

武汉大学计算机学院 本科生课程设计报告

RISC-V CPU 设计

专业名称：计算机科学与技术(弘毅学堂)

课程名称：计算机系统综合设计

指导教师：蔡朝晖 副教授

陈伟清 高级实验师

学生学号：2020300004071

学生姓名：王骏峣

二〇二一年七月

郑重声明

本人呈交的实验报告，是在指导老师的指导下，独立进行实验工作所取得的成果，所有数据、图片资料真实可靠。尽我所知，除文中已经注明引用的内容外，本实验报告不包含他人享有著作权的内容。对本实验报告做出贡献的其他个人和集体，均已在文中以明确的方式标明。本实验报告的知识产权归属于培养单位。

本人签名：_____ 日期：_____

摘要

本实验的实验目的是完成基于 RISC-V 架构 CPU 的设计，使其支持 37 条简单指令集，在五级流水线下能够正常处理指令、避免数据冒险和控制冒险。以及通过 ISE 软件将相关内容综合到 SoC 系统并导入 FPGA 开发板，并完成开发板上基于该 CPU 应用程序的实现。

实验设计主要遵循《计算机组成与设计：硬件/软件接口 RISC-V（第五版）》[1] 中流水线架构 CPU 各模块的输入、输出端口，功能信号的规定以及 RISC-V 指令集中各个指令对应的数据周转流程与对应信号。最终设计成功与否取决于仿真的波形是否与理论波形一致，以 ModelSim 软件仿真结果作为参考，观察在 SWORD 平台上的运行结果。

实验内容主要包括：使用 Verilog 语言在提供的 xgriscv 文件的结构基础上修改并拓展各个模块的功能代码，使得 CPU 能正确执行 RISC-V 精简指令集：{add, sub, and, or, lw, sw, beq, jalr, jal, ori, xor, xori, andi, addi, sll, sra, srl, slt, sltu, srai, slti, sltiu, slli, srli, lui, lb, lh, lbu, lhu, sb, sh, bne, blt, bge, bltu, bgeu, auipc}（注：数据在内存中以小端形式存储）。使用 Venus 平台编写并运行测试用的汇编代码并将其编译为机器码，在 Modelsim SE 10.4 环境中使用项目文件对 Venus 生成的机器码在设计的 CPU 架构上进行仿真，核对仿真产生的波形与寄存器的值和 Venus 平台执行结果是否相同。同时使用流水线寄存器、旁路单元、冒险控制单元等机制处理数据冒险、控制冒险。最后在 Xilinx ISE 14.7 环境中完成对总线、外设等模块的综合连线，生成 SWORD 设备可以读取的比特流，导入设备后测试使用情况。

实验结论为：仿真产生的波形和寄存器的值与 Venus 执行结果相同，控制冒险与数据冒险被解决，导入 SWORD 开发板后可以正常运行并显示内容，证明该流水线 CPU 设计成功。更改综合连线后可以支持键盘、VGA 等外设，说明该 CPU 可以支持更广泛的应用。

关键词：CPU；RISC-V；流水线架构；VGA

目 录

1 引言	8
1.1 实验目的	8
1.2 国内外研究现状	8
2 实验环境介绍	10
2.1 Verilog HDL	10
2.2 Venus	10
2.3 ModelSim	10
2.4 Xilinx ISE	10
2.5 SWORD	11
3 概要设计 - 单周期	12
3.1 总体设计	12
3.2 PC (程序计数器)	12
3.2.1 功能描述	12
3.2.2 模块接口	13
3.3 RF (寄存器文件)	13
3.3.1 功能描述	13
3.3.2 模块接口	13
3.4 Ctrl (控制模块)	14
3.4.1 功能描述	14
3.4.2 模块接口	14
3.5 ALU (运算单元)	14
3.5.1 功能描述	14
3.5.2 模块接口	15
3.6 EXT (符号扩展)	15
3.6.1 功能描述	15

3.6.2	模块接口	15
3.7	DM (数据存储器)	15
3.7.1	功能描述	15
3.7.2	模块接口	16
3.8	IM (指令存储器)	16
3.8.1	功能描述	16
3.8.2	模块接口	16
3.9	NPC (下一指令地址)	16
3.9.1	功能描述	16
3.9.2	模块接口	17
4	详细设计 - 单周期	18
4.1	CPU 总体结构	18
4.2	PC (程序计数器)	21
4.3	RF (程序计数器)	21
4.4	Ctrl (控制模块)	22
4.5	ALU (运算单元)	25
4.6	EXT (符号扩展)	26
4.7	NPC (下一指令地址)	27
5	概要设计 - 流水线	28
5.1	总体设计	28
5.2	PC (程序计数器)	29
5.2.1	功能描述	29
5.2.2	模块接口	29
5.3	RF (寄存器文件)	30
5.3.1	功能描述	30
5.3.2	模块接口	30
5.4	Ctrl (控制模块)	30
5.4.1	功能描述	30
5.4.2	模块接口	31
5.5	ALU (运算单元)	31

5.5.1	功能描述	31
5.5.2	模块接口	32
5.6	EXT (符号扩展)	32
5.6.1	功能描述	32
5.6.2	模块接口	32
5.7	DM (数据存储器)	33
5.7.1	功能描述	33
5.7.2	模块接口	33
5.8	IM (指令存储器)	33
5.8.1	功能描述	33
5.8.2	模块接口	33
5.9	flop (流水线寄存器)	34
5.9.1	功能描述	34
5.9.2	模块接口	34
5.10	hazard (冒险探测)	34
5.10.1	功能描述	34
5.10.2	模块接口	35
5.11	forward (旁路前递)	36
5.11.1	功能描述	36
5.11.2	模块接口	37
6	详细设计 - 流水线	38
6.1	CPU 总体结构	38
6.2	datapath (数据通路)	40
6.3	PC (程序计数器)	45
6.4	RF (寄存器文件)	46
6.5	Ctrl (控制模块)	47
6.6	ALU (运算单元)	49
6.7	EXT (符号扩展)	51
6.8	flop (流水线寄存器)	52
6.9	hazard (冒险探测)	52
6.10	forward (旁路前递)	53

7 测试及结果分析	54
7.1 仿真代码及分析	54
7.2 仿真测试结果	54
7.2.1 数据冒险	54
7.2.2 控制冒险	58
7.3 下载测试代码及分析	58
7.4 下载测试结果	59
8 实验总结	59
8.1 实验总结	59
8.2 取得的收获	61

1 引言

1.1 实验目的

本实验的实验目的是融会贯通计算机组成与设计课程所教授的知识，通过对知识的综合应用，加深对 CPU 系统各模块的工作原理及相互联系的认识。

学习采用 EDA (Electronic Design Automation) 技术设计 RISC-V 单周期 CPU/流水线 CPU 的技术与方法。

培养科学的研究的独立工作能力，取得 CPU 设计与仿真的实践和经验。

了解 SOC 系统，并在 FPGA 开发板上实现简单的 SOC 系统。

最终完成基于 RISC-V 架构 CPU 的设计，使其支持 37 条简单指令集，在五级流水线下能够正常处理指令、避免数据冒险和控制冒险。以及通过 ISE 软件将相关内容综合到 SoC 系统并导入 FPGA 开发板，并完成开发板上基于该 CPU 应用程序的实现。

1.2 国内外研究现状

RISC-V 处理器架构是由美国加州大学伯克利分校的 Krste Asanovic 教授等开发人员于 2010 年开发出来的。

伯克利的开发人员之所以发明一套新的指令集架构，而不是使用成熟的 x86 或者 ARM 架构，是因为这些架构经过多年的发展变得极为复杂和冗繁，存在着高昂的专利和架构授权问题，修改 ARM 处理器的 RTL 代码是不被支持的，而 x86 处理器的源代码不可能得到，并且其他的开源架构（譬如 SPARC、OpenRISC）均有或多或少的问题。[\[3\]](#)

目前，有多种基于 RISC-V 的开源处理器和开源 SoC 已投入使用或处于研发进程之中，主要有：标量处理器——Rocket、超标量乱序执行处理器——BOOM、处理器家族——SHAKTI 等，以及 Rocket-Chip、Rocket-Chip 等 SoC[\[5\]](#)，还有各类嵌入式处理器和多处理器架构等。RISC-V 在处理器方面有着广泛的可用武之地。

国产 CPU 在生产、封装、测试等方面还存在着一定的困难，整体性能上与国际领先水平还有一定的差距。例如主频，国产 CPU 最高可达到 1.6GHz，而 Intel 和 AMD 主流处理器的主频已超过 4GHz。[\[2\]](#)

国内晶心科技、杭州中天微、武汉芯来公司分别开发了一系列商用或开源的 RISC-V 架构芯片。2019 年，华米科技宣布全球首颗可穿戴、RISC-V 开源指令集芯片——黄山 1 号正式量产应用。[\[4\]](#)

微处理器最早在国外研发，因此国内在处理器关键技术上有一定落后。当今时代集成电路飞快发展，基于开源的 RISC-V 架构进行处理器的研究与设计是我国追上西方国家芯片技术的关键。尤其在对软件生态依赖较低的嵌入式领域，进行低功耗、高性能的嵌入式 RISC-V 处理器的研究具有重要意义。[\[6\]](#)

因此，国内外对 RISC-V 以及流水线处理器的研究仍处于上升和持续发展阶段，研究 RISC-V 和流水线处理器将在物联网和各类嵌入式扩展领域得到较为广泛的应用。

2 实验环境介绍

2.1 Verilog HDL

Verilog HDL (Hardware Description Language) 是一种设计硬件电路的语言，由 IEEE 完成了对其标准化的工作。Verilog HDL 总体来讲带有 C 语言的风格，是工业界常用的硬件描述语言。另一种具有 C++ 风格的硬件描述语言是 VHDL，也有 IEEE 的标准。比这两种语言层次更高的是 SystemC，一种系统级的描述语言。
[7]

Verilog HDL 语言可以用来进行数字电路的仿真验证、时序分析、逻辑综合。Verilog HDL 模型既可以用电路的功能描述，也可以用元器件及其之间的连接来建立。

2.2 Venus

Venus 是伯克利大学为 CS61C: Great ideas in Computer Architecture 课程开发的辅助教学工具，主要有将 RISC-V 指令和机器码互相转换、运行 RISC-V 指令、单步调试 RISC-V 指令和即时查看寄存器和内存的值。

使用版本为：RISC-V Disassembler v1.0.1。

2.3 ModelSim

ModelSim 是对 Verilog HDL 语言设计出的程序代码进行仿真的程序。它可以通过调用测试代码将设计出的程序单步调试、运行、重新运行等，同时也可以在仿真运行过程中实时查看内存、寄存器、连线上的值。

使用版本为：ModelSim SE 10.4。

2.4 Xilinx ISE

Xilinx ISE 是由 Xilinx 公司开发的硬件设计工具，有输入、综合、实现、验证、下载等功能。本实验主要使用 ISE 软件进行综合、验证、下载，最终生成比特流并将其导入 SWORD 开发板。

使用版本为：Xilinx ISE 14.7。

2.5 SWORD

SWORD 全称 Simple While Organic aRc Design，它不仅仅是一种单纯的硬件，也不仅仅是一种处理器架构的实现，而是一种计算机系统能力培养方法。其内容涵盖了从数字逻辑硬件设计，到指令集架构设计与扩展，并延伸到编译器设计，甚至涵盖了操作系统设计与实现，及基于上述一切内容的计算机系统集成设计与应用。

使用的芯片：Xilinx Kintex™-7， XC7K160T-FFG676

3 概要设计 - 单周期

3.1 总体设计

CPU 由 DM (数据存储器)、IM (指令存储器)、RF (寄存器文件)、PC (程序计数器)、EXT (符号扩展)、ALU (运算单元)、NPC (控制下一指令地址)、Ctrl (控制模块) 等部分组成。其中 PC、EXT、ALU、NPC、RF、Ctrl 综合在 SCPU.v 文件中；DM、IM 作为存储器，独立于上述模块在 sccomp.v 中进行连线。所有模块单独作为一个 Verilog HDL 语言文件进行定义。

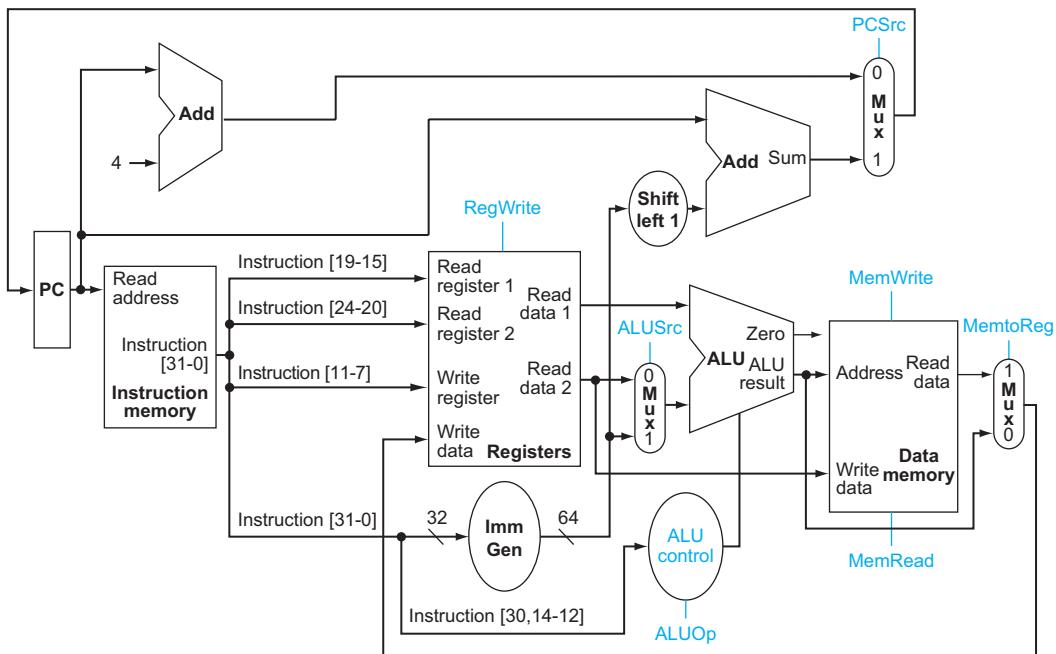


图 1: 单周期 CPU 概要原理图 [1]

此外,ctrl_encode_def.v文件进行各类控制信号和指令内容的宏定义,sccomp_tb.v中存放测试文件,进行仿真时的控制和调试。

3.2 PC (程序计数器)

3.2.1 功能描述

根据输入信号（包括输入数据以及控制信号）输出程序需要执行的指令地址，并向 IM（指令存储器）输出这个地址。该值在每个指令周期用 NPC 模块的

输出进行更新。

3.2.2 模块接口

信号名	方向	描述
clk	input	时钟信号
rst	input	复位信号
NPC	input	来自 NPC 模块的控制下一 PC 的输入
PC	output	输出的 PC 信号

3.3 RF（寄存器文件）

3.3.1 功能描述

控制 32 个寄存器的读写操作。在任意时刻都可以根据输入 A1[4:0] 和 A2[4:0] 的值读取指定编号的寄存器的值，并赋值给 RD1[31:0], RD2[31:0] 作为输出。当时钟处在上升沿时，并且写寄存器信号 RFWr 有效，由 A3[4:0] 的值决定向指定编号的寄存器写入 WD[31:0] 的值。

3.3.2 模块接口

信号名	方向	描述
clk	input	时钟信号
rst	input	复位信号
RFWr	input	寄存器写信号
A1[4:0]	input	读寄存器的第一个地址
A2[4:0]	input	读寄存器的第二个地址
A3[4:0]	input	写寄存器的地址
WD[31:0]	input	写入寄存器的数据
RD1[31:0]	output	第一个读寄存器的数据
RD2[31:0]	output	第二个读寄存器的数据

3.4 Ctrl (控制模块)

3.4.1 功能描述

根据输入的指令，控制其他模块的使能信号和选择信号。

3.4.2 模块接口

信号名	方向	描述
Op [6:0]	input	指令中第 [6:0] 位
Funct7 [6:0]	input	指令中第 [31:25] 位
Funct3 [2:0]	input	指令中第 [14:12] 位
Zero	input	运算单元是否输出为 0
RegWrite	output	寄存器写使能
MemWrite	output	数据存储器写使能
EXTOp [5:0]	output	符号扩展运算信号
ALUOp [4:0]	output	运算单元运算信号
NPCOp [2:0]	output	控制下一地址模块运算信号
ALUSrc	output	运算单元来源选择器信号
DMType [2:0]	output	选择数据存储器访问时的位数
GPRSel [1:0]	output	备用，无特殊功能
WDSel [1:0]	output	选择写入寄存器的内容

3.5 ALU (运算单元)

3.5.1 功能描述

根据输入的信号，将输入的两个数进行运算得到结果后输出。

3.5.2 模块接口

信号名	方向	描述
signed A[31:0]	input	需要运算的第一个数
signed B[31:0]	input	需要运算的第二个数
ALUOp[4:0]	input	控制运算类型的信号
PC[31:0]	input	传入当前 PC 信号，便于调试
signed C[31:0]	output	运算后输出的结果
Zero	output	运算结果是否为 0

3.6 EXT (符号扩展)

3.6.1 功能描述

根据输入的信号，将输入的立即数进行移位或符号扩展运算得到结果后输出到运算单元或 NPC 单元。

3.6.2 模块接口

信号名	方向	描述
iimm_shamt[4:0]	input	指令中第 [24:20] 位，用于特殊的 I 型指令
iimm[11:0]	input	指令中第 [31:20] 位，用于 I 型指令
simmm[11:0]	input	指令中第 [31:25, 11:7] 位，用于 S 型指令
bimm[11:0]	input	指令中第 [31, 7, 30:25, 11:8] 位，用于 SB 型指令
uimm[19:0]	input	指令中第 [31:12] 位，用于 U 型指令
jimm[19:0]	input	指令中第 [31, 19:12, 20, 30:21] 位，用于 J 型指令
EXTOp[5:0]	input	控制 EXT 模块运算类型
immout[31:0]	output	EXT 模块的输出结果

3.7 DM (数据存储器)

3.7.1 功能描述

根据输入的信号和指令，将相应内存地址的数据写入或取出。写入只在时钟上升沿发生，取出可以是任意时候的。同时也支持取出低 1 字节、低 2 字节、低 4

字节。数据在内存中用小端法存储，一个地址储存 1 个字节的数据。内存中预置了 512 个字节空间。在具体结合 SoC 系统在 Xilinx ISE 中实现时，没有该部分。

3.7.2 模块接口

信号名	方向	描述
clk	input	时钟信号
DMWr	input	写使能信号
addr [8:0]	input	需要访问的地址
din [31:0]	input	需要输入的数据
DMType [2:0]	input	控制输入/取出数据的字节数
dout [31:0]	output	从访问地址读出的数据

3.8 IM（指令存储器）

3.8.1 功能描述

根据输入的信号和地址，将相应指令地址的数据取出，送往后续控制器等模块。在具体结合 SoC 系统在 Xilinx ISE 中实现时，没有该部分。

3.8.2 模块接口

信号名	方向	描述
addr [8:2]	input	输入的指令地址
dout [31:0]	output	从指令地址读出的数据

3.9 NPC（下一指令地址）

3.9.1 功能描述

根据输入的信号、输入的立即数和当前指令地址 PC，计算下一指令地址，在下一时钟周期送往 PC 计数器。

3.9.2 模块接口

信号名	方向	描述
PC[31:0]	input	当前的指令地址
NPCOp[2:0]	input	选择下一指令地址的控制信号
IMM[31:0]	input	输入的立即数
aluout[31:0]	input	当前运算单元的输出
NPC[31:0]	output	下一时钟周期的 PC 地址

4 详细设计 - 单周期

4.1 CPU 总体结构

依据教材《计算机组成与设计：软/硬件接口 RISC-V（第五版）》[1] 中单周期架构 CPU 各模块的输入、输出端口，功能信号的规定以及 RISC-V 指令集中各个指令对应的数据周转流程与对应信号。模块中所有连线基于该架构进行实现。

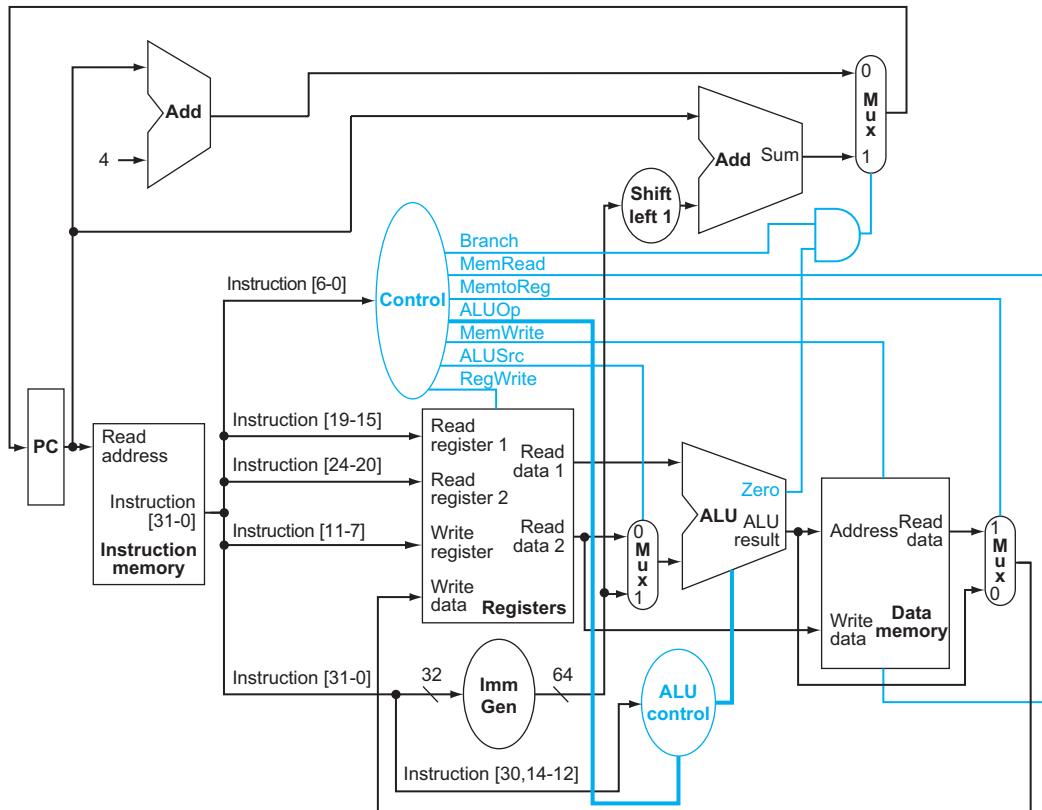


图 2: 单周期 CPU 原理图 [1]

用 SCPU.v 文件将 Ctrl、PC、NPC、EXT、RF、ALU 整合到同一文件中；然后再用 sccomp.v 挂载 DM、IM 文件和 SCPU 进行交互。

此处将 DM、IM 文件独立的原因是：在 SoC 系统中，DM 和 IM 分别用 Xilinx ISE 内置的 IP core，预置内容后直接按要求导入模块。

SCPU.v 设计如下：（省略连线，详见附件中.v 文件）

```
1 `include "ctrl_encode_def.v"
```

```

2 module SCPU(
3     input      clk,           // 时钟信号
4     input      reset,         // 复位信号
5     input [31:0] inst_in,    // 输入指令
6     input [31:0] Data_in,    // 来自 DM 的数据
7
8     output     mem_w,        // DM 写使能
9     output [31:0] PC_out,    // 输出 PC 地址 ( debug 用 )
10
11    output [31:0] Addr_out,  // ALU 输出值 (一般用于计算地址)
12    output [31:0] Data_out,  // 输入 DM 的数据
13    output [2:0] DMTypE     // 选择访问 DM 的字节数
14 );
15
16    assign Addr_out=aluout;
17    assign B = (ALUSrc) ? immout : RD2; // ALU 第二个操作数的来源
18    assign Data_out = RD2;
19
20    assign iimm_shamt=inst_in[24:20]; //UJ 型指令
21    assign iimm=inst_in[31:20];    //I 型指令
22    assign simm={inst_in[31:25],inst_in[11:7]}; //S 型指令
23    assign bimm={inst_in[31],inst_in[7],inst_in[30:25],inst_in
24                  [11:8]}; //SB 型
25    assign uimm=inst_in[31:12];   //U 型
26    assign jimm={inst_in[31],inst_in[19:12],inst_in[20],inst_in
27                  [30:21]}; //J 型
28
29    assign Op = inst_in[6:0];    // opcode 部分
30    assign Funct7 = inst_in[31:25]; // funct7 部分
31    assign Funct3 = inst_in[14:12]; // funct3 部分
32    assign rs1 = inst_in[19:15];  // rs1 序号
33    assign rs2 = inst_in[24:20];  // rs2 序号
34    assign rd = inst_in[11:7];   // rd 序号
35    assign Imm12 = inst_in[31:20];
36    assign IMM = inst_in[31:12];
37
38    // 各模块的例化

```

```

36   ctrl U_ctrl(.Op(Op), .Funct7(Funct7), .Funct3(Funct3), .Zero(
37     Zero), .RegWrite(RegWrite), .MemWrite(mem_w),
38     .EXTOp(EXTOp), .ALUOp(ALUOp), .NPCOp(NPCOp),
39     .ALUSrc(ALUSrc), .GPRSel(GPRSel), .WDSel(WDSel), .DMType(
40       DMType) );
41   PC U_PC(.clk(clk), .rst(reset), .NPC(NPC), .PC(PC_out) );
42   NPC U_NPC(.PC(PC_out), .NPCOp(NPCOp), .IMM(immout), .NPC(NPC),
43     .aluout(aluout) );
44   EXT U_EXT(.iimm_shamt(iimm_shamt), .iimm(iimm), .simm(simmm), .
45     bimm(bimm), .uimm(uimm), .jimm(jimm),
46     .EXTOp(EXTOp), .immout(immout) );
47   RF U_RF(.clk(clk), .rst(reset), .RFWr(RegWrite),
48     .A1(rs1), .A2(rs2), .A3(rd), //Read1, Read2, Write
49     .WD(WD), //Write data
50     .RD1(RD1), .RD2(RD2) //Read1 Read2
51   );
52   alu U_alu(.A(RD1), .B(B), .ALUOp(ALUOp), .C(aluout), .Zero(
53     Zero), .PC(PC_out) );
54
55 always @*
56 begin
57   case(WDSel)
58     `WDSel_FromALU: WD<=aluout;
59     `WDSel_FromMEM: WD<=Data_in;
60     `WDSel_FromPC: WD<=PC_out+4;
61   endcase
62 end
63 endmodule

```

sccomp.v 设计如下：（省略连线，详见附件中.v文件）

```

1 module sccomp(clk, rstn, reg_sel, reg_data);
2   input          clk;
3   input          rstn;
4   input [4:0]    reg_sel;
5   output [31:0]  reg_data;
6
7 // CPU 除存储器外部分的例化

```

```

8 SCPU U_SCPU(.clk(clk), .reset(rst), .inst_in(instr), .Data_in(
9   dm_dout),
10  .mem_w(MemWrite), .PC_out(PC), .Addr_out(dm_addr),
11  .Data_out(dm_din), .reg_sel(reg_sel), .reg_data(reg_data), .
12    DMTyp(DMTyp) );
13
14 // 存储器的例化
15 dm U_DM(.clk(clk), DMWr(MemWrite), .addr(dm_addr),
16 .din(dm_din), .DMType(DMTyp), .dout(dm_dout));
17 // 这里 PC 截掉最低 2 位, 保证 PC 总为 4 的倍数
18
19 endmodule

```

4.2 PC (程序计数器)

```

1
2 module PC( clk, rst, NPC, PC );
3   input           clk;
4   input           rst;
5   input [31:0]    NPC;
6   output reg [31:0] PC;
7
8   always @ (posedge clk, posedge rst)
9     if (rst) // 复位信号
10       PC <= 32'h0000_0000;
11     else
12       PC <= NPC;
13
14 endmodule

```

4.3 RF (程序计数器)

```

2 module RF( input          clk,
3             input          rst,
4             input          RFWr,
5             input [4:0]   A1, A2, A3,
6             input [31:0]  WD,
7             output [31:0] RD1, RD2);
8
9 reg [31:0] rf[31:0];
10
11 integer i;
12
13 always @ (posedge clk, posedge rst)
14     if (rst) begin // 初始化
15         for (i=1; i<32; i=i+1)
16             rf[i] <= 0; // i;
17     end
18
19 else
20     if (RFWr&&A3) begin
21         rf[A3] <= WD;
22         // $display ("r[00-07]=0x%8X, 0x%8X, 0x%8X, 0x%8X, 0x%8X
23         // , 0x%8X, 0x%8X, 0x%8X", 0, rf[1], rf[2], rf[3], rf[4], rf
24         [5], rf[6], rf[7]);
25         // 此处做 debug 用，省略其他 display 语句
26     end
27
28 assign RD1 = (A1 != 0) ? rf[A1] : 0;
29 assign RD2 = (A2 != 0) ? rf[A2] : 0;
30
31 endmodule

```

4.4 Ctrl（控制模块）

Ctrl 根据预置的控制信号，如表3所示，依据输入的指令进行相应类型判断和信号输出。

附件中 `ctrl_encode_def.v` 也依照下表进行定义。

RV32I Base Instruction Set

imm[31:12]			rd	0110111	LUI
imm[31:12]			rd	0010111	AUIPC
imm[20 10:1 11 19:12]			rd	1101111	JAL
imm[11:0]	rs2	rs1	000	rd	JALR
imm[12 10:5]			000	imm[4:1 11]	BEQ
imm[12 10:5]	rs2	rs1	001	imm[4:1 11]	BNE
imm[12 10:5]	rs2	rs1	100	imm[4:1 11]	BLT
imm[12 10:5]	rs2	rs1	101	imm[4:1 11]	BGE
imm[12 10:5]	rs2	rs1	110	imm[4:1 11]	BLTU
imm[12 10:5]	rs2	rs1	111	imm[4:1 11]	BGEU
imm[11:0]		rs1	000	rd	LB
imm[11:0]		rs1	001	rd	LH
imm[11:0]		rs1	010	rd	LW
imm[11:0]		rs1	100	rd	LBU
imm[11:0]		rs1	101	rd	LHU
imm[11:5]	rs2	rs1	000	imm[4:0]	SB
imm[11:5]	rs2	rs1	001	imm[4:0]	SH
imm[11:5]	rs2	rs1	010	imm[4:0]	SW
imm[11:0]		rs1	000	rd	ADDI
imm[11:0]		rs1	010	rd	SLTI
imm[11:0]		rs1	011	rd	SLTIU
imm[11:0]		rs1	100	rd	XORI
imm[11:0]		rs1	110	rd	ORI
imm[11:0]		rs1	111	rd	ANDI
0000000	shamt	rs1	001	rd	SLLI
0000000	shamt	rs1	101	rd	SRLI
0100000	shamt	rs1	101	rd	SRAI
0000000	rs2	rs1	000	rd	ADD
0100000	rs2	rs1	000	rd	SUB
0000000	rs2	rs1	001	rd	SLL
0000000	rs2	rs1	010	rd	SLT
0000000	rs2	rs1	011	rd	SLTU
0000000	rs2	rs1	100	rd	XOR
0000000	rs2	rs1	101	rd	SRL
0100000	rs2	rs1	101	rd	SRA
0000000	rs2	rs1	110	rd	OR
0000000	rs2	rs1	111	rd	AND
fm	pred	succ	rs1	000	FENCE
00000000000000			00000	000	ECALL
00000000000001			00000	000	EBREAK

图 3: RISC-V 指令集

```

1 module ctrl(Op, Funct7, Funct3, Zero, RegWrite, MemWrite, EXTOp,
2             ALUOp, NPCOp, ALUSrc, GPRSel, WDSel, DMTypE );
3
4     input [6:0] Op;           // opcode
5     input [6:0] Funct7;
6     input [2:0] Funct3;
7     input      Zero;        // 运算器输出是否为 0
8
9     output      RegWrite;   // 寄存器写使能
10    output      MemWrite;   // 存储器写使能
11    output [5:0] EXTOp;    // 符号扩展控制信号
12    output [4:0] ALUOp;    // 运算器控制信号
13    output [2:0] NPCOp;    // NPC 控制信号
14    output      ALUSrc;    // 运算器输入选择信号
15    output [2:0] DMTypE;   // 存储器字节数选择信号
16
17
18 // 省略根据指令判断类型部分代码，具体依照表 3 进行判断
19
20
21
22
23
24
25
26
27
28
29
30
31
32

```

```

33
34     assign WDSel[0] = itype_l;
35     assign WDSel[1] = i_jal | i_jalr;
36
37     assign NPCOp[0] = sbtype & Zero;
38     assign NPCOp[1] = i_jal;
39     assign NPCOp[2]=i_jalr;
40
41 // 控制运算器的运算符
42     assign ALUOp[0] = i_jall|i_jalr|itype_l|stype|i_addil|i_oril|
43                 i_add|i_or|i_bne|i_bge|i_bgeu|i_sltiu|i_sltu|i_slli|i_sll|
44                 i_sra|i_srai|i_lui;
45     assign ALUOp[1] = i_jall|i_jalr|itype_l|stype|i_addil|i_addl|
46                 i_and|i_andil|i_auiopc|i_blt|i_bge|i_slt|i_slti|i_sltiu|
47                 i_sltu|i_slli|i_sll;
48     assign ALUOp[2] = i_andil|i_and|i_ori|i_or|i_beq|i_sub|i_bne|
49                 i_bltl|i_bge|i_xor|i_xori|i_sll|i_slli;
50     assign ALUOp[3] = i_andil|i_and|i_ori|i_or|i_bltu|i_bgeu|i_sltl|
51                 i_slti|i_sltiu|i_sltu|i_xor|i_xori|i_sll|i_slli;
52     assign ALUOp[4] = i_srl|i_sra|i_srlil|i_srai;
53
54 endmodule

```

4.5 ALU (运算单元)

```

1 `include "ctrl_encode_def.v"
2 module alu(A, B, ALUOp, C, Zero, PC);
3     input signed [31:0] A, B;
4     input [4:0] ALUOp;
5     input [31:0] PC;
6     output signed [31:0] C;
7     output Zero;
8
9     reg [31:0] C; // 输出值暂存在寄存器里
10
11    always @( * ) begin

```

```

12     case ( ALUOp )
13         `ALUOp_nop:C=A;
14         `ALUOp_lui:C=B;
15         `ALUOp_auipc:C=PC+B;
16         `ALUOp_add:C=A+B;
17         `ALUOp_sub:C=A-B;
18         `ALUOp_bne:C={31'b0,(A==B)};
19         `ALUOp_blt:C={31'b0,(A>=B)};
20         `ALUOp_bge:C={31'b0,(A<B)};
21         `ALUOp_bltu:C={31'b0,($unsigned(A)>=$unsigned(B))};
22         // 备注：无符号类型需要先将 wire 转为 unsigned
23         `ALUOp_bgeu:C={31'b0,($unsigned(A)<$unsigned(B))};
24         `ALUOp_slt:C={31'b0,(A<B)};
25         `ALUOp_sltu:C={31'b0,($unsigned(A)<$unsigned(B))};
26         `ALUOp_xor:C=A^B;
27         `ALUOp_or:C=A|B;
28         `ALUOp_and:C=A&B;
29         `ALUOp_sll:C=A<<B;
30         `ALUOp_srl:C=A>>B;
31         `ALUOp_sra:C=A>>>B;
32     endcase
33 end
34
35 assign Zero = (C == 32'b0);
36
37 endmodule

```

4.6 EXT (符号扩展)

```

1 `include "ctrl_encode_def.v"
2 module EXT(
3     input [4:0] iimm_shamt,
4     input [11:0]      iimm,
5     input [11:0]      simm,
6     input [11:0]      bimm,
7     input [19:0]      uimm,

```

```

8     input  [19:0]      jimm ,
9     input  [5:0]       EXTOp ,
10    output reg [31:0]   immout);
11
12    always @(*)
13    case (EXTOp)
14      `EXT_CTRL_ITYPE_SHAMT: immout<={27'b0,iimm_shamt[4:0]};
15      `EXT_CTRL_ITYPE: immout<={{{32-12}{iimm[11]}}, iimm[11:0]};
16      `EXT_CTRL_STYPE: immout<={{{32-12}{simm[11]}}, simm[11:0]};
17      `EXT_CTRL_BTYPY: immout<={{{32-13}{bimm[11]}}, bimm[11:0],
18          1'b0};
19      //忽略最低位，即乘 2，下同
20      `EXT_CTRL_UTYPE: immout<={uimm[19:0], 12'b0};
21      `EXT_CTRL_JTYPE: immout<={{{32-21}{jimm[19]}}, jimm[19:0],
22          1'b0};
23      default:           immout<=32'b0;
24    endcase
25
26 endmodule

```

4.7 NPC (下一指令地址)

```

1 `include "ctrl_encode_def.v"
2
3 module NPC(PC, NPCOp, IMM, NPC,aluout);
4
5     input [31:0] PC;
6     input [2:0]  NPCOp;
7     input [31:0] IMM;
8     input [31:0] aluout;
9     output reg [31:0] NPC;
10
11    wire [31:0] PCPLUS4;
12
13    assign PCPLUS4 = PC + 4;
14

```

```

15 // 根据不同控制信号进行下一 PC 指令的选择和计算
16
17     always @(*) begin
18         case (NPCOp)
19             `NPC_PLUS4:   NPC = PCPLUS4;
20             `NPC_BRANCH: NPC = PC+IMM;
21             `NPC_JUMP:    NPC = PC+IMM;
22             `NPC_JALR:   NPC =aluout;
23             default:      NPC = PCPLUS4;
24         endcase
25     end
26
27 endmodule

```

5 概要设计 - 流水线

5.1 总体设计

流水线 CPU 由 DM (数据存储器)、IM (指令存储器)、RF (寄存器文件)、PC (程序计数器)、EXT (符号扩展)、ALU (运算单元)、Ctrl (控制模块) 等部分组成，此外，将 EXT、流水线寄存器、冒险探测、旁路选择等模块单独放在 `parts.v` 文件中。PC、EXT、ALU、RF、流水线寄存器和冒险处理综合在 `datapath.v` 文件中；Ctrl 单独作为文件 `controller.v` 进行定义；DM、IM 作为存储器，独立于 CPU 文件，作为 Xilinx ISE 的 IP core 进行连线。

流水线被划分为 IF (取指), ID (译码), EX (运算), MEM (访存), WB (写回) 五个部分，每两个部分之间由流水线寄存器暂时存放数据。同时为了防止冒险，引入冒险探测、旁路选择，并实现了控制冒险的阻塞。由于同时存在 DM 和 IM，不会存在结构冒险。

此外，`xgriscv_defines.v` 文件进行各类控制信号和指令内容的宏定义。

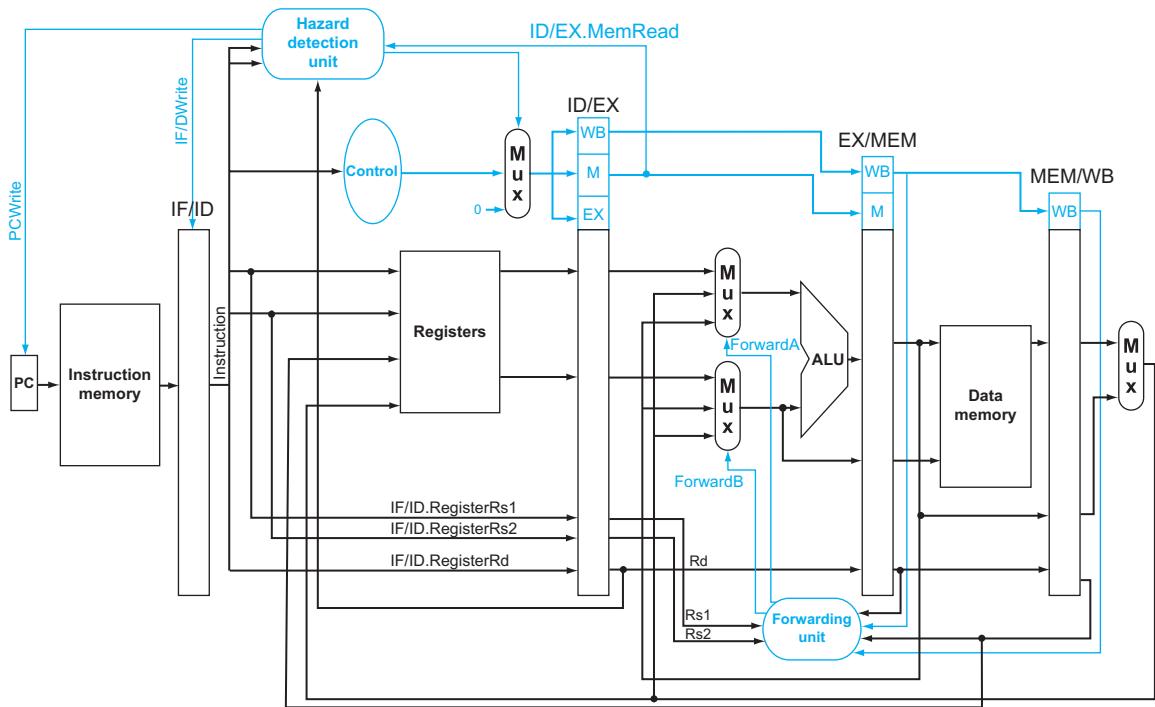


图 4: 流水线 CPU 概要原理图 [1]

5.2 PC (程序计数器)

5.2.1 功能描述

根据输入信号（包括输入数据以及控制信号）输出程序需要执行的指令地址，并向 IM（指令存储器）输出这个地址。该值在每个指令周期根据控制信号更新为 PC+4 或其他跳转后信号。

5.2.2 模块接口

信号名	方向	描述
clk	input	时钟信号
reset	input	复位信号
en	input	使能信号
d	input	输入作为下一 PC 的值
q	output	输出的 PC 信号

5.3 RF（寄存器文件）

5.3.1 功能描述

控制 32 个寄存器的读写操作。在任意时刻都可以根据输入 $A1[4:0]$ 和 $A2[4:0]$ 的值读取指定编号的寄存器的值，并赋值给 $RD1[31:0]$, $RD2[31:0]$ 作为输出。当时钟处在上升沿时，并且写寄存器信号 $RFWr$ 有效，由 $A3[4:0]$ 的值决定向指定编号的寄存器写入 $WD[31:0]$ 的值。

5.3.2 模块接口

信号名	方向	描述
clk	input	时钟信号
ra1[4:0]	input	读寄存器的第一个地址
ra2[4:0]	input	读寄存器的第二个地址
rd1[31:0]	output	第一个读寄存器的数据
rd2[31:0]	output	第二个读寄存器的数据
we3	input	寄存器写使能
wa3[4:0]	input	写寄存器的地址
wd3[31:0]	input	写入寄存器的数据
pc[31:0]	input	PC 值（用于 debug）

5.4 Ctrl（控制模块）

5.4.1 功能描述

根据输入的指令，控制其他模块的使能信号和选择信号。

5.4.2 模块接口

信号名	方向	描述
clk	input	时钟信号
reset	input	复位信号
opcode[6:0]	input	指令中第 [6:0] 位
funct7[6:0]	input	指令中第 [31:25] 位
funct3[2:0]	input	指令中第 [14:12] 位
rd[4:0]	input	用于判断 nop
rs1[4:0]	input	
imm[11:0]	input	
immctrl[4:0]	output	控制符号扩展模块运算信号
itype	output	判断是否为 I 型指令
jal	output	判断是否为 jal 指令
jalr	output	判断是否为 jalr 指令
bunsigned	output	判断是否为无符号的 B 型指令
lunsigned	output	判断是否为无符号的 load 类 I 型指令
pcsrc	output	控制下一个 PC 信号的来源
aluctrl[3:0]	output	控制第一个运算单元的运算类型
aluctrl1[2:0]	output	控制第二个运算单元的运算类型
alusrc1[1:0]	output	控制第一个数的来源
alusrc2	output	控制第二个数的来源
j	output	判断是否为 J 型指令
btype	output	判断是否为 SB 型指令
lwhb	output	load 类指令的 word, half, byte 判断
swhb	output	S 型指令的 word, half, byte 判断

5.5 ALU (运算单元)

5.5.1 功能描述

根据输入的信号，将输入的两个数进行运算得到结果后输出。为了方便计算跳转后的地址，在数据通路中引用了两个 ALU。

5.5.2 模块接口

信号名	方向	描述
signed a[31:0]	input	需要运算的第一个数
signed b[31:0]	input	需要运算的第二个数
aluctrl[3:0]	input	控制第一个 ALU 运算类型的信号
aluctrl1[2:0]	input	控制第二个 ALU 运算类型的信号
aluout[31:0]	output	运算后输出的结果
overflow	output	运算结果是否发生溢出
zero	output	运算结果是否为 0
lt	output	是否满足 $a < b$
ge	output	是否满足 $a \geq b$

5.6 EXT (符号扩展)

5.6.1 功能描述

根据输入的信号，将输入的立即数进行移位或符号扩展运算得到结果后输出到运算单元，用作取址或跳转指令。在 `part.v` 中作为 `imm` 模块实现。

5.6.2 模块接口

信号名	方向	描述
iimm[11:0]	input	指令中第 [31:20] 位，用于 I 型指令
simm[11:0]	input	指令中第 [31:25, 11:7] 位，用于 S 型指令
bimm[11:0]	input	指令中第 [31, 7, 30:25, 11:8] 位，用于 SB 型指令
uimm[19:0]	input	指令中第 [31:12] 位，用于 U 型指令
jimm[19:0]	input	指令中第 [31, 19:12, 20, 30:21] 位，用于 J 型指令
immctrl[4:0]	input	控制 EXT 模块运算类型
immout[31:0]	output	EXT 模块的输出结果

5.7 DM (数据存储器)

5.7.1 功能描述

根据输入的信号和指令，将相应内存地址的数据写入或取出。写入只在时钟上升沿发生，取出可以是任意时候的。同时也支持取出低 1 字节、低 2 字节、低 4 字节，Xilinx ISE 中预置的 IP core 可以支持四个字节的任意读写。数据在内存中用小端法存储，一个地址储存 4 个字节的数据，而汇编指令中一个地址储存 1 个字节的数据，因此需要在总线中进行转换。内存中预置了 1024 个字的空间，共 4096 字节。在具体结合 SoC 系统在 Xilinx ISE 中实现时，引用 Block Memory Generator。

5.7.2 模块接口

信号名	方向	描述
clka	input	时钟信号
WEA [3:0]	input	每个字节的写使能信号
addr[9:0]	input	需要访问的字地址
dina[31:0]	input	需要输入的数据
douta[31:0]	output	从访问地址读出的数据

5.8 IM (指令存储器)

5.8.1 功能描述

根据输入的信号和地址，将相应指令地址的数据取出，送往后续控制器等模块。数据在内存中用小端法存储，一个地址储存 4 个字节的数据，而汇编指令中一个地址储存 1 个字节的数据，因此需要在总线中进行转换。在具体结合 SoC 系统在 Xilinx ISE 中实现时，引用 Distributed Memory Generator。

5.8.2 模块接口

信号名	方向	描述
a[9:0]	input	输入的指令地址
spo[31:0]	output	从指令地址读出的数据

5.9 flop (流水线寄存器)

5.9.1 功能描述

临时存放每个流水线阶段需要传给下个阶段的数据，同时支持复位、写使能、清空，方便插入 `nop` 指令和执行跳转指令。

5.9.2 模块接口

信号名	方向	描述
<code>clk</code>	input	时钟信号
<code>reset</code>	input	复位信号
<code>en</code>	input	写入使能信号
<code>clear</code>	input	清空信号
<code>d [WIDTH-1:0]</code>	input	输入寄存器的数据
<code>q [WIDTH-1:0]</code>	output	寄存器的输出
<code>WIDTH</code>	parameter	寄存器位宽参数

5.10 hazard (冒险探测)

5.10.1 功能描述

用于对 load 型指令在 MEM 阶段才能获取到的数据，需要停顿（阻塞）一个周期才能继续执行。将可能涉及到的数据输入冒险探测模块，即可判断当前状态下各阶段是否需要阻塞。

根据《计算机组成与设计：硬件/软件接口 RISC-V（第五版）》[1]，如果存在下面这样的数据冒险，则需要停顿：

```
1 if (ID/EX.MemRead and  
2   ((ID/EX.RegisterRd = IF/ID.RegisterRs1) or  
3    (ID/EX.RegisterRd = IF/ID.RegisterRs2)))  
4   stall the pipeline
```

如果当前正在阻塞，则不进行冒险探测。

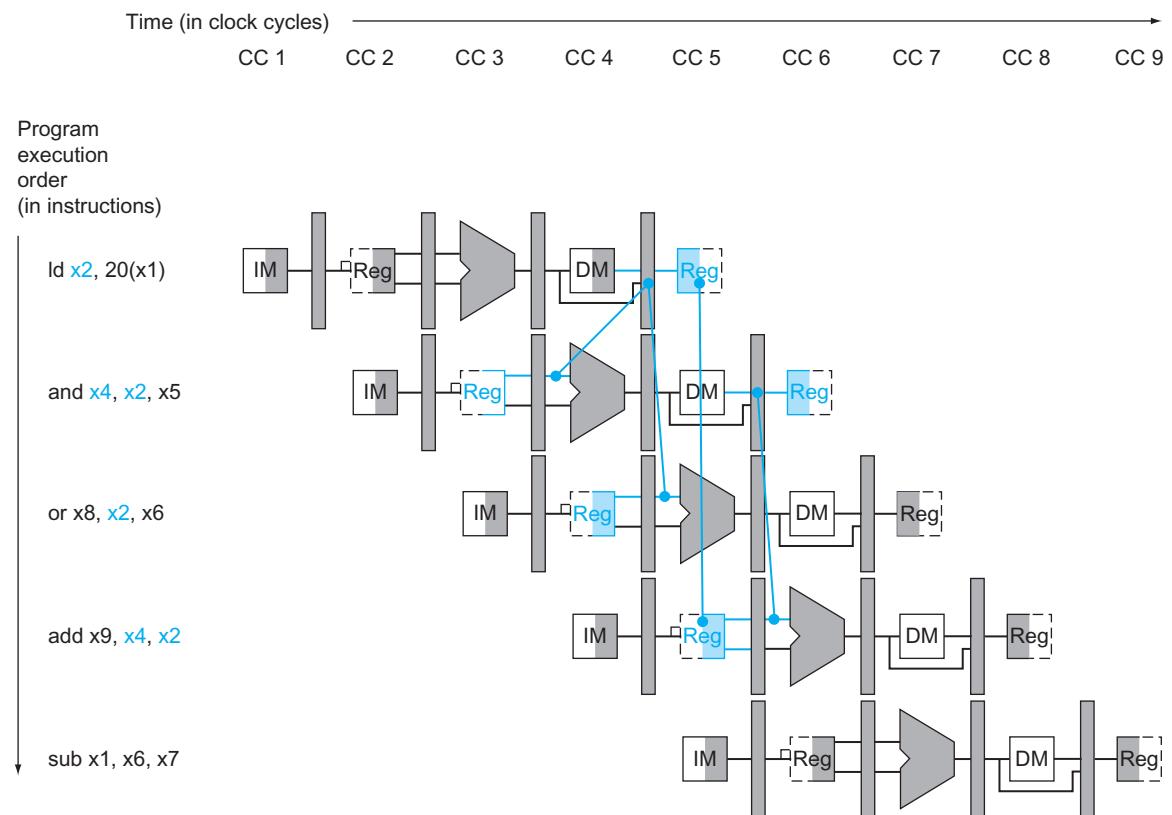


图 5: 流水线 CPU 中出现停顿的情况 [1]

5.10.2 模块接口

信号名	方向	描述
clk	input	时钟信号
memtoreg	input	判断此时是否读取存储器
rdE[4:0]	input	ID/EX 阶段的 rd 序号
rs1D[4:0]	input	IF/ID 阶段的 rs1 序号
rs2D[4:0]	input	IF/ID 阶段的 rs2 序号
writenM	input	上一阶段的阻塞信号
writen	output	该阶段阻塞信号的输出

5.11 forward (旁路前递)

5.11.1 功能描述

对于一些运算指令在流水线中来说，当前的运算的数据可能仍在流水线中进行运算，但多数情况下这样的数据其实已经有了运算结果，处于数据通路中，只是还未写回它应处于的位置。所以要添加旁路前递将这些数据及时取出，以保证流水线架构的正常运作。

根据《计算机组成与设计：硬件/软件接口 RISC-V（第五版）》[1]，如果存在下面这样的数据冒险，则需要进行旁路前递：

```
1 EX/MEM.RegisterRd = ID/EX.RegisterRs1
2 EX/MEM.RegisterRd = ID/EX.RegisterRs2
3 MEM/WB.RegisterRd = ID/EX.RegisterRs1
4 MEM/WB.RegisterRd = ID/EX.RegisterRs2
```

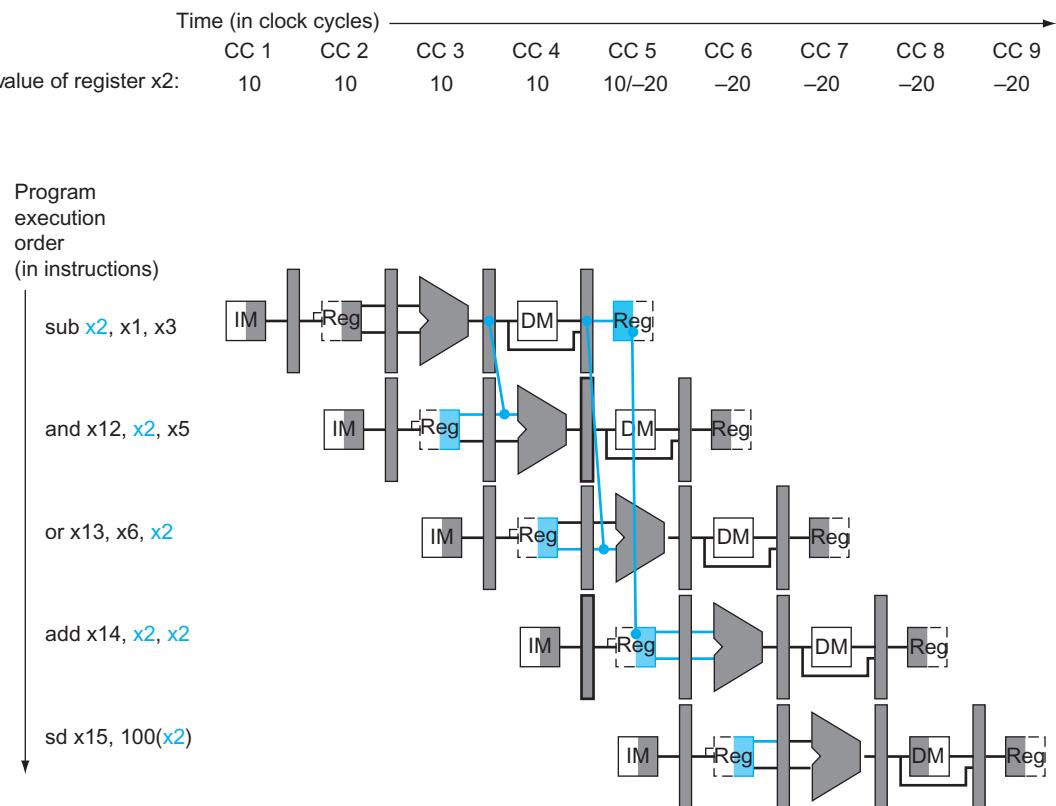


图 6: 流水线 CPU 中出现前递的情况 [1]

如果当前正在阻塞，则不进行冒险探测。

说明：当 `forward` 信号为不同值时，对应情况如下 [1]：

控制信号	数据来源	解释
00	ID/EX	ALU 操作数正常来自 ID/EX 阶段
10	EX/MEM	ALU 操作数从 EX/MEM 阶段前递过来
01	MEM/WB	ALU 操作数从 MEM/WB 阶段前递过来

5.11.2 模块接口

信号名	方向	描述
<code>regwriteM</code>	input	EX/MEM 阶段的寄存器写使能
<code>rdM[4:0]</code>	input	EX/MEM 阶段的 rd 序号
<code>rs1E[4:0]</code>	input	ID/EX 阶段的 rs1 序号
<code>rs2E[4:0]</code>	input	ID/EX 阶段的 rs2 序号
<code>regwriteW</code>	input	MEM/WB 阶段的寄存器写使能
<code>rdW[4:0]</code>	input	MEM/WB 阶段的 rd 序号
<code>forwardA[1:0]</code>	input	旁路 A 的选择信号
<code>forwardB[1:0]</code>	input	旁路 B 的选择信号

6 详细设计 - 流水线

6.1 CPU 总体结构

依据教材《计算机组成与设计：软/硬件接口 RISC-V（第五版）》[1] 中流水线架构 CPU 各模块的输入、输出端口，功能信号的规定以及 RISC-V 指令集中各个指令对应的数据周转流程与对应信号。模块中所有连线基于该架构进行实现。

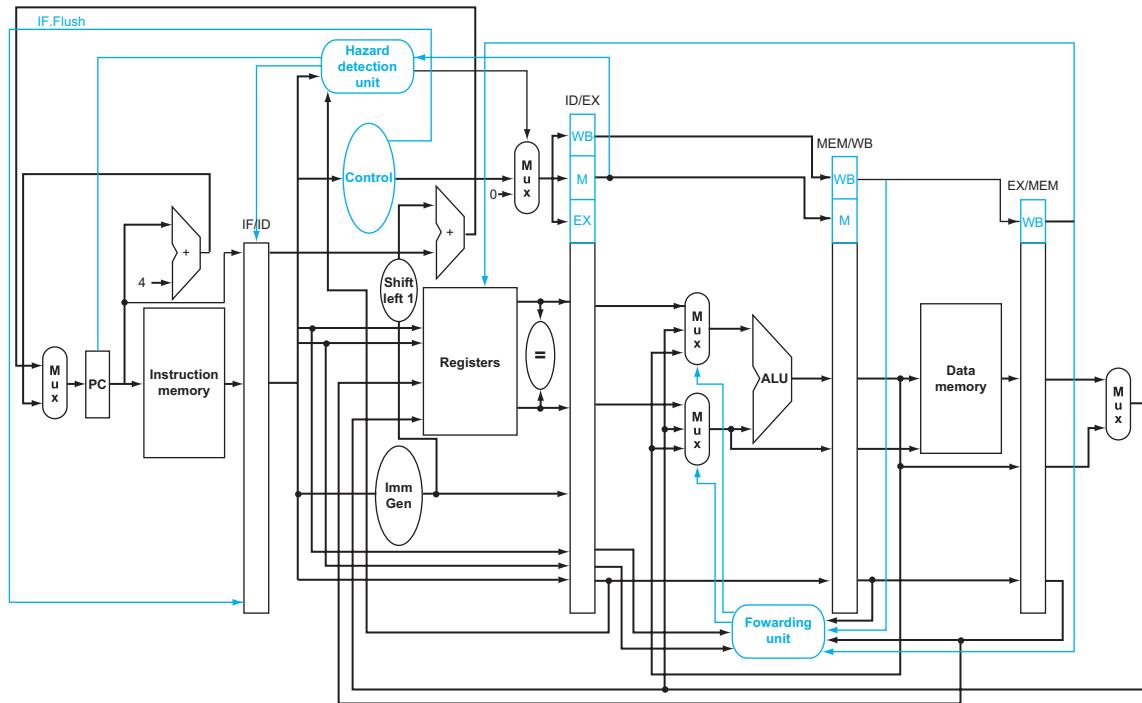


图 7: 流水线 CPU 原理图 [1]

用 datapath.v 文件将 PC、EXT、RF、ALU、流水线寄存器和冒险处理等整合到同一文件中；然后再用 pipelined.v 引入 Ctrl 模块，最终由 TOP 顶层文件进行组织之后，和封装在 IP core 的数据存储器（RAM）和指令存储器（ROM）进行交互。并将该文件综合编译为二进制文件比特流后导入 SWORD 开发板中进行仿真实现并验证。

pipelined.v 设计如下：（省略连线的声明，详见附件中.v 文件）

```
1 `include "xgriscv_defines.v"  
2
```

```

3 // 输入输出接口主要用于顶层文件综合
4 module xgriscv(input clk, reset,
5     input MIO_ready,
6     input [^INSTR_SIZE-1:0] inst_in,
7     input [^XLEN-1:0] Data_in,
8     output mem_w,
9     output [31:0] PC_out,
10    output [^ADDR_SIZE-1:0] Addr_out,
11    output [^XLEN-1:0] Data_out,
12    output CPU_MIO,
13    output [3:0] WEA,
14    input INT
15 );
16
17 controller c(clk, reset, opD, funct3D, funct7D, rdD, rs1D,
18    immD, zeroD, ltD,
19    immctrlD, itypeD, jalD, jalrD, bunsignedD, pcsrcD,
20    aluctrlD, aluctrl1D, alusrcad, alusrcbD, memwriteD,
21    lunsignedD, jD, bD, lwhbD, swhbD, memtoregD, regwriteD
22    );
23 // Ctrl 部分均为 IF/ID 阶段的信号，因此都标注为 D
24
25 // datapath 在接入 Ctrl 部分的信号后传入数据通路
26 datapath dp(clk, reset, inst_in, PC_out, Data_in, Addr_out,
27     Data_out, mem_w,
28     immctrlD, itypeD, jalD, jalrD, bunsignedD, pcsrcD,
29     aluctrlD, aluctrl1D, alusrcad, alusrcbD, memwriteD,
30     lunsignedD, jD, bD, lwhbD, swhbD, memtoregD,
31     regwriteD,
32     opD, funct3D, funct7D, rdD, rs1D, immD, zeroD, ltD,
33     memwriteD, WEA);
34
35 endmodule

```

6.2 datapath (数据通路)

数据通路中省略无初始化的连线声明，具体见 `datapath.v` 文件。

```
1 `include "xgriscv_defines.v"
2
3 module datapath(
4     input                  clk, reset,
5
6     input [`INSTR_SIZE-1:0] instrF,      // from IM
7     output[`ADDR_SIZE-1:0] pcF,        // to IM
8
9     input [`XLEN-1:0]       readdataM,   // from DM
10    output[`XLEN-1:0]        aluoutM,    // to DM - 地址
11    output[`XLEN-1:0]        writedataM, // to DM - 数据
12    output                  memwriteM,   // to DM - 写使能
13
14    // 控制信号如下
15    input [4:0]             immctrlD,
16    input                  itype, jald, jalrD, bunsignedD, pcsrcD,
17    input [3:0]             aluctrlD,
18    input [2:0]             aluctrl1D,
19    input [1:0]             alusrcA,
20    input                  alusrcB,
21    input                  memwriteD, lunsignedD, jD, bD,
22    input [1:0]             lwhbD, swhbD,
23    input                  memtoregD, regwriteD,
24
25    // 输出到 Ctrl 解析的指令
26    output [6:0]            opD,
27    output [2:0]            funct3D,
28    output [6:0]            funct7D,
29    output [4:0]            rdD, rs1D,
30    output [11:0]           immD,
31    output                  zeroD, ltD,
32    input                   data_ram_wed,
```

```

34     output [3:0]      WEAM );
35     wire flushM = 0;
36     mux2 #(`ADDR_SIZE)  pcsrcmux(pcplus4F, pcbranchD, pcsrc,
37           nextpcF);
38     //PC 寄存器
39     pcenr   pcreg(clk, reset, writenE, nextpcF, pcF);
40     addr_adder pcadder1(pcF, `ADDR_SIZE'b100, pcplus4F);
41
42     /////////////////////////////////
43     // IF/ID 流水线寄存器
44     wire flushD = pcsrc;
45
46     // 分别存储指令、 PC 、 PC+4 的流水线寄存器
47     fopenrc #(`INSTR_SIZE)    pr1D(clk, reset, writenE, flushD,
48           instrF, instrD);      // instruction
49     fopenrc #(`ADDR_SIZE)    pr2D(clk, reset, writenE, flushD,
50           pcF, pcD);          // pc
51     fopenrc #(`ADDR_SIZE)    pr3D(clk, reset, writenE, flushD,
52           pcplus4F, pcplus4D); // pc+4
53
54     // Decode stage logic
55     assign opD      = instrD[6:0];
56     assign rdD      = instrD[11:7];
57     assign funct3D  = instrD[14:12];
58     assign rs1D     = instrD[19:15];
59     assign rs2D     = itype?5'b00000:instrD[24:20];
60     // 此处如果使用 itype , 则该阶段可能不为 0 (如 srai )
61     assign funct7D  = instrD[31:25];
62     assign immD     = instrD[31:20];
63
64     // 立即数处理部分
65     wire [11:0] iimmD = instrD[31:20];
66     wire [11:0] simmD = {instrD[31:25], instrD[11:7]};
67     wire [11:0] bimmD = {instrD[31], instrD[7], instrD[30:25], instrD
68           [11:8]};
69     wire [19:0] uimmD = instrD[31:12];

```

```

65     wire [19:0] jimmD = {instrD[31], instrD[19:12], instrD[20],
66                           instrD[30:21]};
67
68     imm im(iimmD, simmD, bimmD, uimmD, jimmD, immctrlD, immoutD);
69     regfile rf(clk, rs1D, rs2D, rdata1D, rdata2D, regwriteW,
70                 waddrW, wdataW, pcW);
71
72     /////////////////////////////////
73     // ID/EX 流水线寄存器
74
75     // 控制信号寄存器
76     assign flushE = pcsrc | ~writenE;
77     flopenrc #(21) regE(clk, reset, writenE, flushE, {regwriteD,
78               memwriteD, memtoregD, lwhbD, swhbD, lunsignedD, alusrcAD,
79               alusrcbD, aluctrlD, aluctrl1D, jD, bD, data_ram_weD}, {
80               regwriteE, memwriteE, memtoregE, lwhbE, swhbE, luE,
81               alusrcAE, alusrcbE, aluctrlE, aluctrl1E, jE, bE,
82               data_ram_weE});
83
84     // 数据寄存器
85     flopenrc #(`XLEN)    pr1E(clk, reset, writenE, flushE, rdata1D,
86                               srca1E);           // data from rs1
87     flopenrc #(`XLEN)    pr2E(clk, reset, writenE, flushE, rdata2D,
88                               srcb1E);           // data from rs2
89     flopenrc #(`XLEN)    pr3E(clk, reset, writenE, flushE, immoutD,
90                               immoutE);          // imm output
91     flopenrc #(`RFIDX_WIDTH) pr4E(clk, reset, writenE, flushE,
92                                   rs1D, rs1E);        // rs1
93     flopenrc #(`RFIDX_WIDTH) pr5E(clk, reset, writenE, flushE,
94                                   rs2D, rs2E);        // rs2
95     flopenrc #(`RFIDX_WIDTH) pr6E(clk, reset, writenE, flushE,
96                                   rdD, rdE);         // rd
97     flopenrc #(`ADDR_SIZE)  pr8E(clk, reset, writenE, flushE, pcD,
98                                   pcE);             // pc
99     flopenrc #(`ADDR_SIZE)  pr9E(clk, reset, writenE, flushE,
100                            pcplus4D, pcplus4E); // pc+4

```

```
86
87     wire [1:0]    forwardA, forwardB;
88
89     wire[`XLEN-1:0] srca, srcb;
90
91     mux3 #(`XLEN)   fA(srca1E, wdataW, aluoutM, forwardA, srca); //
92     mux3 #(`XLEN)   fB(srcb1E, wdataW, aluoutM, forwardB, srcb); //
93
94
95 // ALU 运算数据来源选择
96
97     mux3 #(`XLEN)  srcamux(srca, 0, pcE, alusrcaE, srcaE);
98     mux2 #(`XLEN)  srcbmux(srcb, immoutE, alusrcbE, srcbE);
99
100
101 // 分别用于计算和跳转
102
103     alu alu(srcaE, srcbE, 5'b0, aluctrlE, aluctrl1E, aluoutE,
104         overflowE, zeroE, ltE, geE);
105
106     alu alu1(pcE, immoutE, 5'b0, `ALU_CTRL_ADD, 3'b000, PCoutE,
107         overflowE, zeroE, ltE, geE);
108
109
110 // branch 后的 PC 选择器
111
112     mux2 #(`XLEN) brmux(aluoutE, PCoutE, B, pcbranchD);
113
114
115 // EX/MEM 流水线寄存器
116
117
118 flopvc #(1) wrenM(clk, reset, flushM, writenE, writenM);
119 flopvc #(`XLEN+11) regM(clk, reset, flushM, {srcb1E,
120                     regwriteE, memwriteE, memtoregE, lwhbE, luE, swhbE, jE, bE
121                     , data_ram_weE}, {srcb1M, regwriteM, memwriteM, memtoregM,
122                     lwhbM, luM, swhbM, jM, bM, data_ram_weM});
123
124 flopvc #(`ADDR_SIZE) regpcM(clk, reset, flushM, PCoutE,
125             PCoutM);
```

```

116    // for data
117    floprc #(`XLEN) pr1M(clk, reset, flushM, aluoutE, aluoutM);
118    floprc #(`XLEN) pr5M(clk, reset, flushM, srcb, writedataM);
119    floprc #(`RFIDX_WIDTH) pr2M(clk, reset, flushM, rdE, rdM);
120    floprc #(`ADDR_SIZE) pr3M(clk, reset, flushM, pcE, pcM);
121    floprc #(`ADDR_SIZE) pr4M(clk, reset, flushM, pcplus4E,
122        pcplus4M);
123
124    reg [31:0] intmp;
125    always @(*) begin
126        case(lwhbM) // 处理读取数据
127            2'b11: intmp <= readdataM;
128            2'b10: intmp <= luM?{16'b0, readdataM[15:0]}:{16{
129                readdataM[15]}, readdataM[15:0]};
130            2'b01: intmp <= luM?{24'b0, readdataM[7:0]}:{24{readdataM
131                [7]}}, readdataM[7:0];
132        endcase
133    end
134
135    assign dmoutM = intmp;
136
137    reg [3:0] WEAtmp;
138    always @(*) begin
139        if (memwriteM) begin
140            case(swhbM) // 处理写入数据
141                2'b11: WEAtmp={data_ram_weM, data_ram_weM, data_ram_weM,
142                    data_ram_weM};
143                2'b10: WEAtmp={1'b0, 1'b0, data_ram_weM, data_ram_weM};
144                2'b01: WEAtmp={1'b0, 1'b0, 1'b0, data_ram_weM};
145                default: WEAtmp=4'b0000;
146            endcase
147        end
148        else      WEAtmp=4'b0000;
149    end
150
151    assign WEAM = WEAtmp;

```

```

148 ///////////////////////////////////////////////////////////////////
149 // MEM/WB 流水线寄存器
150 wire flushW = 0;
151 flopvc #(`XLEN+4) regW(clk, reset, flushW, {dmoutM, regwriteM,
152 memtoregM, jM, bM}, {dmoutW, regwriteW, memtoregW, jW, bW
153 });
154 flopvc #(`ADDR_SIZE) regpcW(clk, reset, flushW, PCoutM, PCoutW
155 );
156 forward fw(regwriteM, rdM, rs1E, rs2E, regwriteW, rdW,
157 forwardA, forwardB);
158 flopvc #(`XLEN) pr1W(clk, reset, flushW, aluoutM, aluoutW);
159 flopvc #(`RFIDX_WIDTH) pr2W(clk, reset, flushW, rdM, rdW);
160 flopvc #(`ADDR_SIZE) pr3W(clk, reset, flushW, pcM, pcW);
161 flopvc #(`ADDR_SIZE) pr4W(clk, reset, flushW, pcplus4M,
162 pcplus4W);
163
164 // write-back stage logic
165 //j or B & meet the case
166 //j -> aluoutW
167 //B -> immoutW
168 mux3 #(`XLEN) wdatamux(aluoutW, pcplus4W, dmoutW, {memtoregW,
169 jW}, wdataW);
170 assign waddrW = rdW;
171
172 endmodule

```

6.3 PC (程序计数器)

```

1 module pcenr (
2     input                  clk, reset,
3     input                  en,
4     input      [`XLEN-1:0] d,
5     output reg   [`XLEN-1:0] q);
6

```

```

7      always @(posedge clk, posedge reset)
8          if (reset)
9              q <= `ADDR_SIZE'h00000000 ;
10         else if (en)
11             q <= d;
12     endmodule

```

6.4 RF（寄存器文件）

```

1 `include "xgriscv_defines.v"
2
3 module regfile(
4     input                  clk,
5     input [`RFIDX_WIDTH-1:0] ra1, ra2,
6     output [`XLEN-1:0]      rd1, rd2,
7
8     input                  we3,
9     input [`RFIDX_WIDTH-1:0] wa3,
10    input [`XLEN-1:0]      wd3,
11
12    input [`ADDR_SIZE-1:0] pc );
13
14    reg [`XLEN-1:0] rf[`RFREG_NUM-1:0];
15
16    // 两个读取数据端口，一个写入端口
17    // x0 的值固定为 0
18
19    always @ (negedge clk)
20        if (we3 && wa3!=0) begin
21            rf[wa3] <= wd3;
22        end
23
24        assign rd1 = (ra1 != 0) ? rf[ra1] : 0;
25        assign rd2 = (ra2 != 0) ? rf[ra2] : 0;
26    endmodule

```

6.5 Ctrl（控制模块）

Ctrl 根据预置的控制信号，如前文表3所示，依据输入的指令进行相应类型判断和信号输出。

附件中 `xgriscv_defines.v` 也依照下表进行定义。

```
1 `include "xgriscv_defines.v"
2
3 module controller(
4     input          clk, reset,
5     input [6:0]      opcode,
6     input [2:0]      funct3,
7     input [6:0]      funct7,
8     input [`RFIDX_WIDTH-1:0] rd, rs1,
9     input [11:0]     imm,
10    input           zero, lt,
11
12   output [4:0]      immctrl,
13   output      itype, jal, jalr, bunsigned, pcsrc,
14   output reg [3:0]  aluctrl,
15   output reg [2:0]  aluctrl1,
16   output [1:0]      alusrca,
17   output           alusrcb,
18   output      memwrite, lunsigned, j, btype,
19   output [1:0]      lwhb, swhb,
20   output      memtoreg, regwrite
21 );
22
23 // 省略根据指令判断类型部分代码，具体依照表 3 进行判断
24
25 wire rv32_rs1_x0 = (rs1 == 5'b00000);
26 wire rv32_rd_x0 = (rd == 5'b00000);
27 wire rv32_nop = rv32_addi & rv32_rs1_x0 & rv32_rd_x0 & (imm ==
28     12'b0); //addi x0, x0, 0 is nop
29 assign itype = rv32_load | rv32_addr | rv32_jalr;
```

```

30
31     wire stype = rv32_store;
32     wire utype = rv32_lui | rv32_auipc;
33     wire jtype = rv32_jal;
34     reg jtmp;
35
36     assign immctrl = {itype, stype, btype, utype, jtype};
37
38     assign jal = rv32_jal;
39     assign jalr = rv32_jalr;
40     assign bunsigned = rv32_bltu | rv32_bgeu;
41     assign pcsrc = 0;
42     assign alusrca=rv32_lui?2'b01:(rv32_jal||rv32_auipc?2'b10:2'b00);
43
44     assign alusrcb=rv32_lui||rv32_auipc||itype||rv32_load||
45         rv32_store||rv32_jalr||rv32_jal;
46     assign memwrite = rv32_store;
47     //w:11, h:10, b:01
48     assign swhb = {rv32_sw|rv32_sh, rv32_sw|rv32_sb};
49     assign lwhb = {rv32_lw|rv32_lh|rv32_lhu, rv32_lw|rv32_lb|
50         rv32_lbu};
51     assign lunsigned = rv32_lbu | rv32_lhu;
52
53     assign memtoreg = rv32_load;
54     assign regwrite = rv32_lui | rv32_auipc | rv32_addi |
55         rv32_addr | itype | rv32_jalr | rv32_jal;
56
57     always @(*) begin
58         case(opcode)
59             `OP_LUI:    begin aluctrl1 <= `ALU_EMP;
60                         aluctrl <= `ALU_CTRL_LUI; end
61             `OP_AUIPC: begin aluctrl1 <= `ALU_EMP;
62                         aluctrl <= `ALU_CTRL_AUIPC; end
63             `OP_LOAD:   begin aluctrl1 <= `ALU_EMP;
64                         aluctrl <= `ALU_CTRL_ADD; end

```

```

62     `OP_STORE: begin aluctrl1 <= `ALU_EMP;
63             aluctrl <= `ALU_CTRL_ADD; end
64 // 省略其余 30 种 ALU 控制信号的 case 判断
65 default: begin aluctrl <= `ALU_CTRL_ZERO;
66             aluctrl1 <= 3'b000; end
67         endcase
68
69     case(rv32_jal | rv32_jalr)
70         1'b1: jtmp <= 1'b1;
71         default: jtmp<=1'b0; // 避免 X 信号的出现
72     endcase end
73     assign btype = aluctrl1[2:0]?1:0;
74     assign j = jtmp;
75 endmodule

```

6.6 ALU (运算单元)

```

1 `include "xgriscv_defines.v"
2 module alu(
3     input signed      [`XLEN-1:0] a, b,
4     input      [4:0]      shamt,
5     input      [3:0]      aluctrl,
6     input      [2:0]      aluctrl1,
7
8     output reg   [`XLEN-1:0]  aluout,
9     output          overflow,
10    output          zero,
11    output          lt,
12    output          ge );
13
14    wire op_unsigned = ~aluctrl[3]&~aluctrl[2]&aluctrl[1]&~aluctrl
15        [0]      //ADDU 4'b0010
16    | aluctrl[3]&~aluctrl[2]&aluctrl[1]&~aluctrl[0] //SUBU 4'b1010
17    | aluctrl[3]&aluctrl[2]&~aluctrl[1]&~aluctrl[0] //SLTU 4'b1100
18    | aluctrl1[2]&~aluctrl1[1]&aluctrl1[0] //BLTU
19    | aluctrl1[2]&aluctrl1[1]&~aluctrl1[0]; //BGEU

```

```

19
20     wire sub = aluctrl[3]&~aluctrl[2]&~aluctrl[1]&aluctrl[0]
21     |aluctrl[3]&~aluctrl[2]&aluctrl[1]&~aluctrl[0]
22     |aluctrl[3]&~aluctrl[2]&aluctrl[1]&aluctrl[0]
23     |aluctrl[3]&aluctrl[2]&~aluctrl[1]&~aluctrl[0]
24     |aluctrl1[2]|aluctrl1[1]|aluctrl1[0];
25     //slt or b
26
27     assign b2 = sub ? ~b:b;
28     assign sum = (op_unsigned & ({1'b0, a} + {1'b0, b2} + sub))
29     |(~op_unsigned & ({a[`XLEN-1], a} + {b2[`XLEN-1], b2} + sub));
30     // aluctrl[3]=0 if add, or 1 if sub, don't care if other
31     assign sll = a<<b;
32     assign XOR = a^b;
33     assign OR = a|b;
34     assign AND = a&b;
35     assign srl = a>>b;
36     assign sra = a>>>b[9:0];
37     integer signed i;
38
39     always@(*)
40         case(aluctrl1[2:0])
41             `ALU_BEQ:    aluout <= sum[`XLEN-1:0] !=0?0:1; //ZERO
42             `ALU_BNE:    aluout <= sum[`XLEN-1:0] !=0?1:0; //ZERO
43             `ALU_BLT:    begin //slt
44                 if(a[`XLEN-1] !=b[`XLEN-1])
45                     aluout <= a[`XLEN-1];
46                 else
47                     aluout <= sum[`XLEN-1];
48             end
49             `ALU_BGE:    begin
50                 if(a[`XLEN-1] !=b[`XLEN-1])
51                     aluout <= ~a[`XLEN-1];
52                 else
53                     aluout <= ~sum[`XLEN-1];
54             end

```

```

55     `ALU_BLTU:   aluout <= a[`XLEN-1:0]<b[`XLEN-1:0];
56     `ALU_BGEU:   aluout <= a[`XLEN-1:0]>=b[`XLEN-1:0];
57     default: // 若 aluctrl 为 0 则进行 aluctrl
58         case(aluctrl[3:0])
59             `ALU_CTRL_MOVEA:      aluout <= a;
60             `ALU_CTRL_ADD:       aluout <= sum[`XLEN-1:0];
61             // 省略其他运算符的输出过程，计算结果已在上面 assign 部分
62             // 给出
63             default:           aluout <= `XLEN'b0;
64         endcase
65     endcase
66
66     assign overflow = sum[`XLEN-1] ^ sum[`XLEN];
67     assign zero = (aluout == `XLEN'b0);
68     assign lt = aluout[`XLEN-1];
69     assign ge = ~aluout[`XLEN-1];
70 endmodule

```

6.7 EXT (符号扩展)

```

1 module imm (
2     input [11:0]          iimm, //instr[31:20]
3     input [11:0]          simm, //instr[31:25, 11:7]
4     input [11:0]          bimm, //instrD[31, 7, 30:25, 11:8]
5     input [19:0]          uimm,
6     input [19:0]          jimm,
7     input [4:0]           immctrl,
8
9     output reg [`XLEN-1:0] immout );
10    always @(*)
11        case (immctrl)
12            `IMM_CTRL_ITYPE: immout <= {{{`XLEN-12}{iimm[11]}}, iimm
13                [11:0]};
14            `IMM_CTRL_UTYPE: immout <= {uimm[19:0], 12'b0};
15            `IMM_CTRL_STYPE: immout <= {{{32-12}{simm[11]}}, simm
16                [11:0]};

```

```

15      // 忽略最低位 (乘 2 )
16      `IMM_CTRL_BTYP: immout <= {{{32-13}{bimm[11]}}}, bimm
17          [11:0], 1'b0};
18      `IMM_CTRL_JTYP: immout <= {{{32-21}{jimm[19]}}}, jimm
19          [19:0], 1'b0};
20      default:           immout <= `XLEN'b0;
21      endcase
22 endmodule

```

6.8 flop (流水线寄存器)

```

1 module fopenrc #(parameter WIDTH = 8)
2     (input                  clk, reset,
3      input                  en, clear,
4      input [WIDTH-1:0] d,
5      output reg [WIDTH-1:0] q);
6
7     always @ (posedge clk, posedge reset)
8         if (reset) q <= 0;
9         else if (clear) q <= 0;
10        else if (en)    q <= d;
11 endmodule

```

6.9 hazard (冒险探测)

```

1 module hazard (input clk, input memtoreg,
2     input [`RFIDX_WIDTH-1:0] rdE, input [`RFIDX_WIDTH-1:0] rs1D,
3     input [`RFIDX_WIDTH-1:0] rs2D, input writenM, output reg
4     writen);
5     always @ *
6         if(memtoreg && (rdE == rs1D || rdE == rs2D) && writenM)
7             writen <= 1'b0;
8         else
9             writen <= 1'b1;
10    endmodule

```

6.10 forward (旁路前递)

```
1 module forward (input regwriteM, input[`RFIDX_WIDTH-1:0] rdM,
2   input[`RFIDX_WIDTH-1:0] rs1E, input[`RFIDX_WIDTH-1:0] rs2E,
3   input regwriteW, input[`RFIDX_WIDTH-1:0] rdW, output reg[1:0]
4   forwardA, output reg[1:0] forwardB);
5   always@*
6     begin
7       forwardA=2'b00;
8       forwardB=2'b00;
9       if(regwriteM&&rdM!=0) begin
10         if(rdM == rs1E) forwardA=2'b10;
11         if(rdM == rs2E) forwardB=2'b10;
12         end
13       if(regwriteW&&rdW!=0) begin
14         if(!(regwriteM&&(rdM!=0)&&rdM==rs1E)
15           && rdW == rs1E) forwardA=2'b01;
16         if(!(regwriteM&&(rdM!=0)&&rdM==rs2E)
17           && rdW == rs2E) forwardB=2'b01;
18         end
19       end
20   endmodule
```

7 测试及结果分析

7.1 仿真代码及分析

```
1 addi x5, x0, 1000          22 srai x14, x10, 2
2 addi x4, x0, 10            23 sb x11 16(x5)
3 lui x6, 0xffff            24 and x8, x7, x6
4 sw x6, 4(x5)              25 auipc x12, 4
5 lh x7, 4(x5)              26 lbu x13, 16(x5)
6 sub x8, x7, x6            27 bgeu x8, x7, end
7 ori x9, x0, 111           28
8 andi x9, x9, 0             29 beq x0, x0, end
9 skip1:                      30
10 xori x9, x9, 1            31 mut: #x28*x29
11 bne x0, x9, skip1         32 beq x29, x0, end4
12 or x9, x4, x0              33 addi x30, x0, 0
13 srl x10, x5, 10           34 addi x31, x0, 0
14 skip2:                      35 loop:
15 add x10, x9, x9           36 add x31, x31, x28
16 bge x9, x10, skip2         37 addi x30, x30, 1
17 sw x9, 8(x5)              38 blt x30, x29, loop
18 lw x28, 8(x5)              39 end4:
19 addi x29, x4, 0             40 jalr x0, x1, 0
20 jal x1, mut                41
21 sll x11, x10, x9           42 end:
```

7.2 仿真测试结果

7.2.1 数据冒险

执行到代码第 6 行时，出现了数据冒险，第 6 行代码执行到 EX 阶段（第 8 周期），需要 x7 的数值，但第 5 行代码才执行到 MEM 阶段，还未读出 x7 的值，此时需要进行一次阻塞，如图8。

下一周期时，第 6 行代码仍在 EX 阶段，第 5 行代码执行 WB 阶段，此时

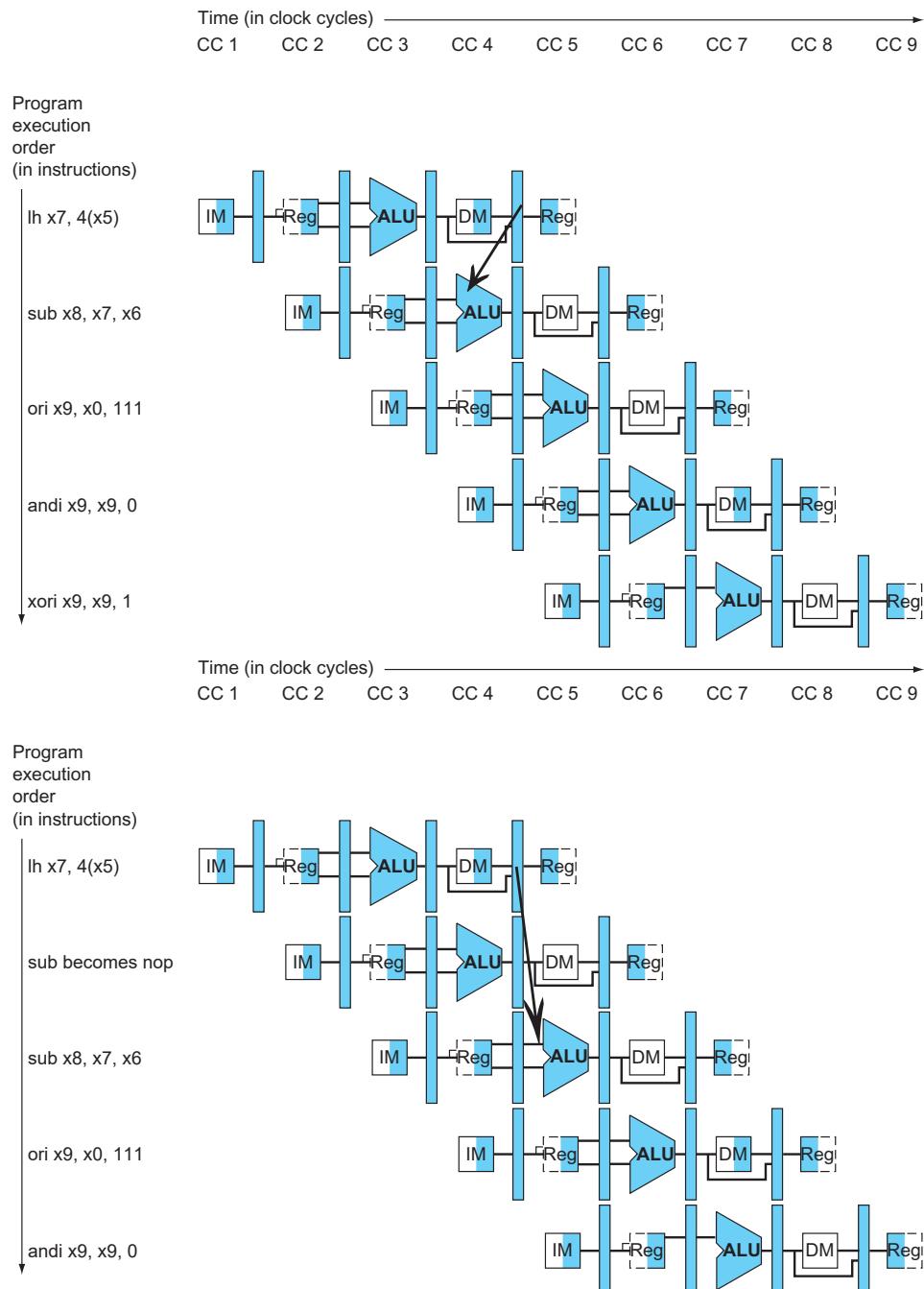


图 8: 出现数据冒险和阻塞后的情况

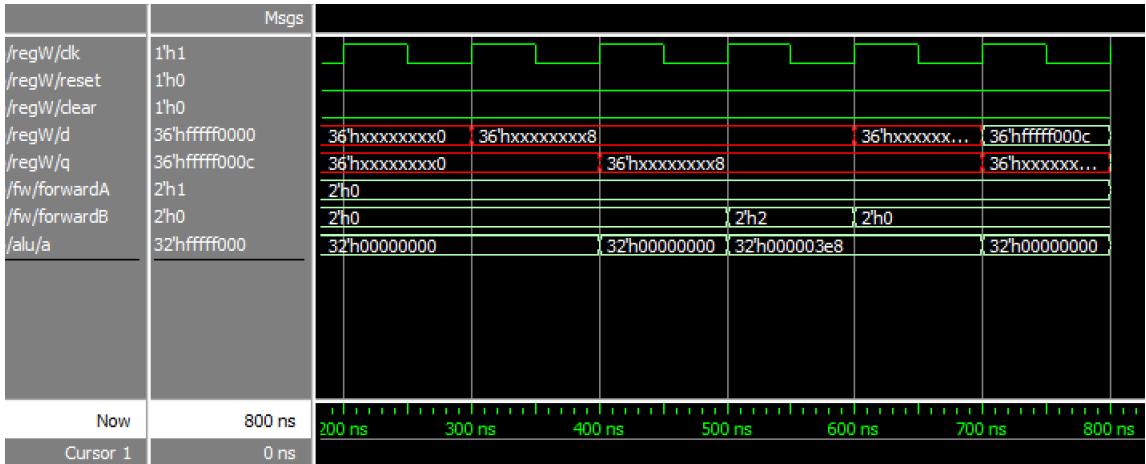


图 9: 出现数据冒险时相关线路的仿真 (第 8 周期)

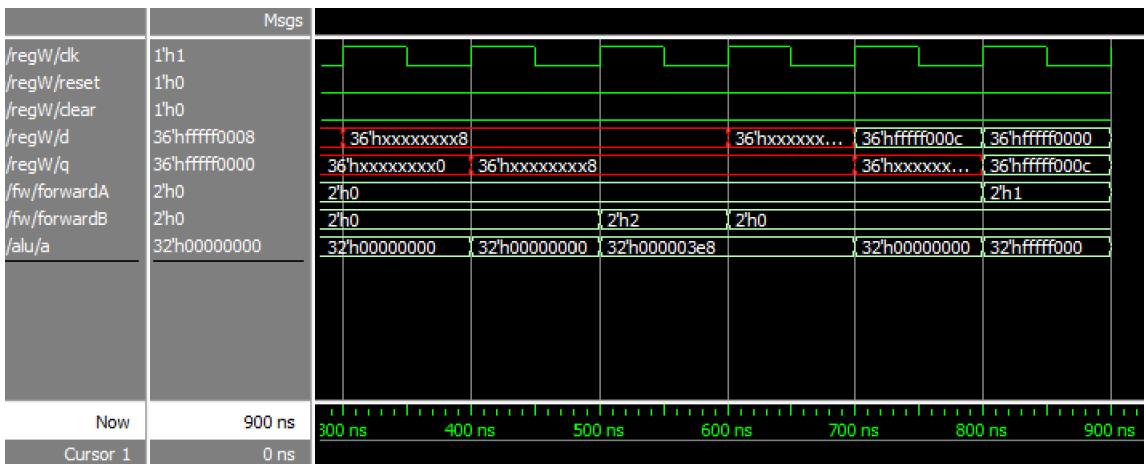


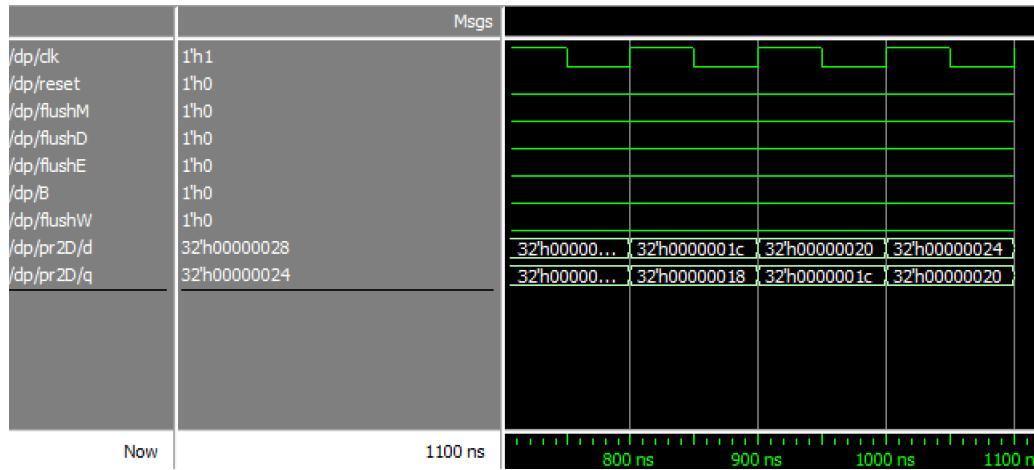
图 10: 出现数据冒险后阻塞的仿真 (第 9 周期)

MEM/WB 里已有 x7 的值，通过旁路将其传入 EX/MEM 阶段即可。

如图9、10分别为第 8 周期、第 9 周期的仿真图像。

此时还没有 x7 的数据 (0xfffff0000) 出现在 MEM/WB 寄存器的输入，此时 forwardA=01，即可将其前递给 ALU 的操作数 a 处。

x7 的数据 (0xfffff0000) 出现在 MEM/WB 寄存器的输入，此时 forwardA=01，即可将其前递给 ALU 的操作数 a 处。



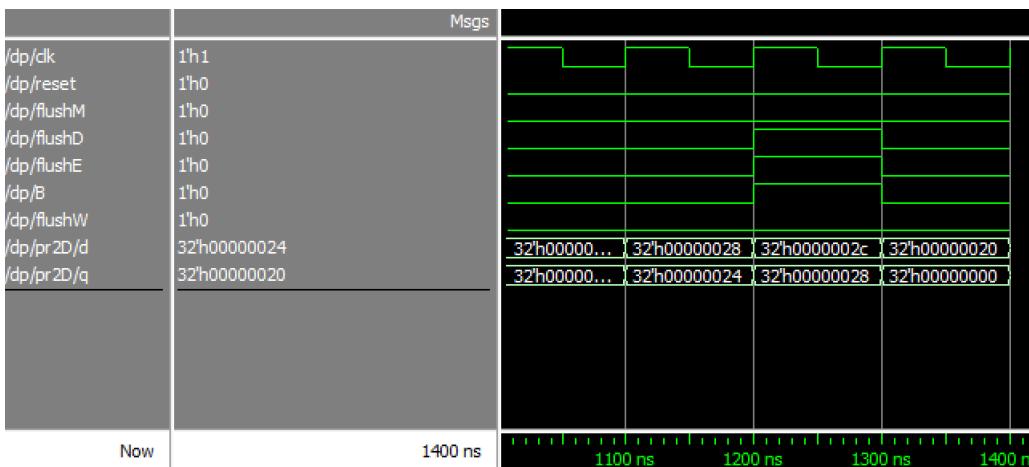


图 13: 第 14 周期, 跳转后 PC=0x20

7.2.2 控制冒险

执行到代码第 11 行时, 出现了控制冒险。此时流水线中已经有 12 行及以后的代码, 由于此时需要跳转, 所以在流水线中 EX 阶段以前的部分需要被清除掉, 重新读取新的 PC 地址的指令。

第 12 周期, 指令 `bne x0, x9, skip1` 仍在 ID 阶段, 所有 flush 信号和 B 信号 (探测到跳转发生) 均为 0。

第 13 周期, 指令 `bne x0, x9, skip1` 在 EX 阶段执行结束后, B 信号跳变为 1, 同时 flushD、flushE 也跳变为 1, 此时这两层 IF/ID, ID/EX 寄存器即将被清空。清空后, 流水线从 PC=0x20 的阶段继续执行。

7.3 下载测试代码及分析

测试汇编代码在 `diy.asm` 文件中。文件实现了 6×8 的矩阵加载和游戏数据处理功能。

在下载测试时, 基于 `VGAIO.v` 等文件, 修改了 TOP 文件和总线文件使其支持 VGA 图像的扫描。为了方便起见, 没有额外设置显存, 而是在 TOP 文件里定义了 48 个寄存器, 作为显示内容。

根据上述技术基础, 在本实验中设计了“方块消消乐”游戏。

本实验结合拨码开关、按键输入、VGA 输出、数码管输出四个交互式模块, 设计了这款游戏。

游戏简要规则为：将高亮光标移动至需要消灭的方块上，如果当前方块四周（四连通区域）有同样颜色的方块，按下消灭键即可将其全部消除。连通的数量越多，本次得分越多，得分在数码管上以 16 进制形式进行显示。

当方块被消灭后，其上方的方块会由于重力掉落，因此游戏的玩法较为丰富，最终目的为达到尽可能高的分数。

7.4 下载测试结果

如图所示，修改 `santi_jitter.v` 文件可以获取按键信号，从而操纵光标上下左右移动，并执行消灭操作。改变输入到七段数码管中的数据，可以实时显示当前得分。

经测试，游戏主体部分结果正确，数码管可以正常显示，可以正常接收按键区信号，认为实验成功。

汇编代码中有较多数据依赖和控制跳转，正常执行后说明流水线架构设计无误，可以正确处理数据冒险等可能存在的问题。

8 实验总结

8.1 实验总结

本实验是基于 RISC-V 架构设计的单周期、流水线 CPU，同时根据 SWORD 开发板特性设计了额外的交互式内容。实验具有较大的挑战性，一方面使用了 Verilog 这一不同于其他高级语言的设计方式，另一方面在复杂的数据周转和信号变化中，调试是一件非常困难的事情。

实现一个 CPU 并不难，但要将单周期 CPU 改进为流水线并支持数据冒险，就是非常考验能力的任务。数据并不只是单向流动，上升沿和下降沿、阻塞和非阻塞的判断都会对数字电路的实现造成非常大的影响，因此能够完整做出本实验，也是对能力的认可和自我的挑战。

在交互式内容中，VGA、按钮等相对陌生的环境往往需要数十小时甚至几天的时间来摸索和实验，在这期间和老师同学的互相交流和上网查阅资料的能力也显得尤为重要，但最主要的是动手实践能力。

不过，在按钮实现时，也遇到了较多软硬件结合（Verilog 和汇编结合）才能解决的情况。游戏的最终版本仍存在按钮响应的问题，在今后的设计实验中仍需

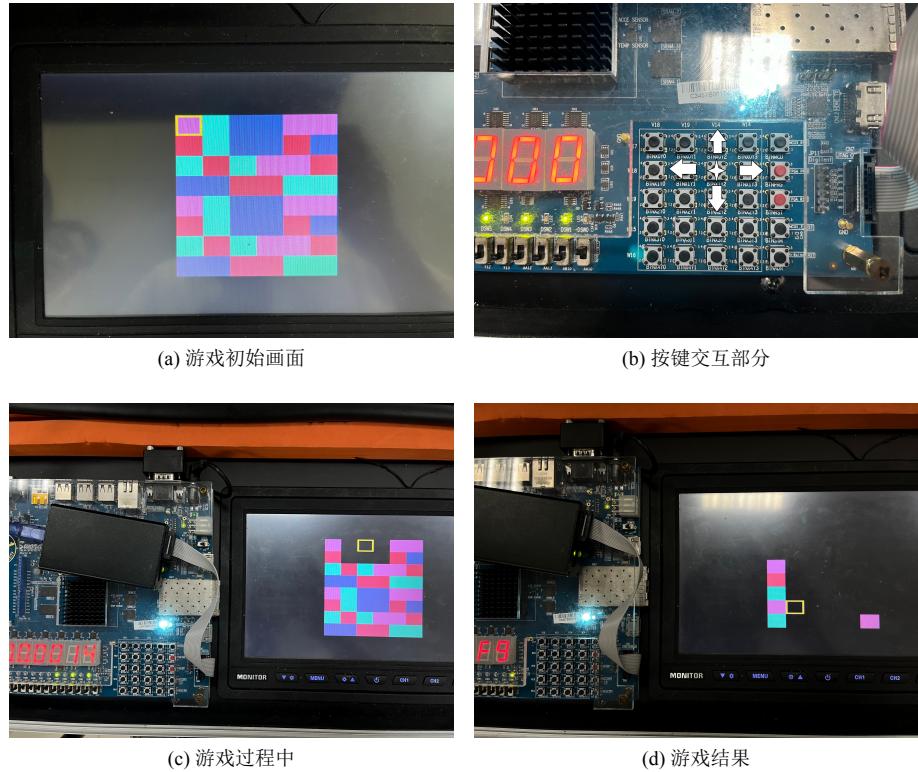


图 14. 基于本实验设计的“方块消消乐”游戏

加强设计的健壮性。

最终，VGA 显示正确，代码正常执行，寄存器和内存内容和 Venus 平台一致，说明实验成功。

8.2 取得的收获

在这段实验的过程中，我提升了自己的学习能力、动手实践能力、解决创新型问题的能力；也将汇编语言程序设计和 Verilog 编程的能力进行了巩固。

在进行流水线 CPU 课间实验时，我受到提交方式的启发，经常将自己的代码上传 github 仓库进行备份，一方面防止丢失，另一方面方便多设备同步，最重要的一点是方便回退版本。在设计实验过程中不仅仅培养的是专业能力，也有应用扩展和解决问题的能力。

在完成流水线 CPU 的过程中，我遇到了时序判断和逻辑信号的多重困难，尤其是莫名其妙出现的 X 信号，我对此深感困惑。后来我发现 ModelSim 软件可以单步调试，并实时观察每个信号的数据情况，在这之后我解决问题的速度有了显著的提升。而对于 X 信号产生的原因，主要是输入信号存在未定义的可能性。

此外，在实验过程中遇到的其他问题，也引发了我的思考，如果出现了难以解决的问题，就刨根问底找到问题的根源，将提供的原始元件的原理研究清楚，最终解决问题或更换元件。

本实验培养了我的探究性精神，也让我在完成这项“大工程”后有着自豪的成就感和满足感。计算机不是只有软件，也不是只有硬件，只有在软硬件和学科交叉相结合之下，计算机的最大作用才能发挥出来，这也是它的魅力所在。

参考文献

- [1] David A. P, John L.H, 计算机组成与设计：硬件/软件接口 [M]. 易江芳, 刘先华. 北京：机械工业出版社，2020.
- [2] 芮雪, 王亮亮, 杨琴. 国产处理器研究与发展现状综述 [J]. 现代计算机 (专业版),2014(08):15-19.
- [3] 潘树朋, 刘有耀.RISC-V 微处理器以及商业 IP 的综述 [J]. 单片机与嵌入式系统应用,2020,20(06):5-8+12.
- [4] 全球首颗智能穿戴领域人工智能芯片发布 [J]. 智能城市,2019,5(10):191.
- [5] 雷思磊.RISC-V 架构的开源处理器及 SoC 研究综述 [J]. 单片机与嵌入式系统应用,2017,17(02):56-60+76.
- [6] 袁攀. 基于嵌入式 RISC-V 微处理器的流水线研究与设计 [D]. 长沙理工大学,2021.DOI:10.26985/d.cnki.gcsjc.2021.000811.
- [7] 李亚民. 计算机原理与设计 :Verilog HDL 版 [M]. 北京: 清华大学出版社,2011.

教师评语评分

评语: _____

评阅人: _____

年 月 日

(备注: 对该实验报告给予优点和不足的评价, 并给出百分制评分。)