计算机体系结构实验报告

LAB02

题目： RISC-V 32I 的实现

姓名： 林祥

学号： PB16020923

实验目的

设计一个 RISC-V 32I 指令集的五段流水线 CPU，实现转发和冒险检测。

实验平台

EDA 工具为 Vivado 2017.4

实验设计与过程

出于完整性考虑，该部分基于上次设计报告进一步编写。

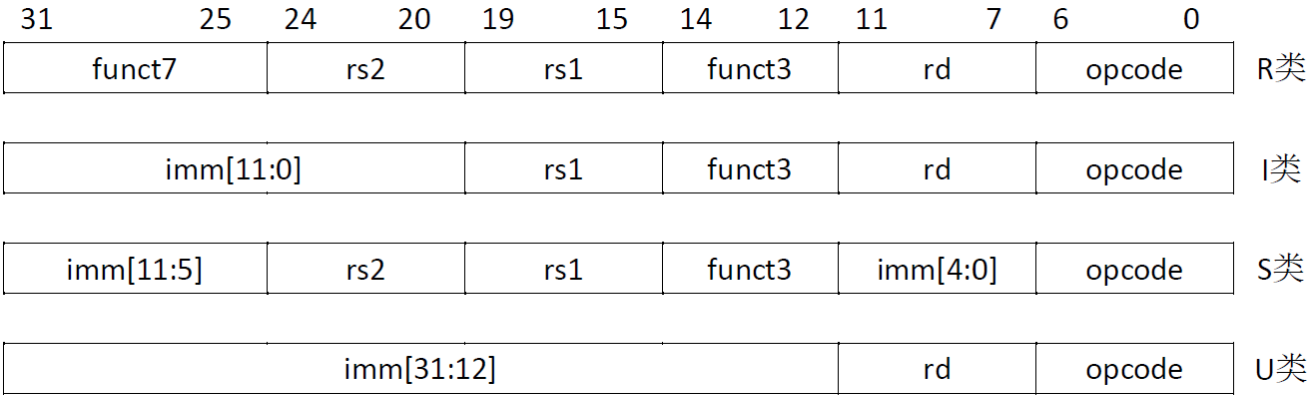
一、 RISC-V 32I 指令格式

有四种核心指令格式（R/I/S/U），所有的指令都是固定 32 位长度的，并且在存储器中必须在 4 字节边界对齐。

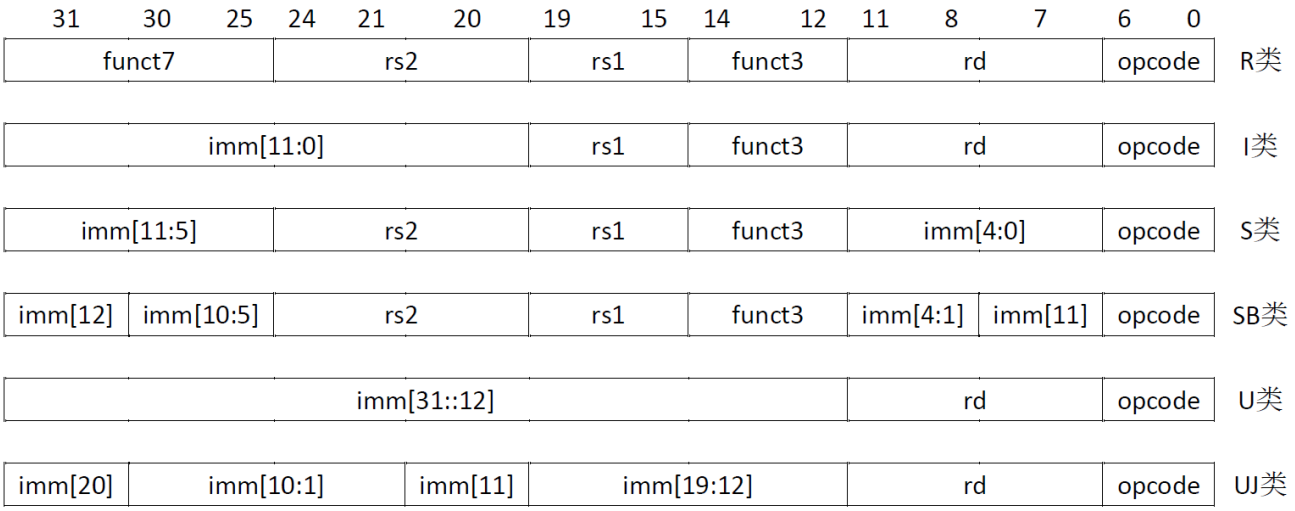
其中 opcode 字段指明了该条指令的操作类型，一般对应一大类指令，再根据 funct3 或/和 funct7 字段来确定具体指令（某些指令直接由 opcode 确定）

rs1,rs2,rd 字段，分别指源操作数 1、源操作数 2、目的操作数的寄存器编号，在所有格式中，这三者**位置是固定的**，方便了指令译码。

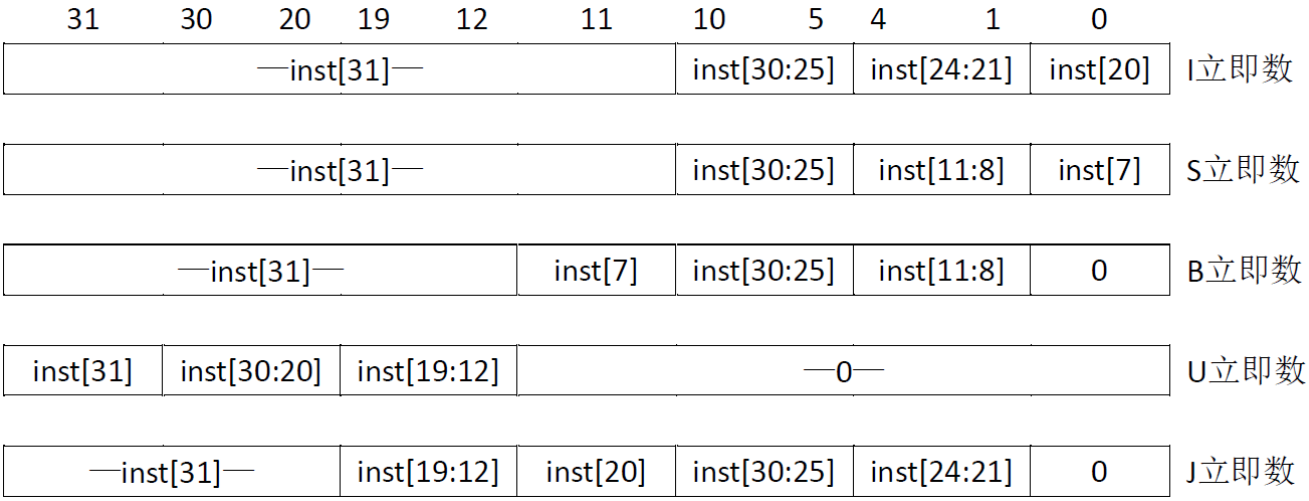
另外，所有立即数的符号位总是在指令的第 31 位，以加速符号扩展电路。



为了方便对立即数的处理，增加了两种指令格式变种，由 S 类派生出的 SB 类，由 U 类派生出的 UJ 类



除了 R 类指令，剩下五类指令中都包含立即数，使用立即数前应该对立即数进行**符号拓展**（32 位），得到以下 5 类立即数



其中 inst[x]表示，立即数的这一位是指令中的第 x 位(0-31)，符号拓展总是在指令的**第 31 位**。

S 和 SB 格式唯一的区别在于，在 SB 格式中，12 位立即数字段用于编码 2 的倍数的分支偏移量。（SB 立即数的第 0 位固定为 0）

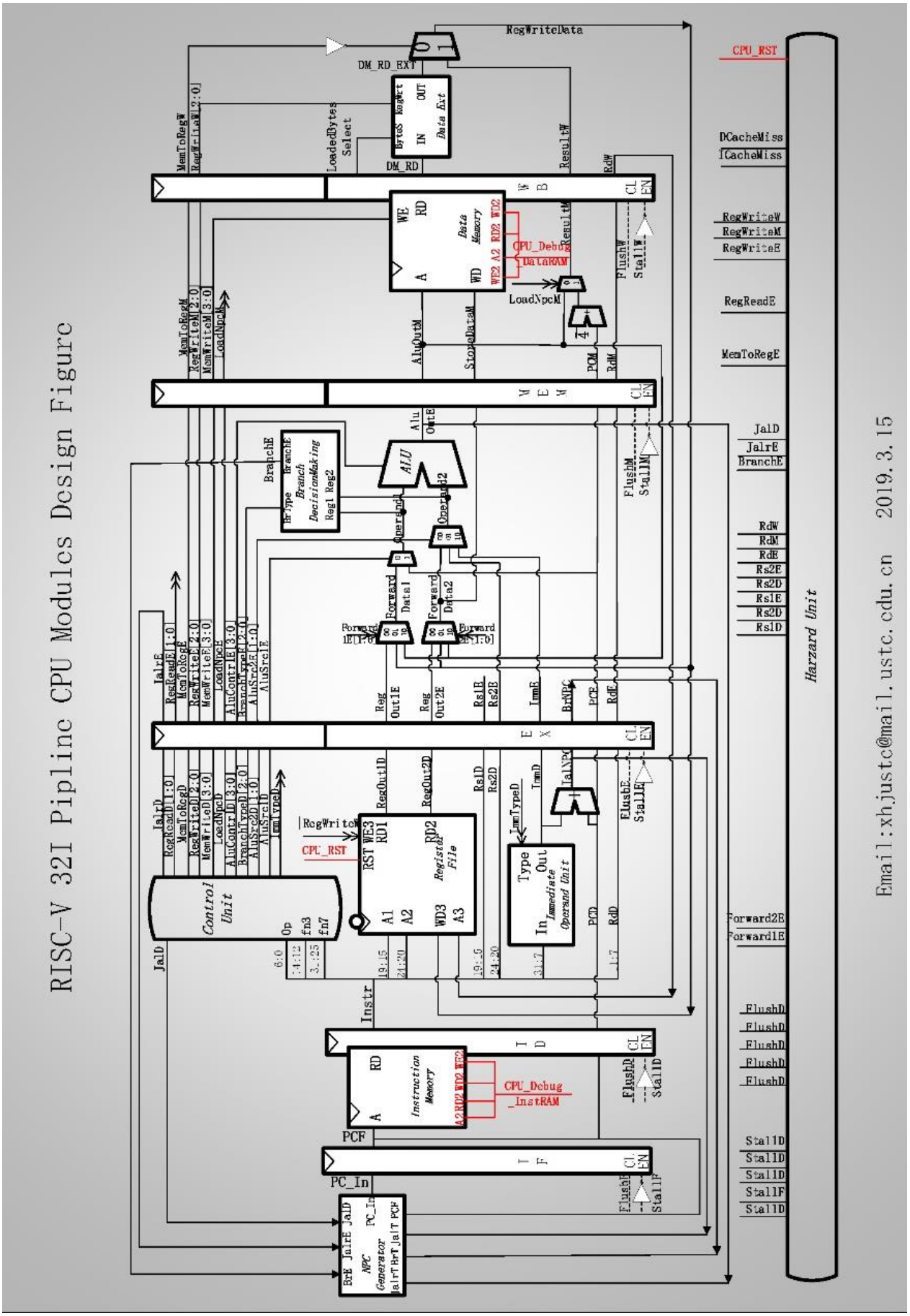
U 和 UJ 格式唯一的区别在于，20 位立即数被左移 12 位以生成 U 立即数，而被左移 1 位以生成 J 立即数。

二、 计划实现的指令列表如下

RISC-V 32I 共 47 条指令，这里选择实现主要的 37 条指令。

指令	类型	Inst[31:25]	Inst[24:20]	Inst[19:15]	Inst[14:12]	Inst[11:7]	Inst[6:0]
LUI	U	imm[31:12]				rd	0110111
AUIPC	U	imm[31:12]				rd	0010111
JAL	J	imm[20|10:1|11|19:12]				rd	1101111
JALR	I	imm[11:0]		rs1	000	rd	1100111
BEQ	B	imm[12|10:5]	rs2	rs1	000	imm[4:1|11]	1100011
BNE		imm[12|10:5]	rs2	rs1	001	imm[4:1|11]	1100011
BLT		imm[12|10:5]	rs2	rs1	100	imm[4:1|11]	1100011
BGE		imm[12|10:5]	rs2	rs1	101	imm[4:1|11]	1100011
BLTU		imm[12|10:5]	rs2	rs1	110	imm[4:1|11]	1100011
BGEU		imm[12|10:5]	rs2	rs1	111	imm[4:1|11]	1100011
LB	I	imm[11:0]		rs1	000	rd	0000011
LH		imm[11:0]		rs1	001	rd	0000011
LW		imm[11:0]		rs1	010	rd	0000011
LBU		imm[11:0]		rs1	100	rd	0000011
LHU		imm[11:0]		rs1	101	rd	0000011
SB	S	imm[11:5]	rs2	rs1	000	imm[4:0]	0100011
SH		imm[11:5]	rs2	rs1	001	imm[4:0]	0100011
SW		imm[11:5]	rs2	rs1	010	imm[4:0]	0100011
ADDI	I	imm[11:0]		rs1	000	rd	0010011
SLTI		imm[11:0]		rs1	010	rd	0010011
SLTIU		imm[11:0]		rs1	011	rd	0010011
XORI		imm[11:0]		rs1	100	rd	0010011
ORI		imm[11:0]		rs1	110	rd	0010011
ANDI		imm[11:0]		rs1	111	rd	0010011
SLLI		0000000	shamt	rs1	001	rd	0010011
SRLI		0000000	shamt	rs1	101	rd	0010011
SRAI		0100000	shamt	rs1	101	rd	0010011
ADD	R	0000000	rs2	rs1	000	rd	0110011
SUB		0100000	rs2	rs1	000	rd	0110011
SLL		0000000	rs2	rs1	001	rd	0110011
SLT		0000000	rs2	rs1	010	rd	0110011
SLTU		0000000	rs2	rs1	011	rd	0110011
XOR		0000000	rs2	rs1	100	rd	0110011
SRL		0000000	rs2	rs1	101	rd	0110011
SRA		0100000	rs2	rs1	101	rd	0110011
OR		0000000	rs2	rs1	110	rd	0110011
AND		0000000	rs2	rs1	111	rd	0110011

三、 整体设计图


RISC-V 32I Pipline CPU Modules Design Figure
Email:xhjustc@mail.ustc.edu.cn 2019.3.15
Harzard Unit
RegWriteData
DM_RD_EXT
DM_RD
LoadedBytes Select
ResultW
CPU_RST
DCacheMiss
ICacheMiss
RegWriteW
RegWriteM
RegWriteE
RegReadE
MemToRegE
JalD
JalrE
BranchE
RdW
RdM
RdE
Rs2E
Rs2D
Rs1E
Rs2D
Rs1D
Forward2E
Forward1E
FlushD
StallD
StallF
Data Memory
CPU_Debug _DataRAM
LoadNpcM
ALU
Branch DecisionMaking
Forward Data1
Forward Data2
Control Unit
Register File
Immediate Operand Unit
Instruction Memory
CPU_Debug _InstRAM
PCF
PC_In
NPC Generator
RegWrite
Instr

四、 常量定义

预先在模块 Parameters.v 中定义实验中使用的常量，方便阅读

```
`ifndef CONST_VALUES
`define CONST_VALUES
//ALUContrl[3:0]
    `define SLL  4'd0
    `define SRL  4'd1
    `define SRA  4'd2
    `define ADD  4'd3
    `define SUB  4'd4
    `define XOR  4'd5
    `define OR  4'd6
    `define AND  4'd7
    `define SLT  4'd8
    `define SLTU  4'd9
    `define LUI  4'd10
//BranchType[2:0]
    `define NOBRANCH  3'd0
    `define BEQ  3'd1
    `define BNE  3'd2
    `define BLT  3'd3
    `define BLTU  3'd4
    `define BGE  3'd5
    `define BGEU  3'd6
//ImmType[2:0]
    `define RTYPE  3'd0
    `define ITYPE  3'd1
    `define STYPE  3'd2
    `define BTYPE  3'd3
    `define UTYPE  3'd4
    `define JTYPE  3'd5
//RegWrite[2:0]  six kind of ways to save values to Register
    `define NOREGWRITE  3'b0    //  Do not write Register
    `define LB  3'd1           //  load 8bit from Mem then signed extended to
32bit
    `define LH  3'd2           //  load 16bit from Mem then signed extended to
32bit
    `define LW  3'd3           //  write 32bit to Register
    `define LBU  3'd4          //  load 8bit from Mem then unsigned extended to
32bit
    `define LHU  3'd5          //  load 16bit from Mem then unsigned extended to
32bit
`endif
```

五、 各模块具体设计与编写

由于我是 3 个阶段的实验一起完成，以下不分阶段讨论。

ALU.v			
输入/输出	宽度	信号名	说明
input	[31:0]	Operand1	无符号型的操作数 a，如果有负数，是以补码存储
input	[31:0]	Operand2	无符号型的操作数 b，如果有负数，是以补码存储
input	[4:0]	AluContrl	运算类型
output	[31:0]	AluOut	无符号型的运算结果，如果有负数，是以补码存储

ALU 模块完成大部分算术逻辑指令的计算，用 case 判断 AluContrl 的值（已定义在 Parameters.v 中），使用 Verilog 语言中的各运算符完成运算。

值得注意的地方有：

（1）SLT 和 SLTU 的实现

由于 Verilog 的信号**默认为无符号数**，如果使用减法进行判断

```
AluOut = (Operand1 - Operand2 < 0) ? 32'b1 : 32'b0;
```

是不对的，无符号数的运算结果总为无符号数，运算结果大于等于 0 是恒成立的，这样的判断将总是指向分支 32’b0。

正确的方法是比较大小，对于 SLTU 可以直接比较

```
AluOut = (Operand1 < Operand2) ? 32'b1 : 32'b0;
```

而 SLT 则应该是有符号的比较，但 Verilog 是视为无符号数的比较，如果直接按上面的方式，那么以补码存储的负数比任何正数大。

一种可选的方法是：

```
if(Operand1[31] == Operand2[31]) AluOut = (Operand1 < Operand2) ? 32'b1 : 32'b0;
//首位相等，即同号情况，直接比较，如果同正，后面 31 位大的，原数就大，如果同负，后面 31 位（补码）大的，依然是原数大
else AluOut = (Operand1[31] < Operand2[31]) ? 32'b0 : 32'b1;
//异号情况，直接比较符号
```

其实，有更简单的方法，使用 signed 函数转为有符号数

```
AluOut = ($signed(Operand1) < $signed(Operand2)) ? 32'b1 : 32'b0;
```

（2）SLL、SRL、SRA

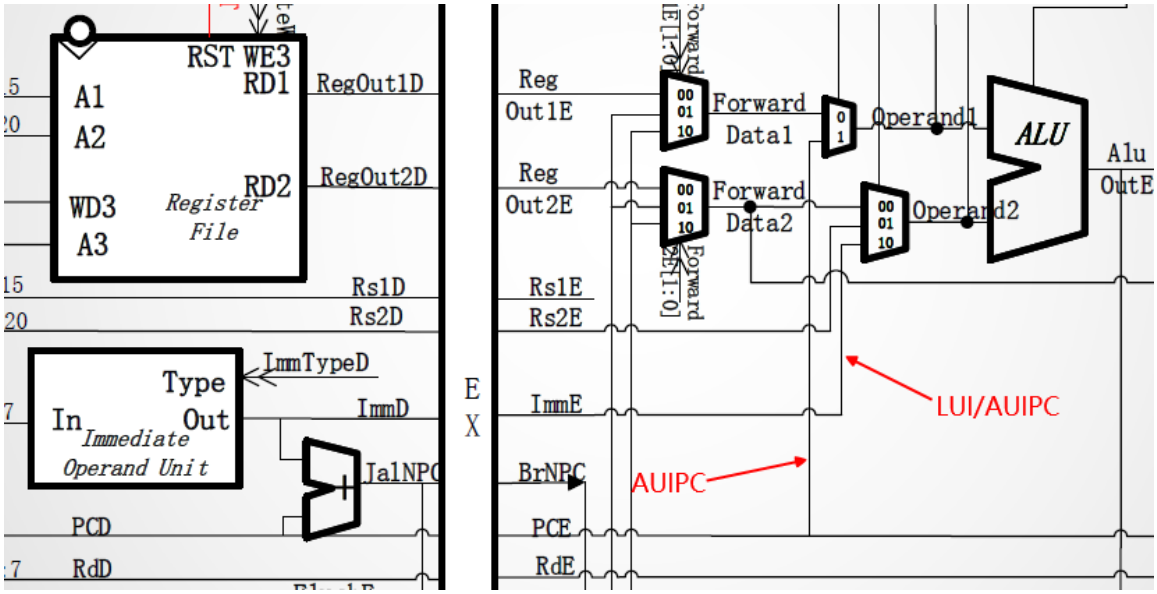
Operand1 为被移位的操作数，Operand2 为移位位数，应该**只取最低 5bit**，因为最多只能移位 32 位。

SRA 是算术右移，高位填充符号位，因此 Operand1 先用`$signed` 转为有符号数。

（3）比较特殊的是 LUI 指令和 AUIPC 指令。

LUI 指令直接输出 Operand2 作为结果，ALU 需要实现该操作。因为当 AluContrl 选择为 LUI 时，AluSrc2D 选择为 2’b10，Operand2 输入即为 U 立即数拓展的结果，拓展过程在 ImmOperandUnit 模块实现，ALU 无需再计算。

而 AUIPC 需要将拓展的 U 立即数加到 PC 上（AluSrc1D 选择为 1’b1 且 AluSrc2D 选择为 2’b10），此时使用 ALU 的 ADD 功能即可，无需特殊实现



具体代码如下：

```
`include "Parameters.v"
module ALU(
    input wire [31:0] Operand1,
    input wire [31:0] Operand2,
    input wire [3:0] AluContrl,
    output reg [31:0] AluOut
    );
    always@(*)
        case (AluContrl)
            `ADD: AluOut = Operand1 + Operand2;
            `SUB: AluOut = Operand1 - Operand2;
            `AND: AluOut = Operand1 & Operand2;
            `OR : AluOut = Operand1 | Operand2;
            `XOR: AluOut = Operand1 ^ Operand2;
            `SLT: AluOut = ($signed(Operand1) < $signed(Operand2)) ? 32'b1 : 32'b0;//使用
$signed()
            `SLTU: AluOut = (Operand1 < Operand2) ? 32'b1 : 32'b0;
            `SLL: AluOut = Operand1 << Operand2[4:0];
            `SRL: AluOut = Operand1 >> Operand2[4:0];
            `SRA: AluOut = $signed(Operand1) >>> Operand2[4:0];
            //使用>>>为算术右移，高位补符号，应该注意，如果是无符号数，>>>仍是逻辑右移，故应该加$signed
            `LUI: AluOut = Operand2;
            default: AluOut = 32'hxxxxxxxx;
        endcase
endmodule
```

ImmOperandUnit.v			
输入/输出	宽度	信号名	说明
input	[31:7]	In	32 位长度的指令除去操作码的部分
input	[2:0]	Type	表示立即数编码类型
output	[31:0]	Out	指令对应的立即数 32 位拓展值

这个模块将分散在指令各个位置的立即数片段按照特定的立即数类型（已定义在 Parameters.v 中）进行组合，并进行符号拓展。

对信号进行**位拼接**可利用 Verilog 的{}，例如

```
Out <= { {20{In[31]}}, In[7], In[30:25], In[11:8], 1'b0 }
```

花括号前面还可以指定重复的次数。

具体代码如下：

```
`include "Parameters.v"
module ImmOperandUnit(
    input wire [31:7] In,
    input wire [2:0] Type,
    output reg [31:0] Out
    );
    //
    always@(*)
    begin
        case(Type)
            `ITYPE: Out <= { {21{In[31]}}, In[30:20] };
            `STYPE: Out <= { {21{In[31]}}, In[30:25], In[11:7] };
            `BTYPE: Out <= { {20{In[31]}}, In[7], In[30:25], In[11:8], 1'b0 };
            `UTYPE: Out <= { In[31:12], {12{1'b0}} };
            `JTYPE: Out <= { {12{In[31]}}, In[19:12], In[20], In[30:21], 1'b0 };
            default: Out <= 32'hxxxxxxxx;
        endcase
    end
endmodule
```

NPC_Generator.v			
输入/输出	宽度	信号名	说明
input	[31:0]	PCF	旧的 PC 值
input	[31:0]	JalrTarget	jalr 指令的对应的跳转目标
input	[31:0]	BranchTarget	branch 指令的对应的跳转目标
input	[31:0]	JalTarget	jal 指令的对应的跳转目标
input	[0:0]	BranchE	Ex 阶段的 Branch 指令确定跳转
input	[0:0]	JalD	ID 阶段的 Jal 指令确定跳转
input	[0:0]	JalrE	Ex 阶段的 Jalr 指令确定跳转
output	[31:0]	PC_In	NPC 的值

这个模块通过输入的跳转信号来生成 NPC 的值，可通过几个 if 判断实现。值得注意的是，判断是**有优先级**的，JalD 的判断不能先于 JalrE 和 BranchE 的判断，因为 EX 传来的信号比 ID 段传来的信号早一条指令，应该先满足最早跳转的指令，其后的指令是要被舍弃的。比如当前指令是 BEQ，下一条指令是 JAL，它们会在同一周期完成确认是否跳转，如果 BEQ 确认跳转应该先满足 BEQ 指令，JAL 指令被清除。

具体代码如下：

```
module NPC_Generator(
    input wire [31:0] PCF,JalrTarget, BranchTarget, JalTarget,
    input wire BranchE,JalD,JalrE,
    output reg [31:0] PC_In
    );
    always@(*)
        if(BranchE)
            PC_In = BranchTarget;
        else if(JalrE)
            PC_In = JalrTarget;
        else if(JalD)//这个不能放在上面两种情况之前，因为 EX 段的信号比 ID 段的信号早一条指令，跳转应该是先满足最早的
            PC_In = JalTarget;
        else
            PC_In = PCF + 4;
endmodule
```

BranchDecisionMaking.v			
输入/输出	宽度	信号名	说明
input	[2:0]	BranchTypeE	Branch 的类型，如 BEQ/BNE…
input	[31:0]	Operand1	操作数 1
input	[31:0]	Operand2	操作数 2
output	[31:0]	BranchE	是否跳转，为 1 表示跳转

这个模块通过输入的 Branch 指令的类型（已定义在 Parameters.v 中）和两个操作数的比较结果来生成决定跳转的信号。注意到 BLT 和 BGE 的比较是有符号数的比较。

具体代码如下：

```
`include "Parameters.v"
module BranchDecisionMaking(
    input wire [2:0] BranchTypeE,
    input wire [31:0] Operand1,Operand2,
    output reg BranchE
    );
    always@(*)
        case (BranchTypeE)
            `NOBRANCH: BranchE = 1'b0;//NOBRANCH
            `BEQ: BranchE = Operand1 == Operand2 ? 1'b1 : 1'b0; //BEQ
            `BNE: BranchE = Operand1 != Operand2 ? 1'b1 : 1'b0; //BNE
            `BLT: BranchE = $signed(Operand1) < $signed(Operand2) ? 1'b1 : 1'b0; //BLT
            `BGE: BranchE = $signed(Operand1) >= $signed(Operand2) ? 1'b1 : 1'b0; //BGE
            `BLTU: BranchE = Operand1 < Operand2 ? 1'b1 : 1'b0; //BLTU
            `BGEU: BranchE = Operand1 >= Operand2 ? 1'b1 : 1'b0; //BGEU
           default: BranchE = 1'b0;
        endcase
endmodule
```

DataExt.v			
输入/输出	宽度	信号名	说明
input	[31:0]	IN	从数据内存中读取的 32 位字
input	[1:0]	LoadedBytesSelect	访存地址的低二位
input	[2:0]	RegWriteW	寄存器写入模式，如 LW,LH，定义在
output	[31:0]	OUT	实际要写入寄存器的数据

这个模块处理 Load 类型指令及访存地址**非字对齐**的情况。

由于 Data Memory 是按字访问的，内存地址的低 2 位未使用，要在访存后根据寄存器写入模式（已定义在 Parameters.v 中）及低 2 位地址进行进一步的选择：

对于 LW 模式，无需处理，OUT = IN.

对于 LH 模式，可以按半字读取，低 2 位地址可能是 00 或者 10，则从访存结果截取低 2 字节或者高 2 字节作为寄存器写入值的低 2 字节，高 2 字节进行符号拓展.

对于 LB 模式，可以按字节读取，低 2 位地址可能是 00/01/10/11，则从访存结果截取第 0/1/2/3 字节作为寄存器写入值的第 0 字节，高 3 字节进行符号拓展.

对于 LHU 模式，与 LH 模式类似，只是高 2 字节位无符号拓展

对于 LBU 模式，与 LB 模式类似，只是高 3 字节位无符号拓展

具体代码如下：

```
`include "Parameters.v"
module DataExt(
    input wire [31:0] IN,
    input wire [1:0] LoadedBytesSelect,
    input wire [2:0] RegWriteW,
    output reg [31:0] OUT
    );
    always@(*)
        case(RegWriteW)
```

```
            `NOREGWRITE: OUT = 32'hxxxxxxxx;
            `LB:
               case(LoadedBytesSelect)
                  2'b00: OUT = {{24{IN[ 7]}}, IN[ 7: 0]};
                  2'b01: OUT = {{24{IN[15]}}, IN[15: 8]};
                  2'b10: OUT = {{24{IN[23]}}, IN[23:16]};
                  2'b11: OUT = {{24{IN[31]}}, IN[31:24]};
                  default: OUT = 32'hxxxxxxxx;
               endcase
            `LH:
               case(LoadedBytesSelect)
                  2'b00: OUT = {{16{IN[15]}}, IN[15: 0]};
                  2'b10: OUT = {{16{IN[31]}}, IN[31:16]};
                  default: OUT = 32'hxxxxxxxx;
               endcase
            `LW:
               OUT = IN;
            `LBU:
               case(LoadedBytesSelect)
                  2'b00: OUT = {{24{1'b0}}, IN[ 7: 0]};
                  2'b01: OUT = {{24{1'b0}}, IN[15: 8]};
                  2'b10: OUT = {{24{1'b0}}, IN[23:16]};
                  2'b11: OUT = {{24{1'b0}}, IN[31:24]};
                  default: OUT = 32'hxxxxxxxx;
               endcase
            `LHU:
               case(LoadedBytesSelect)
                  2'b00: OUT = {{16{1'b0}}, IN[15: 0]};
                  2'b10: OUT = {{16{1'b0}}, IN[31:16]};
                  default: OUT = 32'hxxxxxxxx;
               endcase
            default: OUT = 32'hxxxxxxxx;
         endcase
endmodule
```

RegisterFile.v			
输入/输出	宽度	信号名	说明
input	[0:0]	clk	Cpu 的时钟信号
input	[0:0]	rst	Cpu 的复位信号
input	[0:0]	WE3	3 号读写口的写使能
input	[4:0]	A1	1 号读口的读地址
input	[4:0]	A2	2 号读口的读地址
input	[4:0]	A3	3 号读写口的读写地址
input	[31:0]	WD3	3 号读写口的写数据
output	[31:0]	RD1	1 号读口的读数据
output	[31:0]	RD2	2 号读口的读数据

这个模块实现了 32 个 32 位宽的寄存器组（0 号寄存器不允许写入），有两个读端口，一个写端口，数据为同步读写。与时钟**下降沿**同步的原因是：流水段寄存器与时钟上升沿同步，而两个段寄存器之间的操作应该在一个时钟周期内完成，如果寄存器也与时钟上升沿同步，则需要两个周期。

具体代码：

略

EXSegReg.v			
输入/输出	宽度	信号名	说明
input	[0:0]	clk	时钟信号
input	[0:0]	en	段寄存器使能信号，表示是否更新值
input	[0:0]	clear	段寄存器清空信号
…	…	…	各个数据信号与控制信号的输入与输出，此处略去

这个模块实现将 ID 段的各个控制信号和数据信号传递到 EX 段。

具体代码：

略。

IFSegReg.v			
输入/输出	宽度	信号名	说明
input	[0:0]	clk	时钟信号
input	[0:0]	en	段寄存器使能信号，表示是否更新值
input	[0:0]	clear	段寄存器清空信号
input	[31:0]	PC_In	
output	[31:0]	PCF	

这个模块实现 PC 寄存器，将 NPC_Generator 生成的 PC 值传递到 IF 段。

具体代码如下：

```
module IFSegReg(
    input wire clk,
    input wire en, clear,
    input wire [31:0] PC_In,
    output reg [31:0] PCF
    );
    initial PCF = 0;

    always@(posedge clk)
        if(en) begin
            if(clear)
                PCF <= 0;
            else
                PCF <= PC_In;
        end
endmodule
```

MEMSegReg.v			
输入/输出	宽度	信号名	说明
input	[0:0]	clk	时钟信号
input	[0:0]	en	段寄存器使能信号，表示是否更新值
input	[0:0]	clear	段寄存器清空信号
…	…	…	各个数据信号与控制信号的输入与输出，此处略去

这个模块实现将 EX 段的各个控制信号和数据信号传递到 MEM 段。

具体代码：

略。

IDSegReg.v			
输入/输出	宽度	信号名	说明
input	[0:0]	clk	时钟信号
input	[0:0]	en	段寄存器使能信号，表示是否更新值
input	[0:0]	clear	段寄存器清空信号
input	[31:0]	A	指令存储器的端口 1 读地址
output	[31:0]	RD	指令存储器的端口 1 读数据
input	[31:0]	A2	指令存储器的端口 2 读写地址，用于 DEBUG
input	[31:0]	WD2	指令存储器的端口 2 写数据，用于 DEBUG
input	[3:0]	WE2	指令存储器的端口 2 写使能，用于 DEBUG
output	[31:0]	RD2	指令存储器的端口 2 读数据，用于 DEBUG
input	[31:0]	PCF	
output	[31:0]	PCD	

这个模块实现 IF/ID 段之间的寄存器，将 PC 值从 IF 段传到 ID 段，并且在 ID 寄存器内实现（**嵌入**）了指令存储器（与时钟上升沿同步）。

值得注意的是，流水段的两个段寄存器之间的所有操作应该在**一个周期**内完成。显然，组合逻辑电路可以满足。如果 Memory 是 Distributed Memory 就可以进行异步访问，不用嵌入在段寄存器中。如果 Memory 是 Block Memory 只能同步访问，可以有**两种解决方法**，一是与时钟下降沿同步，和段寄存器错开（这也是之前 Regfile 的做法），二是和时钟上升沿同步，但应该和段寄存器作为一个整体，也就是嵌入在段寄存器中，否则两个段寄存器之间需要额外一个周期。

另外，指令存储器的读地址应该传入 A[31:2]，按字节读取。

例化 InstructionRam 的代码如下：

```
    InstructionRam InstructionRamInst (
        .clk    ( clk       ),                      //请完善代码!!!
        .addra  ( A[31:2]    ),                      //请完善代码!!!
        .douta  ( RD_raw     ),
        .web    ( |WE2       ),
        .addrb  ( A2[31:2]   ),
        .dinb   ( WD2        ),
        .doutb  ( RD2        )
    );
```

WBSegReg.v			
输入/输出	宽度	信号名	说明
input	[0:0]	clk	时钟信号
input	[0:0]	en	段寄存器使能信号，表示是否更新值
input	[0:0]	clear	段寄存器清空信号
…	…	…	各个数据信号与控制信号的输入与输出，此处略去
input	[31:0]	A	数据存储器的端口 1 读写地址
input	[31:0]	WD	数据存储器的端口 1 写数据
input	[3:0]	WE	数据存储器的端口 1 写使能，来自 ControlUnit 的 MemWirte
output	[31:0]	RD	数据存储器的端口 1 读数据
input	[31:0]	A2	数据存储器的端口 2 读写地址，用于 DEBUG
input	[31:0]	WD2	数据存储器的端口 2 写数据，用于 DEBUG
input	[3:0]	WE2	数据存储器的端口 2 写使能，用于 DEBUG
output	[31:0]	RD2	数据存储器的端口 2 读数据，用于 DEBUG

这个模块实现将 MEM 段的各个控制信号和数据信号传递到 WB 段,同时在内部实现了数据存储器，将数据存储器**嵌在**段寄存器的原因同指令存储器。

SW、SH、SB 指令分别将从 rs2 寄存器低位开始的 32 位、16 位、8 位数值保存到存储器中,则 ControlUnit 中 MemWriteD 信号分别置为 4’b1111、4’b0011、4’b0001（让存储器按字节写入，写使能是**独热编码**，1bit 对应写入一字节）。

在例化 DataMem 模块时，并不能将 WB 段寄存器传入的 WE 和 WD 信号直接接入，因为写入地址的低 2 位不一定是 2’b00，应该需要进行移位，这样就可以实现**非字对齐**的写入。

设地址低 2 位为 A[1:0]，则写使能应该接入 `WE << A[1:0]`，写数据应该移位对应 bit 数量的字节，一种方法是用 case 枚举，另一种比较简洁的写法是 `WD << (A[1:0] * 8)`，乘 8 表示移位单位为字节，

这里还有一点不足，乘 8 运算在综合的时候会生成乘法器，使得效率降低（或者可能不能在一个周期内完成运算）。由于乘数正好为 2 的幂次，因此可以利用移位 3 位实现，即 `WD << ({3'b000, A[1:0]} << 3'd3)`

因此，例化 DataRam 的代码如下：

```
    DataRam DataRamInst (
        .clk    ( clk          ),    //请补全
        .wea    ( WE << A[1:0]  ),  //请补全  不能用 LoadedBytesSelect，此时这还是上一个
上升沿的地址的末 2 位
        .addra  ( A[31:2]      ),         //请补全
        .dina   ( WD << ({3'b000, A[1:0]} << 3'd3)  ),       //请补全
        .douta  ( RD_raw        ),
        .web    ( WE2          ),
        .addrb  ( A2[31:2]      ),
        .dinb   ( WD2          ),
        .doutb  ( RD2          )
    );
```

R32VCore.v			
输入/输出	宽度	信号名	说明
input	[0:0]	CPU_CLK	时钟信号
input	[0:0]	CPU_RST	复位信号
…	…	…	用于 DEBUG 的信号

这个模块是顶层模块，实现对各个模块的连接

具体代码：

略。

ControlUnit.v			
输入/输出	宽度	信号名	说明
input	[6:0]	Op	指令的操作码
input	[2:0]	Fn3	指令的 Funct3 字段
input	[6:0]	Fn7	指令的 Funct7 字段
output	[0:0]	JalD	JAL 指令是否跳转
output	[0:0]	JalD	JALR 指令是否跳转
output	[2:0]	RegWriteD	WB 段写回寄存器的写入模式，定义在 Parameters.v 中
output	[0:0]	MemToRegD	写回寄存器的数据是否来自存储器
output	[3:0]	MemWriteD	数据存储器的写使能，独热编码
output	[0:0]	LoadNpcD	用于 JAL 和 JALR，将 PC+4 写入目标寄存器
output	[1:0]	RegReadD	表示当前指令是否用到了寄存器的结果
output	[2:0]	BranchTypeD	Branch 指令的类型，定义在 Parameters.v 中
output	[3:0]	AluContrlD	ALU 的操作类型，定义在 Parameters.v 中
output	[0:0]	AluSrc1D	ALU 的源操作数 1 的来源
output	[1:0]	AluSrc2D	ALU 的源操作数 2 的来源
output	[2:0]	ImmType	当前指令的立即数类型，定义在 Parameters.v 中

这个模块实现控制信号的生成，是 CPU 的核心部分，各个信号具体分析如下：

（1）JalD

表示 JAL 指令是否跳转，为 1 表示跳转，只有当前指令是 JAL 时，置为 1，其他指令均置为 0。

（2）JalrD

表示 JALR 指令是否跳转，为 1 表示跳转，只有当前指令是 JALR 时，置为 1，其他指令均置为 0。

（3）RegWriteD

表示寄存器文件的写入模式，已定义在 Parameters.v 中。

- LW,LH,LB,LHU,LBU 指令分别选择对应模式（`LW,`LH,`LB,`LHU,`LBU）。
- SW,SH,SB,BEQ,BNE,BLT,BGE,BLTU,BGEU 指令选择`NOREGWRITE 模式。

- 其余指令全部按字节写入寄存器，选择`LW 模式

（4）MemToRegD

表示写入 WB 段写入寄存器文件的数据来源,为 1 表示来自数据存储器。

- 只有 LW,LH,LB,LHU,LBU 指令置为 1
- 其他指令都置为 0

（5）MemWriteD

表示数据存储器的写入模式，按独热编码，1bit 对应一字节。

- SB 指令置为 4'b0001
- SH 指令置为 4'b0011
- SW 指令置为 4'b1111
- 其余指令没有写入数据存储器，均置为 4'b0000

关于写入数据非字节对齐的处理，放在 WB 段寄存器中，具体见问题回答部分。

（6）LoadNpcD

表示写入寄存器文件数据来源，在 MemToRegD == 0 的时候起作用，在 ALU 的计算结果和 PC+4 中进行选择。

- 只有 JAL 和 JALR 需要将该信号置为 1
- 其他指令全部置为 0。

（7）RegReadD

表示当前指令是否用到了寄存器文件。

RegReadD[1] == 1 表示 A1 端口对应的寄存器值被使用到了。

RegReadD[0] == 1 表示 A2 端口对应的寄存器值被使用到了。

各类指令对应如下

	RegReadD[1]	RegReadD[0]
R 型指令	1	1
I 型指令	1	0
S 型指令	1	1
B 型指令	1	1
J 型指令	0	0
U 型指令	0	0

（8）BranchTypeD

表示 Branch 指令的类型，已定义在 Parameters.v 中。

- BEQ,BNE,BLT,BGE,BLTU,BGEU 指令分别选择：

  `BEQ,`BNE,`BLT,`BGE,`BLTU,`BGEU

- 其余指令全部选择`NOBRANCH

（9）AluContrlD

表示 ALU 模块的操作类型，已定义在 Parameters.v 中。

- ADD,SUB,SLL,SLT,SLTU,XOR,SRL,SRA,OR,AND 指令分别选择

`ADD,`SUB,`SLL,`SLT,`SLTU,`XOR,`SRL,`SRA,`OR,`AND

- ADDI,SLLI,SLTI,SLTIU,XORI,SRLI,SRAI,ORI,ANDI 指令分别选择

`ADD,`SLL,`SLT,`SLTU,`XOR,`SRL,`SRA,`OR,`AND

- LW,LH,LB,LHU,LBU,SW,SH,SB 指令选择`ADD（需要计算访存地址）
- BEQ,BNE,BLT,BGE,BLTU,BGEU 指令不需要 ALU，可以置为 4'bxxxx
- LUI 指令选择`LUI，AUIPC 指令选择`ADD，具体原因已在 ALU 模块中分析。
- JAL 指令不需要 ALU，置为 4'bxxxx。JALR 指令选择`ADD，因为需要计算跳转目标地址（I 立即数+rs1）

（10）AluSrc1D

表示 ALU 模块第一个操作数的来源。

- AUIPC 指令需要将立即数加到 PC 上，因此置为 1，表示选择 PC。
- JAL 指令不需要 ALU，可以置为 x，也可以置为 0
- 其余指令全部置为 0，表示来自 rs1 寄存器。

（11）AluSrc2D

表示 ALU 模块第二个操作数的来源。

- R 类型的指令全部置为 2'b00，表示来自 rs2 寄存器。
- B 类型的指令全部置为 2'b00，表示来自 rs2 寄存器。这里应该注意，虽然 Branch 类指令没有用到 ALU，但 BranchDecisionMaking 模块的输入就是 ALU 的两个输入操作数，因此不能置为 2'bxx
- S 类型的指令全部置为 2'b10，表示来自立即数。
- U 类型的指令全部置为 2'b10，表示来自立即数。
- J 类型的指令只有 JAL，不需要 ALU，可以置为 2'bxx
- I 类型的指令除了 SLLI,SRLI,SRAI 全部置为 2'b10，表示来自立即数。
- SLLI,SRLI,SRAI 这三条指令比较特殊，置为 2'b01。虽然需要立即数，但不需要拓展，直接来自 Shamt 字段。

（12）ImmType

表示当前指令立即数的类型（事实上 RTYPE 没有立即数），已定义在 Parameters.v 中，作为 ImmOperandUnit 的输入。

- R 型指令，I 型指令，S 型指令，B 型指令，U 型指令，J 型指令分别选择`RTYPE,`ITYPE,`STYPE,`BTYPE,`UTYPE,`JTYPE。

具体代码的编写：

1. 一种方法是对 Op、Fn7、Fn3 按层次进行 case 枚举，分出不同指令的情况，然后在每种指令的情况下对输出的每个信号进行赋值。这样编写出来的代码虽然十分清楚，但过于冗长。

2. 另一种方法是用位拼接进行赋值。先定义一个宏表示输出的信号拼接，

```
`define ControlOut
{{JalD,JalrD},{MemToRegD},{RegWriteD},{MemWriteD},{LoadNpcD},{RegReadD},{Bra
nchTypeD},{AluContrlD},{AluSrc1D,AluSrc2D},{ImmType}}
```

然后在每种情况下对`ControlOut 进行赋值，这种写法比上一种写法紧凑，但依然较为繁杂。

```
always@(*)
    case(Op)
        7'b0000011: //Load
            case(Fn3)
                3'b000: /* LB */ `ControlOut = {{2'b0_0},{1'b1,`LB},{4'b0000},{1'b0},{2'b10},{`NOBRANCH},{`ADD},{3'b0_10},{`ITYPE}};
                3'b001: /* LH */ `ControlOut = {{2'b0_0},{1'b1,`LH},{4'b0000},{1'b0},{2'b10},{`NOBRANCH},{`ADD},{3'b0_10},{`ITYPE}};
                3'b010: /* LW */ `ControlOut = {{2'b0_0},{1'b1,`LW},{4'b0000},{1'b0},{2'b10},{`NOBRANCH},{`ADD},{3'b0_10},{`ITYPE}};
                3'b100: /* LBU */ `ControlOut = {{2'b0_0},{1'b1,`LBU},{4'b0000},{1'b0},{2'b10},{`NOBRANCH},{`ADD},{3'b0_10},{`ITYPE}};
                3'b101: /* LHU */ `ControlOut = {{2'b0_0},{1'b1,`LHU},{4'b0000},{1'b0},{2'b10},{`NOBRANCH},{`ADD},{3'b0_10},{`ITYPE}};
            endcase
        7'b0100011: //Store
            case(Fn3)
                3'b000: /* SB */ `ControlOut = {{2'b0_0},{1'b0,`NOREGWRITE},{4'b0001},{1'b0},{2'b11},{`NOBRANCH},{`ADD},{3'b0_10},{`STYPE}};
                3'b001: /* SH */ `ControlOut = {{2'b0_0},{1'b0,`NOREGWRITE},{4'b0011},{1'b0},{2'b11},{`NOBRANCH},{`ADD},{3'b0_10},{`STYPE}};
                3'b010: /* SW */ `ControlOut = {{2'b0_0},{1'b0,`NOREGWRITE},{4'b1111},{1'b0},{2'b11},{`NOBRANCH},{`ADD},{3'b0_10},{`STYPE}};
            endcase
        7'b0110011: //REG-REG
            case(Fn3)
                3'b000: /*  */
                    case(Fn7)
                        7'b0000000: /* ADD */ `ControlOut = {{2'b0_0},{1'b0,`LW},{4'b0000},{1'b0},{2'b11},{`NOBRANCH},{`ADD},{3'b0_00},{`RTYPE}};
                        7'b0100000: /* SUB */ `ControlOut = {{2'b0_0},{1'b0,`LW},{4'b0000},{1'b0},{2'b11},{`NOBRANCH},{`SUB},{3'b0_00},{`RTYPE}};
                    endcase
                3'b001: /* SLL */ `ControlOut = {{2'b0_0},{1'b0,`LW},{4'b0000},{1'b0},{2'b11},{`NOBRANCH},{`SLL},{3'b0_00},{`RTYPE}};
                3'b010: /* SLT */ `ControlOut = {{2'b0_0},{1'b0,`LW},{4'b0000},{1'b0},{2'b11},{`NOBRANCH},{`SLT},{3'b0_00},{`RTYPE}};
                3'b011: /* SLTU */ `ControlOut = {{2'b0_0},{1'b0,`LW},{4'b0000},{1'b0},{2'b11},{`NOBRANCH},{`SLTU},{3'b0_00},{`RTYPE}};
                3'b100: /* XOR */ `ControlOut = {{2'b0_0},{1'b0,`LW},{4'b0000},{1'b0},{2'b11},{`NOBRANCH},{`XOR},{3'b0_00},{`RTYPE}};
                3'b101:
                    case(Fn7)
                        7'b0000000: /* SRL */ `ControlOut = {{2'b0_0},{1'b0,`LW},{4'b0000},{1'b0},{2'b11},{`NOBRANCH},{`SRL},{3'b0_00},{`RTYPE}};
                        7'b0100000: /* SRA */ `ControlOut = {{2'b0_0},{1'b0,`LW},{4'b0000},{1'b0},{2'b11},{`NOBRANCH},{`SRA},{3'b0_00},{`RTYPE}};
                    endcase
                3'b110: /* OR */ `ControlOut = {{2'b0_0},{1'b0,`LW},{4'b0000},{1'b0},{2'b11},{`NOBRANCH},{`OR},{3'b0_00},{`RTYPE}};
                3'b111: /* AND */ `ControlOut = {{2'b0_0},{1'b0,`LW},{4'b0000},{1'b0},{2'b11},{`NOBRANCH},{`AND},{3'b0_00},{`RTYPE}};
            endcase
        7'b0010011: //REG-IMM
            case(Fn3)
                3'b000: /* ADDI */ `ControlOut = {{2'b0_0},{1'b0,`LW},{4'b0000},{1'b0},{2'b10},{`NOBRANCH},{`ADD},{3'b0_10},{`ITYPE}};
                3'b001: /* SLLI */ `ControlOut = {{2'b0_0},{1'b0,`LW},{4'b0000},{1'b0},{2'b10},{`NOBRANCH},{`SLL},{3'b0_01},{`ITYPE}};
                3'b010: /* SLTI */ `ControlOut = {{2'b0_0},{1'b0,`LW},{4'b0000},{1'b0},{2'b10},{`NOBRANCH},{`SLT},{3'b0_10},{`ITYPE}};
                3'b011: /* SLTIU */ `ControlOut = {{2'b0_0},{1'b0,`LW},{4'b0000},{1'b0},{2'b10},{`NOBRANCH},{`SLTU},{3'b0_10},{`ITYPE}};
                3'b100: /* XORI */ `ControlOut = {{2'b0_0},{1'b0,`LW},{4'b0000},{1'b0},{2'b10},{`NOBRANCH},{`XOR},{3'b0_10},{`ITYPE}};
                3'b101:
                    case(Fn7)
                        7'b0000000: /* SRLI */ `ControlOut = {{2'b0_0},{1'b0,`LW},{4'b0000},{1'b0},{2'b10},{`NOBRANCH},{`SRL},{3'b0_01},{`ITYPE}};
                        7'b0100000: /* SRAI */ `ControlOut = {{2'b0_0},{1'b0,`LW},{4'b0000},{1'b0},{2'b10},{`NOBRANCH},{`SRA},{3'b0_01},{`ITYPE}};
                    endcase
                3'b110: /* ORI */ `ControlOut = {{2'b0_0},{1'b0,`LW},{4'b0000},{1'b0},{2'b10},{`NOBRANCH},{`OR},{3'b0_10},{`ITYPE}};
                3'b111: /* ANDI */ `ControlOut = {{2'b0_0},{1'b0,`LW},{4'b0000},{1'b0},{2'b10},{`NOBRANCH},{`AND},{3'b0_10},{`ITYPE}};
            endcase
        7'b0110111: /* LUI */ `ControlOut = {{2'b0_0},{1'b0,`LW},{4'b0000},{1'b0},{2'b00},{`NOBRANCH},{`LUI},{3'b0_10},{`UTYPE}};
        7'b0010111: /* AUIPC */ `ControlOut = {{2'b0_0},{1'b0,`LW},{4'b0000},{1'b0},{2'b00},{`NOBRANCH},{`ADD},{3'b1_10},{`UTYPE}};
        7'b1100011: //Branch
            case(Fn3)
                3'b000: /* BEQ */ `ControlOut = {{2'b0_0},{1'b0,`NOREGWRITE},{4'b0000},{1'b0},{2'b11},{`BEQ},{4'bxxxx},{3'b0_00},{`BTYPE}};
                3'b001: /* BNE */ `ControlOut = {{2'b0_0},{1'b0,`NOREGWRITE},{4'b0000},{1'b0},{2'b11},{`BNE},{4'bxxxx},{3'b0_00},{`BTYPE}};
                3'b100: /* BLT */ `ControlOut = {{2'b0_0},{1'b0,`NOREGWRITE},{4'b0000},{1'b0},{2'b11},{`BLT},{4'bxxxx},{3'b0_00},{`BTYPE}};
                3'b101: /* BGE */ `ControlOut = {{2'b0_0},{1'b0,`NOREGWRITE},{4'b0000},{1'b0},{2'b11},{`BGE},{4'bxxxx},{3'b0_00},{`BTYPE}};
                3'b110: /* BLTU */ `ControlOut = {{2'b0_0},{1'b0,`NOREGWRITE},{4'b0000},{1'b0},{2'b11},{`BLTU},{4'bxxxx},{3'b0_00},{`BTYPE}};
                3'b111: /* BGEU */ `ControlOut = {{2'b0_0},{1'b0,`NOREGWRITE},{4'b0000},{1'b0},{2'b11},{`BGEU},{4'bxxxx},{3'b0_00},{`BTYPE}};
            endcase
        7'b1101111: /* JAL */ `ControlOut = {{2'b1_0},{1'b0,`LW},{4'b0000},{1'b1},{2'b00},{`NOBRANCH},{4'bxxxx},{3'bx_xx},{`JTYPE}};
        7'b1100111: /* JALR */ `ControlOut = {{2'b0_1},{1'b0,`LW},{4'b0000},{1'b1},{2'b10},{`NOBRANCH},{`ADD},{3'b0_10},{`ITYPE}};
        default: `ControlOut = 26'b0;
    endcase
```

3. 简化写法

事实上，不需要进行复杂的 if 判断或者繁多的 case 语句，我们可以根据关键的信号，利用与或非运算来赋值。

首先对所有指令操作码部分进行宏定义，方便讨论

```
`define OP_JAL    7'b1101111 //JAL 的操作码
`define OP_JALR   7'b1100111 //JALR 的操作码
`define OP_Load   7'b0000011 //Load 类指令的操作码
`define OP_Store  7'b0100011 //Store 类指令的操作码
`define OP_Branch 7'b1100011 //Branch 类指令的操作码
`define OP_LUI    7'b0110111 //LUI 的操作码
`define OP_AUIPC  7'b0010111 //AUIPC 的操作码
`define OP_RegReg 7'b0110011 //寄存器-寄存器算术指令的操作码
`define OP_RegImm 7'b0010011 //寄存器-立即数算术指令的操作码
```

根据上面对各个信号具体分析，可以编写对应代码。

（1）JalD

表示 JAL 指令是否跳转，只有当前指令是 JAL 时，为 1，其他指令均为 0。

```
JalD        = Op == `OP_JAL; //JAL 指令才为 1
```

（2）JalrD

表示 JALR 指令是否跳转，为 1 表示跳转，只有当前指令是 JALR 时，为 1，其他指令均置为 0。

```
JalrD       = Op == `OP_JALR; //JALR 指令才为 1
```

（3）RegWriteD

表示寄存器文件的写入模式，5 种常量已定义在 Parameters.v 中

- Load 类指令分别选择对应模式（`LW,`LH,`LB,`LHU,`LBU）。
- Store 类和 Branch 类指令选择`NOREGWRITE 模式。
- 其余指令全部按字节写入寄存器，选择`LW 模式

观察 Load 类指令的格式，区别部分在 Fn3 部分的 3 位

LB	imm[11:0]	rs1	000	rd	0000011
LH	imm[11:0]	rs1	001	rd	0000011
LW	imm[11:0]	rs1	010	rd	0000011
LBU	imm[11:0]	rs1	100	rd	0000011
LHU	imm[11:0]	rs1	101	rd	0000011

为此，我们修改 Parameters.v 对五种模式的定义

```
`define NOREGWRITE  3'b000
`define LB  3'b001
`define LH  3'b010
`define LW  3'b011
`define LBU  3'b101
`define LHU  3'b110
```

则 `Fn3 + 3'b001` 就是对应模式（设置偏移量为 1 是为了将 `3'b000` 保留给 `NOREGWRITE`）

因此该信号生成代码如下

```
RegWriteD    = Op == `OP_Load ? Fn3 + 3'b001 : (Op == `OP_Store || Op == `OP_Branch ? `NOREGWRITE : `LW);
```

（4）MemToRegD

表示写入 WB 段写入寄存器文件的数据来源，为 1 表示来自数据存储器。

- 只有 Load 类指令为 1
- 其他指令都为 0

```
MemToRegD    = Op == `OP_Load; //Load 类指令才为 1
```

（5）MemWriteD

表示数据存储器的写入模式，按独热编码，1bit 对应一字节。

- SB 指令为 4’b0001
- SH 指令为 4’b0011
- SW 指令为 4’b1111
- 其余指令没有写入数据存储器，均置为 4’b0000

（关于写入数据非字节对齐的情况，放在 WB 段寄存器中移位处理。）

观察 Store 类指令格式，区别部分在 Fn3 部分的 3 位

SB	imm[11:5]	rs2	rs1	000	imm[4:0]	0100011
SH	imm[11:5]	rs2	rs1	001	imm[4:0]	0100011
SW	imm[11:5]	rs2	rs1	010	imm[4:0]	0100011

因此，可以针对不同的 Fn3 对 `4'b1111` 右移不同的位数（$4 - 2^{Fn3}$），简洁的表达如下

```
MemWriteD = Op == `OP_Store ? 4'b1111 >> (3'b100 - (3'b001 << Fn3)) : 4'b0000;
```

（6）LoadNpcD

表示写入寄存器文件数据来源，在 MemToRegD == 0 的时候起作用，在 ALU 的计算结果和 PC+4 中进行选择。

- 只有 JAL 和 JALR 需要将该信号为 1
- 其他指令全部为 0。

```
LoadNpcD    = Op == `OP_JAL || Op == `OP_JALR; //JAL/JALR 指令才为 1
```

（7）RegReadD

表示当前指令是否用到了寄存器文件。

RegReadD[1] == 1 表示 A1 端口对应的寄存器值被使用到了。

RegReadD[0] == 1 表示 A2 端口对应的寄存器值被使用到了。

观察不同指令的数据通路，

- 除了 LUI/AUIPC/JAL 这 3 条指令，其他都用到了寄存器 A1 端口
- Store 类/RegReg 类/Branch 类指令用到了寄存器 A2 端口

该信号简洁的写法如下

```
RegReadD[1] = Op != `OP_LUI && Op != `OP_AUIPC && Op != `OP_JAL;
RegReadD[0] = Op == `OP_Store || Op == `OP_RegReg || Op == `OP_Branch;
```

（8）BranchTypeD

表示 Branch 指令的类型，已定义在 Parameters.v 中。

- Branch 类指令分别选择对应跳转模式

- 其余指令全部选择`NOBRANCH

观察 Branch 类指令格式，区别部分在 Fn3 部分的 3 位

BEQ	imm[12|10:5]	rs2	rs1	000	imm[4:1|11]	1100011
BNE	imm[12|10:5]	rs2	rs1	001	imm[4:1|11]	1100011
BLT	imm[12|10:5]	rs2	rs1	100	imm[4:1|11]	1100011
BGE	imm[12|10:5]	rs2	rs1	101	imm[4:1|11]	1100011
BLTU	imm[12|10:5]	rs2	rs1	110	imm[4:1|11]	1100011
BGEU	imm[12|10:5]	rs2	rs1	111	imm[4:1|11]	1100011

为此，我们修改 Parameters.v 对六种模式的定义

```
`define NOBRANCH  3'b000
`define BEQ  3'b110
`define BNE  3'b111
`define BLT  3'b010
`define BGE  3'b011
`define BLTU  3'b100
`define BGEU  3'b101
```

则 `Fn3 - 3'b010` 就是对应模式（设置偏移量为 2 是为了将 `3'b000` 保留给 NOBRANCH ）

因此该信号生成代码如下

```
BranchTypeD = Op == `OP_Branch ? Fn3 - 3'b010 : `NOBRANCH;
```

（9）AluContrlD

表示 ALU 模块的操作类型，对于之前的具体分析，作如下总结

- Reg-Reg 类的算术指令分别选择对应功能
- Reg-Imm 类的算术指令也分别选择对应功能
- LUI 指令选择`LUI
- 其他指令（包括不需要 ALU 的指令）都选择`ADD

因此，该信号的赋值主要是考虑两类算术指令。

对于两类算术指令，观察指令格式，可以发现 Fn3 部分的 3 位决定了使用的 ALU 功能，并且两类指令中使用相同 ALU 功能的，对应的 Fn3 部分也相同。

ADDI	imm[11:0]		rs1	000	rd	0010011
SLLI	0000000	shamt	rs1	001	rd	0010011
SLTI	imm[11:0]		rs1	010	rd	0010011
SLTIU	imm[11:0]		rs1	011	rd	0010011
XORI	imm[11:0]		rs1	100	rd	0010011
SRLI	0000000	shamt	rs1	101	rd	0010011
SRAI	0100000	shamt	rs1	101	rd	0010011
ORI	imm[11:0]		rs1	110	rd	0010011
ANDI	imm[11:0]		rs1	111	rd	0010011

ADD	0000000	rs2	rs1	000	rd	0110011
SUB	0100000	rs2	rs1	000	rd	0110011
SLL	0000000	rs2	rs1	001	rd	0110011
SLT	0000000	rs2	rs1	010	rd	0110011
SLTU	0000000	rs2	rs1	011	rd	0110011
XOR	0000000	rs2	rs1	100	rd	0110011
SRL	0000000	rs2	rs1	101	rd	0110011
SRA	0100000	rs2	rs1	101	rd	0110011
OR	0000000	rs2	rs1	110	rd	0110011
AND	0000000	rs2	rs1	111	rd	0110011

为此，我们修改 Parameters.v 对不同 ALU 模式的定义如下，使之正好对应指令的 Fn3 部分。注意，SUB，SRA 单独考虑，并且使 SUB 相对 ADD，SRA 相对 SRL 的偏移为 4'b1000，方便统一处理。

```
`define ADD  4'b0000
`define SLL  4'b0001
`define SLT  4'b0010
`define SLTU 4'b0011
`define XOR  4'b0100
`define SRL  4'b0101
`define OR   4'b0110
`define AND  4'b0111
`define SUB  4'b1000
`define SRA  4'b1101
`define LUI  4'b1010
```

修改后，Fn3 的高位补 0 变成 4 位就是 AluContrlD 的值，针对 Fn7 部分为 7'b0100000 的只需要再加上偏移 4'b1000 就行了。

因此 AluContrlD 信号生成代码如下，

```
        case(Op) //AluContrlD 信号生成，Parameters 中直接按指令 Fn3 对应的编码进行定义
            `OP_RegReg: AluContrlD = {1'b0 ,Fn3} + (Fn7 == 7'b0100000 ? 4'b1000 :
4'b0000);//后面附加一项 4'b1000 偏移针对 SUB 和 SRA 指令
            `OP_RegImm: AluContrlD = {1'b0 ,Fn3} + (Fn7 == 7'b0100000 && Fn3 ==
3'b101 ? 4'b1000 : 4'b0000);//后面附加一项 4'b1000 偏移针对 SRAI 指令
            `OP_LUI:    AluContrlD = `LUI;
            default:    AluContrlD = `ADD;
        endcase
```

（10）AluSrc1D

表示 ALU 模块第一个操作数的来源。

- AUIPC 指令需要将立即数加到 PC 上，因此置为 1，表示选择 PC。
- JAL 指令不需要 ALU，可以置为 x，也可以置为 0
- 其余指令全部置为 0，表示来自 rs1 寄存器。

该信号生成代码如下

```
        AluSrc1D     =  Op == `OP_AUIPC; //AUIPC 指令的 src1 才为 1
```

（11）AluSrc2D

表示 ALU 模块第二个操作数的来源。

- R 类型的指令全部置为 2'b00，表示来自 rs2 寄存器。
- B 类型的指令全部置为 2'b00，表示来自 rs2 寄存器。这里应该注意，虽然 Branch 类指令没有用到 ALU，但 BranchDecisionMaking 模块的输入就是 ALU 的两个输入操作数，因此不能置为 2'bxx
- S 类型的指令全部置为 2'b10，表示来自立即数。
- U 类型的指令全部置为 2'b10，表示来自立即数。
- J 类型的指令只有 JAL，不需要 ALU，可以置为 2'bxx
- I 类型的指令除了 SLLI,SRLI,SRAI 全部置为 2'b10，表示来自立即数。
- SLLI,SRLI,SRAI 这三条指令比较特殊，置为 2'b01。虽然需要立即

数，但不需要拓展，直接来自 Shamt 字段。

这个信号有 2 位宽，我们对 2 位分别赋值可以避免上面繁琐的判断

观察不同指令数据通路，有

- Reg-Reg 类/Branch 类/SLLI/SRLI/SRAI 的 src2 为 01 或 00，即高位为 0
- 只有 SLLI/SRLI/SRAI 三条指令 src2 为 2’b01，即低位为 1

因此 src2 信号简洁生成如下

```
        AluSrc2D[1] = Op != `OP_RegReg && Op != `OP_Branch && ~(Op == `OP_RegImm && (Fn3 == 3'b001 || Fn3 == 3'b101)); //RegReg 类/Branch 类/SLLI/SRLI/SRAI 的 src2 高位为 0
        AluSrc2D[0] = Op == `OP_RegImm && (Fn3 == 3'b001 || Fn3 == 3'b101); //SLLI/SRLI/SRAI 三条指令 src2 为 2’b01，低位为 1
```

（12）ImmType

表示当前指令立即数的类型（事实上 RTYPE 没有立即数），已定义在 Parameters.v 中，作为 ImmOperandUnit 的输入。

这个信号没有特别的方法，枚举如下

```
        case(Op)
            `OP_Load:   ImmType = `ITYPE;
            `OP_Store:  ImmType = `STYPE;
            `OP_RegReg: ImmType = `RTYPE;
            `OP_RegImm: ImmType = `ITYPE;
            `OP_LUI:    ImmType = `UTYPE;
            `OP_AUIPC:  ImmType = `UTYPE;
            `OP_Branch: ImmType = `BTYPE;
            `OP_JAL:    ImmType = `JTYPE;
            `OP_JALR:   ImmType = `ITYPE;
            default:    ImmType = `RTYPE;
        endcase
```

最终，得到 ControlUnit 的代码如下

```
    always@(*) begin
```

```
        JalD        = Op == `OP_JAL; //JAL 指令才为 1
        JalrD       = Op == `OP_JALR; //JALR 指令才为 1
        MemToRegD   = Op == `OP_Load; //Load 类指令才为 1
        RegWriteD   = Op == `OP_Load ? Fn3 + 3'b001 : (Op == `OP_Store || Op ==
`OP_Branch ? `NOREGWRITE : `LW);
//Load 类指令按 Fn3 选择（Parameters 中直接按 Fn3 编码定义，设置偏移量为 1 是为了将 3'b000 保留
给 NOREGWRITE），Store 和 Branch 类指令为 NOREGWRITE，其余为 LW
        MemWriteD   = Op == `OP_Store ? 4'b1111 >> (3'b100 - (3'b001 << Fn3)) :
4'b0000;
//3 种 Store 指令分别对 4'b1111 右移 4 - pow(2, Fn3) 位
        LoadNpcD    = Op == `OP_JAL || Op == `OP_JALR; //JAL/JALR 指令才为 1
        RegReadD[1] = Op != `OP_LUI && Op != `OP_AUIPC && Op != `OP_JAL;
//除了 LUI/AUIPC/JAL 这 3 条指令，其他都用到了寄存器 A1 端口
        RegReadD[0] = Op == `OP_Store || Op == `OP_RegReg || Op == `OP_Branch;
//Store 类/RegReg 类/Branch 类指令用到了寄存器 A2 端口
        BranchTypeD = Op == `OP_Branch ? Fn3 - 3'b010 : `NOBRANCH;
//Branch 类指令按 Fn3 选择（Parameters 中直接按 Fn3 编码定义，偏移 3'b010，为了将 000 留给
NOBRANCH），其余都是 NOBRANCH
        AluSrc1D    = Op == `OP_AUIPC; //AUIPC 指令的 src1 才为 1
        AluSrc2D[1] = Op != `OP_RegReg && Op != `OP_Branch && ~(Op == `OP_RegImm
&& (Fn3 == 3'b001 || Fn3 == 3'b101));
//RegReg 类/Branch 类/SLLI/SRLI/SRAI 的 src2 高位为 0
        AluSrc2D[0] = Op == `OP_RegImm && (Fn3 == 3'b001 || Fn3 == 3'b101);
//SLLI/SRLI/SRAI 三条指令 src2 为 2'b01，低位为 1
        case(Op)
            `OP_Load:   ImmType = `ITYPE;
            `OP_Store:  ImmType = `STYPE;
            `OP_RegReg: ImmType = `RTYPE;
            `OP_RegImm: ImmType = `ITYPE;
            `OP_LUI:    ImmType = `UTYPE;
            `OP_AUIPC:  ImmType = `UTYPE;
            `OP_Branch: ImmType = `BTYPE;
            `OP_JAL:    ImmType = `JTYPE;
            `OP_JALR:   ImmType = `ITYPE;
           default:     ImmType = `RTYPE;
        endcase
        case(Op) //AluContrlD 信号生成，Parameters 中直接按指令 Fn3 对应的编码进行定义
            `OP_RegReg: AluContrlD = {1'b0 ,Fn3} + (Fn7 == 7'b0100000 ? 4'b1000 :
4'b0000);//后面附加一项 4'b1000 偏移针对 SUB 和 SRA 指令
            `OP_RegImm: AluContrlD = {1'b0 ,Fn3} + (Fn7 == 7'b0100000 && Fn3 ==
3'b101 ? 4'b1000 : 4'b0000);//后面附加一项 4'b1000 偏移针对 SRAI 指令
            `OP_LUI:    AluContrlD = `LUI;
           default:     AluContrlD = `ADD;
        endcase
    end
endmodule
```

HarzardUnit.v			
输入/输出	宽度	信号名	说明
input	[0:0]	CpuRst	CPU 的复位信号
input	[0:0]	ICacheMiss	用来处理 cache miss
input	[0:0]	DCacheMiss	用来处理 cache miss
input	[4:0]	Rs1D	ID 段指令的 rs1 寄存器地址
input	[4:0]	Rs2D	ID 段指令的 rs2 寄存器地址
input	[4:0]	Rs1E	EX 段指令的 rs1 寄存器地址
input	[4:0]	Rs2E	EX 段指令的 rs2 寄存器地址
input	[4:0]	RdE	EX 段指令的 rd 寄存器地址
input	[4:0]	RdM	MEM 段指令的 rd 寄存器地址
input	[4:0]	RdW	WB 段指令的 rd 寄存器地址
input	[1:0]	RegReadE	表示 EX 段指令是否用到了寄存器的结果
input	[0:0]	MemToRegE	EX 段指令将要写回寄存器的数据是否来自存储器
input	[2:0]	RegWriteM	MEM 段指令的寄存器写回模式
input	[2:0]	RegWriteW	WB 段指令的寄存器写回模式
output	[0:0]	StallF	IF 段寄存器是否停顿
output	[0:0]	StallD	ID 段寄存器是否停顿
output	[0:0]	StallE	EX 段寄存器是否停顿
output	[0:0]	StallM	MEM 段寄存器是否停顿
output	[0:0]	StallW	WB 段寄存器是否停顿
output	[0:0]	FlushF	IF 段寄存器是否清空
output	[0:0]	FlushD	ID 段寄存器是否清空
output	[0:0]	FlushE	EX 段寄存器是否清空
output	[0:0]	FlushM	MEM 段寄存器是否清空
output	[0:0]	FlushW	WB 段寄存器是否清空
output	[1:0]	Forward1E	Rs1 转发选择信号
output	[1:0]	Forward2E	Rs2 转发选择信号

这个模块处理转发与冒险。

1.数据相关

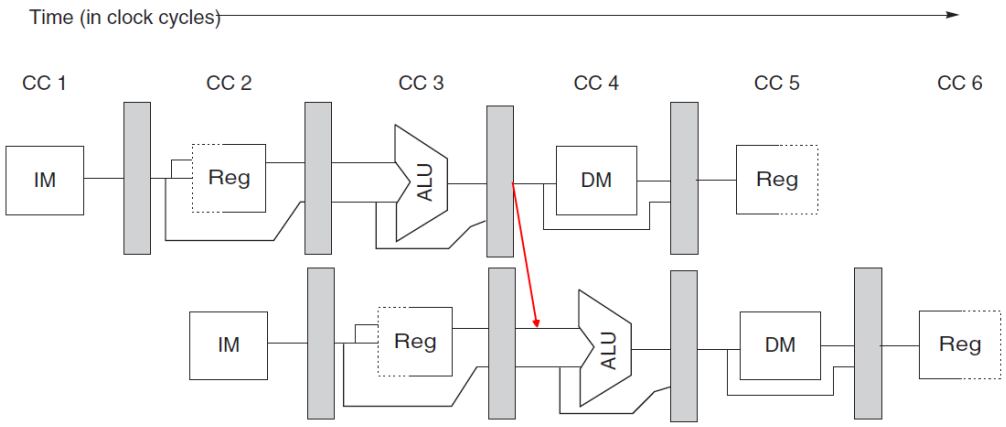
由于某些指令会对**0 号寄存器**进行写入（比如不需要 JAL 的链接结果时，可以选择 0 号寄存器作为目标寄存器），但是实际是不会进行写入的。这时如果下一条指令用到了 0 号寄存器作为常量，转发会导致当前指令写入的无用数据被转发给下一条指令。因此，再默认情况下或者要使用的寄存器是 0 号寄存器，Forward 为 2'b00，表示不使用转发。

下面以第一个操作数 Rs1 为例进行分析，另一个同理。

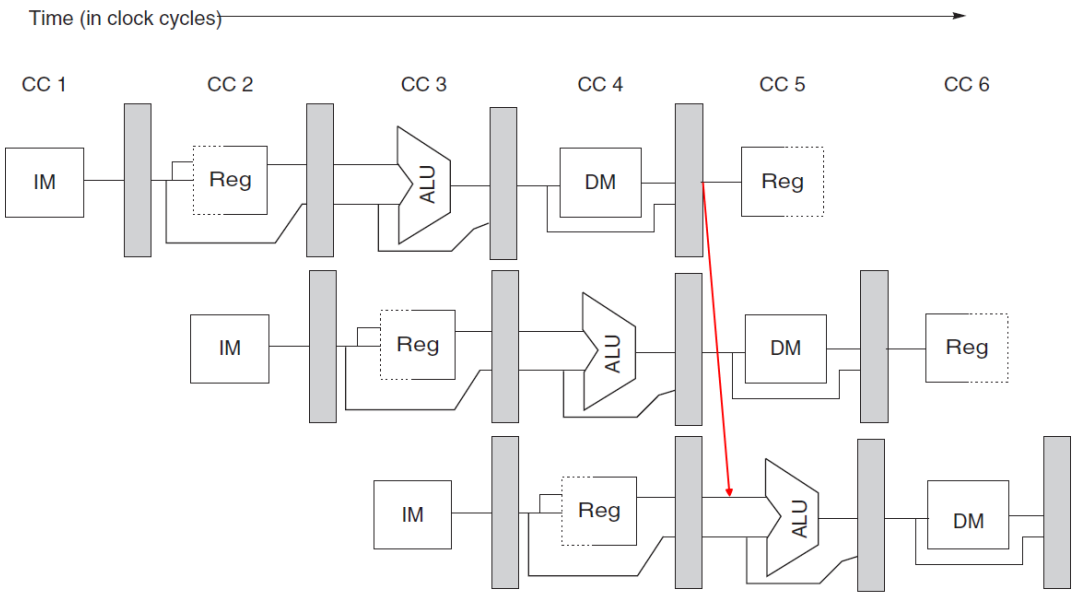
（1）数据相关之转发

考虑当前指令在 EX 阶段。

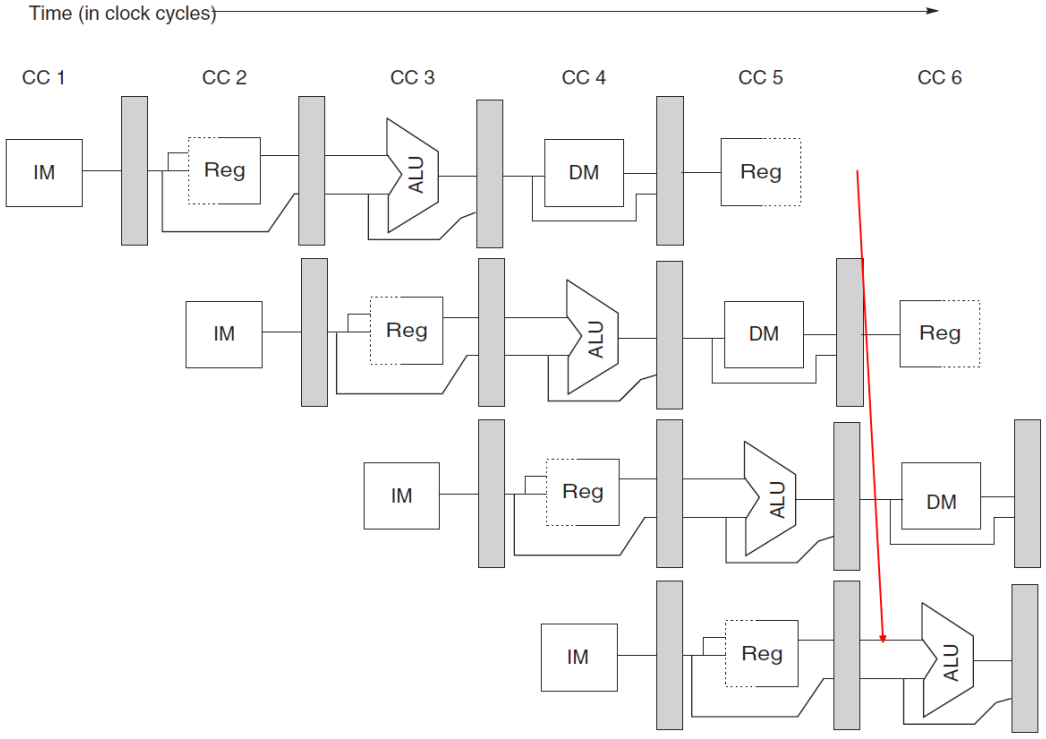
- 如果 RegWriteM 不为 0，说明上一条指令（此时在 MEM 阶段）的 ALU 结果要写回寄存器，并且如果当前 EX 阶段要读取寄存器与上一条指令要写回的寄存器是同一个(RdM == Rs1E)，那么需要进行转发，置 forward 为 2'b10——情况 1，**来自 MEM 段**的转发



- 如果 RegWriteW 不为 0，说明上上一条指令（此时在 WB 阶段）的访存结果要写回寄存器，并且如果当前 EX 阶段要读取寄存器与上上一条指令要写回的寄存器是同一个(RdW == Rs1E)，那么需要进行转发，置 forward 为 2'b01——情况 2，**来自 WB 段**的转发



- 另外，我们不需要考虑上上上条指令转发这种情况，因为我们的寄存器文件是异步读取的，不需要在下一个周期才能得到当前周期写入的值。



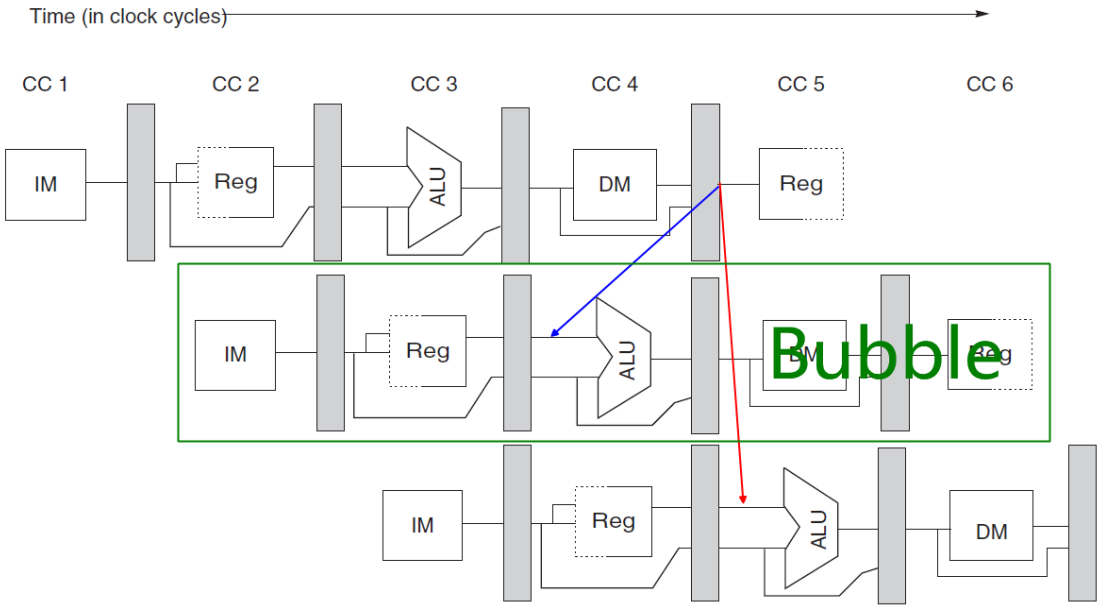
（2）数据相关之停顿

考虑当前指令在 ID 阶段。

- 如果上一条指令访存并写回 且 当前指令 ID 阶段读的是同一个寄

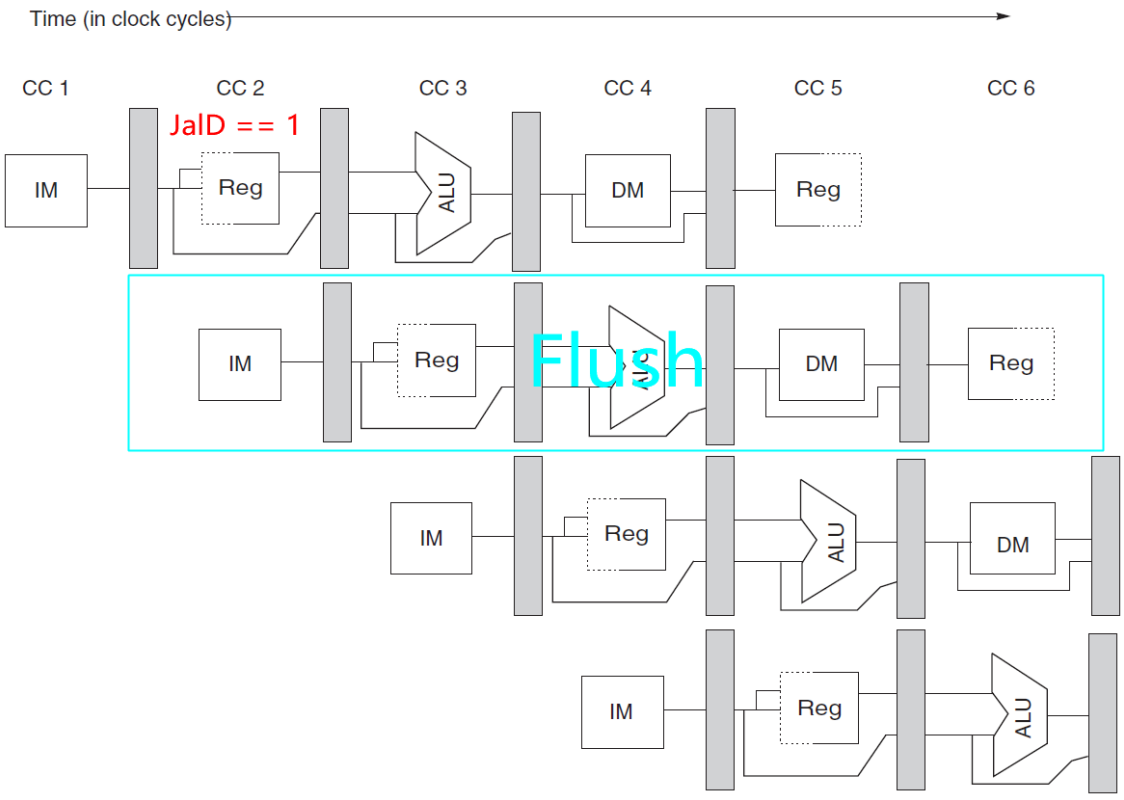
存器，则停顿（插入 bubble）。——情况 3

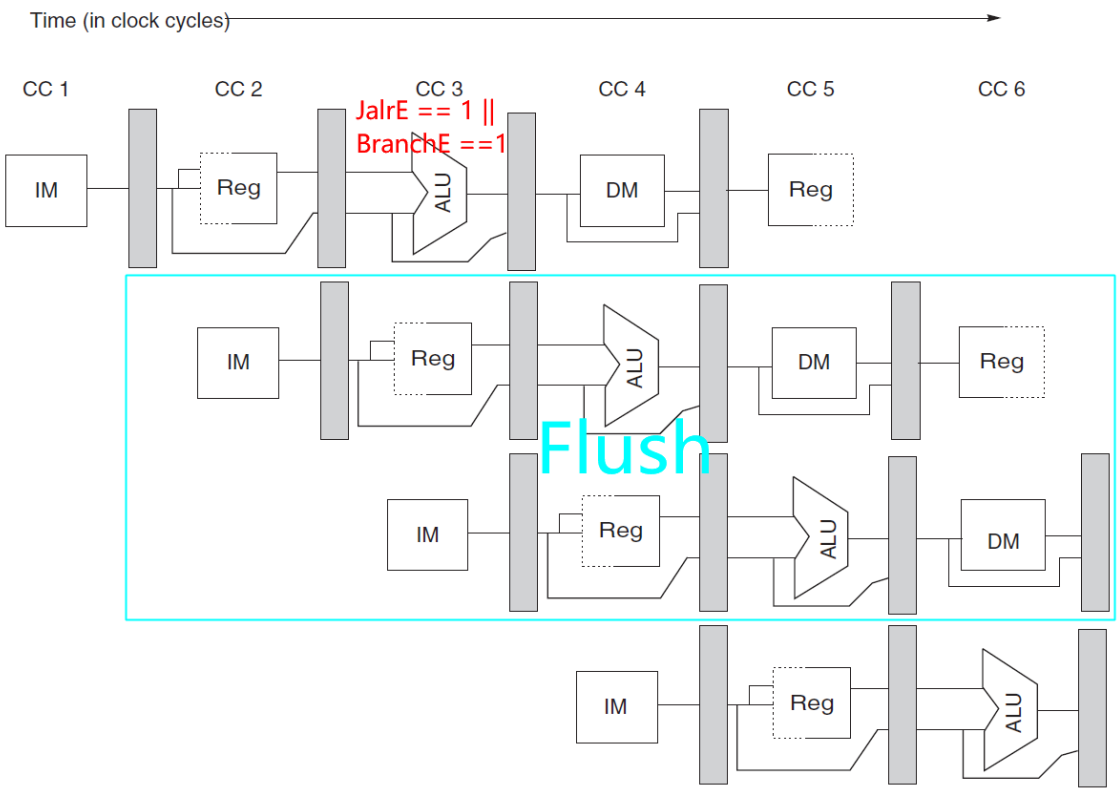
这里并不像 Forward 那样判断 RegWriteE 非 0，因为写入寄存器的数据可能是 ALU 的结果也可能是访存的结果。只有上一条指令写回寄存器的结果是**访存**的结果，才需要停顿，因此用 MemToRegE 判断。如果上一条指令写回寄存器的结果是 ALU 的结果，那么这就等价于情况 1，等到下一个周期即当前指令在 EX 阶段时，会用 Forward 处理。



2. 控制相关

由于采用默认不跳转的策略，遇到确认跳转的指令时，需要**清空**已经读取的后续指令。对于 Branch 指令和 JALR 指令，需要清空 ID 和 EX 段寄存器，对于 JAL 指令，需要清空 ID 段寄存器。


Time (in clock cycles)
CC 1
CC 2
CC 3
CC 4
CC 5
CC 6
JalD == 1
IM
Reg
ALU
DM
Reg
Flush


Time (in clock cycles)
CC 1
CC 2
CC 3
CC 4
CC 5
CC 6
JalrE == 1 ||
BranchE ==1
IM
Reg
ALU
DM
Reg
Flush

具体代码的编写：

在编写代码的时候，和控制模块同样地，不需要进行复杂的 if 判断或者繁多的 case 语句，我们可以根据关键的信号，利用与或非运算来生成转发信号。

以 Forward1E 为例，只有 3 种取值，2’b00、2’b01、2’b10，我们可以按位赋值。

对于高位，只有 10 时需要赋值 1，而 10 的情况对应 MEM 段转发，因此只需要写`|RegWriteM && RegReadE[1] && (RdM == Rs1E)`，表示 MEM 段的 RegWrite 非 0，且当前指令用到了寄存器 A1 端口，并且 MEM 段指令写回寄存器的地址和当前指令要用的寄存器地址相等。当然，需要加上 `RdM != 0` 表示不是 0 号寄存器。

对于低位，只有 01 时需要赋值 1，而 01 的情况对应 WB 段转发，因此需要写`|RegWriteW && RegReadE[1] && (RdW == Rs1E)`，表示 WB 段的 RegWrite 非 0，且当前指令用到了寄存器 A1 端口，并且 WB 段指令写回寄存器的地址和当前指令要用的寄存器地址相等。当然，需要加上 `RdW != 0` 表示不是 0 号寄存器。

这样写还有一点问题，上述两种判断可能**同时**成立，导致 Forward1E = 2’b11，显然，我们不能同时进行 **MEM 段转发**（来自上条指令）和 **WB 段转发**（来自上上条指令），在它们同时成立的时候，我们应该选择 MEM 段转发，因为我们要选择最新的数据进行转发，而上条指令写入的数据比上上条指令的更新。因此对于低位的赋值还要加上 WB 段转发不成立的判断，把同时成立的情况转化为 Forward1E = 2’b10

对应 Stall 和 Flush 信号，进行类似的分析，可以写出简洁的代码。

具体代码如下：

```
module HarzardUnit(
    input wire CpuRst, ICacheMiss, DCacheMiss,
```

```
    input wire BranchE, JalrE, JalD,
    input wire [4:0] Rs1D, Rs2D, Rs1E, Rs2E, RdE, RdM, RdW,
    input wire [1:0] RegReadE,
    input wire MemToRegE,
    input wire [2:0] RegWriteM, RegWriteW,
    output reg StallF, FlushF, StallD, FlushD, StallE, FlushE, StallM, FlushM, StallW,
FlushW,
    output reg [1:0] Forward1E, Forward2E
    );
    //Stall and Flush signals generate
    always@(*) begin
        FlushF <= CpuRst;//IF 寄存器（PC 寄存器）只有初始化时需要清空
        FlushD <= CpuRst || (BranchE || JalrE || JalD);//ID 寄存器（处于 IF/ID 之间的寄存器）
        FlushE <= CpuRst || (MemToRegE && (RdE == Rs1D || RdE == Rs2D)) || (BranchE ||
JalrE);
        FlushM <= CpuRst;
        FlushW <= CpuRst;
        StallF <= ~CpuRst && (MemToRegE && (RdE == Rs1D || RdE == Rs2D));
        StallD <= ~CpuRst && (MemToRegE && (RdE == Rs1D || RdE == Rs2D));
        StallE <= 1'b0;
        StallM <= 1'b0;
        StallW <= 1'b0;
    end
    always@(*) begin
        //Forward Register Source 1
        Forward1E[0] <= RdW != 0 && |RegWriteW && RegReadE[1] && (RdW == Rs1E) &&
~(|RegWriteM && RegReadE[1] && (RdM == Rs1E));//如果上上条指令写回位置是 Rs1E，上条指令也是，则应
该取上条指令写的值
        Forward1E[1] <= RdM != 0 && |RegWriteM && RegReadE[1] && (RdM == Rs1E);
        //Forward Register Source 2
        Forward2E[0] <= RdW != 0 && |RegWriteW && RegReadE[0] && (RdW == Rs2E) &&
~(|RegWriteM && RegReadE[0] && (RdM == Rs2E));//如果上上条指令写回位置是 Rs2E，上条指令也是，则应
该取上条指令写的值
        Forward2E[1] <= RdM != 0 && |RegWriteM && RegReadE[0] && (RdM == Rs2E);
    end
endmodule
```

实验结果

进行仿真测试

1. 3 个小文件

（1）Number2Ascii

实现数字转 10 进制 ascii 码

```
.org 0x0
    .global _start
_start:

    ori   a0, zero, 1395            # a0 = 1395
    add   a0, a0  , a0              # a0 = 2790
    add   a0, a0  , a0              # a0 = 5580
    jal   ra, Number2DecimalAscii   # 调用函数Number2DecimalAscii计算 a0 的十进制ASCII，结果应该是 0x30383535，存在 a0 里

    lui   t0, 0x00000               # t0 = 0x00000000
    sw    a0, 0(t0)                 # a0写入(t0)
infinity_loop:
    jal   zero, infinity_loop       # 死循环


Number2DecimalAscii:
    # 函数:Number2DecimalAscii: 计算a0里低13位二进制数对应的十进制的ASCII码，存放在a0里
```

结果符合预期

i	Addr	Addr	Data	Data
0	00000000	0	30383535	808990005
1	00000004	4	30383535	808990005
2	00000008	8	00a50533	10814771
3	0000000c	12	00a50533	10814771
4	00000010	16	010000ef	16777455
5	00000014	20	000002b7	695
6	00000018	24	00a2a023	10657827
7	0000001c	28	0000006f	111
8	00000020	32	01fff2b7	33551031
9	00000024	36	00c2d293	12767891
10	00000028	40	005572b3	5599923
11	0000002c	44	30303537	808465719
12	00000030	48	03056513	50685203
13	00000034	52	3e806313	1048601363
14	00000038	56	0062e863	6482019
15	0000003c	60	c1828293	3246555795
16	00000040	64	00150513	1377555
17	00000044	68	ff5ff06f	4284477551
18	00000048	72	06406313	104882963
19	0000004c	76	0062e863	6482019
20	00000050	80	f9c28293	4190274195
21	00000054	84	10050513	268764435
22	00000058	88	ff5ff06f	4284477551
23	0000005c	92	000103b7	66487

（2）Fibonacci

实现斐波那契数列计算

```
.org 0x0
    .global _start
_start:
    lui   a0, 0x00000   # 设置DataRam的起始地址为0x00000000，也用作被排序数组的起始地址是，即DataRam的起始地址
    addi  sp, a0, 0x400 # 为栈分配0x400Byte的空间

    or    t0, zero, 8   # t0 = 8
    jal   ra, Fibonacci # 计算 fib(8)，正确结果= 34 = 0x22
    sw    t1, (a0)      # 计算结果放在DataRam的首地址

infinity_loop:
    jal    zero, infinity_loop # 排序结束，死循环


Fibonacci:              # 递归计算斐波那契数列的第n项，
                        # n放在t0寄存器中
                        # 结果放在t1寄存器中
                        # 使用ra作为返回地址，并使用堆栈，堆栈指针为sp
```

结果符合预期

i	Addr	Addr	Data	Data
0	00000000	0	00000022	34
1	00000004	4	00000022	34
2	00000008	8	40050113	1074069779
3	0000000c	12	00806293	8413843
4	00000010	16	00c000ef	12583151
5	00000014	20	00652023	6627363
6	00000018	24	0000006f	111
7	0000001c	28	00306e93	3174035
8	00000020	32	01d2f663	30602851
9	00000024	36	0002e313	189203
10	00000028	40	00008067	32871
11	0000002c	44	fff28293	4294083219
12	00000030	48	ffc10113	4290838803
13	00000034	52	00112023	1122339
14	00000038	56	ffc10113	4290838803
15	0000003c	60	00512023	5316643
16	00000040	64	fddff0ef	4259311855
17	00000044	68	00012283	74371
18	00000048	72	fff28293	4294083219
19	0000004c	76	00612023	6365219
20	00000050	80	fcdff0ef	4242534639
21	00000054	84	00012383	74627
22	00000058	88	00410113	4260115
23	0000005c	92	00730333	7537459

（3）QuickSort

实现快速排序

```
.org 0x0
    .global _start
_start:

main:                           # main函数开始，在DataRam里初始化一段数据，然后调用QuickSort进行排序，排序后进入死循环。
    lui     a0, 0x00000         # 设置DataRam的起始地址为0x00000000，也用作被排序数组的起始地址，即DataRam的起始地址
    addi    sp, a0  , 0x400     # 设置栈顶指针

    or      a2, a0, zero

    addi    t0, zero, -3        # 用一系列指令向a0里写入被排序的数组，可以是负数
    sw      t0,  (a2)
    addi    a2, a2, 4
    addi    t0, zero, -7
    sw      t0,  (a2)
    addi    a2, a2, 4
    addi    t0, zero, 6
    sw      t0,  (a2)
    addi    a2, a2, 4
    addi    t0, zero, 5
    sw      t0,  (a2)
    addi    a2, a2, 4
```

结果符合预期。前面的 10 个数据为负数，因此排在 0 前面。

i	Addr	Addr	Data	Data
0	00000000	0	fffffff7	4294967287
1	00000004	4	fffffff7	4294967287
2	00000008	8	fffffff8	4294967288
3	0000000c	12	fffffff9	4294967289
4	00000010	16	fffffffa	4294967290
5	00000014	20	fffffffb	4294967291
6	00000018	24	fffffffc	4294967292
7	0000001c	28	fffffffd	4294967293
8	00000020	32	fffffffe	4294967294
9	00000024	36	ffffffff	4294967295
10	00000028	40	00000000	0
11	0000002c	44	00000001	1
12	00000030	48	00000002	2
13	00000034	52	00000003	3
14	00000038	56	00000004	4
15	0000003c	60	00000005	5
16	00000040	64	00000006	6
17	00000044	68	00000007	7
18	00000048	72	00000008	8
19	0000004c	76	00000009	9
20	00000050	80	00562023	5644323
21	00000054	84	00460613	4589075
22	00000058	88	ff700293	4285530771
23	0000005c	92	00562023	5644323

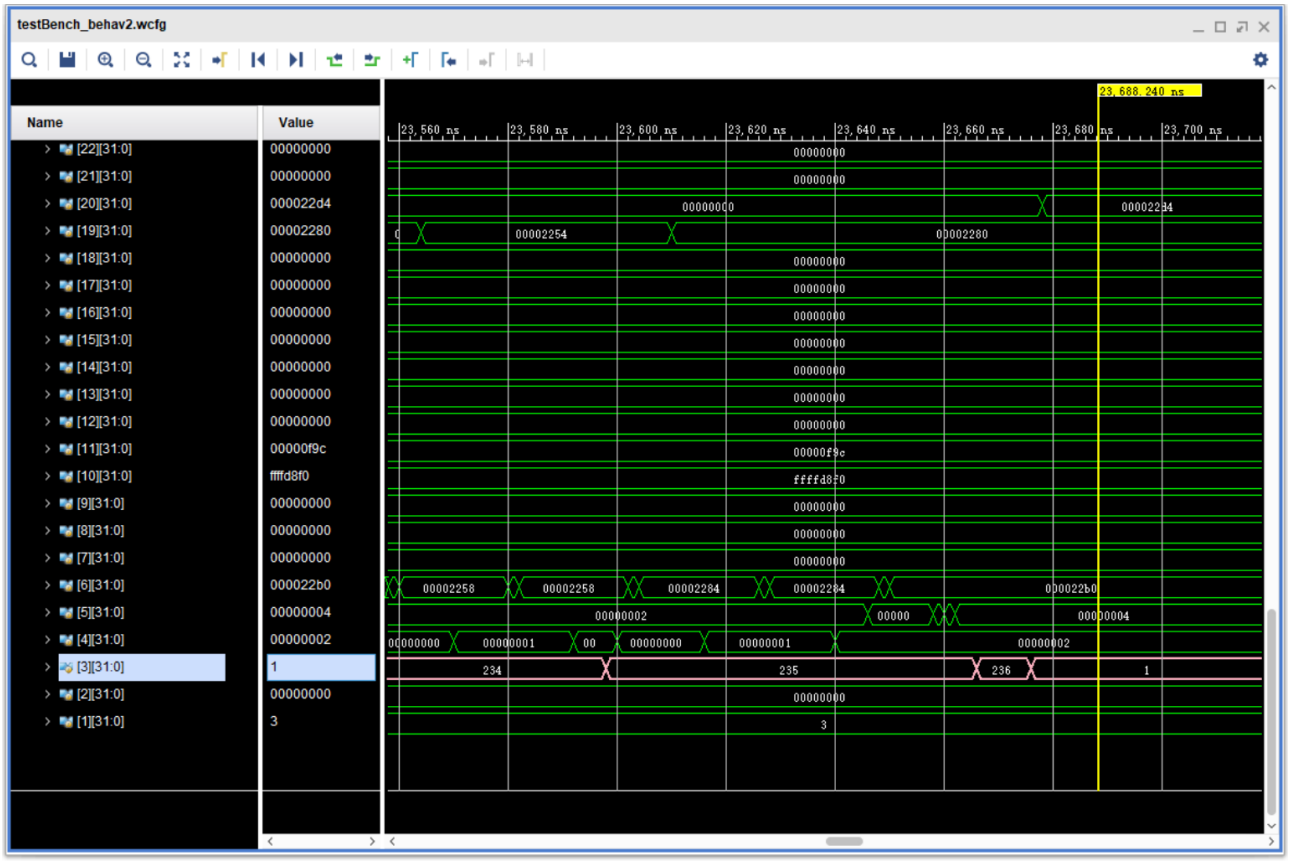
2. 3 个 testAll 文件。

这 3 个文件对所有指令进行了全面的测试。每个测试进行编号，编号存在 3 号寄存器中。

如果某个测试没有通过，就会跳转到一个死循环，3 号寄存器保留了最后出错的测试编号，方便排查。如果通过了所有测试，最后 3 号寄存器的值为 1。

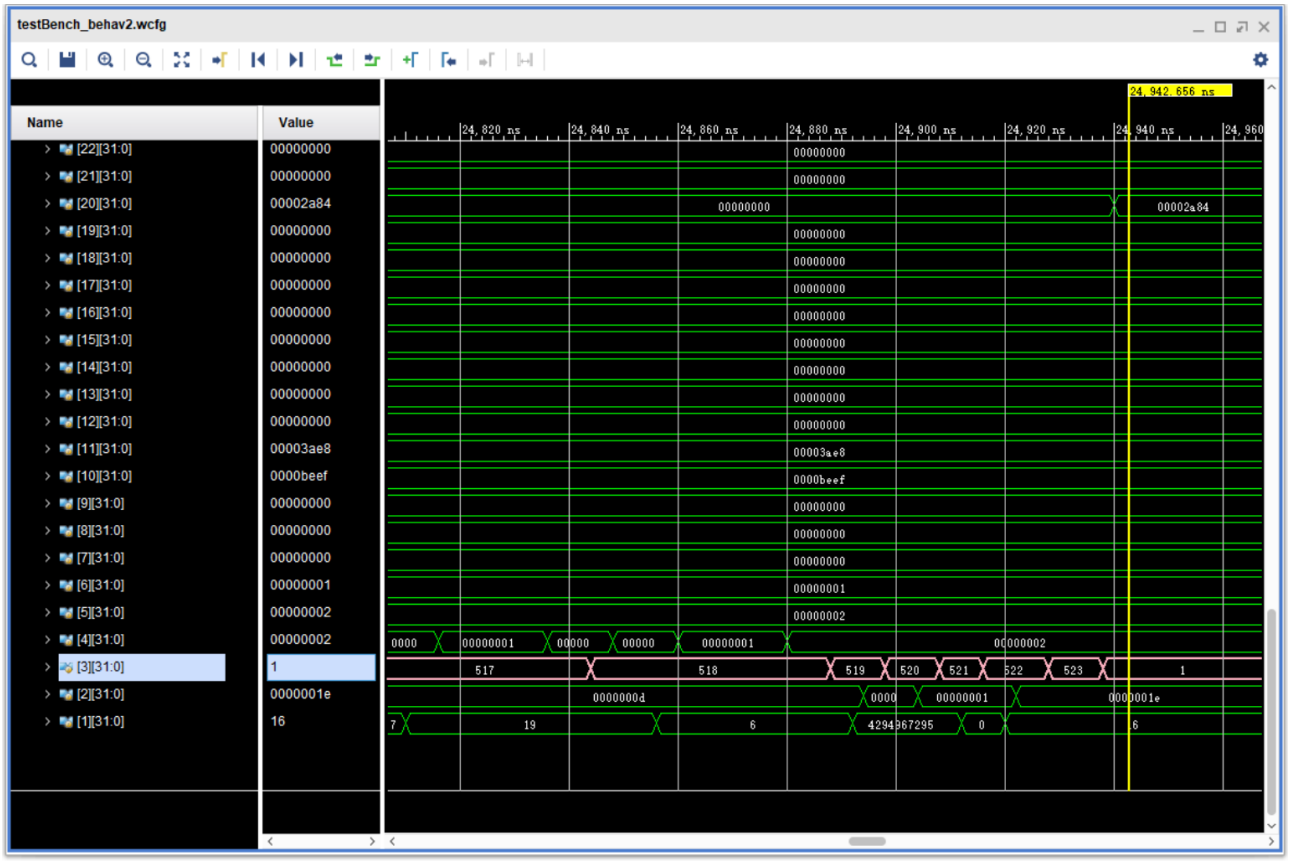
（1）1testAll.inst

仿真结果，236 号测试后 3 号寄存器的值变为 1，且不再改变。



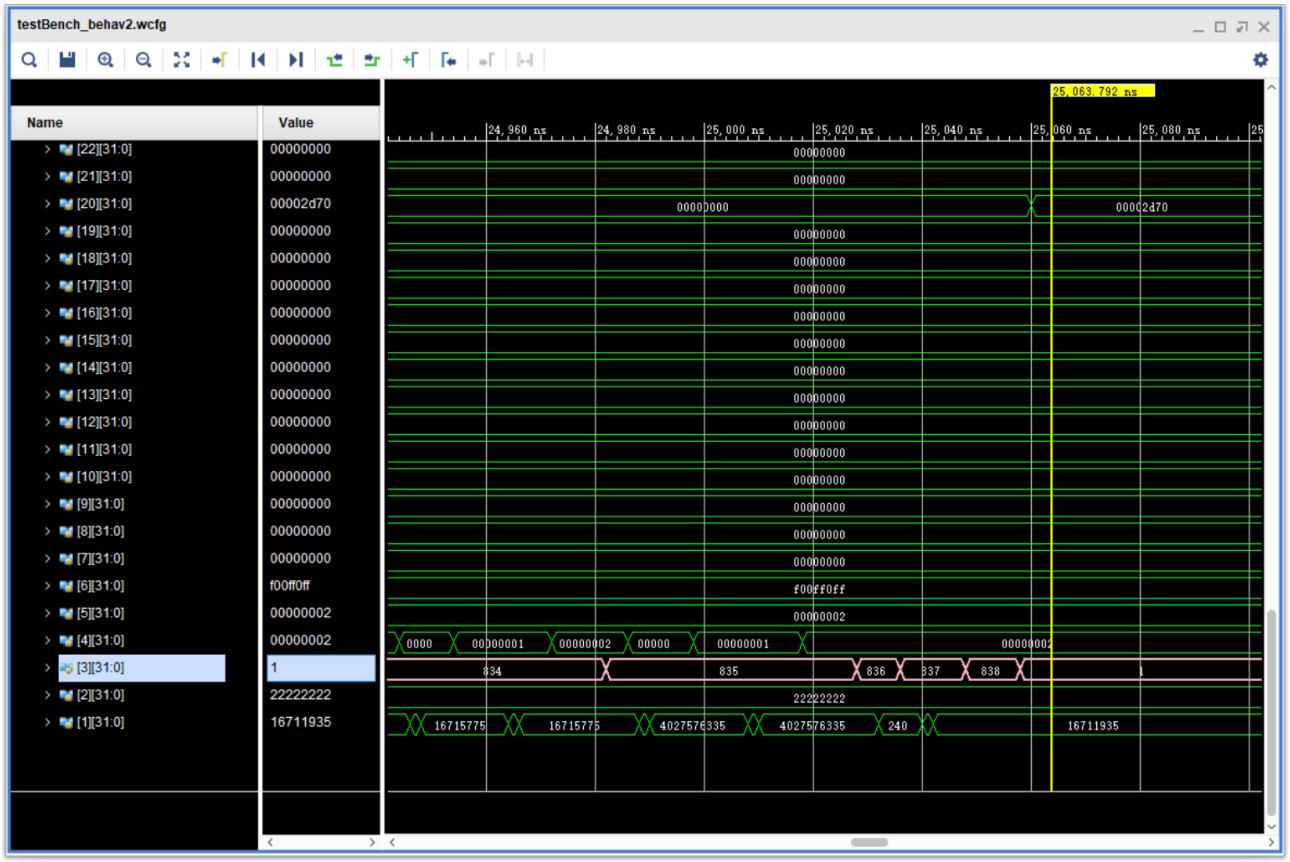
（2）2testAll.inst

仿真结果，523 号测试后 3 号寄存器的值变为 1，且不再改变。



（3）3testAll.inst

仿真结果，838 号测试后 3 号寄存器的值变为 1，且不再改变。



实验总结

1. 花费时间约一周。先是复习了 Verilog 相关的知识，然后用一天时间完成代码的初步编写，随后 4-5 天都在 debug，由于助教已经帮我们完成了整体框架，因此并没有出现特别奇怪的问题，大多是自己写代码时的考虑不周和疏忽造成的。通过 3 个 testAll 测试后，又用了一天时间对 ControlUnit 的代码进行重构、精简。
2. 编写过程中遇到的一些问题/教训

   （1）根据 CpuRst 信号设置 Stall 和 Flush 信号后，Flush 忘记恢复（非 Rst 的情况没有判断完全，最后的 else 应该默认没有 Flush）

   （2）NPC_Generator 中的各个 Target 输入进来就是加过旧的 PC 了，不用再加。（除了不跳转的情况用 PCF+4）

   （3）ControlUnit 中用位拼接的方式赋值时，某个信号对应的赋值少了一位，导致之后的高位信号赋值都错位了。

   （4）SLL/SRL/SRA 的两个操作数不要写反。

   （5）SLL/SRL/SRA 的第二个操作数，即移位数，应该只取低 5 位。

   （6）Jalr 指令跳转的基址是 rs1 而不是旧 PC。

   （7）NPC_Generator 中对 JalD 这个不能放在 EX 的两种情况之前，因为 EX 段的信号比 ID 段的信号早一条指令，跳转应该是先满足最早的

   （8）Control 模块输出的默认值（特别时几个跳转确定信号）不应该置为 x（置为 0），否则导致 HarzardUnit 中会出现不确定状态，导致无法正确输出 Stall 和 Flush 信号

   （9）+和-的优先级比<<高，3'b100 - 3'b001 << Fn3 会先计算减法

3. 仿真过程总结

（1）点击 Run Simulation 选择 Run Behavioral Simulation 开始仿真

（2）把 Scope 或者 Objects 里的信号拖到波形图里，可以查看该信号（需要点 Restart 再点 Run All）

（3）如果修改了源代码，不用退出仿真，点 Relunch Simulation 即可

（4）仿真也可指定时间，避免一次性执行太长时间

（5）Save Waveform Configuration 可保存当前波形配置（显示哪些信号，高亮，标记，进制等信息），下次仿真时可以载入。

改进实验的意见

1. 直接提供 riscv32-gcc 交叉编译器，而不是到 github 上克隆几个 g 的源码自行编译和配置，浪费时间且容易出问题（已改进）
2. 三个阶段的实验可以合并检查