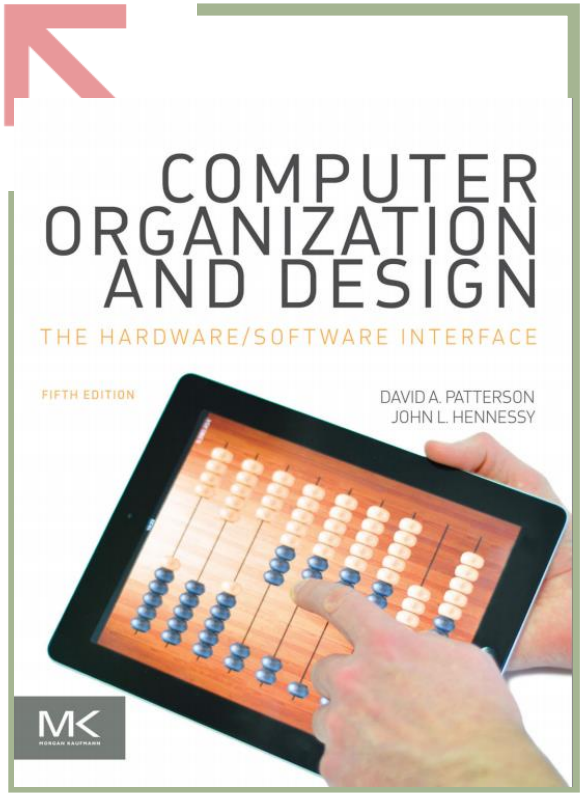
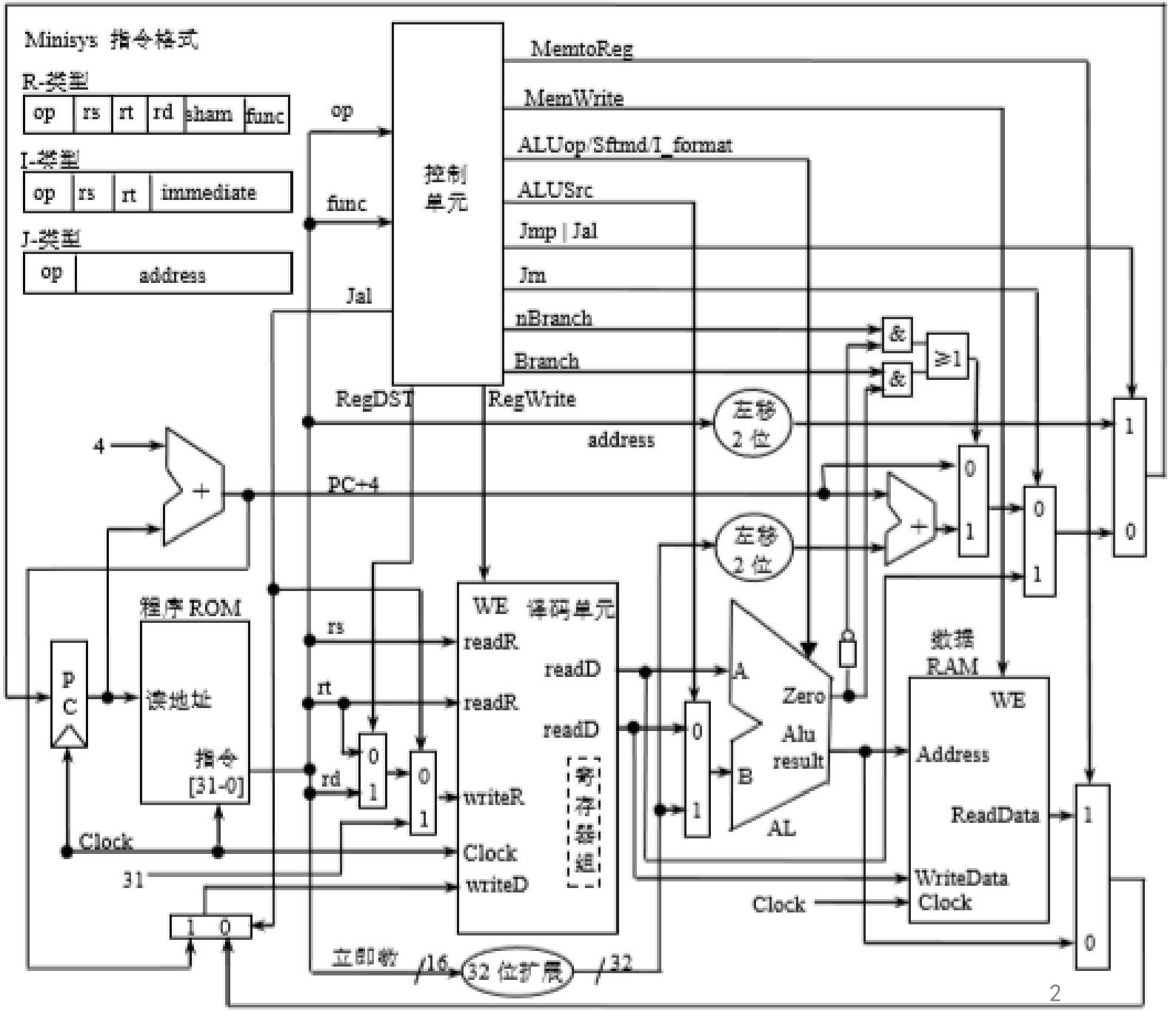
取址单元及控制单元的设计



单周期CPU
数据通路回顾



取址单元设计

1. 定义指令ROM存储器

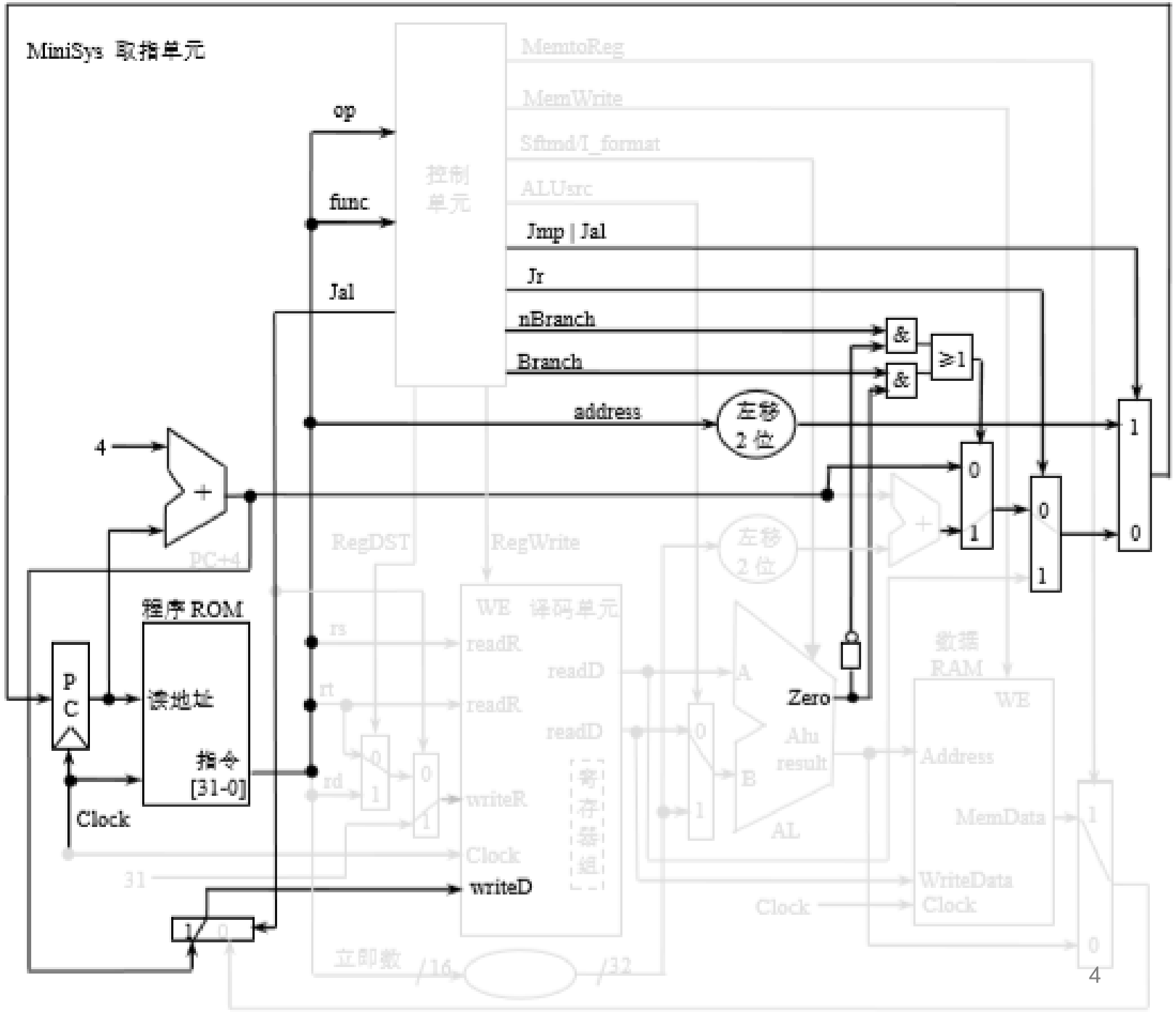
2. 到指令ROM中取指令

3. 对PC值进行＋4处理

4. 完成几种跳转指令的PC修改功能

5. 最终修改PC值

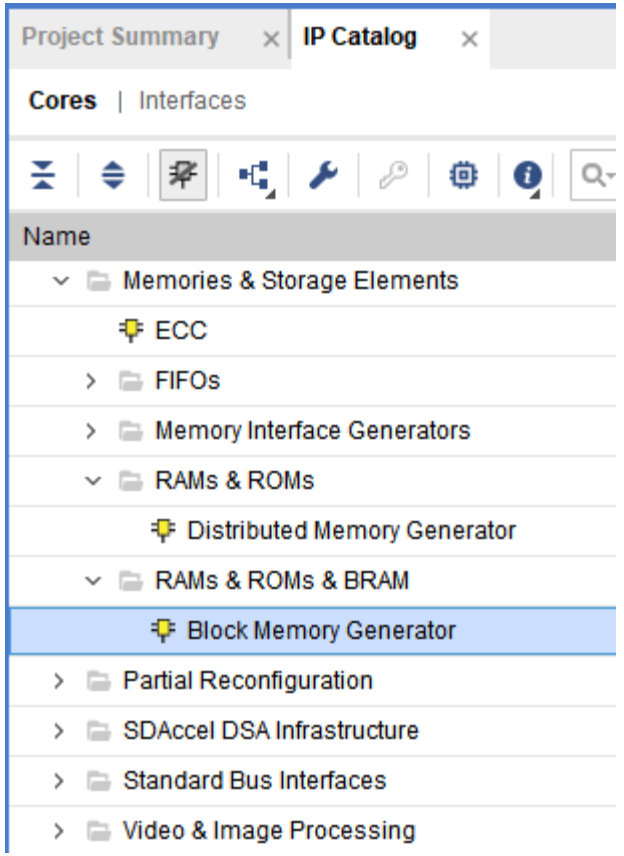
取址单元设计


MiniSys 取指单元
op
func
Jal
控制单元
MemtoReg
MemWrite
Sfimd/I_format
ALUsrc
Jmp | Jal
Jr
nBranch
Branch
address
左移 2 位
&
&
≥1
4
+
PC+4
RegDST
RegWrite
程序 ROM
P C
读地址
指令 [31-0]
Clock
rs
rt
rd
writeR
Clock
31
writeD
WE 译码单元
readR
readR
readD
readD
寄存器组
立即数
/16
/32
Zero
A
B
Alu result
AL
数据 RAM
WE
Address
MemData
WriteData
Clock
Clock
0
1

取址单元设计

定义指令ROM

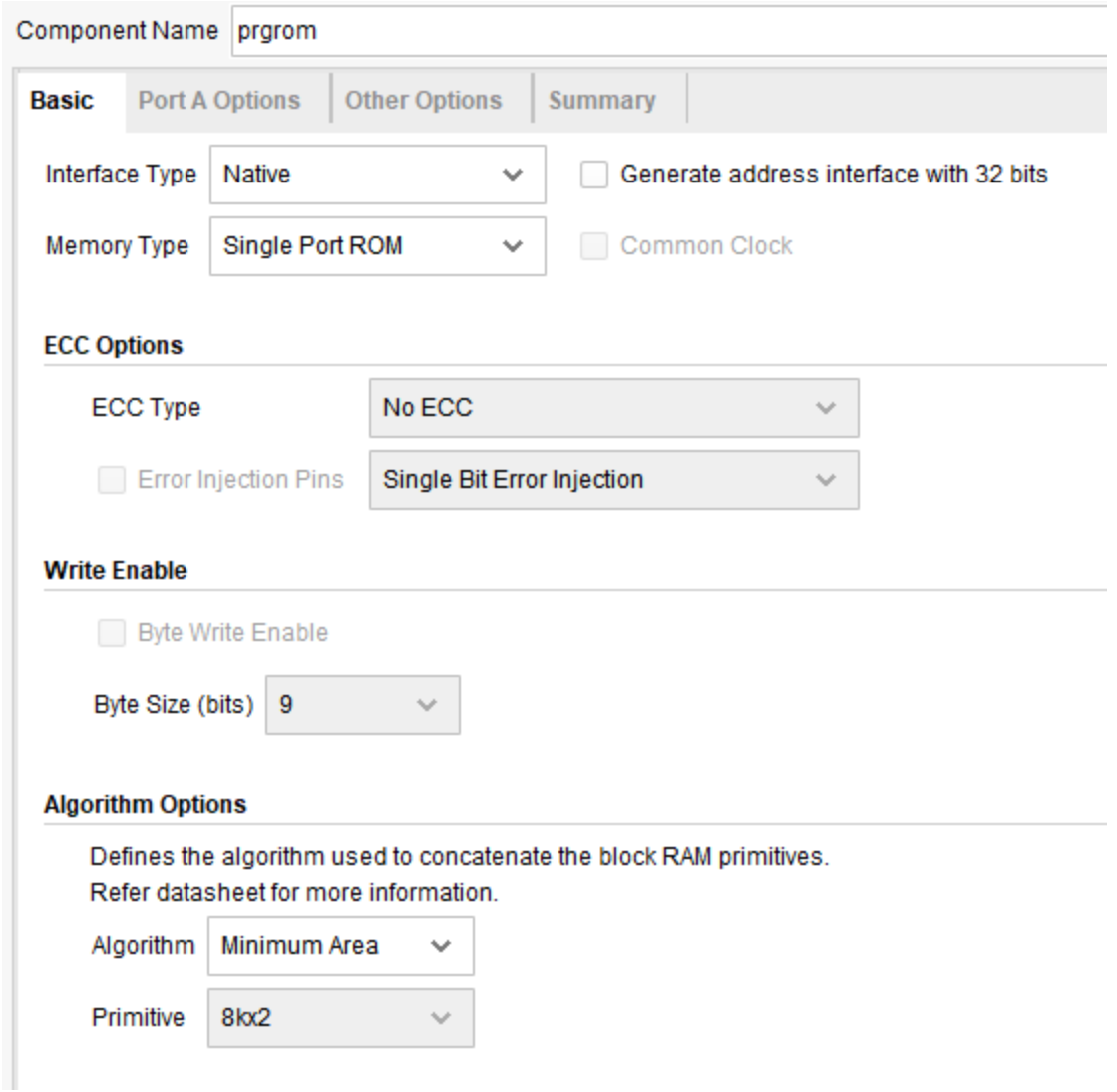
IP Catalog > Memories & Storage Elements > RAMs & ROMs & BRAM> Block Memory Generator



取址单元设计

定义指令ROM

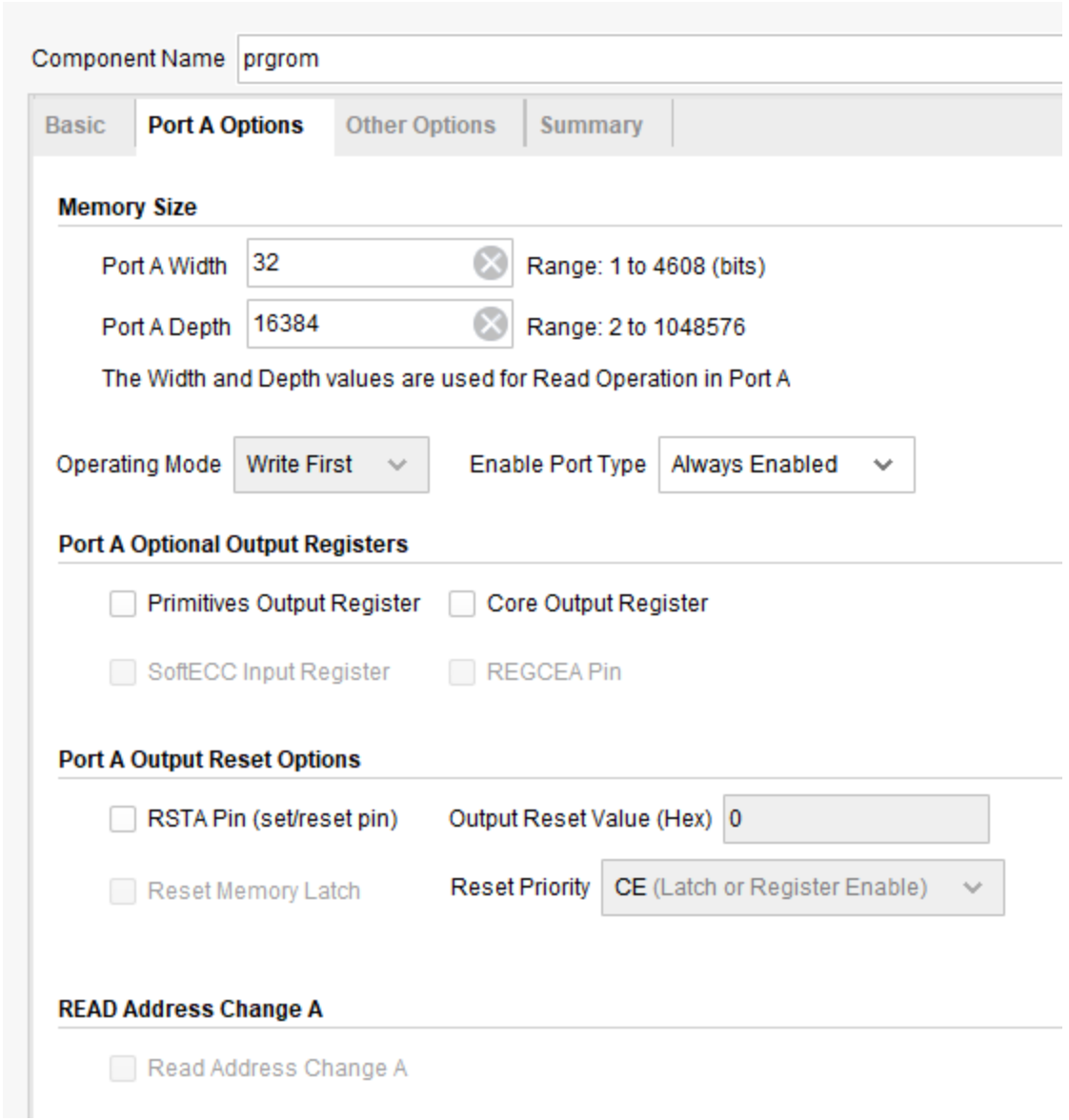
部件名称prgprom，Native
单端口ROM, 不要 ECC校验，
最小面积算法



取址单元设计

定义指令ROM

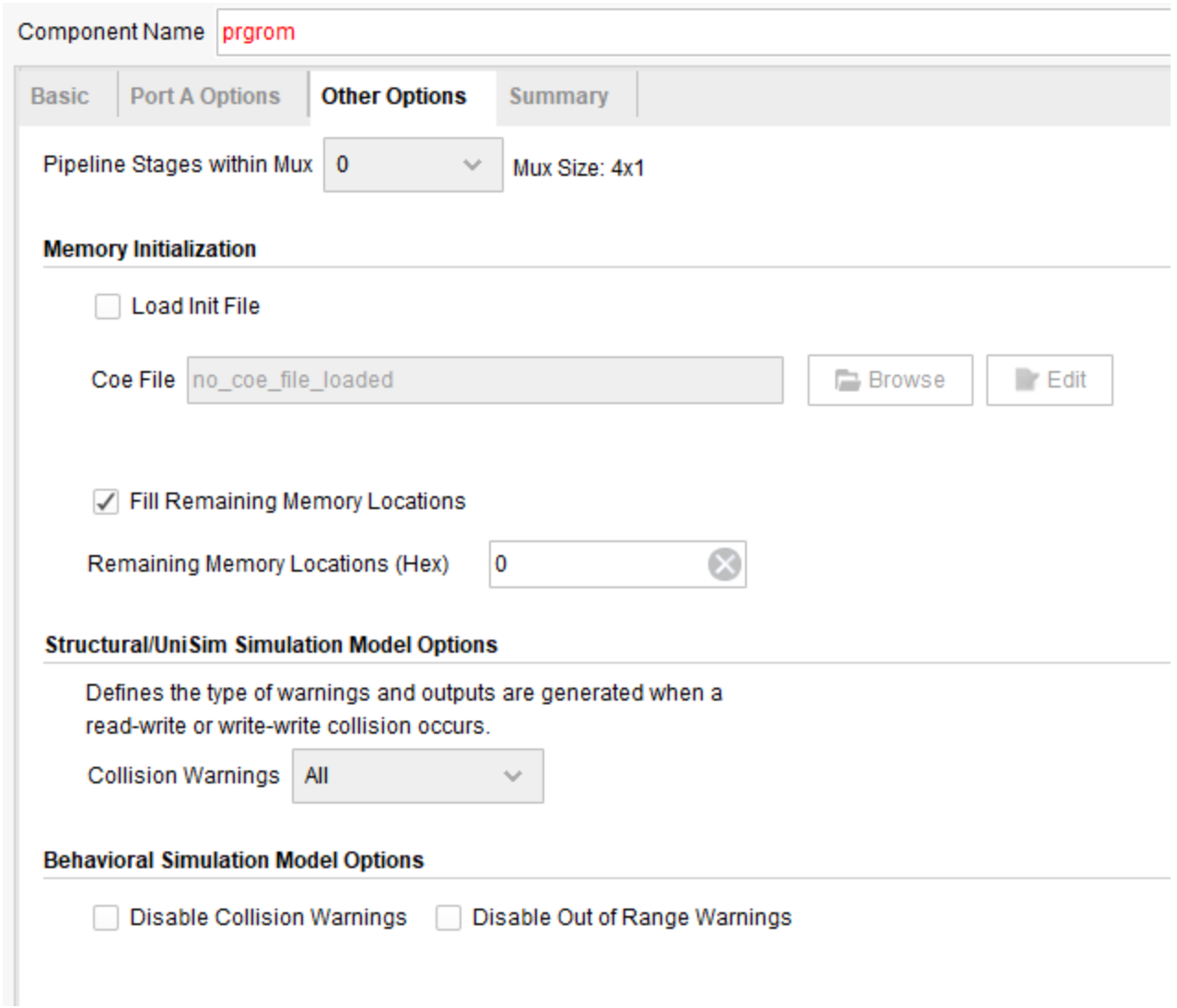
创建 64KB 的 ROM，数据宽度 32 位，要 16384 个数据单元，地址线 14 根，始终使能，写优先。



取址单元设计

定义指令ROM

在创建 prgrom 的时候先设置成没有初始文件，点击OK，并生成（Generater）IP 核



创建存储器初始化文件prgmip32.coe

.coe 文件需要有一定的格式。

```
memory_initialization_radix = 16;   // 表明是 16 进制形式
memory_initialization_vector =      // 下面放数据，要放满
00000000,
……                                    // 正式文档要用数据填满
00000000,
```

16384 个数据单元

创建存储器初始化文件prgmip32.coe

创建好prgrom后，创建存储器初始化文件prgmip32.coe并拷贝到minisys/minisys.srcs/sources_1/ip/prgrom/中

双击刚建立的prgrom IP核，重新设置其为有初始化文件，并选择已经拷贝好的prgmip32.coe文件。

Ifetc32.v中的ROM元件例化

```
//分配64KB ROM，编译器实际只用 64KB ROM
prgrom instmem(
    .clka(clock),          // input wire clka
    .addra(PC[15:2]),      // input wire [13 : 0] addra
    .douta(Instruction)         // output wire [31 : 0] douta
);
```

练习：完成对 PC+4 的处理

```
output[31:0] PC_plus_4_out;    // (pc+4)送执行单元
wire[31:0]   PC_plus_4;

assign PC_plus_4[31:2] =? ? ?
assign PC_plus_4[1:0] =? ? ?
assign PC_plus_4_out = PC_plus_4[31:0];
//   PC＋4 送到执行单元，以便执行单元在必要的时候算出 ADDRESULT
```

练习：完成 beq，bne，jr 跳转对 PC 的修改

```
input[31:0]   Add_result;     // 来自执行单元,算出的跳转地址
input[31:0]   Read_data_1; // 来自译码单元，jr 指令用的地址
input         Branch;                // 来自控制单元
input         nBranch;               // 来自控制单元
input         Jrn;        //来自控制单元
input         Zero;       //来自执行单元
reg[31:0]     next_PC;   // 下条指令的 PC（不一定是 PC+4）

always @* begin      // beq, bne, jr
……? ? ? ? ? ……    // 请考虑以上三条指令的判断条件，
// 以及三条指令的执行该给 next_PC 赋什么值
end
```

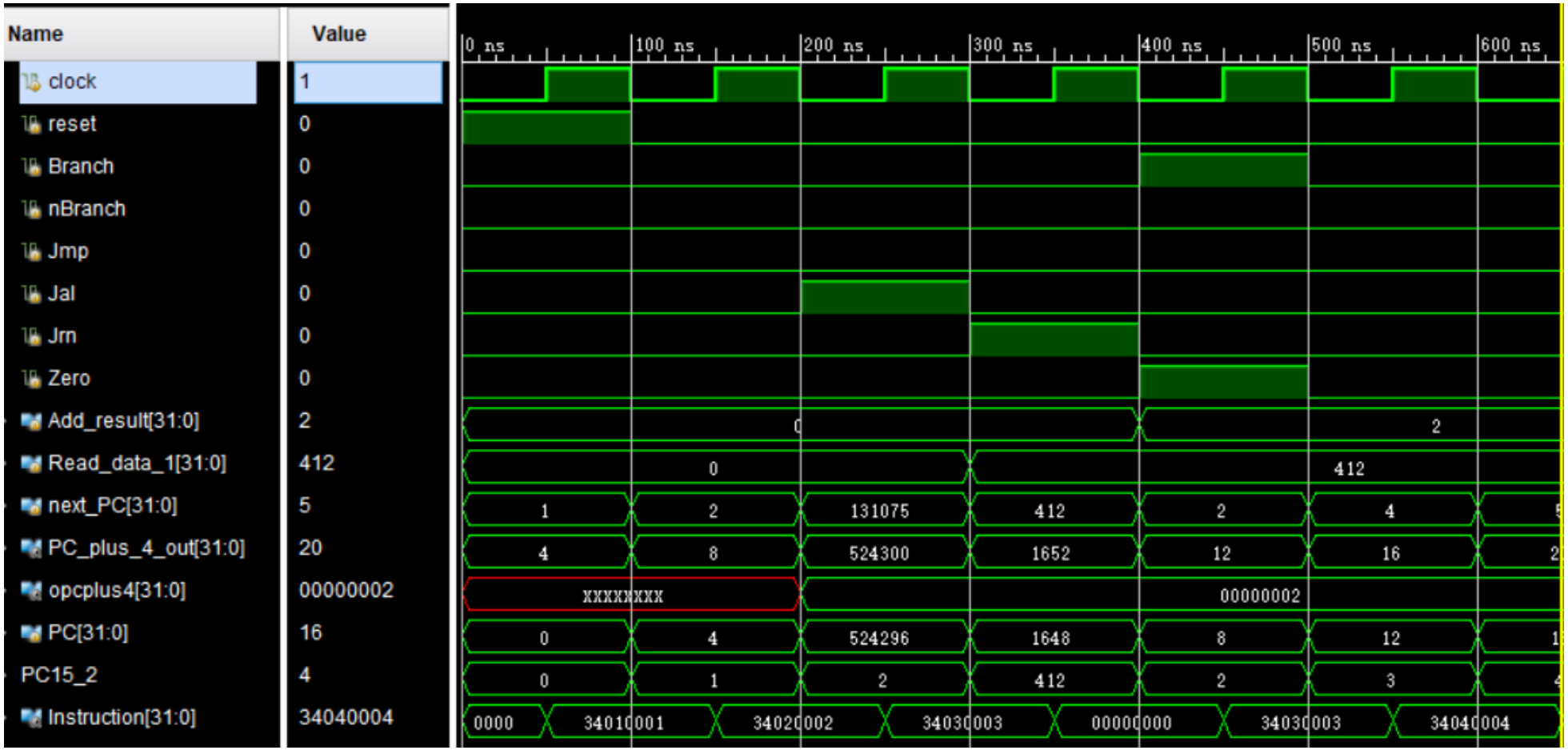
练习：完成 PC 的最终修改（含 J，Jal 指令和 reset 的处理）

```
input         Jmp;       //来自控制单元
input         Jal;       //来自控制单元
output[31:0] opcplus4;  // JAL 指令专用的 PC+4
reg[31:0]     opcplus4;

always @(negedge clock) begin
……   ? ? ……
end
```

仿真参考波形

使用sakai站点中lab12下的仿真文件，结合完成的取址单元进行功能验证，参考波形如下：



测试

- 请根据不同类型的输入信号，确定对应的nextPc，pcPlus4 是否正确
- 请打开coe文件，以pc[15:2]为地址，确认是否从rom中正确读取到该地址对应的指令地址

time(ns)	与指令跳转相关的操作
100	reset信号失效
200	jal // 跳转到 label对应的地址
300	j $rs // 待跳转的指令地址为寄存器读取的数值：32'h0000019c
400	beq //待跳转的指令由执行单元计算得到：32'h00000002
500	非跳转指令，普通指令

prgmip32.coe

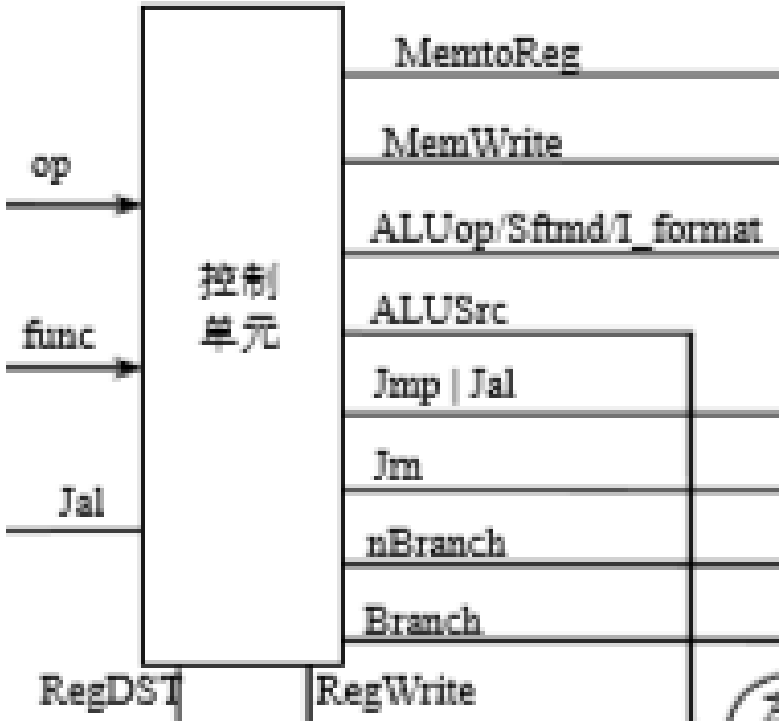
```
memory_initialization_radix = 16;
memory_initialization_vector =
34010001,
34020002,
34030003,
34040004,
34050005,
34060006,
34070007,
34080008,
34090009,
340a000a,
340b000b,
340c000c,
340d000d,
340e000e,
340f000f,
34100010,
34110011,
34120012,
34130013,
```

控制单元设计

设计方法：

考察每条指令在数据通路中的执行过程和涉及到的控制信号的取值

根据列出的指令和控制信号之间的关系，写出每个控制信号的逻辑表达式。



控制单元设计

control32.v 文件中给出了控制单元的 Verilog HDL 程序中信号定义部分

```
module control32(Opcode,Function_opcode,Jrn,RegDST,ALUSrc,MemortoReg,RegWrite,MemWrite,Brar
    input[5:0]   Opcode;           // 来自取指单元instruction[31..26]
    input[5:0]   Function_opcode;  // 来自取指单元r-类型 instructions[5..0]
    output       Jrn;              // 为1表明当前指令是jr
    output       RegDST;           // 为1表明目的寄存器是rd，否则目的寄存器是rt
    output       ALUSrc;           // 为1表明第二个操作数是立即数（beq，bne除外）
    output       MemtoReg;         // 为1表明需要从存储器读数据到寄存器
    output       RegWrite;         // 为1表明该指令需要写寄存器
    output       MemWrite;         // 为1表明该指令需要写存储器
    output       Branch;           // 为1表明是Beq指令
    output       nBranch;          // 为1表明是Bne指令
    output       Jmp;              // 为1表明是J指令
    output       Jal;              // 为1表明是Jal指令
    output       I_format;         // 为1表明该指令是除beq，bne，LW，SW之外的其他I-类型指令
    output       Sftmd;            // 为1表明是移位指令
    output[1:0]  ALUOp;            // 是R-类型或I_format=1时位1为1，beq、bne指令则位0为1
```

控制单元设计

ALU 的控制电路设计

ALU采用分级控制的方法（减轻控制器负担），在控制器中，只发出ALUop信号

指令操作码	ALUop
LW	00
SW	00
BEQ,BNE	01
R-format	10
I-format	10

```
assign ALUOp = {(R_format || I_format),(Branch || nBranch)};  // 是R-type或需要立即数作32位扩展的指令1位为1，beq、bne指令则0位为1
```

控制单元设计

RegDST 的控制电路设计

op→	001101	001001	100011	101011	000100	000010	000000
指令操作码	ori	addiu	lw	sw	beq	j	R-format
RegDST	0	0	0	x	x	x	1

```
assign R_format = (Opcode==6'b000000)? 1'b1:1'b0;        //=00h
assign RegDST = R_format;                                //说明目标是rd，否则是rt
```

控制单元设计

op→	001xxx	000000	100011	101011	000011	000010	000000
指令操作码	I-format	jr	lw	sw	jal	j	R-format
RegWrite	1	0	1	x	1	x	1

练习1：根据以上表格的提示，在此基础上完成以下控制信号电路的verilog描述

```
assign I_format = ?????
assign Lw = ?????
assign Jal = ?????
assign Jrn = ?????
assign RegWrite = ?????
```

控制单元设计

练习2：请给出以下控制信号电路的 Verilog 描述

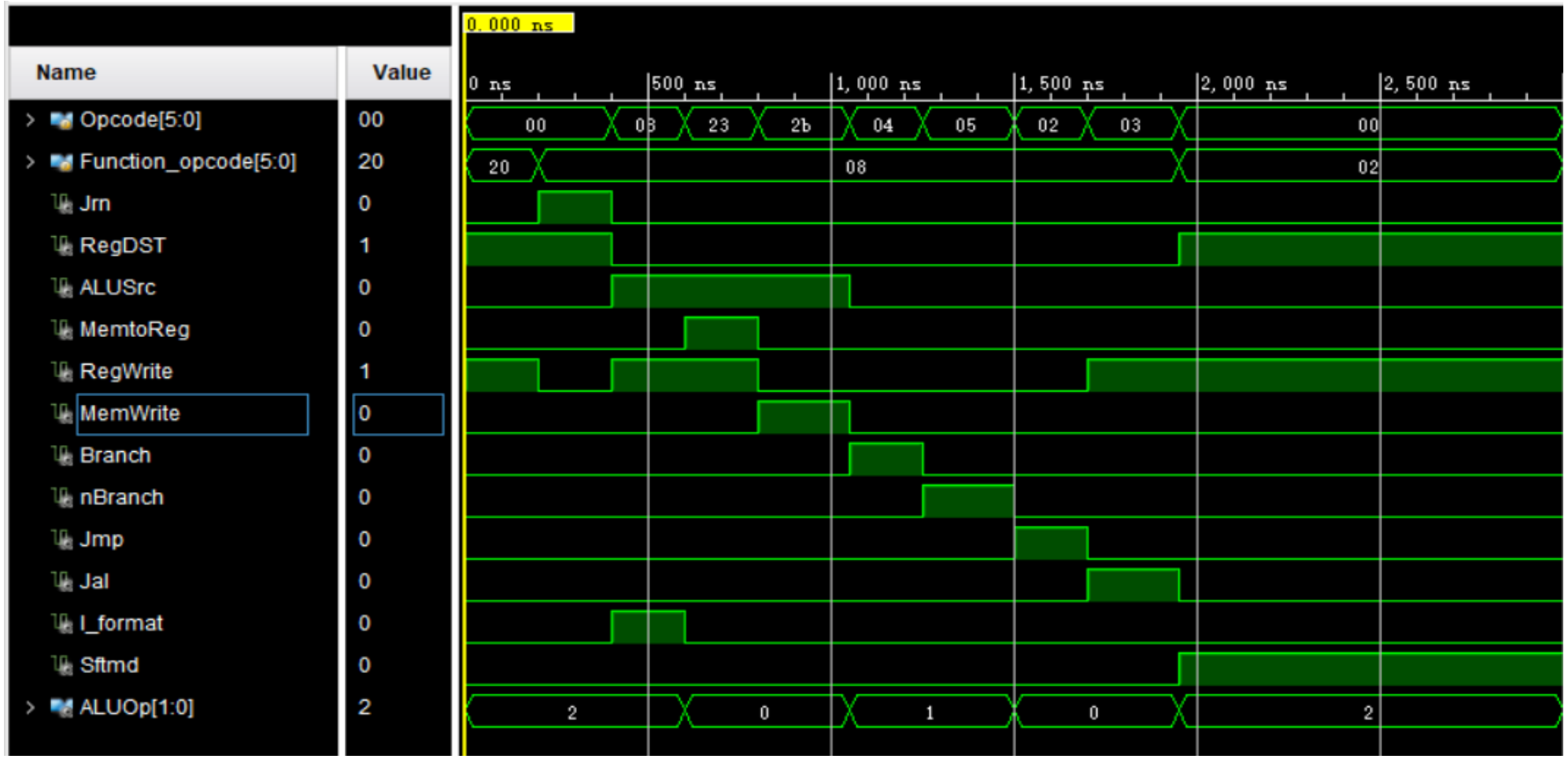
```
output      ALUSrc;    // 为 1 表明第二个操作数是立即数（beq，bne 除外）
output      MemtoReg; //  为 1 表明需要从存储器读数据到寄存器
output      MemWrite; //  为 1 表明该指令需要写存储器
output      Branch;       //  为 1 表明是 Beq 指令
output      nBranch;      //  为 1 表明是 Bne 指令
output      Jmp;          //  为 1 表明是 J 指令
output      Sftmd;        //  为 1 表明是移位指令

assign Sw = ? ? ? ? ?
assign ALUSrc = ? ? ? ? ?
assign Branch = ? ? ? ? ?
assign nBranch = ? ? ? ? ?
assign Jmp = ? ? ? ? ?

assign MemWrite = ? ? ? ? ?
assign MemtoReg = ? ? ? ? ?
assign Sftmd = ? ? ? ? ?
```

参考波形

使用sakai站点中lab12下的仿真文件，结合完成的控制单元进行功能验证，参考波形如下：



参考波形

time(ns)	opcode	function_opcode	instruction
0	6'h00	6’h20	add rd,rs,rt //RegDST=1，RegWrite=1，ALUSrc=0，ALUOp=10
200	6'h00	6’h08	jr rs //RegDST=1，RegWrite=0，ALUSrc=0，ALUOp=10，jrn=1，
400	6'h08	6’h08	addi rt,rs,imm //RegDST=0，RegWrite=1，ALUSrc=1， I_format=1
600	6'h23	6’h08	lw rt,imm(rs) //RegDST=0，RegWrite=1，ALUSrc=1，ALUOp=00，MemtoReg=1
800	6'h2b	6’h08	sw rt,imm(rs) //RegDST=0，RegWrite=0，ALUSrc=1，ALUOp=00，MemtoReg=0, MemWrite=1
1050	6'h04	6’h08	beq rs,rt,label //RegDST=0，RegWrite=0，ALUSrc=0，ALUOp=01，Branch=1
1250	6'h05	6’h08	bne rs,rt,label //RegDST=0，RegWrite=0，ALUSrc=0，ALUOp=01，Branch=0, nBranch=1
1500	6'h02	6’h08	j label //RegDST=0，RegWrite=0，ALUSrc=0，ALUOp=00，Branch=0, nBranch=0, Jmp=1
1700	6'h03	6’h08	jal label //RegDST=0，RegWrite=1，ALUSrc=0，ALUOp=00，Branch=0, nBranch=0, Jmp=0, Jal=1
1950	6'h00	6'h02	srl rd,rt,shamt //RegDST=1，RegWrite=1，ALUSrc=0，ALUOp=10，sftmd=1