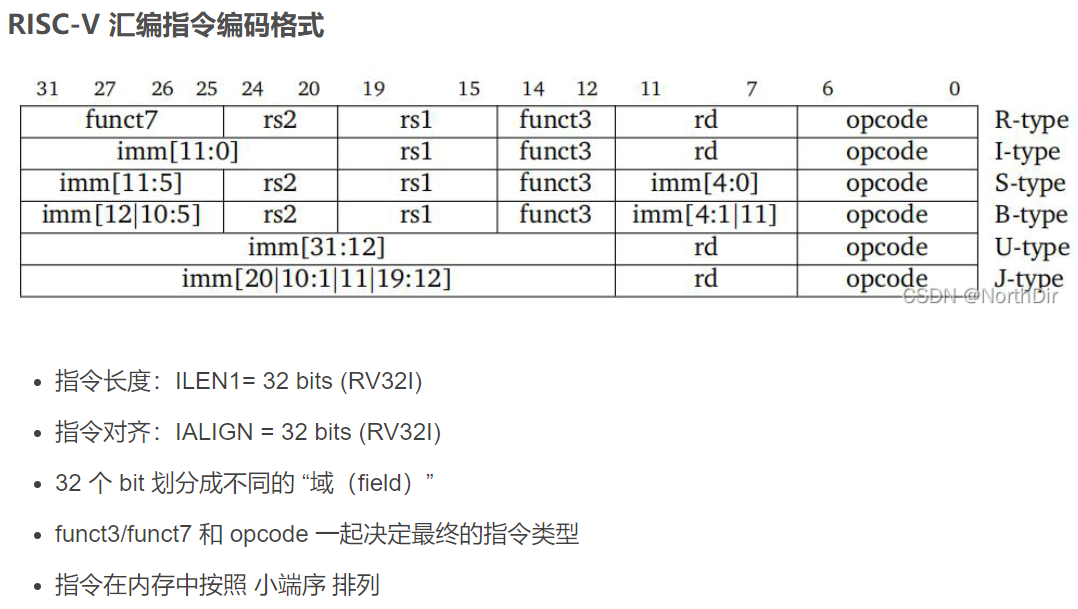
指令

RV32I

根据立即数在指令中的位置分布对指令进行分类

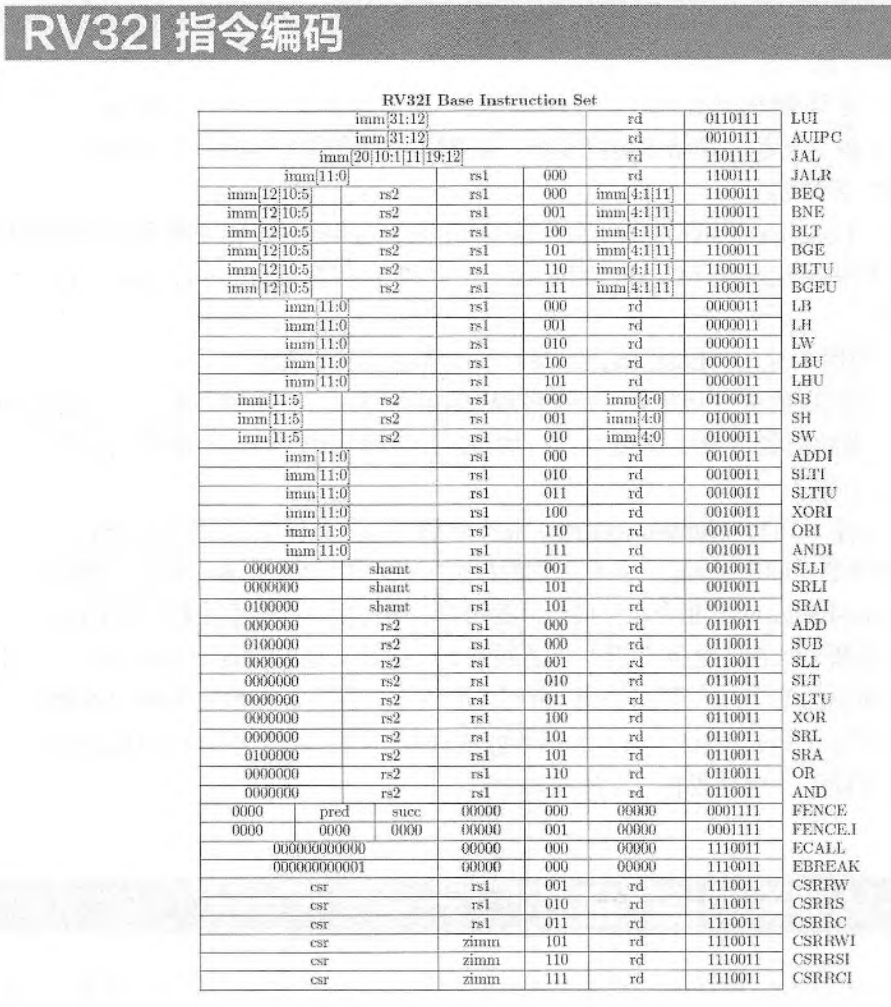


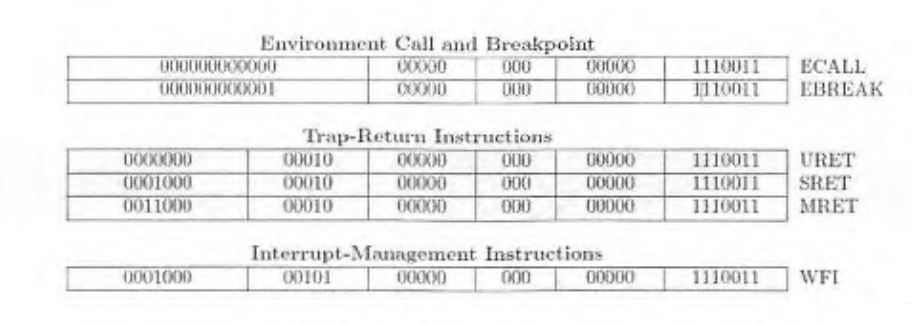


此处分类倾向于分为，算术运算指令、逻辑运算指令、比较运算指令、移位运算指令、访存指令、分支与跳转指令。

其中lui与auipc不属于算术预算指令，并且在运行中也不涉及任何ALU的运算功能。访存指令需要用到ALU的加法功能来计算内存的读写地址。分支指令运用ALU的比较功能来判断跳转。

此处设计，考虑为PC+imme设置独立加法器，原因1：分支指令判断大小时占用ALU加减法器（大小判断是通过减法完成的），如果用ALU完成PC+imme的运算会出现硬件资源冲突。原因2：方便五级流水线在译码阶段获得PC+imme的值完成jal指令

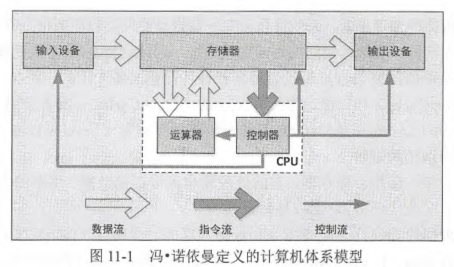




计算机体系结构

常见的计算机提下结构主要有两种，冯·诺依曼结构和哈佛结构

(1)冯·诺依曼结构也称普林斯顿结构，是一种将指令存储器和数据存储器合并在一起的存储器结构。程序的指令存储地址和数据存储地址指向同一个存储器的不同物理位置。



(2）哈佛结构是一种将指令存储器和数据存储器分开的存储器结构。如图11-2所示，它的主要特点如下。

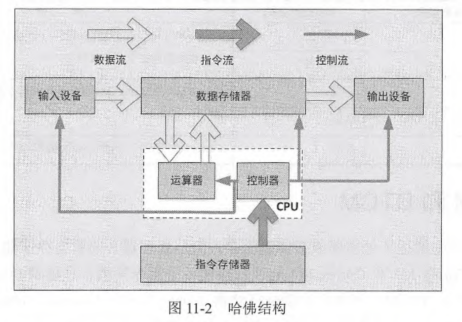
·将程序和数据存储在不同的存储空间中，即程序存储器和数据存储器是两个独立的

存储器，每个存储器独立编址、独立访问。

·与两个存储器相对应的是两条独立的指令总线和数据总线。这种分离的总线使得处

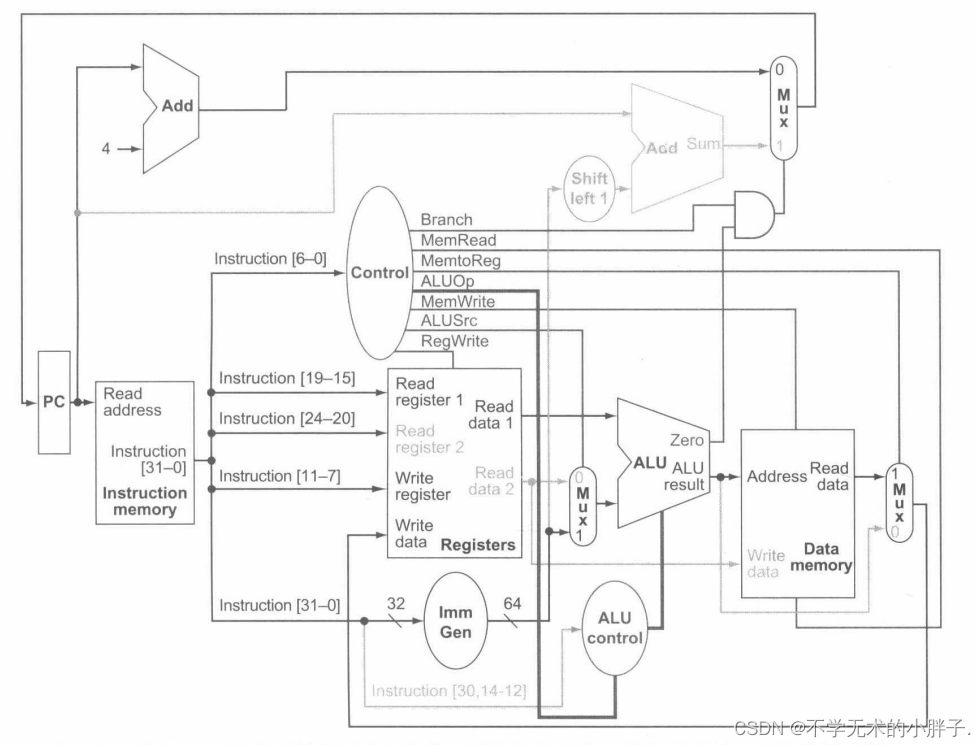
理器可以在一个周期内同时获得指令字（来自指令存储器）和操作数（来自数据存储器)，从而提高了执行速度和数据的吞吐率。

由于指令和数据存储在两个分开的物理空间中，因此取址和执行能完全并行。



单周期CPU

单周期CPU数据通路



CPU指令的执行过程

1. 以PC为地址，读取指令存储器中指令。一条指令包含以下信息：指令类型（opcode），指令执行的功能（func3,func7）,立即数（imme）,源寄存器地址（rs1,rs2），目的寄存器地址（rd）。
2. 对读取的指令进行译码。解读指令包含所需的信息，将这些信息送至控制模块和执行单元。
3. 执行单元根据指令执行功能，对数据进行运算。执行单元指图中的算术逻辑运算单元（ALU），其数据来源为指令信息中的立即数，以及根据源寄存器地址读取的通用寄存器堆中的数据。
4. 数据存储器的数据写入和读取，根据指令类型判断是否对数据存储器进行操作，需要进行操作的指令有I-type 中的load指令（lw，lb，lh，lbu，lhu）和s-type指令
5. 根据数据类型和目的寄存器地址rd将结果写回通用寄存器堆。结果的来源有PC+4、imme、ALU的计算结果、数据寄存器的load的数据。

CPU需要的部件

1. 指令存储器
2. PC寄存器
3. 译码器
4. 通用寄存器
5. ALU
6. 数据存储器
7. 各单元的控制器

存储器

指令RAM与数据RAM均采用异步RAM的形式，指令RAM由于测试阶段不涉及写入功能，用ROM替代RAM

具体代码：

指令RAM

端口说明

|  |  |  |  |
| --- | --- | --- | --- |
| 端口名 | 位宽 | 输入输出 | 说明 |
| addr | 8bits | input | 存储器读地址 |
| instr | 32bits | output | 输出指令 |

module instr\_RAM (

    input [`addrwidth-1:0] addr,

    output [`datawidth-1:0] instr

);

localparam  MEMDEPTH = 1<<`addrwidth;

reg [`datawidth-1:0] ROM [0:MEMDEPTH-1];

//rom进行初始化

    initial begin

        $readmemb("./file/RV32I.txt", ROM);

    end

assign instr = ROM [addr] ;

endmodule //instr\_RAM

数据RAM

|  |  |  |  |
| --- | --- | --- | --- |
| 端口名 | 位宽 | 输入输出 | 说明 |
| clk | 1bit | input | 系统时钟 |
| W\_en | 1bit | input | 写使能 |
| addr | 8bits | input | 读写地址 |
| din | 32bits | input | 写数据 |
| dout | 32bits | output | 读数据 |

module dataRAM(

    clk,

    W\_en,

    addr,

    din,

    dout

    );

    localparam  MEMDEPTH = 1<< `addrwidth;

    input clk;

    input W\_en;

    input [`addrwidth-1:0]addr;

    input [`datawidth-1:0]din;

    output [`datawidth-1:0]dout;

    reg [`datawidth - 1 : 0] ram [ MEMDEPTH - 1 : 0];

  always @(posedge clk ) begin

        if (W\_en)

            ram[addr] <= din;

  end

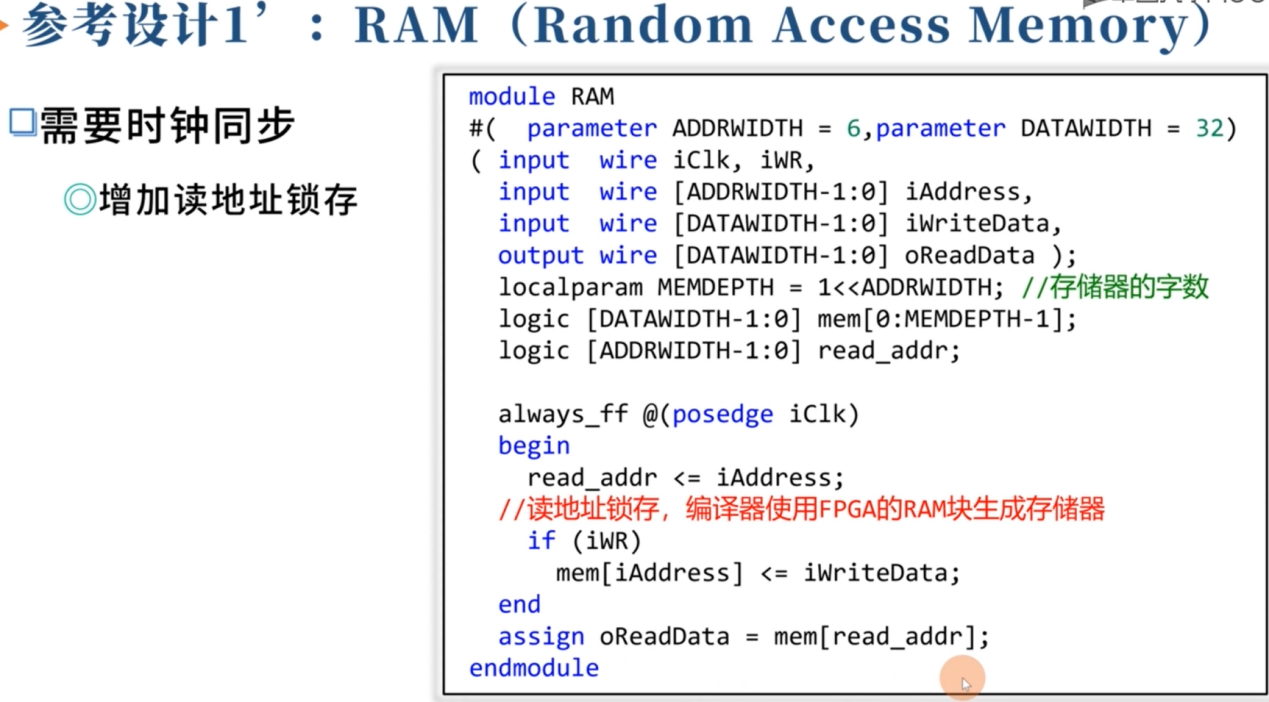
  assign dout = ram[addr] ;

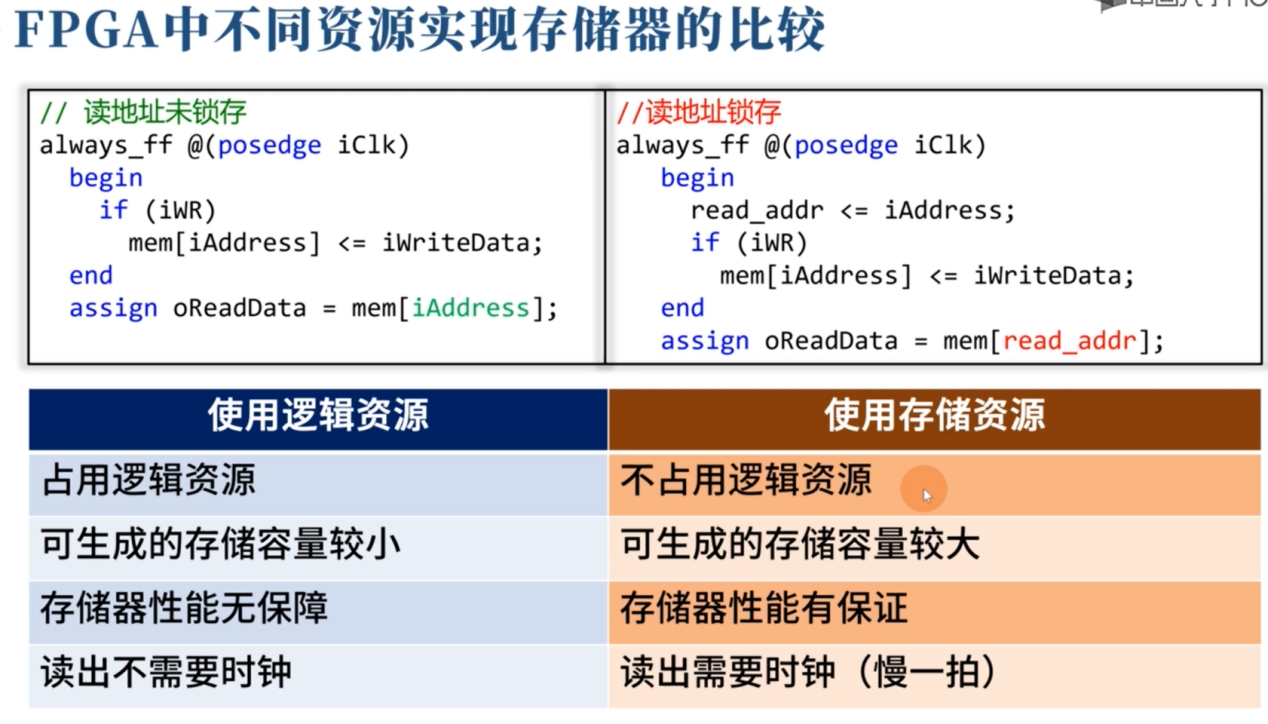
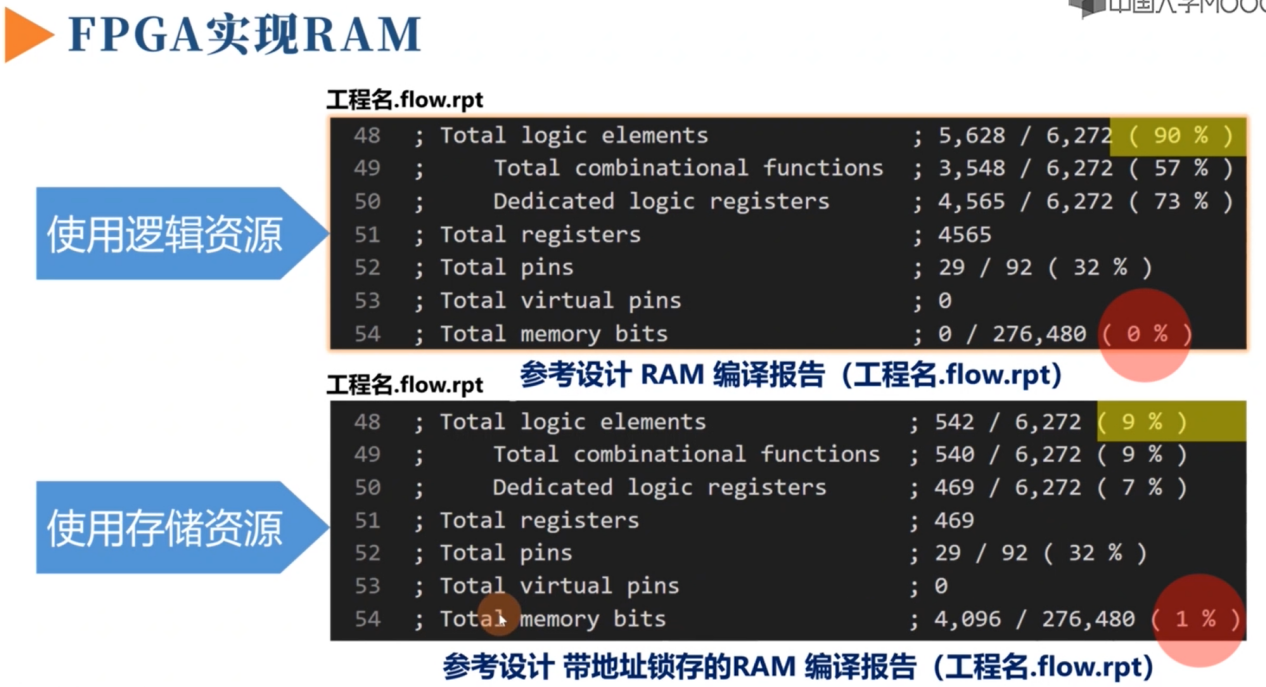
endmodule

异步RAM由于没有读地址缓存，综合时不会使用FPGA板子的BRAM资源而是使用逻辑资源生成RAM。

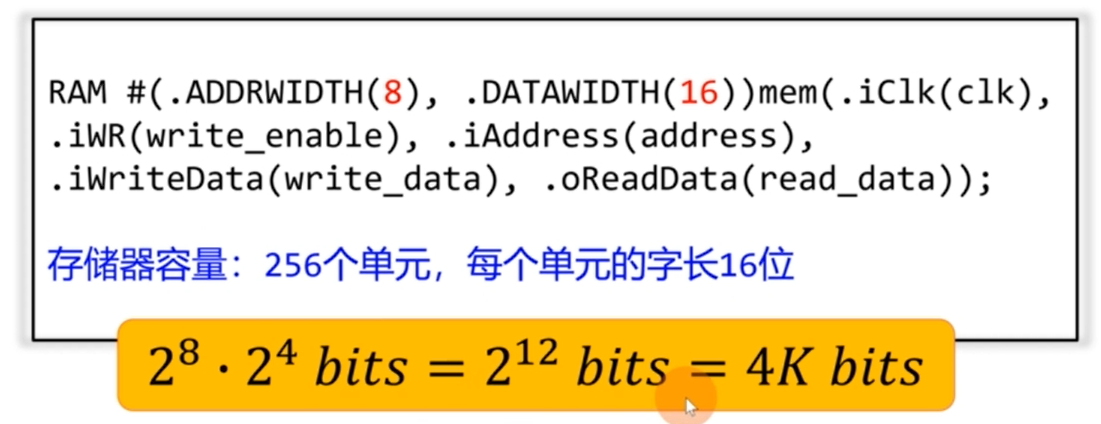
同步RAM与异步RAM

同步RAM设计参考





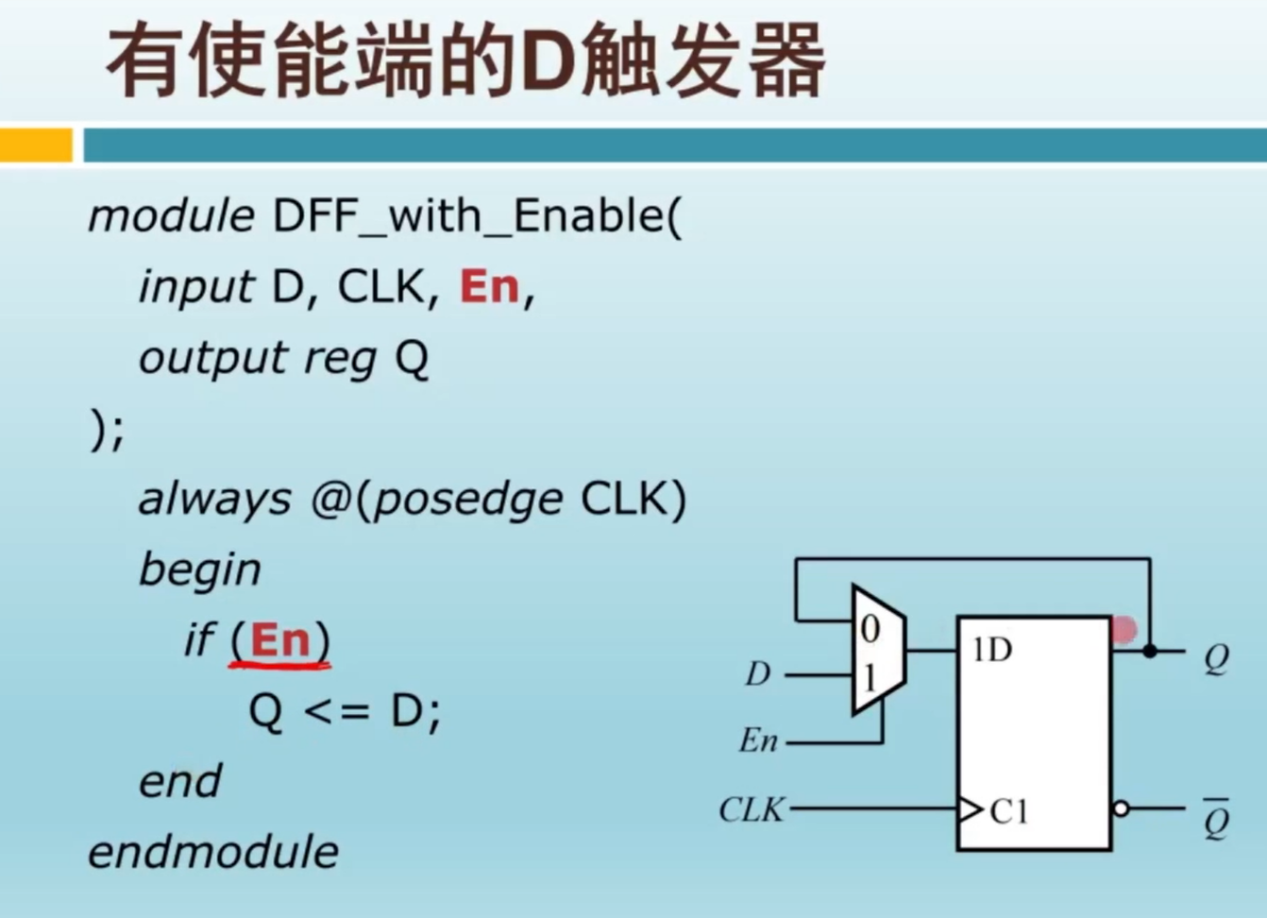
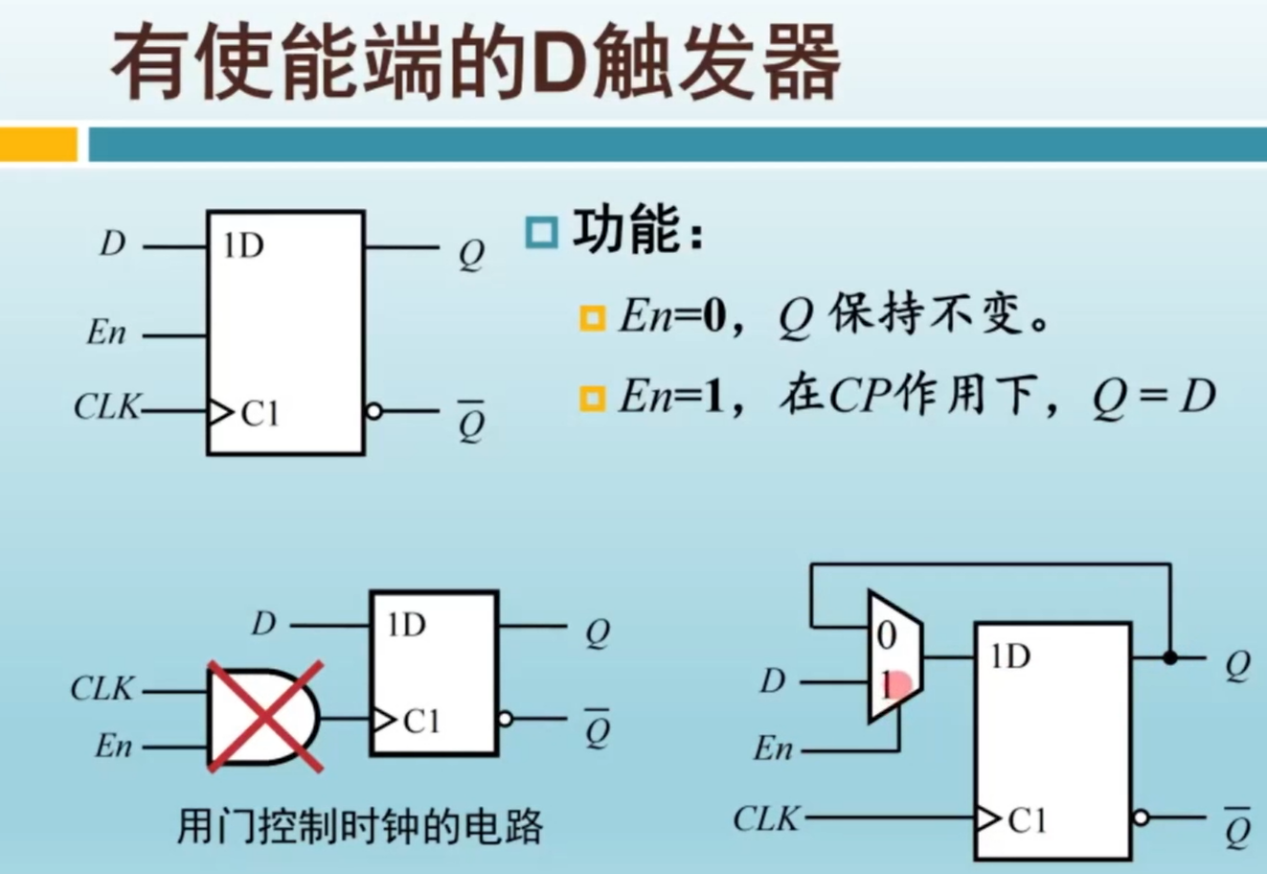
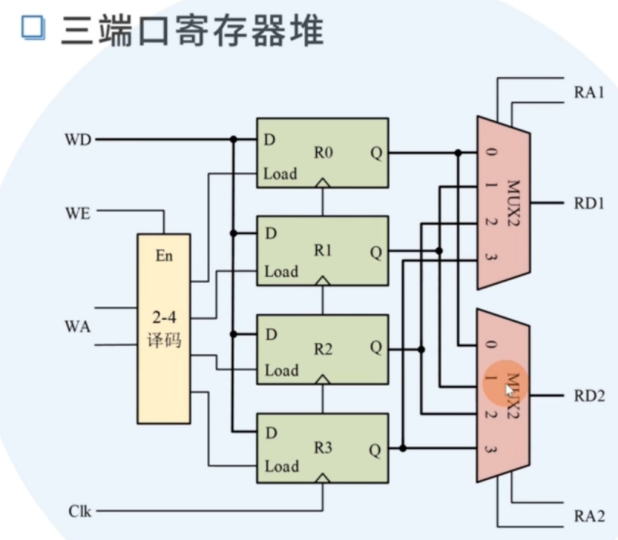
异步RAM在读地址发生变化时读数据也会发生相应的变化，同步RAM在对地址发生变化时，读数据需要等待下一个时钟边沿到来才会发生相应变化。

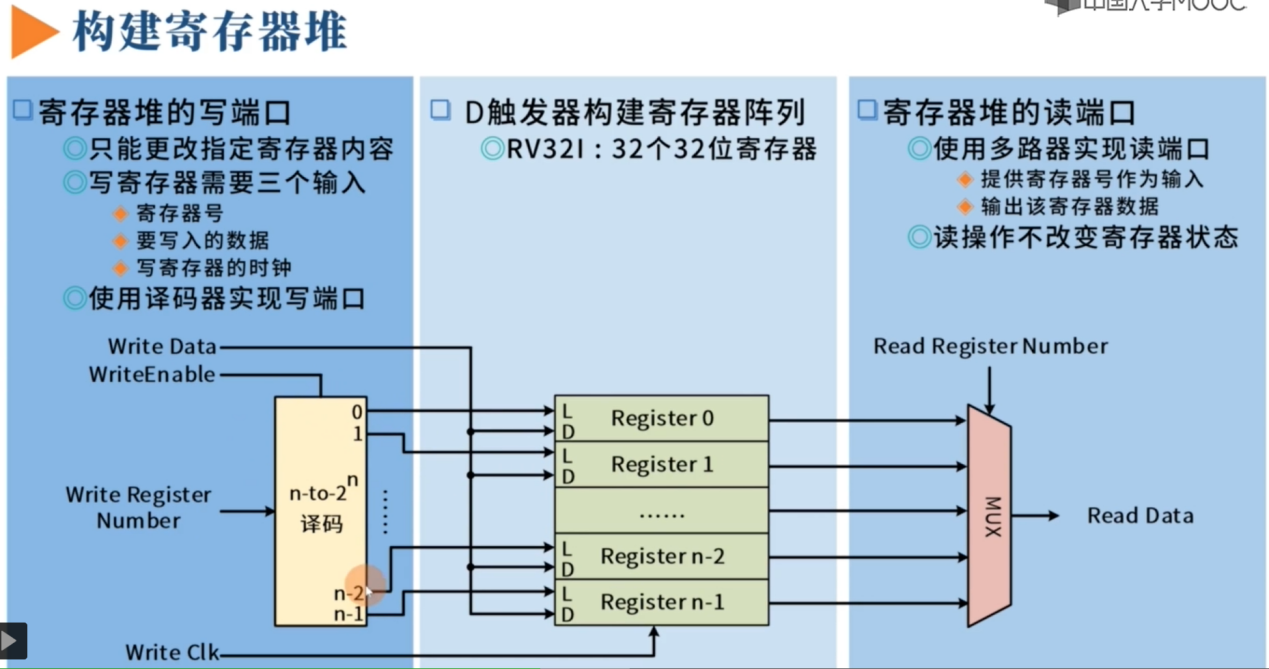


通用寄存器堆

RV32I指令集的通用寄存器堆有32个32bits寄存器阵列组成，其中0号寄存器值恒为0，只读不写。

通用寄存器采用三端口寄存器对结构，即有一个写地址，两个读地址。





寄存器堆的写端口由一个带使能的译码器构成，该使能信号来源为寄存器的写使能。由译码器的输出作为寄存器阵列的使能信号，完成写操作。寄存器的输出端口为一个32选1的数据选择器构成，选择信号为寄存器读地址。

|  |  |  |  |
| --- | --- | --- | --- |
| 端口名 | 位宽 | 输入输出 | 说明 |
| clk | 1bit | input | 系统时钟 |
| W\_en | 1bit | input | 写使能 |
| Rs1 | 5bits | input | 读地址1 |
| Rs2 | 5bits | input | 读地址2 |
| Rd | 5bits | input | 写地址 |
| Wr\_data | 32bits | input | 写数据 |
| Rd\_data1 | 32bits | output | 读数据1 |
| Rd\_data2 | 32bits | output | 读数据2 |

module registers(

    clk,

    W\_en,

    Rs1,

    Rs2,

    Rd,

    Wr\_data,

    Rd\_data1,

    Rd\_data2

    );

    input clk;

    input W\_en;

    input [4:0]Rs1;

    input [4:0]Rs2;

    input [4:0]Rd;

    input [31:0]Wr\_data;

    output [31:0]Rd\_data1;

    output [31:0]Rd\_data2;

    reg [31:0] regs [31:0];

///write

    always@(posedge clk )

        begin

            if(W\_en & (Rd!=0))

            regs[Rd] <= Wr\_data;

        end

//read

    assign Rd\_data1=(Rs1==5'd0)?`ZeroWord: regs[Rs1];

    assign Rd\_data2=(Rs2==5'd0)?`ZeroWord: regs[Rs2];

Endmodule

译码器

一条指令包含以下信息：指令类型（opcode），指令执行的功能（func3,func7）,立即数（imme）,源寄存器地址（rs1,rs2），目的寄存器地址（rd）。其中除了imme在不同指令会有不同的位置分布，其余信息在指令中的位置是固定。

如果只考虑输出立即数，只需根据指令分类，判断R、I、S、B、U、J六种指令类型，然后根据立即数在指令中的不同位置输出相应的立即数即可。此处考虑对所有指令类型进行判断，是为了后续控制模块不必在根据opcode判断指令类型。即不在向后输出opcode信息，而是直接输出指令类型信息

|  |  |  |  |
| --- | --- | --- | --- |
| 端口名 | 位宽 | 输入输出 | 说明 |
| Instr | 32bits | input | 指令ROM输入指令 |
| func3 | 3bits | output | 指令执行功能 |
| func7 | 1bits | output | 指令执行功能 |
| Rs1 | 5bits | output | 通用寄存器对地址1 |
| Rs2 | 5bits | output | 通用寄存器读地址2 |
| Rd | 5bits | output | 通用寄存器写地址 |
| imme | 32bits | output | 立即数 |
| I\_type | 1bit | output | 指令类型 |
| L\_type | 1bit | output | 指令类型 |
| S\_type | 1bit | output | 指令类型 |
| R\_type | 1bit | output | 指令类型 |
| JAL\_instr | 1bit | output | 指令类型 |
| JALR\_instr | 1bit | output | 指令类型 |
| LUI\_instr | 1bit | output | 指令类型 |
| AUIPC\_instr | 1bit | output | 指令类型 |

module decode(

    input [31:0]instr,

    //output [4:0]opcode,

    output [2:0]func3,

    output func7,

    output [4:0]Rs1,

    output [4:0]Rs2,

    output [4:0]Rd,

    output [31:0]imme,

    output I\_type ,

    output L\_type,

    output S\_type,

    output R\_type,

    output JAL\_instr,

    output JALR\_instr,

    output LUI\_instr,

    output AUIPC\_instr,

    output B\_type

    );

    wire [31:0]I\_imme;

    wire [31:0]U\_imme;

    wire [31:0]J\_imme;

    wire [31:0]B\_imme;

    wire [31:0]S\_imme;

    //一条指令包含：操作码opcode，功能码func3、func7，寄存器读地址 Rs1、Rs2，寄存器写地址 Rd，立即数imme

    //操作码opcode，功能码func3、func7，寄存器读地址 Rs1、Rs2，寄存器写地址 Rd，在指令里的位置是固定的

    //assign opcode=instr[6:2];

    assign func3=instr[14:12];

    assign func7=instr[30];

    assign Rd =instr[11:7];

    assign I\_type     = instr[6:0]==`I\_TYPE                               ;

    assign L\_type     = instr[6:0]==`L\_TYPE                               ;

    assign S\_type     = instr[6:0]==`S\_TYPE                               ;

    assign R\_type     = instr[6:0]==`R\_TYPE                               ;

    assign JAL\_instr  = instr[6:0]==`INST\_JAL                             ;

    assign JALR\_instr = instr[6:0]==`INST\_JALR                            ;

    assign LUI\_instr  = instr[6:0]==`INST\_LUI                             ;

    assign AUIPC\_instr= instr[6:0]==`INST\_AUIPC                           ;

    assign B\_type     = instr[6:0]==`B\_TYPE                               ;

    assign Rs1=(I\_type|L\_type|S\_type|R\_type|JALR\_instr|B\_type)? instr[19:15] : 0;

    assign Rs2=(S\_type|R\_type|B\_type)?instr[24:20]:0;

    //根据立即数在指令中的不同位置，对立即数进行分类

    assign I\_imme={{20{instr[31]}},instr[31:20]};

    assign U\_imme={instr[31:12],{12{1'b0}}};

    assign J\_imme={{12{instr[31]}},instr[19:12],instr[20],instr[30:21],1'b0};

    assign B\_imme={{20{instr[31]}},instr[7],instr[30:25],instr[11:8],1'b0};

    assign S\_imme={{20{instr[31]}},instr[31:25],instr[11:7]};

    assign imme= (I\_type|L\_type|JALR\_instr)?I\_imme :

                 (LUI\_instr|AUIPC\_instr)?U\_imme :

                 JAL\_instr?J\_imme :

                 B\_type?B\_imme :

                 S\_type?S\_imme : 32'd0;

endmodule

ALU

ALU需要具有算术运算，比较，逻辑运算，移位功能，并根据不同指令执行功能共，对数据进行不同的运算。

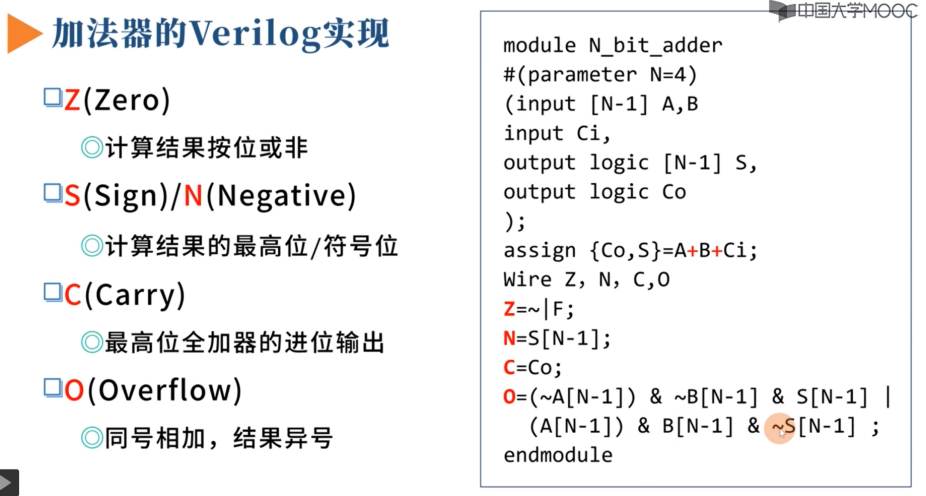
设计中四种功能是同时计算的，除了算术和比较，因为它们都需要使用加减法器。

算术运算：加减法

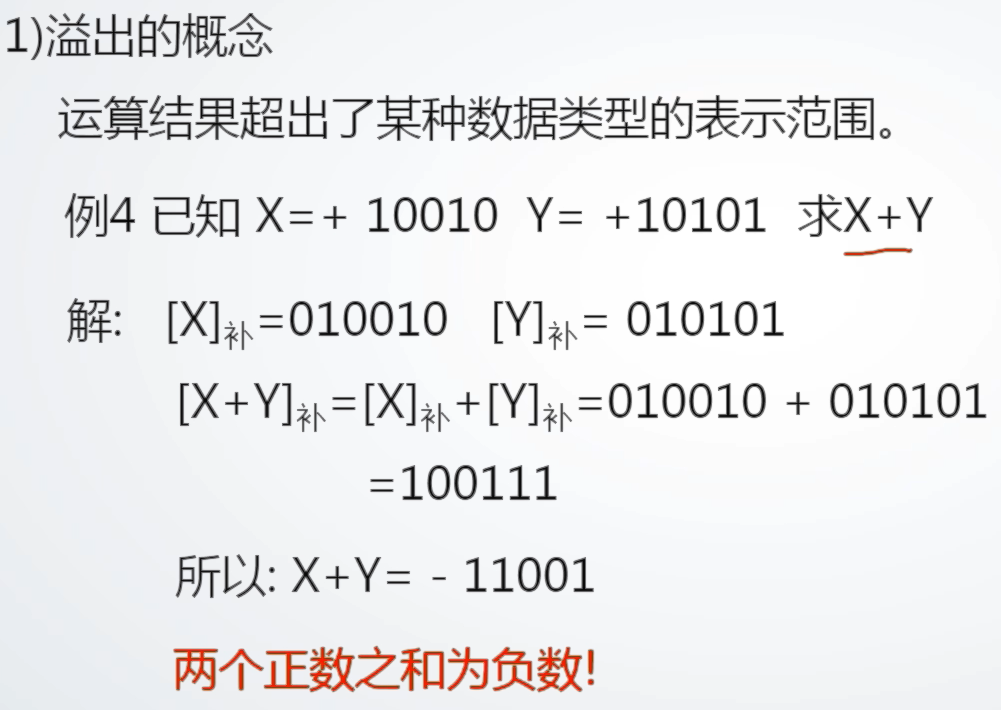
CPU内的数据都用补码形式存储。

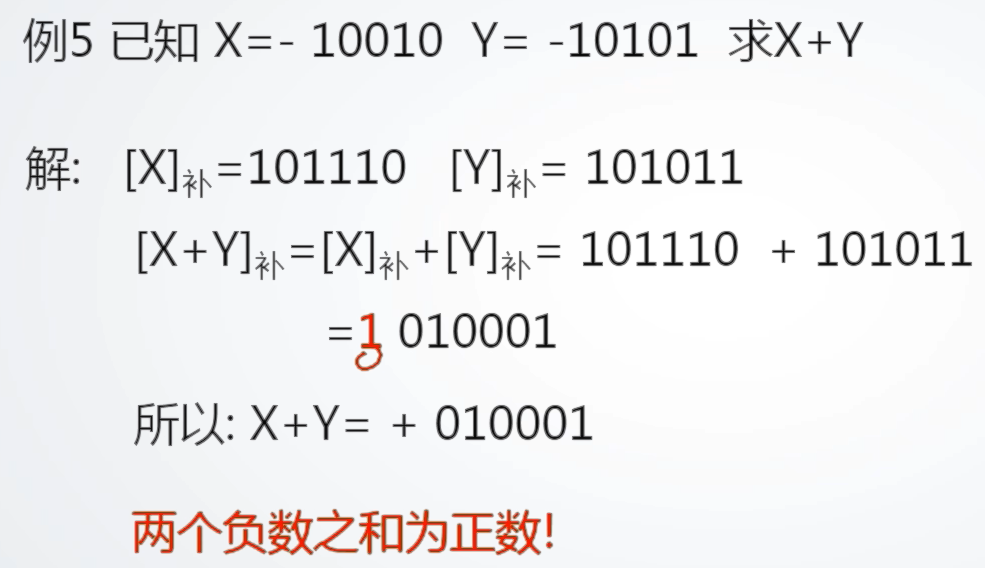
加法：

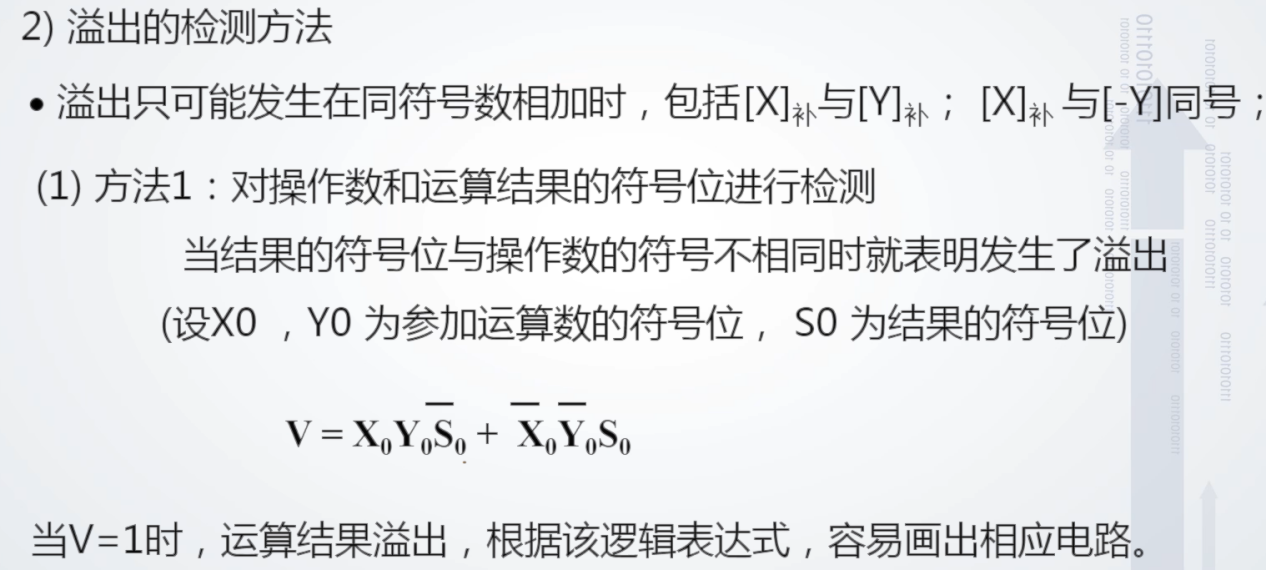
32位全加器



有符号数溢出检测







代码：

|  |  |  |  |
| --- | --- | --- | --- |
| 端口名 | 位宽 | 输入输出 | 说明 |
| data\_A | 32bits | input | 输入数据1 |
| data\_B | 32bits | input | 输入数据2 |
| ci | 1bit | input | 最低位低位的进位 |
| result | 32bits | output | 计算结果 |
| co | 1bit | output | 最高位的进位标志 |
| zero | 1bit | output | 结果为0的标志 |
| Result\_sign | 1bit | output | 结果的符号 |
| overflow | 1bit | output | 结果已出标志 |

module adder(

    input [`datawidth-1:0] data\_A,data\_B,

    input ci,

    output [`datawidth-1:0] result,

    output co,zero,result\_sign,overflow

    );

    assign {co,result}=data\_A+data\_B+ci;

    assign overflow = data\_A[`datawidth-1]&data\_B[`datawidth-1]&(~result[`datawidth-1])|(~data\_A[`datawidth-1])&(~data\_B[`datawidth-1])&result[`datawidth-1];

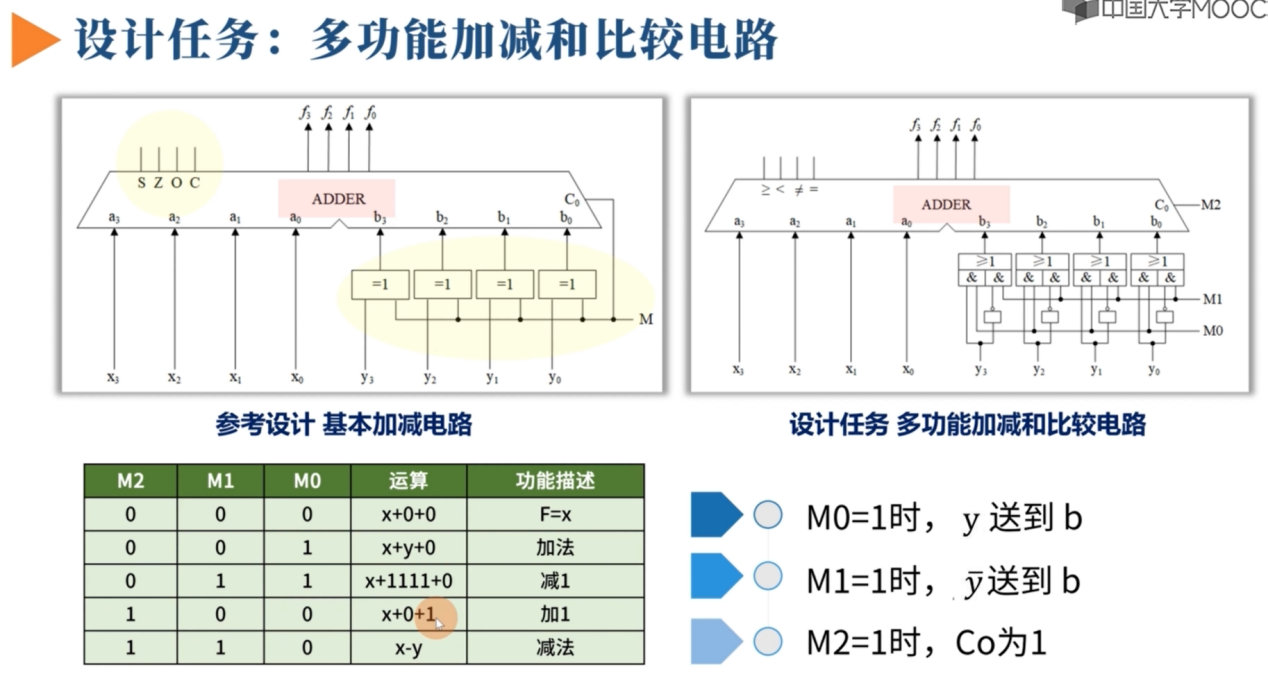
    assign zero = ~(|result);

    assign result\_sign=result[`datawidth-1];

endmodule

该加法器实现了有符号数的加法与溢出判断。

减法：由加法器实现减法，组成加减法器



加减法器的主要结构变化是在加法器的的一个输入数据前加上了一组异或门

1^A=~A;0^A=A;

M=0时加减法器运行加法，输入数据为原值。

M=1时输入数据按位取反，并将M接入最低位进位，实现+1，从而完成原码与反码的转换。

进而实现减法功能。

|  |  |  |  |
| --- | --- | --- | --- |
| 端口名 | 位宽 | 输入输出 | 说明 |
| data\_A | 32bits | input | 输入数据1 |
| data\_B | 32bits | input | 输入数据2 |
| ci | 1bit | input | 最低位低位的进位 |
| result | 32bits | output | 计算结果 |
| co | 1bit | output | 最高位的进位标志 |
| zero | 1bit | output | 结果为0的标志 |
| Result\_sign | 1bit | output | 结果的符号 |
| overflow | 1bit | output | 结果已出标志 |
| sub\_flag | 1bit | input | 减法标志位 |

module add\_suber(

    input [`datawidth-1:0] data\_A,data\_B,

    input sub\_flag,

    output [`datawidth-1:0] result,

    output co,zero,result\_sign,overflow

    );

    wire [`datawidth-1:0] Y;

    assign Y = data\_B ^ {`datawidth{sub\_flag}};//补码转换,实现运行加法时不转换，运行减法是转换

    adder u\_adder(

        .data\_A      ( data\_A      ),

        .data\_B      ( Y      ),

        .ci          ( sub\_flag    ),

        .result      ( result      ),

        .co          ( co          ),

        .zero        ( zero        ),

        .result\_sign ( result\_sign ),

        .overflow    ( overflow    )

    );

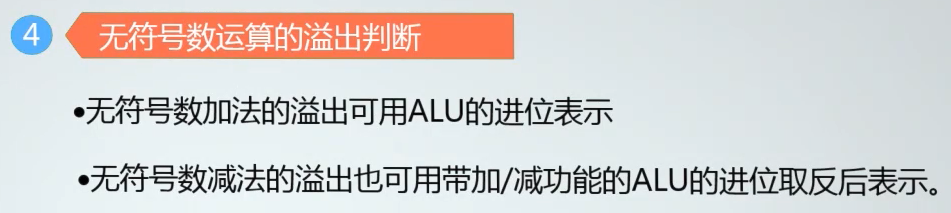
Endmodule

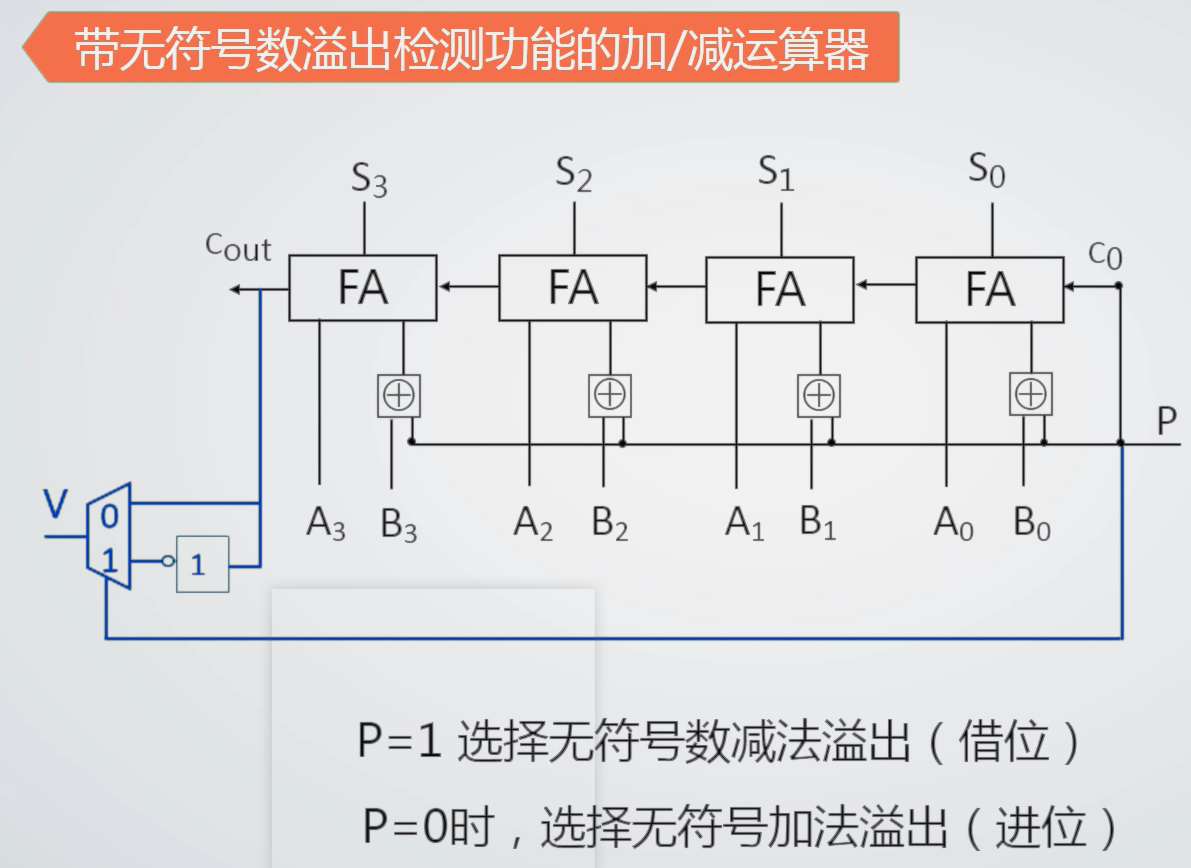
由于是根据有符号数加法器进行改造，减法器实现的也是有符号数的减法与溢出位判断。

无符号数运算

无符号数加法与减法可以通过上面的加法器正常计算，由于结果仍然是无符号数，结果符号位无效，结果为零标志位与最高位进位依旧有效。

无符号数加法与减法溢出判读





有符号数比较大小

有符号数比较大小用有符号数减法实现，在不发生溢出的情况下，结果符号位为1，即结果为负数时，被减数小于减数，反之。发生溢出时，结果符号位为0，被减数小于减数，反之。

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Result\_sign | 1 | 0 | 0 | 1 |
| overflow | 0 | 0 | 1 | 1 |
| Small | 1 | 0 | 1 | 0 |

结果符号位^溢出标志为被减数是否小于减数的结果。

逻辑运算功能

根据指令功能实现按位与、或、异或的功能

移位运算功能

逻辑左移与算术左移的功能都通过低位补零的操作来完成。

逻辑右移的功能通过高位补零的的操作来完成。

算术右移，需要对数据高位用符号位数据进行补充。

|  |  |  |  |
| --- | --- | --- | --- |
| 端口名 | 位宽 | 输入输出 | 说明 |
| Indata | 32bits | input | 输入数据 |
| SHIFT\_NUM | 5bits | input | 移位数值 |
| Shiftctr | 2bits | input | 移位类型 |
| shift\_result | 32bits | output | 计算结果 |

module shifter(

    input [`datawidth-1:0] Indata,

    input [`SHIFT\_NUM\_width] SHIFT\_NUM,

    input [`shift\_ctr] Shiftctr,

    output [`datawidth-1:0] shift\_result);

    wire [5:0] shift\_n;

    assign shift\_n = 6'd32 - SHIFT\_NUM;

    wire [`datawidth-1:0] left,right,A\_right;

    assign left = Indata<<SHIFT\_NUM;

    assign right = Indata>>SHIFT\_NUM;

    assign A\_right =({32{Indata[`datawidth-1]}}<<shift\_n)| right ;

    //11:算术右移；10：逻辑右移；01，00：左移

    assign shift\_result = Shiftctr[1]&Shiftctr[0]? A\_right:~Shiftctr[1]?left:right;

endmodule

ALU：ALU有10种指令功能码，根据运算类型分为四类，用一个两位的Op表示

00：加减；01：逻辑运算；10：比较；11：移位

最后根据Op设计4选1的选择器，输出对应的运算结果。

根据不同指令功能码列卡诺图

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| op1 | 00 | 01 | 11 | 10 |
| 00 | 0 | 1 | 1 | 1 |
| 01 | 0 | 1 | 0 | 0 |
| 11 | × | 1 | × | × |
| 10 | 0 | × | × | × |
|  |  |  |  |  |
| op0 | 00 | 01 | 11 | 10 |
| 00 | 0 | 1 | 0 | 0 |
| 01 | 1 | 1 | 1 | 1 |
| 11 | × | 1 | × | × |
| 10 | 0 | × | × | × |

仅有jal、jalr、lui、auipc不需要使用ALU，这四个指令的运行不受ALU结果影响，所以可以将10种指令功能码之外的情况视为无关项。

 //用2位的Op对ALU结果进行分类，根据真值表获得Op与ALUop的逻辑关系

assign Op[1] = (~ALUop[1])&ALUop[0]|((~ALUop[3])&(~ALUop[2])&ALUop[1]);

assign Op[0] = (~ALUop[1])&ALUop[0]|((~ALUop[3])&ALUop[2]);

|  |  |  |  |
| --- | --- | --- | --- |
| 端口名 | 位宽 | 输入输出 | 说明 |
| ALU\_DA | 32bits | input | 输入数据1 |
| ALU\_DB | 32bits | input | 移位数据2 |
| ALUop | 4bits | input | 指令功能码 |
| ALU\_ZERO | 1bit | output | 零标志 |
| ALU\_OverFlow | 1bit | output | 溢出标志 |
| ALU\_res | 32bits | output | 计算结果 |

module ALU(

    ALU\_DA,

    ALU\_DB,

    ALUop,

    ALU\_ZERO,

    ALU\_OverFlow,

    ALU\_res

    );

    input [`datawidth-1:0]    ALU\_DA;

    input [`datawidth-1:0]    ALU\_DB;

    input [`ALU\_CLT]     ALUop;//{funct7,funct3}

    output          ALU\_ZERO;

    output          ALU\_OverFlow;

    output reg [`datawidth-1:0]   ALU\_res;

    //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*generate ctr\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

    wire SUBctr;

    wire SIGctr;

    wire [1:0] Op;//将操作分类，00：加减；01：逻辑运算；10：比较；11：移位

    wire [1:0] Logicctr;

    wire [1:0] Shiftctr;

    assign SUBctr = ~ALUop[1]&ALUop[3]|ALUop[1];//比较和减法用到减操作，为1，加法操作必为0，其余为无关项

    //用2位的Op对ALU结果进行分类，根据真值表获得Op与ALUop的逻辑关系

    assign Op[1] = (~ALUop[1])&ALUop[0]|((~ALUop[3])&(~ALUop[2])&ALUop[1]);

    assign Op[0] = (~ALUop[1])&ALUop[0]|((~ALUop[3])&ALUop[2]);

    assign Logicctr = ALUop[1:0];

    assign Shiftctr = {ALUop[2],ALUop[7]};

    assign SIGctr = ALUop[0];

    //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

    //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*logic op\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

    reg [`datawidth-1:0] logic\_result;

    always@(\*) begin

        case(Logicctr)

        2'b00:logic\_result = ALU\_DA ^ ALU\_DB;

        //2'b01:logic\_result = ALU\_DA | ALU\_DB;

        2'b10:logic\_result = ALU\_DA | ALU\_DB;

        2'b11:logic\_result = ALU\_DA & ALU\_DB;

        default : logic\_result=32'bx;

        endcase

    end

    //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

    //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*shift op\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

    wire [`SHIFT\_NUM\_width]     ALU\_SHIFT;

    wire [`datawidth-1:0] shift\_result;

    assign ALU\_SHIFT=ALU\_DB[ `SHIFT\_NUM\_width];

    shifter u\_shifter(

        .Indata    ( ALU\_DA    ),

        .SHIFT\_NUM ( ALU\_SHIFT ),

        .Shiftctr  ( Shiftctr  ),

        .shift\_result ( shift\_result )

    );

    //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

    //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*add sub op\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

    wire ADD\_carry,result\_sign,overflow;

    wire [`datawidth-1:0] ADD\_result;

    add\_suber u\_add\_suber(

        .data\_A      ( ALU\_DA      ),

        .data\_B      ( ALU\_DB      ),

        .sub\_flag    ( SUBctr    ),

        .result      ( ADD\_result   ),

        .co          ( ADD\_carry    ),

        .zero        ( ALU\_ZERO     ),

        .result\_sign ( result\_sign ),

        .overflow    ( overflow    )

    );

    //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

    //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*slt op\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

    //无符号数比较，即无符号数减法，加法器进位取反为1时表示发生溢出，即被减数小于减数发生借位

    //溢出发生在相同符号数相加的情况。即正数-负数和负数-正数两种情况。

    //正数-负数：正数>负数，结果为0。发生溢出，溢出标志为1，结果符号位为1；不发生溢出：溢出标志为0，结果符号位为0

    //负数-正数：负数<正数，结果为1。发生溢出，溢出标志为1，结果符号位为0；不发生溢出：溢出标志为0，结果符号位为1

    //其他情况：溢出标志为0，结果符号位为0时结果为0，结果符号位为1时结果为1。

    //有符号数比较结果=结果符号^溢出标志

    wire [`datawidth-1:0] SLT\_result;

    assign SLT\_result = SIGctr? {31'd0,~ADD\_carry} : {31'd0,result\_sign^overflow};

    assign ALU\_OverFlow = overflow;

    //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

    //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*ALU result\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

    always @(\*)

    begin

    case(Op)

        2'b00:ALU\_res=ADD\_result;

        2'b01:ALU\_res=logic\_result;

        2'b10:ALU\_res=SLT\_result;

        2'b11:ALU\_res=shift\_result;

        default : ALU\_res=32'b0;

    endcase

    end

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

endmodule

控制器

ALU控制器，功能是根据不同的指令类型，输出对应指令功能数，在imme和rs2数据中做出选择作为ALU\_DB的输入