息。假设我们要输出中音C，查表发现它的频率是261.63Hz。我们从 261.63Hz 计算半周期，即 $191110 \times 10^{-8}$ 秒，然后我们只需在 $191110 \times 10^{-8}$ 秒后反转输出蜂鸣器。

我们将半周期编码为 period\_code 以避免数据冗余。

基本思想是将音符表示为周期代码。使用多路复用器选择由“Index”索引的当前周期代码，它表示当前正在播放哪个音符。索引每 0.15 秒递增 1，这意味着每个音符只会持续 0.15 秒。当选择当前周期代码时，它被翻译成实际的半周期数并让输出蜂鸣器在半周期后反转。

### “音符—bit串”转换脚本

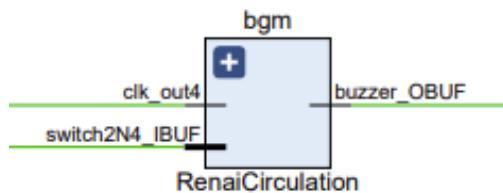
```
#include <iostream>
using namespace std;
int main(){
    string s = "131ef0fe101ef0fe1030ef0fe1012101f30" ;
    s = s + "ef0fe101ef0fe301ef0fe1012101310";
    s = s + "1012301e1012301e10e10e010e10e04310f03";
```

```

.....
string out;
for(int i = 0; i < s.length(); i ++){
    string tmp;
    switch(s[i]){
        case ('0') : tmp = "0000"; break;
        .....
        default : tmp = "0000";
    }
    out = tmp + out;
}
cout << s.size() * 4 << "\b" << out << endl << s.size();
}

```

## 模块结构



## 引脚说明

输入/输出	端口类型	位宽	端口名称
input	wire	1	bgm_clk
input	wire	1	enable
output	wire	1	buzzer

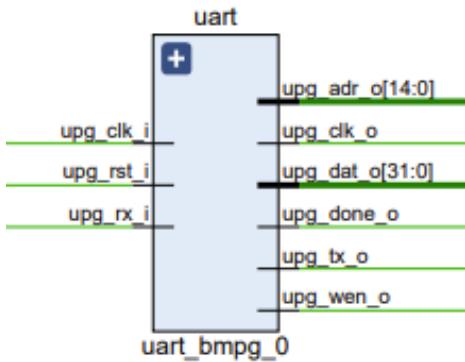
## UART串口烧录模块

### 模块说明

UART (Universal Asynchronous Receiver Transmitter) 是一种常用的串行通信协议，通常用于将数据从一个设备传输到另一个设备。在FPGA (Field Programmable Gate Array) 和CPU (Central Processing Unit) 中，UART通常被实现为一个模块，该模块可以接收和发送串行数据。

UART模块通常由两个主要组件组成：接收器和发射器。接收器负责从串行数据流中接收数据，并将其转换成并行数据。发射器则负责将并行数据转换成串行数据流，并将其发送到目标设备。

### 模块结构



## 7段数码管

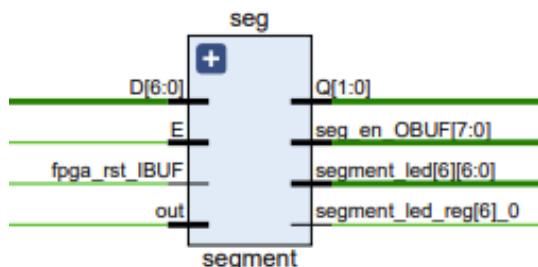
### 模块说明

在该设计中，显示模块要求能够显示8位16进制数和1位16进制样例输入，分为以下几个场景。使用时序逻辑，敏感于时钟上升沿。

1. 输入测试样例，在第5位数码管显示
2. 输入测试数据，在第0-4位数码管显示
3. 输出计算结果，在第0-4位数码管显示

在项目的硬件设计中，7段数码管与第24号寄存器绑定，在允许修改的信号位1的情况下，修改24号寄存器内的数值即可实时的更改7段数码管的显示结果。

### 模块结构



### 引脚说明

输入/输出	端口类型	位宽	端口名称
input	wire	1	clk, rst
input	wire	32	in
output	reg	8	segment_led
output	reg	8	seg_en

## 测试说明

测试方法(仿真、上板)	测试类型(单元、集成)	测试用例	测试结果(通过、不通过)
上板	集成	场景1	通过
上板	集成	场景2	通过

## 测试结论

测试通过，CPU功能符合预期。

## 安装

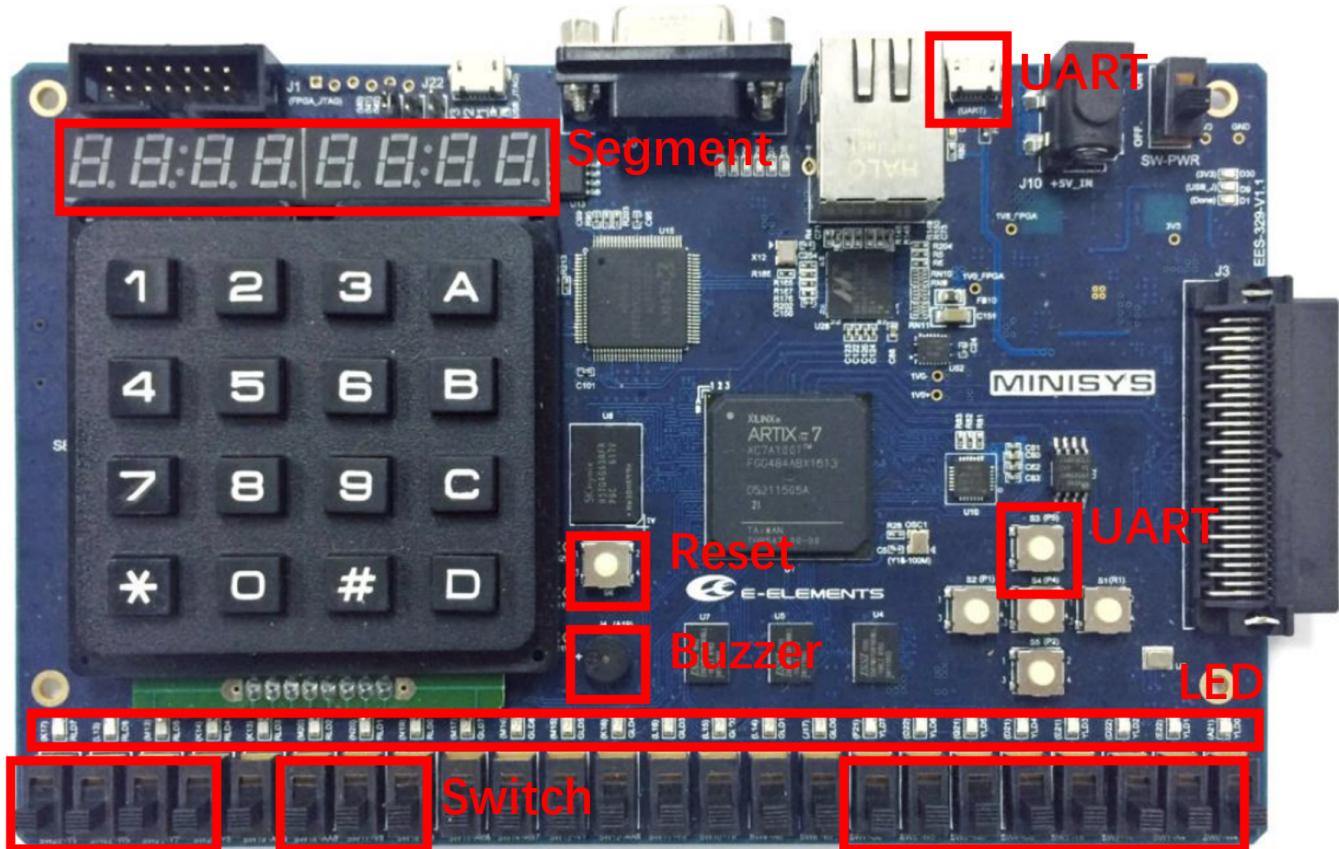
1. 这个项目使用 [Vivado](#) 和 [Mars](#)。请确保您本地安装了它们。
2. 将项目克隆到本地。

```
git clone https://github.com/Kazawaryu/Simple_CPU_in_FPGA
```

3. 运行 **project.xpr** 文件。
4. 生成 **BitStream** 文件，并烧录到您的FPGA硬件设备。
5. 重启您的FPGA硬件设备。

## 使用说明

### 外设使用



## 按键说明

1. 开启CPU：23号开关置1，开启保险，进入main状态。
2. 输入case：修改18 - 16号开关，7段数码管会同步更新您的输入。
3. 提交case：21号开关置1，跳转到对应的case，对于不同的case，可能由不同的输入需求。
4. 输入num：修改7 - 0号开关，7段数码管会同步更新您的输入。
5. 提交num：20号开关置1，CPU执行对应汇编指令，7段数码管会显示对您输入num的处理结果。
6. 回到main：21号开关置0，推荐您同时将20号开关置0。

23	22	21	20	19	18 - 16	15 - 8	7 - 0
保险	BGM	Case 提交	Num 提交	None	3bits Case 输入	None	8bits Num 输入

如果出现了卡死状况，可能是为您跳转到了某个模块的独立状态中，先复位21号开关后，再重拨复位20号开关可以修复该问题。

## 场景一

1. **Case 0** 输入测试数a，输入完毕后在led灯上显示a，同时用1个led灯显示a是否为二的幂的判断。
2. **Case 1** 输入测试数a，输入完毕后在输出设备上显示a，同时用1个led灯显示a是否为奇数的判断。
3. **Case 2** 先执行Case 7, 再计算 a 和 b的按位或运算，将结果显示在输出设备。
4. **Case 3** 先执行Case 7, 再计算 a 和 b的按位或非运算，将结果显示在输出设备。
5. **Case 4** 先执行Case 7, 再计算 a 和 b的按位异或运算，将结果显示在输出设备。
6. **Case 5** 先执行Case 7, 再计算 a 和 b的按位或运算，再执行 slt 指令，将a和b按照有符号数进行比较，用输出设备展示a<b的关系是否成立。
7. **Case 6** 先执行Case 7, 再计算 a 和 b的按位或运算，再执行 sltu 指令，将a和b按照无符号数进行比较，用输出设备展示a<b的关系是否成立。
8. **Case 7** 输入测试数a, 输入测试数b，在输出设备上展示a和b的值。需要提交两次输入数值。

以上用例全部通过上板测试

## 场景二

1. **Case 0** 输入a的数值（a被看作有符号数），计算1到a的累加和，在输出设备上显示累加和（如果a是负数，以闪烁的方式给与提示）。
2. **Case 1** 输入a的数值（a被看作无符号数），以递归的方式计算1到a的累加和，记录本次入栈和出栈次数，在输出设备上显示入栈和出栈的次数之和。
3. **Case 2** 输入a的数值（a被看作无符号数），以递归的方式计算1到a的累加和，记录入栈和出栈的数据，在输出设备上显示入栈的参数，每一个入栈的参数显示停留2-3秒。
4. **Case 3** 输入a的数值（a被看作无符号数），以递归的方式计算1到a的累加和，记录入栈和出栈的数据，在输出设备上显示出栈的参数，每一个出栈的参数显示停留2-3秒。
5. **Case 4** 输入测试数a和测试数b，实现有符号数（a, b以及相加和都是8bit，其中的最高bit被视作符号位，如果符号位为1，表示的是该负数的补码）的加法，并对是否溢出进行判断，输出运算结果以及溢出判断。
6. **Case 5** 输入测试数a和测试数b，实现有符号数（a, b以及差值都是8bit，其中的最高bit被视作符号位，如果符号位为1，表示的是该负数的补码）的减法，并对是否溢出进行判断，输出运算结果以及溢出判断。
7. **Case 6** 输入测试数a和测试数b，实现有符号数（a, b都是8bit，乘积是16bit，其中的最高bit被视作符号位，如果符号位为1，表示的是该负数的补码）的乘法，输出乘积。
8. **Case 7** 输入测试数a和测试数b，实现有符号数（a, b，商和余数都是8bit，其中的最高bit被视作符号位，如

果符号位为1，表示的是该负数的补码）的除法，输出商和余数（商和余数交替显示，各持续5秒）。

以上用例全部通过上板测试

## Insight

- 在开始开发的时候就要确定好reset键按下是高/低电频。
- 如果generate IP核之后发现.dcp文件无法找到，可能是由于IP核生成较慢，未生成结束。此时可以关注右上角的关闭vivado的叉号下方紧邻的位置，如果有绿色圆环转动，说明生成还未结束。
- 在编写汇编代码的时候，注意区分\$3,\$s3的区别，不要忘记在需要的时候加s。
- 在测试的时候请多测试几组，特别是涉及有符号数的运算，最好每一种正负号的组合都测试一下。
- 当uart异常的时候：检查波特率是否是128000（三个零）；选择的文件数据源是否正确，是否随需求的改变重新选择文件数据源；换数据线；如果持续失败可以联系老师更换开发板。
- 注意在绑端口的时候，使用的端口名称对应的是开发板的手册电路图中与写在板子表示符号上面(XC7A100T FGG484C-1)的编号。

## 示例

想了解我们项目如何运行和使用的的，请参考[演示视频](#)。

## 相关仓库

- [CS202\\_CPU\\_Project](#) — ❤ 高质量单周期CPU
- [SUSTech-CS202\\_214-Computer\\_Organization-Project](#) — ❤ 规范模板CPU

## 维护者

[@Kazawaryu](#) [@酷酷的阿涵](#) [@houfm](#)

学号	姓名	负责工作	贡献比
12011126	加泽瑄	场景1，上板测试，蜂鸣器音乐播放	1/3
12111811	刘宇涵	CPU及pipeline设计，上板测试	1/3
12111448	侯芳曼	场景2，上板测试，VGA显示	1/3

## 如何贡献

非常欢迎你的加入！[提一个Issue](#) 或者[提交一个Request](#)。

如果您喜欢我们的项目，给我们一个★是对我们最大的支持。

## 贡献者

感谢以下参与项目的人：



# 使用许可

---

[MIT](#) © Kazawaryu