1. CPU 概要设计说明

1.1 项目实现的体系结构特征

1. 实现的特权态：M 态
2. 实现的指令：

分类	指令	数目
算术和逻辑运算	LUI/AUIPC ADDI/SLTI/SLTIU/ADD/SUB/SLT/SLTU SLLI/SRLI/SRAI/SLL/SRL/SRA ANDI/ORI/XORI/AND/OR/XOR	21
控制	JAL/JALR BEQ/BNE/BLT/BGE/BLTU/BGEU	8
数据传输	LB/LH/LW/LBU/LHU SB/SH/SW	8
总计		37

RV32I 37 条指令均已实现且测试通过。

各个指令的含义如下：

Table B.1 RV32I: RISC-V integer instructions

op	funct3	funct7	Type	Instruction	Description	Operation
0000011 (3)	000	–	I	lb rd, imm(rs1)	load byte	rd = SignExt([Address]$_{7:0}$)
0000011 (3)	001	–	I	lh rd, imm(rs1)	load half	rd = SignExt([Address]$_{15:0}$)
0000011 (3)	010	–	I	lw rd, imm(rs1)	load word	rd = [Address]$_{31:0}$
0000011 (3)	100	–	I	lbu rd, imm(rs1)	load byte unsigned	rd = ZeroExt([Address]$_{7:0}$)
0000011 (3)	101	–	I	lhu rd, imm(rs1)	load half unsigned	rd = ZeroExt([Address]$_{15:0}$)
0010011 (19)	000	–	I	addi rd, rs1, imm	add immediate	rd = rs1 + SignExt(imm)
0010011 (19)	001	0000000*	I	slli rd, rs1, uimm	shift left logical immediate	rd = rs1 << uimm
0010011 (19)	010	–	I	slti rd, rs1, imm	set less than immediate	rd = (rs1 < SignExt(imm))
0010011 (19)	011	–	I	sltiu rd, rs1, imm	set less than imm. unsigned	rd = (rs1 < SignExt(imm))
0010011 (19)	100	–	I	xori rd, rs1, imm	xor immediate	rd = rs1 ^ SignExt(imm)
0010011 (19)	101	0000000*	I	srli rd, rs1, uimm	shift right logical immediate	rd = rs1 >> uimm
0010011 (19)	101	0100000*	I	srai rd, rs1, uimm	shift right arithmetic imm.	rd = rs1 >>> uimm
0010011 (19)	110	–	I	ori rd, rs1, imm	or immediate	rd = rs1 | SignExt(imm)
0010011 (19)	111	–	I	andi rd, rs1, imm	and immediate	rd = rs1 & SignExt(imm)
0010111 (23)	–	–	U	auipc rd, upimm	add upper immediate to PC	rd = {upimm, 12'b0} + PC
0100011 (35)	000	–	S	sb rs2, imm(rs1)	store byte	[Address]$_{7:0}$ = rs2$_{7:0}$
0100011 (35)	001	–	S	sh rs2, imm(rs1)	store half	[Address]$_{15:0}$ = rs2$_{15:0}$
0100011 (35)	010	–	S	sw rs2, imm(rs1)	store word	[Address]$_{31:0}$ = rs2
0110011 (51)	000	0000000	R	add rd, rs1, rs2	add	rd = rs1 + rs2
0110011 (51)	000	0100000	R	sub rd, rs1, rs2	sub	rd = rs1 − rs2
0110011 (51)	001	0000000	R	sll rd, rs1, rs2	shift left logical	rd = rs1 << rs2$_{4:0}$
0110011 (51)	010	0000000	R	slt rd, rs1, rs2	set less than	rd = (rs1 < rs2)
0110011 (51)	011	0000000	R	sltu rd, rs1, rs2	set less than unsigned	rd = (rs1 < rs2)
0110011 (51)	100	0000000	R	xor rd, rs1, rs2	xor	rd = rs1 ^ rs2
0110011 (51)	101	0000000	R	srl rd, rs1, rs2	shift right logical	rd = rs1 >> rs2$_{4:0}$
0110011 (51)	101	0100000	R	sra rd, rs1, rs2	shift right arithmetic	rd = rs1 >>> rs2$_{4:0}$
0110011 (51)	110	0000000	R	or rd, rs1, rs2	or	rd = rs1 | rs2
0110011 (51)	111	0000000	R	and rd, rs1, rs2	and	rd = rs1 & rs2
0110111 (55)	–	–	U	lui rd, upimm	load upper immediate	rd = {upimm, 12'b0}
1100011 (99)	000	–	B	beq rs1, rs2, label	branch if =	if (rs1 == rs2) PC = BTA
1100011 (99)	001	–	B	bne rs1, rs2, label	branch if ≠	if (rs1 ≠ rs2) PC = BTA
1100011 (99)	100	–	B	blt rs1, rs2, label	branch if <	if (rs1 < rs2) PC = BTA
1100011 (99)	101	–	B	bge rs1, rs2, label	branch if ≥	if (rs1 ≥ rs2) PC = BTA
1100011 (99)	110	–	B	bltu rs1, rs2, label	branch if < unsigned	if (rs1 < rs2) PC = BTA
1100011 (99)	111	–	B	bgeu rs1, rs2, label	branch if ≥ unsigned	if (rs1 ≥ rs2) PC = BTA
1100111 (103)	000	–	I	jalr rd, rs1, imm	jump and link register	PC = rs1 + SignExt(imm), rd = PC + 4
1101111 (111)	–	–	J	jal rd, label	jump and link	PC = JTA, rd = PC + 4

*Encoded in instr$_{31:25}$, the upper seven bits of the immediate field

图 1　RV32I 各指令的含义[1]

[1] Sarah Harris, David Ha. *Digital Design and Computer Architecture RISC-V Edition*

3. 流水线架构：

五阶流水线：取指、译码、执行、存储、写回

采用顺序发射、顺序执行、顺序写回

4. 通过停顿（stall）流水线以及数据旁路解决了数据冲突，通过清空（flush）流水线寄存器解决了控制冲突：

（1）数据旁路：通过前递电路检测存入 ID/EX 流水站的指令与其前面的指令的数据相关性（对于 store 指令则是在 EX/MEM 流水站检测将要存入的 memory 的数据是否存在相关性），自动从 EX/MEM, MEM/WB 流水站以及寄存器堆（即寄存器堆内部的旁路，可实现立即读取同时刻写入的值）中获取所需的操作数。数据旁路直接解决了整数算数指令与整数算数指令之间、整数算数指令与 store 指令之间的数据冲突，不需要停顿流水线；同时也使得加载指令之后只需要停顿（停止更新）一拍即可解决数据冲突。

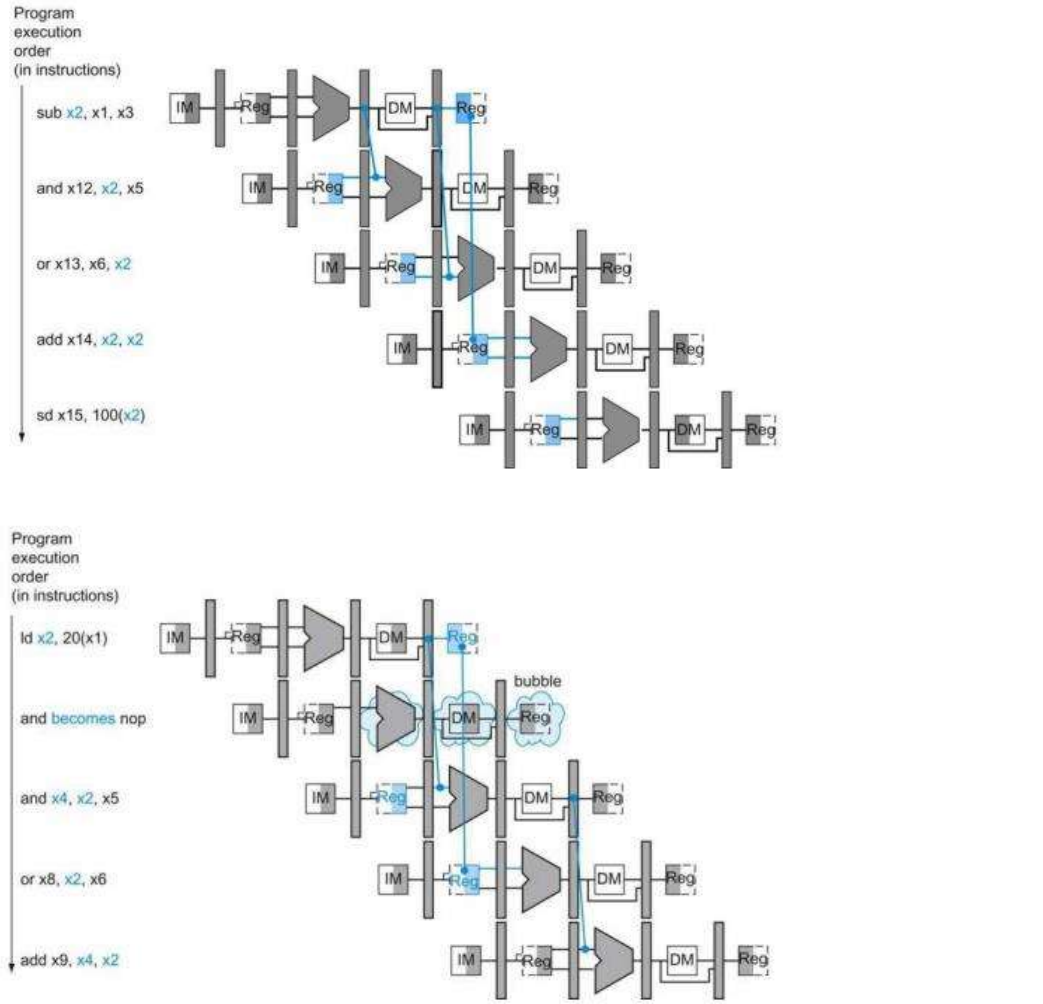

图 2　数据旁路采用的思路[1]：整数算数指令（上） 加载指令（下）

[1] David A. Patterson, John L. Hennessy. *Computer Organization and Design : The Hardware / Software Interface: RISC-V Edition*

（2）清空流水线：在 EX STAGE 完成是否跳转的判断以及跳转地址的计算。（因为考虑到要实现多种不同的分支指令，同时希望降低电路复杂度减小电路面积，就没有采用将跳转的判断以及跳转地址的计算移至 ID STAGE）。因此当跳转发生后，将会更新 PC 值，同时清空 IF/ID, ID/EX 两个流水站，即产生两拍的空泡。

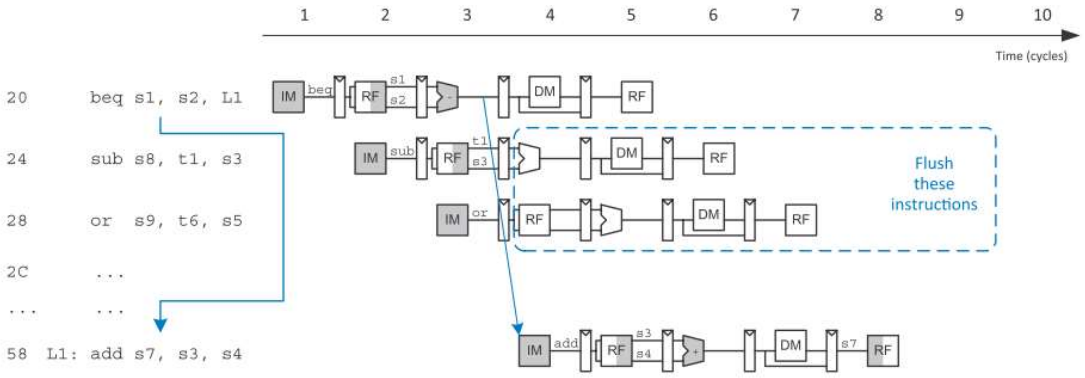

图 3 控制相关的解决方案[1]

[1] Sarah Harris, David Ha. *Digital Design and Computer Architecture RISC-V Edition*

2. CPU 详细设计说明

2.1 模块划分

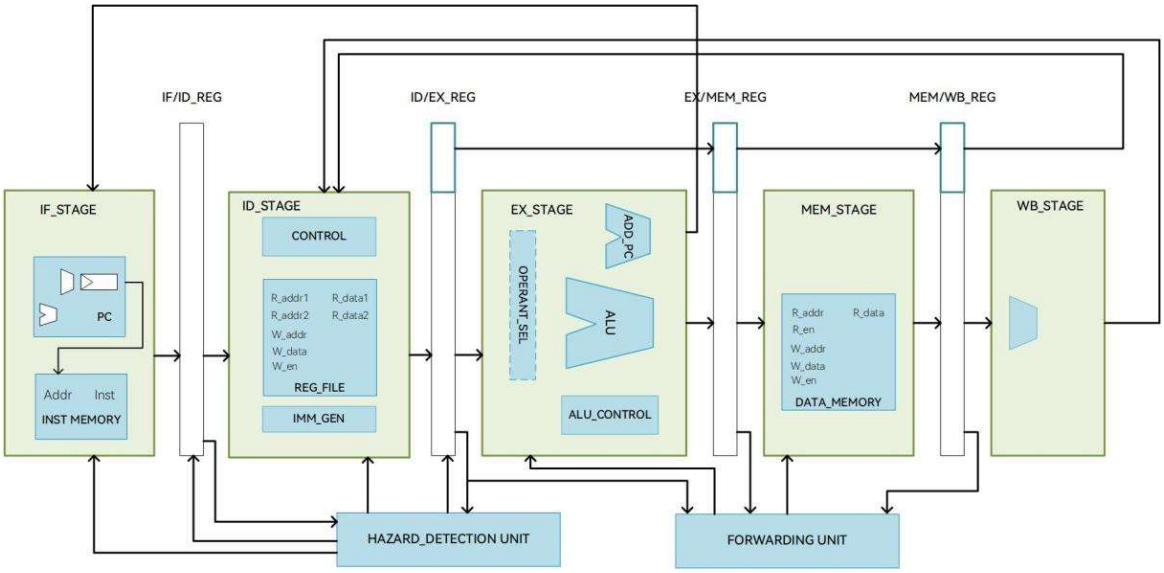

图 4 模块划分示意

整个 CPU 核各个部分的大致划分如上图所示（上图省略了细节部分）。分为五个阶段、四个流水站寄存器、一个冲突检测单元和前递单元。

2.1.1 IF_STAGE 模块

IF_STAGE 模块主要包含 PC 与指令存储器。PC 子模块实现了 PC 地址随时钟自加 4，以及根据跳转信号 br_crtl，选择跳转地址；根据停顿信号暂停 pc 更新。指令存储器子模块中通过$readmemh 系统函数将汇编程序机器码存入存储器。指令存储器采用按字节小端序编址，每个地址的存储单元位宽为 1byte，地址字长为 32 位，但是存储器的大小只设置为 1024×1btye。

2.1.2 ID_STAGE 模块

ID_STAGE 模块主要包含控制信号单元 control unit，寄存器堆，立即数生成单元。

control unit 子模块根据指令操作码识别当前指令类型，发出 EX, MEM, WB 阶段所需的各种控制信号，使得这些阶段能针对不同的指令类型执行不同的功能。这些信号的具体含义在 2.2 节中有详细介绍。在接收到 stall 信号后会将输出的控制信号全部置零。

寄存器堆中包含 32 个 32 位寄存器。读取寄存器堆的数据不需要时钟信号以及使能信号即可读出数据；写入数据则是在时钟沿上升且写使能信号有效时才可写入数据，且不允许向 x0 寄存器写入任何数据。寄存器堆内部设置了内部旁路，可以直接读出同时刻内写入寄存器堆相同位置的数据。

立即数生成单元根据指令操作码提取出指令中的立即数部分并完成位移、

扩展操作，生成可被 EX 阶段直接使用的立即数。

2.1.3 EX_STAGE 模块

EX_STAGE 模块包含 ALU 控制子模块，ALU 子模块，以及操作数选择逻辑，跳转判断逻辑，跳转地址计算逻辑。

ALU 控制子模块根据 control unit 传递的型号来判断指令的类型，根据指令的类型以及 func3，func7 字段进一步区分出每个指令的操作，将其对应的操作控制信号给 ALU 子模块，控制 ALU 的运算模式。如当前指令为 BEQ 时，控制 ALU 对操作数作减法计算，以比较两个操作数是否相等，来判断分支是否发生。

ALU 子模块共有 AND, OR, ADD, SUB, SLT, SLL, SRL, SRA, JUMP, NOT_EQ 等 14 种模式。除了直接输出计算结果，ALU 模块也会根据计算模式，来输出跳转条件计算结果信号 br_mark。如当前指令为 BEQ 时（ALU 为 SUB 模式），ALU 对操作数作减法计算，若结果为零，则 br_mark 输出 1；当前指令为无条件跳转时（ALU 为 JUMP 模式），br_mark 直接输出 1。

操作数选择逻辑部分首先根据前递信号 ForwardA，ForwardB，在 ID/EX 寄存器数据（无数据冒险）、EX/MEM 寄存器（EX 冒险）、MEM/WB（MEM 冒险）中选择源操作数寄存器数据来源。再根据 control unit 传递的选择信号 ALU_Src（由指令类型确定），判断输入给 ALU 的操作数选用立即数/源操作数寄存器数据/EX 阶段 PC 值（无条件跳转指令中的 Rd = pc+4）/4（无条件跳转指令中的 Rd = pc+4）/0（lui 只需要立即数这一个操作数，则另一个操作数视为 0）。

跳转地址的计算逻辑是一个专门的加法器，但也需要根据 control unit 传递的选择信号 br_addr_mode，来选择一个操作数是来自 PC/Rs1（jalr 指令中，跳转 pc=rs1+imm）。而另一个操作数固定为立即数。跳转的判断逻辑为 ALU 输出的 br_mark（满足跳转条件）同 control unit 传递的 br 信号（表示当前指令是否为跳转或分支指令）作与运算。跳转判断信号与跳转地址结果均返回到 IF_stage 的 PC 中。、

2.1.4 MEM_STAGE 模块

MEM_STAGE 模块主要包含一个数据存储器，与指令存储器一样，采用按字节小端序编址，地址字长为 32 位，存储器的大小设置为 1024×1Btye。数据存储器只有一个地址端口，因此不允许同时读写数据。MEM_STAGE 模块使用 func3 字段作用片选的判断信号，用以区分 sb/sh/sw 以及区分 lb/lh/lw/lbu/lhu。

2.1.5 WB_STAGE 模块

WB_STAGE 模块实际只有一个 MUX，根据 control unit 传递的 mem2reg 信号来决定写回寄存器堆的数据来自于 ALU 计算结果还是数据存储器读出的值。

2.1.6 Forward Unit 模块与 Hazard Detection Unit 模块

Forward Unit 模块中主要包含了 EX 冒险，MEM 冒险，Store 指令冒险的判断逻辑，并生成相应的控制信号，具体原理在 2.3 节中由详细介绍。Hazard Detection Unit 模块主要为 load 指令的数据冲突判断逻辑，以及生成 stall 信号、跳转指令 flush 信号。

2. 2 模块间接口信号

1. IF_STAGE 模块

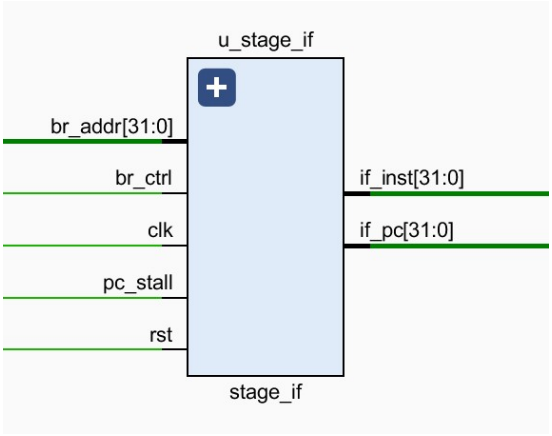

图 5　IF_STAGE 模块接口示意

- **br_addr**：分支指令、无条件跳转指令的跳转 pc 地址；
- **br_ctrl**：决定是否跳转；
- **pc_stall**：决定是否需要停顿流水线，为 true 时，pc 保持原来的值不变；
- **if_inst**：从指令存储器中取出的指令，发往 IF/ID 流水站寄存器；
- **if_pc**：当前指令的 pc 值发往 IF/ID 流水站寄存器；

2. REG_IF_ID

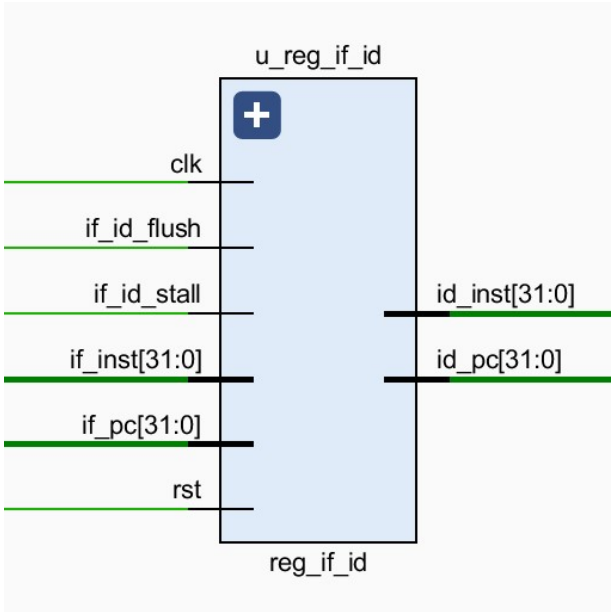

图 6　REG_IF_ID 模块接口示意

- **if_id_flush**：决定是否清零 IF/ID 寄存器所有值（用于分支跳转）；
- **if_id_stall**：决定是否需要停顿，为 true 时，id_inst 与 id_pc 保持原来的值不变;
- **if_inst， if_pc**：由 IF_STAGE 发送的当前 pc 值、当前指令；
- **id_inst**： IF/ID 寄存器保存的当前指令，发送给 ID_STAGE 模块；
- **id_pc**：IF/ID 寄存器保存的当前 pc 值、当前指令，发送给 ID_STAGE 模块与 ID/EX 寄存器；

3. ID_STAGE 模块

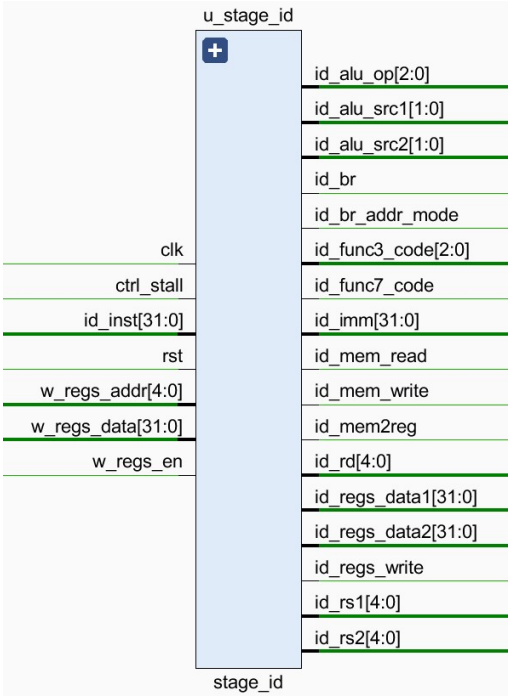

图 7　ID_STAGE 模块接口示意

（1）控制信号单元输出

- **w_regs_en，w_regs_data，w_regs_addr**：由写回模块发送的写回使能、需写回寄存器堆的数据以及相应的寄存器编号；
- **id_alu_op**：由 id_stage 内的 control 子模块根据指令操作码部分生成的控制信号，作用于 ex_stage 内的 alu_control 子模块，告知当前指令属于哪种格式（R-type，I-type 等等）；
- **id_alu_src1**：由 id_stage 内的 control 子模块根据指令操作码部分生成的控制信号，作用于 ex_stage 内的操作数 A 的 MUX，控制操作数 A 的来源为寄存器数据/操作的指令对应的 PC 值/零（即不需要操作数 A，如 lui 指令）；
- **id_alu_src2**：由 id_stage 内的 control 子模块根据指令操作码部分生成的控制信号，作用于 ex_stage 内的操作数 B 的 MUX，控制操作数 B 的来源为寄存器数据/立即数/4（用于实现 Rd = PC+4，如 jal 指令）；

- **id_br**：由 id_stage 内的 control 子模块根据指令操作码部分生成的控制信号，作用于 ex_stage 内的 br_crtl 信号的运算，值为 true 时说明当前指令为分支跳转指令；
- **id_br_addr_mode**：由 id_stage 内的 control 子模块根据指令操作码部分生成的控制信号，作用于 ex_stage 内跳转地址的计算的操作数选择 MUX，控制操作数的来源为指令对应的 PC 值/Rs1（如 jalr 指令）；
- **id_mem_read**：由 id_stage 内的 control 子模块根据指令操作码部分生成的控制信号，作用于 mem_stage 内的数据存储器，作为读使能，在执行 load 指令时为 true；
- **id_mem_write**：由 id_stage 内的 control 子模块根据指令操作码部分生成的控制信号，作用于 mem_stage 内的数据存储器，作为写使能，在执行 store 指令时为 true；
- **id_mem2reg**：由 id_stage 内的 control 子模块根据指令操作码部分生成的控制信号，作用于 wb_stage 内的 MUX，决定写回寄存器堆的数据来自于 alu 输出还是 mem_stage 内的数据存储器的读输出；
- **id_regs_write**：由 id_stage 内的 control 子模块根据指令操作码部分生成的控制信号，沿流水线传递到 MEM/WB 寄存器最后回传到 ID_STAGE 的寄存器堆，作为写回寄存器堆的写使能；

（2）其他接口

- **Ctrl_stall**：发生停顿时，控制 id_stage 内的 control 子模块输出（id_alu_op, id_alu_src1，id_alu_src2，id_br， id_br_addr_mode，id_mem_read，id_mem_write，id_mem2reg，id_regs_write）为 0，使得被停顿的指令走完流水线后不会对存储与状态造成任何修改。
- **id_inst**：由 REG_IF_ID 发送的当前指令；
- **id_func3_code，id_func7_code**：传递指令的 func3，func7 字段，经由 REG_ID_EXE 传到 EX_STAGE，作用于 ex_stage 内的 alu_control 子模块；
- **id_imm**：由 id_stage 内的 imm_gen 子模块根据指令类型所生成的，经过拓展的、可直接使用的完整 32 位立即数；
- **id_rs1，id_rs2**：传递该指令的源操作数寄存器编号给后续的流水站寄存器，用于冲突检测单元与前递单元的判断；
- **id_rd**：传递该指令的目的寄存器编号给后续的流水站寄存器，作为写回寄存器堆时的地址，也用于冲突检测单元与前递单元的判断
- **id_regs_data1, id_regs_data2**：在 id_stage 内的寄存器堆中取出的源操作数寄存器 Rs1、Rs2 值；

4. REG_ID_EX

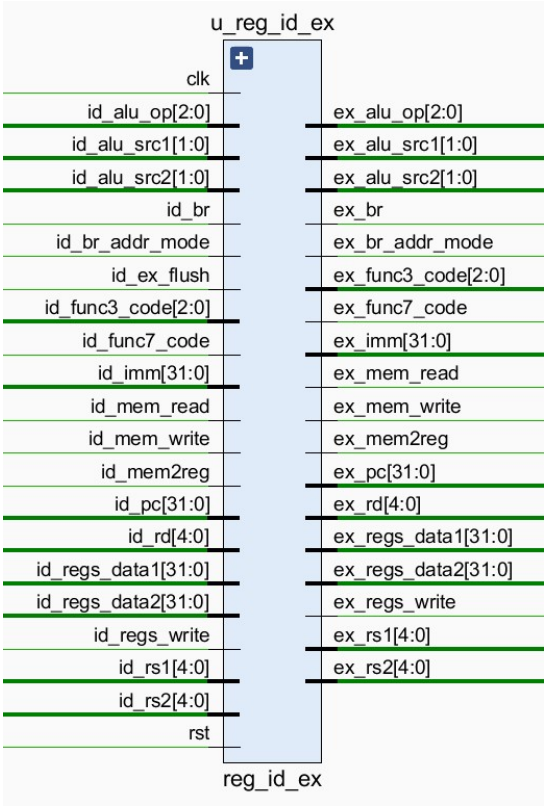

图 8　REG_ID_EX 模块接口示意

- **id_ex_flush**：决定是否清零 ID/EX 寄存器所有值（用于分支跳转）；
- **id_pc，ex_pc**：暂存、转发来自 IF/ID 寄存器的当前指令的 PC 地址；
- 其余接口均为用于暂存、转发来自 ID_STAGE 的信号；

5. EX_STAGE 模块

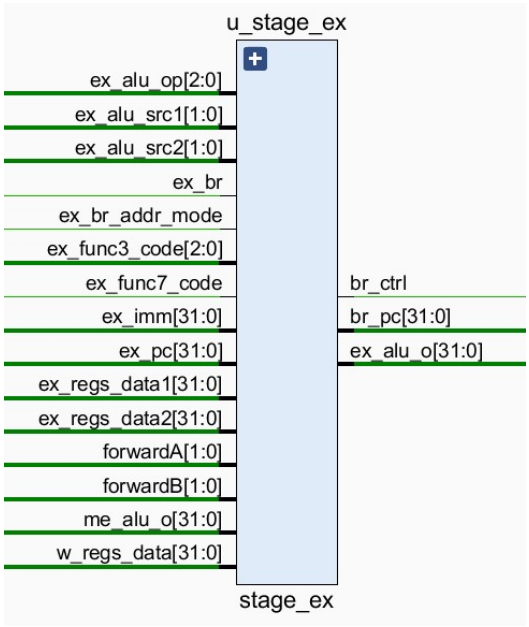

图 9　EX_STAGE 模块接口示意

- **forwardA**：由 Forward Unit 发送的信号，在当操作数 A 来源为寄存器时，用作决定数据来自寄存器堆还是前面的指令的结果；
- **forwardB**：由 Forward Unit 发送的信号，在当操作数 B 来源为寄存器时，用作决定数据来自寄存器堆还是前面的指令的结果；
- **me_alu_o**：上一条指令的计算结果，来自 REG_EX_MEM；
- **w_regs_data**：写回阶段将要写回寄存器堆的结果，即上上条指令的计算结果；
- **br_ctrl**：由 EX_STAGE 的 alu 计算结果（判断分支条件是否成立）与 ex_br 信号共同生成，作用于 IF_STAGE 模块的 PC 子模块，控制 PC 是否跳转；
- **br_ctrl**：由 EX_STAGE 生成的 pc 跳转地址计算结果；
- **ex_alu_o**：EX_STAGE 的 ALU 计算结果；

6. REG_EX_MEM

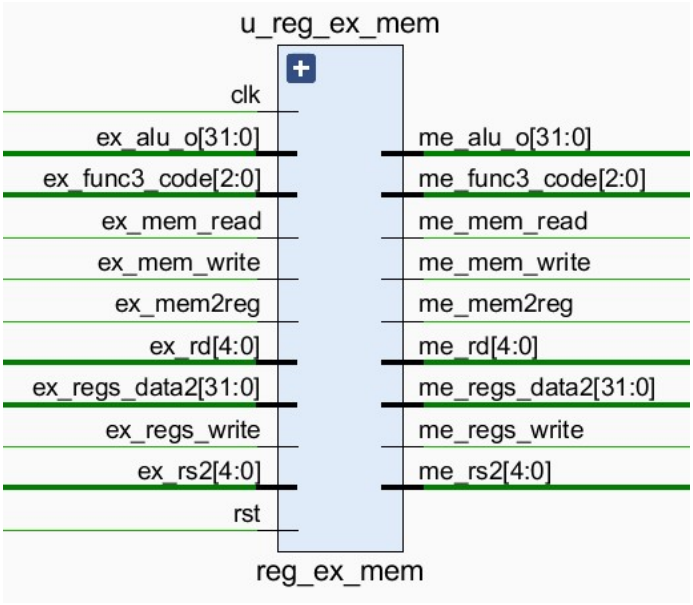

图 10　REG_EX_MEM 模块接口示意

接口均为暂存、转发来自 EX_STAGE 和 REG_ID_EX 模块的部分信号；

7. MEM_STAGE 模块

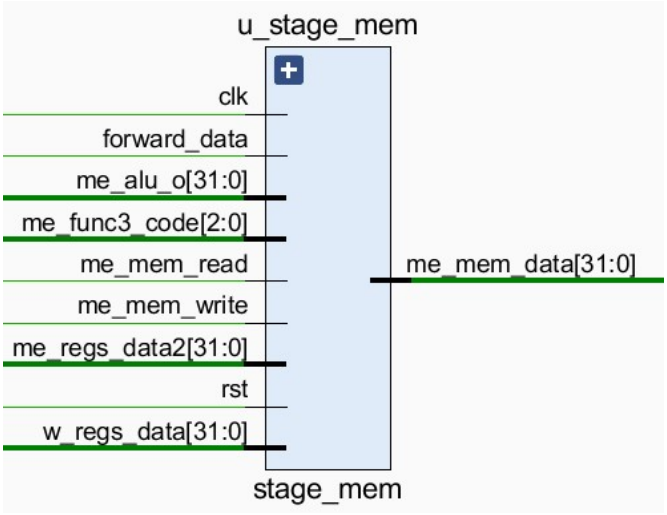

图 11　MEM_STAGE 模块接口示意

- forward_data：由 Forward Unit 发送的信号，用作决定要存储的数据来自寄存器堆 Rs2（me_regs_data2）还是前面的指令的结果（即写回阶段的数据，w_regs_data）；
- me_alu_o：ex_stage 的 ALU 的计算结果，用作读写数据存储器的地址；
- me_func3_code：指令的 func3 字段，用作区分读写字长、有无符号位的信号（区分 lb，lh，lw，lbu，lhu，以及区分 sb，sh，sw）；
- me_mem_data：从数据存储器中读出的数据；
- 其余信号皆为前面已经介绍过的信号；

8. REG_MEM_WB 与 WB_STAGE

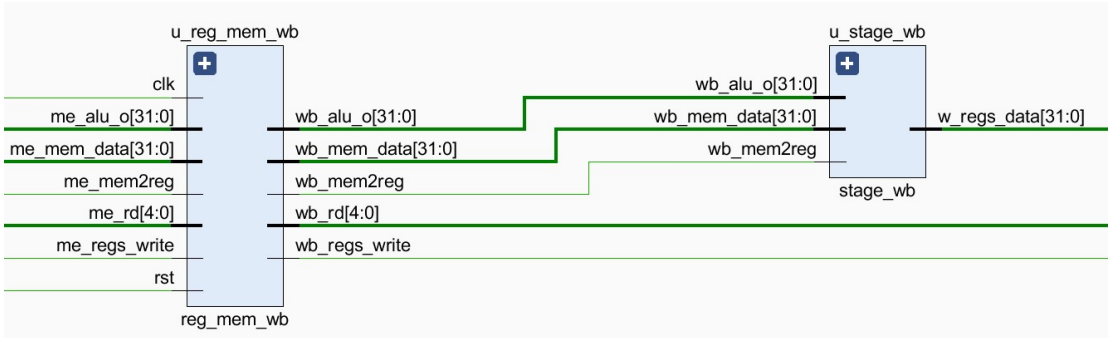

图 12　REG_MEM_WB 与 WB_STAGE 模块接口示意

- **wb_alu_o，wb_mem_data**：写回阶段的数据来源，前者为 ALU 结果，后者为数据存储器读出值；
- **wb_mem_data**：用于选择数据来源是 ALU 结果还是数据存储器读出值的信号；
- **wb_rd，wb_regs_write**：写回地址与写使能，与 ID_STAGE 的 w_regs_addr ，w_regs_en 接口对应。

9. Forward Unit

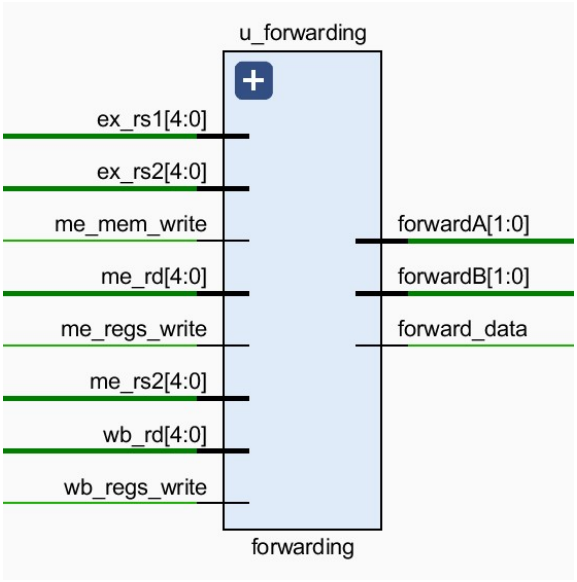

图 13　Forward Unit 模块接口示意

- **ex_rs1, ex_rs2**：处于 EX 阶段的指令中的源操作数寄存器编号，来自 REG_ID_ EX 的输出；
- **me_regs_write，wb_regs_write**：同一时刻 REG_EX_MEM 与 REG_MEM_WB 中的 regs_write 写使能信号，用来标志当前在 MEM 阶段和 WB 的指令均要将数据写回寄存器堆；
- **me_rd，wb_rd**：处于 MEM 阶段、WB 阶段的指令中的目的寄存器编号；
- **me_rs2**：处于 MEM 阶段的 store 指令要存储的数据来源；
- **me_mem_write**：MEM 阶段的数据存储器写使能信号，用于标志 MEM 阶段的指令为 store 指令。
- ForwardA，ForwardB，Forward_data 信号前面分别在 EX_STAGE、MEM_ STAGE 中介绍了。

10. Hazard Detection Unit

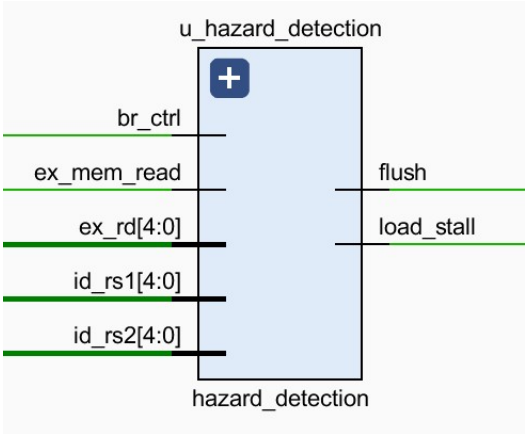

图 14　Hazard Detection Unit 模块接口示意

- **id_rs1，id_rs2**：处于 ID 阶段的指令的源操作数寄存器编号；
- **ex_rd**：处于 EX 阶段的指令的目的寄存器编号；
- **br_ctrl**：发生分支跳转的标志信号；
- **ex_mem_read**：标志处于 EX 阶段的指令为 load 指令；
- **flush**：对应其他模块中名字带“flush”的信号，当发生分支跳转时，清零 REG_IF_ID 与 REG_ID_EX;
- **load_stall**：当 load 指令后一条的指令在译码阶段发现与 load 指令存在数据相关时，发出的停顿信号，使得 PC 寄存器、REG_IF_ID 停止更新同时解除 ID_STAGE 阶段生成的控制信号，使得流水线停顿，新进入流水线的指令变为空指令。

2. 3 数据冲突的解决方案

2.3.1 数据旁路的实现

为了不必等待数据写回寄存器堆，通过检查数据相关性，设置数据旁路（前递），在获得数据后立即同步给需要该数据的模块。前递分 4 种情况：

1.EX 冒险：

进入 EX 阶段的指令与完成 EX 阶段的指令存在数据相关；

2. MEM 冒险：

进入 EX 阶段的指令与完成 MEM 阶段的指令存在数据相关；

3. WB 冒险：

进入 EX 阶段的指令与完成 WB 阶段的指令存在数据相关；

4. store 指令与其他指令相关造成的冒险：

进入 MEM 阶段的 store 指令所需的数据与完成 MEM 阶段的指令存在数据相关。

又因为不是所有指令都会写回寄存器，同时还需要排除掉以 x0 为目标寄存器的指令，因此在 forward unit，判断相关性的关系式如下：

（1）EX 相关：

```
assign ex_hazard_a=me_regs_write&&(me_rd!=0)&&(me_rd == ex_rs1);
```

（2）MEM 相关：

```
assign mem_hazard_a=wb_regs_write&&(wb_rd!=0)&&(wb_rd == ex_rs1);
assign forwardA     = ex_hazard_a ? 2'b10 : mem_hazard_a ? 2'b01 :
2'b00;
```

（3）WB 相关

这种情况下，ID 阶段读取的寄存器与 WB 阶段要写入的寄存器相同，因此可通过在寄存器堆内部设置前递，直接输出正要写入的值：

```
assign wb_hazard_a = w_regs_en && (w_regs_addr != 0) && (w_regs_addr ==
r_regs_addr1);
assign r_regs_o1 = wb_hazard_a ? w_regs_data : regs_file[r_regs_addr1];
```

（4）store 指令与其他指令相关造成的冒险：

```
assign hazard_data_w = wb_regs_write && (wb_rd != 0) && (wb_rd ==
me_rs2) && me_mem_write; //for store
assign forward_data = hazard_data_w;
```

forwardA, forwardB, forward_data 作为前递冒险检测的结果，输入到 EX_STAGE（forward_data 为 MEM_STAGE）中，作为操作数来源的片选信号，从而实现控制操作数来源。

2.3.2 流水线停顿的实现

通过数据旁路，已经能解决大多数情况的数据冲突问题，但是对于 load 指令，由于它在 MEM 阶段结束时才得到所需要的结果，即使运用数据旁路，产生结果的时刻仍然比需要结果的时刻晚了一拍，因此需要停顿流水线一周期，再借助 MEM 相关的旁路通道将结果传递给下个指令。因此在 Hazard Detection Unit 模块，通过

```
assign load_stall = ex_mem_read && (ex_rd == id_rs1 || ex_rd ==
id_rs2);
```

来检测冒险，其中 ex_mem_read 用于检测 EX 阶段的指令为 load 指令，表达式后半部分表示 EX 阶段的加载指令的目标寄存器是否与 ID 阶段的指令中的某一个源寄存器匹配。一旦 load_stall 被激活后， PC 寄存器、REG_IF_ID 停止更新一周期，使得流水线停顿一拍；同时解除 ID_STAGE 阶段生成的控制信号，即将其全部设置为 0，使得原 ID 阶段的指令变为没有任何操作的指令，也就是空指令，直至它排出流水线；停顿一拍后，流水线重新开始执行该指令。

2. 4 控制冲突的解决方案

本项目采用方案为始终预测条件分支不发生，并持续执行顺序指令流，一旦条件分支发生，则已经被读取和译码的指令就将被丢弃，流水线继续从分支目标处开始取指、译码……另外，本项目的分支地址计算通过 EX_STAGE 中的独立的加法器完成，而分支条件的判断则通过识别指令操作码来将 ALU 设置为相应的运算模式，来计算分支条件的判断结果。

因此得到分支指令是否发生跳转的时间节点为分支指令运行到 EX 阶段，一旦发生跳转，就需要将 IF，ID 阶段已经取到的指令清除。因此在 Hazard Detection Unit 模块中，将 flush 信号与跳转控制信号 br_ctrl 相连，flush 信号将会重置 REG_IF_ID 与 REG_ID_EX，PC 在 br_ctrl 作用下从跳转地址 br_addr 处计数。

3. CPU 的功能验证

3.1 验证功能点

1. RV32I 输入 37 条指令是否能正常工作？

2. 整数计算指令与整数计算指令之间，load 指令与其他指令之间，store 指令与其他指令之间等等的数据相关问题能否解决，数据旁路是否有效、停顿能否实现，停顿时长是否符合预期；

3. 能否正确处理条件分支指令、无条件跳转指令，跳转是否正确，跳转后是否实现流水站清空，流水停顿时长是否符合预期。

4. 一些细节能否实现，如禁止对 x0 写入数据等；

3.2 验证手段

1. 编写汇编测试程序，尽可能用上全部指令，并且能体现指令之间的功能差异，同时在注释中标注正确结果。将汇编程序通过工具链转为机器码，使用$readmemh 系统函数写入指令存储器中。后期考虑学习如何使用官方测试集。

2. 在关键数据路径中插入$display,$strobe 系统函数，检测部分控制信号、数据读写行为等等，在控制面板中查看工具的输出记录。其中重点关注写回阶段结果、存储器写入结果，并与汇编程序的正确结果进行比对。

3. 利用仿真工具查看仿真结束后的存储器——寄存器堆、数据存储器中存储的数据是否符合预期。

4. 根据$display,$strobe 系统函数输出记录定位错误的范围，在仿真工具中查看关联信号波形，查找问题原因。

3.3 验证过程

1. 测试程序

（1）核心指令测试程序：

Test1.s

```
.org 0x0
.section .text
.global _start
_start:
main:   addi x2, x0, 5          # x2 = 5                0  00500113
        addi x3, x0, 12         # x3 = 12               4  00C00193
#check_forward
        addi x7, x3, -9         # x7 = (12 - 9) = 3     8  FF718393   #signed
#check_forward
        or   x4, x7, x2         # x4 = (3 OR 5) = 7     C  0023E233
#check_forward
        and  x5, x3, x4         # x5 = (12 AND 7) = 4   10 0041F2B3
#check_forward
        add  x5, x5, x4         # x5 = 4 + 7 = 11       14 004282B3
#check_forward & branch
        beq  x5, x7, end        # shouldn't be taken    18 02728863
        slt  x4, x3, x4         # x4 = (12 < 7) = 0     1C 0041A233
#check_forward & branch
        beq  x4, x0, around     # should be taken       20 00020463
        addi x5, x0, 0          # shouldn't execute     24 00000293
around: slt x4, x7, x2          # x4 = (3 < 5) = 1      28 0023A233
#check_forward
        add x7, x4, x5          # x7 = (1 + 11) = 12    2C 005203B3
#check_forward
        sub x7, x7, x2          # x7 = (12 - 5) = 7     30 402383B3
#check_forward_store
        sw x7, 84(x3)           # [96] = 7              34 0471AA23
        lw x2, 96(x0)           # x2 = [96] = 7         38 06002103
#check stall
        add x9, x2, x5          # x9 = (7 + 11) = 18    3C 005104B3
# jump
        jal x3, end             # jump to end, x3 = 0x44  40 008001EF
        addi x2, x0, 1          # shouldn't execute     44 00100113
#check_forward
end:    add x2, x2, x9          # x2 = (7 + 18) = 25    48 00910133
#check_forward_store
        sw x2, 0x20(x3)         # [100] = 25            4C 0221A023
#check_forward & branch
done:   beq x2, x2, done        # infinite loop         50 00210063
```

（2）其他指令测试程序：

Test2.s

```
.org 0x0
    .global _start
_start:
    ori x1, x0, 0x210           # x1 = 0210
    ori x2, x1, 0x021           # x2 = 0x231
    slli x3, x2, 1              # x3 = 0b010001100010 = 0x462
    andi x4, x3, 0x568          # x4 = 0b010001100000 = 0x460
    ori x5, x0, 0x68a           # x5 = 0b011010001010 = 0x68a
    ori x7, x0, 22              # x7 = 22 = 0x16
    sll x5, x5, x7              # x5 = 0xa2800000
    xori x7, x7, 0x19           # x7 = 0b00001111 = 0xf
```

```
    srli x7, x7, 2              # x7 = 0b00000011 = 0x3
    addi x7, x7, 1              # x7 = 0x4
    srl  x8, x5, x7             # x8 = 0xa280000
    sra x6, x5, x7              # x6 = 0xfa280000
    srai x6, x5, 16             # x6 = 0xffffa280

    ori x5, x0, 0x723           # x5 = 0b011100100011 = 0x723
    xor x5, x5, x4              # x5 = 0b001101000011 = 0x343

    slti x7, x6, 0x7a4          # x7 = 1
    sltiu x8, x6, 0x7a4         # x8 = 0

    slt  x7, x6, x5             # x7 = 1
    sltu x8, x6, x5             # x8 = 0

    add x6, x5, x4              # x6 = 0x7a3

    slti x8, x6, 0x7a3          # x8 = 0
    slt x8, x6, x5              # x8 = 0
    slt x8, x5, x6              # x8 = 1
    sub x9, x6, x5              # x9 = 0x460
    lui x10, 0x45b27            # x10 = 0x45b27000
    auipc x11, 0x21c43          # x11 = 0x21c43064
es_j1:
    blt x10, x11, es_j2         # no jump
    lui x12, 0xffff             # x12 = 0x0ffff000
    bge x10, x11, es_j2         # jump to es_j2
es_j4:
    ori x12, x0, 0x456          # x12 = 0x456
    jalr x18,x8, 3              # jump to 0x04,beginning
    ori x13, x0, 0x2bc          # shouldn't execute

    nop
    nop
    nop
es_j2:
    ori x12, x0, 0x5ef          # x12 = 0x5ef
    ori x13, x0, 0x123          # x13 = 0x123
    sb x11, 2(x13)              # store 0x64 to mem:0x125
    lb x14, 2(x13)              # x14 = 0x64
    sb x12, 1(x13)              # store 0xef to mem:0x124
    lh x14, 1(x13)              # x14 = 0xffff64ef
    add x15, x14, x0            # x15 = 0x64ef
    sh x5, 3(x13)               # store 0x0343 to mem:0x126
```

```
    lw x15, 1(x13)              # x15 = 0x034364ef
    add x17, x7, x15            # x17 = 0x034364f0
    sw x11, 5(x13)              # store 0x21c43064 to mem:0x128
    lw x16, 5(x13)              # x16 = 0x21c43064
    add x17, x7, x17            # x17 = 0x034364f1
es_j3:
    lui x18, 0xfffff            # x18 = 0xfffff000
    addi x19, x18, 1            # x19 = 0xfffff001
    bgeu x19, x18,es_j5         # jump 2 es_j5

es_j6:
    addi x18, x18, 1            # x18 = 0xfffff001
    bne x19, x18,es_j6          # no jump
    jal x0, es_j4               # jump to es_j3

es_j5:
        bge x19, x18,es_j6          # jump to es_j6
```

(2)仿真工具输出

（节选自 test2.s 的测试输出）

```
0<<Starting simulation>>
    # if_id_flush pc: xxxxxxxx
    # ex_regs_data1: xxxxxxxx
    # ex_regs_data2: xxxxxxxx
    # ex_imm: xxxxxxxx
    # ex_alu_op: x
    # me_alu_o: xxxxxxxx
    # wb_mem_data    : xxxxxxxx
    # wb_alu_o       : xxxxxxxx
    # wb_mem2reg     : x
    # wb_regs_write: x
    # if_id_flush pc: 00000000
    # ex_regs_data1: 00000000
    # ex_regs_data2: 00000000
    # ex_imm: 00000000
    # ex_alu_op: 0
    # me_alu_o: 00000000
    # wb_mem_data    : 00000000
    # wb_alu_o       : 00000000
    # wb_mem2reg     : 0
    # wb_regs_write: 0
    # ----------------------
```

```
# PC_O = PC_next: 00000000
# id_inst: 00000000
# ex_regs_data1: 00000000
# ex_regs_data2: 00000000
# ex_imm: 00000000
# ex_alu_op: 0
# me_alu_o: 00000000
# wb_mem_data    : 00000000
# wb_alu_o       : 00000000
# wb_mem2reg     : 0
# wb_regs_write: 0
# ----------------------
# PC_O = PC_next: 00000004
# id_inst: 21006093
# ex_regs_data1: 00000000
# ex_regs_data2: 00000000
# ex_imm: 00000000
# ex_alu_op: 7
# me_alu_o: 00000000
# wb_mem_data    : 00000000
# wb_alu_o       : 00000000
# wb_mem2reg     : 0
# wb_regs_write: 0
# ----------------------
# PC_O = PC_next: 00000008
# id_inst: 0210e113
# ex_regs_data1: 00000000
# ex_regs_data2: xxxxxxxx
# ex_imm: 00000210
# ex_alu_op: 3
# me_alu_o: 00000000
# wb_mem_data    : 00000000
# wb_alu_o       : 00000000
# wb_mem2reg     : 0
# wb_regs_write: 0
# forwardA! OP_A: 00000210
# forwardA! ex_hazard: 1, mem_hazard: 0
# ----------------------
# PC_O = PC_next: 0000000c
# id_inst: 00111193
```

```
# ex_regs_data1: xxxxxxxx
# ex_regs_data2: xxxxxxxx
# ex_imm: 00000021
# ex_alu_op: 3
# me_alu_o: 00000210
# wb_mem_data    : 00000000
# wb_alu_o       : 00000000
# wb_mem2reg     : 0
# wb_regs_write: 0
# forwardA! OP_A: 00000231
# forwardA! ex_hazard: 1, mem_hazard: 0
# ----------------------
# PC_O = PC_next: 00000010
# id_inst: 5681f213
# WRITE REGISTER FILE: x 1 = 00000210
# ex_regs_data1: xxxxxxxx
# ex_regs_data2: xxxxxxxx
# ex_imm: 00000001
# ex_alu_op: 3
# me_alu_o: 00000231
# wb_mem_data    : 00000000
# wb_alu_o       : 00000210
# wb_mem2reg     : 0
# wb_regs_write: 1
# forwardA! OP_A: 00000462
# forwardA! ex_hazard: 1, mem_hazard: 0
# ----------------------
# PC_O = PC_next: 00000014
# id_inst: 68a06293
# WRITE REGISTER FILE: x 2 = 00000231
# ex_regs_data1: xxxxxxxx
# ex_regs_data2: xxxxxxxx
# ex_imm: 00000568
# ex_alu_op: 3
# me_alu_o: 00000462
# wb_mem_data    : 00000000
# wb_alu_o       : 00000231
# wb_mem2reg     : 0
# wb_regs_write: 1
# ----------------------
```

```
# PC_O = PC_next: 00000018
# id_inst: 01606393
# WRITE REGISTER FILE: x 3 = 00000462
# ex_regs_data1: 00000000
# ex_regs_data2: xxxxxxxx
# ex_imm: 0000068a
# ex_alu_op: 3
# me_alu_o: 00000460
# wb_mem_data    : 00000000
# wb_alu_o       : 00000462
# wb_mem2reg     : 0
# wb_regs_write: 1
# ----------------------
# PC_O = PC_next: 0000001c
# id_inst: 007292b3
# WRITE REGISTER FILE: x 4 = 00000460
# ex_regs_data1: 00000000
# ex_regs_data2: xxxxxxxx
# ex_imm: 00000016
# ex_alu_op: 3
# me_alu_o: 0000068a
# wb_mem_data    : 00000000
# wb_alu_o       : 00000460
# wb_mem2reg     : 0
# wb_regs_write: 1
# forwardA! OP_A: 0000068a
# forwardA! ex_hazard: 0, mem_hazard: 1
# ----------------------
# PC_O = PC_next: 00000020
# id_inst: 0193c393
# WRITE REGISTER FILE: x 5 = 0000068a
# ex_regs_data1: xxxxxxxx
# ex_regs_data2: xxxxxxxx
# ex_imm: 00000000
# ex_alu_op: 2
# me_alu_o: 00000016
# wb_mem_data    : 00000000
# wb_alu_o       : 0000068a
# wb_mem2reg     : 0
# wb_regs_write: 1
```

forwardA! OP_A: 00000016
forwardA! ex_hazard: 0, mem_hazard: 1

……

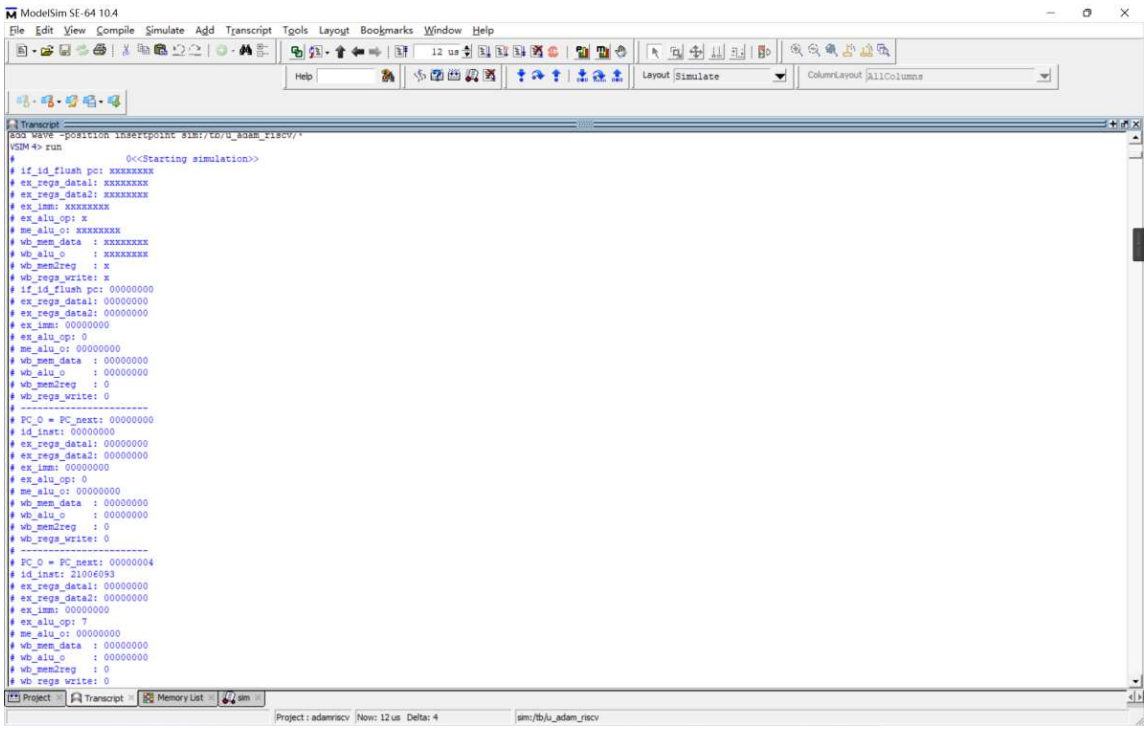

图 15 modelsim 仿真输出截图

（3）存储器结果(以运行 test2.s 为例)

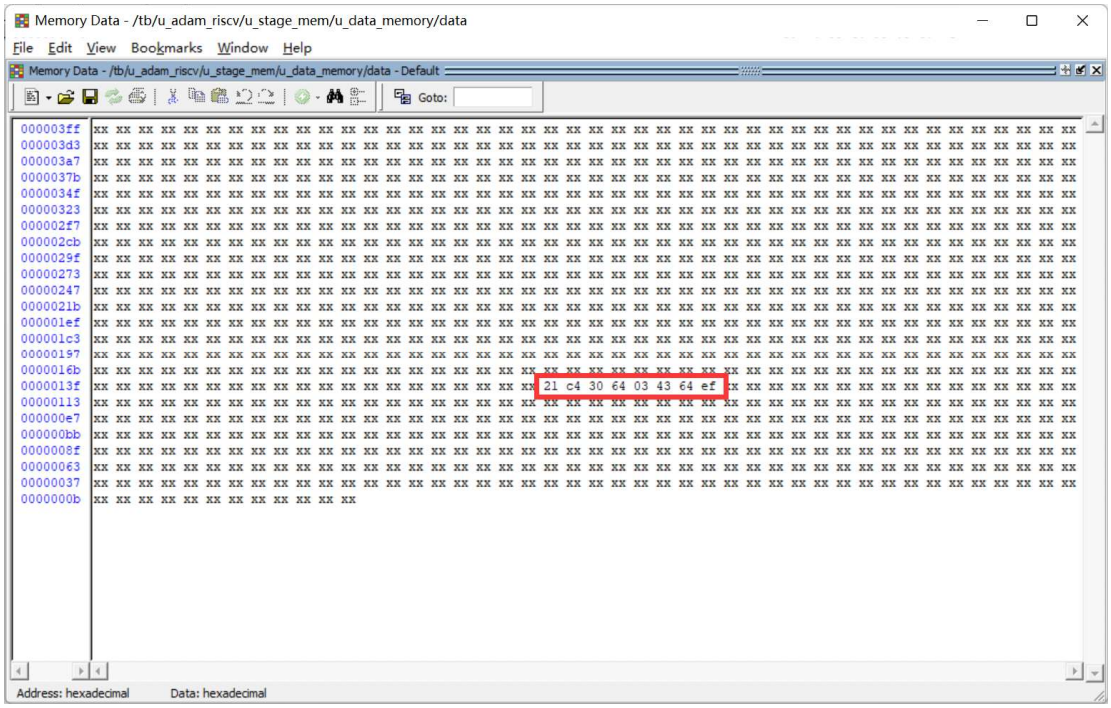

图 16 modelsim data memory 截图

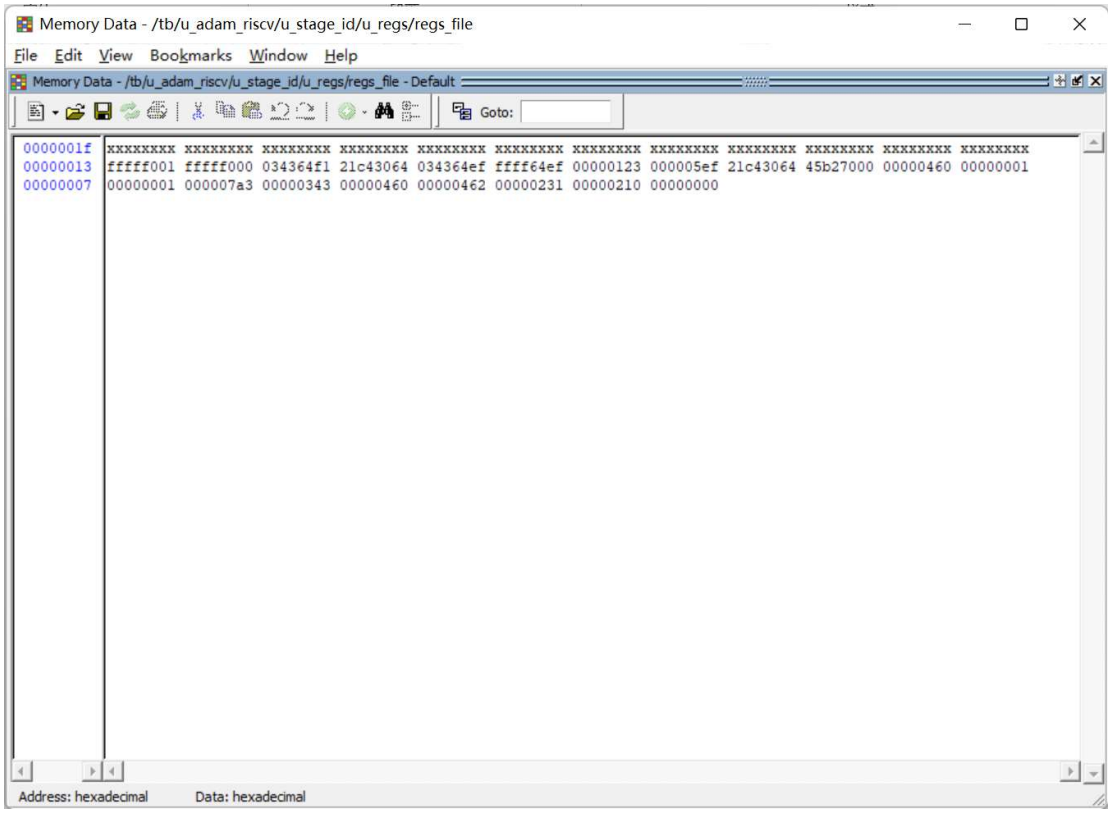

图 17 modelsim regs 截图

(4)仿真波形示意

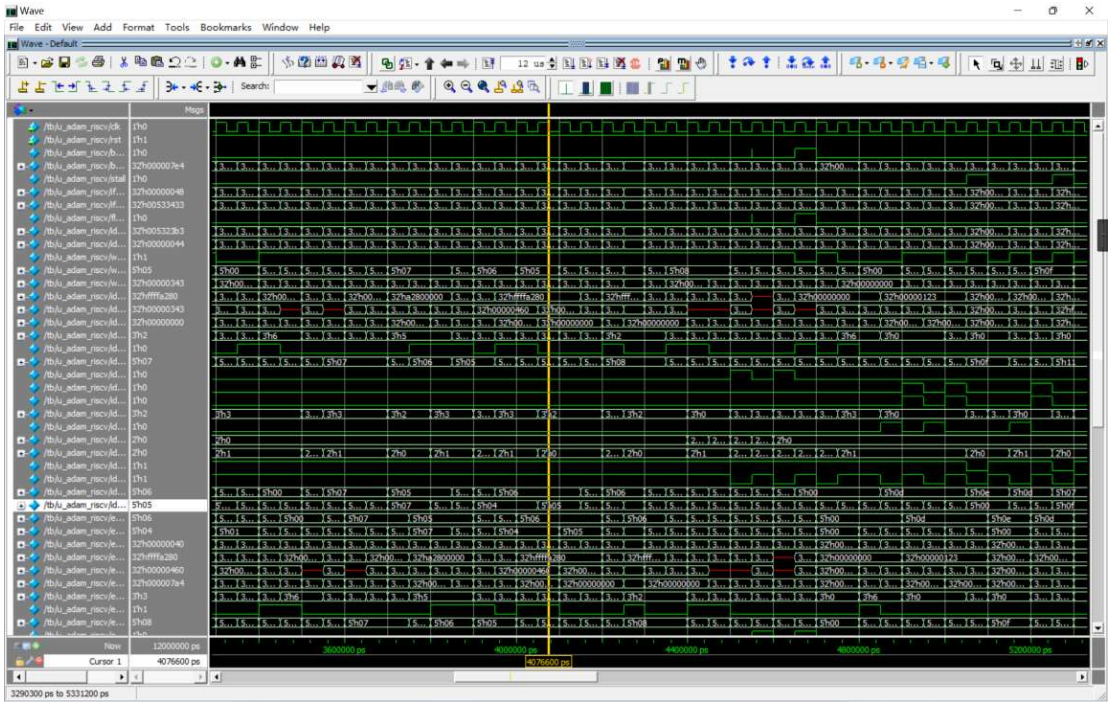

图 18 modelsim 仿真波形图截图

3. 4 验证结果

经过上述验证过程，可得本项目能实现对 RV32I 37 条指令的全部兼容，同时实现 load 指令数据相关仅需停顿一拍，其余数据相关不需停顿流水线，分支

跳转与无条件跳转留下两拍的空泡。下图为 test2.s 的运行结果与参考对照，可见处理器核运作符合预期。

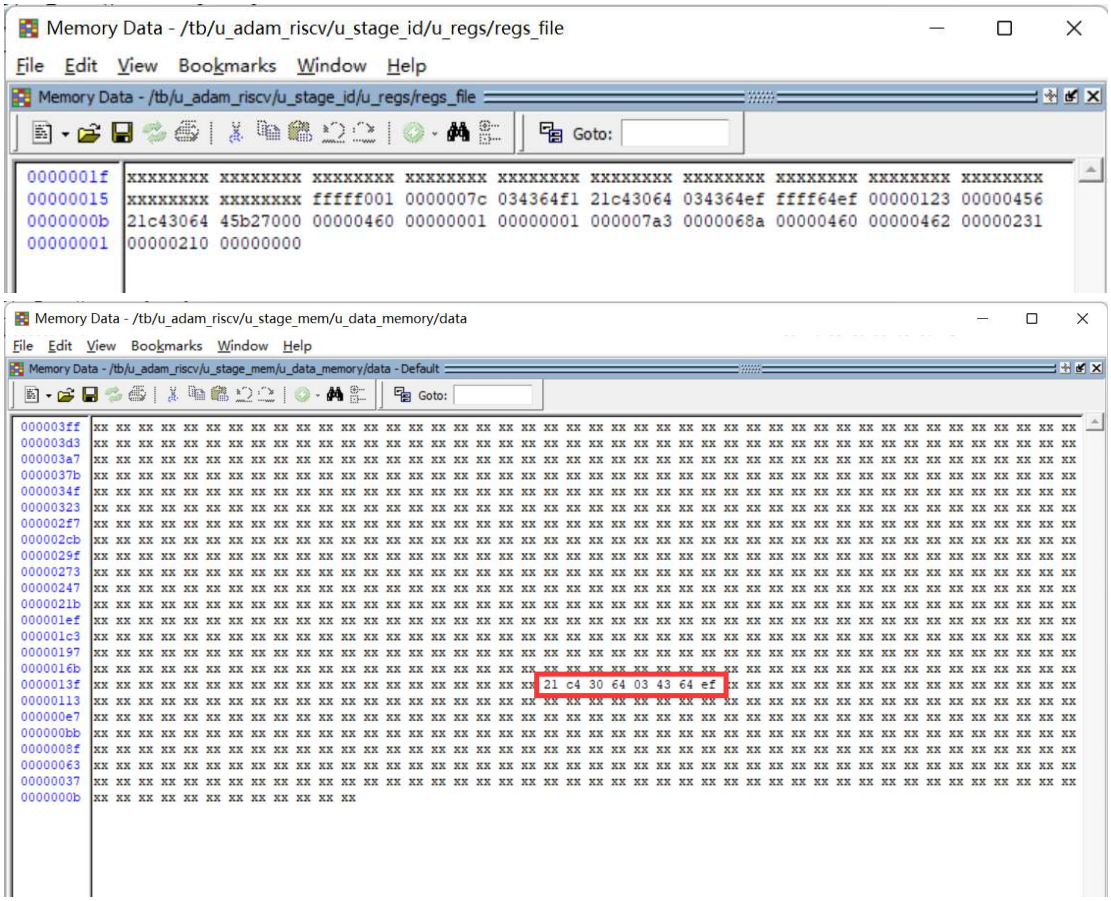

图 19 modelsim 存储器数据截图

golden.txt U

rom > golden.txt

```
1   # x0 = 0
2   # x1 = 0210
3   # x2 = 0x231
4   # x3 = 0x462
5   # x4 = 0x460
6   # x5 = 0x343
7   # x6 = 0x7a3
8   # x7 = 1
9   # x8 = 1
10  # x9 = 0x460
11  # x10 = 0x45b27000
12  # x11 = 0x21c43064
13  # x12 = 0x5ef
14  # x13 = 0x123
15  # x14 = 0xffff64ef
16  # x15 = 0x034364ef
17  # x16 = 0x21c43064
18  # x17 = 0x034364f1
19  # x18 = 0x7c
20  # x19 = 0xfffff001
21  # mem 0x128: 0x21c43064
22  # mem 0x124: 0x034364ef
```

图 20 test2.s 运行结果对照标准