

# 合肥工业大学



## 2023~2024 学年 第一学期

## 《系统硬件综合设计》

### 设计报告

班 级\_计算机科学与技术3班\_\_\_\_

学 号\_2021214813\_\_\_\_\_

姓 名\_\_丁盛鹏\_\_\_\_\_

2023 年 12 月

# 目 录

1	设计要求.....	4
1.1	CPU 处理指令的过程.....	4
1.2	指令集.....	4
1.1.1	RISC-V 指令格式.....	4
1.1.2	RISC-V 指令集.....	5
1.3	名称对照表.....	7
1.4	功能描述.....	9
2	设计.....	9
2.1	整体架构.....	9
2.1.1	主要功能部件.....	9
2.1.2	数据通路.....	10
2.2	模块设计.....	16
2.2.1	Data Flip-Flop & Mux.....	17
2.2.2	Instruction Fetch.....	18
2.2.3	Instruction Decode.....	23
2.2.4	Execute.....	33
2.2.5	Memory Access.....	40
2.2.6	Write Back.....	43
3	功能实现.....	44
3.1	单周期 CPU.....	44
3.1.1	模块连接.....	44
3.1.2	时序分析.....	45
3.2	五级流水段.....	46
3.2.1	功能划分.....	47
3.2.2	具体实现.....	48
3.2.3	时序分析.....	49
3.3	流水执行指令.....	50
3.3	分支预测.....	51
3.3.1	静态预测工作机制.....	51
3.3.2	静态预测思路.....	52
3.3.3	静态预测实现.....	53
3.5	数据冒险检测与解决.....	54
3.5.1	数据冒险检测.....	55
3.5.2	数据前推解决冒险.....	58
3.5.3	LOAD 冒险.....	60
4	问题解决与改进.....	63
4.1	问题解决.....	63
4.2	设计改进.....	65
5	结果分析.....	66
5.1	测试数据与代码.....	66

---

5.2	仿真结果.....	69
5.3	开发板运行.....	78
5.3.1	八位七段数码管.....	78
5.3.2	开发板实现.....	81
5.3.3	运行结果.....	82
	总结.....	82
	参考文献.....	82
	附件.....	82

# 1 设计要求

基于先修课程，根据系统设计思想，使用硬件描述语言设计实现一款基于 MIPS32, ARM, RISC-V 或者自定义指令集的微处理器（CPU）。要求：完成单周期 CPU 设计，或多周期 CPU 设计，或 5 级流水线 CPU 设计（递进式、难度依次提升。所有学生必须至少完成单周期 CPU 的设计工作），并将设计的 CPU。以此贯穿数字逻辑、计算机组成原理、计算机体系结构课程，实现从逻辑门至完整 CPU 处理器的设计<sup>[1]</sup>。

## 1.1 CPU 处理指令的过程

CPU 处理指令分为五个过程，按顺序依次为取指令->指令译码->执行指令->存取数据->数据写回寄存器。下面简单介绍每个过程。

**取指（Instruction Fetch）：**根据指令计数器（PC）中的地址，将指令从内存（或缓存）中取出，存储到指令寄存器（IR）中。这个过程对应一个机器周期。

**指令译码（Instruction Decode）：**CPU 将指令寄存器（IR）中的指令进行解析，并确定执行当前指令所需的操作。这个过程对应一个机器周期。

**执行（Execution）：**根据指令寄存器（IR）中的操作码和操作数，进行相应的运算和处理。这个过程通常需要多个机器周期。

**存储（Memory Access）：**存取操作数或将数据写入内存（或缓存）。这个过程通常需要一个或多个机器周期。

**写回（Write Back）：**将执行结果写回到寄存器或内存中，保持指令执行的正确性。这个过程通常需要一个机器周期。

## 1.2 指令集

### 1.1.1 RISC-V 指令格式

表格 1 RISC-V 指令格式

31 27 26 25	24 20	19 15	14 12	11 7	6 0	
Funct7	Rs2	Rs1	Funct3	rd	Opcode	R-type

Imm[11:0]		Rs1	Funct3	rd	Opcode	I-type
Imm[11:5]	Rs2	Rs1	Funct3	Imm[4:0]	Opcode	S-type
Imm[12 10:5]	Rs2	Rs1	Funct3	Imm[4:1 11]	Opcode	B-type
Imm[31:12]				Rd	Opcode	U-type
Imm[20 10:1 11 19:12]				rd	opcode	J-type

### 1.1.2 RISC-V 指令集

表格 2 RISC-V 指令集

Inst	Name	FM T	Opcod e	Funct 3	Funct7	Description	Note
add	ADD	R	011001 1	0x0	0x00	$Rd = rs1 + rs2$	
sub	SUB	R	011001 1	0x0	0x20	$Rd = rs1 - rs2$	
xor	XOR	R	011001 1	0x4	0x00	$Rd = rs1 \wedge rs2$	
or	OR	R	011001 1	0x6	0x00	$Rd = rs1   rs2$	
and	AND	R	011001 1	0x7	0x00	$Rd = rs1 \& rs2$	
sll	Shift Left Logical	R	011001 1	0x1	0x00	$Rd = rs1 \ll rs2$	
srl	Shift Right Logical	R	011001 1	0x5	0x00	$Rd = rs1 \gg rs2$	
sra	Shift Right Arith	R	011001 1	0x5	0x20	$Rd = rs1 \ggg rs2$	Msb- exten ds
slt	Set Less Than	R	011001 1	0x2	0x00	$Rd = (rs1 < rs2) ? 1 : 0$	
sltu	Set Less Than(U)	R	011001 1	0x3	0x00	$Rd = (rs1 < rs2) ? 1 : 0$	Zero- exten ds
add	ADD	I	001001	0x0		$Rd = rs1 + imm$	

i	Immediate		1				
xori	XOR Immediate	I	001001 1	0x4		$Rd = rs1 \wedge imm$	
ori	OR Immediate	I	001001 1	0x6		$Rd = rs1 \mid imm$	
andi	AND Immediate	I	001001 1	0x7		$Rd = rs1 \& imm$	
slli	Shift Left Logical Imm	I	001001 1	0x1	Imm[5:11]=0 x00	$Rd = rs1 \ll imm[0:4]$	
srli	Shift Right Logical Imm	I	001001 1	0x5	Imm[5:11]=0 x00	$Rd = rs1 \gg imm[0:4]$	
srai	Shift Right Arith Imm	I	001001 1	0x5	Imm[5:11]=0 x20	$Rd = rs1 \ggg imm[0:4]$	Msb-extends
slti	Set Less Than Imm	I	001001 1	0x2		$Rd = (rs1 < imm) ? 1 : 0$	
sltiu	Set Less Than Imm(U)	I	001001 1	0x3		$Rd = (rs1 < imm) ? 1 : 0$	Zero-extends
lb	Load Byte	I	000001 1	0x0		$Rd = M[rs1+imm][0:7]$	
lh	Load Half	I	000001 1	0x1		$Rd = M[rs1+imm][0:15]$	
lw	Load Word	I	000001 1	0x2		$Rd = M[rs1+imm][0:31]$	
lbu	Load Byte(U)	I	000001 1	0x4		$Rd = M[rs1+imm][0:7]$	Zero-extends
lhu	Load	I	000001	0x5		$Rd = M[rs1+imm]$	Zero-

	Half(U)		1			][0:15]	exten ds
sb	Store Byte	S	010001 1	0x0		M[rs1+imm][0: 7] = rs2[0:7]	
sh	Store Half	S	010001 1	0x1		M[rs1+imm][0: 15] = rs2[0:15]	
sw	Store Word	S	010001 1	0x2		M[rs1+imm][0: 31] = rs2[0:31]	
beq	Branch ==	B	110001 1	0x0		If(rs1==rs2)PC +=imm	
bne	Branch !=	B	110001 1	0x1		If(rs1!=rs2)PC+ =imm	
blt	Branch <	B	110001 1	0x4		If(rs1<rs2)PC+ =imm	
bge	Branch >=	B	110001 1	0x5		If(rs1>=rs2)PC +=imm	
bltu	Branch <(U)	B	110001 1	0x6		If(rs1<rs2)PC+ =imm	Zero- exten ds
bgeu	Branch >=( U)	B	110001 1	0x7		If(rs1>=rs2)PC +=imm	Zero- exten ds
lui	Load Upper Imm	U	011011 1			Rd = imm<<12	

### 1.3 名称对照表

表格 3 名称对照表

名称	模块	命名规则	功能
InsMemory	InsMemory	英文名称/英文缩略形式	指令存储器
RegFile	RegFile		寄存器文件
ALU	ALU		算数逻辑单

			元
Ext	Ext		立即数扩展单元
XxxMux	Mux		多路选择器
AluResult	ALU	英文名称/缩略形式	ALU 计算结果
RegOutB/A	RegFile		寄存器 B 读出结果
MemDataOut	DataMemory		数据存储器读出结果
Instruction	InsMemory		指令
InsAddr	Program Counter		指令地址
nextpc	PCsrcMux		下一条指令地址
lw_stall	ConUnit		LOAD 冒险停顿信号
.....	.....		.....
IF/ID/EX/MEM/WB	--	Instruction Fetch/...缩略形式	流水功能段
IF_ID/...	IF_ID/...	段间寄存器采用“前一流水段_后一流水段”的命名方式。如“EX_MEM”。	段间寄存器，传递段间信号与数据
IF_ID_xxx_out	IF_ID	段间寄存器输出信号采用“段间名_信号名_out”的命名方式。 如“IF_ID_instruction_out”	表示段间寄存器向下个流水段的数据输出
ID_EX_xxx_out	ID_EX		
EX_MEM_xxx_out	EX_MEM		
MEM_WB_xxx_out	MEM_WB		
Con_signal_xxx	ConUnit	控制单元中控制信号采用	控制信号



		“Con_signal_信号名”的命名方式。 如“Con_signal_RegWrite”	
Xx_xx_Cs_xxx_out	段间寄存器	段间寄存器输出的控制单元信号采用“段间名_Cs_信号名_out”的命名方式。 如“ID_EX_Cs_sign_out”	传递的控制信号
xxxMuxA/B/Out	Mux	选择器的输入输出采用“选择内容 MuxA/B/Out”的命名方式。 如“BranchMuxA”、“AluSrcMuxOut”	多路选择

## 1.4 功能描述

- 基于 RISC-V 指令集的 34 条指令功能实现，包括 10 条 R 类型指令、14 条 I 类型指令、3 条 S 类型指令、6 条 B 类型指令、1 条 U 类型指令。
- 基于哈佛存储结构的五级流水段指令执行。
- 检测流水线中的数据冒险，并解决 3 种写后读冒险。
- 基于静态分支预测的分支跳转实现。
- 完成对八位数码管的显示控制，并下载至 FPGA 开发板（ego-1）上运行，可正确测试简单程序。

## 2 设计

### 2.1 整体架构

#### 2.1.1 主要功能部件

- 程序计数器：根据传来的下一条指令的地址，输出下一条指令的地址。
- 指令存储器：根据指令地址取出指令。

- 寄存器文件：包含 30 个通用寄存器，用于指令译码和计算。
- 立即数扩展单元：对指令种立即数整合并扩展。
- 控制单元：对指令译码，检测冒险。
- 算数逻辑单元：完成指令种数据的计算要求。
- 分支控制单元：控制产生分支信号。
- 数据存储器：存储数据，根据地址与外界交换数据。
- 数据选择器：对输入的 2 或 3 个数据根据控制信号选择输出。

## 2.1.2 数据通路

接下来介绍 CPU 架构中各指令类型的数据通路，包括指令计算中数据所通过的模块和具体线路。通路类型包括 R、I、I(LOAD)、B、U、S 类型。

### ● R-type 数据通路

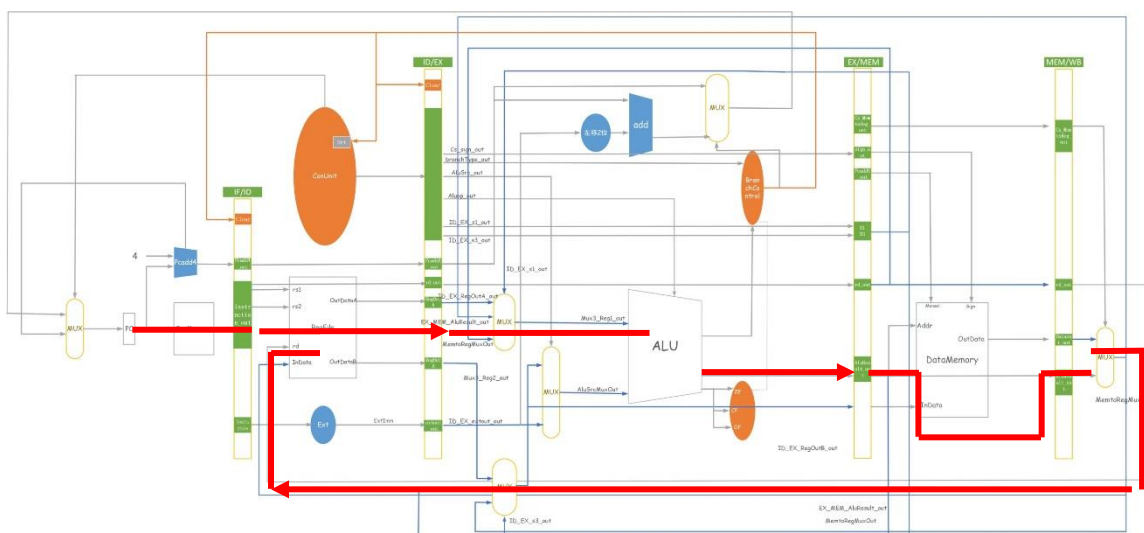


图 1 R-type 数据通路

R 类型指令的数据通路大致相同，步骤如下。（此处不考虑数据冒险的处理，数据冒险在之后的章节进行讲解）

- PC 产生下一指令地址送入到 InsMemory 中，在 InsMemory 中根据地址将指令取出。
- 对取出的指令进行译码（为简便图形，图中省略传入 ConUnit 的通路线），在 RegFile 中取出指令所指定的两个寄存器中的值。
- 将两个数据送入 ALU 中根据控制信号和 ALU 操作信号进行计算，将结果传入到 MEM 阶段。

- iv. 在 MEM 阶段，由于 R 类型指令不进行访存操作，故将 ALU 结果直接传入 WB 阶段。
- v. 在 WB 阶段，根据控制信号对 ALU 结果和 DataMemory 结果进行选择，选择 ALUResult 根据译码阶段产生的写回寄存器编号写回到寄存器中。

- I-type 数据通路

I 类型指令数据通路大致相同（除 LOAD 类指令），步骤如下。（此处不考虑数据冒险的处理，数据冒险在之后的章节进行讲解）

- i. PC 产生下一指令地址送入到 InsMemory 中，在 InsMemory 中根据地址将指令取出。
- ii. 对取出的指令进行译码（为简便图形，图中省略传入 ConUnit 的通路线），在 RegFile 中取出指令所指定的第一个寄存器中的值。
- iii. 将指令中的立即数进行扩展。
- iv. 根据译码选择信号选择立即数作为第二个操作数，同时第一个操作数为寄存器 A 中的值。在 ALU 模块中进行计算，将结果送到 MEM 阶段。
- v. 由于 I 类型指令不进行访存操作（除 LOAD 类指令），所以直接将 ALUResult 送到 WB 阶段。
- vi. 在 WB 阶段，根据控制信号对 ALU 结果和 DataMemory 结果进行选择，选择 ALUResult 根据译码阶段产生的写回寄存器编号写回到寄存器中。

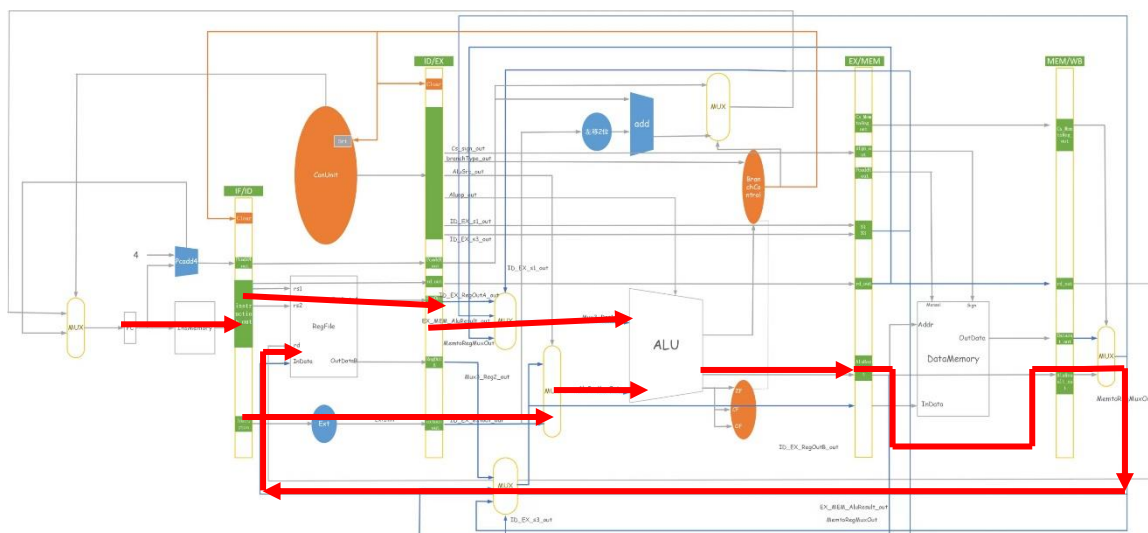


图 2 I-type 数据通路

### ● I-type (LOAD) 数据通路

- i. PC 产生下一指令地址送入到 InsMemory 中，在 InsMemory 中根据地址将指令取出。
- ii. 对取出的指令进行译码（为简便图形，图中省略传入 ConUnit 的通路线），在 RegFile 中取出指令所指定的第一个寄存器中的值。
- iii. 对指令中的立即数进行扩展。
- iv. 根据译码选择信号选择立即数作为第二个操作数，同时第一个操作数为寄存器 A 中的值。在 ALU 模块中进行计算，将结果送到 MEM 阶段。
- v. 由于 LOAD 类型指令需要读存储器，所以将 ALU 结果作为访存地址，并从 DataMemory 中取出对应的数据，送往 WB 阶段。
- vi. 在 WB 阶段，根据控制信号对 ALU 结果和 DataMemory 结果进行选择，选择 DataMemory 根据译码阶段产生的写回寄存器编号写回到寄存器中。

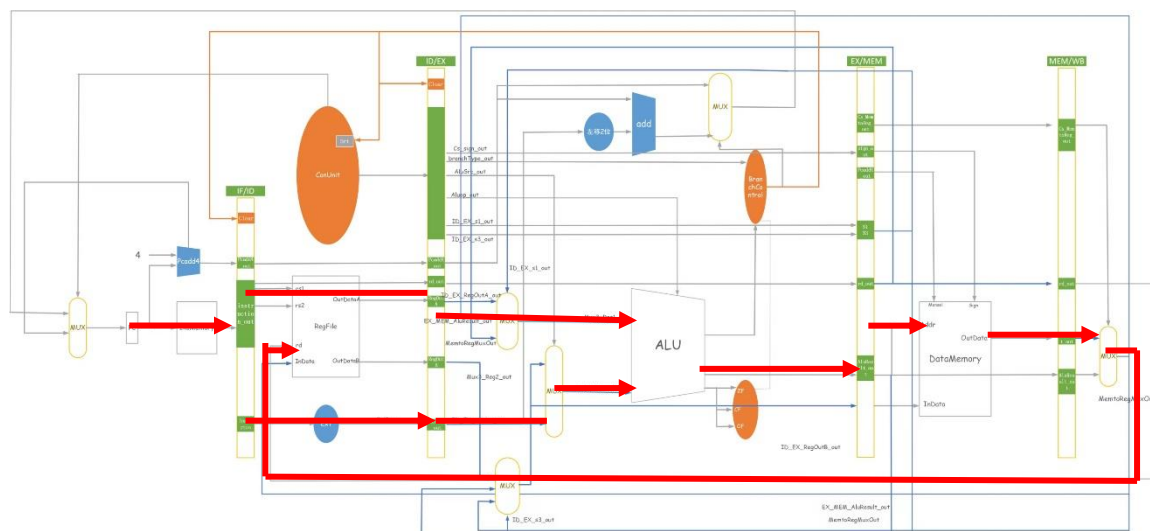


图 3 I-type LOAD 数据通路

- B-type 数据通路

- PC 产生下一指令地址送入到 InsMemory 中，在 InsMemory 中根据地址将指令取出。
- 对取出的指令进行译码（为简便图形，图中省略传入 ConUnit 的通路线），在 RegFile 中取出指令所指定的两个寄存器中的值。
- 对指令中的立即数进行扩展。
- 将两个寄存器取出的数据送入 ALU 中根据控制信号和 ALU 操作信号进行计算，将产生的结果和设置的标志位送到 BranchCon 模块中进行下一步的跳转判断。
- 将扩展后的立即数左移两位作为分支跳转地址送往选择器。
- BranchCon 模块中根据传入的 B 指令类型、ALU 结果和标志判断此类型的 B 指令是否跳转，产生 branchE 信号。
- branchE 信号传入 IF\_ID、ID\_EX、ConUnit 产生停机操作，传入 BranchMux、PCsrcMux 模块选择下一指令地址。

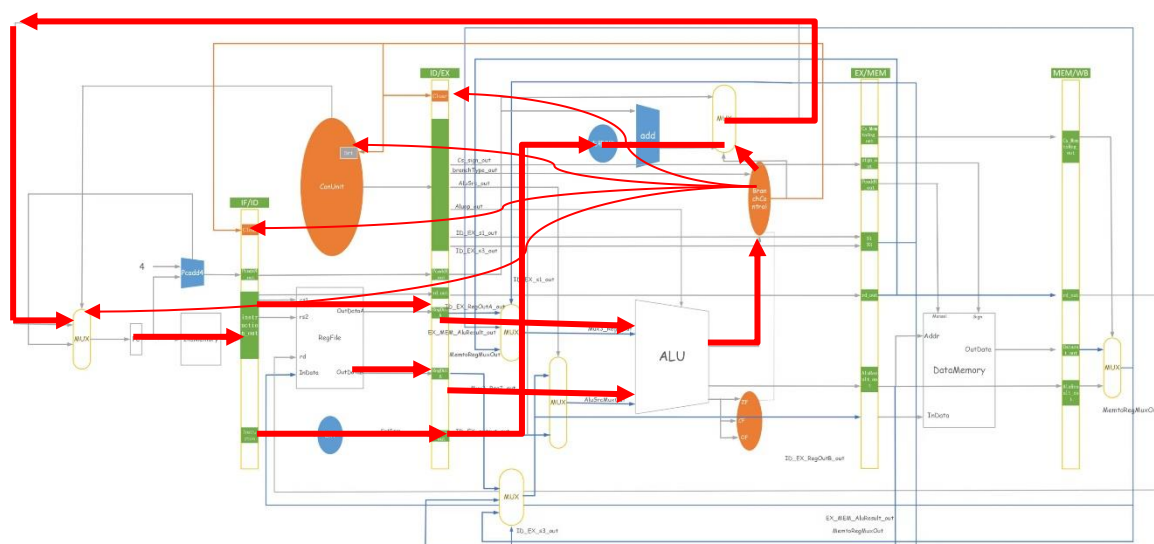


图 4 B-type 数据通路

● S-type 数据通路

- i. PC 产生下一指令地址送入到 InsMemory 中，在 InsMemory 中根据地址将指令取出。
- ii. 对取出的指令进行译码（为简便图形，图中省略传入 ConUnit 的通路线），对指令中的立即数扩展传入到 EX 阶段。  
将 RegFile 中的两个寄存器的数据取出，向后传递到 MEM。
- iii. 根据控制信号在 AluSrcMux 中选择立即数扩展结果送入 ALU 作为操作数。ALU 的另一个操作数为寄存器 1 的值。  
在 ALU 中根据控制单元产生的 ALUOp 信号进行加法计算。将结果送入 MEM 阶段作为访存地址。。
- iv. 访存阶段使用 AluResult 的值作为访存地址，将 ID 阶段读出的寄存器 2 的数据存入存储器。
- v. WB 阶段 S 类型指令不回写。

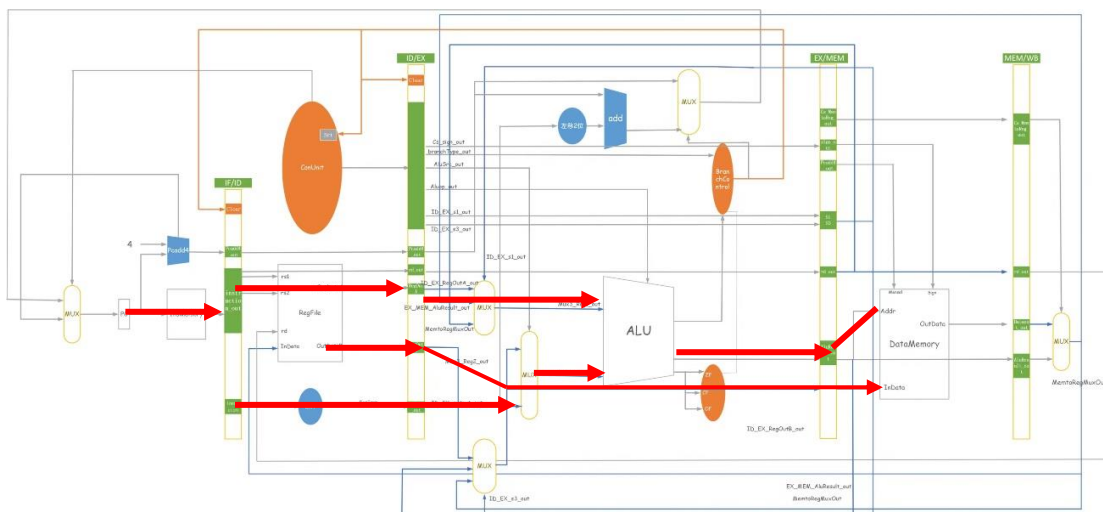


图 5 S-type 数据通路

### ● U-type 数据通路

- i. PC 产生下一指令地址送入到 InsMemory 中，在 InsMemory 中根据地址将指令取出。
- ii. 对取出的指令进行译码（为简便图形，图中省略传入 ConUnit 的通路线），对指令中的立即数扩展传入到 EX 阶段。
- iii. 根据控制信号在 AluSrcMux 中选择立即数扩展结果送入 ALU 作为操作数。

在 ALU 中根据控制单元产生的 ALUOp 信号进行计算。将结果送入 MEM 阶段。

- iv. 由于 U 类型指令不进行访存操作，所以直接将 ALU 结果送入 WB 阶段。
- v. 在 WB 阶段，根据控制信号对 ALU 结果和 DataMemory 结果进行选择，选择 ALUResult 根据译码阶段产生的写回寄存器编号写回到寄存器中。

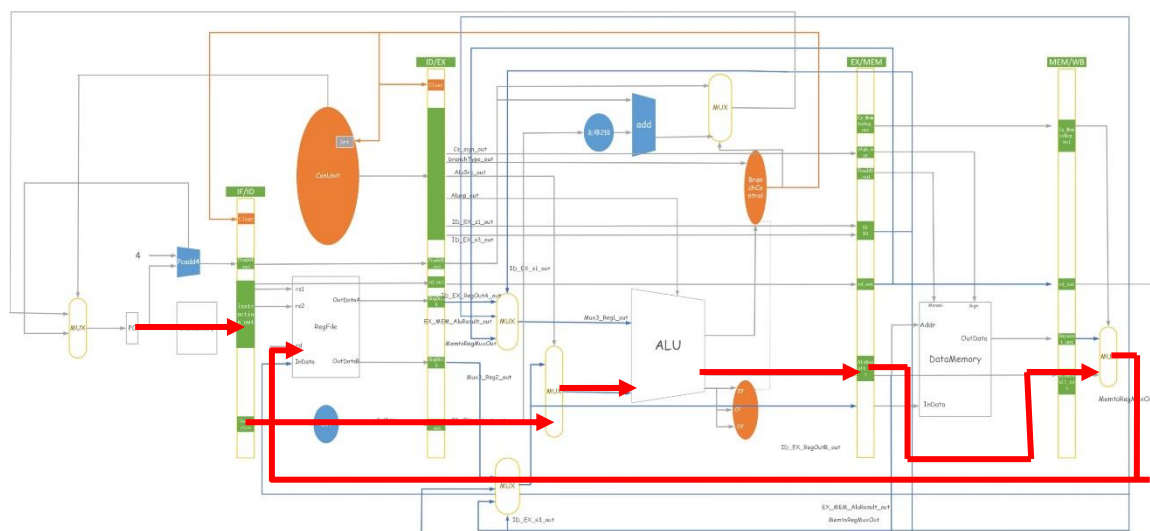


图 6 U-type 数据通路

## 2.2 模块设计

在以下内容中将展示 CPU 架构中各个模块的详细设计与功能实现。下图为工程目录结构。



图 7 工程目录

在 MAIN 模块作为顶层模块包含 CPU 模块和开发板显示模块 HEX8，CPU 模块执行的输出值作为 HEX8 的输入，在 ego 开发板上显示。



## 2.2.1 Data Flip-Flop & Mux

### Data Flip-Flop

Data flip-flop，简称为 dff，数据触发型触发器。具有记忆功能，用于存储数据的触发器，功能为将输入的数据存储一个周期，在下一周期信号的上升沿被触发，将数据送往输出端口。

设计中设置并使用了多种触发器，如 32bit、5bit、4bit、3bit、2bit、1bit 触发器，用于在段间寄存器中维持数据以实现流水段数据互通。下图为 32bit 的 D 触发器。

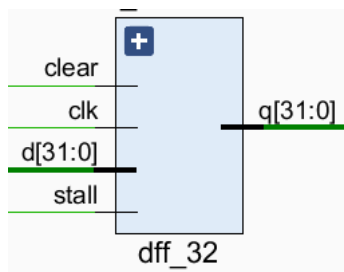


图 8 Data\_Flip-Flop 设计图

表格 4 Data\_Flip\_Flop 信号表

名称	输入输出	功能
D	Input[]	D 触发器的输入，不同位数
Clk	Input	时钟信号
Clear	Input	清除信号，清空触发器，在输出段输出 0
Stall	Input	暂停信号，维持原有数据输出
Q	Output[]	输出端

### Mux

Multiplexer，多路复用器，简称为 Mux，多路选择器 Mux 是一个多输入、单输出的组合逻辑电路，一个 n 输入的多路选择器就是一个 n 路的数字开关，可以根据通道选择控制信号的不同，从 n 个输入中选取一个输出到公共的输出端。

设计中实现了 32bit 二路选择器、32bit 三路选择器，主要用在功能部件的选择输入。下图为 32bit 三路选择器。

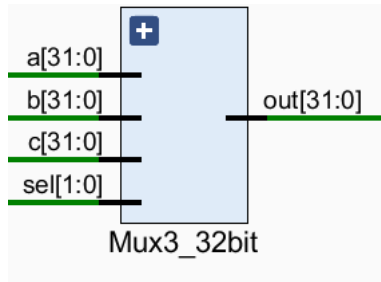


图 9 Mux3\_32bit 设计图

表格 5 Mux 信号表

名称	输入输出	功能
a	Input[]	a 输入端口
b	Input[]	b 输入端口
c	Input[]	c 输入端口
Sel	Input	三个端口选择信号
out	Output[]	输出端口

### 2.2.2 Instruction Fetch

Instruction fetch 阶段，简称为 IF 阶段，是五级流水线中起始的阶段。在本阶段中，各模块相互协调完成取指令操作。由 PC（程序计数器）计算下一条指令的地址传给 InsMemory（指令存储器），后者会在存储器内根据传入的指令地址取出相应位置上的指令，送到下一个流水阶段。下面是 IF 段内各模块的具体实现。

#### PC

Program counter，程序计数器，主要功能为产生下一条指令地址，根据暂停信号暂停一个周期产生地址。

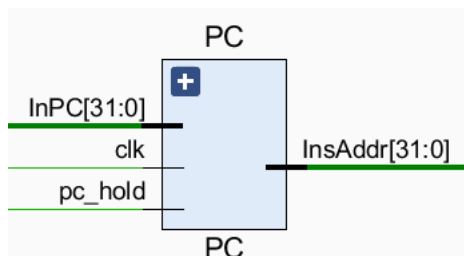


图 10 PC 设计图

表格 6 PC 信号表

名称	输入输出	功能
----	------	----

InPC	Input[31:0]	输入的下一条指令地址
Clk	Input	时钟信号
Pc_hold	Input	用于 load 指令的周期暂停，上升沿时检测到有效，将 PC 暂停一个周期。
InsAddr	Output[31:0]	输出的取指令地址

### 具体实现

增加两个寄存器 hold\_PC 和 pre\_PC 分别保存上个周期 PC 和上上个周期 PC，用于实现周期停顿（LOAD 指令所需）。

```
begin
```

```
    cur_pc<=InPC;
```

```
    pre_pc<=hold_pc;
```

```
    hold_pc<=InPC;
```

```
end
```

（上）在时钟上升沿来临时，当前 PC(cur\_PC)=传入的 PC；上个周期的 PC(hold\_PC)更新为本周期的 PC，为下个周期使用；上上个周期的 PC(pre\_PC)更新为上个周期的 PC(hold\_PC)。

```
if(pc_hold)begin
```

```
    cur_pc<=pre_pc;
```

```
    pre_pc<=hold_pc;
```

```
    hold_pc<=pre_pc;
```

```
end
```

当检测到停顿信号的上升沿时，PC 应停顿一个周期，具体表现为本周期 PC 的值等于上个周期的值。此处（上）的 PC 赋值在问题解决模块会详细介绍。

```
if(InsAddr==4)begin//
```

```
    cur_pc<=InsAddr+4;
```

```
    pre_pc<=hold_pc;
```

```
hold_pc<=InsAddr+4;
```

```
end
```

（上）第一次取指直接加四，因为 PC 初始赋成-4（为了第一次产生 PC 的值为 0），也同样因为第一次输入 InPC（来自 PCsrcMux）为高阻，因为至少要等到 IF 阶段完成后产生 PC+4 才能有值。

### PCadd4

Pc add 4，pc+4 即本次的指令地址加四，主要功能为将本次的 pc 地址加四，产生下一条指令地址。

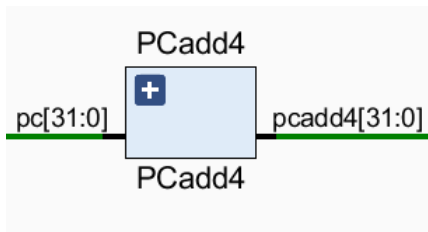


图 11 PCadd4 设计图

表格 7 PCadd4 信号表

名称	输入输出	功能
Pc	Input[31:0]	本周期 PC 产生的指令地址
Pcadd4	Output[31:0]	下一周期输入 PC 的地址选择之一，即紧跟的下一条指令的地址。

### 具体实现

将输入的本次 PC 值与常数 4 相加，得到程序顺序执行的下一指令地址 pcadd4。

```
assign pcadd4 = pc+4;
```

### PCsrcMux

PC source mux，PC 源多路选择器，实例化 Mux2\_32bit 这个 32 位两路选择器，根据 sel 选择信号选择输出 PC+4 或者 B 指令跳转地址两个输入的指令地址。

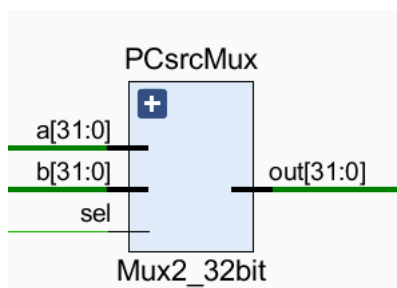


图 12 PCsrcMux 设计图

表格 8 PCsrcMux 信号表

名称	输入输出	功能
a(BranchMuxOut)	Input[31:0]	分支指令的跳转地址
b(pcadd4)	Input[31:0]	程序顺序执行时的下一条指令地址
Sel(branchE)	Input	分支指令是否跳转的控制信号
Out	Output[31:0]	送入 PC 的下一条指令地址

### 具体实现

例化 Mux2\_32bit 这个 32 位 2 路选择器，在两个输入分别传入跳转地址和顺序执行地址，根据是否跳转指令 branchE 选择输出为下一指令地址。

```
Mux2_32bit PCsrcMux(
    .a(BranchMuxOut),.b(pcadd4),.sel(branchE),.out(nextpc)
);
```

### InsMemory

Instruction memory，指令存储器，主要功能为根据输入的指令地址，在存储器中寻址并取出对应位置上的指令输出。

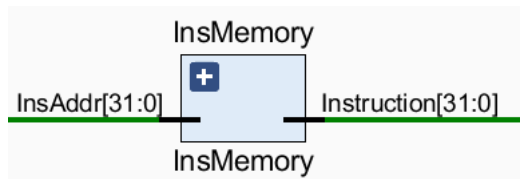


图 13 InsMemory 设计图

表格 9 InsMemory 信号表

名称	输入输出	功能
InsAddr	Input[31:0]	指令地址
Instruction	Output[31:0]	指令

### 具体实现

使用指令长度个 8 位寄存器模拟存储器的实现，用于存储指令。

```
reg[7:0] InsMem[0:length];
```

指令地址为触发信号，按地址取出指令，逐字节按小端存储方式读取。

```
always@(InsAddr)begin //按字节读取指令
```

```
    Instruction[7:0]<=InsMem[InsAddr];
```

```
    Instruction[15:8]<=InsMem[InsAddr+1];
```

```
    Instruction[23:16]<=InsMem[InsAddr+2];
```

```
    Instruction[31:24]<=InsMem[InsAddr+3];
```

```
end
```

### IF\_ID

IF 阶段和 ID 阶段的段间寄存器，实例化 2 个 32 位 D 锁存器，完成对数据锁存一个时钟周期的操作。IF\_ID 连接 IF 和 ID 阶段，将 IF 段产生的之后要用到的数据传到 ID 阶段，是多周期与流水线实现的重要组成部分。

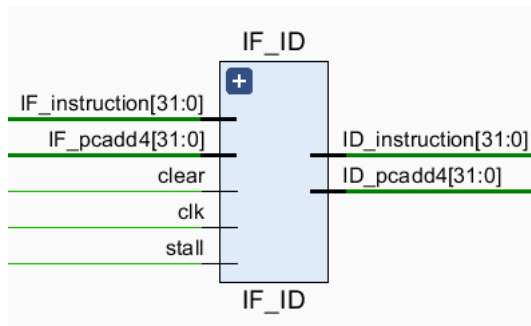


图 14 IF\_ID 设计图

表格 10 IF\_ID 信号表

名称	输入输出	功能
IF_instruction	Input[31:0]	IF 阶段产生的指令
IF_pcadd4	Input[31:0]	IF 阶段产生的顺序执行时下一条指令地址
Clear	Input	清空寄存器信号，将所有寄存器清位
Clk	Input	时钟信号
Stall	Input	暂停信号，将之前的数据维持一个周期
ID_instruction	Output[31:0]	向 ID 段送出的指令
ID_pcadd4	Output[31:0]	向 ID 段送出的顺序执行时下一条指令地址

### 具体实现

将 instruction、pcadd4 锁存一个周期后向后传递，使用的是 32 位 D 触发器。如下以 instruction 为例。

```
dff_32 dff_instruction(
    .clk(clk),.d(IF_instruction),.q(ID_instruction),.clear(clear),.stall(stall)
);
```

### 2.2.3 Instruction Decode

Instruction decode 阶段，简称为 ID 阶段。在本阶段中，各模块相互协调完成以

下工作：

- 译码，根据 IF 阶段传来的指令进行指令译码。
- 立即数产生，对 I 类型、U 类型、B 类型的立即数扩展。
- 数据冒险检测，对 EX 冒险和 MEM 冒险进行检测。
- 数据冒险解决，对检测到的冒险发出控制信号，控制数据前推与周期停顿以解决冒险。

## ConUnit

Control Unit，控制单元，主要完成指令译码并产生控制信号、数据冒险检测并产生数据前推及周期暂停的控制信号。

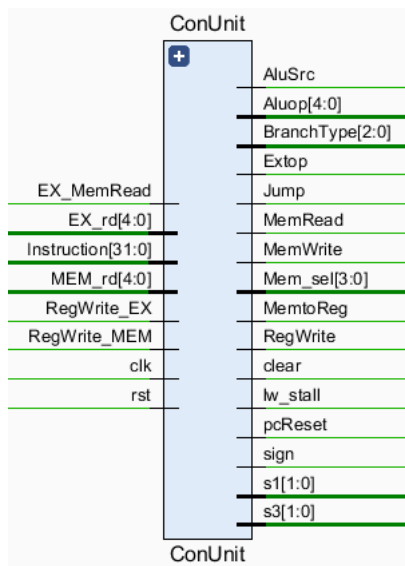


图 15 ConUnit 设计图

表格 11 ConUnit 信号表

名称	输入输出	功能
Instruction	Input[31:0]	指令输入
Clk	Input	时钟信号
Rst	Input	停机信号
RegWrite_EX	Input	译码本条指令时，此时正处于 EX 阶段的指令写寄存器的信号
RegWrite_MEM	Input	译码本条指令时，此时正



		处于 MEM 阶段的指令写寄存器的信号
EX_MemRead	Input	译码本条指令时，此时正处于 EX 阶段的指令读存储器的信号
EX_rd	Input[4:0]	译码本条指令时，此时正处于 EX 阶段的指令要写入的寄存器
MEM_rd	Input[4:0]	译码本条指令时，此时正处于 MEM 阶段的指令要写入的寄存器
AluSrc	Output	AluY 的输入选择信号，选择寄存器 B 中的数据或者是扩展后的立即数
Aluop	Output[4:0]	产生的 ALU 运算符，选择 ALU 执行的运算
BranchType	Output[2:0]	B 指令的类型
Extop	output	Ext 扩展立即数时是否进行有符号扩展
Jump	output	是否 J 跳转信号
MemRead	output	是否读存储器信号
MemWrite	output	是否写存储器信号
Mem_sel	Output[3:0]	选择访问存储器的位数，字节或者半字或者字。 4'b0001 按字节访问， 4'b0011 按半字访问， 4'b1111 按字访问
MemtoReg	Output	选择时 Alu 结果还是存储器取出的数据送入寄存器

RegWrite	Output	是否写入寄存器信号
Clear	Output	清除段间寄存器信号
Lw_stall	Output	Load 指令冒险时暂停一个周期信号
PcReset	Output	重置 PC 信号
sign	Output	Load 指令按字节访问或半字访问数据存储器时，高位是否符号扩展。
S1	Output[1:0]	选择 ALUX 前是 EX 段或 MEM 段数据前推的数据还是 RegOutA 寄存器 A 输出端口的数据
S3	Output[1:0]	选择 ALUY 前是 EX 段或 MEM 段数据前推的数据还是 RegOutA 寄存器 B 输出端口的数据

### 具体实现

表格 12 ConUnit 控制信号条件表

控制信号	条件成立	条件不成立	条件
Jump	1 (JumpMux=JumpAddr)	0(JumpMux=BranchAddr or pcadd4)	Opcode==J
BranchType[2:0]	Xxx (branch taken, branchE=1,branchMux=branchAddr,pcSrcMux=branchAddr)	000 (branch not taken,branchE=0,branchMux=pcadd4,pcSrcMux=pcadd4)	Opcode==B
MemRead	1(memory read enabled)	0(memory read disabled)	(Rst!=1)&&(Opcode==LOAD)
MemtoR	1(MemtoRegMux=aluResult to)	0(MemtoRegMux=da)	Opcode==R I U

eg	reg)	ta memory to reg)	
Aluop[4:0]	Xxxxx		All inst
MemWrite	1(memory write enabled)	0(memory write disabled)	Opcode==S
AluSrc	1(AluSrcMux=Imm to AluY)	0(AluSrcMux=RegOutBor data forwarding to AluY)	Opcode==I LOAD S U
RegWrite	1(regfile write enabled)	0(regfile write disabled)	Opcode==R I LOAD U
pcReset	X	x	none
Sign	1(memory access msb extends)	0 x(memory access zero extends)	Inst!=lbu lhu
Extop	0		all
Mem_sel[3:0]	Byte:0001;half:0011;word:1111	xxxx	Opcode==LOAD S
s1[1:0]	EX:00;MEM:01(Reg1Mux=data forwarding)	10(Reg1Mux=RegOutA)	Rs1 data hazard
s3[1:0]	EX:00;MEM:01(Reg2Mux=data forwarding)	10(Reg2Mux=RegOutB)	Rs2 data hazard
lw_stall	1(PC hold, IF_ID clear)	0	LOAD data hazard
clear	0		all

如下以 R-Type 的 ADD 指令为例。取出指令中 opcode、funct3、funct7，根据 opcode 判断指令类型；根据指令类型设置公共的控制信号；根据 funct3 和 funct7 判断具体指令，设置不同的 ALU 操作类型。

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

```
opcode <= Instruction[6:0];
```

```
funct3 <= Instruction[14:12];
```

```
funct7 <= Instruction[31:25];
```

```
case(opcode)
```

```
7'b0110011:begin//R-TYPE
    Jump<=0;
    BranchType<=3'b000;
    MemRead<=0;
    MemtoReg<=1;//ALU
    MemWrite<=0;
    AluSrc<=0;//REG
    RegWrite<=1;
    pcReset<=0;
    Extop<=0;
    case({funct3,funct7})
    10'b000_0000000:begin Aluop<=5'b00001;end//add
```

我们将数据冒险检测 and 解决放到功能实现来说明。

RegFile

Register file，寄存器文件，含有 32 个通用寄存器，其中 0 号寄存器恒为 0，主要功能为寄存器和外部数据进行交换，可进行写入和读取操作，包含由两个读端口和一个写端口。

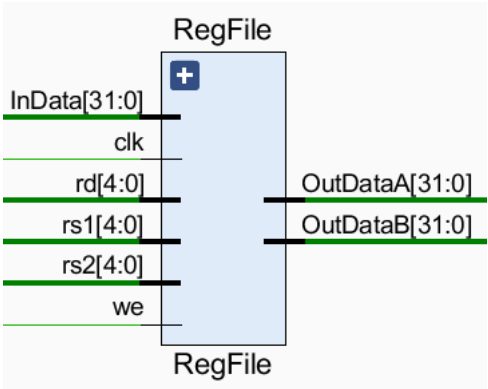


图 16 RegFile 设计图

表格 13 RegFile 信号表

名称	输入输出	功能
Clk	Input	时钟信号

We	Input	写寄存器信号
Indata	Input[31:0]	写入的数据
Rd	Input[4:0]	要写入的寄存器编号
Rs1	Input[4:0]	读寄存器编号 1
Rs2	Input[4:0]	读寄存器编号 2
OutDataA	Output[31:0]	读寄存器 1 的数据
OutDataB	Output[31:0]	读寄存器 2 的数据

### 具体实现

使用寄存器类型的 32 个寄存器作为寄存器文件中的 32 个通用寄存器。

```
reg[31:0] Reg[31:0]; // 32 个 32 位寄存器
```

读取数据时采用总是赋值的方法，但数据是否写入由控制信号决定，所以不用担心读的数据会写入某些地方。根据译码的 rs1 和 rs2 作为读寄存器编号，在 RegFile 中寻找，将数据赋给两个输出端口。

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

```
    OutDataA <= Reg[rs1];
```

```
    OutDataB <= Reg[rs2];
```

```
end
```

写入数据时选择在时钟信号的上升沿，同时需要允许写信号有效。根据输入的写寄存器编号，在 RegFile 中寻找，将数据写入寄存器。

```
always@(posedge clk)begin
```

```
    if(we)begin
```

```
        Reg[rd]<=InData;
```

```
    end
```

```
end
```

**Ext**

Extend unit，立即数扩展单元，主要功能为将指令中的立即数拼接并进行 32 位扩展。

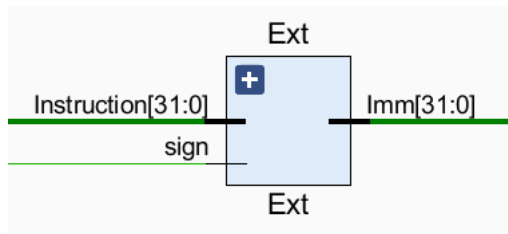


图 17 Ext 设计图

## ID\_EX

ID 阶段和 EX 阶段的段间寄存器，实例化 4 个 32 位、2 个 5 位 D 锁存器、1 个 4 位、1 个 3 位、2 个 2 位、8 个 1 位 D 锁存器，完成对数据锁存一个时钟周期的操作。ID\_EX 连接 ID 和 EX 阶段，将 ID 段产生的之后要用到的数据传到 EX 阶段，是多周期与流水线实现的重要组成部分。

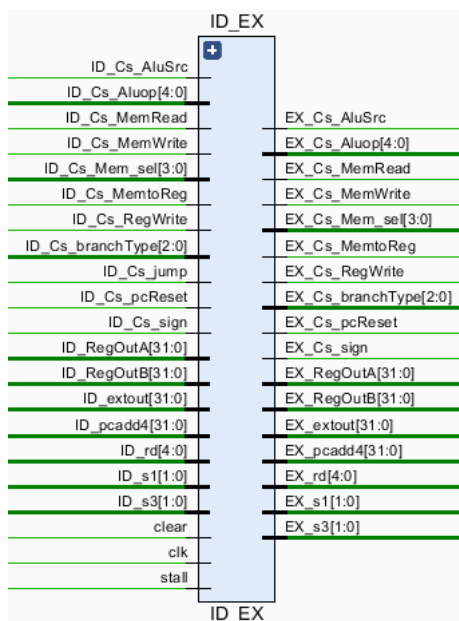


图 18 ID\_EX 设计图

表格 14 ID\_EX 信号表

名称	输入输出	功能
ID_Cs_Alusrc	input	ID 段传来的控制信号，控制选择 ALUY 的输入数据，选择寄存器 B 中的数据或者是扩展后的立即数

ID_Cs_Aluop	Input[4:0]	ID 段传来的控制信号，ALU 运算符，选择 ALU 执行的运算
ID_Cs_MemRead	input	ID 段传来的控制信号，是否读存储器信号
ID_Cs_MemWrite	Input	ID 段传来的控制信号，是否写存储器信号
ID_Cs_Mem_sel	Input[3:0]	ID 段传来的控制信号，选择访问存储器的位数，字节或者半字或者字。 4'b0001 按字节访问， 4'b0011 按半字访问， 4'b1111 按字访问
ID_Cs_MemtoReg	Input	ID 段传来的控制信号，选择时 Alu 结果还是存储器取出的数据送入寄存器
ID_Cs_RegWrite	Input	ID 段传来的控制信号，是否写入寄存器信号
ID_Cs_branchType	Input[2:0]	ID 段传来的控制信号，B 指令的类型
ID_Cs_jump	Input	ID 段传来的控制信号，是否 J 跳转信号
ID_Cs_pcReset	Input	ID 段传来的控制信号，重置 PC 信号
ID_Cs_sign	Input	ID 段传来的控制信号，Load 指令按字节访问或半字访问数据存储器时，高位是否符号扩展
ID_RegOutA	Input[31:0]	ID 段传来的读寄存器 A

		的数据
ID_RegOutB	Input[31:0]	ID段传来的读寄存器B的数据
ID_extout	Input[31:0]	ID 段传来的扩展后的立即数
ID_pcadd4	Input[31:0]	ID 段传来的顺序执行时下一条指令地址
ID_rd	Input[4:0]	ID段传来的在 WB段要写入的寄存器编号
ID_sl	Input[1:0]	ID 段传来的 ALUX 数据前推选择信号
ID_s3	Input[1:0]	ID 段传来的 ALUY 数据前推选择信号
Clear	Input	清除寄存器信号
Clk	Input	时钟信号
stall	input	暂停信号，将之前的数据维持一个周期
EX_Cs_Alusrc	output	向 EX 段送出的 Alusrc 信号
EX_Cs_Aluop	Output[4:0]	向 EX 段送出的 Aluop 信号
EX_Cs_MemRead	Output	向 EX 段送出的 MemRead 信号
EX_Cs_RegWrite	Output	向 EX 段送出的 RegWrite 信号
EX_Cs_MemWrite	Output	向 EX 段送出的 MemWrite 信号
EX_Cs_Mem_sel	Output[3:0]	向 EX 段送出的 Mem_sel 信号



EX_Cs_MemtoReg	Output	向 EX 段送出的 MemtoReg 信号
EX_Cs_branchType	Output[2:0]	向 EX 段送出的 branchType 信号
EX_Cs_pcReset	Output	向 EX 段送出的 pcReset 信号
EX_Cs_sign	Output	向 EX 段送出的 sign 信号
EX_RegOutA	Output[31:0]	向 EX 段送出的寄存器 A 的数据
EX_RegOutB	Output[31:0]	向 EX 段送出的寄存器 B 的数据
EX_extout	Output[31:0]	向 EX 段送出的扩展后的立即数
EX_pcadd4	Output[31:0]	向 EX 段送出的顺序执行下一条指令地址
EX_rd	Output[4:0]	向 EX 段送出的 WB 阶段要写入的寄存器编号
EX_s1	Output[1:0]	向 EX 段送出的 ALUX 数据前推选择器选择信号
EX_s3	Output[1:0]	向 EX 段送出的 ALUY 数据前推选择器选择信号

### 2.2.4 Execute

Execute 阶段，简称为 EX 阶段。在本阶段中，各模块相互协调完成以下工作：

- ALU 运算，对译码段产生的两个操作数（或者数据前推送过来的操作数）进行算数或者逻辑运算。
- 分支地址产生，IF 阶段产生的 PC+4、ID 段产生的分支地址立即数扩展，根据 ID 段的指令译码结果信号，在本段完成下条地址的选择。

- 分支条件判断，根据 ID 段取出的分支条件判断寄存器中的值，在本段对其进行计算，判断分支是否跳转。

## ALU

Algorithm logic unit，算数逻辑单元，主要功能为将输入的 X、Y 两位操作数根据指令译码产生的运算符进行算数或逻辑运算，并产生运算结果，同时将三个标志寄存器置位。

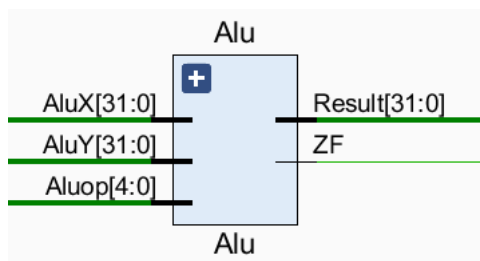


图 19 ALU 设计图

表格 15 ALU 信号表

名称	输入输出	功能
AluX	Input[31:0]	操作数 1
AluY	Input[31:0]	操作数 2
Aluop	Input[4:0]	运算符，指定何种运算
Result	Output[31:0]	运算产生的结果
ZF	Output	零标志位

表格 16 ALU 功能表

Aluop	功能	Aluop	功能
00001	Add	01011	>>>
00010	Sub	01100	Set 1 if <
00011	Addu	01101	Set 1 if <(U)
00100	Subu	01110	None
00101	And	01111	None
00110	Or	10000	Mul
00111	Xor	10001	Div
01000	<<	10010	Mulh
01001	>>	10011	Divu

01010	<<<	10100	Lui(<<12bit)
-------	-----	-------	--------------

AluSrcMux

Algorithm and logic unit source mux，算术逻辑单元源选择器，实例化 32 位二路选择器，主要功能为控制选择 ALUY 操作数的输入数据，是来自寄存器还是来自立即数扩展。

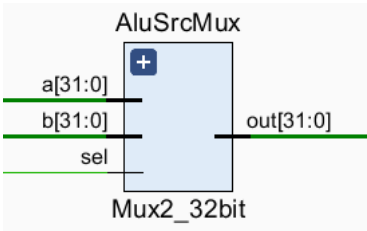


图 20 AluSrcMux 设计图

表格 17 AluSrcMux 信号表

名称	输入输出	功能
a(ID_EX_extout_out)	Input[31:0]	来自立即数扩展单元的数据
b(Mux3_Reg2_out)	Input[31:0]	来自寄存器或者数据前推的数据
Sel(ID_EX_Cs_Alusrc_out)	Input	控制 AluSrcMux 的选择信号
Out	Output[31:0]	数据输出

Mux3\_Reg1

AluX 的数据前推控制选择单元，实例化 32 位三路选择器，主要功能为根据数据前推控制信号选择使用寄存器的数据、EX 段前送来的数据还是 MEM 段前送来的数据。

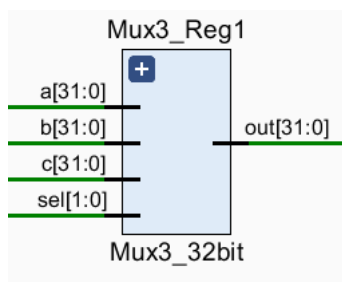


图 21 Mux3\_Reg1 设计图

表格 18 Mux3\_Reg1 信号表

名称	输入输出	功能
a(EX_MEM_AlarResult_out)	Input[31:0]	EX 阶段前送的数据
b(MemtoRegMuxOut)	Input[31:0]	MEM 阶段前送的数据
c(ID_EX_RegOutA_out)	Input[31:0]	正常无冒险时寄存器送来的数据
Sel(ID_EX_s1_out)	Input[1:0]	数据冒险控制选择信号
out	Output[31:0]	数据输出

## Mux3\_Reg2

AluY 的数据前推控制选择单元，实例化 32 位三路选择器，主要功能为根据数据前推控制信号选择使用寄存器的数据、EX 段前送来的数据还是 MEM 段前送来的数据。

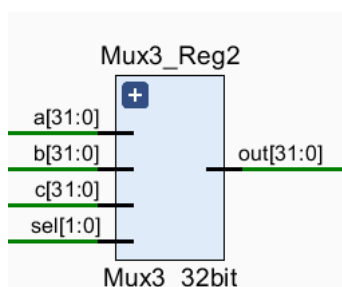


图 22 Mux3\_Reg2 设计图

表格 19 Mux3\_Reg2 信号表

名称	输入输出	功能
a(EX_MEM_AlarResult_out)	Input[31:0]	EX 阶段前送的数据
b(MemtoRegMuxOut)	Input[31:0]	MEM 阶段前送的数据
c(ID_EX_RegOutB_out)	Input[31:0]	正常无冒险时寄存器送来的数据

Sel(ID_EX_s3_out)	Input[1:0]	数据冒险控制选择信号
Out	output[31:0]	数据输出

### BranchCon

Branch control unit，分支控制单元，主要功能为根据 ALU 运算的结果和设置的标志位，对不同的分支指令进行判断是否跳转。

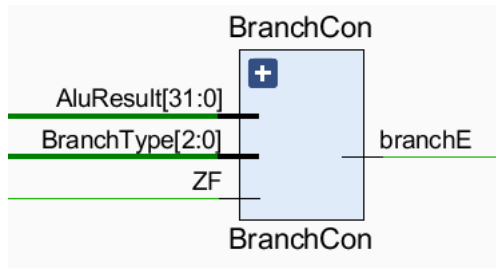


图 23 BranchCon 设计图

表格 20 BranchCon 信号表

名称	输入输出	功能
AluResult	Input[31:0]	ALU 的运算结果
BranchType	Input[2:0]	分支指令类型
ZF	Input	零标志位
branchE	output	分支跳转信号

表格 21 branchCon 功能表

BranchType	类型	branchE
000	none	0
001	Beq	$(ZF == 1) ? 1:0$
010	Bne	$(ZF == 1) ? 0:1$
011	Blt	$AluResult == 1 ? 1:0$
100	Bge	$AluResult == 1 ? 0:1$
101	Bltu	$AluResult == 1 ? 1:0$
110	bgeu	$AluResult == 1 ? 0:1$

### BranchMux

Branch mux，分支地址选择器，实例化 32 位二路选择器，主要功能为根据分支

控制信号选择下一条指令地址是分支地址还是顺序执行程序时的地址。

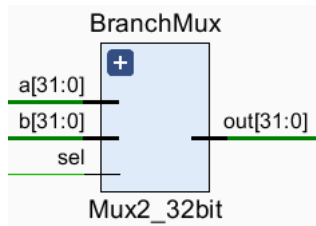


图 24 BranchMux 设计图

表格 22 BranchMux 信号表

名称	输入输出	功能
a(BranchMuxA)	Input[31:0]	分支目标地址
b(ID_EX_pcadd4_out)	Input[31:0]	顺序执行时下一条指令地址
sel(branchE)	Input	分支控制选择信号
Out(BranchMuxOut)	Output[31:0]	数据输出

## EX\_MEM

EX 阶段和 MEM 阶段的段间寄存器，实例化 3 个 32 位、1 个 5 位 D 锁存器、1 个 4 位、6 个 1 位 D 锁存器，完成对数据锁存一个时钟周期的操作。EX\_MEM 连接 EX 和 MEM 阶段，将 EX 段产生的之后要用到的数据传到 MEM 阶段，是多周期与流水线实现的重要组成部分。

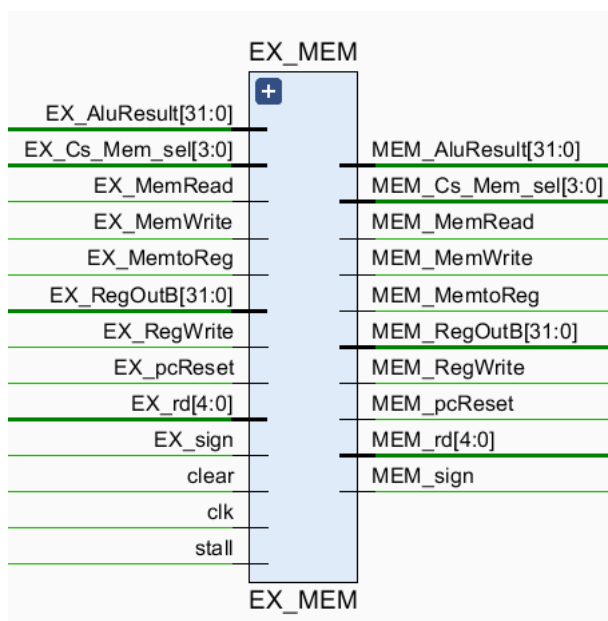


图 25 EX\_MEM 设计图

表格 23 EX\_MEM 信号表

名称	输入输出	功能
EX_AlarResult	Input[31:0]	EX 段传来的 ALU 计算结果
EX_Cs_Mem_sel	Input[3:0]	EX 段传来的存储器访问位数
EX_MemRead	Input	EX 段传来的存储器读信号
EX_MemWrite	Input	EX 段传来的存储器写信号
EX_MemtoReg	Input	EX 段传来的选择 Alu 结果还是存储器取出的数据送入寄存器信号
EX_RegOutB	Input[31:0]	EX 段传来的寄存器 B 数据
EX_RegWrite	Input	EX 段传来的写寄存器信号
EX_rd	Input[4:0]	EX 段传来的写回寄存器编号
EX_sign	Input	EX 段传来的按半字或字节访问存储器时是否符号扩展
Clear	Input	清空寄存器信号
Clk	Input	时钟信号
Stall	Input	暂停信号，将之前的数据维持一个周期
MEM_AlarResult	Output[31:0]	向 MEM 段送出的 ALU 计算结果
MEM_Cs_Mem_sel	Output[3:0]	向 MEM 段送出的按字、

		半字、字节访问存储器信号
MEM_MemRead	Output	向 MEM 段送出的读存储器信号
MEM_MemWrite	Output	向 MEM 段送出的写存储器信号
MEM_MemtoReg	Output	向 MEM 段送出的存储器数据还是 ALU 结果写寄存器的选择信号
MEM_RegOutB	Output[31:0]	向 MEM 段送出的读出的寄存器 B 的数据
MEM_RegWrite	Output	向 MEM 段送出的写寄存器信号
MEM_sign	Output	向 MEM 段送出的按半字或字节访问存储器是是否符号扩展
MEM_rd	Output[4:0]	向 MEM 段送出的写入寄存器的编号

### 2.2.5 Memory Access

Memory Access 阶段，简称为 MEM 阶段。在本阶段中主要完成对 DataMemory（数据存储器）的访问。

#### DataMemory

Data memory，数据存储器，主要功能为存储程序数据，以及与外界交换数据，支持按字、半字、字节访问。



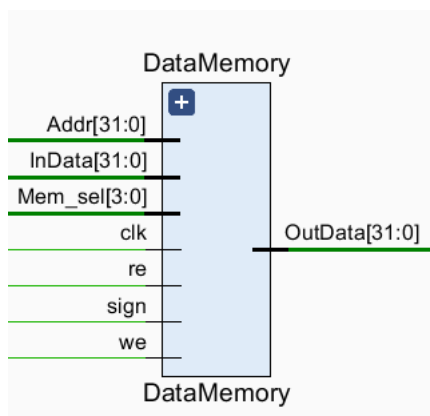


图 26 DataMemory 设计图

表格 24 DataMemory 信号表

名称	输入输出	功能
Addr	Input[31:0]	访问存储器的地址
InData	Input[31:0]	要写入的数据
Mem_sel	Input[3:0]	选择按字、半字还是字节访问的信号
Clk	Input	时钟信号
Re	Input	读存储器信号
Sign	Input	按半字、字节访问时是否符号扩展
we	Input	写存储器信号
OutData	Output[31:0]	要读出的数据

## MEM\_WB

MEM 阶段和 WB 阶段的段间寄存器，实例化 2 个 32 位、1 个 5 位 D 锁存器、3 个 1 位 D 锁存器，完成对数据锁存一个时钟周期的操作。MEM\_WB 连接 MEM 和 WB 阶段，将 MEM 段产生的之后要用到的数据传到 WB 阶段，是多周期与流水线实现的重要组成部分。

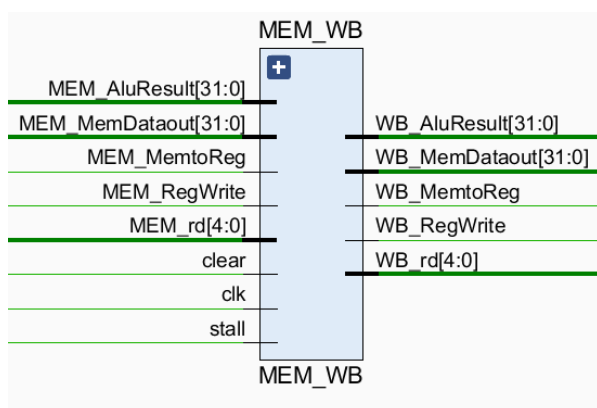


图 27 MEM\_WB 设计图

表格 25 MEM\_WB 信号表

名称	输入输出	功能
MEM_AlarResult	Input[31:0]	MEM 段传来的 ALU 计算结果
MEM_MemDataout	Input[31:0]	MEM 段传来的读取的存储器数据
MEM_MemtoReg	Input	MEM 段传来的选择 AluResult 还是 MemData 写入寄存器
MEM_RegWrite	Input	MEM 段传来的是否写入寄存器控制信号
MEM_rd	Input[4:0]	MEM 段传来的写入的寄存器编号
Clear	Input	清空寄存器信号
clk	Input	时钟信号
Stall	Input	暂停信号，将之前的数据维持一个周期
WB_AlarResult	Output[31:0]	向 WB 段送出的 ALU 结果
WB_MemDataout	Output[31:0]	向 WB 段送出的读存储器结果
WB_MemtoReg	Output	向 WB 段送出的选择 ALU

		结果还是 Mem 数据写入寄存器的控制信号
WB_RegWrite	Output	向 WB 段送出的写寄存器控制信号
WB_rd	Output[4:0]	向 WB 段送出的要写入的寄存器编号

### 2.2.6 Write Back

Write Back 阶段，简称为 WB 阶段。在本阶段中主要完成对寄存器的数据写回操作。根据之前阶段传来的数据和写回地址，将数据写入寄存器中。

#### MemtoRegMux

Memory to register file mux，数据到寄存器选择器，实例化 32 位二路数据选择器，主要功能为根据译码阶段产生的数据选择信号，选择将写入寄存器的数据是 Alu 结果还是 Memory 取出的数据。

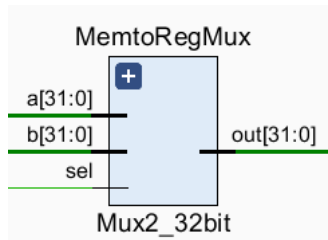


图 28 MemtoRegMux 设计图

表格 26 MemtoRegMux 信号表

名称	输入输出	功能
a(MEM_WB_AlarResult_out)	Input[31:0]	Alu 计算得出的结果
b(MEM_WB_MemDataout_out)	Input[31:0]	从 Memory 中取出的数据
Sel(MEM_WB_Cs_MemtoReg_out)	Input	控制选择信号
Out	Output[31:0]	数据输出

## 3 功能实现

之前的章节中介绍了 CPU 的整体设计和各个模块的详细设计，接下来我们将各个模块综合起来实现其应有的功能。3.1-3.4 节为实现完成体 CPU 的基本实现过程。

### 3.1 单周期 CPU

在实现各个基本模块的基础上，在顶层模块将各部分按数据通路设计相连，便可实现单周期 CPU。在单周期 CPU 中，各模块根据传入的时钟信号激励运行，相互合作，计算结果。与多周期不同的是，单周期在时钟信号到来时，各个模块同时工作（当然多周期也如此，但此“同时工作”非彼“同时工作”），在一个时钟周期内处理完一条指令。在不考虑时钟延迟的情况下，所有模块所有信号似乎在同一时刻产生，并传遍整个 CPU。下面我们分两个部分介绍这一阶段的实现。

#### 3.1.1 模块连接

在此部分中，将介绍繁杂的各模块是如何相互连接成一个有机的整体。主要是各个端口的连接。

- 取指部分

主要由 PC 产生指令地址（InsAddr），传入 InsMemory 中的指令地址端口。在 InsMemory 中取得指令（Instruction）传出，送往需要指令信息的模块。故其输出端口应连接 InsMemory 的地址输入端口。另外，由于分支跳转，PC 应将 nextPC 输入端口连接分支地址产生模块的输出端口。

- 指令译码

在单周期 CPU 中主要在 ConUnit 中完成对指令的译码操作，而 ConUnit 主要完成两个操作。一是 ConUnit 需要得到指令（instruction）本身，并根据指令结构逐位解析。二是将解析产生的控制信号（signal）传往各个模块的各个端口。比如将控制 ALU 操作选择的信号（Aluop）送到 ALU 相应的端口（Aluop）中。所以在 ConUnit 的输入端口应获得 InsMemory 取出的指令（Instruction），输出端口的各信号连接哥哥需要的模块。

- 立即数产生

立即数产生在 Ext 模块中实现。Ext 模块根据指令值（instruction）对其中的立即数部分进行整合扩展，向外输出最终 32 位的立即数（Ext\_Imm）。在本次 CPU 设计中，立即数的目的地主要为分支跳转地址的产生和 ALU 操作数。故其输入端口应连接 InsMemory 的指令输出以获得指令，输出的立即数送往产生分支地址的模块和 ALU 的输入端口。

- 算数逻辑运算

算数逻辑运算主要由 ALU 模块完成。ALU 模块完成对输入的两个/一个操作数运算操作，并输出结果（AluResult）。根据其功能，其输入数据端口，根据指令的数据通路，需要得到 RegFile 产生的 RegOutA/B 和 Ext 产生的 Ext\_Imm，至于每个端口选择哪个数据，则需要在端口前添加多路选择器 Mux。ALU 的输出端口连接需要得到 ALU 结果的地点。

- 分支地址产生

分支地址主要由立即数经过变换得到。所以分支地址产生的输入为立即数（Ext\_Imm），输出为分支地址（branchAddr）。而分支地址与顺序执行时的地址选择由数据选择器（BranchMux）完成，故选择器两个输入分别为分支地址（BranchMux）和程序顺序执行的地址（pcadd4），输出为 branchMuxAddr，为下一指令地址送往 PC。

- 访问存储器

存储器主要完成数据的读写操作。数据的读取和写入需要根据输入的地址（InAddr）取寻找数据，其输入的地址主要由 ALU 运算结果（AluResult）提供，故输入地址端口连接 AluResult，而输入的数据为寄存器文件中的输出（RegOut），故输入数据连接 RegFile 的输出端口 B。输出为读取到的数据，送往寄存器中，连接 RegFile 的 Indata。

### 3.1.2 时序分析

本模块会根据一个指令执行实例对单周期 CPU 的执行时序进行分析。

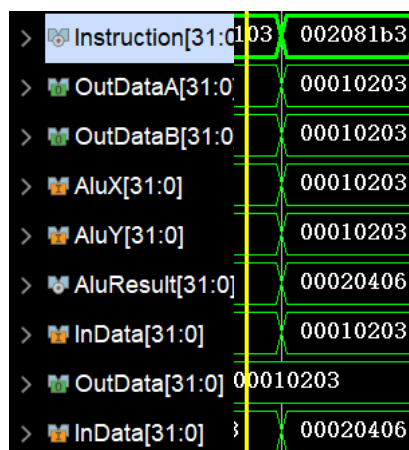


图 29 时序分析示例

单周期 CPU 会在一个周期内所有模块完成工作，在仿真时序图中如上图所示，一个周期内所有模块产生关于同一条指令的数据（对比于多周期一条指令的数据在不同的周期内产生）。具体表现为数据从上到下依次对齐，当然这是考虑在所有模块同一时钟沿触发的基础上。

上图为一条 ADD 指令在单周期 CPU 中的执行过程。左侧为数据名称，由 PC 在 InsMemory 中取指得到 Instruction（002081b3）；送到寄存器文件中取出相应的寄存器中的数据，分别为 OutDataA（00010203）和 OutDataB（00010203）；ADD 指令执行 ALU 操作数均来自 RegFile，所以两个操作数 X 和 Y 分别对应 OutDataA 和 OutDataB，计算出结果 AluResult 为 00020406；ADD 指令没有访存操作，故跳过；在最后的 Indata 处将数据 00020406 写入到寄存器中。至此完成 ADD 指令的执行过程。

尽管在时序图上显示为一个周期完成所有操作，但这一个周期会在实际中显得十分漫长，从而大大增加了指令执行开销。

### 3.2 五级流水段

单周期 CPU 完成一条指令在冗长的一个周期内，我们可以对不同模块工作进行分类从而将所有工作分在不同的工作段内完成。每个工作段完成分配到的工作，每个工作段占一个时钟周期，这样尽管会增加一条指令完成的时钟周期数，但在每个周期内完成的工作量大大缩减，从而缩短了一个周期的时长。值得一提的是，仅仅划分为多个工作段并没有意义，但这会为后面实现流水线工作打下基础。

本次设计采用五级流水段的工作方式，它们分别是取指阶段 IF、译码阶段 ID、

执行阶段 EX、访存阶段 MEM、回写阶段 WB。通过划分五个流水段，将 CPU 执行一条指令的工作分在不同的段内。下面分三个部分介绍五级流水段的功能与实现。

### 3.2.1 功能划分

在本段中，主要介绍五级流水段 CPU 中各段具体的功能和模块划分。

- 取指阶段 IF

取指阶段的模块主要包含 PC 和 InsMemory，完成产生指令地址（PC）并根据地址在存储器中取指令（InsMemory）的操作。

由 PC 根据 nextPC 判断下一指令地址是来自分支还是顺序执行的指令地址 pc+4，将其作为下一指令地址送往 InsMemory；InsMemory 得到来自 PC 的地址后在存储器中寻找该地址处的指令，将指令输出，送往下一个阶段。

- 译码阶段 ID

译码阶段的模块主要包含 RegFile、ConUnit 和 Ext，完成对指令的译码并产生对后续各模块的控制信号（ConUnit），根据指令中指定的寄存器取出其中的数据送往下一个阶段（RegFile），对指令中分散的立即数整合并扩展，送往下一个阶段（Ext）。

- 执行阶段 EX

执行阶段的模块主要包含 ALU、BranchCon，对输入的操作数完成算术逻辑运算（ALU）；同时在本阶段完成分支指令的判断（BranchCon）。

根据上一阶段送来的操作数（RegOut、Ext\_Imm）和译码控制信号（signal、Aluop）在 ALU 中进行算术逻辑运算，并产生运算结果 AluResult，将标志 ZF、CF、OF 置位。

BranchCon 模块根据 ALU 结果、标志位和译码阶段传来的控制信号（BranchType）对不同的 B 指令进行判断是否跳转；同时在其后产生分支地址；最后根据跳转信号选择下一指令地址是 branchAddr 还是 pcadd4。

- 访存阶段 MEM

访存阶段的模块主要包含 DataMemory，根据上一阶段产生的访存地址完成对数据存储器的读或写操作。

DataMemory 根据 EX 段传来的 AluResult 作为访存地址，在此地址处根据访存控制信号决定是写还是读，写则把传入的 RegOut 数据写入对应地址处，读则把对应地址处的数据读出，传到下一个阶段。

- 回写阶段 WB

回写阶段的模块主要包含 MemtoRegMux，根据上个阶段传来的 AluResult 和 MemDataOut 以及回写控制信号选择两者之一写入到寄存器中，此时寄存器应控制为允许写。

### 3.2.2 具体实现

单周期 CPU 在一个周期内将各个模块产生的信号与数据传往各处，那么多周期中，由于一条指令的工作需要在不同周期内完成，所以仅仅使用单周期的连线与设计是不行的，因为在一个周期产生的信号和数据并不能传递到后面需要用到其的周期中。

之前完成的单周期 CPU 已经实现了基本模块，为了使前面阶段产生的数据和信号能够保存并传递到后面周期的模块中，我们在基本模块（流水段）之间添加寄存器用于保存数据和信号。段间寄存器应有维持数据一个周期的功能，并能支持在段与段之间传递。在这里我们使用 D 触发器（锁存器）实现，在前一流水段产生数据后，送到触发器的 D 端口，锁存一个周期，在下一时钟上升沿到来时将锁存的信号和数据送往 Q 端口，从而送往下一流水段。大致示意图如下图。

值得注意的是，段间寄存器并不一定是传递上一段产生的数据或信号，也会传递之前几个段的数据或信号，也就是说，一个数据或信号可能传递多个流水段才能到达最终被使用的地方。

例如在译码段产生的写回寄存器的编号 rd，其要经过 ID\_EX、EX\_MEM、MEM\_WB 多个段间寄存器才能到达 WB 阶段被使用。



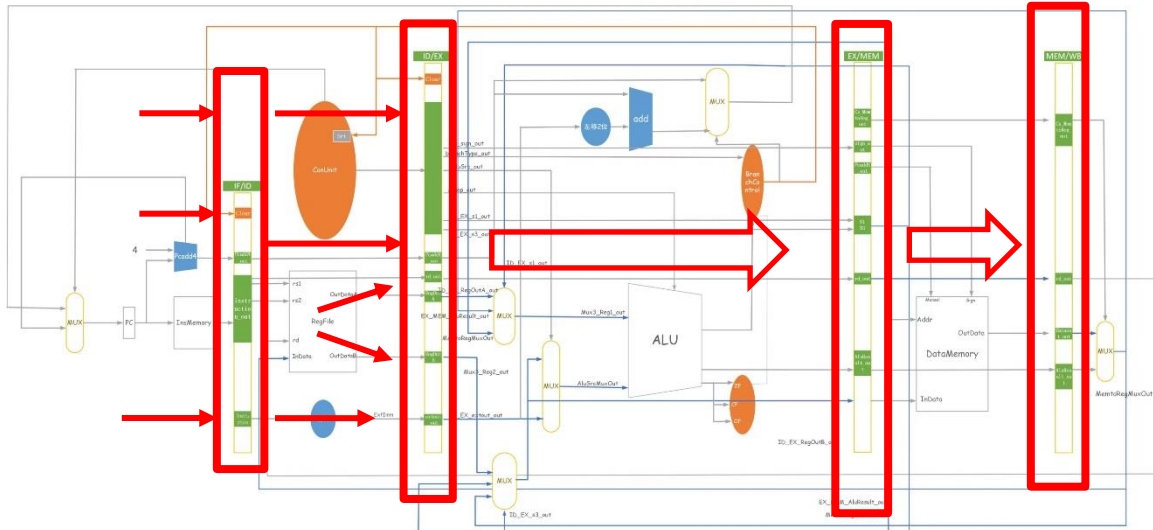


图 30 五级流水段信号传递

### 3.2.3 时序分析

本模块会根据具体的指令分析五段流水段的实际运行过程。

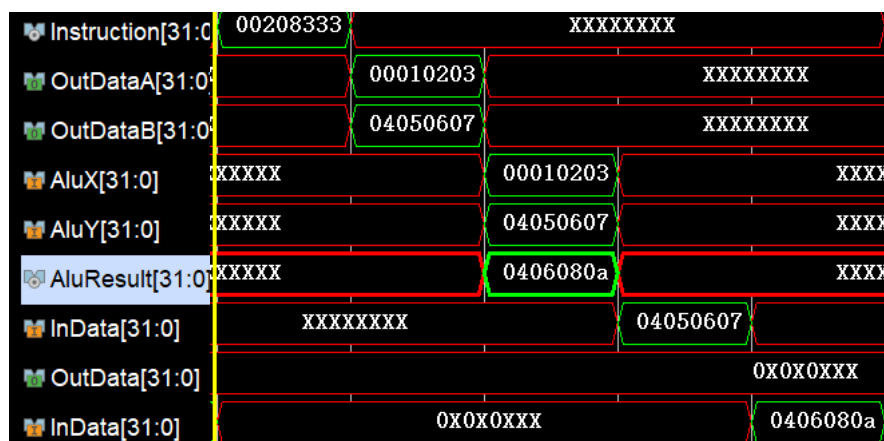


图 31 ADD 指令仿真时序图

多周期无流水 CPU 会分在五个时钟周期完成一条指令的处理，在仿真时序图上如上图所示。由于加入了能够保存传递数据的段间寄存器，指令的完成显得并不像单周期那样仓促，具体表现为在不同段的数据会在不同的周期内有效，形成向下阶梯式的时序图。

上图为 ADD 指令在五级流水段中的仿真时序图。左侧为数据名称，在 IF 阶段由 PC 产生的指令地址送入 InsMemory 中取出相应的指令 Instruction (00208333)，送到 ID 阶段；在 ID 阶段访问寄存器文件取出两个寄存器中的数据 OutDataA (00010203) 和 OutDataB (04050607)，送到 EX 阶段；在 EX 阶段，对送来的操作数执行加法操作，可以看到两个操作数分别为 AluX (00010203) 和 AluY

(04050607), 执行结果为 `AluResult` (0406080a), 送往 MEM 阶段; 在访存阶段, ADD 指令并不访存, 故继续向下传递数据, 送到 WB 阶段; 在 WB 阶段, 将 EX 阶段产生并通过 MEM 阶段传来的 `AluResult` 写回到寄存器中, 可以看到最后一行的 `Indata` 为 0406080a, 正好为刚才 ALU 计算结果。

五级功能段执行指令在仿真时序图上会呈现多个阶梯式的信号图, 依次相连, 只是每次会从取指 (表现为 `Instruction` 的数据) 开始。但模块间断工作的方式显然会浪费大量资源, 因为在执行完一条指令的分出来的操作后此功能段会闲置 4 个周期。

### 3.3 流水执行指令

将指令执行过程分为多个功能段显然只是过渡阶段, 那样会浪费大量的空闲的时钟周期, 不利于效率和吞吐量的提升。假使我们在 IF 阶段取指令后紧接着让它在下一个时钟周期取下一条指令, 那么将能实现所有功能段的不间断工作, 就像工作车间中真正的流水线那样。

我们已经实现了多周期无流水的 CPU, 是否可以改变一些部分, 使之变成流水线 CPU 呢?

多周期无流水 CPU 每条指令会运行五个时钟周期, 即使在 IF 阶段产生了顺序执行时下一条指令的地址 `PC+4`, 或在 EX 阶段产生了分支地址 `BranchAddr` 也只能向后传递, 等待五个周期全部执行结束后, 在 WB 阶段写寄存器的同时将下一指令地址 `PC+4` or `BranchAddr` 送往 PC 中, 从而在新的周期取下一指令。

倘若在 IF 取指后产生 `pc+4` 送往 PC, 并在下个周期直接用其取指, 那么或许就可以实现接连取指, 不停运行了。我们可以模拟一下。

IF 取得第一条指令, 并同时产生 `pc+4` 作为下个周期的取指地址, 维持过这个周期; 在下一周期开始时, `pc+4` 送入 IF 段取下一条指令, 而上次取得的指令送入到了 ID 段进行译码, 本周取得指令送入 IF\_ID 寄存器中, ID 段译码得到的结果送入 ID\_EX 寄存器中, 同时 IF 段产生 `pc+4` 用于下个周期取指, 维持过这个周期; 下一周期开始时, IF 段根据 `pc+4` 取新的指令送入 IF\_ID 寄存器中, 同时产生新的 `pc+4` 用于下个周期取指, 同时上个周期的指令从 IF\_ID 中送往 ID 段译码, 同时 ID\_EX 中的信号送往 EX 中.....

经过模拟我们发现并没有发生流水线中的数据冲突（这里的数据冲突指的是下个周期的数据把上个周期的数据覆盖掉，从而无法实现数据传递）。

但值得一提的是，以上的流水执行并没有考虑到分支指令，分支的不确定性将会是我们下一个要解决的问题。

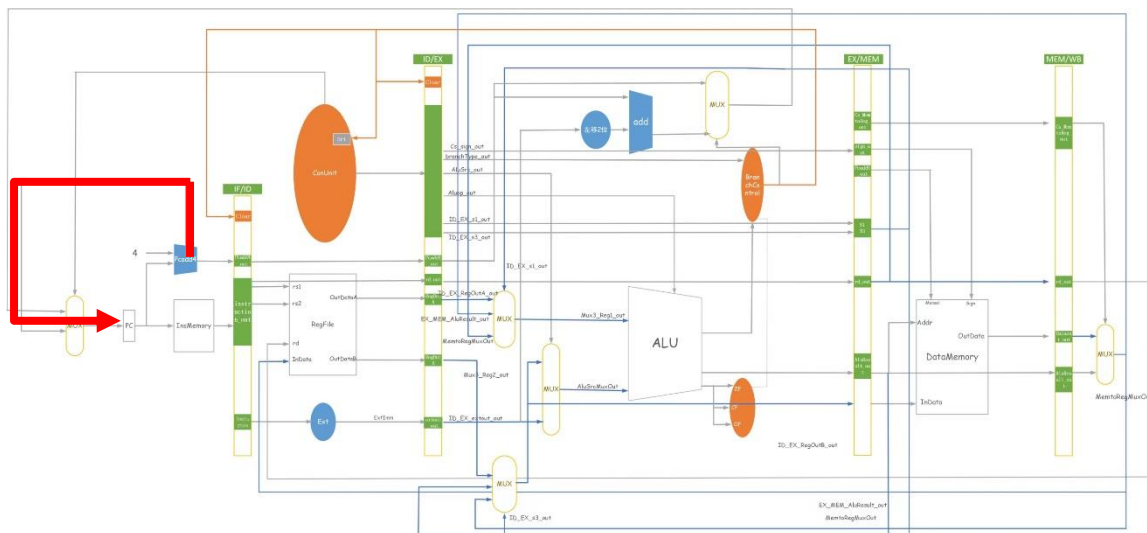


图 32 pcnext=PCadd4

### 3.3 分支预测

实现流水执行指令后我们首先要解决的问题是面对分支指令时的取指问题，因为实现流水处理指令的关键操作就是不停的在新的周期使用  $PC+4$  取指。但分支指令可能会打破这一常规，所以遇到分支指令时下一周期的取值地址是未知的。接下来将分两个部分详细介绍如何处理分支指令的。

#### 3.3.1 静态预测工作机制

之前的章节中已经完成了流水线工作，其在新的周期默认在  $PC+4$  处取得新的指令。但遇到分支指令时，倘若分支不跳转，则在  $PC+4$  处取得指令并没有错误；若分支成功跳转，则会则新的目标地址处取得下一条指令。

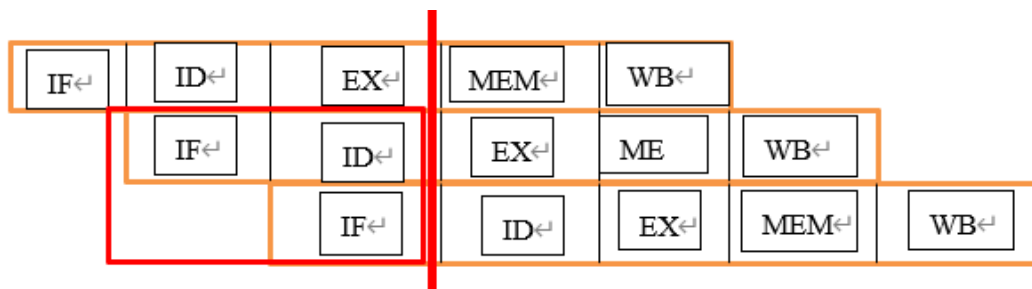


图 33 分支指令

### 3.3.2 静态预测思路

在本模块中将介绍流水线 CPU 中实现静态分支预测（not taken）的思路。

- 判断分支是否成功

判断分支是否成功（taken）显然是首要任务，它将决定下一步的方向。

根据本 CPU 中 B 指令含义，其需要判断两个寄存器是否满足一定条件，从而判断出分支是否成功。不加改造的五级流水段判断两个寄存器的值是否满足一定条件只有在 EX 阶段的 ALU 中计算所得，所以将分支判断放在 EX 段中完成。通过译码阶段读出来两个寄存器的数据送到 EX 段中的 ALU 中（就如同 R 类型指令那样），在 ALU 中完成两个操作数的算数逻辑运算，产生结果和标志位。根据结果和标志位判断条件是否满足，分支是否成功。

- 产生目标地址

得到分支结果之后应选择下个周期的取指地址，根据分支判断的地点选择在 EX 段产生目标地址。若分支失败选择 PC+4，分支成功应将分支指令中含有的地址作为下一目标地址。

- 分支成功后的工作

由于流水线工作机制（接连取指），我们判断出来出错时已经是执行阶段（EX）了，因为跳转指令（主要是 B 指令）需要 EX 段计算条件。而由于判断缓慢，在已经经过的两个指令周期中有两条指令已经进入到流水线中（如下图所示，其中一条执行完 ID 段，一条指令完 IF 段），我们需要将多取的两条指令在流水线中清除它们存在过的痕迹，因为它们并不是正确的下一条指令。

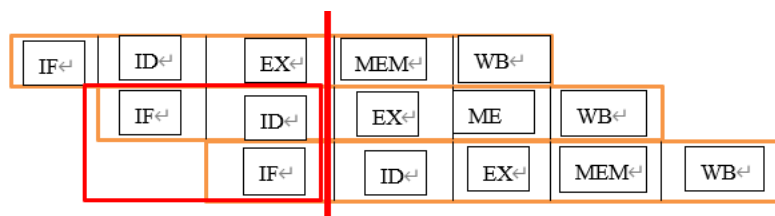


图 34 分支指令

当然清除两个错误指令会浪费两个周期，然后再第三个周期在目标地址处重新取指。

### 3.3.3 静态预测实现

本模块中将介绍静态预测的具体实现方法。

- 判断分支是否成功

判断分支的工作放在 EX 阶段进行，在 EX 段之前的所有操作与 R 类型指令类似，这里不在介绍。在 EX 段中 ALU 计算出结果后，将 `AluResult` 和标志送往分支判断模块 `BranchCon`。`BranchCon` 模块根据译码段产生的 B 指令类型选择不同的判断条件，对 ALU 结果和标志判断，若判断分支成功则产生分支信号 `BranchE`。以上过程所用到的信号和功能可查看模块介绍中的 `BranchCon` 和 ALU 模块。

- 产生目标地址

默认情况下 CPU 取指选择的是 `PC+4`，但此时并不知道此时是否要分支，所以将下一地址产生放在了 EX 阶段，在 EX 阶段选择 `PC+4` 或是 `BranchAddr`。所以需要将 IF 段产生的 `PC+4` 向后面的流水段传递，传递方法已经在五级流水段模块具体介绍过了。EX 段拿到 `PC+4` 后，同时也应拿到了分支地址 `BranchAddr`（通过变换译码段的 `Ext` 中产生的立即数），将两者送入选择器并根据分支信号进行选择，若分支成功，根据产生的 `branchE` 选择 `BranchAddr` 作为下一指令地址地址送往 PC，反之选择 `PC+4`。

```
Mux2_32bit BranchMux(
    .a(BranchMuxA),.b(ID_EX_pcadd4_out),.sel(branchE),.out(BranchMuxOut)
);
```

- 清除错误的流水段

根据我们流水线结构，在各流水段间设置了寄存器，所以现在一条指令结束 ID 段应该把数据放到了 `ID_EX` 寄存器中，另一条放到 `IF_ID` 寄存器中，两条指令所有的痕迹就是这两个寄存器的内容，所以我们要清空这两个寄存器。如下图所示。

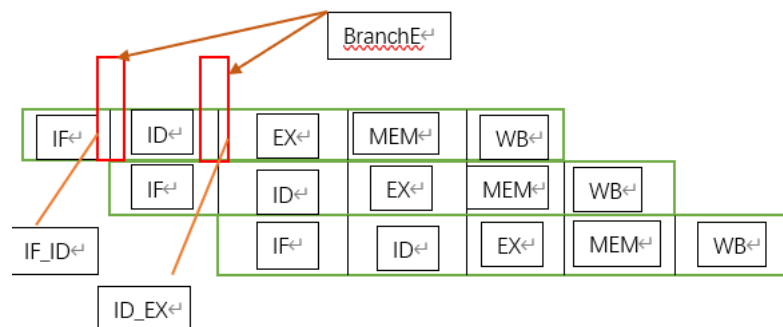


图 35 分支预测错误处理

下面介绍如何实现。

在各 D 触发器中设置输入信号 **clear** 表示在下一个周期将输出端 Q 置为 0，相应的在段间寄存器中设置输入信号 **clear** 并将其传入各 D 触发器中（段间寄存器本质是由各个数据信号的 D 触发器实例化组成的）。当分支成功判断出来后，会在 **BranchCon** 发出控制信号 **BranchE**，将其连到两个段间寄存器 **IF\_ID** 和 **ID\_EX** 的 **clear** 端口，从而在 **branchE** 信号有效时清空两个寄存器。

值得一提的是，仅仅置 0 两个段间寄存器并不能完成分支成功后的工作，因为置零寄存器中的数据依旧会向下一个流水段传递数据 0，从而导致错误，这部分内容会在问题解决模块详细介绍。

### 3.5 数据冒险检测与解决

完成流水线的分支控制已经能够处理简单的程序，但由于五级流水段的工作特性，指令在执行期间会产生难以避免的错误。在指令的译码阶段，由于前几条指令并没有完成执行，若其需要在 **WB** 阶段将结果写入寄存器，此时并没有写入。这就导致后面的指令在读取相应寄存器时会“提前”读取，并不能读取应得的值，这就是数据冒险。下面详细介绍数据冒险的处理方法。

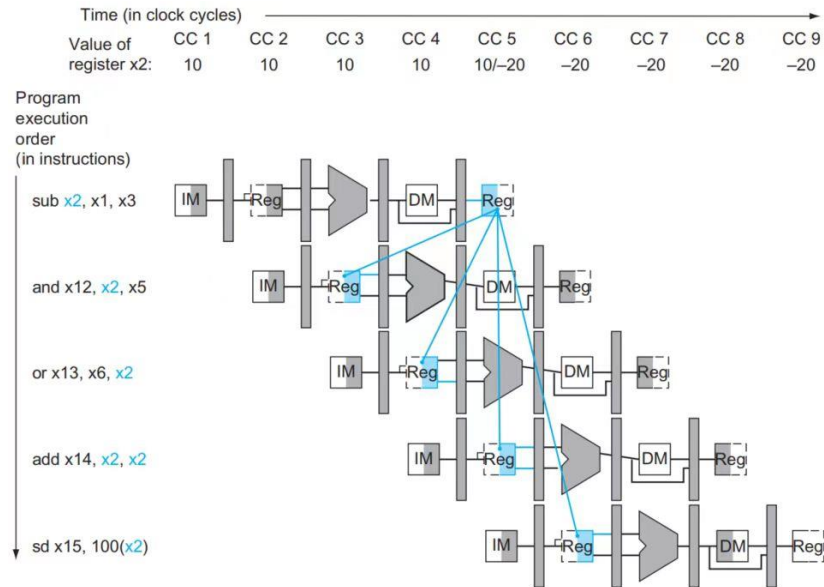


图 36 数据冒险

### 3.5.1 数据冒险检测

数据冒险发生在后一条指令要读取前一条指令要写入的寄存器时，故数据冒险的检测工作应放在后一条指令的读寄存器操作时，也就是译码阶段。我们可以在逻辑上设置一个数据冒险检测处理模块，但本次设计在实际上将其放在了 ConUnit 中。下面分两个部分介绍数据冒险检测工作，分别对应原理和实现。

#### ● EX 冒险检测

此时正在译码指令需要读取它的前面一条指令要写入的寄存器，而它前面一条指令目前在流水线中正在运行 EX 阶段，而这条指令所需的所有数据都来自 ID\_EX 寄存器，这是由上个周期译码这条指令得到的，所以要得到这条在 EX 段的指令要写入的寄存器可以在 ID\_EX 寄存器中获取。我们应在译码阶段判断当前指令要读取的寄存器和处在 EX 段指令的写入的寄存器是否相等。

另外，并不是每一条指令都要吸入寄存器，所以应判断处在 EX 段指令将来是否写回寄存器。

RISC-V 架构处理器中存在一个特殊的寄存器 x0，它的值恒为 0，如果写回的寄存器是 x0，寄存器的值仍是 0。所以冒险检测条件应排除目标寄存器为 0 的指令。

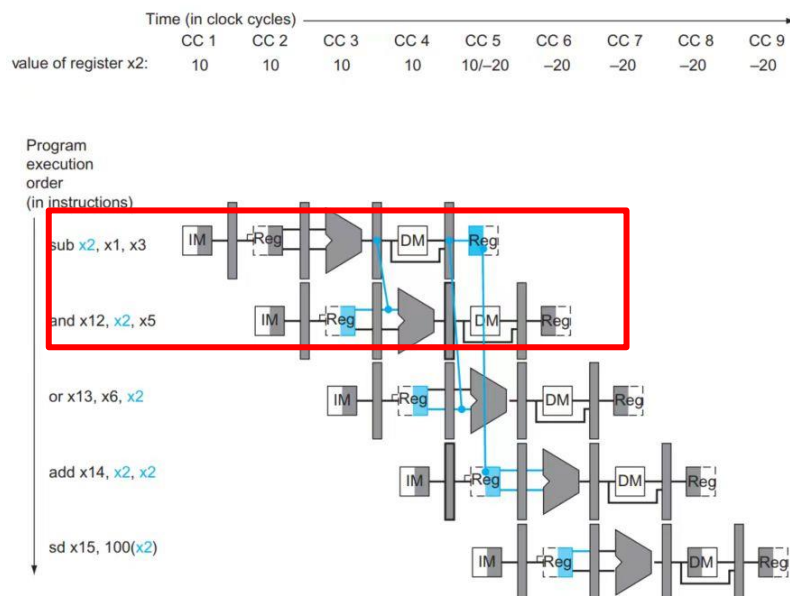


图 37 数据前推 EX 冒险

- MEM 冒险检测

此时正在译码指令需要读取它的前面第二条指令要写入的寄存器，而它前面第二条指令目前在流水线中正在运行 MEM 阶段，而这条指令所有数据都来自 EX\_MEM 寄存器，这时又上个周期执行这条指令得到的，所以要得到这条在 MEM 段的指令要写入的寄存器可以在 EX\_MEM 寄存器中获取。同样的，我们应在译码阶段判断当前指令要读取的寄存器和处在 EX 段指令的写入的寄存器是否相等。

同样的，应判断这条指令是否要写回寄存器和写回的寄存器是否为 0 号寄存器。



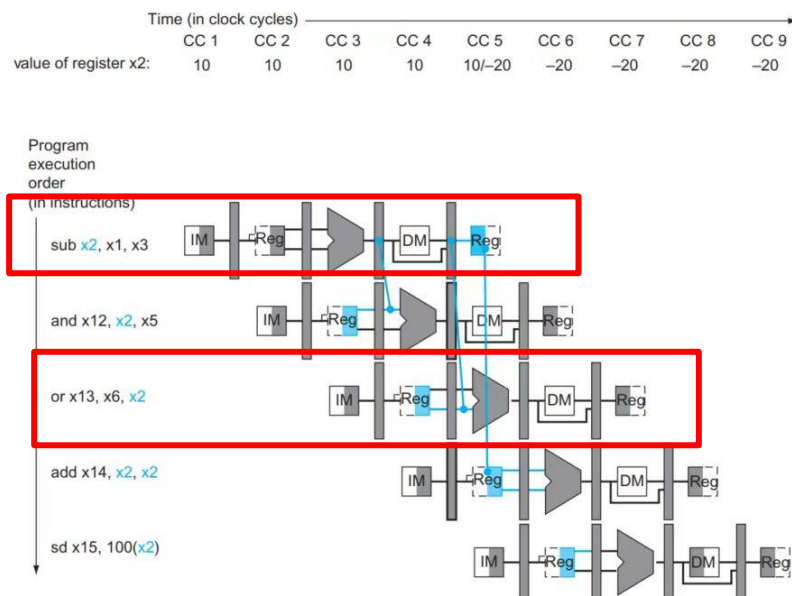


图 38 数据前推 MEM 冒险

- 冒险检测的实现

EX 冒险的检测须判断 ID\_EX 寄存器中写入的寄存器是否和译码中指令要写入的寄存器相等，而 MEM 冒险的检测判断 EX\_MEM 寄存器中写入的寄存器是否和译码中指令要写入的寄存器相等；同时处于后面流水段的指令应在将来写入寄存器；同时写入的寄存器不会为 0 号寄存器。如下所示。

```
if(Instruction[19:15]==EX_rd&&RegWrite_EX==1&&EX_rd!=0)//检测 EX 冒
```

```
险
```

```
//处理 EX 冒险
```

```
else
```

```
if(Instruction[19:15]==MEM_rd&&RegWrite_MEM==1&&MEM_rd!=0)//检
```

```
测 MEM
```

```
//处理 MEM 冒险
```

Instruction[19:15]表示当前译码指令要读取的寄存器，EX\_rd 和 MEM\_rd 分别表示处在 EX 段和 MEM 段指令要写入的寄存器，需判断它们是否相等；RegWrite\_EX 和 RegWrite\_MEM 分别表示处在 EX 段和 MEM 段指令是否要写寄存器；EX\_rd 和 MEM\_rd 则表示不会写入 0 号寄存器。

值得一提的是，数据冒险存在优先级，EX 冒险的优先级要高于 MEM 冒险，因为引起 EX 冒险的指令在流水上要晚于引起 MEM 冒险的指令，所以最终结果一定是 EX 冒险指令写入的结果覆盖掉 MEM 冒险指令写入的结果。

在冲突检测上表现为先判断 EX 冒险，再“else if”MEM 冒险。

当然上述只是一个寄存器的检测逻辑，一条指令可能同时读取两个寄存器且两个寄存器可能都存在冒险。

### 3.5.2 数据前推解决冒险

数据冒险产生是因为提前读取寄存器而读到错误的值，当然可以避免提前读取，从而将其“延后”，即暂停流水线直到正确的值被写入到了寄存器中，但很显然这不是明智的做法。产生冒险的原因并不是数据没有产生，而是数据没有到位，所以可以提前获得未到位的数据以解决冲突，这就是数据前推。接下来介绍本次设计中数据前推解决冒险的方法。

- 前推原理

在译码的指令要读取寄存器时，前面的指令可能会写入但还未写入，要写入的值一定是 ALU 的计算结果（这里不考虑 LOAD 指令），在本条指令使用寄存器中的值在 EX 段计算时（读取的时错误的的数据），即使是离本条指令最近的上一条指令已经结束 EX 段并产生了结果，这个结果就是本条指令应该使用的正确的值（EX 冒险）。所以说前面的指令计算出要写回的结果后只是在流水线中将其无意义的向后传递。我们可以提前拿到并使用它。对于 EX 冒险，译码指令所需的是此时处于 EX 段中计算所得的结果。我们可以先读取错误的寄存器数据，而在下一阶段使用数据时拿到上一条指令在 EX 段计算的结果，这个是正确的数据。本条指令使用数据计算时其在 EX\_MEM 寄存器中。

同理，对于 MEM 冒险，译码指令所需的是此时处于 MEM 段中的结果。我们可以先读取错误的寄存器数据，而在下一阶段使用数据时拿到上面第二条指令在 MEM 段的结果，这个是正确的数据。本条指令使用数据计算时其在 MEM\_WB 寄存器中。

需要将两个数据前推到 EX 段的 ALU 前使用。

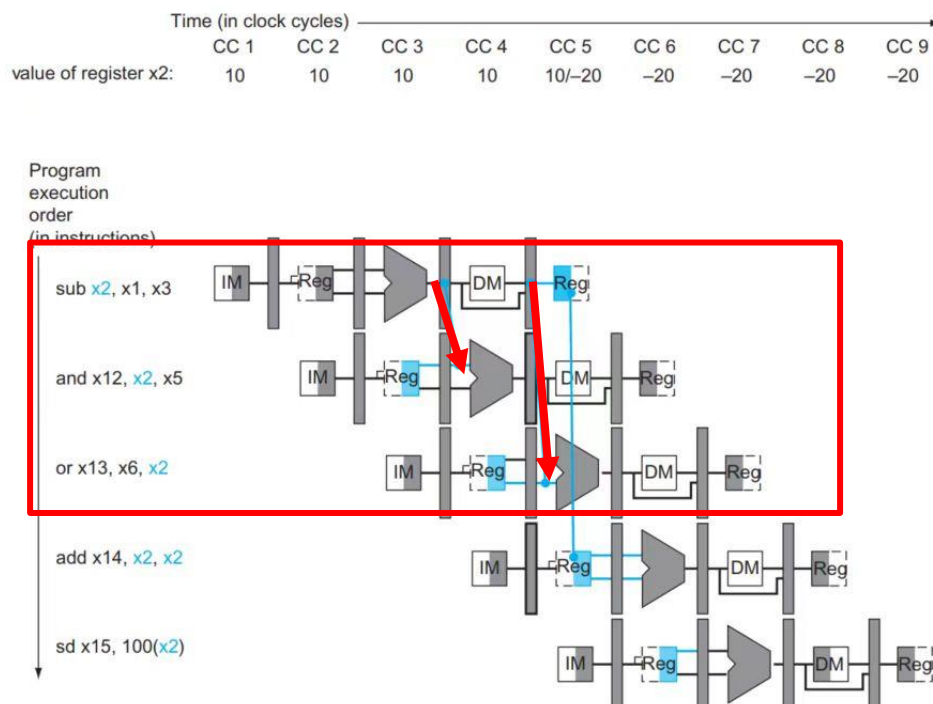


图 39 数据前推

### ● 前推实现

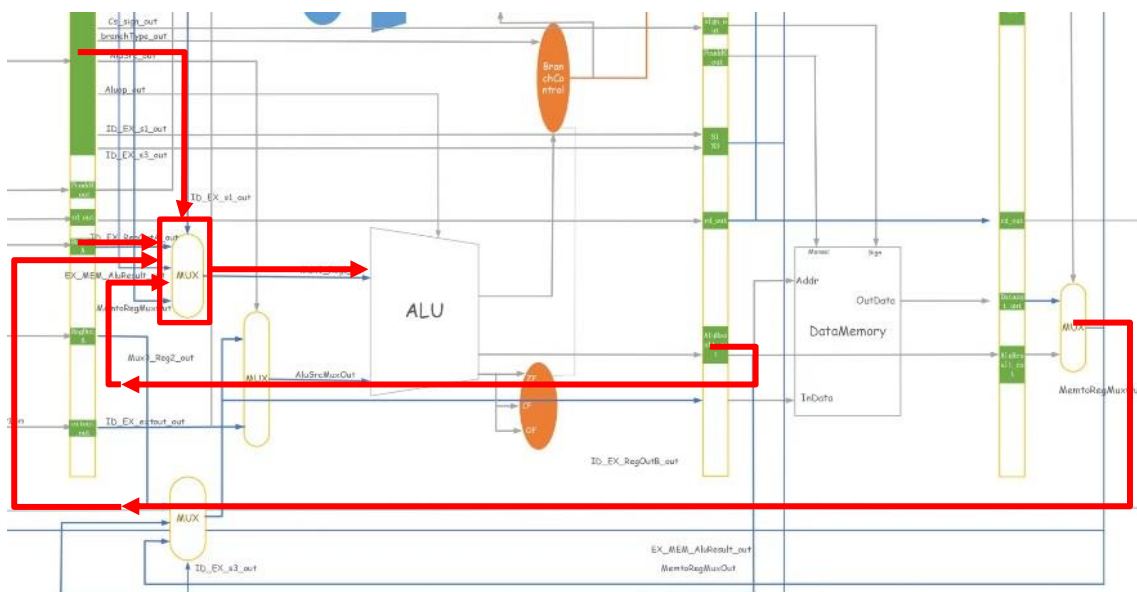


图 40 数据前推数据通路

对于 EX 和 MEM 冒险，在指令译码时需要先取出寄存器中错误的数据，因为此时处于 EX 和 MEM 阶段的指令并没有执行结束，也就无法拿到正确的数据。到本指令 EX 执行时，上个周期在 EX 和 MEM 段处理的数据已经到了 EX\_MEM 和 MEM\_WB 寄存器中，所以在 EX 执行前需要取出这两处的数据（具体哪一处根据冒险类型）。所以应在 ALU 输入端之前添加多

路选择器用来选择前推的数据或是没有冒险的数据。如下所示。

```
Mux3_32bit Mux3_Reg1(
    .sel(ID_EX_s1_out),.a(EX_MEM_AlarResult_out),.b(MemtoRegMuxOut),.c(
    ID_EX_RegOutA_out),.out(Mux3_Reg1_out)
);
```

表格 27 Mux3\_Reg1 控制选择表

Sel (ID_EX_s1_out)	Out (Mux3_Reg1_out)	描述
00 (a)	EX_MEM_AlarResult_out	EX 冒险选择 EX_MEM 中结果
01 (b)	MemtoRegMuxOut	MEM 冒险选择 MEM_WB 结果
10 (c)	ID_EX_RegOutA_out	无冒险选择 ID 寄存器中结果

多路选择器的三个输入分别为发生两种冒险时需要的数据和正常执行时寄存器中的数据。选择信号为检测冒险时在 ConUnit 中产生的冒险信号，定义如下。

```
if(Instruction[19:15]==EX_rd&&RegWrite_EX==1&&EX_rd!=0)
    s1<=2'b00; //处理 EX 冒险
else
    if(Instruction[19:15]==MEM_rd&&RegWrite_MEM==1&&MEM_rd!=0)
        s1<=2'b01; //处理 MEM 冒险
    else s1<=2'b10; //默认没有冲突
```

当然上述描述以寄存器 1 为例，实际过程中两个寄存器都有可能发生冒险。

### 3.5.3 LOAD 冒险

MEM 冒险和 EX 冒险上面的指令 EX 段已经在 ALU 中计算的得出，后续的周期只是在流水线中无意义的传递，这两种冒险可以对应大多数指令的数据冒险情况。而 I 类型的 LOAD 指令需要在 MEM 段才能产生下面指令所需要的数据，获得

则要再推迟一个周期，它产生的冒险并不能单单通过数据前推来解决，因为即使使用前推 LOAD 的数据也要在下个周期产生。

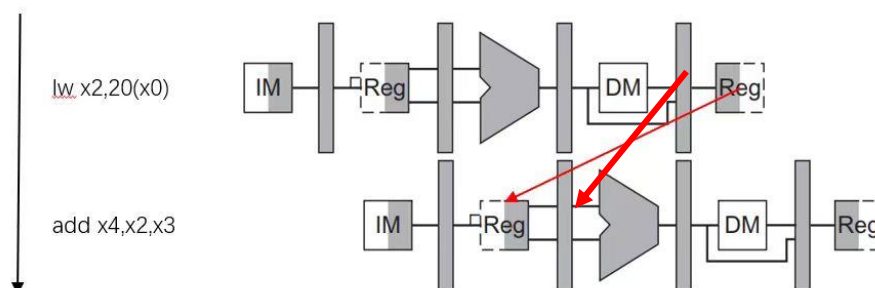


图 41 LOAD 冒险

下面分两个方面来介绍设计中是如何处理 LOAD 类型的冒险的。

### ● 处理方案

由于本条指令在 ID 阶段发现 LOAD 冒险时通过前推处理还提前了一个周期（指的是还需要再等一个周期 LOAD 指令才能产生数据），所以可以将正在译码的指令延迟（等待）一个周期。暂停一个周期后，前推逻辑就可以处理这个冒险了。

当指令在译码阶段时，ConUnit 检测是否发生 LOAD 冒险，若检测到 LOAD 冒险，在流水线中插入一个暂停周期，暂停之后重新处理本条指令，这时会满足 MEM 前推的条件，如下图所示，之后可以根据 MEM 冒险处理。

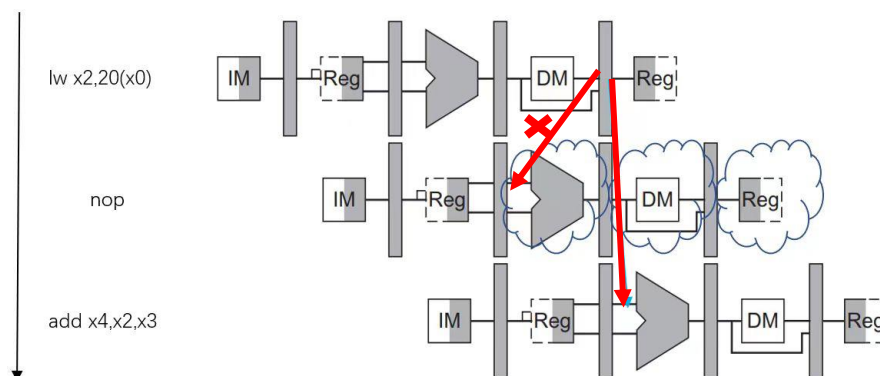


图 42 LOAD 冒险周期暂停

检测 LOAD 冒险的条件。正在译码的指令发现即将要读取的寄存器与上一条 LOAD 指令要写入的寄存器相同，且写入的寄存器不为 0 号寄存器。检测上一条指令是 LOAD 指令的条件为读存储器信号为有效，因为 RISC-V 指令架构中只有 LOAD 类型指令才会读取数据存储器。

值得一提的是，LOAD 指令只有紧邻本条指令才会暂停流水，若 LOAD 指令是上面第二条指令则使用 MEM 冒险策略可以解决。所以检测 LOAD 冒险发生时，只需要检测上一条指令的信息即可，而上一条指令在本指令译码时正在 EX 段但未完成，所以其所有数据应来自 ID\_EX 段间寄存器，我们需要在 ID\_EX 中获取信息并进行判断。

而流水线暂停简单来说就是将 IF、ID 阶段暂停一个周期、EX 及后面阶段延后一个周期。对于 IF 和 ID 阶段，将 PC 程序计数器与 IF/ID“暂停”，禁止 PC 寄存器和 IF/ID 流水线寄存器的改变以阻止这两条指令的执行。如果这些寄存器被保护，在 IF 阶段的指令就会继续使用相同的 PC 值取指令，同时在 ID 阶段的寄存器就会继续使用 IF/ID 流水线寄存器中相同的字段读寄存器，以达到流水线“暂停”的效果。对于 EX 阶段开始的流水线后半部分，执行没有任何效果的指令。

#### ● 具体实现

LOAD 冒险的检测。

据之前的分析，LOAD 指令的数据此时已到 IF\_ID 寄存器中，所以在 IF\_ID 寄存器中取出 LOAD 指令要写入的寄存器编号，将其与译码指令要读取的寄存器编号比较。同时我们应判断上一条指令为 LOAD 指令，将 IF\_ID 中 LOAD 指令特有的信号 MemRead 取出判断。代码实现如下。

```
if(Instruction[19:15]==EX_rd&&EX_MemRead==1)begin
```

```
//load 冒险处理 end
```

```
else begin
```

```
//正常处理 end
```

这里的 EX\_rd 为正处于 EX 段执行的 LOAD 指令要写入的寄存器，EX\_MemRead 为 LOAD 指令的读存储器信号。当然这只是其中一个寄存器的实现，实际中两个寄存器都有可能产生 LOAD 冒险。

LOAD 冒险的解决。

应该将 PC 和 IF\_ID 暂停一个周期，可以根据冒险信号，让其控制 IF\_ID 和 PC 在下一个周期暂停，即 nextPC=prePC、IF\_ID\_out=preIF\_ID\_data。具体代码实现如下。

```
if(Instruction[19:15]==EX_rd&&EX_MemRead==1)begin
```

```
    lw_stall<=1; end//产生 LOAD 冒险信号让其控制 PC 和 IF_ID 寄存器
```

```
else begin
```

```
    lw_stall<=0; end//正常处理
```

在 PC 中收到 stall 信号让其在下个周期的值等于上个周期即可，在 IF\_ID 中收到 stall 信号让其传入各个触发器（段间寄存器本质是各个数据锁存器的集合），让触发器下个周期输出端输出上个周期的值。

## 4 问题解决与改进

在本单元中将介绍设计过程中出现的主要问题，以及它们是怎么被解决的。

### 4.1 问题解决

注：由于设计过程中出现的问题较多，且大多并未被记录，故本单元只会介绍部分。

1. 在 run implementation 时报 Conuint 中 lw\_stall 的 multi-driven nets 错误  
**解决：**将 rs1、rs2 的 lw 数据冲突检测放到一个 always 块中。
2. 起初 CPU 并未执行到 EX 周期，此时 branchE 信号为高阻状态，但 PC 的产生需要 branchE 这个选择信号。  
**解决：**起初并没有跳转指令，将 branchE 信号初始化为选择 PC+4。
3. 直接把 ID\_EX\_regoutB 送到了 EX\_MEM，导致 Store 指令无法处理冲突。  
**解决：**应该把 ID\_EX\_RegOutB 先送到 Mux\_Reg2 和前推的数据选择一下，再分别送到 AlusrcMux 和 EX\_MEM 中。
4. 原有的数据通路无法处理 slli、srli、srai 指令，原因是其与其他 I 类型指令的数据通路不同，这三条会使用立即数的部分作为操作数。  
**解决：**添加新的数据通路，完善 Ext 模块的功能，遇到这三条指令时特殊处理产生不同于其他 I 类型指令的立即数。
5. 在处理分支指令时，将 IF\_ID 寄存器清空之后，因为选择的是将寄存器的值清“零”，所以之后的周期不是不向后传数据，而是传 0，这就导致了下一周期的 ConUnit 误以为 0 就是正确的数据，所以会将 0x00000000 作为指令



进行译码。不知是否此原因，在下个周期会将 MemRead 信号错误置 1，导致错误满足 LOAD 数据前推的条件，从而得到错误的数。

**解决：**在 ConUnit 中，清空寄存器之后的一个周期，将 ConUnit 停机，主要将产生错误的 MemRead 信号置 0，设计中使用 BranchE 信号作为 ConUnit 的 rst 信号。

```
always@(negedge rst)begin
```

```
    if(!rst)begin //lw 处理，避免错误导致 MemRead=1
```

```
        MemRead<=0;
```

```
    end
```

```
end
```

- LOAD 指令暂停 PC 一个周期时，使用 stall 信号传入 PC，传入 PC 时已经是下一个取指周期，在这个取指周期应该取上个周期相同的 PC，检测到暂停信号后 PC 并没有保存上个周期的 PC 值，无法延续上个周期的 PC。

**解决：**在 PC 中加入寄存器保存上个和上上个周期的 PC 值，只保存上个周期是不行的，因为新的周期到来时（也是检测到停顿信号的周期），PC 中保存上个周期 PC 的寄存器值 hold\_PC 已经改变（在时钟上升沿已经改变，并没有来的及获取），所以增加一个寄存器 pre\_PC 保存上上周期的 PC，上升沿到来时（与 pc\_hold 信号同时上升沿），hold\_PC 的值变为 cur\_PC，而 pre\_PC 的值正好变为 hold\_PC，为应获得的 PC 值（说上个周期 PC 值不太准确，因为保存上个周期 PC 的寄存器已经更新）。让 cur\_PC=pre\_PC 即可得到正确的 PC。

- 第一次产生 PC 时由于 PC+4 还未存在，而又不存在分支指令，故第一次无法通过 InPC 来获得 PC。仿真表现为信号时序图上 PC 为高阻。

**解决：**在 PC 中实现功能：当 PC 为第一次产生值时，直接输出 PC(初始化的 PC)+4。

```
else if(InsAddr==-4)begin
```

```
    cur_pc<=InsAddr+4 //InsAddr 为输出的 PC，初始化为-4，为的是产
```

```
生第一条指令时为 0。这里产生第一条指令直接加四
```

```
end
```



## 4.2 设计改进

基于设计验收的问题，对现有设计分析以下问题。

### 1. 动态分支预测实现。

可简单实现基于两位饱和计数器的动态分支预测。

运行过程：为每条分支指令配置一个分支历史条目，保存在一张分支历史寄存表中。分支历史条目包含指令地址、一个两位饱和计数器用于对分支进行判断和目标地址用于跳转。动态分支预测模块放在 IF 阶段，在取得指令之前在分支历史寄存表中根据 PC 寻址，若找到对应的分支条目，根据两位饱和计数器的值预测是否跳转，若跳转则根据目标地址取指。取指之后的动作与静态分支预测的后续相同。

条目更新：若之后判断出预测正确，则首先不需要更改流水线中的数据；第二需要更新对应条目的两位饱和计数器。若判断预测错误，则需要更改流水线中的数据（与静态预测相同）；第二需要更新对应条目的两位饱和计数器。

若发现一条表中不存在的分支指令，则需要将其加入预测表作为新的一项。若表不满则直接加入；若表满，则需要根据某一替换算法选择一项替换掉。

### 2. 对于无分支的 for 循环，分析静态分支预测和两位饱和计数器的分支预测。

无分支的 for 循环特征为在循环体中无其他分支指令，意味着仅在循环结尾处有一条固定向后跳转的分支指令。

静态分支预测：静态预测不跳转意味着在 for 循环结束总是会多取两条无用指令，之后在判断出跳转，每次会浪费 3 个时钟周期（2 个多余指令和一个重新取指），若循环次数为  $N$ ，则浪费的时钟周期数为  $3(N-1)$ ，在最后一次循环结束时预测正确。若预测总是跳转则会在循环时 100 percent 正确，仅会在最后一次循环结束时预测错误从而浪费 3 个周期。

两位饱和计数器预测：两位饱和计数器特征为每次需要经过两次确认才能更改预测的结果，但从初始状态到学习完成需要一定的出错尝试。若初始为“11”或“10”状态在 for 循环进行时每次均预测正确，在循环结束时也预测正确（此时状态为“11”->“10”）；若初始为“01”，则第一次预测错误，浪费

3 个周期，其余均正确；若初始为“00”，则前两次预测错误，浪费 6 个周期，其余均正确。由此可见基于两位饱和计数器的分支预测在无分支的 for 循环的预测成功率上，与初始值有密切关系。

注：以上循环默认次数大于 2。

综上所述，初始为“11”、“10”状态的两位饱和计数器预测正确率可达 100 percent，静态预测不跳转对于 for 循环预测仅最后一次预测正确。同时应注意到，两位计数器的预测方法需要较多硬件的开销，每一条指令均需要一个两位计数器及地址信息。

## 5 结果分析

本单元中将介绍测试使用的代码数据、测试的结果以及主要功能的测试与分析。

### 5.1 测试数据与代码

- 起始数据设置

为便于测试，起初将数据存储器中各单元数据设为单元编号。

将 32 个通用寄存器初始数据设为 0x00000000。

- 基本指令测试

Test2.txt 中含有 15 条指令集中包含的基本指令，包括 R、I、LOAD、S、B 型指令。

指令如下

```
//lw r1          r1=00010203
//lw r2          r2=04050607
//lw r3          r3=08090a0b
//lb r4          r4=0000000d
//lb r5          r5=00000003
//add r6,r5,r1    与 lb r5 发生数据冲突 r6=r1+r5=00010203+00000003=00010206
//slt r2,r7,r1    r1=00010203 r7=00000000 r7<r1 r2=00000001
//addi r7,r6 ,imm  r6=00010206 imm=00000000
r7=r6+imm=00010206+00000000=00010206
```

```

//sw r3 [r0+imm]    r3=08090a0b
//add r8 r6,r7      与 addi r7,r6,imm 发生冲突 r6=000102006 r7=00010206
r8=r6+r7=0002040c
//add r9 r8,r1      与 add r8 r6,r7 发生冲突 r1=00010203 r8=0002040c
r9=0003060f
//add r10 r9,r8     与 addi r7,r6,imm & add r8 r6,r7 冲突 r8=0002040c r9=0003060f
r10=00050a1b
//beq r1 r2  ->insMem[0]    taken
//lb r4  r4=0000000d        //无效指令， as bne is taken
//lb r5  r5=00000003        //无效指令， as bne is taken

```

```

10000011 00100000 00000000 00000000
00000011 00100001 01000000 00000000
10000011 00100001 10000000 00000000
00000011 00000010 10100000 00000000
10000011 00000010 00000000 00000000
00110011 10000011 01010000 00000000
00110011 10100001 00010011 00000000
10010011 00000011 00000011 00000000
00100011 00100000 00110000 00000000
00110011 10000100 01100011 00000000
10110011 10000100 10000000 00000000
00110011 00000101 10010100 00000000
01100011 10010000 00100000 00000000
00000011 00000010 10100000 00000000
10000011 00000010 00000000 00000000

```

图 43 基本指令测试

### ● 程序测试

Test1.txt 中含有 18 条指令集中包含的基本指令，包括 R、I、U、S、B 型指令。完成简单的向内存中循环存数功能。功能如下。

```

#80000000 取反->MEM[0]
#08000000 取反->MEM[4]
#00800000 取反->MEM[8]

```

#00080000 取反->MEM[12]

#00008000 取反->MEM[16]

#00000800 取反->MEM[20]

#00000080 取反->MEM[24]

#00000008 取反->MEM[28]

#00000000 取反->MEM[32]

#不断循环

指令如下。

0#0            ADDI x8,x0,0;        #清空 x8

4#4        start:    ADDI x6,x0,0;        #清空 x6

8#8            ADDI x5,x0,0;        #清空 x5

C#12            ADDI x7, x0,3; #循环次数

10#16        LUI x10,0x00080;

14#20    loop:ADDI x6,x6,1;

18#24        BNE x6,x7,loop;

1C#28        SRLI x10, x10, 4;

20#32        ADDI x6,x0,0;        #清空 x6

24#34        ANDI x18,x10,all one;

28#40        SW x18,0(x5);

2C#44        ADDI x5,x5,4;

30#48        BNE x10, x0, loop;

34#52        ADDI x8,x0,0;        #nop

38#56        ADDI x8,x0,0;        #nop

3C#60        BEQ x0, x8, start

40#64        ADDI x8,x0,0;        #nop

44#68        ADDI x8,x0,0;        #nop

```

10001001 00000100 00000000 00000000
00010011 00000011 00000000 00000000
10010011 00000010 00000000 00000000
10010011 00000011 00110000 00000000
00110111 00000101 00001000 00000000
00010011 00000011 00010011 00000000
01100011 10011010 01100011 00000000
00010011 01010101 01000101 00000000
00010011 00000011 00000000 00000000
00010011 01001001 11110101 11111111
00100011 10100000 00100010 00000001
10010011 10000010 01000010 00000000
01100011 00011010 10100000 00000000
00010011 00000100 00000000 00000000
00010011 00000100 00000000 00000000
01100011 00000010 00000100 00000000
00010011 00000100 00000000 00000000
00010011 00000100 00000000 00000000

```

图 44 程序测试

## 5.2 仿真结果

### ● 指令测试

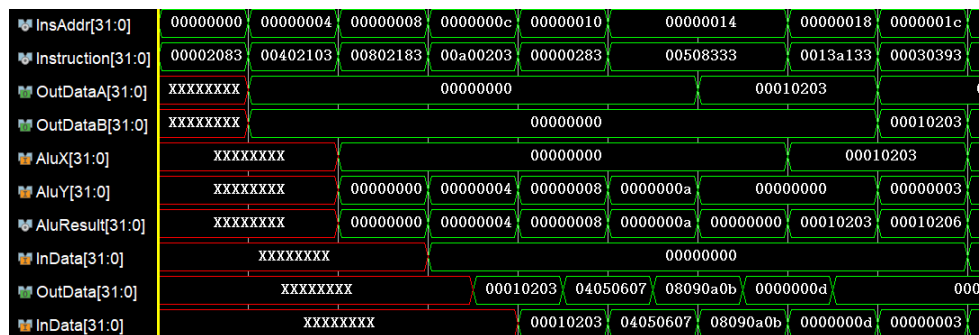


图 45 指令测试仿真结果

左侧为信号名称，自上到下对应流水的五个阶段，自上到下依次为指令地址、指令、两个寄存器读出数据、ALU 两个操作数和结果、存储器的输入和输出数据、寄存器的写回数据。右侧为仿真结果。

```
//lw r1          r1=00010203
```

```
//lw r2          r2=04050607
```

```
//lw r3          r3=08090a0b
```

```
//lb r4          r4=0000000d
```

```
//lb r5          r5=00000003
```

起初是五条 LOAD 指令，包含 LOAD WORD 和 LOAD BYTE 指令。对 x1 到 x5 五个寄存器赋值，作为测试的基础。

InsAddr[31:0]	00000004	00000008	0000000c	00000010	00000014	00000018	0000001c	00000020
Instruction[31:0]	00000000	00000004	00000008	0000000c	00000010	00000014	00000018	0000001c
OutDataA[31:0]	00002083	00402103	00802183	00a00203	00000283	00508333	0013a133	00030393
OutDataB[31:0]	XXXXXXXX			00000000			00010203	
AluX[31:0]	XXXXXXXX			00000000			00010203	
AluY[31:0]	XXXXXXXX	00000000	00000004	00000008	0000000a	00000000	00000003	
AluResult[31:0]	XXXXXXXX	00000000	00000004	00000008	0000000a	00000000	00010203	00010206
InData[31:0]	XXXXXXXX				00000000			
OutData[31:0]	XXXXXXXX			00010203	04050607	08090a0b	0000000d	0
InData[31:0]	XXXXXXXX			00010203	04050607	08090a0b	0000000d	00000003

图 46 LOAD 指令仿真

在仿真图上可以看到，取指阶段根据指令地址依次取出五条 LOAD 指令；在 AluResult 行根据 LOAD 指令的设计和 data 通路可知，EX 段的 ALU 中计算出的是访问数据存储器的地址，为测试方便分别设置为 0、4、8、a、0；在倒数第二行的 MEM 阶段在数据存储器中取出五个数据分别为 00010203、04050607、08090a0b、0000000d；在 WB 阶段的寄存器 Indata 中连续向寄存器中写入五个取出的数据。

//add r6,r5,r1          与 lb r5 发生数据冒险

//slt r2,r7,r1          r1=00010203 r7=00000000 r7<r1 r2=00000001

之后是两条 R 类型的数据，分别为 ADD 和 SLT。ADD 指令将 x1 和 x5 的值相加存到 x6 中，由于 ADD 指令需要读取 x5 的数据，故与上面一条 LOAD BYTE x5 发生 LOAD 型数据冒险。测试结果如下。

InsAddr[31:0]	00000014	00000018	0000001c	00000020	00000024
Instruction[31:0]	00508333	0013a133	00030393	00302023	00638433
OutDataA[31:0]	00000000	00010203			00000000
OutDataB[31:0]		00000000	00010203	00000000	08090a0b
AluX[31:0]		00000000	00010203	00000000	00010206
AluY[31:0]	0000000a	00000000	00000003	00010203	00000000
AluResult[31:0]	0000000a	00000000	00010203	00010206	00000001
InData[31:0]		00000000		00000003	00010203
OutData[31:0]	040	08090a0b	0000000d		
InData[31:0]	04050607	08090a0b	0000000d	00000003	00010203

图 47 add&slt 仿真

由上图可以看出，在第一次取到 ADD 指令（00508333）时，译码阶段发现发生了 LOAD 型数据冒险，于是在紧邻的第二个周期暂停一个周期，不取新的指令，而是重新取本指令，与之前介绍的 LOAD 冒险的处理方法一致。在第二个周期重新取得 ADD 指令后开始译码，这时又满足 MEM 型冒险，所以在寄存器中取出的数据（00000000）并不是正确的数据；到了 EX 阶段，在 ALUY 前面通过数据前推拿到正确数据 00010203，这正好时上面 LOAD BYTE 的结果，与 ALUX 计算加法得到 00010206（AluResult）；ADD 指令不访存（读写存储器信号为低电平）；在 WB 阶段，写回寄存器 00010206，完成指令执行。

InsAddr[31:0]	00000018	0000001c	00000020	00000024	00000028
Instruction[31:0]	0013a133	00030393	00302023	00638433	008084b3
OutDataA[31:0]	00010203		00000000		
OutDataB[31:0]	00000000	00010203	00000000	08090a0b	00010206
AluX[31:0]		00010203	00000000	00010206	00000000
AluY[31:0]	00000000	00000000	00010203	00000000	
AluResult[31:0]	00010203	00010206	00000001	00010206	00000000
InData[31:0]	00000000		00000003	00010203	00000000
OutData[31:0]	000				
InData[31:0]	0000000d	00000003	00010203	00010206	00000001

图 48 slt 仿真

而第二条 SLT 指令没有数据冒险。在 IF 阶段根据 PC（00000018）取得指令（0013a133）；在 ID 阶段读出两个寄存器中的值分别为 x1（00010203）、x7（00000000）；在 EX 阶段将两个数据比大小  $x7 < x1$  所以  $x2 = 00000001$ ；没有 MEM 阶段（读写存储器信号为低电平）；在 WB 阶段写回 x2 寄存器 00000001。

```
//addi r7,r6 ,imm      r6=00010206 imm=00000000
```

```
//r7=r6+imm=00010206+00000000=00010206
```

下一条指令为 I 类型指令，ADDI，将 x6 和立即数相加结果存到 x7 中。与上面第二条指令 ADD r6,r5,r1 发生数据冒险，指令距离为 2，需要 MEM 阶段的数据前推。如下图所示。

InsAddr[31:0]	0000001c	00000020	00000024	00000028	0000002c
Instruction[31:0]	00030393	00302023	00638433	008084b3	00940533
OutDataA[31:0]		00000000			00010203
OutDataB[31:0]	00010203	00000000	08090a0b	00010206	00000000
Imm[31:0]		00000000			00000008
AluX[31:0]	0203	00000000	00010206	00000000	00010206
AluY[31:0]	00000003	00010203	00000000		00010206
AluResult[31:0]	00010206	00000001	00010206	00000000	0002040c
InData[31:0]		00000003	00010203	00000000	08090a0b
OutData[31:0]					00000003
InData[31:0]	00000003	00010203	00010206	00000001	00010206

图 49 addi 仿真

IF 阶段根据 PC (0000001c) 取指 (00030393); 在 ID 阶段由于发生数据冒险, 此时取出的寄存器 1 中的数据 00000000 并不是正确的数据, 另一操作数是由 Ext 模块产生的立即数 Imm=00000000; 在 EX 阶段, 根据 MEM 冒险的前推, 将 MEM 阶段的数据前送到 ALUX, 与 ALUY 中的立即数执行加法计算, 结果为 00010206; MEM 不访存; 在 WB 阶段将其写回到寄存器中 (InData=00010206)。

```
//sw r3 [r0+imm]      r3=08090a0b
```

下一条时 STORE WORD 指令, 将 x3 中的数据 (08090a0b) 存到存储器中, 简便起见将访存地址设为  $x0+imm=00000000+00000000=00000000$ 。

InsAddr[31:0]	00000020	00000024	00000028	0000002c	00000030
Instruction[31:0]	00302023	00638433	008084b3	00940533	00209063
OutDataA[31:0]	00000000			00010203	00000000
OutDataB[31:0]	00000000	08090a0b	00010206	00000000	
Imm[31:0]	00000000			00000008	
AluX[31:0]	00000000	00010206	00000000	00010206	00010203
AluY[31:0]	00010203	00000000		00010206	0002040c
AluResult[31:0]	00000001	00010206	00000000	0002040c	0003060f
InData[31:0]	00000003	00010203	00000000	08090a0b	00010206
OutData[31:0]					00000003
InData[31:0]	00010203	00010206	00000001	00010206	00000000

图 50 store 仿真



IF 阶段根据 PC (00000020) 取得指令 00302023; 在 ID 阶段, 取出寄存器 1 中的值 00000000, 和立即数 00000000; 在 EX 阶段将二者相加得到结果 00000000; 在 MEM 阶段, STORE 需要写入存储器, 访存地址为上个阶段的 AluResult=00000000, 在上图中倒数第二行将 x3 中的数据 08090a0b 写入 0 号存储单元 (Indata=08090a0b)。WB 阶段不回写 (RegWrite 为低电平)。

//add r8 r6,r7      与 addi r7,r6,imm 发生冲突  $r8=r6+r7=0002040c$

//add r9 r8,r1      与 add r8 r6,r7 发生冲突  $r9=0003060f$

//add r10 r9,r8      与 addi r7,r6,imm & add r8 r6,r7 冲突  $r10=00050a1b$

上述三条指令均为 R 类型 ADD 指令, 均发生了数据冒险 (既有 EX 冒险也有 MEM), 分析方法与上面类似, 这里不在赘述, 仅观察结果。

Instruction[31:0]	00638433	008084b3	00940533	00209063	00a00203	00000283	00002083
OutDataA[31:0]	00000000	00010203	00000000	00010203			
OutDataB[31:0]	08090a0b	00010206	00000000	00000001		00000000	
Imm[31:0]	00000000		00000008		00000000	0000000a	0000
AluX[31:0]	00010206	00000000	00010206	00010203	0002040c	00010203	00010202
AluY[31:0]		00000000	00010206	0002040c	0003060f	00000001	00010202
AluResult[31:0]	00010206	00000000	0002040c	0003060f	00050a1b		00010202
InData[31:0]	00010203	00000000	08090a0b	00010206	0002040c	0003060f	00000001
OutData[31:0]					00000003		
InData[31:0]	00010206	00000001	00010206	00000000	0002040c	0003060f	00050a1b

图 51 add 指令仿真

与 ADDI r7,r6,imm 发生冒险  $x6=000102006$ ,  $x7=00010206$ ,  $r8=x6+x7=0002040c$ 。ADD r9,r8,r1 与 ADD r8 r6,r7 发生冒险  $x1=00010203$ ,  $x8=0002040c$ ,  $x9=0003060f$ 。ADD r10 r9,r8 与 ADDI r7,r6,imm & ADD r8 r6,r7 冒险,  $x8=0002040c$ ,  $x9=0003060f$ ,  $x10=x8+x9=00050a1b$ 。

//beq r1 r2 ->insMem[0]      taken

//lb r4     $r4=0000000d$       //无效指令, as bne is taken

//lb r5     $r5=00000003$       //无效指令, as bne is taken

最后几条指令时跳转指令相关。若  $x1$  和  $x2$  相等则跳转到第一条指令处, 开始循环。根据之前的指令可知,  $x1$  和  $x2$  相等, 故 B 指令跳转。

Instruction[31:0]	00209063	00a00203	00000283	00002083	00402103	00802183	00a00203
OutDataA[31:0]	00000000	00010203	00000000				
OutDataB[31:0]	0000	00000001	00000000		0000000d	0002040c	
Imm[31:0]		00000000	0000000a	00000000		00000004	00000008
AluX[31:0]	00010203	0002040c	00010203	00010202	00000000		
AluY[31:0]	0002040c	0003060f	00000001	00010202	00000000	00000004	
AluResult[31:0]	0003060f	00050a1b	00010202		00000000	00000004	
InData[31:0]	00010206	0002040c	0003060f	00000001	00010202	00000000	
OutData[31:0]	00000003						0809
InData[31:0]	00000000	0002040c	0003060f	00050a1b	00010202	00000003	

图 52 B 指令仿真

取指 00209063 为 BNE 指令后续两条为 LOAD BYTE 指令，放在此处为了观测这两条并不会被执行，是无效的。在 LB r5 指令后面接着取指 00002083 正是第一条指令，完成跳转，说明在 EX 阶段发现 B 指令跳转后，在新的周期从目标地址取指成功。最后一行的 Indata 只是用来对指令结果进行定位，因为 B 指令和紧跟的两条无效指令并不会执行到 WB 阶段。由于在新地址取指执行，在 Indata 的 00000003 后面应该是第一条指令的结果了。

#### ● 程序测试

0#0            ADDI x8,x0,0;

4#4            start: ADDI x6,x0,0;

8#8            ADDI x5,x0,0;

C#12            ADDI x7, x0,3; #循环次数

10#16            LUI x10,0x00080;

前五条完成寄存器初始化的操作。将 x8、x6、x5 清空，将 x7 置为 3 作为 loop 循环次数，将 x10 置为 0x80000000。仿真如下。

Instruction[31:0]	00000413	00000313	00000293	00300393	00080537	00130313	00639a63	00455513	00000313
OutDataA[31:0]	XXXXXXXX					00000000			
OutDataB[31:0]	XXXXXXXX					00000000			
InsAddr[31:0]	00000000	00000004	00000008	0000000c	00000010	00000014	00000018	0000001c	00000020
AluX[31:0]	XXXXXXXX					00000000			
AluY[31:0]	XXXXXXXX			00000000		00000003	00080000	00000001	
AluResult[31:0]	XXXXXXXX			00000000		00000003	80000000	00000001	ffff
InData[31:0]	XXXXXXXX					00000000			
InData[31:0]		XXXXXXXX				00000000		00000003	80000000

图 53 ADDI 仿真

在 Instruction 一行表示取得的指令，依次为 00000413、00000313、00000293、00300393、00080537 分别对应上述五条指令，其在 WB 阶段依次向目的寄存器中写入对应得数据 0x00000000、0x00000000、0x00000000、0x00000003、0x80000000。

```
14#20 loop:ADDI x6,x6,1;
```

```
18#24 BNE x6,x7,loop;
```

接下来执行一个小循环。不断将 x6 和 x7 比较，若不相等就跳回 loop 出，起始 x6 置为 0，x7 置为 3，所以循环三次。仿真如下。

Instruction[31:0]	00130313	00639a63	00455513	00000313	00130313	00639a63	00455513	00000313
OutDataA[31:0]				00000001		00000001	00000003	80000000
OutDataB[31:0]				00000000			00000001	
InsAddr[31:0]	00000014	00000018	0000001c	00000020	00000014	00000018	0000001c	00000020
AluX[31:0]				00000000	ffffffff	00000000	00000001	00000003
AluY[31:0]	00000003	00080000	00000001	ffffffff	00000004	00000001	00000002	
AluResult[31:0]	00000003	80000000	00000001	ffffffff	00000000	00000002	0000	
InData[31:0]				00000000	00000001	ffffffff	00000000	
InData[31:0]	00000000	00000003	80000000	00000001	ffffffff	XXXXXXXX	00000000	

图 54 ADDI&amp;BNE 仿真

在 instruction 一行表示取得的指令，依次为 00130313、00639a63、00455513、00000313 分别对应 ADDI、BNE、两条由于 B 指令取得的错误指令，可以看到 ADDI 在 WB 阶段向目标寄存器中写入 00000001（ADDI 指令 0 加一），跳过两条错误指令，00000000 是 B 指令（由于 WriteReg 信号为低电平，并不会写入）。也可以看到后面跟着的四条指令也是与这四条相同的，这也验证了本段代码会循环三次。每一次 x6 中的值加一并于 x7 比较，直到第三次跳转条件满足，并不会回退，完成跳转。

```
1C#28 SRLI x10, x10, 4;
```

```
20#32 ADDI x6,x0,0; #清空 x6
```

循环结束后面两条指令为将 0x80000000 逻辑右移一位，并将 x6 清空。

Instruction[31:0]	00130313	00639a63	00455513	00000313	fff54913	0122a023	00428293	00a01a63
OutDataA[31:0]	00000000	00000002	00000003	80000000	00000000	80000000		
OutDataB[31:0]		00000000	00000002			00000000		
InsAddr[31:0]	00000014	00000018	0000001c	00000020	00000024	00000028	0000002c	00000030
AluX[31:0]	00000001	00000000	00000002	00000003	80000000	00000000	08000000	
AluY[31:0]	00000001	00000004	00000001	00000003	00000004	00000000	0000ffff	00000000
AluResult[31:0]	00000001	00000000	00000003	00000000	08000000	00000000	0800ffff	00000000
InData[31:0]	00000002	00000001	00000000	00000003			00000000	
InData[31:0]	00000002	00000001	XXXXXXXX	00000000	00000003	00000000	08000000	00000000

图 55 SRLI&ADDI 仿真

可以看到在 instruction 一行前面四条和上面一样为一次循环，但本次到 00455513 时并不会变为废弃指令，而会执行，在 WB 阶段写入 08000000（左移一位），和 00000000（将 x6 清空）。

24#34      ANDI x18,x10,all one;

28#40      SW x18,0(x5);

2C#44      ADDI x5,x5,4;

之后三条指令（上）功能分别为将 x10 中的值（刚才左移一位的值）取反，将其存到 0 号单元中，在这里 x5 作为偏移寄存器，初始值为 0，在存过一次后加四。

Instruction[31:0]	fff54913	0122a023	00428293	00a01a63	00000413	00130313
OutDataA[31:0]	00000000	80000000			00000000	
OutDataB[31:0]		00000000		08000000		
InsAddr[31:0]	00000024	00000028	0000002c	00000030	00000034	00000038
AluX[31:0]	80000000	00000000	08000000		00000000	f8000000
AluY[31:0]	00000004	00000000	0000ffff	00000000	00000004	08000000
AluResult[31:0]	08000000	00000000	0800ffff	00000000	00000004	f8000000
InData[31:0]	00000003		00000000		0800ffff	00000000
InData[31:0]	00000003	00000000	08000000	00000000	0800ffff	00000004

图 56 ADDI&SW 仿真

Instruction 一行中前三条指令，即为 ANDI、SW、ADDI。可以看到在 indata 处将 x10 中的值取反并存到了存储器中（上面的 indata 为访存，下面的为写寄存器，这里是处理了数据冒险，所以写存储器和写寄存器同时发生）。最后的 00000004 为偏移寄存器 x5 更新加四（每个存储单元四个字节，指向下一个单元）。

30#48      BNE x10, x0, loop;

之后时跳转指令，比较 x10 和 x0 中的值，若不等就跳转到 loop。X0 中的值恒为 0，而 x10 为左移的寄存器，起始为 80000000，每次左移一位，移完 8 次就等于 0 不会跳转了。刚才移动一次为 08000000，显然跳转。如下图。

Instruction[31:0]	00a01a63	00000413	00130313	00639a63	00455513	00000313	00130313
OutDataA[31:0]	00000000				00000003	08000000	00000000
OutDataB[31:0]	00000000	08000000	00000000				
InsAddr[31:0]	00000030	00000034	00000038	00000014	00000018	0000001c	00000020
AluX[31:0]	00000000		f8000000	00000000		00000003	00000002
AluY[31:0]	00000000	00000004	08000000	f8000000	00000000	00000001	00000002
AluResult[31:0]	00000000	00000004	f8000000	00000000	00000001	00000002	
InData[31:0]	00000000	08000fff	00000000	08000000	f8000000	00000000	00000001
InData[31:0]	00000000	08000fff	00000000	00000000	f8000000	XXXXXXXX	00000000

图 57 B 指令仿真

指令 00a01a63 即为本条跳转指令。可以看到在取完两条错误指令后，重新在 loop 处取指 00130313。而最下面对应的寄存器 Indata 的值仅供定位指令，B 指令和错误的两条指令并不写寄存器，并没有意义。可以看到一共占据了三个周期。后面的周期跟着的就是 00000001，为 loop 处的 x6 加一操作。

由于本次跳转指令比较 x10 和 x0 的值，据上面分析需要循环 8 次（再有 7 次，共 8 次），等到第八次循环 x10 移位到 00000000 时与 x0 相等，B 指令不跳转继续向下执行。这样做结果为将 80000000 中的 8 向右移动 7 次，每次移动后将其取反存入存储器中。

00a01a63	00000413	00040263	00000413	00000313	00000293	00300393	00080537	0013		
0000001c	00000000									
			00000000							
00000030	00000034	00000038	0000003c	00000040	00000044	00000004	00000008	0000000c	00000010	0000
0000001c		00000000								
00000000	00000004	00000000							0000	
0000001c	00000020	00000000							0000	
00000000	00000fff	00000000								
00000000	00000fff	0000001c	00000020	00000000	XXXXXXXX		00000000			

图 58 最后一次循环 B 指令仿真

直到最后一次执行 00a01a63 这条跳转指令时，判断条件发现不跳转，继续执行下面设置的两条空指令（观测 B 指令错取两条指令而设置的，空指令

为把 x8 清 0)，下面三个红框即为 B 指令和两条空指令对应的位置。

3C#60      BEQ x0, x8, start

40#64      ADDI x8,x0,0; #nop

44#68      ADDI x8,x0,0; #nop

接下来会执行一条跳转指令，跳转指令后面跟着两条废弃指令（程序末尾）。而这条跳转指令是一定会跳转的，因为 x8 和 x0 都为 0（x8 刚刚右移为 0，x0 恒为 0），这样就形成了一个无限循环。



图 59 B 指令仿真

上面为 B 指令和两条废弃指令的取指。同时可以看到，后面又从新开始在 start 处取指了 00000313、00000293.....

## 5.3 开发板运行

本模块将介绍如何将 CPU 下载到开发板上运行，其中关键的步骤以及 8 段数码管的显示机理。

### 5.3.1 八位七段数码管

数码管为共阴极数码管，即公共极输入低电平。共阴极由三极管驱动，FPGA 需要提供正向信号。同时段选端连接高电平，数码管上的对应位置才可以被点亮。因此，FPGA 输出有效的片选信号和段选信号都应该是高电平。

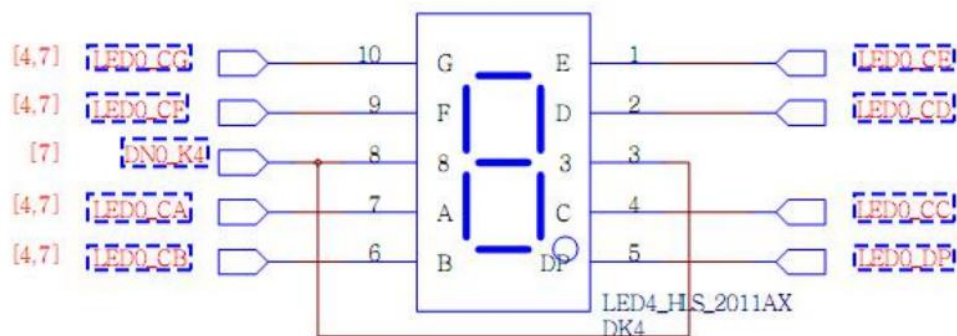


图 60 七段数码管

- 管脚约束

表格 28 八位七段数码管管脚约束表

名称	原理图标号	FPGA IO PIN
A0	CA0	B4
B0	CB0	A4
C0	CC0	A3
D0	CD0	B1
E0	CE0	A1
F0	CF0	B3
G0	CG0	B2
DP0	DP0	D5
A1	CA1	D4
B1	CB1	E3
C1	CC1	D3
D1	CD1	F4
E1	CE1	F3
F1	CF1	E2
G1	CG1	D2
DP1	DP1	H2

- 显示原理

8 位 7 段译码管分为高 4 位和低 4 位两部分，这两部分分别由不同的引脚控制。所以在亮灯方面设置了两个信号，seg、seg1 分别控制高位和低位，但 sel 信号只有一个。

开发板每次只可实现一位的显示（通过 sel 信号选通，高电平选通，比如

00000001 是最低位亮)，即 1 个 8 的显示，所以，要想实现多位显示，就要在极短的时间内在 8 位中来回选通（极短时间导致视觉暂留）。

但每个数据产生后只停留 1 个时钟周期，导致无法切换位选，所以要进行分频。即在一个数据停留的周期内，对八位循环选通，当一位被选通时，这一位灯亮，由于在这个数据停留周期（即 CPU 周期）内数据是不变的（停留），所以切换到每一位时，仅需把数据中的这一位选出显示即可。

每一位的循环显示时间很短，所以视觉上是八位在同时显示。

设计中设置的是 50000000 倍分频，即 50000000 个时钟周期作为 CPU 的一个周期。实现位选是在这 CPU 周期内，设置的位选分频为 50000，即 50000 个时钟周期后位选信号变化一次，但还是时间很短，可以视觉暂留实现同时显示。

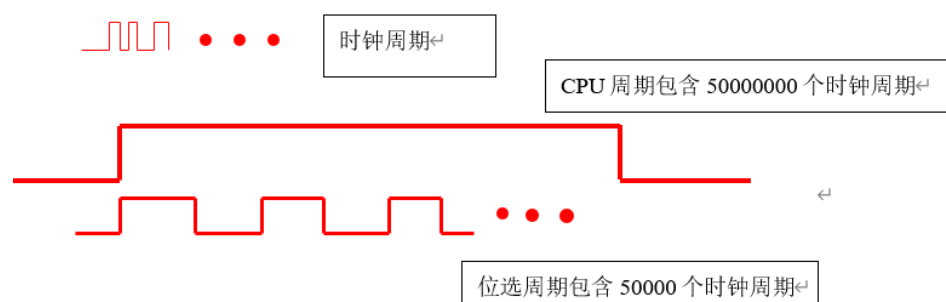


图 61 分频

### ● 具体实现

```
if(cnt_d == maxcnt)begin
    -----clk_1k <= ~clk_1k;
    -----cnt_d <= 0;
end else
    -----cnt_d <= cnt_d + 1'b1;
end
```

分频实现（上）设置计数器 cnt\_d，和分频倍数 maxcnt，当时钟上升沿到来时，计数器加一。等到时钟数量达到分频倍数时产生一个分频时钟信号。

```
always@(posedge clk_1k)begin
    case(disb_bit)
        4'b0000:begin
```



```
sel_reg<=8'b00000001;
```

```
data_reg<=data[3:0];disp_bit<=disp_bit+1'b1;
```

```
end
```

```
.....
```

换位显示（上）根据换位分频时钟，每周期将所要展示的位加一，到 8 清零，这样便能循环展示八位。到具体某一位设置本位控制信号，包括位选控制信号 sel\_reg 和数据信号 data\_reg。

```
Case(data_reg):
```

```
4'b0000:seg<=7'b1111110;
```

```
.....
```

十六位数显示（上）将 data\_reg 中十六位的数据转换成 7 段数码管的数显示。注意位选信号分高四位和低四位，所以要判断位选控制信号高电平处于哪四位

### 5.3.2 开发板实现

本模块主要介绍下载到开发板上的过程，包括配置文件中各管脚的连接。

在整个 CPU 模块设计实现完成后，将其与控制开发板数码管的模块结合，形成顶层模块。在顶层模块中进行综合（Synthesis）与实现（Implementation），打开 IO 管脚控制窗口，将其如下连接。连接完成后会自动形成约束文件。

Name	Direction	Neg Diff Pair	Package Pin	Fixed	Bank	I/O Std	Vcco	Vref	Drive Strength	Slew Type
LEDL (7)	OUT			<input checked="" type="checkbox"/>	35	LVC MOS33*	-	3.300	12	SLOW
LEDL[6]	OUT		B4	<input checked="" type="checkbox"/>	35	LVC MOS33*	-	3.300	12	SLOW
LEDL[5]	OUT		A4	<input checked="" type="checkbox"/>	35	LVC MOS33*	-	3.300	12	SLOW
LEDL[4]	OUT		A3	<input checked="" type="checkbox"/>	35	LVC MOS33*	-	3.300	12	SLOW
LEDL[3]	OUT		B1	<input checked="" type="checkbox"/>	35	LVC MOS33*	-	3.300	12	SLOW
LEDL[2]	OUT		A1	<input checked="" type="checkbox"/>	35	LVC MOS33*	-	3.300	12	SLOW
LEDL[1]	OUT		B3	<input checked="" type="checkbox"/>	35	LVC MOS33*	-	3.300	12	SLOW
LEDL[0]	OUT		B2	<input checked="" type="checkbox"/>	35	LVC MOS33*	-	3.300	12	SLOW
LEDL1 (7)	OUT			<input checked="" type="checkbox"/>	35	LVC MOS33*	-	3.300	12	SLOW
LEDL1[6]	OUT		D4	<input checked="" type="checkbox"/>	35	LVC MOS33*	-	3.300	12	SLOW
LEDL1[5]	OUT		E3	<input checked="" type="checkbox"/>	35	LVC MOS33*	-	3.300	12	SLOW
LEDL1[4]	OUT		D3	<input checked="" type="checkbox"/>	35	LVC MOS33*	-	3.300	12	SLOW
LEDL1[3]	OUT		F4	<input checked="" type="checkbox"/>	35	LVC MOS33*	-	3.300	12	SLOW
LEDL1[2]	OUT		F3	<input checked="" type="checkbox"/>	35	LVC MOS33*	-	3.300	12	SLOW
LEDL1[1]	OUT		E2	<input checked="" type="checkbox"/>	35	LVC MOS33*	-	3.300	12	SLOW
LEDL1[0]	OUT		D2	<input checked="" type="checkbox"/>	35	LVC MOS33*	-	3.300	12	SLOW

图 62 IO ports










▼  sel (8)	OUT			<input checked="" type="checkbox"/>	35	LVC MOS33* ▼	3.300		12	▼	SLOW
 sel[7]	OUT		G2 ▼	<input checked="" type="checkbox"/>	35	LVC MOS33* ▼	3.300		12	▼	SLOW
 sel[6]	OUT		C2 ▼	<input checked="" type="checkbox"/>	35	LVC MOS33* ▼	3.300		12	▼	SLOW
 sel[5]	OUT		C1 ▼	<input checked="" type="checkbox"/>	35	LVC MOS33* ▼	3.300		12	▼	SLOW
 sel[4]	OUT		H1 ▼	<input checked="" type="checkbox"/>	35	LVC MOS33* ▼	3.300		12	▼	SLOW
 sel[3]	OUT		G1 ▼	<input checked="" type="checkbox"/>	35	LVC MOS33* ▼	3.300		12	▼	SLOW
 sel[2]	OUT		F1 ▼	<input checked="" type="checkbox"/>	35	LVC MOS33* ▼	3.300		12	▼	SLOW
 sel[1]	OUT		E1 ▼	<input checked="" type="checkbox"/>	35	LVC MOS33* ▼	3.300		12	▼	SLOW
 sel[0]	OUT		G6 ▼	<input checked="" type="checkbox"/>	35	LVC MOS33* ▼	3.300		12	▼	SLOW

图 63 IO ports

5.3.3 运行结果

在本模块中将展示设计完成的 CPU 在开发板上的运行显示结果。

详情见附件。

总结

分阶段依次完成了单周期 CPU、多周期 CPU、流水线 CPU、添加数据冒险检测与处理单元、添加控制冒险检测与处理单元，最终下载到开发板上显示。对 CPU 的设计与工作机制有了更为深刻的理解。

参考文献

[1] 张晨曦，王志英. 计算机体系结构（第二版）.2014.8. 高等教育出版社

附件

- 工程文件
- 开发板运行视频

## 一、原创性声明

本人郑重声明本报告内容，是由作者本人独立完成的。有关观点、方法、数据和文献等的引用已在文中指出。除文中已注明引用的内容外，本报告不包含任何其他个人或集体已经公开发表的作品成果，不存在剽窃、抄袭行为。

特此声明！

作者签字：丁盛鹏

	评价内容	权重	得分
验收		0.4	
设计报告		0.6	
合计			
指导教师（签章）： 2023 年 6 月 30 日			