组成原理实验课程第六次实验报告

实验名称：五级流水线　学生姓名：王众 学号：2313211

指导老师：董前琨　实验地点：实验楼A306 实验时间：2025.05.30

一、实验内容说明

请根据实验指导手册及source_code中的五级流水源码，分析现有的五级流水线存在的问题，并实现五级流水线CPU实验的bug修复，然后参考《CPU设计实战》这本书完成对五级流水线的指令扩展，满分100分，细分要求如下：

1、针对现有五级流水线存在问题的分析，不只是bug，包括指令相关、流水线冲突等各种能分析的问题进行分析（20分）

2、五级流水线指令运行时的bug修复（20分）

3、五级流水线指令扩展，运算类指令扩展至少一条（10分），乘除类指令至少一条（20分），转移指令至少一条（10分），访存指令一读一写两条（20分）

4、实验报告中针对bug修复过程、指令添加过程进行关键说明，并最终验证，验证需要有波形图或实验箱照片，并对波形图和实验箱照片进行分析解释。

5、实在无法完成某些指令扩展也没关系，把遇到的问题和失败的尝试写入报告，也会有相应分数。

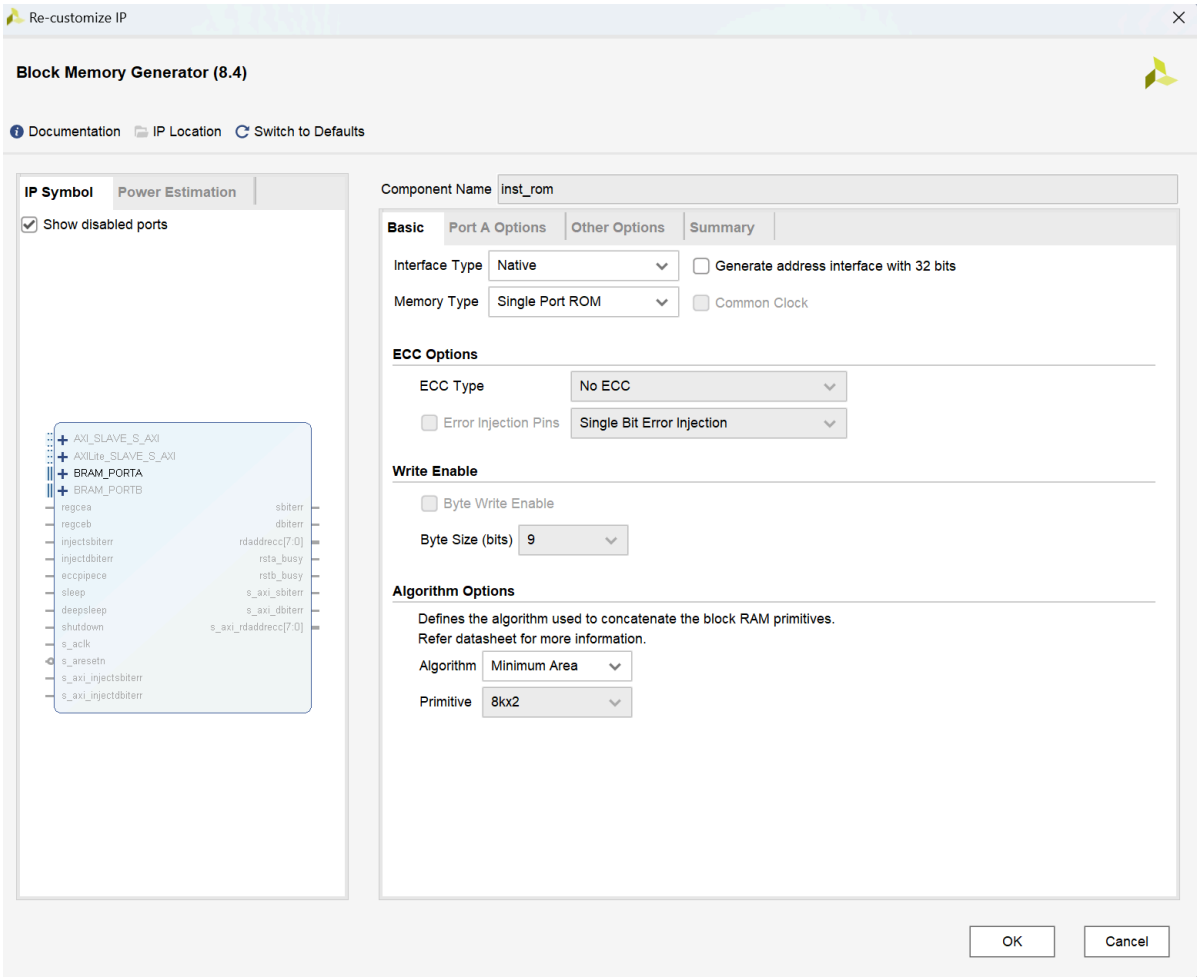
二、项目ip核配置

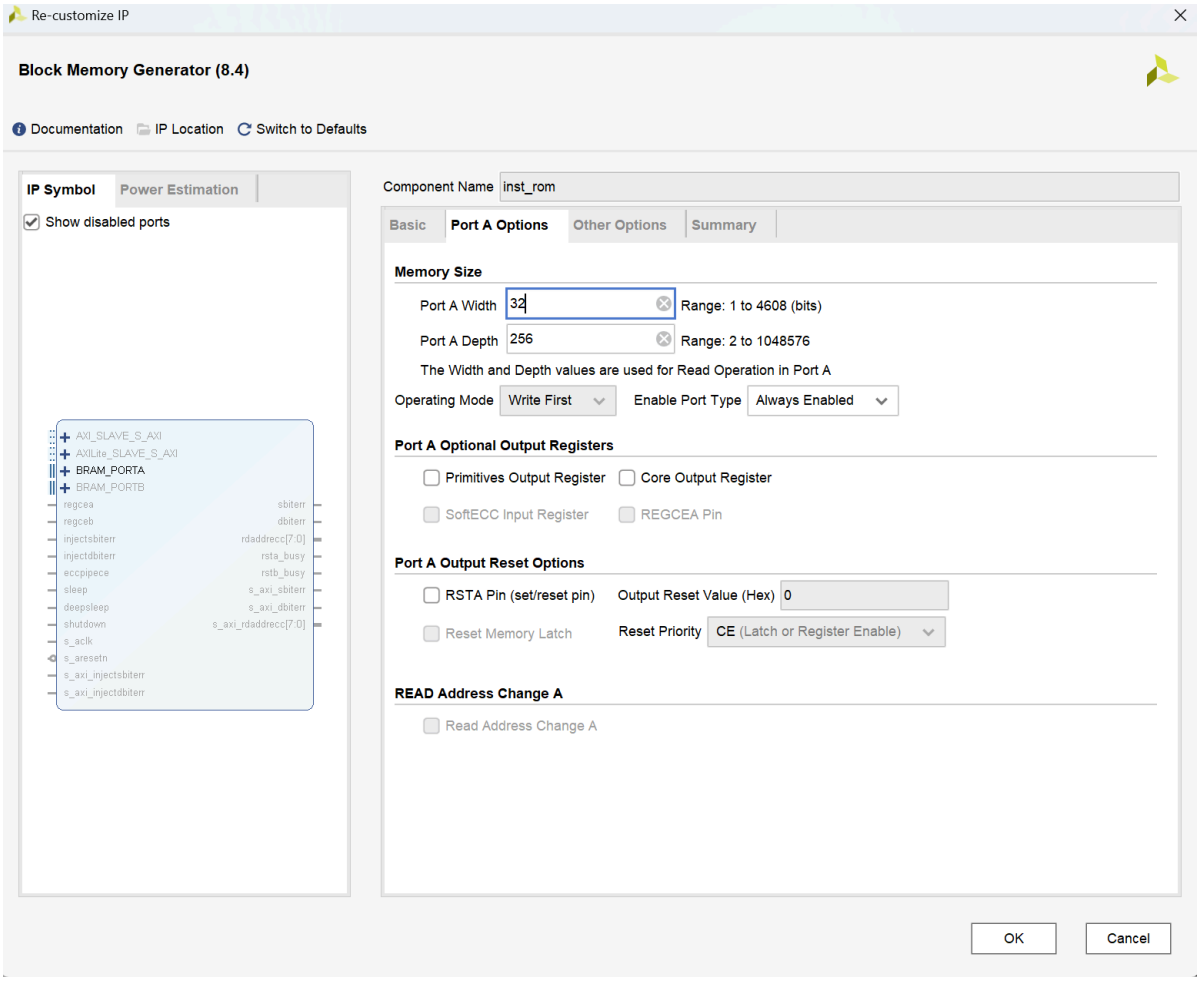
与上次多周期cpu实验相同，本次实验也需要我们自行配置ip核，分别是一个rom和一个ram。其配置流程如下：

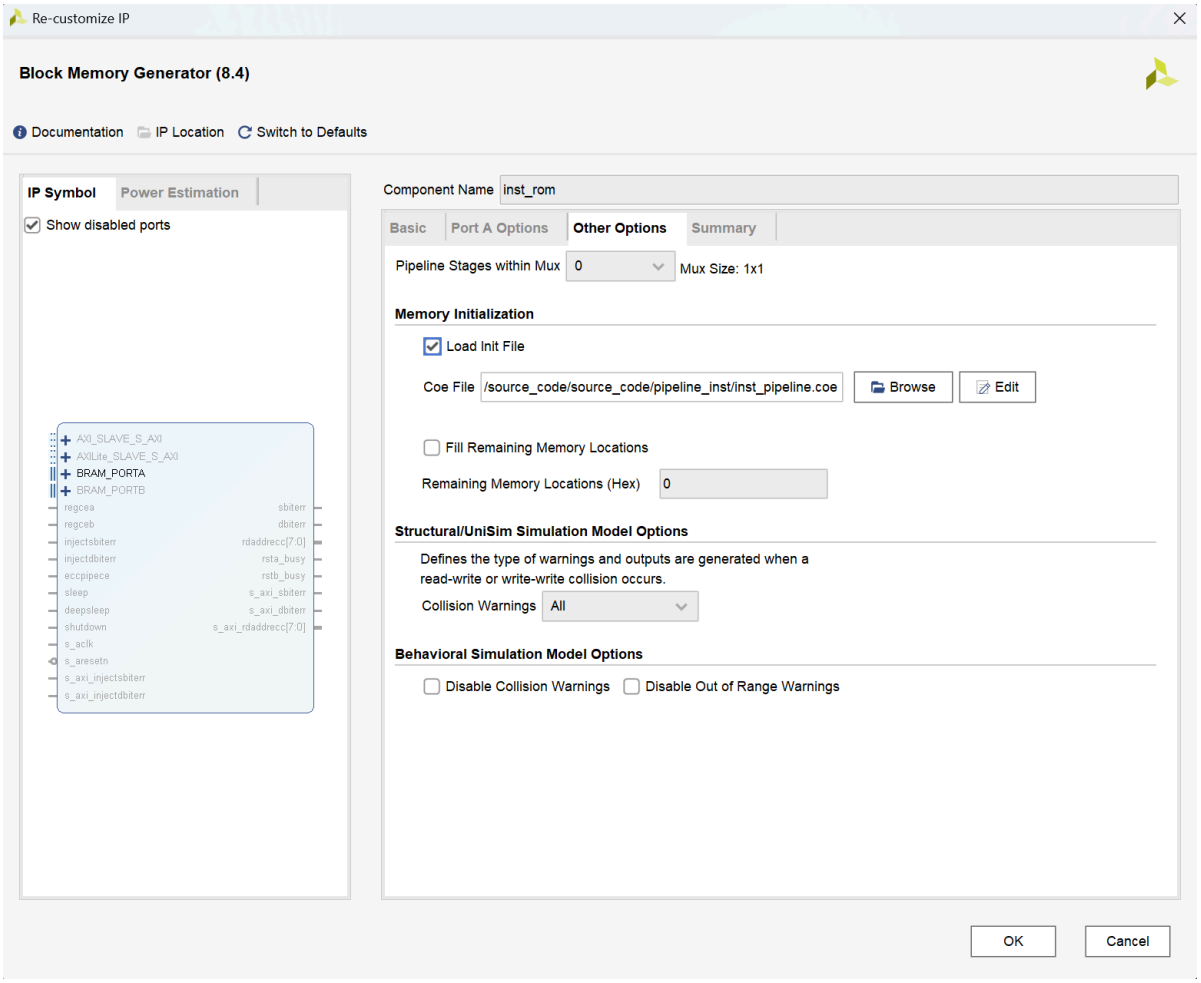
按照上述过程配置完毕ip核之后，运行仿真，查看波形，发现并没有bug的异常波形。于是我们推测：本次实验中五级流水线的源码里的bug和上次多周期cpu的bug一样，属于ip核配置的时候的配置bug，可能也是primitive选项，这里因为我们配置的时候就没有选择勾选该选项，因此避免了该bug的出现。

所以我们开始以以下配置开始配置IP核：

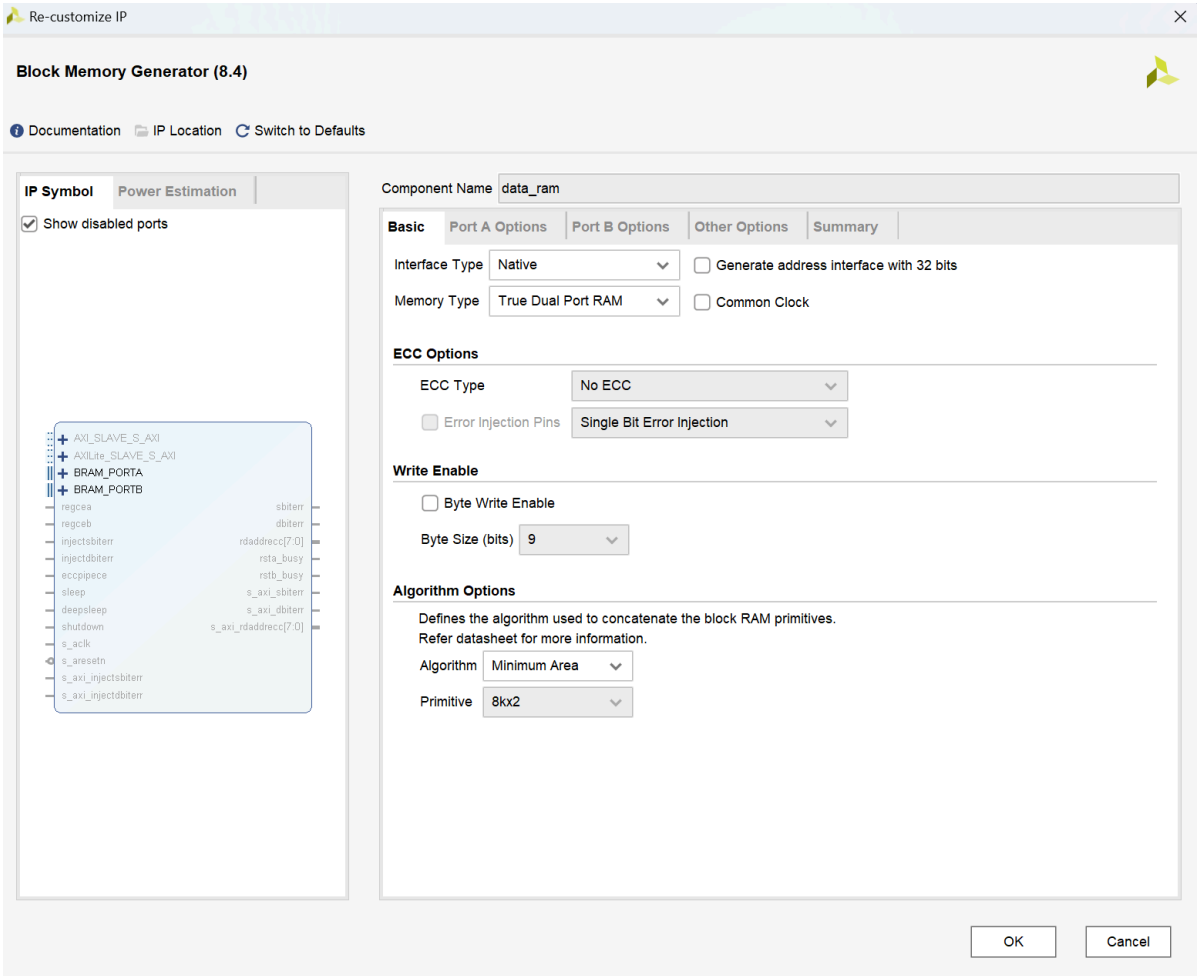
2.1 inst_rom

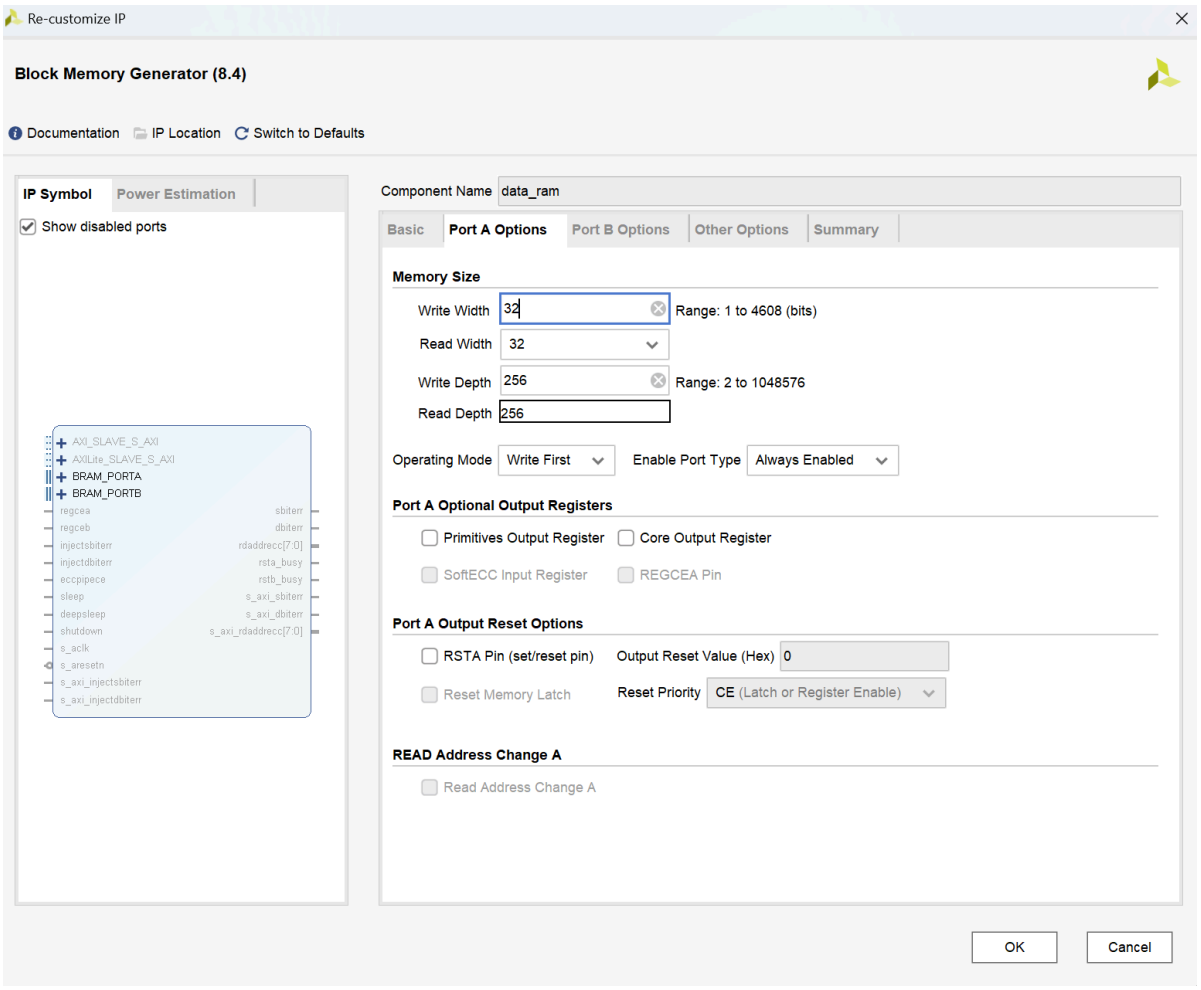

Re-customize IP
Block Memory Generator (8.4)
Documentation
IP Location
Switch to Defaults
IP Symbol
Power Estimation
Show disabled ports
AXI_SLAVE_S_AXI
AXILite_SLAVE_S_AXI
BRAM_PORTA
BRAM_PORTB
regcea
regceb
injectsbiterr
injectdbiterr
eccpipece
sleep
deepsleep
shutdown
s_aclk
s_aresetn
s_axi_injectsbiterr
s_axi_injectdbiterr
sbiterr
dbiterr
rdaddrecc[7:0]
rsta_busy
rstb_busy
s_axi_sbiterr
s_axi_dbiterr
s_axi_rdaddrecc[7:0]
Component Name
inst_rom
Basic
Port A Options
Other Options
Summary
Interface Type
Native
Generate address interface with 32 bits
Memory Type
Single Port ROM
Common Clock
ECC Options
ECC Type
No ECC
Error Injection Pins
Single Bit Error Injection
Write Enable
Byte Write Enable
Byte Size (bits)
9
Algorithm Options
Defines the algorithm used to concatenate the block RAM primitives.
Refer datasheet for more information.
Algorithm
Minimum Area
Primitive
8kx2
OK
Cancel

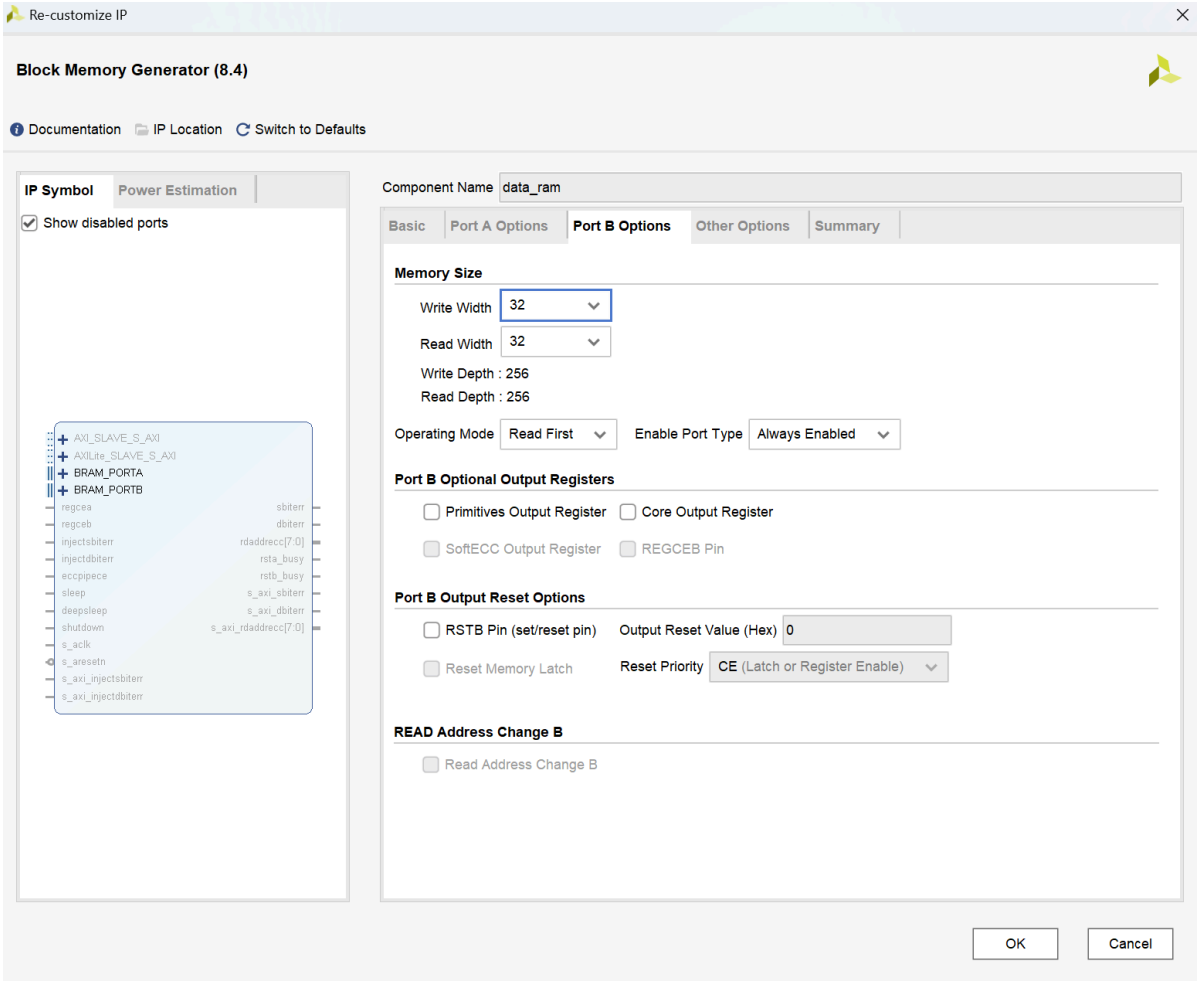

Re-customize IP
Block Memory Generator (8.4)
Documentation
IP Location
Switch to Defaults
IP Symbol
Power Estimation
Show disabled ports
AXI_SLAVE_S_AXI
AXILite_SLAVE_S_AXI
BRAM_PORTA
BRAM_PORTB
regcea
regceb
injectsbiterr
injectdbiterr
eccpipece
sleep
deepsleep
shutdown
s_aclk
s_aresetn
s_axi_injectsbiterr
s_axi_injectdbiterr
sbiterr
dbiterr
rdaddrecc[7:0]
rsta_busy
rstb_busy
s_axi_sbiterr
s_axi_dbiterr
s_axi_rdaddrecc[7:0]
Component Name
inst_rom
Basic
Port A Options
Other Options
Summary
Memory Size
Port A Width
32
Range: 1 to 4608 (bits)
Port A Depth
256
Range: 2 to 1048576
The Width and Depth values are used for Read Operation in Port A
Operating Mode
Write First
Enable Port Type
Always Enabled
Port A Optional Output Registers
Primitives Output Register
Core Output Register
SoftECC Input Register
REGCEA Pin
Port A Output Reset Options
RSTA Pin (set/reset pin)
Output Reset Value (Hex)
0
Reset Memory Latch
Reset Priority
CE (Latch or Register Enable)
READ Address Change A
Read Address Change A
OK
Cancel

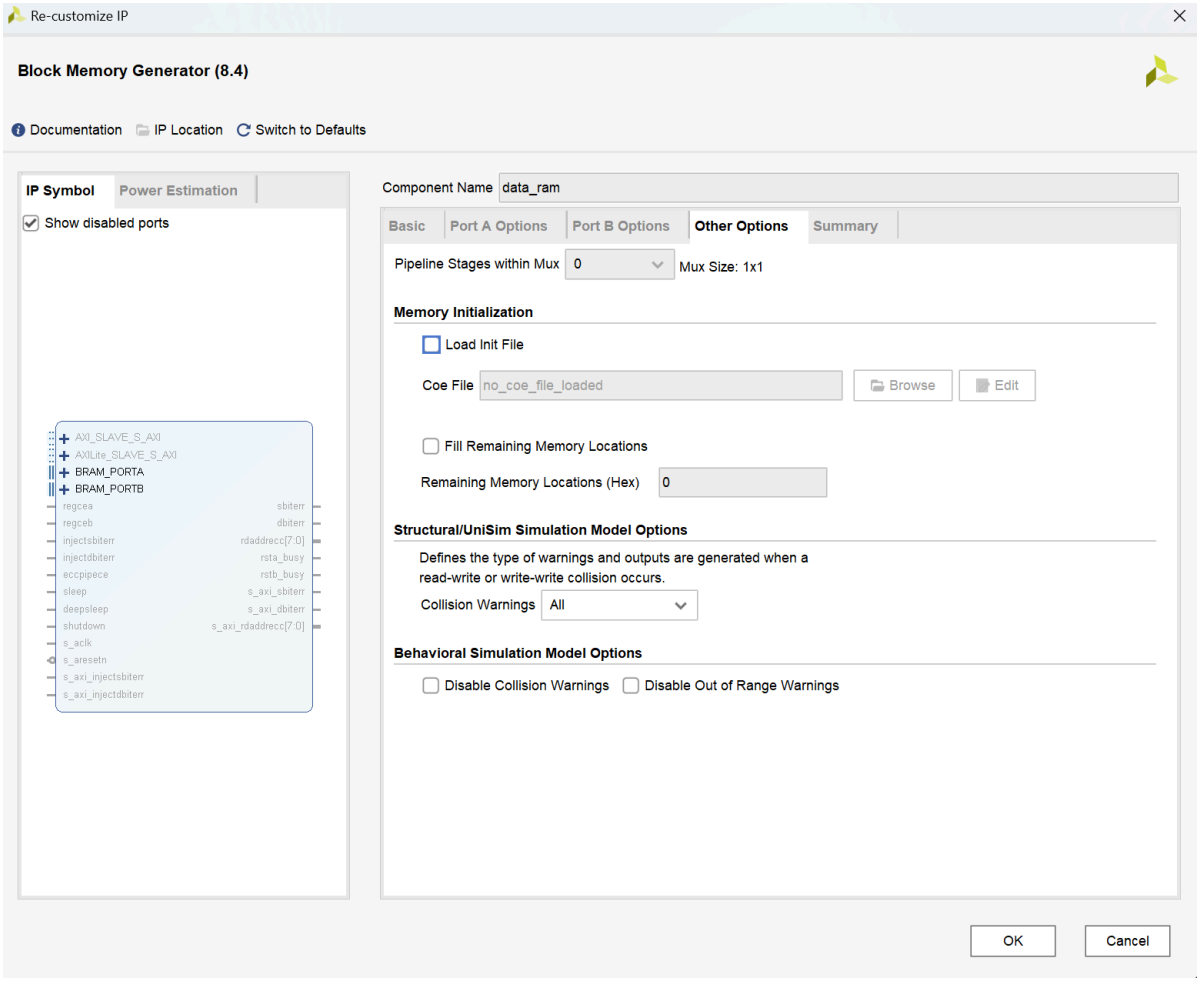

Re-customize IP
Block Memory Generator (8.4)
Documentation
IP Location
Switch to Defaults
IP Symbol
Power Estimation
Show disabled ports
AXI_SLAVE_S_AXI
AXILite_SLAVE_S_AXI
BRAM_PORTA
BRAM_PORTB
regcea
regceb
injectsbiterr
injectdbiterr
eccpipece
sleep
deepsleep
shutdown
s_aclk
s_aresetn
s_axi_injectsbiterr
s_axi_injectdbiterr
sbiterr
dbiterr
rdaddrecc[7:0]
rsta_busy
rstb_busy
s_axi_sbiterr
s_axi_dbiterr
s_axi_rdaddrecc[7:0]
Component Name
inst_rom
Basic
Port A Options
Other Options
Summary
Pipeline Stages within Mux
0
Mux Size: 1x1
Memory Initialization
Load Init File
Coe File
/source_code/source_code/pipeline_inst/inst_pipeline.coe
Browse
Edit
Fill Remaining Memory Locations
Remaining Memory Locations (Hex)
0
Structural/UniSim Simulation Model Options
Defines the type of warnings and outputs are generated when a read-write or write-write collision occurs.
Collision Warnings
All
Behavioral Simulation Model Options
Disable Collision Warnings
Disable Out of Range Warnings
OK
Cancel

2.2 data_ram

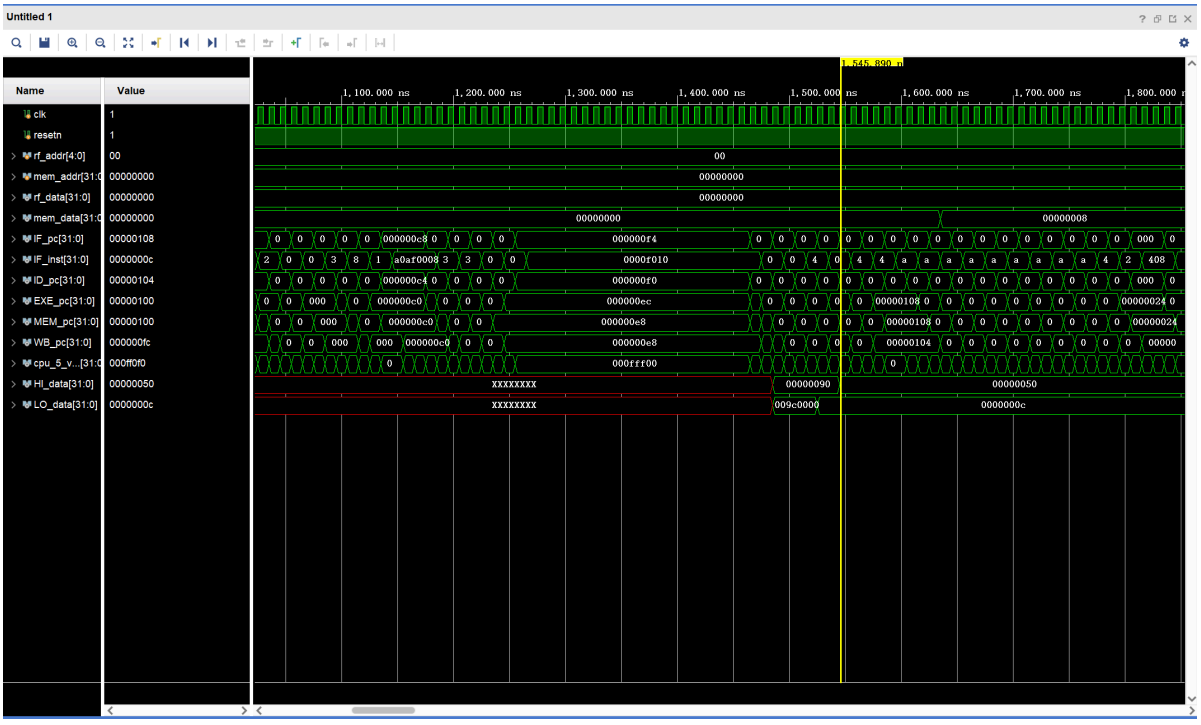

Re-customize IP
Block Memory Generator (8.4)
Documentation
IP Location
Switch to Defaults
IP Symbol
Power Estimation
Show disabled ports
AXI_SLAVE_S_AXI
AXILite_SLAVE_S_AXI
BRAM_PORTA
BRAM_PORTB
regcea
regceb
injectsbiterr
injectdbiterr
eccpipece
sleep
deepsleep
shutdown
s_aclk
s_aresetn
s_axi_injectsbiterr
s_axi_injectdbiterr
sbiterr
dbiterr
rdaddrecc[7:0]
rsta_busy
rstb_busy
s_axi_sbiterr
s_axi_dbiterr
s_axi_rdaddrecc[7:0]
Component Name
data_ram
Basic
Port A Options
Port B Options
Other Options
Summary
Interface Type
Native
Generate address interface with 32 bits
Memory Type
True Dual Port RAM
Common Clock
ECC Options
ECC Type
No ECC
Error Injection Pins
Single Bit Error Injection
Write Enable
Byte Write Enable
Byte Size (bits)
9
Algorithm Options
Defines the algorithm used to concatenate the block RAM primitives.
Refer datasheet for more information.
Algorithm
Minimum Area
Primitive
8kx2
OK
Cancel


Re-customize IP
Block Memory Generator (8.4)
Documentation
IP Location
Switch to Defaults
IP Symbol
Power Estimation
Show disabled ports
AXI_SLAVE_S_AXI
AXILite_SLAVE_S_AXI
BRAM_PORTA
BRAM_PORTB
regcea
regceb
injectsbiterr
injectdbiterr
eccpipece
sleep
deepsleep
shutdown
s_aclk
s_aresetn
s_axi_injectsbiterr
s_axi_injectdbiterr
sbiterr
dbiterr
rdaddrecc[7:0]
rsta_busy
rstb_busy
s_axi_sbiterr
s_axi_dbiterr
s_axi_rdaddrecc[7:0]
Component Name
data_ram
Basic
Port A Options
Port B Options
Other Options
Summary
Memory Size
Write Width
32
Range: 1 to 4608 (bits)
Read Width
32
Write Depth
256
Range: 2 to 1048576
Read Depth
256
Operating Mode
Write First
Enable Port Type
Always Enabled
Port A Optional Output Registers
Primitives Output Register
Core Output Register
SoftECC Input Register
REGCEA Pin
Port A Output Reset Options
RSTA Pin (set/reset pin)
Output Reset Value (Hex)
0
Reset Memory Latch
Reset Priority
CE (Latch or Register Enable)
READ Address Change A
Read Address Change A
OK
Cancel


Re-customize IP
Block Memory Generator (8.4)
Documentation
IP Location
Switch to Defaults
IP Symbol
Power Estimation
Show disabled ports
AXI_SLAVE_S_AXI
AXILite_SLAVE_S_AXI
BRAM_PORTA
BRAM_PORTB
regcea
regceb
injectsbiterr
injectdbiterr
eccpipece
sleep
deepsleep
shutdown
s_aclk
s_aresetn
s_axi_injectsbiterr
s_axi_injectdbiterr
sbiterr
dbiterr
rdaddrecc[7:0]
rsta_busy
rstb_busy
s_axi_sbiterr
s_axi_dbiterr
s_axi_rdaddrecc[7:0]
Component Name
data_ram
Basic
Port A Options
Port B Options
Other Options
Summary
Memory Size
Write Width
32
Read Width
32
Write Depth : 256
Read Depth : 256
Operating Mode
Read First
Enable Port Type
Always Enabled
Port B Optional Output Registers
Primitives Output Register
Core Output Register
SoftECC Output Register
REGCEB Pin
Port B Output Reset Options
RSTB Pin (set/reset pin)
Output Reset Value (Hex)
0
Reset Memory Latch
Reset Priority
CE (Latch or Register Enable)
READ Address Change B
Read Address Change B
OK
Cancel



2.3仿真结果局部展示



我们可以看到，各项指令都在正常进行。配置好ip核之后，接下来就可以着手添加指令了。

三、指令的添加过程

3.1 运算类指令

我们检查了源码已实现的指令之后，发现运算类指令中，缺少有符号加法ADD，因此这里选择增加有符号加法ADD

decode.v

先在指令声明部分添加ADD

```
 wire inst_ADD;
```

然后编写对应的编码设定

```
assign inst_ADD = op_zero & sa_zero & (funct == 6'b011110);
```

经过检查，发现现有指令中还没有funct == 6'b011110的指令，因此这里将其设置为6'b011110

接下来，在add类指令的归类中加上新添加的ADD

```
assign inst_add = inst_ADDU | inst_ADDIU | inst_load
                    | inst_store | inst_j_link | inst_ADD;
```

最后，ADD指令也是R型指令，也要写回rd寄存器，因此在写回rd寄存器的标志信号中添加该指令信号

```
assign inst_wdest_rd = inst_ADDU | inst_SUBU | inst_SLT  | inst_SLTU
                         | inst_JALR | inst_AND  | inst_NOR  | inst_OR
                            | inst_XOR  | inst_SLL  | inst_SLLV | inst_SRA
                         | inst_SRAV | inst_SRL  | inst_SRLV
                         | inst_MFHI | inst_MFLO | inst_ADD;
```

这样，ADD的指令添加就完成了。

3.2 乘除类指令

检查已有指令，发现目前为止还没有无符号乘法，因此决定在乘除类指令中增加无符号乘法MULTU

decode.v

先增加MULTU的指令声明

```
wire inst_MULTU; // 添加无符号乘法声明
```

然后设计指令编码

```
assign inst_MULTU = op_zero & (rd==5'd0) & sa_zero & (funct == 6'b011001); //无符号乘法
```

接下来，我们发现有一个multiply信号用于判断是否调用乘法器，现在我们需要把新加的这个无符号乘法指令加进去

```
assign multiply = inst_MULT | inst_MULTU;
```

另外，乘法器中目前实现了有符号乘法的逻辑，我们还需要增加一个标志信号，告诉乘法器它应该执行的是有符号乘法还是无符号乘法

```
 wire mul_sign;
 assign mul_sign = inst_MULT;
```

这里我们设定当有符号乘法的时候，mul_sign信号为1。

增加了这个信号之后，因为我们是在exe阶段才会实例化并调用乘法器multiply，因此需要将该信号传递到exe中，这里我们将该新增的信号加入总线中，不要忘了修改总线的位宽

```
output     [167:0] ID_EXE_bus, //原先167位位宽，现在168位
assign ID_EXE_bus = {mul_sign,multiply,mthi,mtlo,                    //这里将
mul_sign放在第一位，后续也需要保持顺序
                        alu_control,alu_operand1,alu_operand2,
                        mem_control,store_data,
                        mfhi,mflo,
                        mtc0,mfc0,cp0r_addr,syscall,eret,
                        rf_wen, rf_wdest,
                        pc};
```

multiply.v

接下来，我们需要修改乘法器的逻辑，让他也能支持无符号乘法

先增加信号变量

```
module multiply(
    input         clk,
    input         mult_begin,
    input  [31:0] mult_op1,
    input  [31:0] mult_op2,
    output [63:0] product,
    output        mult_end,
    input         mul_sign // 新增加的信号变量，判断是有符号乘法还是无符号乘法
);
```

接着，符号检测逻辑也需要修改，只有在有符号乘法的时候才检测符号位

```
assign op1_sign = mult_sign & mult_op1[31]; // 只有在有符号乘法时才检查符号位
assign op2_sign = mult_sign & mult_op2[31];

// 修改最终结果符号处理
assign product = (mult_sign & product_sign) ? (~product_temp+1) : product_temp;
```

exe.v

同样，在总线传输的逻辑中增加标志信号mul_sign

这里先修改总线传输信号的位宽

```
module exe(
    input               EXE_valid,
    input      [167:0] ID_EXE_bus_r,// 增加位宽
    output              EXE_over,
    output     [153:0] EXE_MEM_bus,


     input              clk,
     output     [  4:0] EXE_wdest,

    //ڣ'PC
    output     [ 31:0] EXE_pc
);
```

然后修改总线解包逻辑

```
assign {mul_sign,
            multiply,
            mthi,
            mtlo,
            alu_control,
            alu_operand1,
            alu_operand2,
            mem_control,
            store_data,
            mfhi,
            mflo,
            mtc0,
            mfc0,
            cp0r_addr,
            syscall,
            eret,
            rf_wen,
            rf_wdest,
            pc          } = ID_EXE_bus_r;
```

修改乘法器实例化代码

```
multiply multiply_module (
        .clk       (clk       ),
        .mult_begin(mult_begin  ),
        .mult_op1  (alu_operand1),
        .mult_op2  (alu_operand2),
        .product   (product   ),
        .mult_end  (mult_end  ),
        .mul_sign  (mul_sign)
    );
```

至此，无符号乘法的指令也添加完毕了。需要注意的是，这里的无符号乘法是64位乘法，因此乘法的结果会被存储到HI、LO两个寄存器里，而不是rd，因此判断是否需要写回rd的信号中并不需要添加MULTU，同时，在编写MULTU指令的机器码的时候，不仅仅shamt字段，rd字段也应该为全零。

3.3 转移指令

这里我们选择添加BLTZAL指令，它的功能是：`bltzal rs, offset`，如果rs寄存器里存储的字段的值小于0的话，就跳转到目标地址。

decode.v

先增加指令声明

```
wire inst_BLTZAL;
```

然后编写指令编码

```
assign inst_BLTZAL = (op == 6'b000001) & (rt == 5'b10001); // 小于0分支并链接
```

在编写此处指令编码的时候，通过询问大模型，我得知这样的编码方式是mips指令规定的，因此直接沿用

接下来，修改分支指令判断信号，加上我们新增的BLTZAL

```
assign inst_jbr = inst_J    | inst_JAL  | inst_jr
                | inst_BEQ  | inst_BNE  | inst_BGEZ
                | inst_BGTZ | inst_BLEZ | inst_BLTZ | inst_BLTZAL;
```

然后，修改链接指令的判断信号，增加我们的新指令

```
assign inst_j_link = inst_JAL | inst_JALR | inst_BLTZAL;
```

不要忘记修改分支逻辑

```
/ 修改分支判断逻辑
assign br_taken = inst_BEQ    & rs_equql_rt       // 相等跳转
                | inst_BNE    & ~rs_equql_rt      // 不等跳转
                | inst_BGEZ   & ~rs_ltz           // 大于等于0跳转
                | inst_BGTZ   & ~rs_ltz & ~rs_ez  // 大于0跳转
                | inst_BLEZ   & (rs_ltz | rs_ez)  // 小于等于0跳转
                | inst_BLTZ   & rs_ltz            // 小于0跳转
                | inst_BLTZAL & rs_ltz;           // 小于0分支并链接
```

最后，修改一下写回寄存器逻辑即可

```
// 修改写回$31寄存器的指令
assign inst_wdest_31 = inst_JAL | inst_BLTZAL;
```

这里的逻辑是跳转的时候保存返回地址到$31

3.4 访存指令

3.4.1 读

这里我们选择添加的指令是取半字无符号扩展LHU。由于五级流水线CPU中已经实现了LBU，而这两者的逻辑非常近似，因此我们只需要参考它的逻辑来完成即可。

decode.v

先增加指令声明

```
wire inst_LHU;
```

增加指令编码

```
assign inst_LHU = (op == 6'b100101);
```

修改load指令的组合，加上新加的LHU指令

```
assign inst_load = inst_LW | inst_LB | inst_LBU | inst_LHU;
```

我们这个指令理论上是需要写回rt寄存器的，但是写回rt信号中已经有了inst_load，因此我们不需要再多修改了

另外，我们还需要增加一个lh_sign信号来标注是否需要无符号扩展

```
wire lh_sign;  // load半字的符号扩展标志
assign lh_sign = 1'b0;  //这里，其实是应该用！LHU的逻辑的，但是我们没有加有符号半字扩展的打算，因此直接默认0即可
```

另外，这里会一直到mem阶段才用得上这个信号，所以需要一路把这个信号传到mem阶段。所以这里把这个信号加入`mem_control`里，同时，总线长度也需要修改。

```
wire [4:0] mem_control;
assign mem_control = {inst_load,
                      inst_store,
                      ls_word,
                      inst_LHU,    // 新增：标识是否为半字操作
                      lb_sign};
output     [168:0] ID_EXE_bus, //原先168位位宽，现在169位
```

exe.v

exe中，对应的总线位宽和控制信号位宽也需要做出更改

```
input      [168:0] ID_EXE_bus_r,
wire [4:0] mem_control;
output     [154:0] EXE_MEM_bus,
```

mem.v

这里面，首先需要更改总线和控制信号的位宽

```
input      [154:0] EXE_MEM_bus_r,
wire [4 :0] mem_control;
```

mem控制信号解包的逻辑也应该更改（这里解包的顺序记得遵循封装的顺序）

```
 wire inst_load;
    wire inst_store;
    wire ls_word;
    wire is_half;//新增的半字加载
    wire lb_sign;
    assign {inst_load,inst_store,ls_word,is_half,lb_sign} = mem_control;
```

然后，我们修改load_result的逻辑，增加半字数据逻辑

```
  // 需要添加半字数据选择
  wire [15:0] halfword_data;
  assign halfword_data = (dm_addr[1] == 1'b0) ? dm_rdata[15:0] : dm_rdata[31:16];

  // 修改高位处理逻辑
  assign load_result[31:8] = ls_word ? dm_rdata[31:8] :              // 字操作
                         is_half ? {16'd0, halfword_data[15:8]} :  // 半字操作（LHU
无符号扩展）
                         {24{lb_sign & load_sign}};               // 字节操作//----
-{load/store??}end
                         // 修改低8位处理（当是半字操作时）
  assign load_result[7:0] = is_half ? halfword_data[7:0] :           // 半字操作
                        ((dm_addr[1:0]==2'd0) ? dm_rdata[ 7:0 ] : // 字节操作
                         (dm_addr[1:0]==2'd1) ? dm_rdata[15:8 ] :
                         (dm_addr[1:0]==2'd2) ? dm_rdata[23:16] :
                                                dm_rdata[31:24]);
```

3.4.2 写

这里我们选择增加写入内存后四位的半位写入逻辑指令，它的功能是：

decode.v

先增加指令声明

```
wire inst_SWL;
```

增加指令编码

```
assign inst_SWL = (op == 6'b101010);
```

修改store指令组合，加上新加的SWL指令

```
 assign inst_store = inst_SW | inst_SB | inst_SWL;
```

另外，我们还需要增加一个信号来标注是否是SWL指令

```
wire swl_type;  // SWL操作标志
assign swl_type = inst_SWL;
```

同理，一直到mem阶段才用得上这个信号，所以需要一路把这个信号传到mem阶段。所以这里把这个信号加入`mem_control`里，同时，总线长度也需要修改。

```
wire [5:0] mem_control;
assign mem_control = {inst_load,
                      inst_store,
                      ls_word,
                      inst_LHU,
                      swl_type,//新增
                      lb_sign};
output     [169:0] ID_EXE_bus,
```

exe.v

exe中，对应的总线位宽和控制信号位宽也需要做出更改

```
input      [169:0] ID_EXE_bus_r,
wire [5:0] mem_control;
output     [155:0] EXE_MEM_bus,
```

mem.v

更改总线和控制信号的位宽

```
input      [155:0] EXE_MEM_bus_r,
wire [5 :0] mem_control;
```

mem控制信号解包的逻辑

```
 wire inst_load;
    wire inst_store;
    wire ls_word;
    wire is_half;
        wire swl_type;
    wire lb_sign;
assign {inst_load,inst_store,ls_word,is_half,swl_type,lb_sign} = mem_control;
```

然后，修改写入逻辑

```
  always @ (*)
    begin
        if (swl_type) begin  // SWL指令特殊处理 - 修改为后四位
            case (dm_addr[1:0])
                2'b00   : dm_wdata = store_data;
                2'b01   : dm_wdata = {8'd0, store_data[31:8]};
                2'b10   : dm_wdata = {16'd0, store_data[31:16]};
                2'b11   : dm_wdata = {24'd0, store_data[31:24]};
                default : dm_wdata = store_data;
            endcase
        end
```

```
        else if (ls_word) begin  // SW指令直接存储整个32位数据
            dm_wdata = store_data;  // SW指令忽略地址偏移，直接存储整个数据
        end
        else begin  // SB指令按字节重组数据
            case (dm_addr[1:0])
                2'b00   : dm_wdata = store_data;                    // 保持原样
                2'b01   : dm_wdata = {16'd0, store_data[7:0], 8'd0};
                2'b10   : dm_wdata = {8'd0, store_data[7:0], 16'd0};
                2'b11   : dm_wdata = {store_data[7:0], 24'd0};
                default : dm_wdata = store_data;
            endcase
        end
    end
```

修改写使能逻辑

```
    always @ (*)
    begin
        // 确保MEM_valid稳定且为store指令时才产生写使能
        if (MEM_valid && inst_store)
        begin
            if (swl_type) begin  // SWL指令的写使能 - 修改为后四位
                case (dm_addr[1:0])
                    2'b00   : dm_wen = 4'b1111;  // 写全部4字节
                    2'b01   : dm_wen = 4'b0111;  // 写低3字节
                    2'b10   : dm_wen = 4'b0011;  // 写低2字节
                    2'b11   : dm_wen = 4'b0001;  // 只写最低字节
                    default : dm_wen = 4'b0000;
                endcase
            end
            else if (ls_word) begin
                dm_wen = 4'b1111; // SW指令写使能全1
            end
            else begin // SB指令原有逻辑
                case (dm_addr[1:0])
                    2'b00   : dm_wen = 4'b0001;
                    2'b01   : dm_wen = 4'b0010;
                    2'b10   : dm_wen = 4'b0100;
                    2'b11   : dm_wen = 4'b1000;
                    default : dm_wen = 4'b0000;
                endcase
            end
        end
        else begin
            dm_wen = 4'b0000;
        end
    end
```

另外，上述修改过程中忘记了cpu顶层模块的总线位宽，这里同样也需要修改。

```
    wire [169:0] ID_EXE_bus;
wire [155:0] EXE_MEM_bus;
reg [169:0] ID_EXE_bus_r;
reg [155:0] EXE_MEM_bus_r;
```

以上，我们就完成了作业要求加入的所有指令！

四、流水线冒险避免

在我们实现的功能中虽然他们能够再单独使用的情况下进行执行，但是当他们和起来的时候可能会出现冒险的情况，所以我们进一步改进，避免冒险。

4.1 加法

原本的流水线已经对R型指令可能出现的冒险采取了以下几种措施：

1. **冒险检测逻辑 (在 `decode.v` 中):**

   `rs_wait` 和 `rt_wait` 信号被用来检测当前指令的源寄存器 (`rs`, `rt`) 是否与后续流水线阶段 (EXE, MEM, WB) 中指令要写入的目标寄存器 (`EXE_wdest`, `MEM_wdest`, `WB_wdest`) 发生冲突。

   `inst_ADD` 指令的目标寄存器 `rd` 会通过 `inst_wdest_rd` 信号正确地被识别，并最终影响到 `rf_wdest`，这个目标寄存器地址会随着指令在流水线中传递（即成为 `EXE_wdest`, `MEM_wdest`, `WB_wdest`）。

2. **流水线暂停/阻塞机制 (在 `decode.v` 和 `pipeline_cpu.v` 中):**

   如果 `rs_wait` 或 `rt_wait` 为高（表示检测到数据冒险），`decode.v` 中的 `ID_over` 信号会变为低。

   在 `pipeline_cpu.v`中，`ID_over` 信号控制着是否将当前ID阶段的指令和数据传递到EXE阶段（通过 `EXE_valid <= ID_over` 和 `ID_EXE_bus_r` 的锁存条件 `if(ID_over && EXE_allow_in)`）。

   同时，`ID_over` 也会影响 `ID_allow_in` 信号 (`ID_allow_in = ~ID_valid | (ID_over & EXE_allow_in)`), 进而阻塞IF阶段向ID阶段输送新的指令。

这种机制通过暂停流水线来等待前一条指令完成写回操作（或者至少到达一个可以安全读取其结果的阶段，尽管这里没有显式的数据前推逻辑，但暂停直到WB阶段完成也是一种解决方法），从而避免了后续指令读取到错误的数据。

4.2乘法

对于乘法来说可能产生的冒险有两个：

1. **对HI/LO寄存器的RAW (Read After Write) 数据冒险**：

   `inst_MULTU` 指令会计算64位结果，并将高32位写入HI寄存器，低32位写入LO寄存器。这个过程通常在EXE（执行）阶段完成，或者至少启动，并在后续某个时刻完成。

   如果紧随 `inst_MULTU` 之后的指令是 `MFHI` (Move From HI) 或 `MFLO` (Move From LO)，它们需要读取HI或LO寄存器的值。

   若 `inst_MULTU` 尚未完成计算并将结果写入HI/LO寄存器，而 `MFHI` 或 `MFLO` 就尝试读取，则会读到旧的、错误的数据。

2. **对源操作数 (rs, rt) 的RAW数据冒险**：

   `inst_MULTU` 指令需要读取源寄存器 `rs` 和 `rt` 的值。

   如果 `inst_MULTU` 之前有指令会写入 `rs` 或 `rt`，并且该写操作尚未完成，那么 `inst_MULTU` 可能会读到旧值。这种冒险由 decode.v 中的 `rs_wait` 和 `rt_wait` 逻辑处理，这与 `inst_ADD` 等其他算术指令类似。

对此我们的代码也已经给出了对策：

1. **乘法完成信号 (`mult_end`)**:

   `exe.v` 模块实例化了一个 `multiply_module`，该模块有一个输出信号 `mult_end`，当乘法操作完成时，`mult_end` 变为高电平。乘法操作通常需要多个时钟周期。

2. **EXE阶段完成信号 (`EXE_over`) 的控制**:

   在 `exe.v`中，`EXE_over` 信号的逻辑如下:

```
// filepath: d:\Desktop\source_code\source_code\8_pipeline_cpu\exe.v
// ...existing code...
assign EXE_over = EXE_valid & (~multiply | mult_end);
// ...existing code...
```

这里的 `multiply` 信号指示当前指令是否为乘法指令 (包括 `inst_MULTU`)。

如果当前指令是乘法指令 (`multiply` 为高)，则 `EXE_over` 信号会一直保持为低，直到 `mult_end` 信号变为高（即乘法完成）。

如果 `EXE_over` 为低，根据 `pipeline_cpu.v`中的流水线控制逻辑 (`MEM_allow_in = ~MEM_valid | (MEM_over & WB_allow_in )` 和 `EXE_allow_in = ~EXE_valid | (EXE_over & MEM_allow_in)` 等)，EXE阶段将不会把数据传递给MEM阶段，并且ID阶段也不会把数据传递给EXE阶段。这相当于**暂停 (stall)** 了流水线。

总的来说，它的执行原理如下:

当 `inst_MULTU` 指令进入EXE阶段时:

1. `multiply` 信号为高。
2. 只要乘法器没有完成 (`mult_end` 为低)，`EXE_over` 就会为低。
3. 流水线会暂停在EXE阶段（以及之前的IF和ID阶段）。
4. 直到乘法器完成计算，`mult_end` 变为高，`EXE_over` 才会变为高。
5. 此时，`inst_MULTU` 的结果（HI和LO的值）已经计算出来，并通过 `EXE_MEM_bus` 传递给后续的MEM阶段，最终在WB阶段写入实际的HI/LO寄存器（或者准备好被 `MFHI`/`MFLO` 读取）。
6. 后续的 `MFHI` 或 `MFLO` 指令在 `inst_MULTU` 完成后才能继续执行，从而能够读取到正确更新后的HI/LO值。

4.3转移指令

关于转移指令的对应冒险的方法如下:

1. **分支/跳转检测与目标地址计算 (在 `decode.v` 中):**
   - `inst_jbr` 信号标识当前指令是否为任何类型的跳转或分支指令。
   - `jbr_taken` 信号判断分支是否会发生（条件满足或无条件跳转）。
   - `jbr_target` 计算出跳转的目标地址。
   - `jbr_bus = {jbr_taken, jbr_target}` 将这些信息传递给取指 (IF) 阶段。
2. **分支延迟槽的实现暗示:**
   - 在 `decode.v]` 中，计算分支目标地址时使用了 `bd_pc = pc + 3'b100;`，这是分支指令之后那条指令的地址（即延迟槽指令的地址）。跳转目标地址的计算基于这个 `bd_pc`。
   - 这意味着，紧跟在分支或跳转指令后面的那条指令（位于延迟槽中）**总是会被执行**，无论分支是否发生。这是MIPS处理控制冒险的典型策略，目的是利用这个周期做一些有用的工作。

3. **流水线暂停以确保分支决策正确传递 (在 `decode.v` 和 `pipeline_cpu.v` 中):**
    - 关键在于 `[decode.v` 中的 `ID_over` 信号逻辑:

```
// filepath: d:\Desktop\source_code\source_code\8_pipeline_cpu\decode.v
// ...existing code...
assign ID_over = ID_valid & ~rs_wait & ~rt_wait & (~inst_jbr | IF_over);
// ...existing code...
```

    如果当前指令是转移指令 (`inst_jbr` 为高)，那么 `ID_over` 信号要变为高，除了满足数据冒险解除的条件外，还必须等待 `IF_over` 信号为高。

    - `IF_over` (来自 `fetch.v` 模块) 表示取指阶段已经完成其当前操作。对于分支指令，这意味着取指阶段已经根据上一周期ID阶段传递过来的 `jbr_bus` 信息（决定是否跳转以及目标地址）来确定了 *下一个* 要取的指令地址（即延迟槽指令之后的指令地址）。
    - 如果 `ID_over` 因为 `(~inst_jbr | IF_over)` 条件中的 `IF_over` 未满足而变低，那么:
        - `pipeline_cpu.v` 中的 `ID_allow_in` 会阻止新的指令进入ID阶段。
        - EXE阶段也不会接收来自ID阶段的新指令。
        - 这实质上是**暂停了ID阶段**，直到IF阶段处理完分支决策并准备好取正确的下一条指令（延迟槽指令的下一条）。

1. **指令冲刷/取消 (`cancel` 信号):**
    - `WB_module` 可以产生 `cancel` 信号（例如，在 `ERET` 或 `SYSCALL` 指令，或者更一般地，在异常发生时）。
    - 这个 `cancel` 信号会传递到 `pipeline_cpu.v` 并用于清空流水线中较早阶段的 `valid` 位，例如:

```
// filepath:
d:\Desktop\source_code\source_code\8_pipeline_cpu\pipeline_cpu.v
// ...existing code...
always @(posedge clk)
begin
    if (!resetn || cancel)
    begin
        ID_valid <= 1'b0; // EXE_valid, MEM_valid, WB_valid 类似
    end
    // ...existing code...
end
```

    - 这确保了如果发生需要改变控制流的事件（如异常），流水线中错误的指令会被作废。虽然这里没有显式的分支预测失败后的冲刷，但 `cancel` 机制可以用于更广泛的控制流改变。

4.4 读写内存代码

MEM阶段的加载延迟处理 (`mem.v`):

- 在 `mem.v` 模块中，`MEM_over` 信号的产生考虑了加载指令的特性。对于 `inst_load` 指令，`MEM_over` 的断言会延迟一个周期（通过 `MEM_valid_r` 寄存器实现）。

```
// filepath: d:\Desktop\source_code\source_code\8_pipeline_cpu\mem.v
// ...existing code...
reg MEM_valid_r;
```

```
always @(posedge clk)
begin
    if (MEM_allow_in)
    begin
        MEM_valid_r <= 1'b0;
    end
    else
    begin
        MEM_valid_r <= MEM_valid;
    end
end
assign MEM_over = inst_load ? MEM_valid_r : MEM_valid;
// ...existing code...
```

- 这意味着加载指令在MEM阶段实际会花费两个时钟周期才能完成（第一个周期发出地址，第二个周期数据返回并使 `MEM_over` 为高）。这模拟了同步RAM读取通常需要的延迟。
- **ID阶段的数据冒险检测与流水线暂停 (`decode.v` 和 `pipeline_cpu.v`):**

  这一块与加法相同。

tore-Load Hazard (存储后加载冒险)

- **场景**：一条存储指令（如 `SW`, `SB`）向某个内存地址写入数据，紧随其后的加载指令从相同的内存地址读取数据。
- **问题**：如果加载指令在存储指令实际完成内存写入之前就尝试读取，可能会读到旧数据。
- 代码中的处理：
  - 存储指令在MEM阶段将数据写入数据存储器 (`data_ram`)。
  - 加载指令在MEM阶段从数据存储器读取数据。
  - 由于流水线的顺序执行，如果一条 `SW` 指令在MEM阶段，下一条 `LW` 指令（即使是紧随其后）最早也会在其后的时钟周期才到达MEM阶段。
  - 假设您的 `data_ram` 是同步写入的（即在时钟边沿完成写入），那么当 `LW` 指令在MEM阶段读取时，前一个周期的 `SW` 指令已经完成了写入。因此，这种基本的Store-Load冒险通常由流水线的自然顺序和同步内存操作来正确处理。

五、实验结果分析

我们在实现了上述的代码更改之后就可以开始进行我们的$Verilog$仿真验证了：

5.1无符号加法

我们首先编写我们如何去实现的汇编代码：

```
addiu $s2,$s0,FFFFFFFE     ->$s2=FFFFFFFE(在原代码中$s1=1)
addiu  $s3,$s0,3           ->$s3=3
add    $s1,$s2,$s1         ->$s1=-1
add    $s3,$s2,$s3         ->$s3=1
```

然后将其变为机器码：

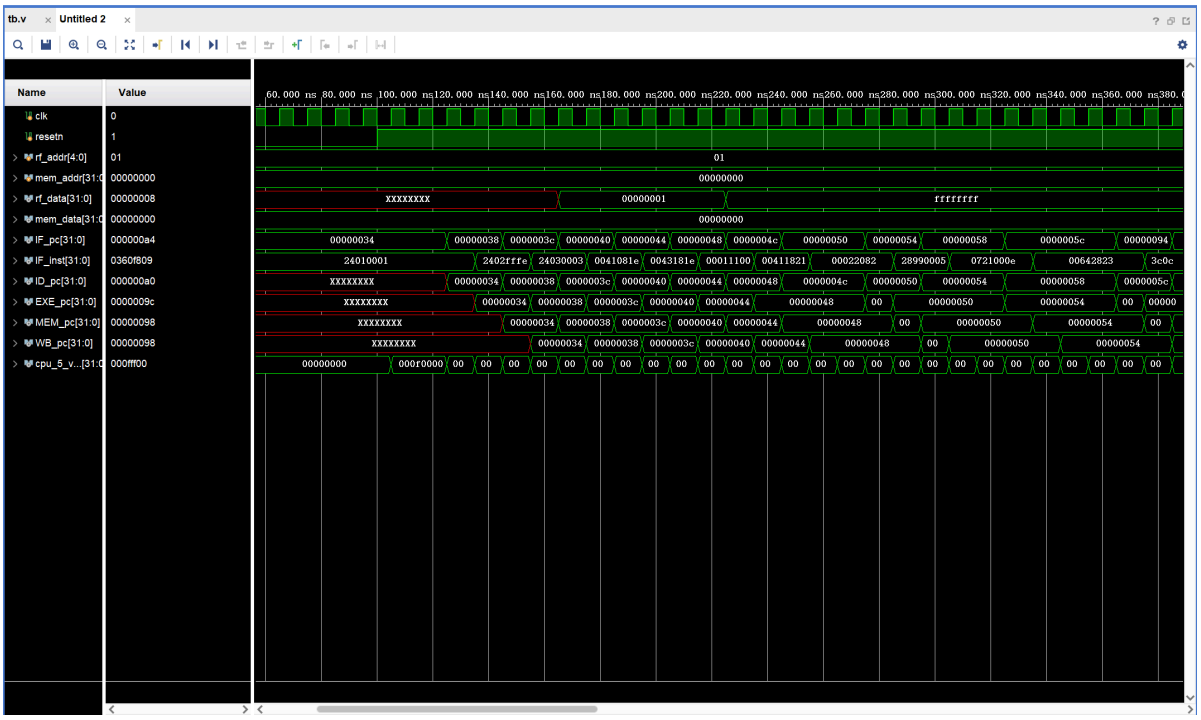
```
00000000010000010001100000100001
00000000000000010001000001000010
00101000100110010000000000000101
00000111001000010000000000001110
```

和相应的十六进制机器码：

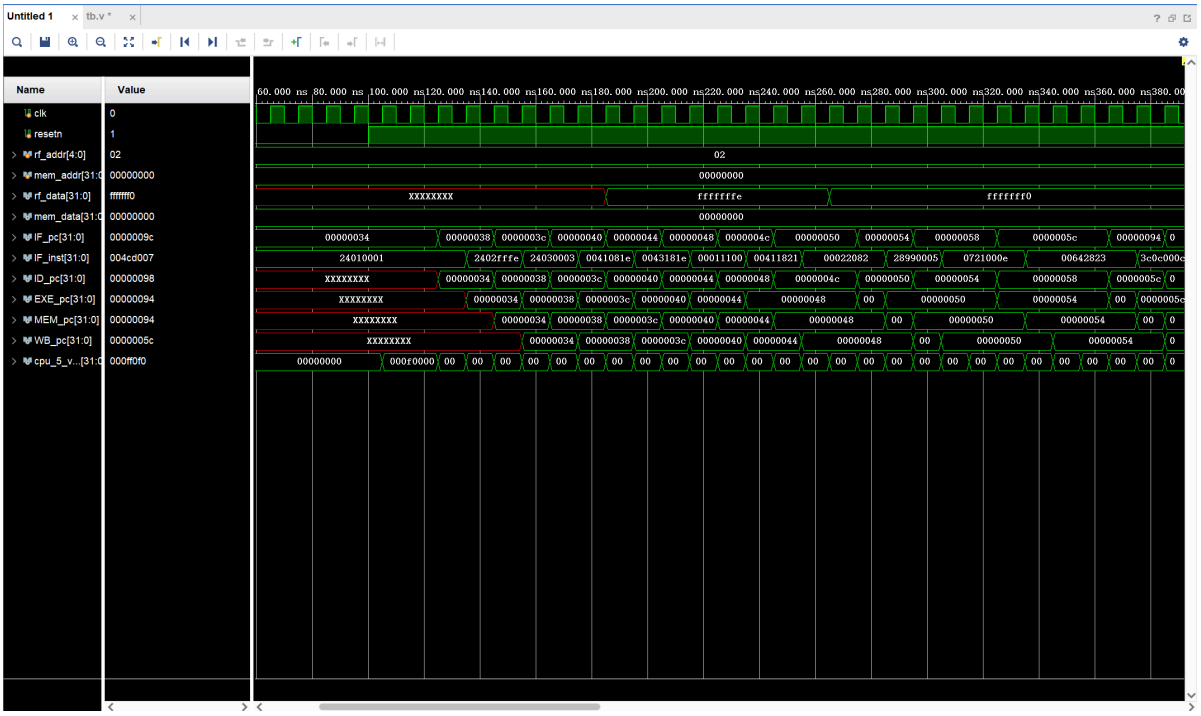
```
2402FFFE
24030003
0041081E
0043181E
```

在将十六进制代码粘贴至coe文件之后我们可以得到如下仿真结果：

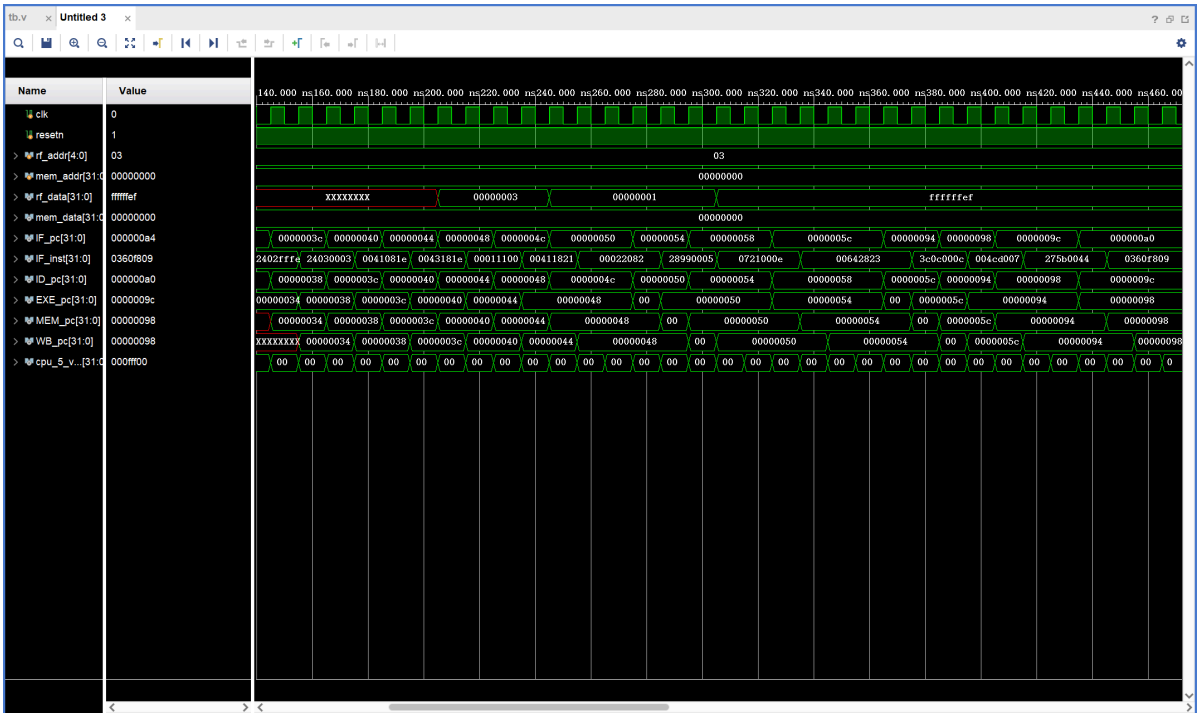
5.1.1 s1



5.1.2 s2



5.1.3 s3



我们可以看到 `s2` 变为了我们预期的-1， `s3` 变为了我们预期的1。代表我们的功能添加成功

5.2 有符号乘法

首先我们先来编写汇编代码：

```
addiu $s1,$s0,2          ->$s1=2
addiu $s2,$s0,FFFFFF     ->$s2=FFFFFFFF
multu $s1,$s2            ->HI=00000001,LO=FFFFFFFE
```

接着我们得到机器码：

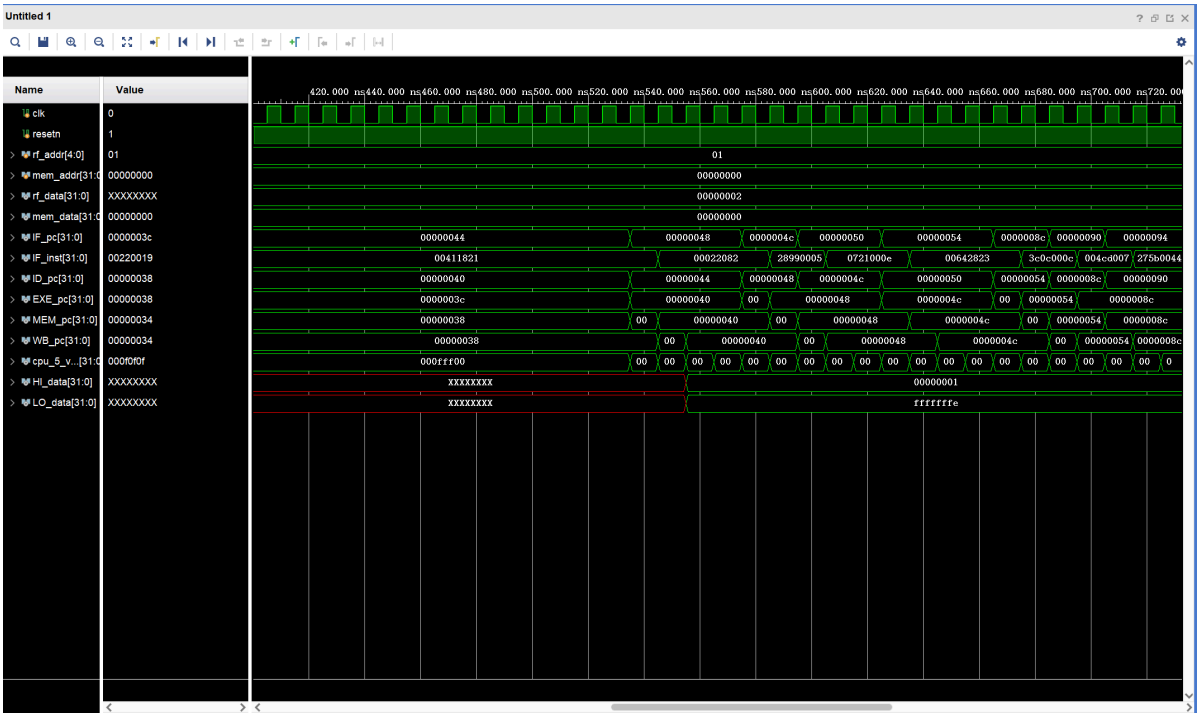
```
24010002
2402FFFF
00220019
```

因为我们的乘法结果保存在 `HI_data` 和 `LO_data` 中，所以我们对 `tb.v` 进行一定的更改使得这两个数据能够显示出来：

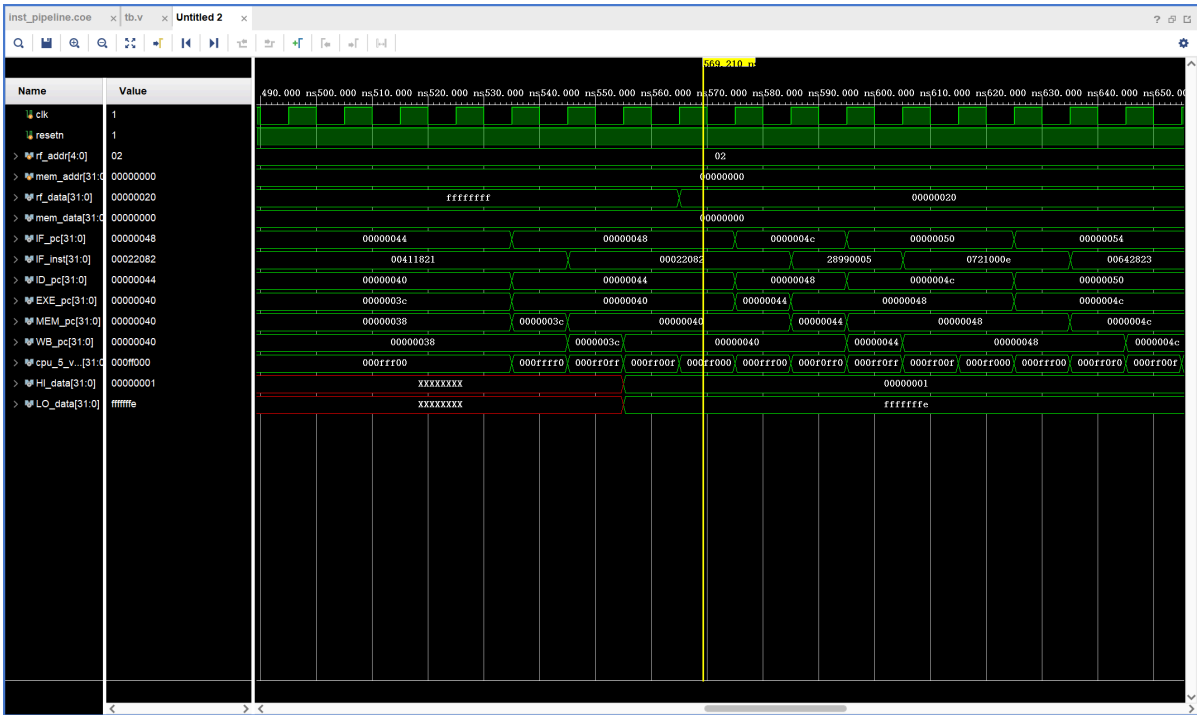
```
wire [31:0] HI_data;
wire [31:0] LO_data;

 pipeline_cpu uut (
        .clk(clk),
        .resetn(resetn),
        .rf_addr(rf_addr),
        .mem_addr(mem_addr),
        .rf_data(rf_data),
        .mem_data(mem_data),
        .IF_pc(IF_pc),
        .IF_inst(IF_inst),
        .ID_pc(ID_pc),
        .EXE_pc(EXE_pc),
        .MEM_pc(MEM_pc),
        .WB_pc(WB_pc),
        .cpu_5_valid(cpu_5_valid),
        .HI_data(HI_data),
        .LO_data(LO_data)
    );
```

接下来我们进行仿真实验：

5.2.1 s1



5.2.2 s2



我们可以看到有符号乘法的结果确实是正确的，结果为负数。

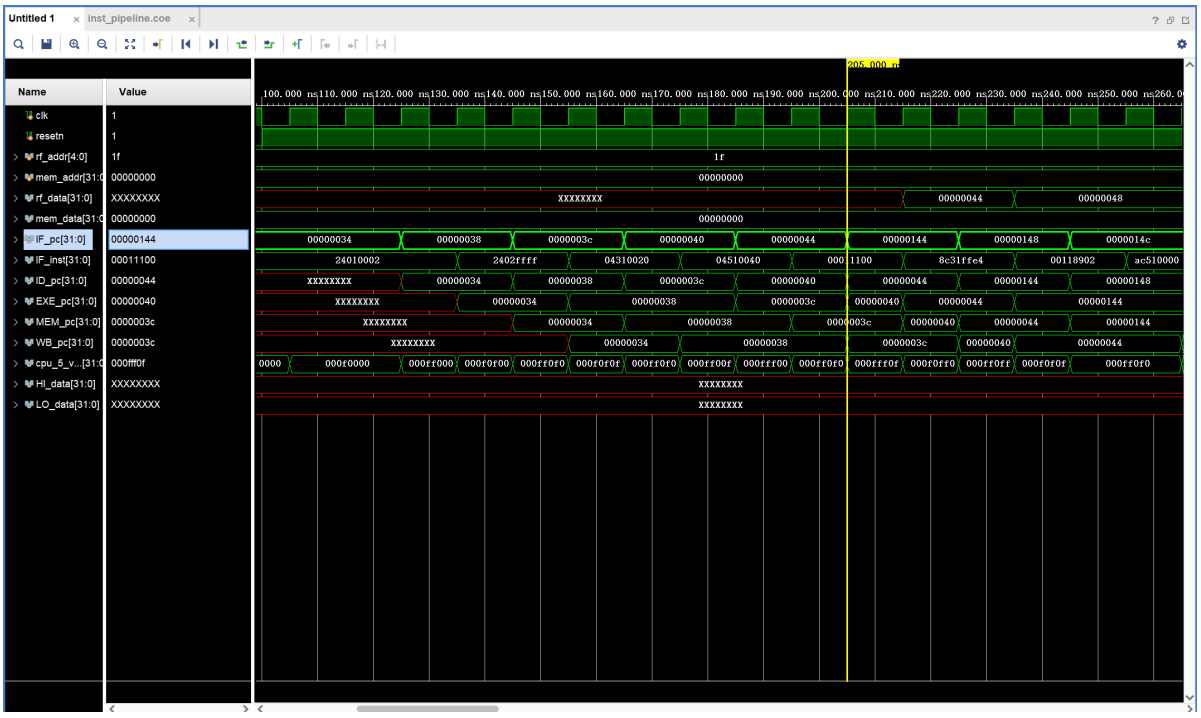
5.3 转移指令（小于则跳转）

我们将比较BLTZAL在大于0和小于0两种情况下的值的变化，而且因为跳转指令同时具有两种功能：跳转和链接，所以我们将观察31号寄存器的值：

```
addiu  $s1,$s0,1        $s1=1
addiu  $s2,$s0,FFFF     $s2=FFFFFFFF
BLTZAL $S1,+32          $31=44H(不跳转)
BLTZAL $S2,+64          $31=48H(跳转)
跳转地址：
bd_pc = 50 (0x44)
offset = 0x0040 = 64
offset << 2 = 64 << 2 = 256 (0x100)
br_target = bd_pc + (offset << 2) = 50 + 256 = 306 (0x144)
```

然后我们来编写机器码：

```
24010001
2402FFFF
04510020
04520040
```



我么可以看到因为$1的值大于0所以并没有进行跳转，但是44H被储存到了$31中，$2的值小于0，所以48H被储存到了$31中，并且pc跳转到了0x144。与我们预期的跳转地址一致。

5.4 内存读操作

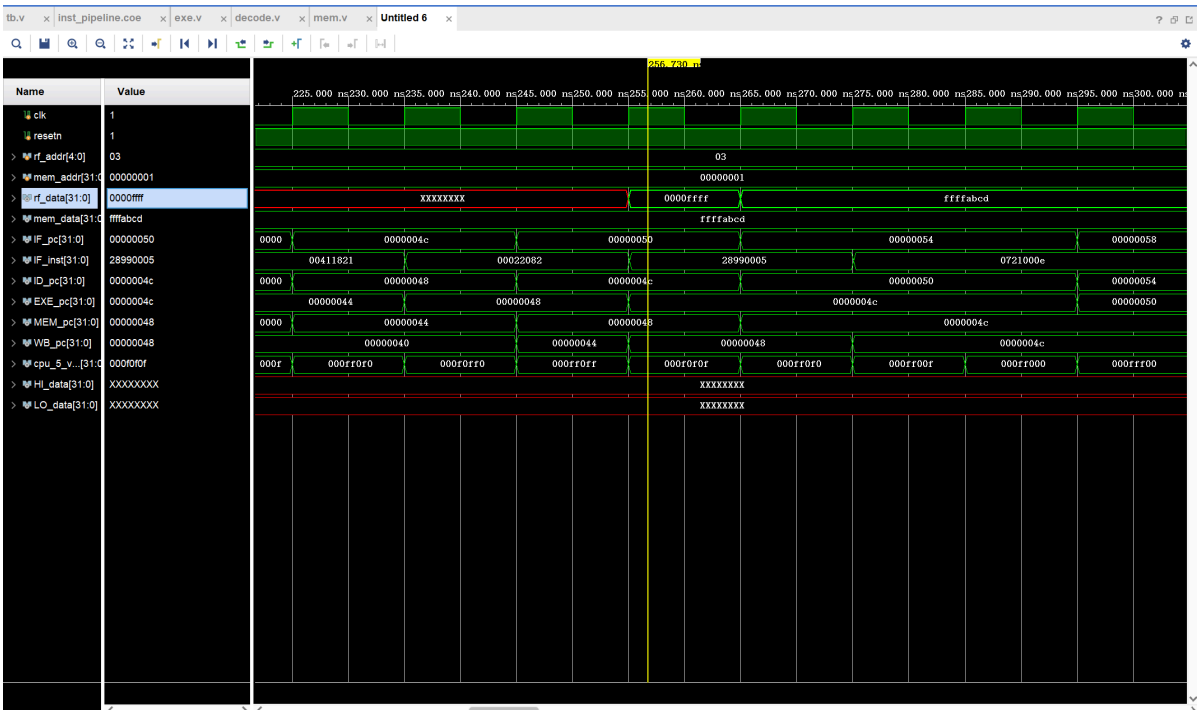
汇编代码如下：

```
addiu $s1,$s0,ABCD      ->$s1=FFFFABCD(A的二进制首位为1)
sw    $s1,2($s0)        ->mem[2]=FFFFABCD
LHU   $S3,2($s0)        ->$s3=0000FFFF
```

我们接着编写机器码：

```
2401ABCD
AC010002
94030002
```

接着我们进行仿真实验：



我们可以看到mem[2]的值确实为`ffffabcd`，我们的读操作也确实读到了高四位并保存了下来。`ffff0000`。

5.5 内存写操作

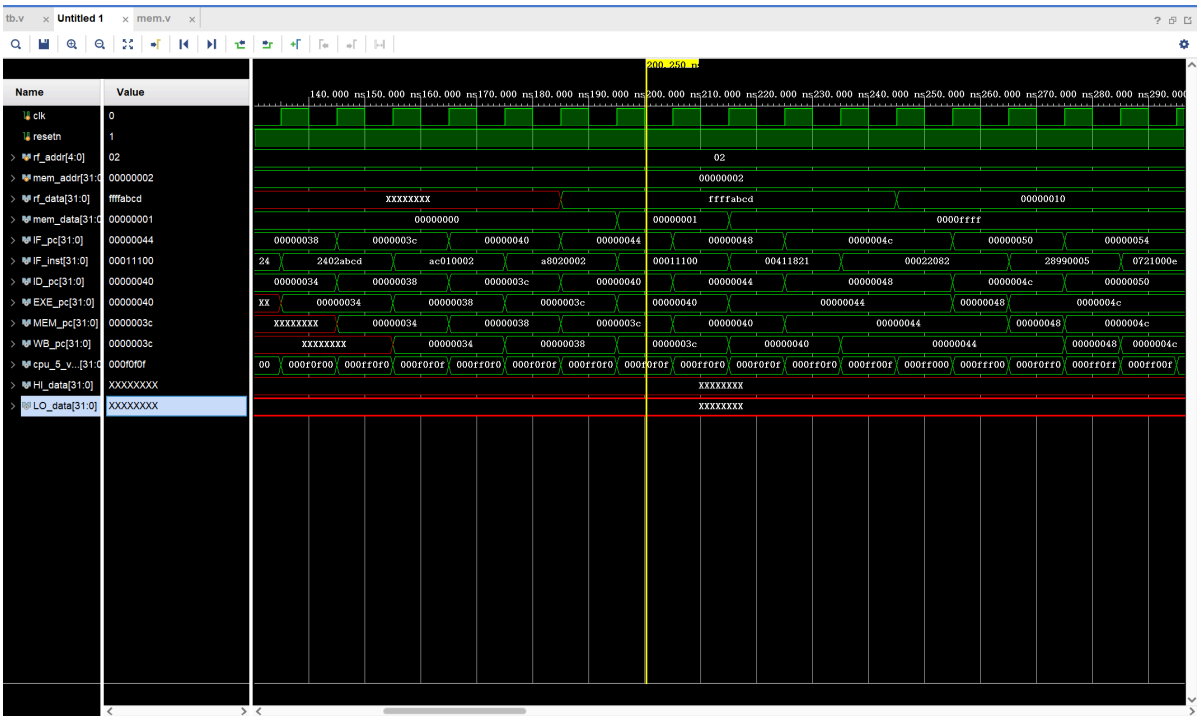
我们的思路是先将mem[2]变为`ffffabcd`，再利用写入高半位指令LHU将上四位变为0.

```
addiu   $s1,$s0,ABCD      ->$s1=00000001
addiui  $s2,$s0,7BCD      ->$s2=FFFFABCD
SW      $S1,2($s0)        ->MEM[2]=00000001
SWL     $s2,2($s0)        ->MEM[2]=0000FFFF
```

接着我们来编写机器码：

```
24010001
2402ABCD
AC010002
A8020002
```

接着我们进行仿真实验验证：



我们看到了实验结果却是和我们预想的一样，之前我们写入了`00000001`，在进行半位写入之后变为了`0000ffff`。到此我们所有的指令都已经添加成功了。

七、总结感想

本次五级流水线CPU的实验，是一次全面而深刻的计算机组成原理实践。通过亲手分析和修改五级流水线的源码，我对CPU内部的复杂工作机制、指令执行的精妙流程以及流水线设计中的关键问题有了更为具体和深入的理解。

在实验初期，对现有五级流水线进行问题分析，特别是指令相关和流水线冲突的分析，让我认识到理论知识与实际设计的紧密联系。虽然本次实验通过IP核的正确配置幸运地规避了之前实验中遇到的配置bug，但也提醒了我在未来的硬件设计中，对每一个配置选项都需要有清晰的认识。

指令扩展的过程是本次实验的核心挑战，也是收获最大的部分。添加ADD、MULTU、BLTZAL、LHU和SWL这几条指令，不仅仅是简单地增加几行代码。每条指令的添加，都涉及到对其MIPS编码规则的理解、数据通路和控制信号的修改。例如，为MULTU指令添加`mul_sign`信号并在`decode.v`、`exe.v`和`multiply.v`中逐级传递和处理，让我体会到信号在流水线各级之间传递的重要性。同样，为LHU和SWL指令修改`mem_control`总线，并调整`mem.v`中的读写逻辑，也让我对访存指令的复杂性有了新的认识。特别是SWL指令，需要根据地址的低两位来确定写入的字节，其写使能信号`dm_wen`和数据组织`dm_wdata`的逻辑设计，让我对非对齐内存访问的处理有了初步的实践。

处理流水线冒险是另一个重要的学习点。虽然源码中已经包含了一些冒险处理机制，但在添加新指令后，需要重新审视这些机制是否依然有效。例如，乘法指令MULTU执行时间较长，其对HI/LO寄存器的写后读冒险通过`mult_end`信号配合`EXE_over`实现流水线暂停；转移指令BLTZAL则依赖于分支延迟槽和ID阶段的暂停（等待IF阶段处理完分支决策）来处理控制冒险。这些都加深了我对数据冒险、控制冒险以及结构冒险解决策略的理解，如流水线暂停（stall）、数据前推（forwarding，虽然本次实验主要依赖暂停）和指令调度（如延迟槽）等。

仿真验证环节同样不可或缺。通过编写汇编代码、转换为机器码、进行波形仿真，并仔细分析`$s1`、`$s2`、`$s3`、`HI_data`、`LO_data`、`$31`寄存器以及内存单元`mem[2]`的值，确保了每一条新增指令的功能都符合预期。这个过程不仅锻炼了我的调试能力，也让我学会了如何通过具体的测试用例来验证设计的正确性。

总的来说，这次实验让我从理论走向实践，深刻体会了CPU设计中的细节与挑战。从指令解码、执行到访存、写回，再到流水线控制和冒险处理，每一个环节都充满了值得学习的知识。虽然过程复杂，但成功实现功能并验证通过后，成就感油然而生。这次实验不仅巩固了课堂所学的理论知识，更提升了我的硬件设计和调试能力，为后续更复杂的数字系统设计打下了坚实的基础。