组成原理课程第六次实验报告

实验名称：　五级流水线 CPU

学号:2310764 姓名:王亦辉 班次:计科一班

1　实验目的

1. 在多周期 CPU 实验完成的提前下，深入理解 CPU 流水线的概念。
2. 熟悉并掌握流水线 CPU 的原理和设计。
3. 最终检验运用 verilog 语言进行电路设计的能力。
4. 通过亲自设计实现静态5级流水线 CPU，加深对计算机组成原理和体系结构理论知识的理解。
5. 培养对CPU设计的兴趣，加深对CPU现有架构的理解和深思。

2　实验内容说明

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。

3　实验原理图

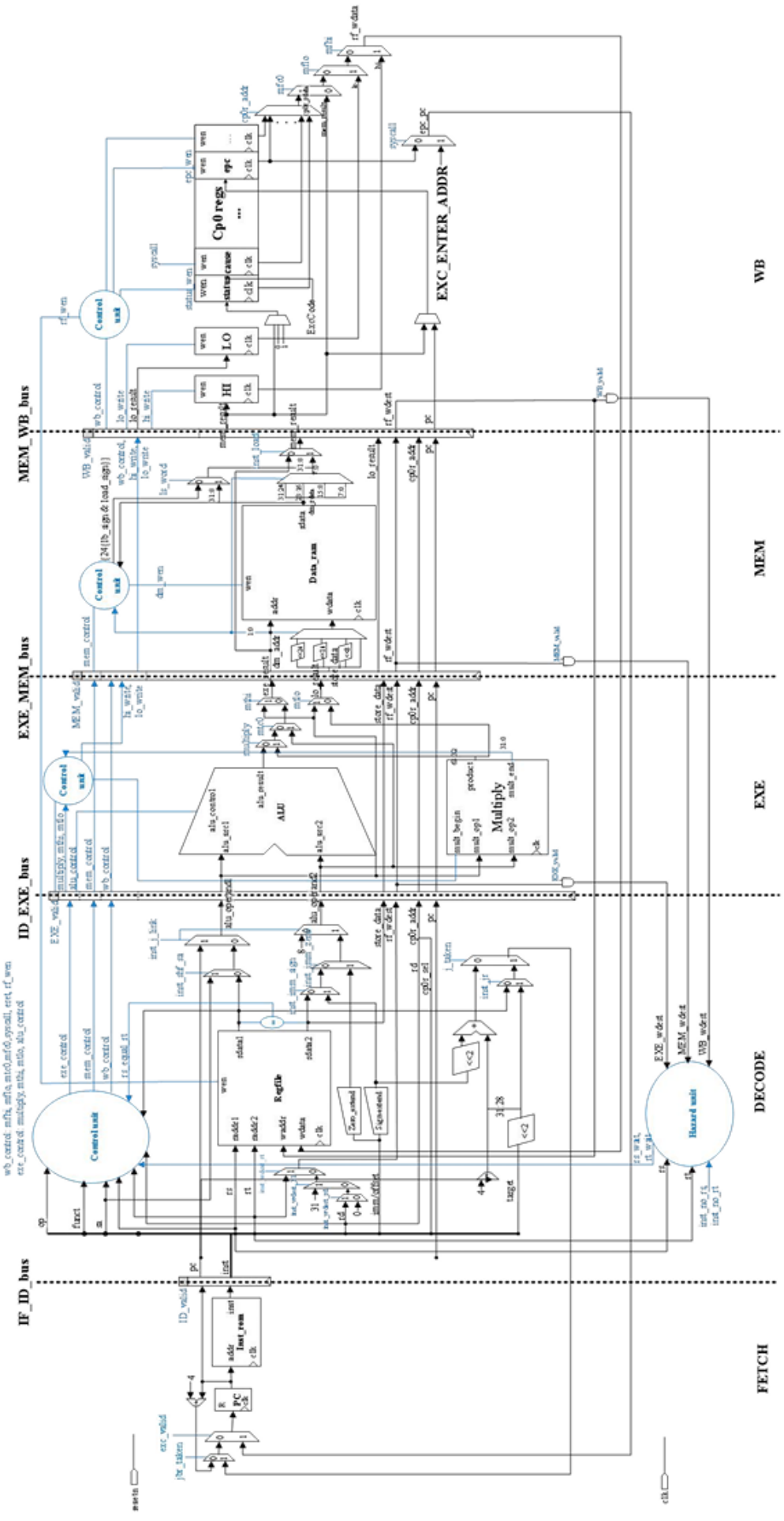


图 9.4 5 级流水线 CPU 的实现框图

4 实验步骤

4.1 问题方面

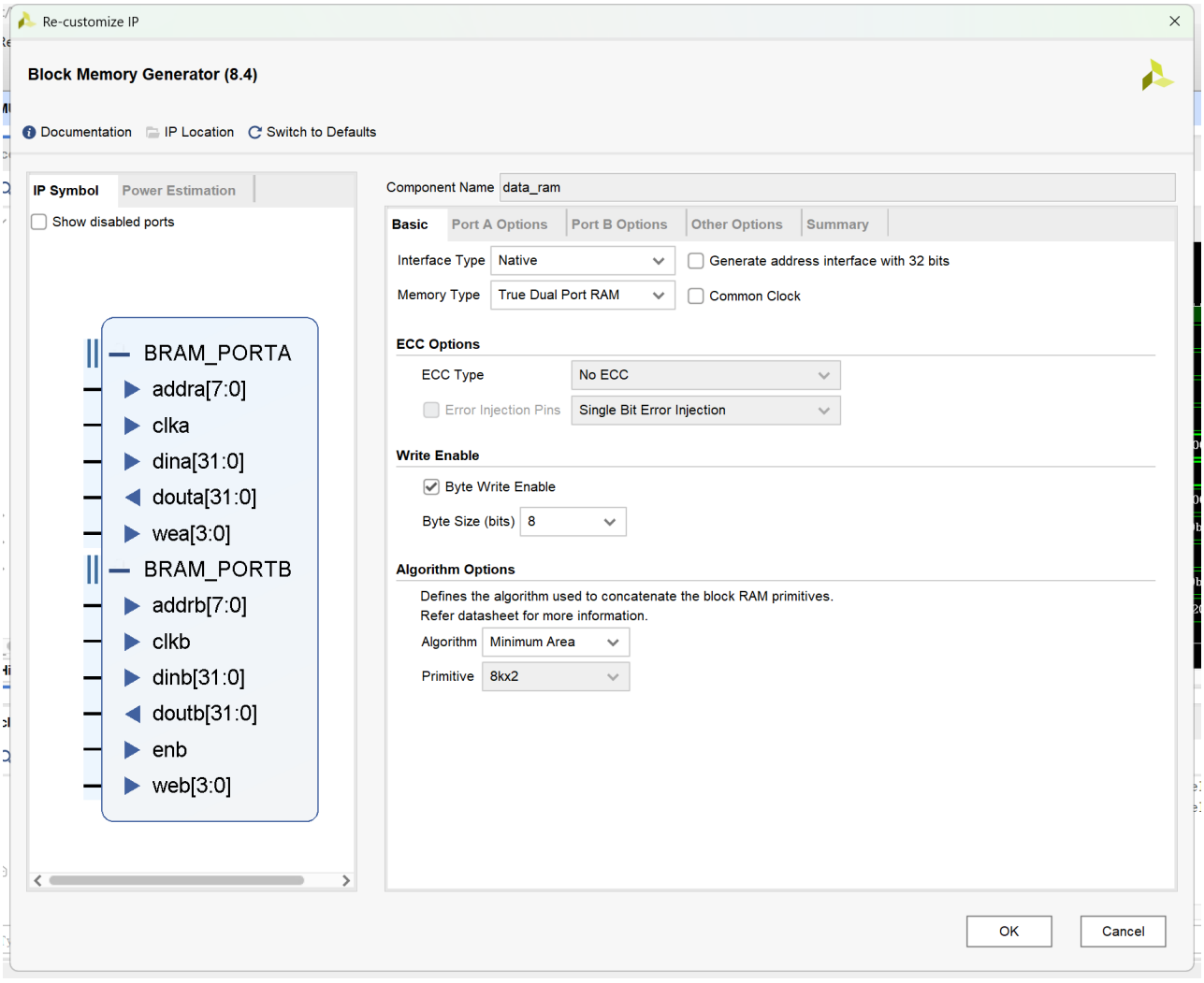
现阶段的五级流水线 CPU 有如下问题

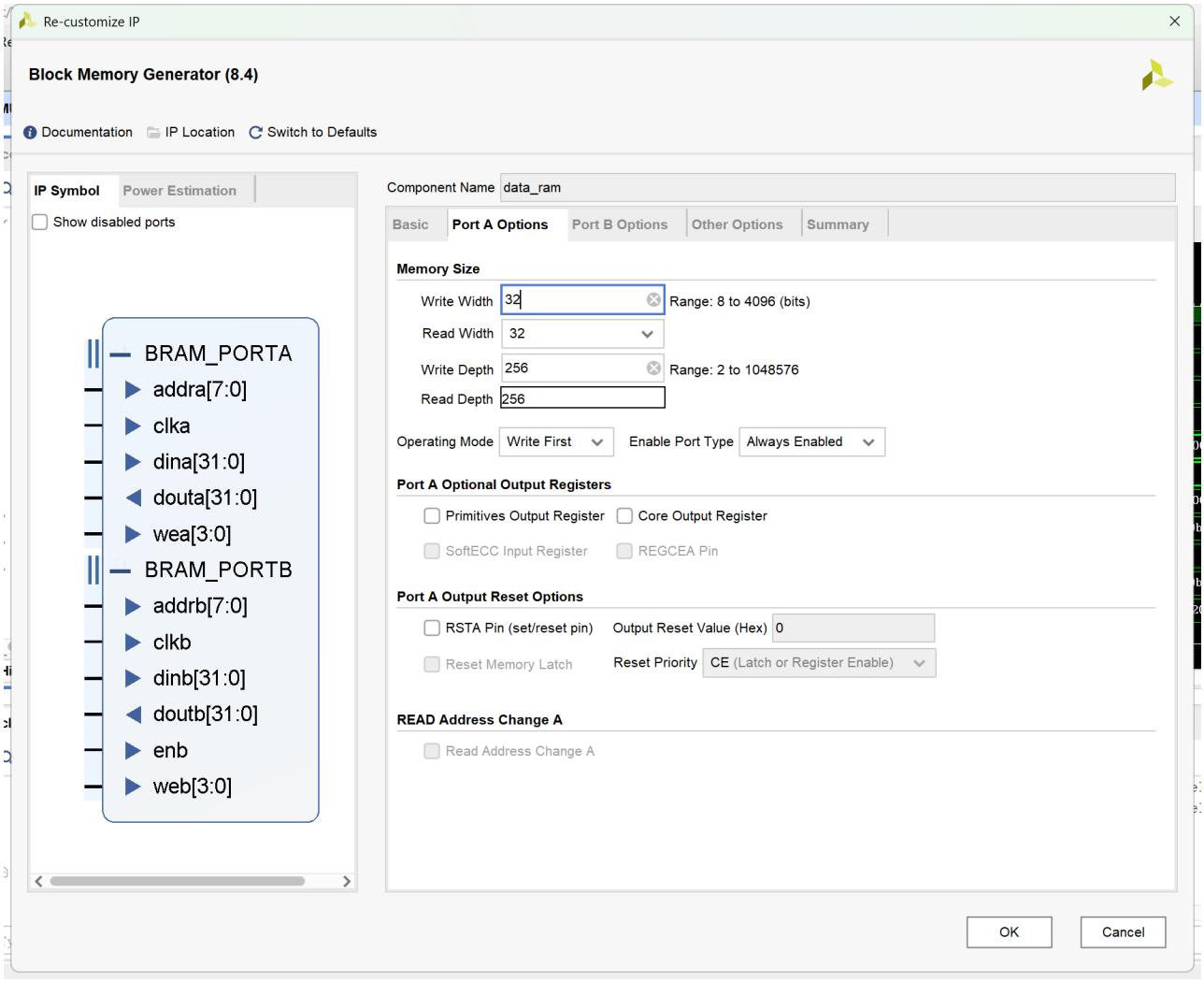
1. 控制冒险：在跳转指令尚未完成判断和写回时，不知是否需要跳转，但地址与之相邻的下一条指令已经被取指，进入流水线，并将被完整执行，如果跳转成立，则该条指令本不应被执行却被执行，导致错误。
2. 数据冒险：某条指令如果需要写内存/寄存器的某个数据，而下一条指令需要读取这个数据，由于流水线中，写回是在执行和内存操作之后，并且没有对这种情况特殊处理，因此将导致下一条指令读取不正确的值。

4.2 IP 核配置方面的额外努力

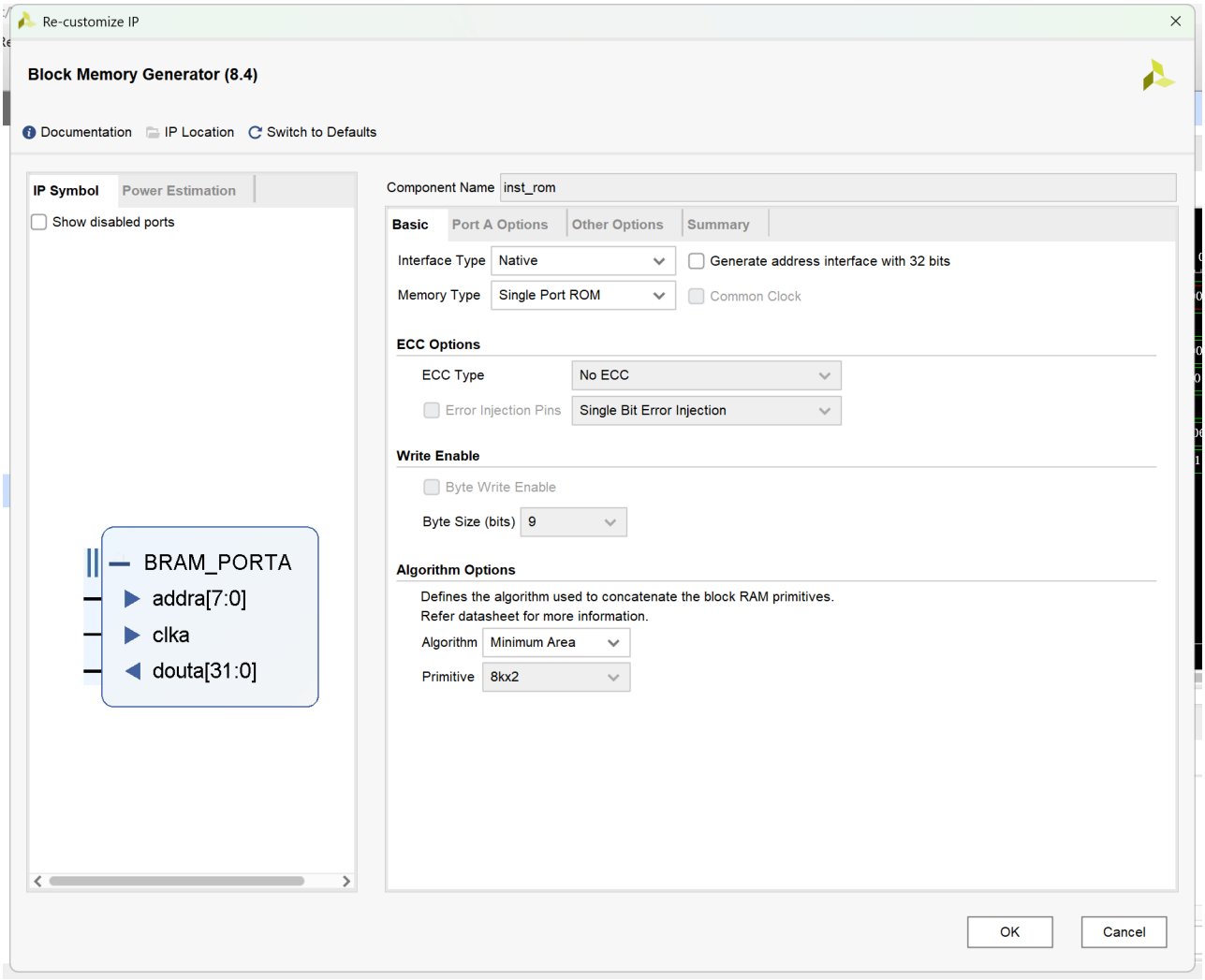
data_ram 和 inst_rom 姑且这样配置的

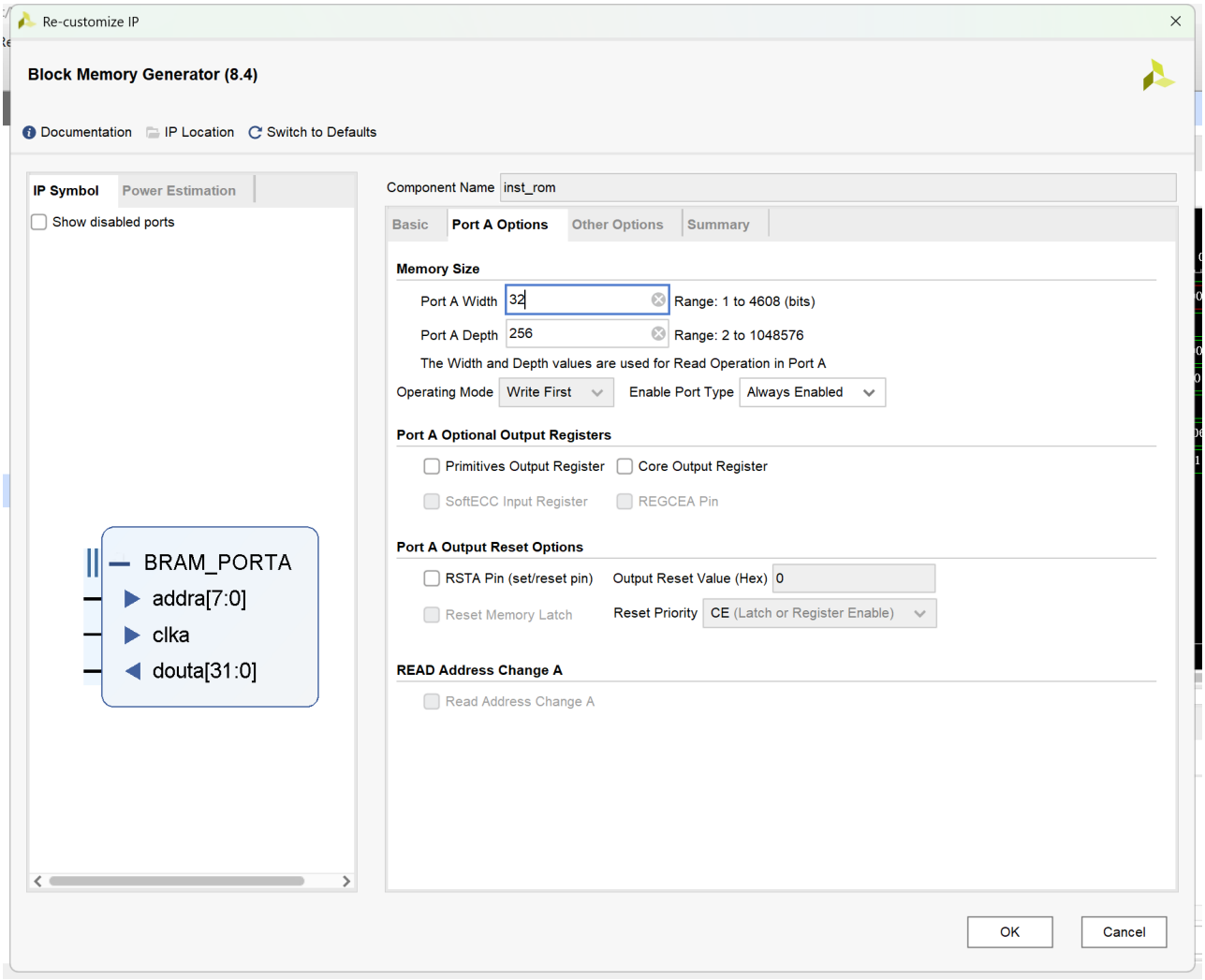
dataram（port B 似乎是不重要的 (用来展示数据的？)）（取消勾选 Show disabled ports，就方便观察 IP 核的输入输出了）

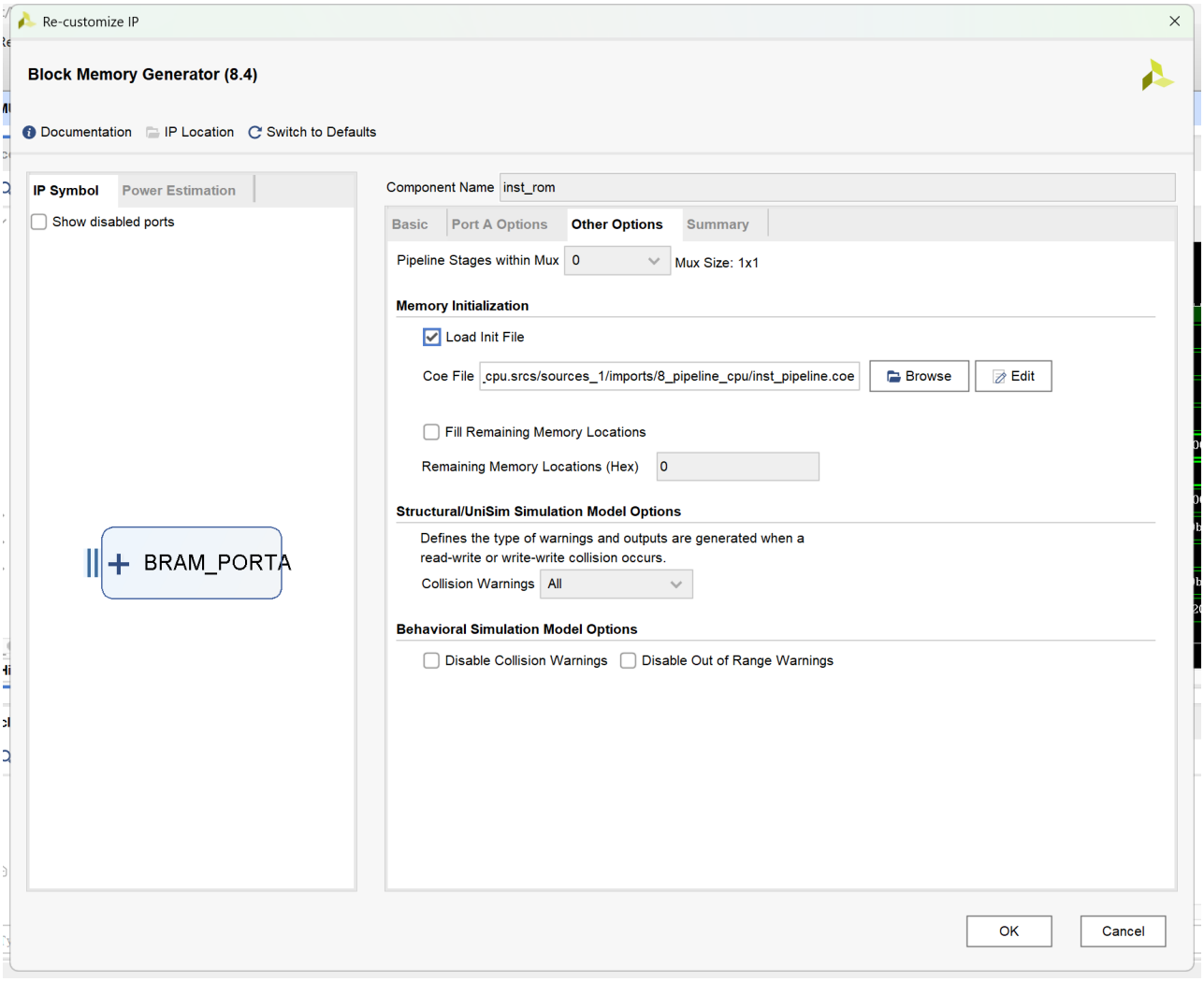



Re-customize IP
Block Memory Generator (8.4)
Documentation
IP Location
Switch to Defaults
IP Symbol
Power Estimation
Show disabled ports
BRAM_PORTA
addra[7:0]
clka
dina[31:0]
douta[31:0]
wea[3:0]
BRAM_PORTB
addrb[7:0]
clkb
dinb[31:0]
doutb[31:0]
enb
web[3:0]
Component Name
data_ram
Basic
Port A Options
Port B Options
Other Options
Summary
Memory Size
Write Width
32
Range: 8 to 4096 (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

inst_rom

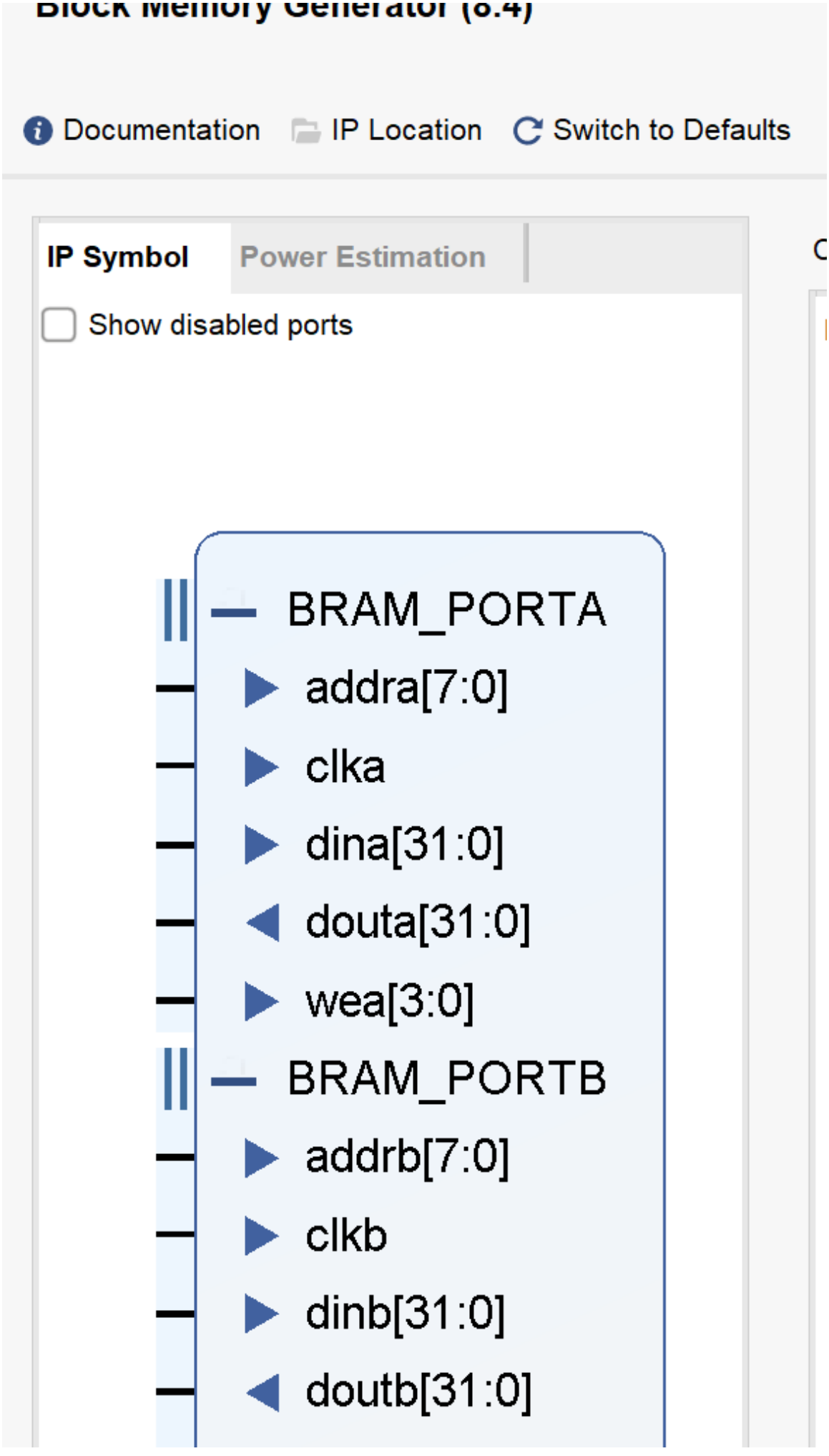

Re-customize IP
Block Memory Generator (8.4)
Documentation
IP Location
Switch to Defaults
IP Symbol
Power Estimation
Show disabled ports
BRAM_PORTA
addra[7:0]
clka
douta[31:0]
Component Name
inst_rom
Basic
Port A Options
Other Options
Summary
Interface Type
Native
Generate address interface with 32 bits
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
BRAM_PORTA
addra[7:0]
clka
douta[31:0]
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

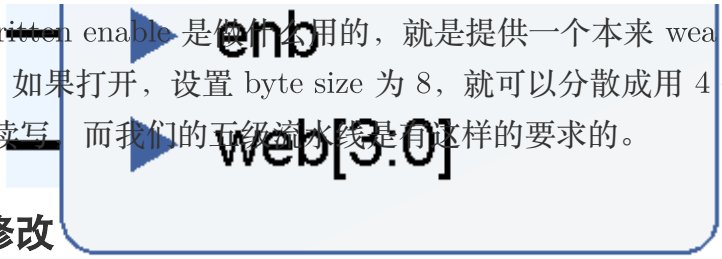


设计实战上的同步 ram 配置不能用，应该在实验手册的基础上，取消勾选 Primitive output register。开始时，联想到多周期 CPU，设计实战中让我们取消勾选 primitive output register ，这是需要的，于是按照那里配的，结果有问题，在这里 Debug 花了很久。知道了 write depth 是做什么用的，就是内存的大小，需要与输入的 `dm_addr[9:2]` 匹配，因此应该使用 8 位的。

发现这个展开图方便我们看 ip 核的接口是怎样的。


Block Memory Generator (8.4)
Documentation
IP Location
Switch to Defaults
IP Symbol
Power Estimation
Show disabled ports
BRAM_PORTA
addra[7:0]
clka
dina[31:0]
douta[31:0]
wea[3:0]
BRAM_PORTB
addrb[7:0]
clkb
dinb[31:0]
doutb[31:0]

知道了 Byte written enable 是做什么用的，就是提供一个本来 wea 是控制单字的一整个 32 位是否允许写，如果打开，设置 byte size 为 8，就可以分散成用 4 个位控制单字的 4 个 byte 是否允许读写，而我们的五级流水线是有这样的要求的。



4.3 代码修改

新增指令

`inst_NXOR`：同或（运算类指令）

`inst_DIV`：除法（乘除类指令）

`inst_BNEZ`：不等于零则跳转（转移指令）

`inst_LH`：低半字 load（访存指令读）

`inst_SH`：低半字 store（访存指令写）

下面是修改处。

4.3.1 `pipeline_cpu.v`

更新总线位宽

```
132-    wire [166:0] ID_EXE_bus;  // ID->EXE级总线
133-    wire [153:0] EXE_MEM_bus; // EXE->MEM级总线
134     wire [117:0] MEM_WB_bus;  // MEM->WB级总线
135
136     //锁存以上总线信号
137     reg [ 63:0] IF_ID_bus_r;
138-    reg [166:0] ID_EXE_bus_r;
139-    reg [153:0] EXE_MEM_bus_r;
```

```
132+    wire [170:0] ID_EXE_bus;  // ID->EXE级总线
133+    wire [155:0] EXE_MEM_bus; // EXE->MEM级总线
134     wire [117:0] MEM_WB_bus;  // MEM->WB级总线
135
136     //锁存以上总线信号
137     reg [ 63:0] IF_ID_bus_r;
138+    reg [170:0] ID_EXE_bus_r;
139+    reg [155:0] EXE_MEM_bus_r;
```

4.3.2 `decoder.v

更新总线位宽

```
    output     [166:0] ID_EXE_bus,  // ID->EXE总线
18+ output     [170:0] ID_EXE_bus,  // ID->EXE总线
19+ // 新增 divide 1 ,  inst_NXOR 1, mem_control 4位->6位,  故166->170位
20
```

```
-   assign ID_EXE_bus = {multiply,mthi,mtlo,                    //EXE需用的信息,新增
342+ assign ID_EXE_bus = {multiply,divide,mthi,mtlo,            //EXE需用的信息,新增
343                       alu_control,alu_operand1,alu_operand2,//EXE需用的信息
344+                      inst_NXOR,                            //NXOR指令标识
345                       mem_control,store_data,               //MEM需用的信号
346                       mfhi,mflo,                            //WB需用的信号,新增
347                       mtc0,mfc0,cp0r_addr,syscall,eret,     //WB需用的信号,新增
348                       rf_wen, rf_wdest,                     //WB需用的信号
349                       pc};                                  //PC值
```

新增 5 条指令

```
70+     // 新增指令
71+     wire inst_NXOR, inst_DIV  , inst_BNEZ, inst_LH;
72+     wire inst_SH;
```

对应的解码方式

```
133+    // 新增指令实现
134+    assign inst_NXOR  = op_zero & sa_zero    & (funct == 6'b111000);//异或非运算(NXOR)
135+
136+    assign inst_DIV   = op_zero & (rd==5'd0)
137+                      & sa_zero & (funct == 6'b011010);             //除法指令
138+    assign inst_BNEZ  = (op == 6'b010101) & (rt==5'd0);             //不等于0跳转
139+    assign inst_LH    = (op == 6'b100001);              //load低半字（符号扩展）
140+    assign inst_SH    = (op == 6'b101001);              //store低半字
141+
```

NXOR 实现

利用 xor 的结果

```
168+    assign inst_xor = inst_XOR | inst_XORI | inst_NXOR;    // 逻辑异或(包括异或非)
```

结果需要写入 rd

```
192     assign inst_wdest_rd = inst_ADDU | inst_SUBU | inst_SLT  | inst_SLTU
193                          | inst_JALR | inst_AND  | inst_NOR  | inst_OR
194                          | inst_XOR  | inst_SLL  | inst_SLLV | inst_SRA
195                          | inst_SRAV | inst_SRL  | inst_SRLV
    -                        | inst_MFHI | inst_MFLO;
196+                         | inst_MFHI | inst_MFLO | inst_NXOR;
```

BNEZ 实现

添加进分支跳转标志

```
147     assign inst_jbr = inst_J    | inst_JAL  | inst_jr
148                     | inst_BEQ  | inst_BNE  | inst_BGEZ
                        | inst_BGTZ | inst_BLEZ | inst_BLTZ;
    -
149+                    | inst_BGTZ | inst_BLEZ | inst_BLTZ
150+                    | inst_BNEZ;
151     //load store
```

满足不等于 0 的条件则跳转

```
229     assign br_taken = inst_BEQ  & rs_equql_rt        // 相等跳转
230                     | inst_BNE  & ~rs_equql_rt       // 不等跳转
231                     | inst_BGEZ & ~rs_ltz            // 大于等于0跳转
232                     | inst_BGTZ & ~rs_ltz & ~rs_ez   // 大于0跳转
233                     | inst_BLEZ & (rs_ltz | rs_ez)   // 小于等于0跳转
    -                   | inst_BLTZ & rs_ltz;           // 小于0跳转
234+                    | inst_BLTZ & rs_ltz            // 小于0跳转
235+                    | inst_BNEZ & ~rs_ez;           // 不等于0跳转
```

load/store 低半字实现

增加到 load/store 标志

```
155+    assign inst_load  = inst_LW | inst_LB | inst_LBU | inst_LH;  // load指令
156+    assign inst_store = inst_SW | inst_SB | inst_SH;             // store指令
```

低半字操作需要给 mem 的信号，`mem_control` 新增两位。

```
304+    wire lh_sign;  //load一半字为有符号load
305     wire ls_word;  //load/store为字节还是字,0:byte;1:word
        wire [3:0] mem_control;  //MEM需要使用的控制信号
306+    wire ls_half;  //load/store为半字,1:half
307+    wire [5:0] mem_control;  //MEM需要使用的控制信号
308     wire [31:0] store_data;  //store操作的存的数据
309     assign lb_sign = inst_LB;
310+    assign lh_sign = inst_LH;
311     assign ls_word = inst_LW | inst_SW;
312+    assign ls_half = inst_LH | inst_SH;
313     assign mem_control = {inst_load,
314                           inst_store,
315                           ls_word,
    -                      lb_sign };
316+                        ls_half,
317+                        lb_sign,
318+                        lh_sign };
```

div 需要的信号

```
270     wire multiply;          //乘法MULT
271+    wire divide;            //除法DIV
272     wire mthi;              //MTHI
273     wire mtlo;              //MTLO
274     assign multiply = inst_MULT;
275+    assign divide   = inst_DIV;
```

4.3.3 exe.v

更新总线位宽

```
    input      [166:0] ID_EXE_bus_r,// ID->EXE总线
10+ input      [170:0] ID_EXE_bus_r,// ID->EXE总线
11  output             EXE_over,    // EXE模块执行完成
    output     [153:0] EXE_MEM_bus, // EXE->MEM总线
12+ output     [155:0] EXE_MEM_bus, // EXE->MEM总线 (mem_control 4->6位，总线156位)
    wire [3:0] mem_control;  //MEM需要使用的控制信号
32+ wire [5:0] mem_control;  //MEM需要使用的控制信号
```

从总线中取数据，新增两个信号，且 `mem_control` 位宽改变

```
    wire [3:0] mem_control;  //MEM需要使用的控制信号
32+ wire [5:0] mem_control;  //MEM需要使用的控制信号
```

```
24+ wire divide;             //除法
```

```
30+ wire        inst_nxor;   //NXOR指令标识
```

```
48      assign {multiply,
49+             divide,
50              mthi,
51              mtlo,
52              alu_control,
53              alu_operand1,
54              alu_operand2,
55+             inst_nxor,
56              mem_control,
57              store_data,
58              mfhi,
59              mflo,
60              mtc0,
61              mfc0,
62              cp0r_addr,
63              syscall,
64              eret,
65              rf_wen,
66              rf_wdest,
67              pc          } = ID_EXE_bus_r;
```

同或 NXOR

利用 xor 的结果完成 NXOR

```
129+    wire [31:0] final_alu_result;
130+    assign final_alu_result = inst_nxor ? ~alu_result : alu_result;
```

除法 div

除法器简单实现

```
96+
97+ //-----{除法器}begin
98+     wire [31:0] quotient;  // 商
99+     wire [31:0] remainder; // 余数
100+    wire        div_by_zero;
101+
102+    assign div_by_zero = (alu_operand2 == 32'd0);
103+    assign quotient    = div_by_zero ? 32'hFFFFFFFF :
104+                         (alu_operand1[31] == alu_operand2[31]) ?
105+                         (alu_operand1 / alu_operand2) :
106+                         (~(alu_operand1 / alu_operand2) + 1'b1);
107+    assign remainder   = div_by_zero ? alu_operand1 : (alu_operand1 % alu_operand2);
108+ //-----{除法器}end
```

像乘法一样，除法的结果也需要放到 `lo`、`hi` 寄存器中

```
-                       multiply ? product[63:32] : alu_result;
-     assign lo_result  = mtlo ? alu_operand1 : product[31:0];
-     assign hi_write   = multiply | mthi;
      assign lo_write   = multiply | mtlo;
134+                    multiply ? product[63:32] :
135+                    divide   ? remainder : final_alu_result;
136+    assign lo_result  = mtlo     ? alu_operand1 :
137+                        multiply ? product[31:0] :
138+                        divide   ? quotient : 32'd0;
139+    assign hi_write   = multiply | divide | mthi;
140+    assign lo_write   = multiply | divide | mtlo;
```

4.3.4 `mem.v`

更新总线位宽

```
      input       [153:0] EXE_MEM_bus_r,// EXE->MEM总线
11+   input       [155:0] EXE_MEM_bus_r,// EXE->MEM总线
12+   // (增加2位: mem_control 4->6位)
```

从 `mem_control` 中取数据，新增了两个信号 `ls_half` 、`lh_sign`

```
    wire [3 :0] mem_control;   //MEM需要使用的控制信号
29+ wire [5 :0] mem_control;   //MEM需要使用的控制信号
```

```
    wire ls_word;     //load/store为字节还是字,0:byte;1:word
72+ wire ls_word;     //load/store为字节还是字,0:byte;1:word  (但是 half优先级更高)
73+ wire ls_half;     //load/store为半字操作
74  wire lb_sign;     //load一字节为有符号load
    assign {inst_load,inst_store,ls_word,lb_sign} = mem_control;
75+ wire lh_sign;     //load一半字为有符号load
76+ assign {inst_load,inst_store,ls_word,ls_half,lb_sign,lh_sign} = mem_control;
```

load 低半字

添加 load 低半字的逻辑，原来只需要分成两部分（`[31:8]、[7:0]`）来读，由于添加了 load 低半字，需要分成三部分。

```
137 //load读出的数据
138 wire        load_sign;
139 wire [31:0] load_result;
    assign load_sign = (dm_addr[1:0]==2'd0) ? dm_rdata[ 7] :
140+ assign load_sign = ls_half ? dm_rdata[15] :                    // 半字操作: 看低半字符
141+                    (dm_addr[1:0]==2'd0) ? dm_rdata[ 7] :       // 字节操作
142                     (dm_addr[1:0]==2'd1) ? dm_rdata[15] :
143                     (dm_addr[1:0]==2'd2) ? dm_rdata[23] : dm_rdata[31] ;
    assign load_result[7:0] = (dm_addr[1:0]==2'd0) ? dm_rdata[ 7:0 ] :
144+
145+ assign load_result[7:0] = ls_half ? dm_rdata[7:0] :           // 半字操作: 取低半字的低
146+                           (dm_addr[1:0]==2'd0) ? dm_rdata[ 7:0 ] :  // 字节操作
147                            (dm_addr[1:0]==2'd1) ? dm_rdata[15:8 ] :
148                            (dm_addr[1:0]==2'd2) ? dm_rdata[23:16] :
149                                                  dm_rdata[31:24] ;
    assign load_result[31:8]= ls_word ? dm_rdata[31:8] : {24{lb_sign & load_sign}};
150+ assign load_result[15:8] = ls_half ? dm_rdata[15:8] :          // 半字操作: 取低半字的
151+                           ls_word ? dm_rdata[15:8] :          // 字操作: 直接取
152+                           {8{lb_sign & load_sign}};           // 字节操作: 符号扩展
153+
154+ assign load_result[31:16] = ls_half ? {16{lh_sign & load_sign}} : // 半字操作: 符号扩展
155+                           ls_word ? dm_rdata[31:16] :         // 字操作: 直接取高半字
156+                           {16{lb_sign & load_sign}};          // 字节操作: 符号扩展
```

store 低半字

写使能，添加判断 load 半字的逻辑且判断优先级置为最高（因为 is_word 用的是 0 表示字节、1 表示字的含义，因此需要先判断是否在写半字）。

```
84      if (MEM_valid && inst_store) // 访存级有效时,且为store操作
85      begin
-           if (ls_word)
86+         if (ls_half) // SH指令，半字写使能 - 只操作低半字
87+         begin
88+             if (dm_addr[1:0] == 2'b00) // 地址必须是2字节对齐
89+                 dm_wen <= 4'b0011; // 只写低半字 (bits [15:0])
90+             else
91+                 dm_wen <= 4'b0000; // 地址不对齐，不写
92+         end
93+         else if (ls_word) // SW指令，字写使能
94          begin
```

先添加 load 低半字的逻辑，优先级置为最高。

```
114     //store操作的写数据
-       always @ (*)  // 对于SB指令，需要依据地址底两位，移动store的字节至对应位置
115+    always @ (*)  // 对于SB/SH指令，需要依据地址底两位，移动store的数据至对应位置
116+    begin
117+        if (ls_half) // SH指令 - 只操作低半字
118+        begin
119+            dm_wdata <= {16'd0, store_data[15:0]};  // 存到低半字位置
120+        end
121+        else if (ls_word) // SW指令
122+        begin
123+            dm_wdata <= store_data; // 字操作，直接存储
124+        end
125+        else // SB指令
126         begin
127             case (dm_addr[1:0])
128                 2'b00   : dm_wdata <= store_data;
129                 2'b01   : dm_wdata <= {16'd0, store_data[7:0], 8'd0};
130                 2'b10   : dm_wdata <= {8'd0, store_data[7:0], 16'd0};
131                 2'b11   : dm_wdata <= {store_data[7:0], 24'd0};
132                 default : dm_wdata <= store_data;
133             endcase
134+         end
```

5 实验结果分析

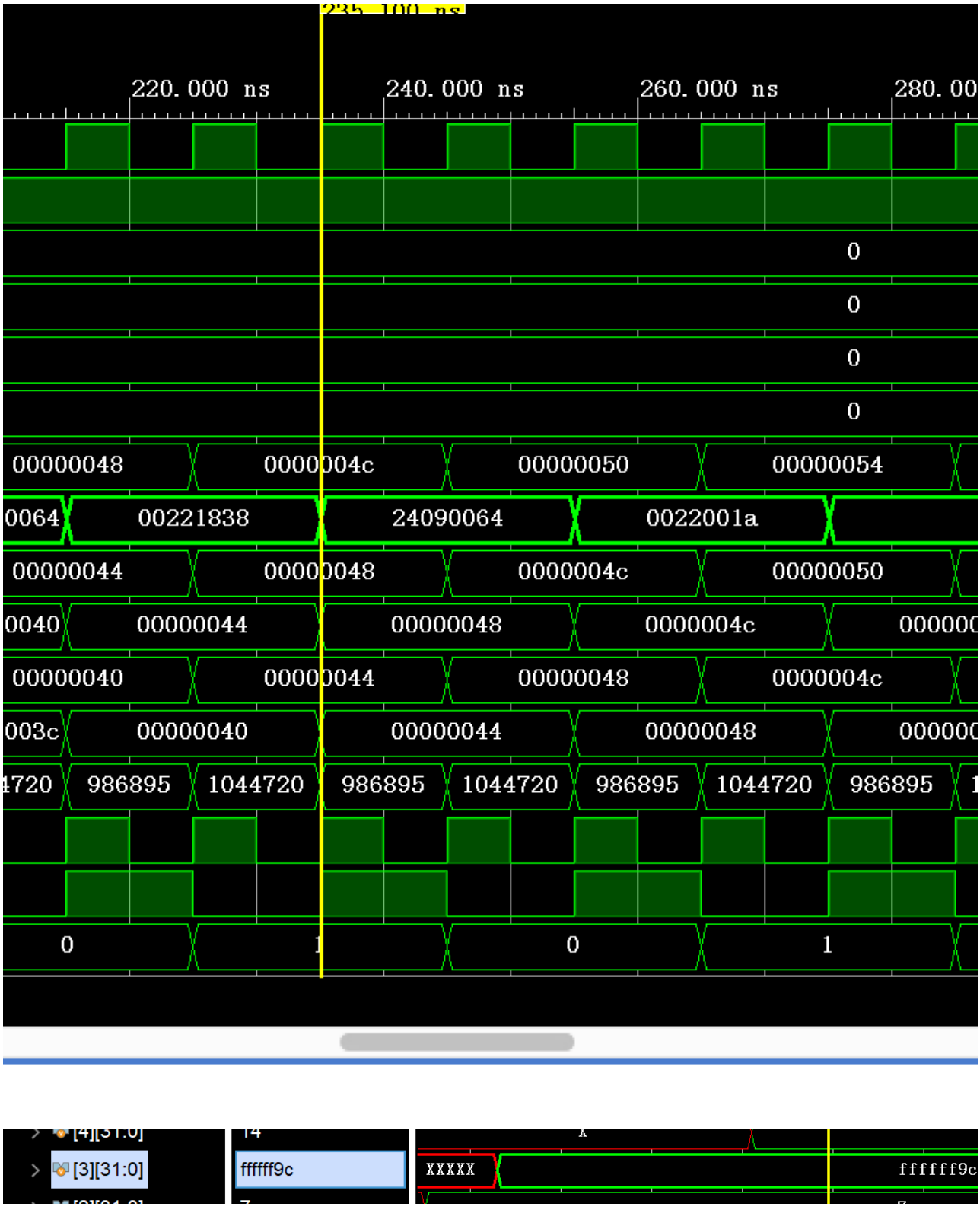
（仿真结果截图或者实验箱运行结果拍照，注意需要对实验结果进行分析，输入是什么，输出是什么，结果是什么，是否验证了正确性）

测试的 5 个指令

```
nxor $3, $1, $2: 0x00221838
div $1, $2: 0x0022001A
sh $4, #0($6): 0xA5240000
lh $7, #0($6): 0x85270000
bnez $1, #-11: 0x5420FFF5
会提前使用
addiu $1, $zero, 100: 0x24010064
addiu $2, $zero, 7: 0x24020007
addiu $9, $zero, 0x100: 0x24090064
初始化三个寄存器
```

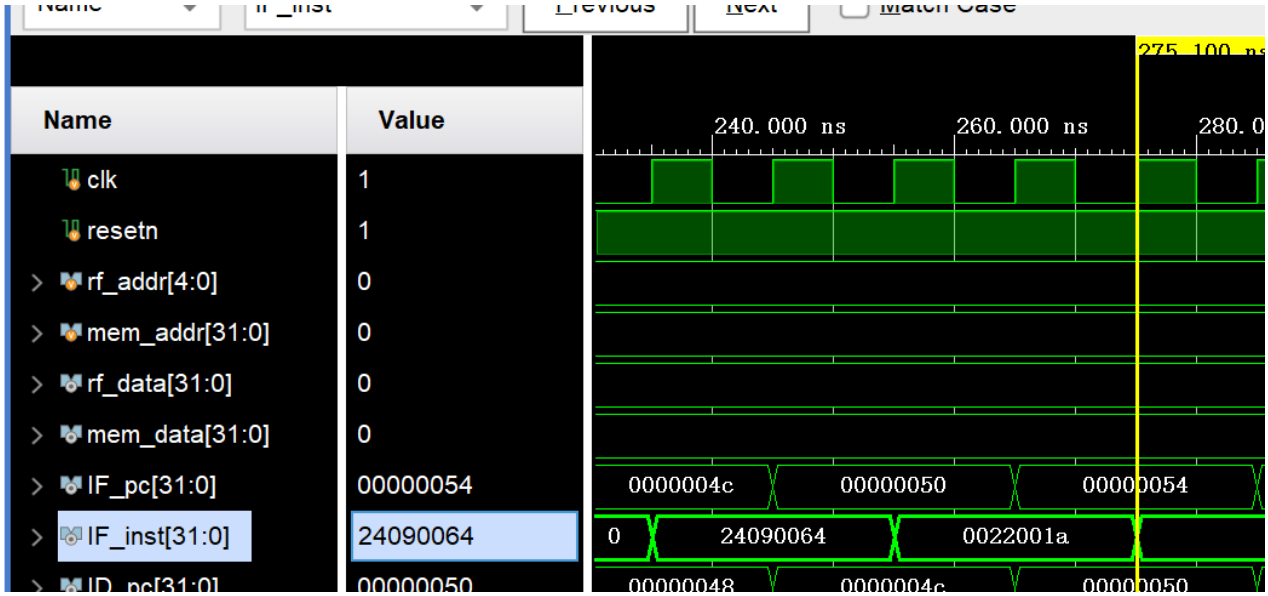
5.1 NXOR

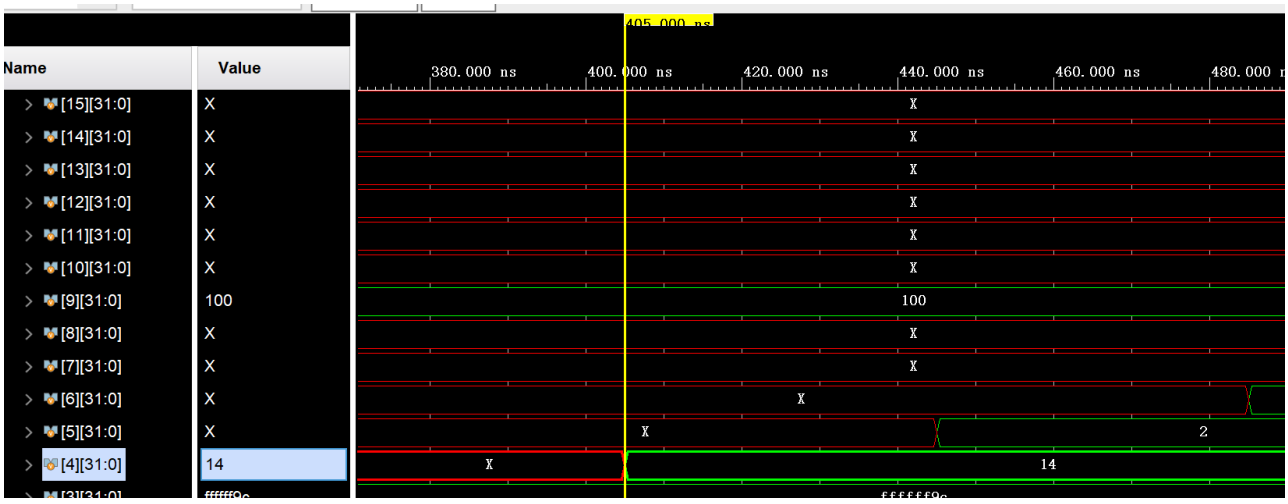
可以看到 0x64 与 0x7 做同或，结果是 0xFFFF_FF9C。通过计算器验证，结果正确。



5.2 DIV

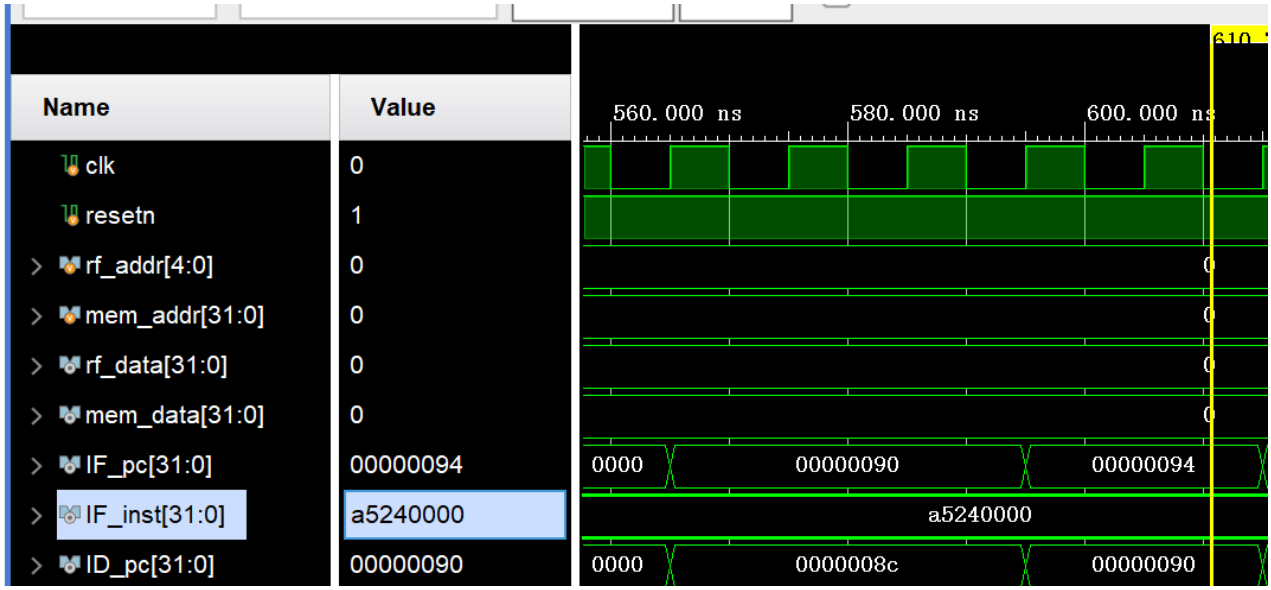
除法，使用 100%7 = 14 余 2，结果正确。

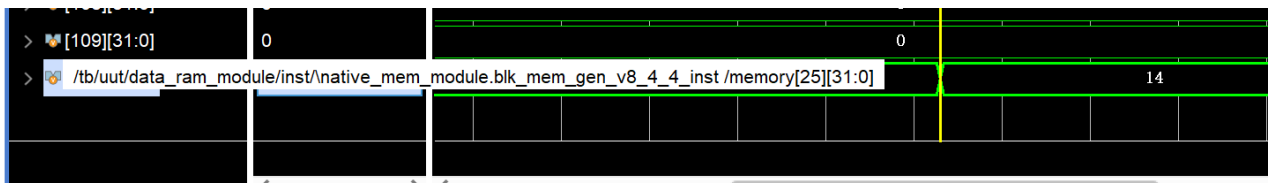




5.3 SH

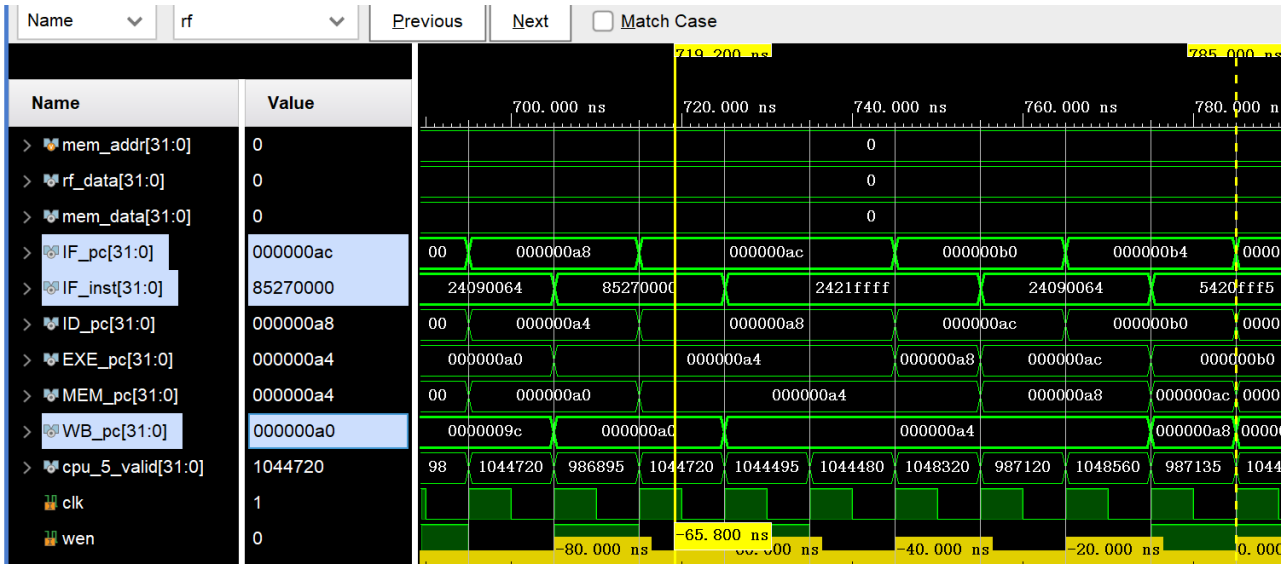
将 `rf[4]`，即之前除法的商 14，存到 ram IP 核 memoy 的相应位置（地址为 100）了。





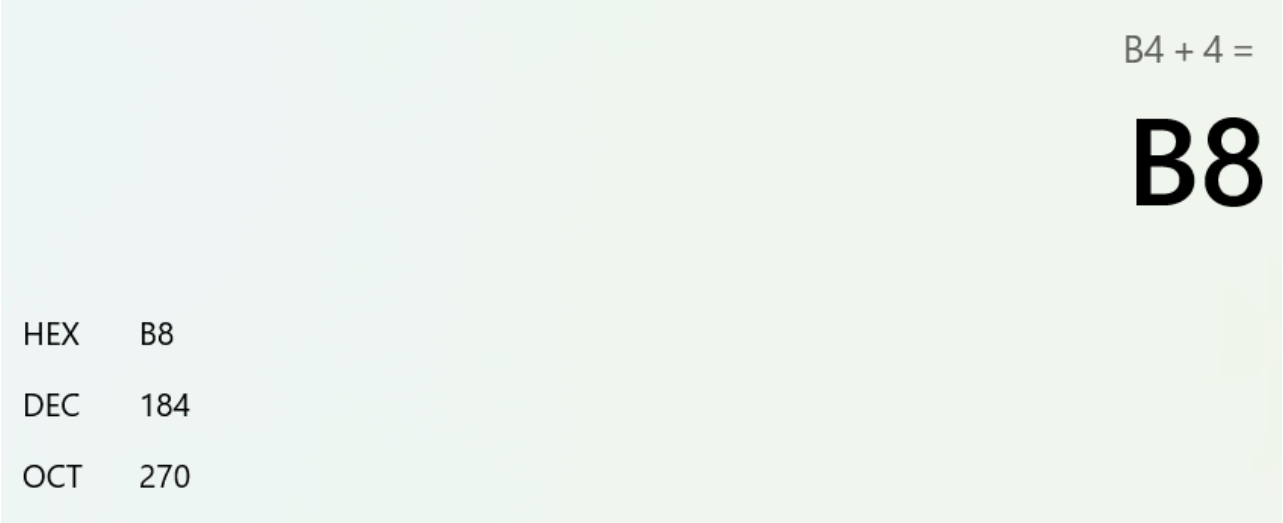
5.4 LH

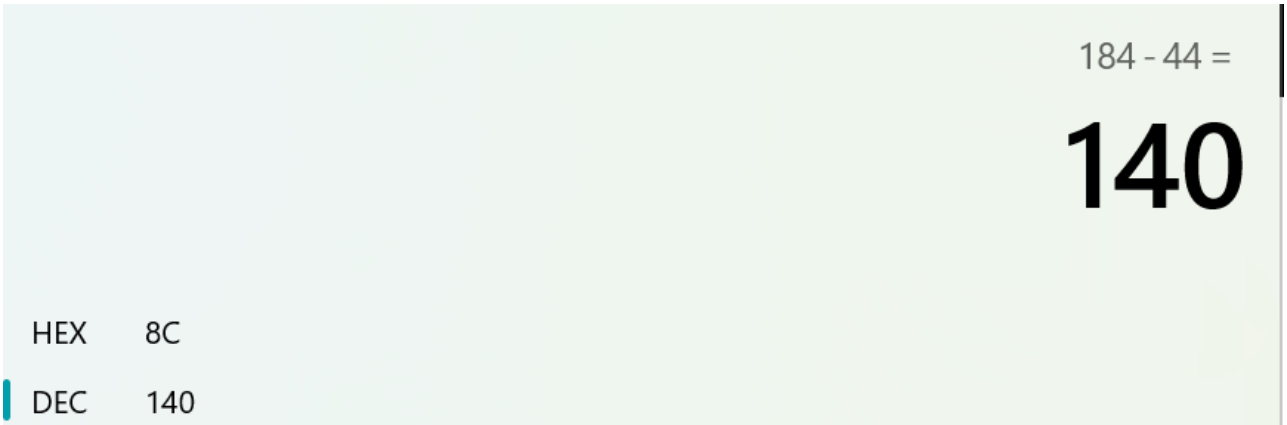
将内存地址 100 处的 14 再次取到 7 号寄存器。



5.5 BNEZ

使用 1 号寄存器 `$1=100` 进行比较，它不等于 0，于是跳转，`b4+4-b*4=8c`，跳转到了正确的地址。





6 总结感想

五级流水线结束，将书上的五级流水线实现出来真是困难重重，简单的理念，进行实现时却面临着相当大的复杂度。模块化对于添加新指令是比较友好的，算是降低了一点复杂度。五级流水线相比多周期 CPU，面临的主要难点是指令间的依赖与冲突，解决这些难点较为麻烦，也会增加复杂度。

全部实验结束。在一定程度上感受了 FPGA 开发和芯片制作这些硬件层面的工作相较软件开发的困难之处。无论是开发工具还是调试上，硬件都比软件更困难。特别是由于硬件层面大多语句是并行的，更增大了调试难度，这在五级流水线上体现得淋漓尽致。