兰州大学

32 位 MIPS 多周期 CPU 设计方案

（本科生）

2022 年春季学期计算机组成原理课程

负责人：韩旭达 联系方式：13078878819

刘文博 林骋楷 李清扬

班级：2020 级数据科学四班

2022 年 5 月 20 日星期五

一、 项目简介

1. 项目环境

1.1 设计语言：

Verilog 硬件设计语言

1.2 仿真环境：

Vivado 2018.3

2. 任务目标

2.1 课程设计要求：

设计一个兼顾 32 位 MIPS 或 RISC-V 指令集的多周期 CPU（核）

2.2 设计目标：

1. 实现无流水多周期 CPU；
2. 兼容 32 位 MIPS 指令集（部分）；
3. 实现指令覆盖 R 型、I 型计算类、I 型取数类、I 型存数类、I 型条件判断类、J 型；
4. 前仿成功；

3. 参考资料

数字逻辑与组成原理实践教程.清华大学出版社[引用日期 2022-05-20]

4. 项目文件夹结构

- MultiCycleCPU
  - Code
  - Project
  - Materials
  - Result
  - 设计方案.docx
  - 设计方案.pdf
  - 设计方案.pptx
  - 讲解视频.mov
  - Readme.txt

二、 实现细节

1. 指令集选取

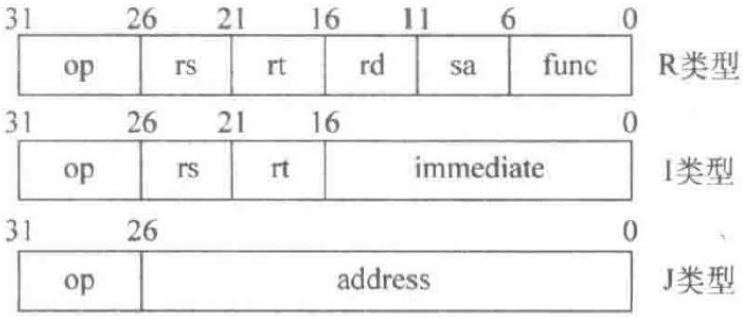
1.1 指令集：

MIPS32

1.2 指令类型：

R 型、I 型、J 型

1.3 指令格式：



1.4 指令操作：

1. R 型：该类型指令从寄存器堆中读取两个源操作数，计算结果写回寄存器堆。具体操作由op、$func$结合指定，rs和rt是源寄存器的编号，rd是目的寄存器的编号。
2. I 类型，该类型指令使用一个 16 位的立即数作为一个源操作数。具体操作由op指定，指令的低 16 位是立即数，运算时要将其扩展至 32 位，然后作为其中一个源操作数参与运算。
3. J 类型，该类型指令使用一个 26 位的立即数作为跳转的目标地址。具体操作由op指定，一般是跳转指令，低 26 位是字地址，用于产生跳转的目标地址。

1.5 完整 MIPS 指令集如下：

Mnemonic Symbol	Format						Sample
Bit #	31..26	25..21	20..16	15..11	10..6	5..0	
R-type	op	rs	rt	rd	shamt	func	
add	000000	rs	rt	rd	0	100000	add $1,$2,$3
addu	000000	rs	rt	rd	0	100001	addu $1,$2,$3
sub	000000	rs	rt	rd	0	100010	sub $1,$2,$3
subu	000000	rs	rt	rd	0	100011	subu $1,$2,$3
and	000000	rs	rt	rd	0	100100	and $1,$2,$3
or	000000	rs	rt	rd	0	100101	or $1,$2,$3
xor	000000	rs	rt	rd	0	100110	xor $1,$2,$3
nor	000000	rs	rt	rd	0	100111	nor $1,$2,$3
slt	000000	rs	rt	rd	0	101010	slt $1,$2,$3
sltu	000000	rs	rt	rd	0	101011	sltu $1,$2,$3
sll	000000	0	rt	rd	shamt	000000	sll $1,$2,10
srl	000000	0	rt	rd	shamt	000010	srl $1,$2,10
sra	000000	0	rt	rd	shamt	000011	sra $1,$2,10
sllv	000000	rs	rt	rd	0	000100	sllv $1,$2,$3
srlv	000000	rs	rt	rd	0	000110	srlv $1,$2,$3
srav	000000	rs	rt	rd	0	000111	srav $1,$2,$3
jr	000000	rs	0	0	0	001000	jr $31

Bit #	31..26	25..21	20..16	15..0	
I-type	op	rs	rt	immediate	
addi	001000	rs	rt	Immediate(- ~ +)	addi $1,$2,100
addiu	001001	rs	rt	Immediate(- ~ +)	addiu $1,$2,100
andi	001100	rs	rt	Immediate(0 ~ +)	andi $1,$2,10
ori	001101	rs	rt	Immediate(0 ~ +)	andi $1,$2,10
xori	001110	rs	rt	Immediate(0 ~ +)	andi $1,$2,10
lw	100011	rs	rt	Immediate(- ~ +)	lw $1,10($2)
sw	101011	rs	rt	Immediate(- ~ +)	sw $1,10($2)
beq	000100	rs	rt	Immediate(- ~ +)	beq $1,$2,10
bne	000101	rs	rt	Immediate(- ~ +)	bne $1,$2,10
slti	001010	rs	rt	Immediate(- ~ +)	slti $1,$2,10
sltiu	001011	rs	rt	Immediate(- ~ +)	sltiu $1,$2,10
lui	001111	00000	rt	Immediate(- ~ +)	Lui $1, 10

Bit #	31..26	25..0	
J-type	op	Index	
j	000010	address	j 10000
jal	000011	address	jal 10000

1.6 实现指令：

本设计我们选取MIPS32指令集中的七条指令进行实现，分别覆盖R型、I型、J型：

1. ADD：R型，求和指令；
2. SUB：R型，求差指令；
3. OR：R型，按位或指令；
4. LW：I型，取数指令；
5. SW：I型，存数指令；
6. BEQ：I型，条件判断跳转指令；
7. J：J型，跳转指令；

2. 设计原理

多周期CPU的中心思想是把一条指令的执行过程分成若干个小周期完成，根据每条指令的复杂程度，使用不同数量的小周期去执行，许多个小周期加在一起相当于单周期CPU中的一个周期。

在我们实现的7条指令中，最复杂的指令之一是$BEQ\ rs,rt,offset$，它需要5个周期，其整个执行的过程如下：

1） 根据pc取指令；

2） 完成pc加4；

3） 读出rs和rt两个寄存器的数据并锁存，比较是否相等；

4） 相等，ALU计算转移地址并锁存，否则指令结束；

5） 将转移地址写入pc，指令结束；

而最简单的指令是$J\ target$，具有以下三个周期：

1） 根据pc取指令；

2） 完成pc加4；

3） 指令中的$target$左移两位与pc的高4位拼接成32位地址写入pc；

ALU计算类型的指令需要以下4个周期：

1） 根据pc取指令；

2） 完成pc加4；

3） 读出rs和rt两个寄存器的内容进行运算；

4） 把运算结果写入寄存器堆中的rd（或rt）寄存器；

访问内存类型的指令，需要以下 4 个周期：

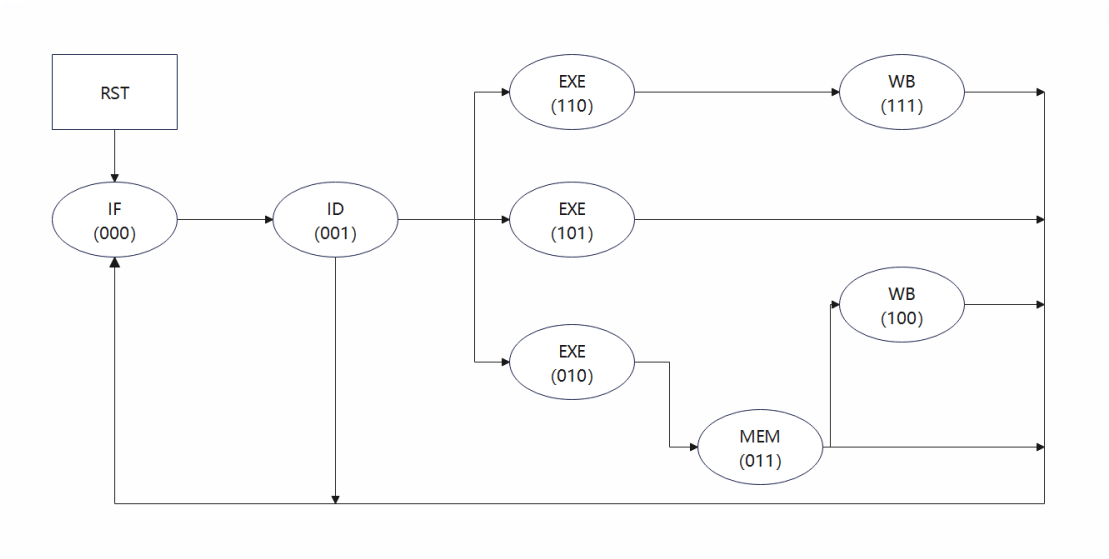
1） 根据pc取指令；

2） 完成pc加 4；

3） rs寄存器的内容与指令中的偏移量$offset$相加，计算得到的存储器地址；

4） 使用计算好的地址访问存储器，从中读写出一个 32 位的数据；

由上述给出的特定指令执行过程可以看出，多周期 CPU 在处理指令时，一般需要经过IF（取指）、ID（译码）、EXE（执行）、MEM（访存）、WB（写回）五个阶段。执行一条指令最长需要五个时钟周期。下图为多周期 CPU 状态转移图，状态转移图不是唯一的，只要能实现各条指令所经过的周期即可：



在多周期 CPU 核中确定状态转移图及输出逻辑的部件是控制单元，我们用有限状态机实现多周期 CPU 的控制部件，可以用时序电路来实现多周期 CPU 的控制单元。多周期控制器逻辑结构图如下：

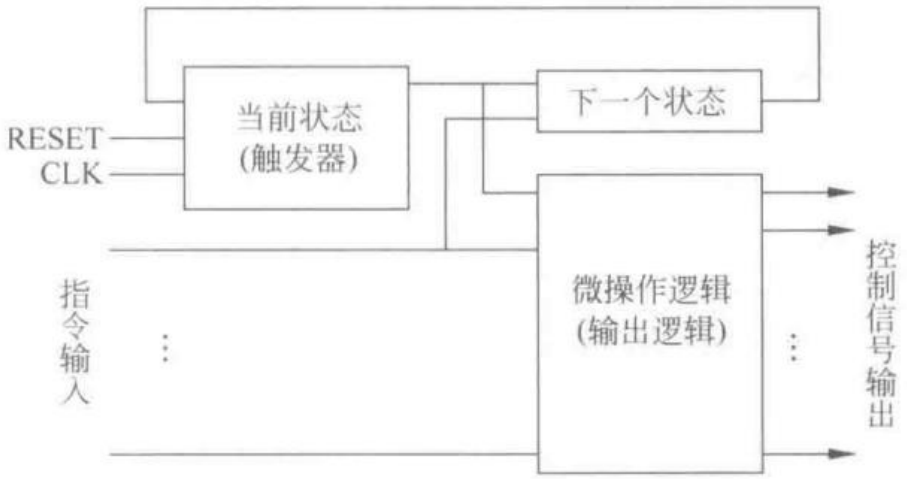


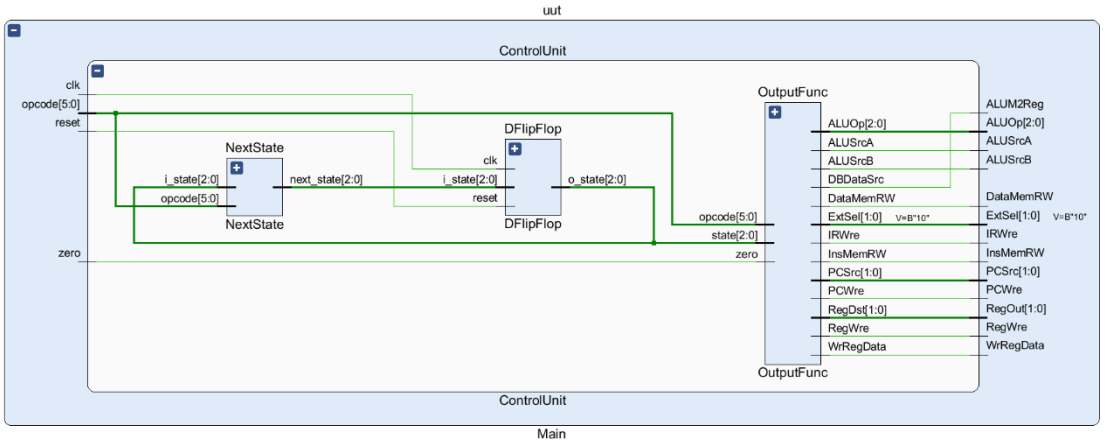
图 7.31　多周期控制器逻辑结构

3. 模块设计

3.1　控制器设计：

在控制器中设计一个状态寄存器（D 触发器）用于保存当前状态，属于时序逻辑电路，当 RST 复位信号来临时初始化状态为‘000’（*IF*阶段）。再设计两个组合逻辑电路，一个（状态机）用于产生下一阶段的状态，另一个（硬布线控制器）用于产生每个阶段的控制信号。下一状态取决于指令操作码和当前状态，而每个阶段的控制信号取决于指令操作码、当前状态和反映运算结果的 zero 状态标志。

- **控制器结构：**



- **控制器描述：**

```
`timescale 1ns / 1ps
```

```
module ControlUnit(opcode, clk, reset, zero, PCWre,
InsMemRW, IRWre, WrRegData, RegWre, ALUSrcA, ALUSrcB,
DataMemRW, ALUM2Reg, ExtSel, RegOut, PCSrc, ALUOp);
    input [5:0]opcode;
    input zero, clk, reset;
    output PCWre, InsMemRW, IRWre, WrRegData,
RegWre,ALUSrcA, ALUSrcB, DataMemRW, ALUM2Reg;
    output [1:0]ExtSel, RegOut, PCSrc;
    output [2:0]ALUOp;

    wire [2:0]i_state, o_state;

    DFlipFlop DFlipFlop(i_state, reset, clk, o_state);
    NextState NextState(o_state, opcode, i_state);
    OutputFunc OutputFunc(o_state, opcode, zero, PCWre,
InsMemRW, IRWre, WrRegData, RegWre, ALUSrcA, ALUSrcB,
DataMemRW, ALUM2Reg, ExtSel, RegOut, PCSrc, ALUOp);

endmodule
```

- 状态寄存器（D 触发器）描述：

```
`timescale 1ns / 1ps

module DFlipFlop(i_state, reset, clk, o_state);
    input [2:0]i_state;
    input reset, clk;
    output reg[2:0]o_state;

    always @(posedge clk) begin
        if (reset) o_state = 3'b000;
        else o_state = i_state;
    end

endmodule
```

- 状态机描述：

```
`timescale 1ns / 1ps
module NextState(i_state, opcode, next_state);
    input [2:0]i_state;
    input [5:0]opcode;
    output reg[2:0]next_state;
    parameter [2:0] IF = 3'b000,
                    ID = 3'b001,
                    aEXE = 3'b110,
```

```
                    bEXE = 3'b101,
                    cEXE = 3'b010,
                    MEM = 3'b011,
                    aWB = 3'b111,
                    cWB = 3'b100;
    always @(i_state or opcode) begin
        case (i_state)
            IF: next_state = ID;
            ID: begin
                case (opcode[5:3])
                    3'b110: begin
                        if (opcode == 6'b110100) next_state
= bEXE;
                        else next_state = cEXE;
                    end
                    3'b111: next_state = IF;
                    default: next_state = aEXE;
                endcase
            end
            aEXE: next_state = aWB;
            bEXE: next_state = IF;
            cEXE: next_state = MEM;
            MEM: begin
                if (opcode == 6'b110001) next_state = cWB;
                else next_state = IF;
            end
            aWB: next_state = IF;
            cWB: next_state = IF;
            default: next_state = IF;
        endcase
    end
endmodule
```

- 硬布线控制器描述：

```
`timescale 1ns / 1ps

module OutputFunc(state, opcode, zero, PCWre, InsMemRW,
IRWre, WrRegData, RegWre, ALUSrcA, ALUSrcB, DataMemRW,
DBDataSrc, ExtSel, RegDst, PCSrc, ALUOp);
    input [2:0]state;
    input [5:0]opcode;
    input zero;
    output reg PCWre, InsMemRW, IRWre, WrRegData, RegWre,
ALUSrcA, ALUSrcB, DataMemRW, DBDataSrc;
```

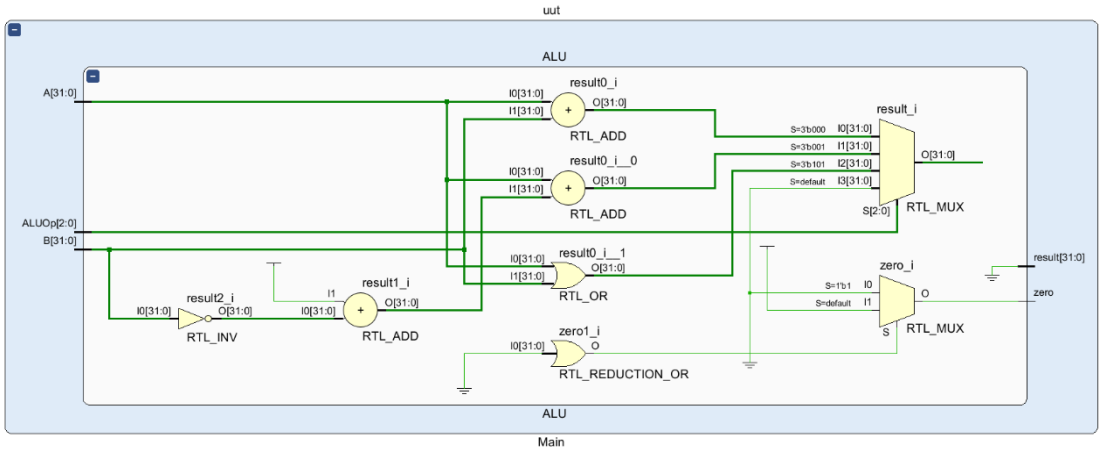
```
output reg[1:0]ExtSel, RegDst, PCSrc;
output reg[2:0]ALUOp;
parameter [2:0] IF = 3'b000,
                ID = 3'b001,
                aEXE = 3'b110,
                bEXE = 3'b101,
                cEXE = 3'b010,
                MEM = 3'b011,
                aWB = 3'b111,
                cWB = 3'b100;
parameter [5:0] add = 6'b000000,
                sub = 6'b000001,
                sw = 6'b110000,
                lw = 6'b110001,
                beq = 6'b110100,
                j = 6'b111000,
                Or = 6'b010000,
                halt = 6'b111111;

always @(state) begin
    if (state == IF && opcode != halt) PCWre = 1;
    else PCWre = 0;
    InsMemRW = 1;
    if (state == IF) IRWre = 1;
    else IRWre = 0;
    if (state == aWB || state == cWB) WrRegData = 1;
    else WrRegData = 0;
    if (state == aWB || state == cWB) RegWre = 1;
    else RegWre = 0;
ALUSrcA = 0;
    if (opcode == sw || opcode == lw) ALUSrcB = 1;
    else ALUSrcB = 0;
    if (state == MEM && opcode == sw) DataMemRW = 1;
    else DataMemRW = 0;
    if (state == cWB) DBDataSrc = 1;
    else DBDataSrc = 0;
    ExtSel = 2'b10;
if (opcode == lw) RegDst = 2'b01;
    else RegDst = 2'b10;
    case(opcode)
        j: PCSrc = 2'b11;
        beq: begin
            if (zero) PCSrc = 2'b01;
            else PCSrc = 2'b00;
```

```
            end
            default: PCSrc = 2'b00;
        endcase
        case(opcode)
            sub: ALUOp = 3'b001;
            Or: ALUOp = 3'b101;
            beq: ALUOp = 3'b001;
            default: ALUOp = 3'b000;
        endcase
        if (state == IF) begin
            RegWre = 0;
            DataMemRW = 0;
        end
    end

endmodule
```

3.2 ALU 设计：

ALU 模块接收两个源操作数和控制码作为输入，输出计算结果和零标志位。初始化输出结果为 0。在多周期 CPU 的*EXE*（执行）阶段，每当时钟上升沿到来时，根据控制模块给出的控制信号进行相应计算。

- ALU 结构：



- ALU 描述：

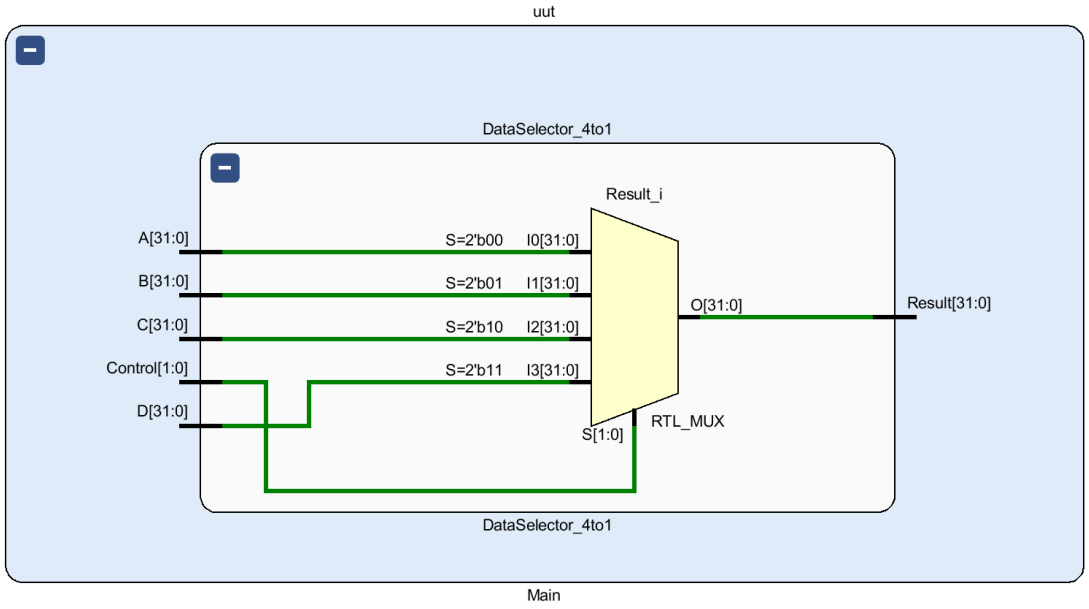
```
`timescale 1ns / 1ps

module ALU(A, B, ALUOp, zero, result);
    input [31:0] A, B;
    input [2:0] ALUOp;
    reg [31:0] reg_A,reg_B;
    output zero;
    output reg [31:0] result;
    initial begin
```

```
            result = 0;
        end
        assign zero = (result? 0 : 1);
        always @(A or B or ALUOp) begin
            case(ALUOp)
                3'b000: begin reg_A=A; reg_B=B; result = reg_A +
    reg_B; end
                3'b001: begin reg_A=A; reg_B=(~B+1);result =
    reg_A + reg_B; end
                3'b101: result = A | B;
                default: result = 0;
            endcase
        end
    endmodule
```

3.3 选择器设计：

多周期 CPU 核中涉及多种多路选择器：2 选 1、3 选 1、4 选 1，但其结构类似。多路选择器根据控制器给出的控制信号选择相对应的输入进行输出。

- **多路选择器结构：**



- **多路选择器描述：**

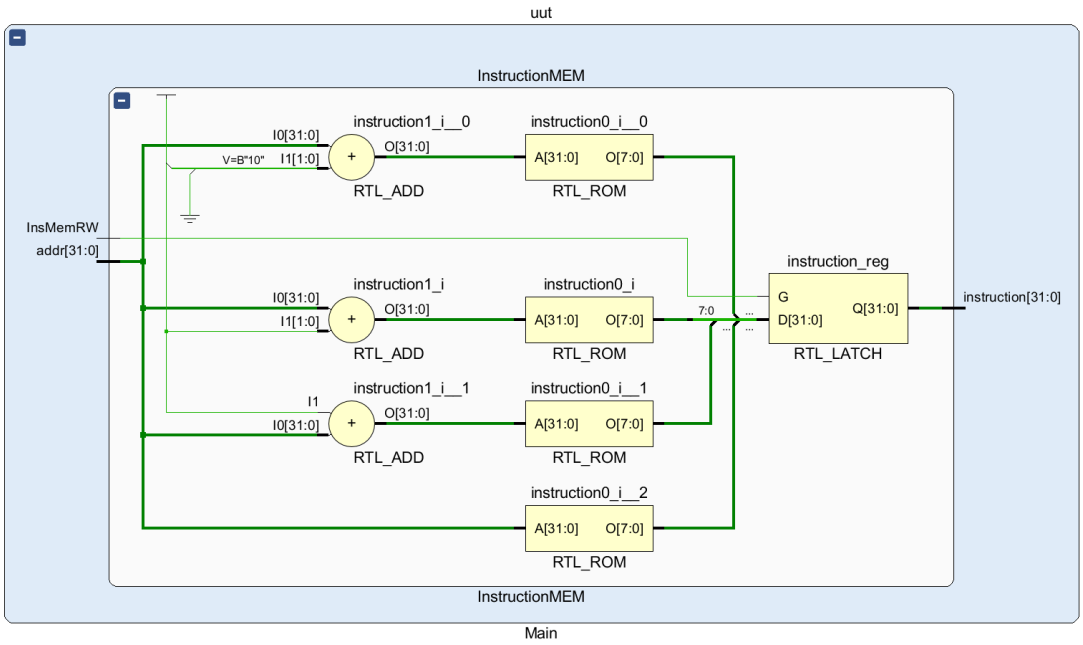
```
`timescale 1ns / 1ps

module DataSelector_4to1(A, B, C, D, Control, Result);
    input [31:0] A, B, C, D;
    input [1:0]Control;
    output reg[31:0] Result;
```

```
    always @(Control or A or B or C or D) begin
        case(Control)
            2'b00: Result = A;
            2'b01: Result = B;
            2'b10: Result = C;
            2'b11: Result = D;
            default: Result = 0;
        endcase
    end

endmodule
```

3.4 存储器设计：

多周期 CPU 核中涉及两种存储器：指令存储器、数据存储器，两者结构类似。其核心功能负责连续读取/写入 4 个 8 位二进制数。

- **指令存储器结构：**

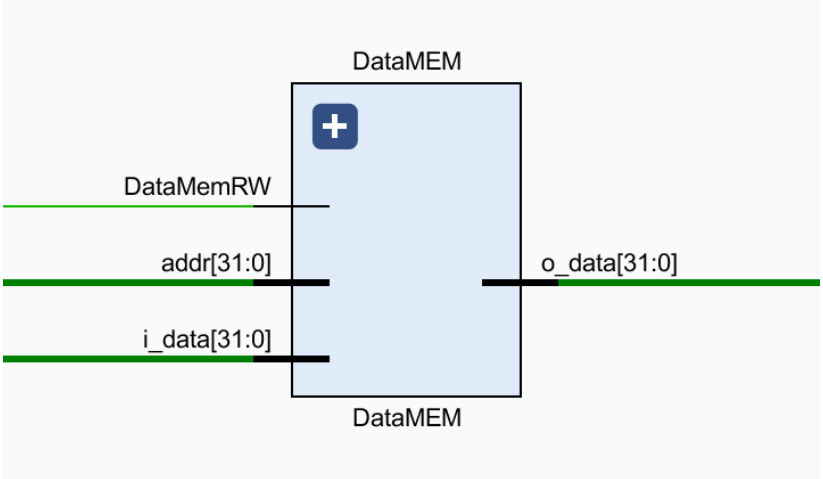


- **指令存储器描述：**

```
`timescale 1ns / 1ps

module InstructionMEM (addr, InsMemRW, instruction);
    input InsMemRW;
    input [31:0] addr;
    output reg [31:0] instruction;
    reg [7:0] mem [0:127];
    initial begin
        $readmemb("E:\\Courses\\Computer_Organization\\Mult
iCycleCPU\\Code\\instructions.txt", mem);
```

```
            instruction = 0;
        end
        always @(addr or InsMemRW) begin
            if (InsMemRW) begin
                instruction[31:24] = mem[addr];
                instruction[23:16] = mem[addr+1];
                instruction[15:8] = mem[addr+2];
                instruction[7:0] = mem[addr+3];
            end
        end

    endmodule
```

- 数据存储器结构：



- 数据存储器描述：

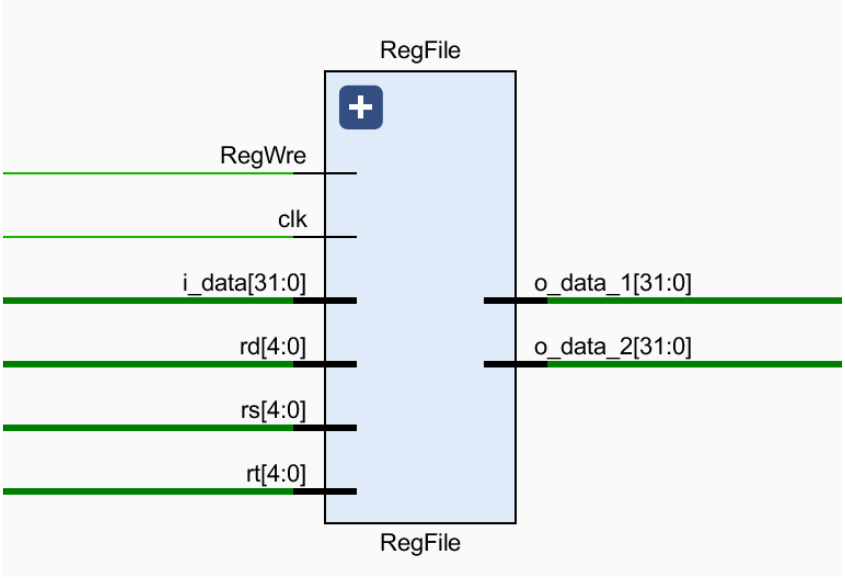
```
    `timescale 1ns / 1ps

    module DataMEM (i_data, addr, DataMemRW, o_data);
        input [31:0] i_data;
        input [31:0] addr;
        input DataMemRW;
        output reg [31:0] o_data;
        reg [7:0] memory [0:63];
        initial begin
            o_data = 0;
        end

        always @(addr or i_data or DataMemRW) begin
            if (DataMemRW) begin
                memory[addr] = i_data[31:24];
                memory[addr+1] = i_data[23:16];
                memory[addr+2] = i_data[15:8];
```

```
            memory[addr+3] = i_data[7:0];
        end
        else begin
          o_data[31:24] = memory[addr];
          o_data[23:16] = memory[addr+1];
          o_data[15:8] = memory[addr+2];
          o_data[7:0] = memory[addr+3];
        end
    end

endmodule
```

3.5 寄存器设计：

多周期 CPU 核具有一个由 32 个 32 位寄存器构成的寄存器组。初始化 0 号寄存器的值为 0。

- **寄存器结构：**



- **寄存器描述：**

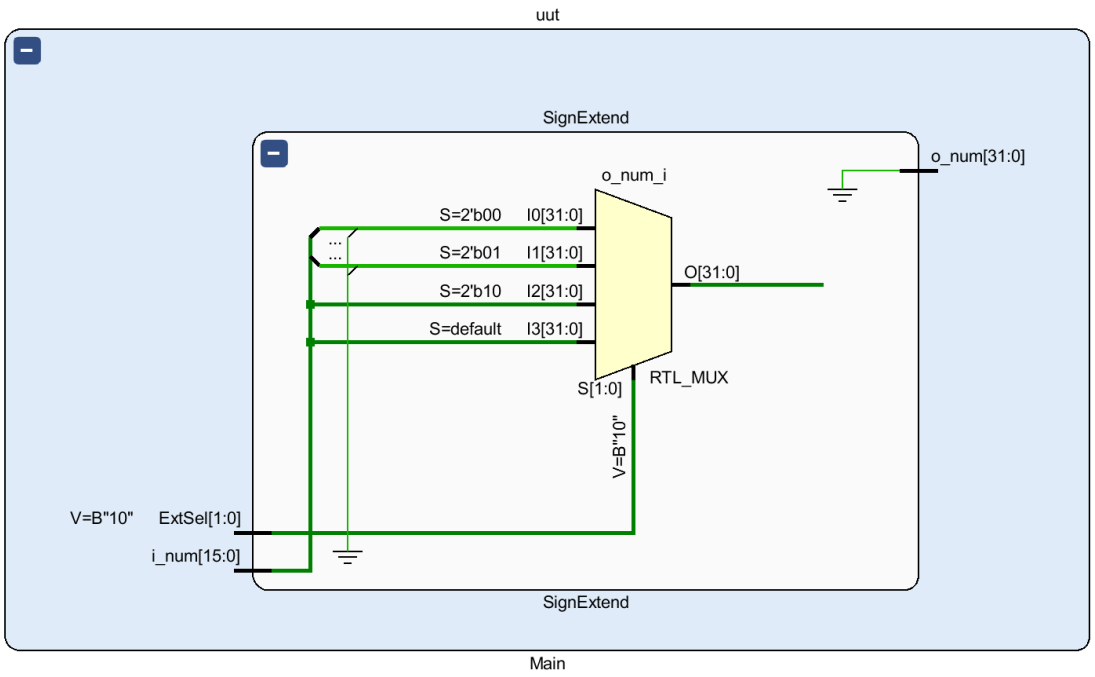
```
`timescale 1ns / 1ps

module RegFile (rs, rt, rd, i_data, RegWre, clk, o_data_1,
o_data_2);
    input [4:0] rs, rt, rd;
    input [31:0] i_data;
    input RegWre, clk;
    output [31:0] o_data_1, o_data_2;
    reg [31:0] register [0:31];

    integer i;
    initial begin
```

```
            i = 1;
            register[0]=0;
            while (i<32) begin
                register[i]=1;
                i = i+1;
            end
        end

        assign o_data_1 = register[rs];
        assign o_data_2 = register[rt];

        always @(i_data or rd) begin
            if ((rd != 0) && (RegWre == 1)) begin
                register[rd] = i_data;
            end
        end

    endmodule
```

3.6 位扩展器设计：

多周期 CPU 核具有一个位扩展器，可以根据控制器给出的控制信号进行零扩展、符号位扩展等操作。本 CPU 核所实现指令中，只有*BEQ*指令涉及位扩展器有关操作，且为立即数符号位扩展。

- **位扩展器结构：**



- **位扩展器描述：**

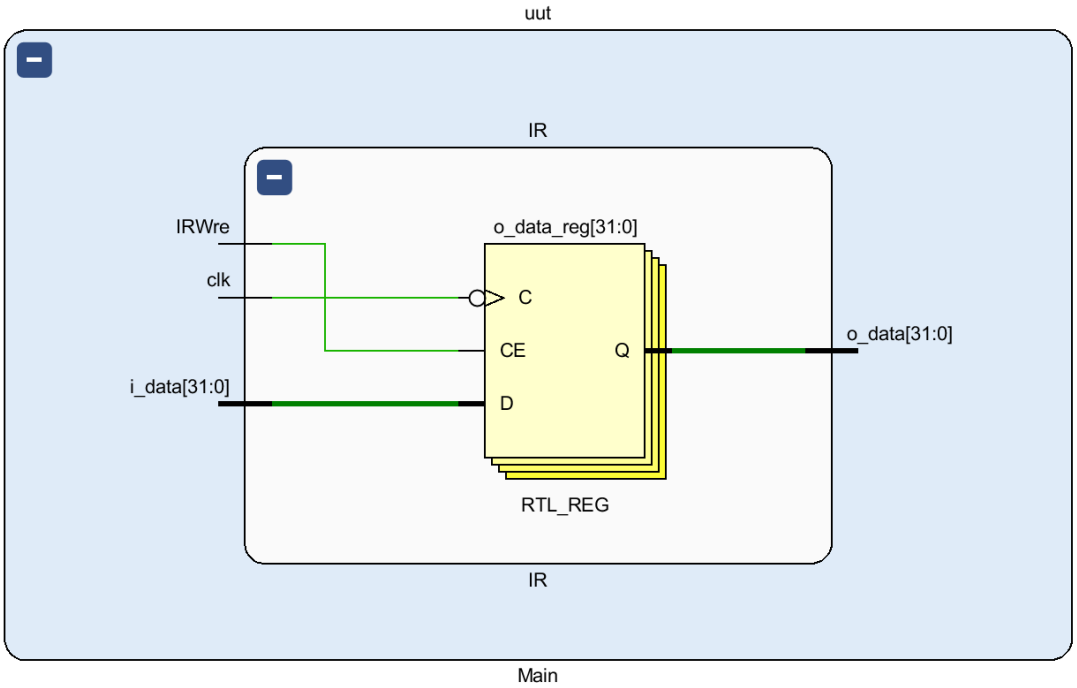
```
`timescale 1ns / 1ps
```

```
module SignExtend(i_num, ExtSel, o_num);
    input [15:0] i_num;
    input [1:0] ExtSel;
    output reg[31:0] o_num;
    initial begin
        o_num = 0;
    end
    always @(i_num or ExtSel) begin
        case(ExtSel)
            2'b00: o_num <= {{27'o000000000}, i_num[10:6]};
            2'b01: o_num <= {{16'h0000}, i_num[15:0]};
            2'b10: o_num <= {{16{i_num[15]}}, i_num[15:0]};
            default: o_num <= {{16{i_num[15]}},
i_num[15:0]};
        endcase
    end
endmodule
```

3.7 指令寄存器设计：

多周期 CPU 核具有一个指令寄存器，用于临时存储指令供下一阶段使用。

- **指令寄存器结构：**



- **指令寄存器描述：**

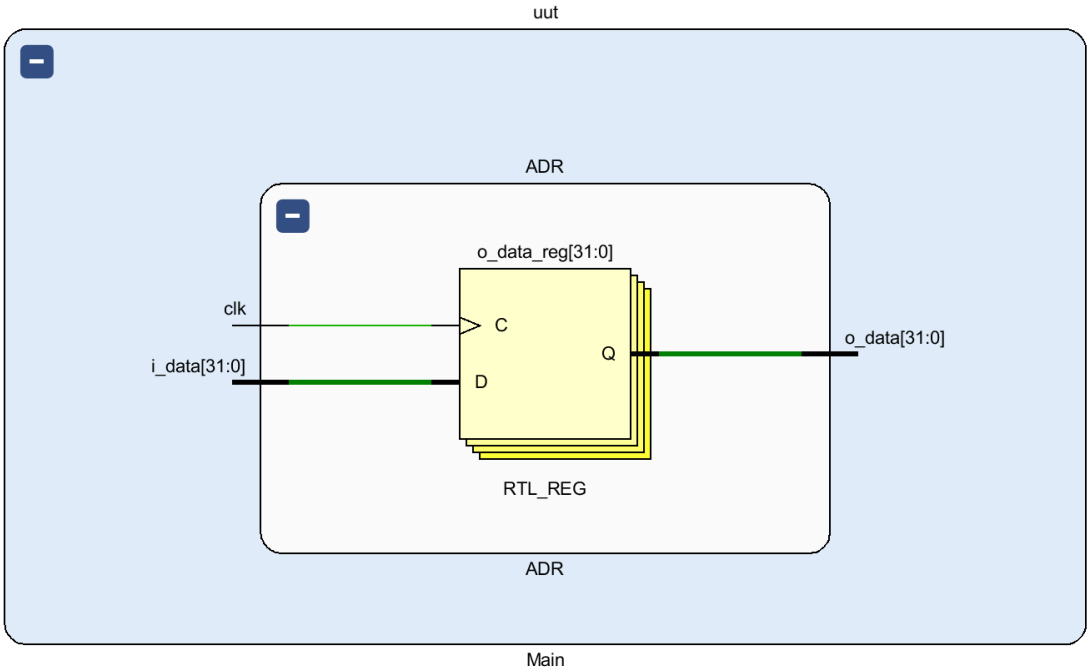
```
`timescale 1ns / 1ps

module IR(i_data, clk, IRWre, o_data);
```

```
    input clk, IRWre;
    input [31:0] i_data;
    output reg[31:0] o_data;

    always @(negedge clk) begin
        if (IRWre) begin
            o_data = i_data;
        end
    end

endmodule
```

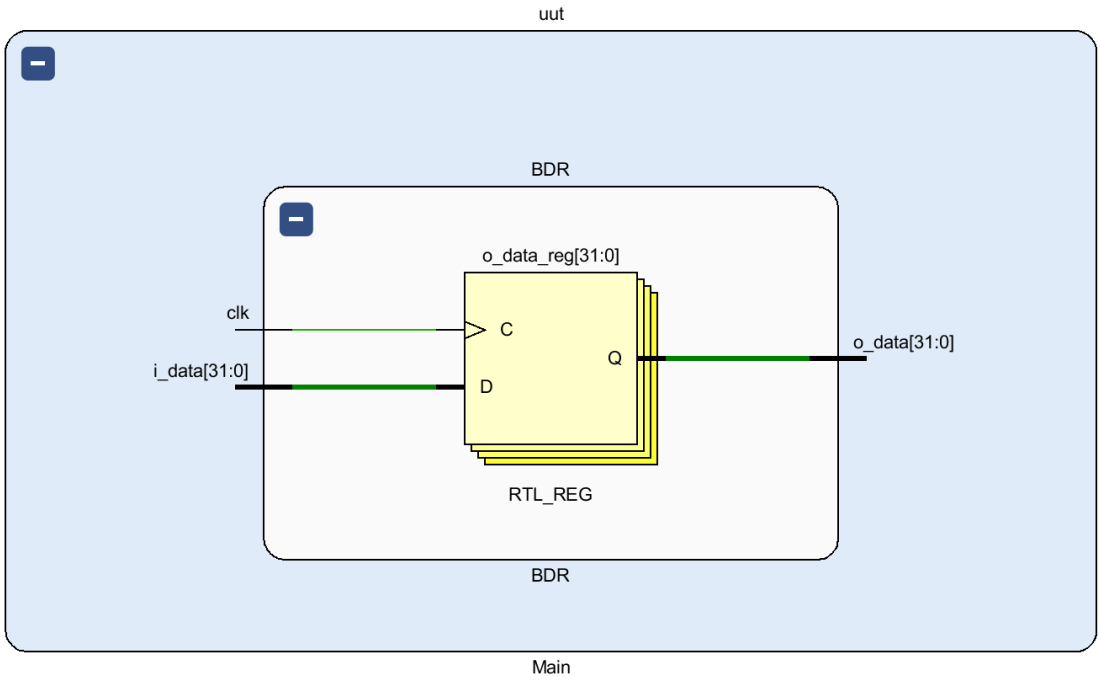
3.8 操作数寄存器设计：

多周期 CPU 核具有两个操作数寄存器，用于临时存储源操作数供下一阶段使用。

- 操作数寄存器结构（ADR）：



- 操作数寄存器结构（BDR）：



- 操作数寄存器描述（ADR）：

```
`timescale 1ns / 1ps

module ADR(i_data, clk, o_data);
    input clk;
    input [31:0] i_data;
    output reg[31:0] o_data;
    always @(posedge clk) begin
        o_data = i_data;
    end
endmodule
```

- 操作数寄存器描述（BDR）：

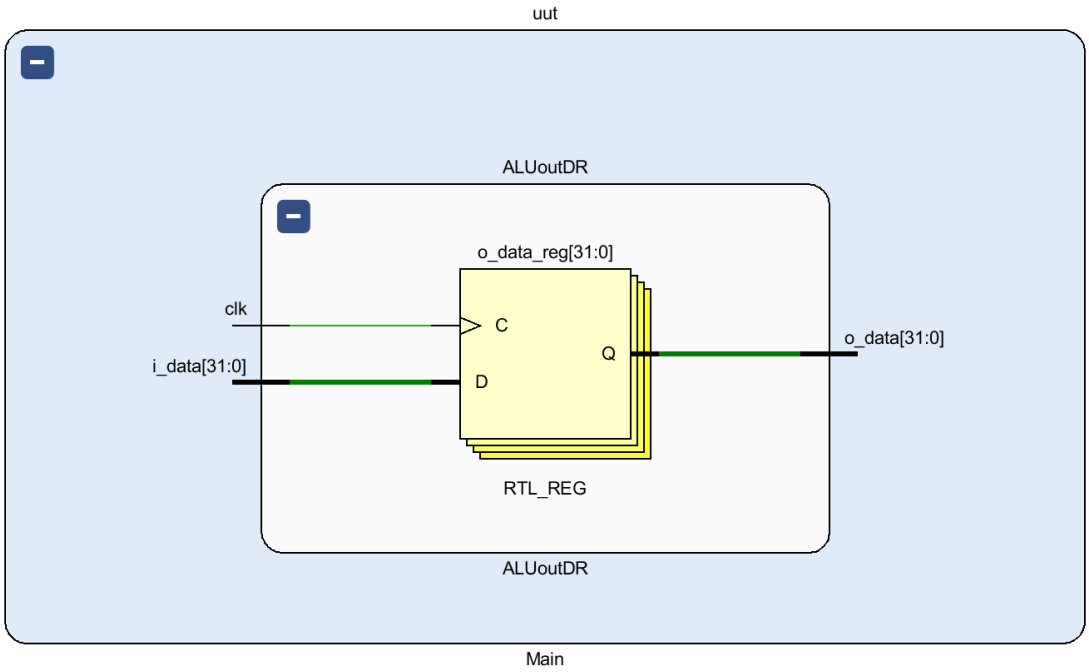
```
`timescale 1ns / 1ps

module BDR(i_data, clk, o_data);
    input clk;
    input [31:0] i_data;
    output reg[31:0] o_data;
    always @(posedge clk) begin
        o_data = i_data;
    end
endmodule
```

3.9 结果寄存器设计：

多周期 CPU 核具有一个结果寄存器，用于临时存储 ALU 运算结果供下

一阶段使用。

- **结果寄存器结构：**



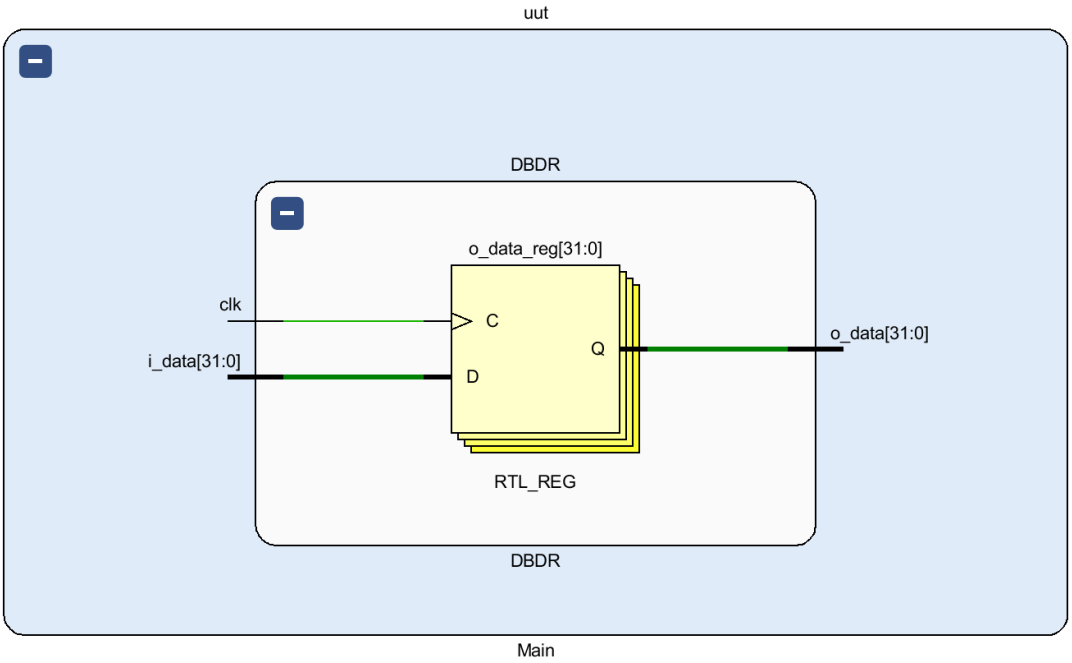
- **结果寄存器描述：**

```
`timescale 1ns / 1ps

module ALUoutDR(i_data, clk, o_data);
    input clk;
    input [31:0] i_data;
    output reg[31:0] o_data;
    always @(posedge clk) begin
        o_data = i_data;
    end
endmodule
```

3.10 数据寄存器设计：

多周期 CPU 核具有一个数据寄存器，用于临时存储 ALU 运算结果或数据存储器访存结果供下一阶段使用。

- **数据寄存器结构：**

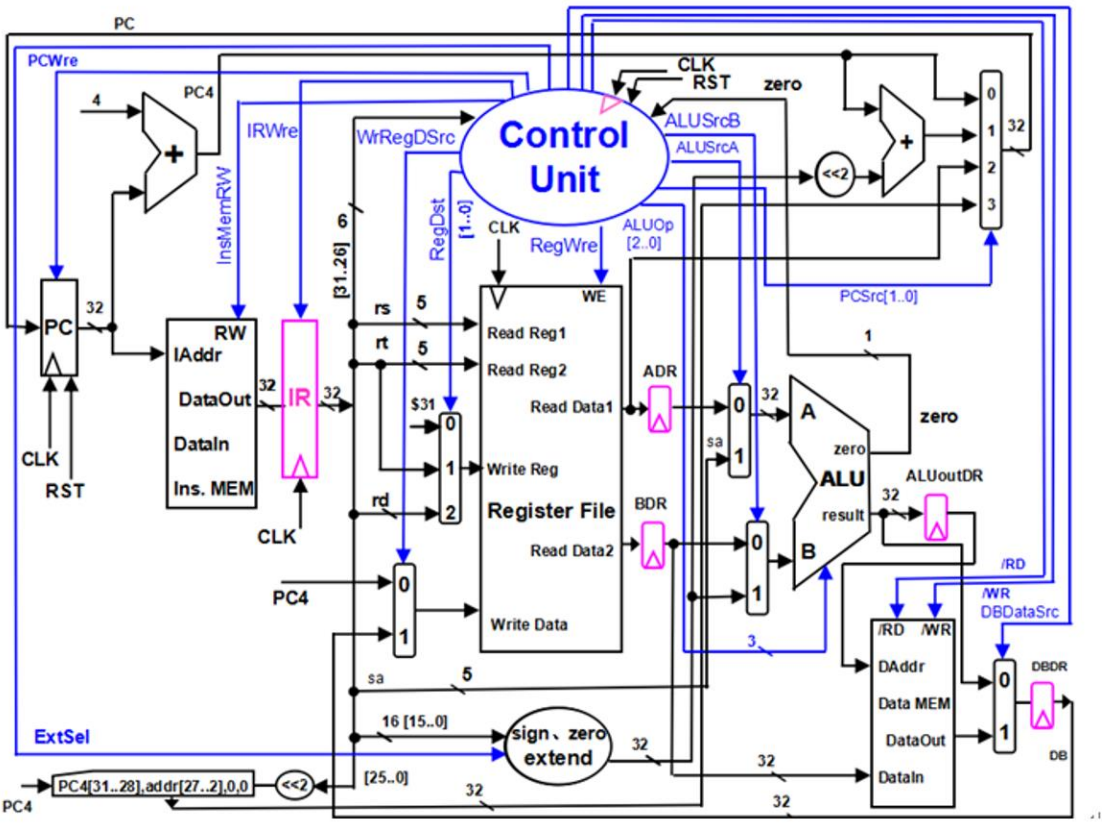


- **数据寄存器描述：**

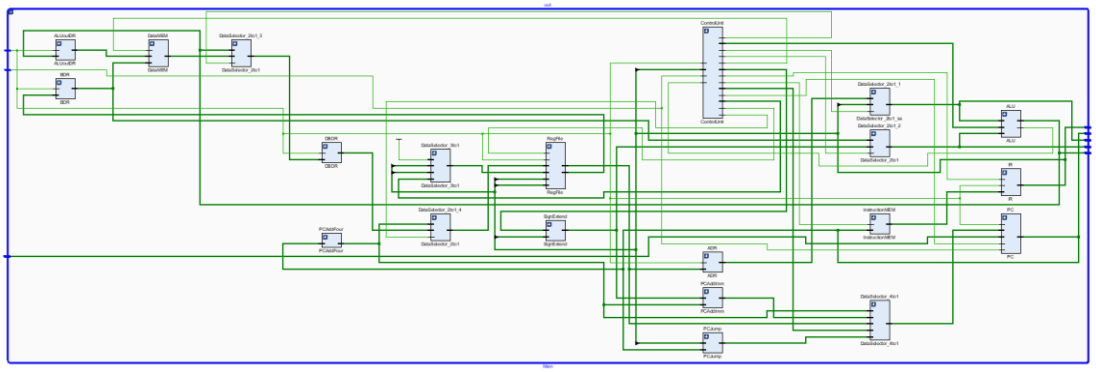
```
`timescale 1ns / 1ps

module DBDR(i_data, clk, o_data);
    input clk;
    input [31:0] i_data;
    output reg[31:0] o_data;

    always @(posedge clk) begin
        o_data = i_data;
    end

endmodule
```

4. 数据通路

多周期 CPU 核数据通路如下：



Vivado Schematic：



三、 仿真测试

1. 测试指令：

为了对 CPU 运行状态进行测试，我们编写测试指令如下，覆盖 7 条 MIPS 指令：

```
j 2 //J 型：跳转指令
111000 00000000000000000000000010
E000 0002

\\空指令：J 指令跳转测试
```

00000000000000000000000000000000

sw $1 0($0) \\I 型：存数指令
110000 00000 00001 0000000000000000
C001 0000

sw $2 4($0) \\I 型：存数指令
110000 00000 00010 0000000000000100
C002 0004

add $1 $1 $2 \\R 型：求和指令
000000 00001 00010 00001 00000 000000
0022 0800

add $1 $1 $2 \\R 型：求和指令
000000 00001 00010 00001 00000 000000
0022 0800

sub $2 $1 $2 \\R 型：求差指令
000001 00001 00010 00010 00000 000000
0422 1000

or $2 $1 $2 \\R 型：按位或指令
010000 00001 00010 00010 00000 000000
4022 1000

lw $1 0($0) \\I 型：取数指令
110001 00000 00001 0000000000000000
C401 0000

lw $2 4($0) \\I 型：取数指令
110001 00000 00010 0000000000000100
C402 0004

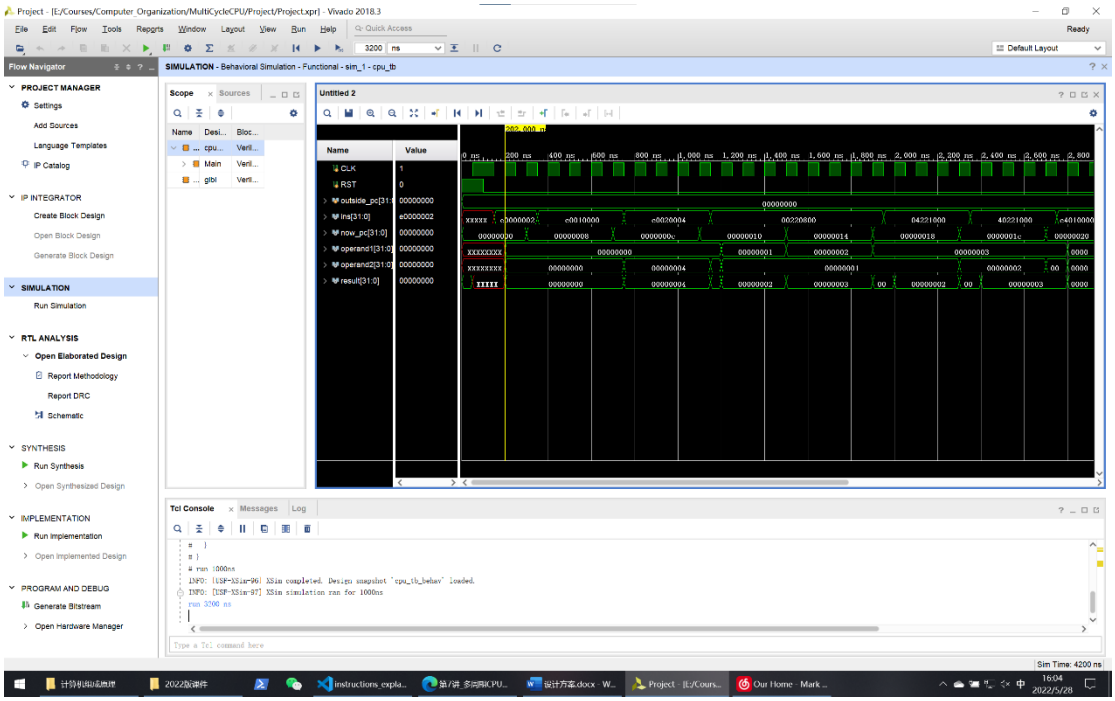
beq $1 $2 1\\I 型：条件判断指令
110100 00001 00010 0000000000000001
D022 0001

\\空指令：beq 指令跳转测试
00000000000000000000000000000000

halt \\停机指令
111111 00000000000000000000000000
FC00 0000

2. 仿真结果：

从下面的仿真结果可以看出，指令正常读取执行，CPU 核正常运行：



四、 性能指标分析

- 支持 7 条指令，多周期 CPI=3.9；
- 时钟周期：100ns；
- 等效频率：10MHz；
- 多周期 CPU 中各部件的利用率依然偏低。

五、 总结

通过本次课程设计任务，我们学习了多周期 CPU 的基本设计原理，对于 CPU 的构成有了更加深入的了解。

在设计数据通路的过程中，我们学习了各类型 MIPS 指令的执行过程，让我们对于精简指令集有了更加充分的认识，对精简指令集的优势有了更加直观的体会。

在编写测试指令的过程中，我们学习了有关 MIPS 指令集的基本汇编语言和各类型 MIPS 指令的算数逻辑功能、寄存器功能，掌握了有关 MIPS 指令集的基本知识。