Digital Design with Verilog

Verilog - Synthesis

Lecture 19: Logic Synthesis with Verilog HDL – Part2

Objectives

- Synthesis of Verilog Constructs
  - Logical Operators
  - Arithmetic Operators
  - Shift operators
  - Case operators
  - Vector Operators -Part Select

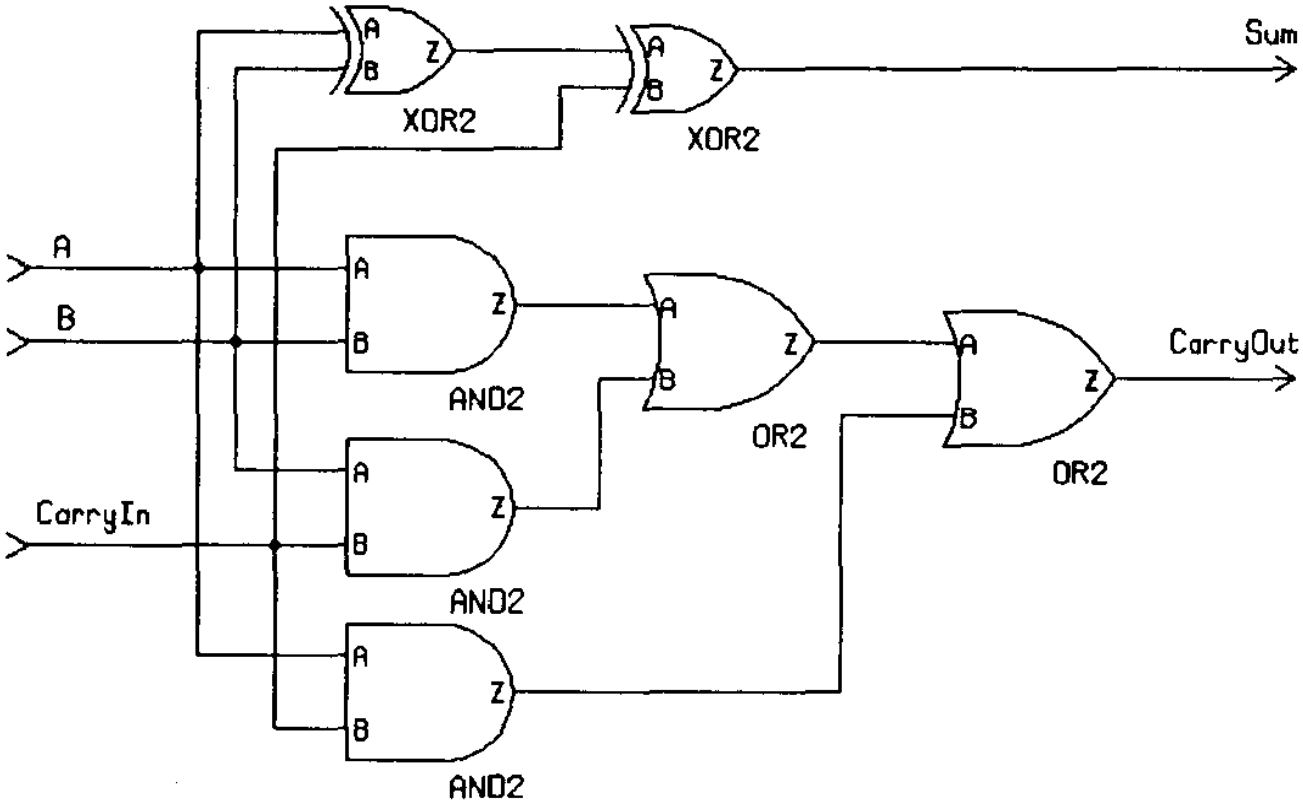
Logical Operators

- Logical Operators get directly mapped onto primitive logic gates in hardware.

```
module FullAdder (A, B, Carryin, Sum, CarryOut);
    input A, B, Carryin;
    output Sum, CarryOut;

    assign Sum = (A ^ B) ^ Carryin;
    assign CarryOut = (A & B) | (B & Carryin) | (A & Carryin);
endmodule
```

- This gets modeled as 3, 2-input AND; 2, 2-input OR and 2, 2-input XOR.

Synthesized Circuit



Logical operators map to primitive logic gates.

Arithmetic Operators

- A reg type is interpreted as an unsigned number and an integer type is interpreted as a signed number in 2's complement form with the rightmost bit as the least significant bit.

- Thus, to synthesize an unsigned arithmetic operator, the reg type is used.

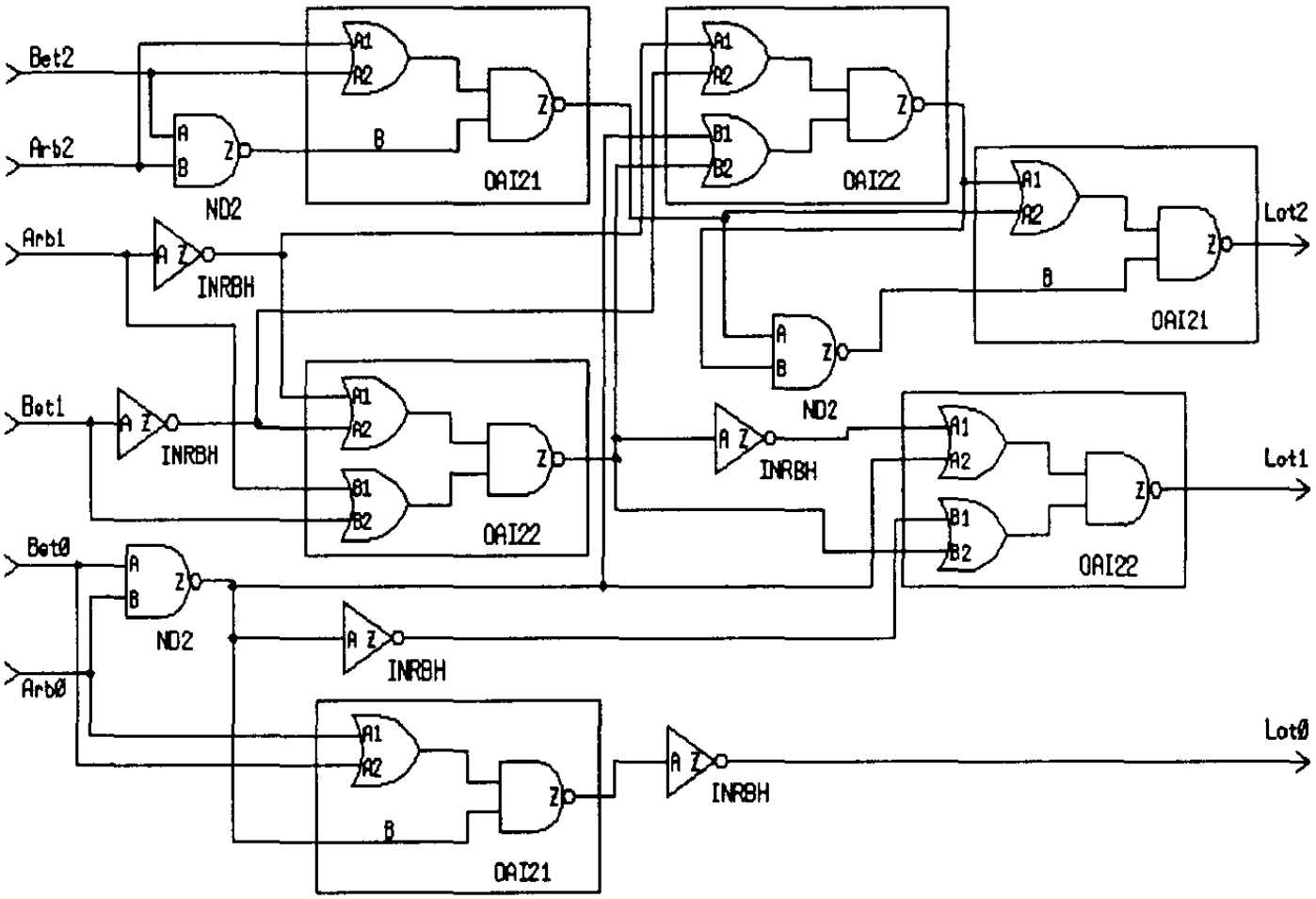
- To get a signed arithmetic operator, the integer type is used.

- The net type is interpreted as unsigned numbers.

Unsigned Arithmetic

```
module UnsignedAdder (Arb, Bet, Lot);
  input [2:0] Arb, Bet;
  output [2:0] Lot;

  assign Lot =Arb + Bet;

endmodule
```

Unsigned Arithmetic

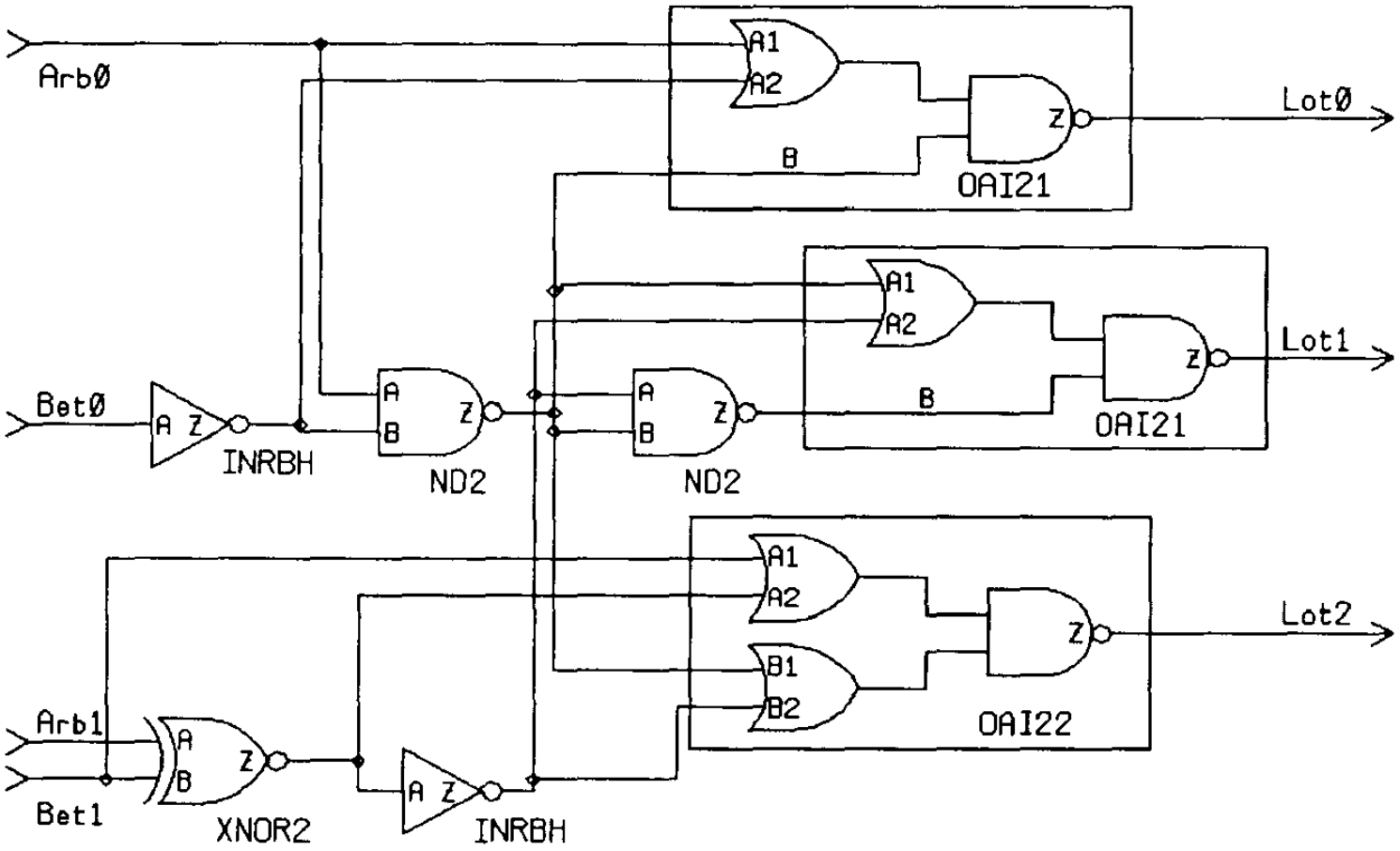


A 3-bit adder

Signed Arithmetic

```
module SignedAdder (Arb, Bet, Lot);
    input [1:0] Arb, Bet;
    output [2:0] Lot;

    reg [2:0]  Lot;

    always @ (Arb or Bet)
    begin: LABEL_A
    //A sequential block requires a label if local
    //declarations are present.

    integer Arbint, Betint;

    Arbint = - Arb; // Store negative number just to show
                    //that the + is operating on signed operands.
    Betint = Bet;
    Lot = Arbint + Betint;
    end
endmodule
```

Synthesized Circuit



Signed adder

Synthesized Circuit

- The synthesized circuit will be a 2-bit adder and not a 32-bit one.

- How it is found –Due to ‘Lot’ or Due to “Arb” or “Bet”.

- The adder logic with signed and unsigned operands is the same, as signed numbers are stored in 2’s complement form.

Modeling a Carry

- It is natural to model a carry by simply using the result size to be one bit larger than the largest of the two operands.

- Alternatively, a concatenation could also be used as the target of an assignment with the carry bit explicitly specified.

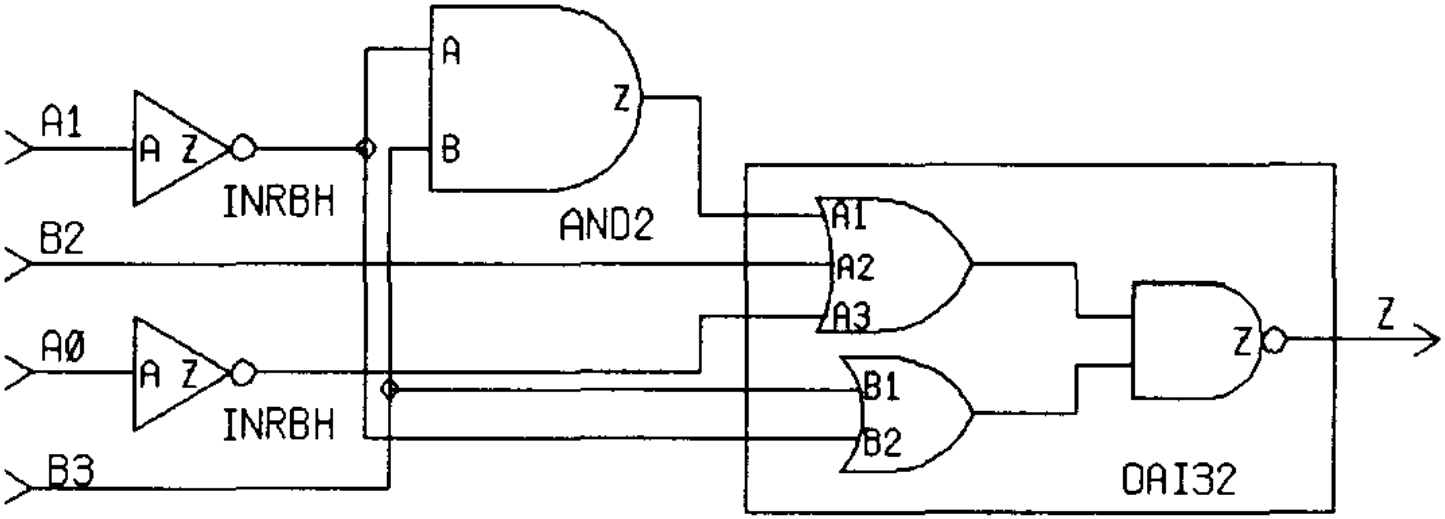
```
wire [3 :0] CdoBus, Sum;
wire [4:0] OneUp;
wire Bore;
assign OneUp = CdoBus + 1;
assign {Bore, Sum} = CdoBus- 2;
```

Relational and Equality Operators

- >,<, <=, >=, ==, !=
- Modeled like Arithmetic operators.
- Logic produced from Synthesis is different, depending on whether unsigned or signed.
- If variables of net or reg types are compared then, unsigned relational operator is synthesized
- If variables of integer types are compared then, signed relational operator is synthesized.

Relational Operators

```
module GreaterThan (A, B, Z);
    input [3:0] A, B;
    output Z;

    assign z = A[1:0] > B[3:2];
//Variables A and B are of net type.
endmodule
```

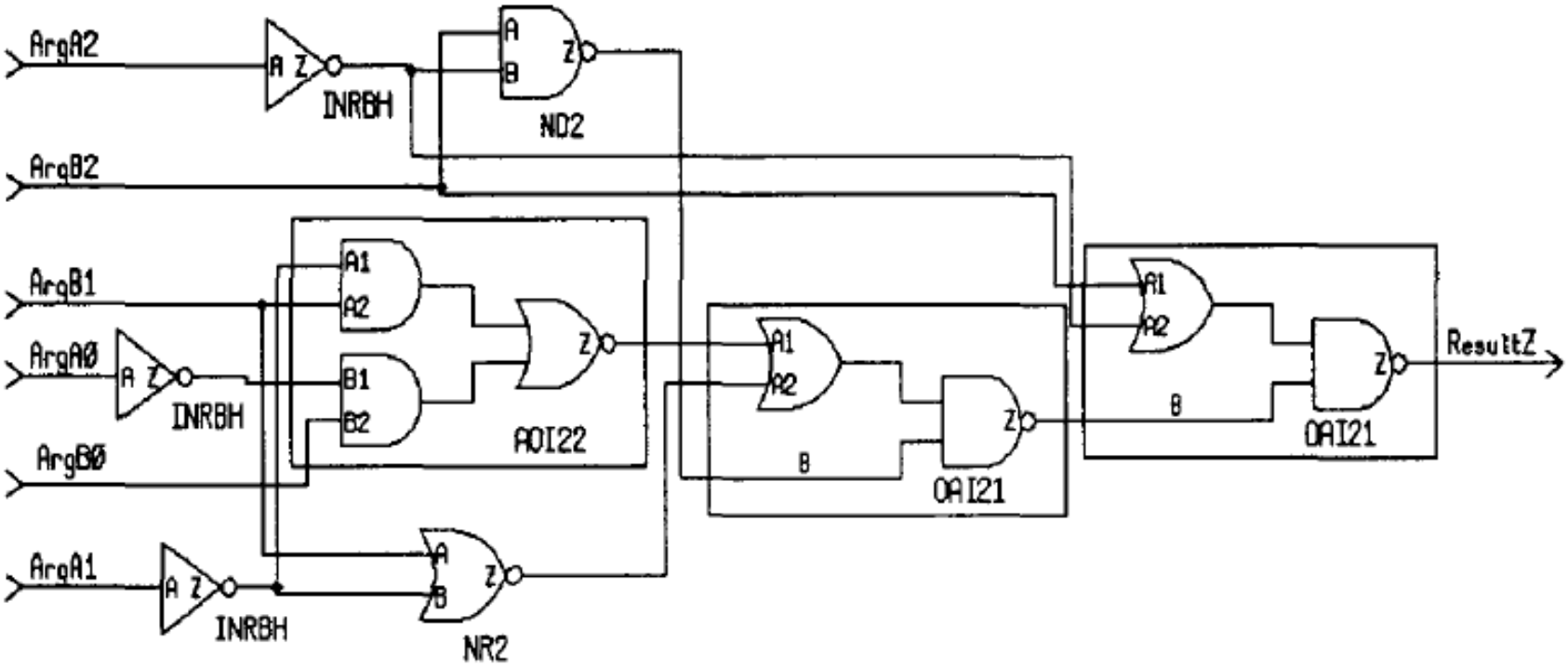


Unsigned ">" relational operator

Relational Operators

```
module LessThanEquals (ArgA, ArgB, ResultZ);
    input [2:0] ArgA, ArgB;
    output ResultZ;
    reg ResultZ;
    integer ArgAInt, ArgBInt;

    always @ (ArgA or ArgB)
    begin
    ArgAInt = - ArgA;
    ArgBInt = - ArgB;

    //Store negative values just to show that the
    //comparison is on signed numbers.
    ResultZ = ArgAInt <= ArgBInt;
    end
endmodule
```

Relational Operators



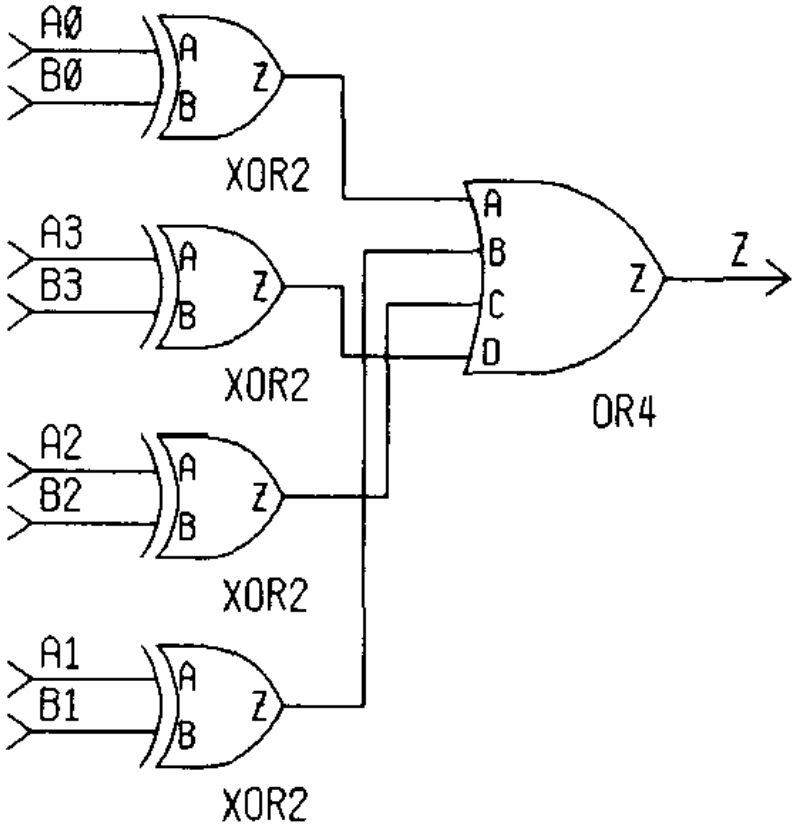
Signed "<=" relational operator.

Equality Operators

- The equality operators supported are: ==, !=

```
module NotEquals (A, B, Z);
input [0:3] A, B;
output Z;
reg Z;
    always @ (A or B)
    begin: DF_LABEL
        integer IntA, IntB;
        IntA =A;
        IntB = B;
        Z = IntA ! = IntB;
    end
endmodule
```

Equality Operators



Signed "!=" relational operator

Case Operators

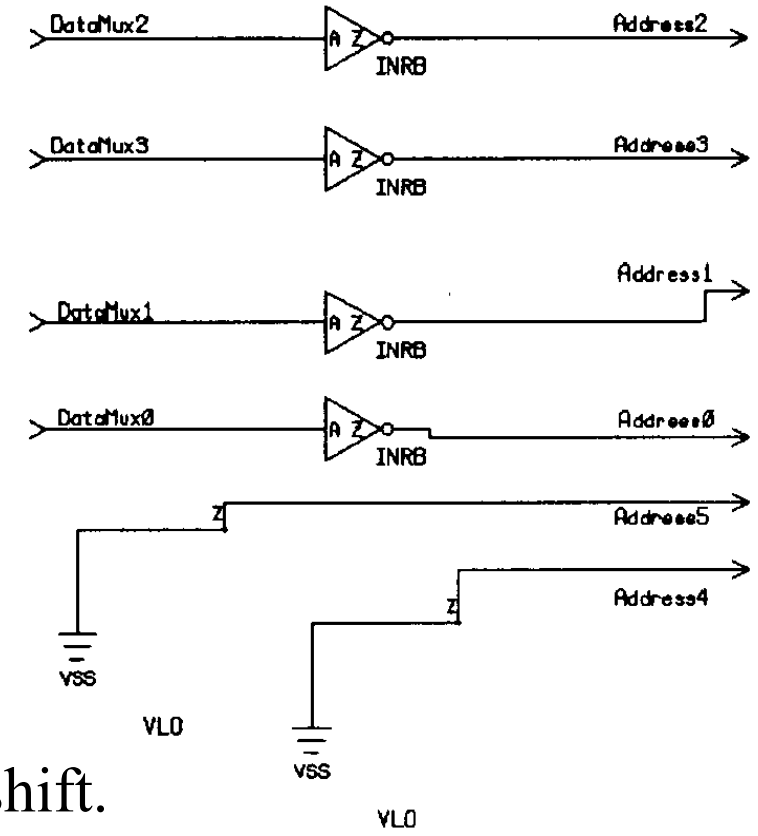
- Case equality and Case inequality operators (===) and (!==) are not supported for synthesis.

Shift Operators

- For synthesis, the left shift (<<) and the right shift (>>) operators are supported.
- The vacated bits are filled with 0.
- If the amount of shift is a constant then, simple rewiring is done, else a general-purpose shifter is synthesized

Shift Operators

```
module ConstantShift (DataMux, Address);
input [0:3] DataMux;
output [0:5] Address;
    assign Address = (~DataMux) << 2;
endmodule
```

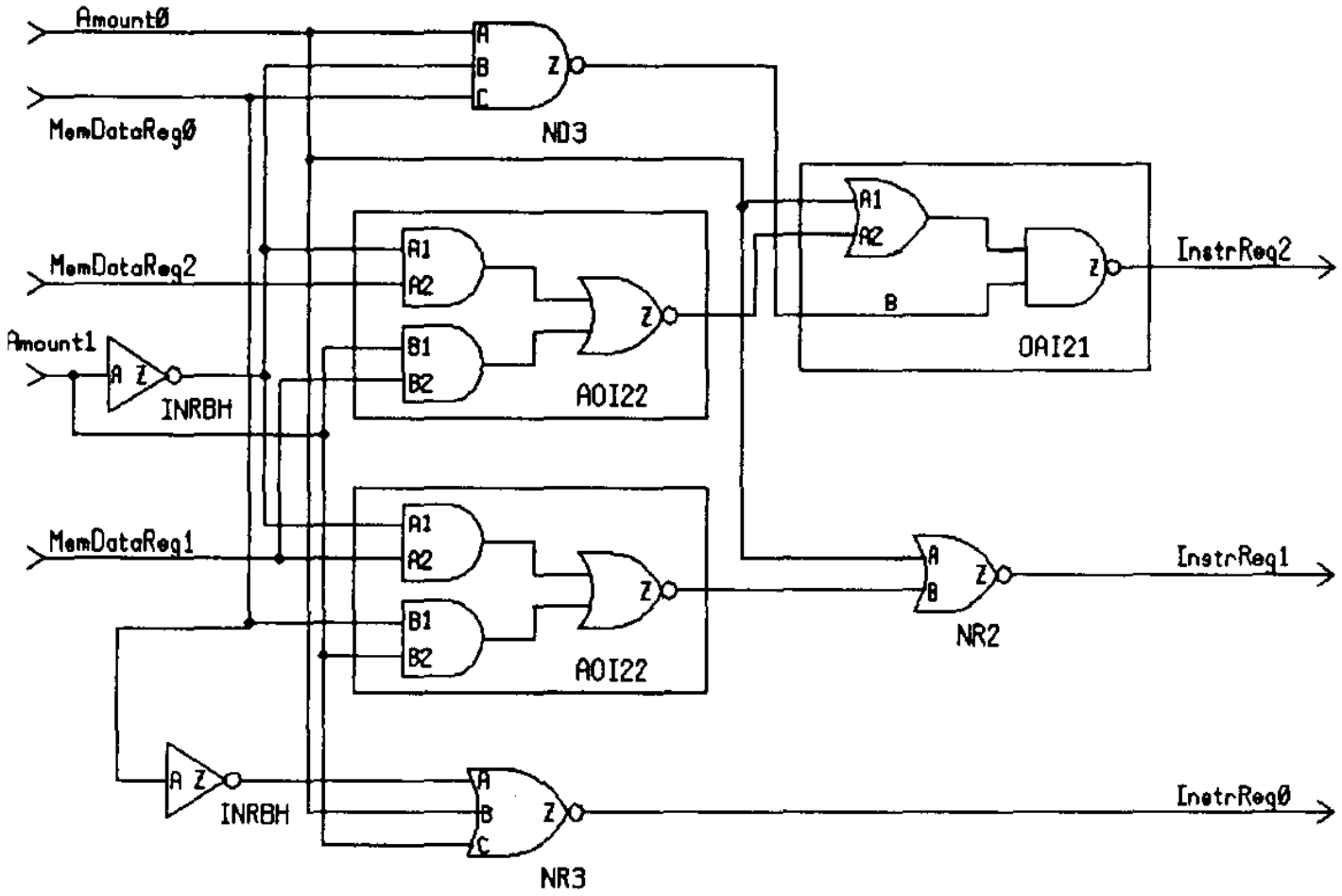


Constant shift.

Shift Operators

```
module VariableShift (MemDataReg, Amount, InstrReg);
    input [0:2] MemDataReg;
    input [0:1] Amount;
    output [0:2] InstrReg;
        assign InstrReg = MemDataReg >>Amount;
endmodule
```

Shift Operators

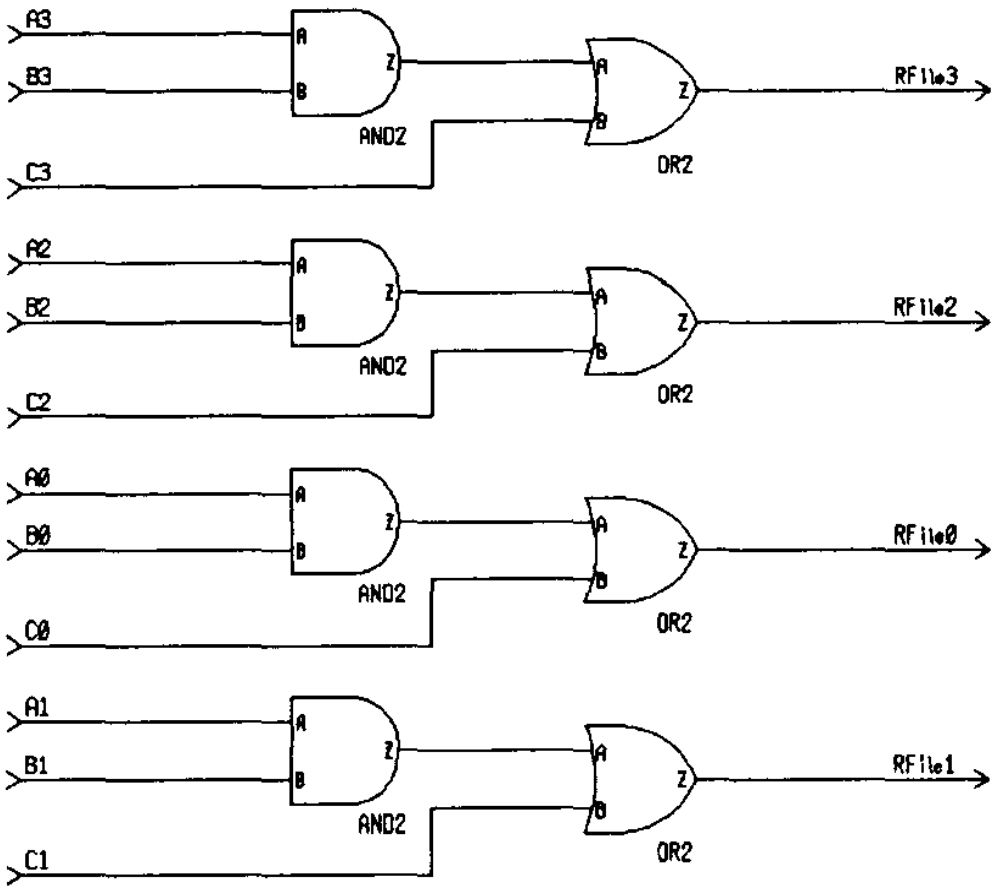


Variable shift

Vector Operations

```
module VectorOperations (A, B, C, RFile);
    input [3:0] A, B, C;
    output [3:0] RFile;

    assign RFile = (A & B) | C;
endmodule
```

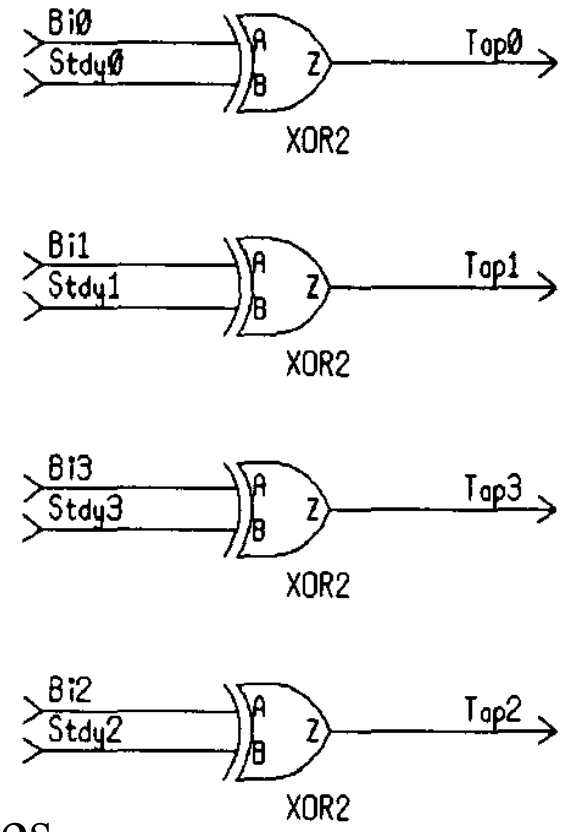
Vector Operations



Vector operations.

Vector Operations

```
module VectorOperands (Bi, Stdy, Tap);
    input [0:3] Bi, Stdy;
    output [0:3] Tap;
        assign Tap = Bi ^ Stdy;
endmodule
```



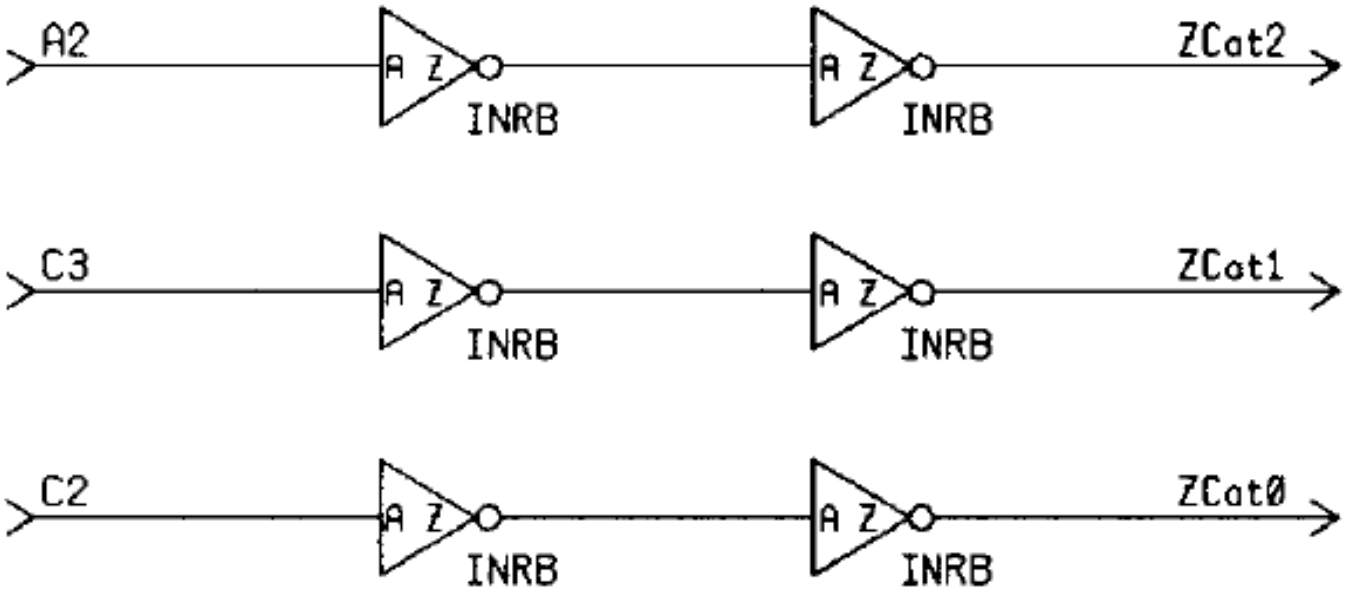
A bank of logic gates

Part-selects

```
module PartSelect (A, C, ZCat) ;
input [3:0] A, C;
output [3:0] ZCat;
    assign ZCat [2:0] = {A[2], C[3:2]};
endmodule
```

- Rule: Non-constant part-selects are not supported in VerilogHDL

Part-selects



Part-select example.

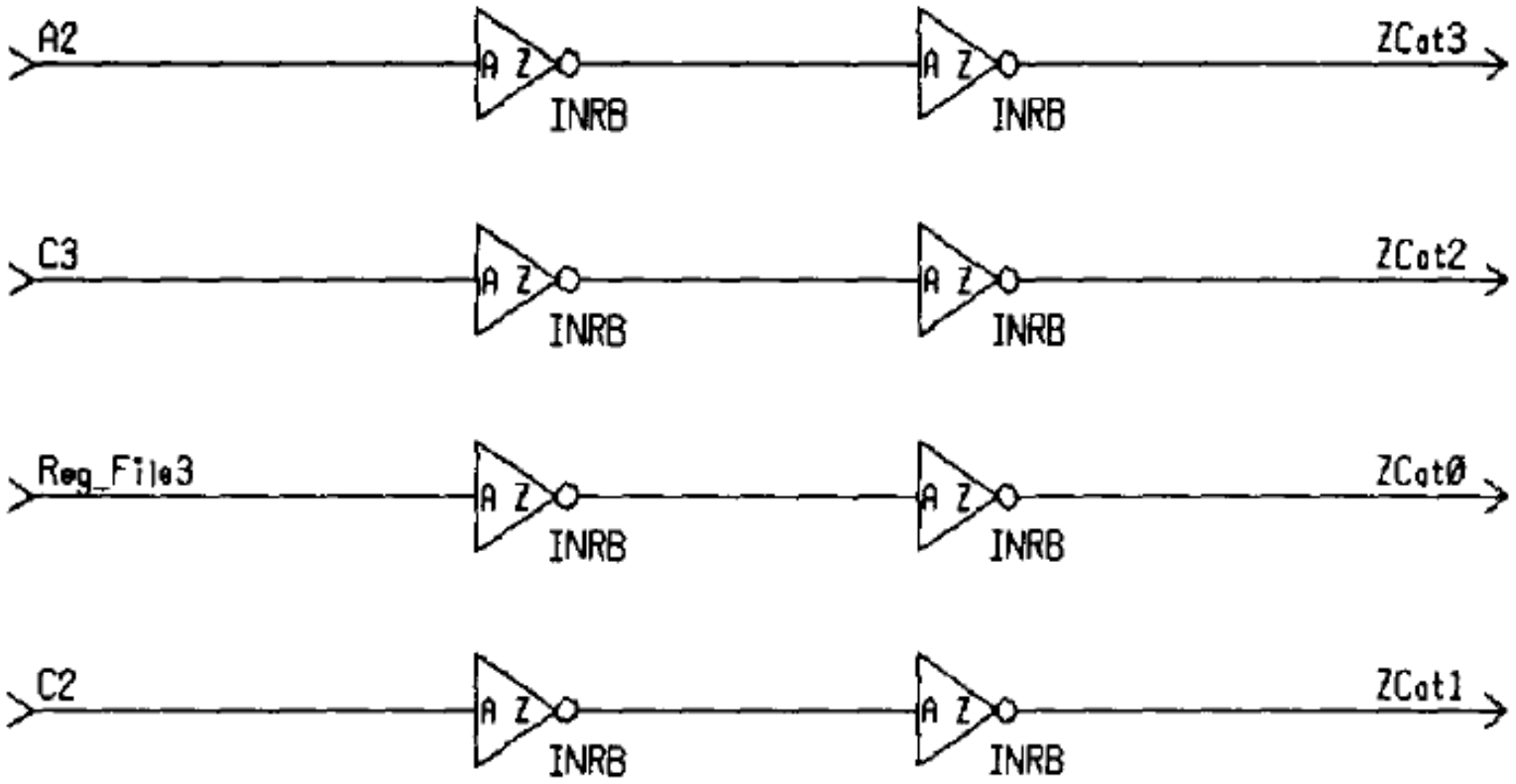
Bit-Selects and Assignment

- A bit-select can be a constant index or a non-constant index.
  - Constant Index in Expression/Target.
  - Non-Constant Index in Expression.
  - Non-Constant Index in Target

Constant Index

```
module Constantindex (A, C, Reg_File, ZCat);
    input [3:0] A, C;
    input [3:0] Reg_File;
    output [3:0] ZCat;

    assign ZCat[3:1] = {A[2], C[3:2]};
    assign ZCat[0] = Reg_File[3];
endmodule
```

Constant Index

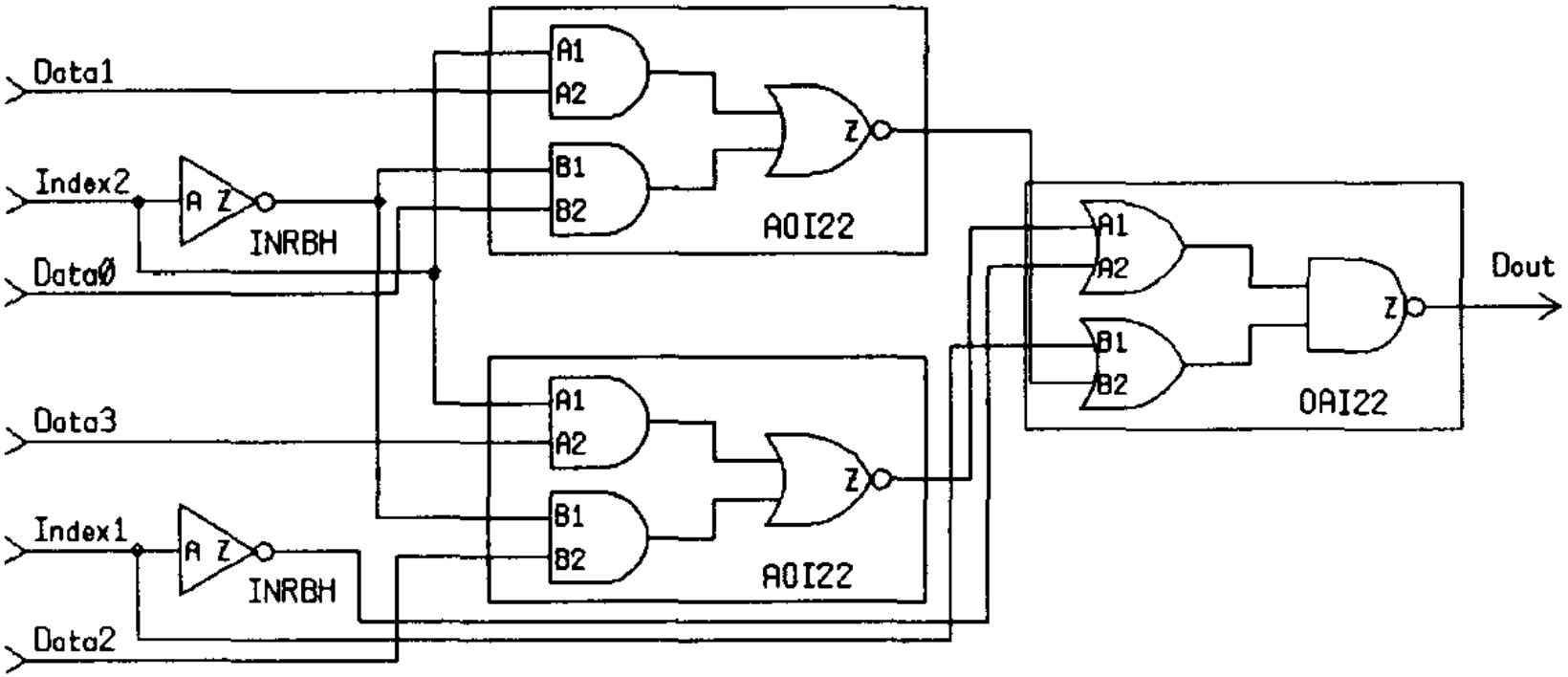


Constant bit-select.

Non-constant Index in Expression

```
module NonComputeRight (Data, Index, Dout);
input [0:3] Data;
input [1:2] Index;
output Dout;
    assign Dout = Data [Index];
endmodule
```

Non-constant Index in Expression

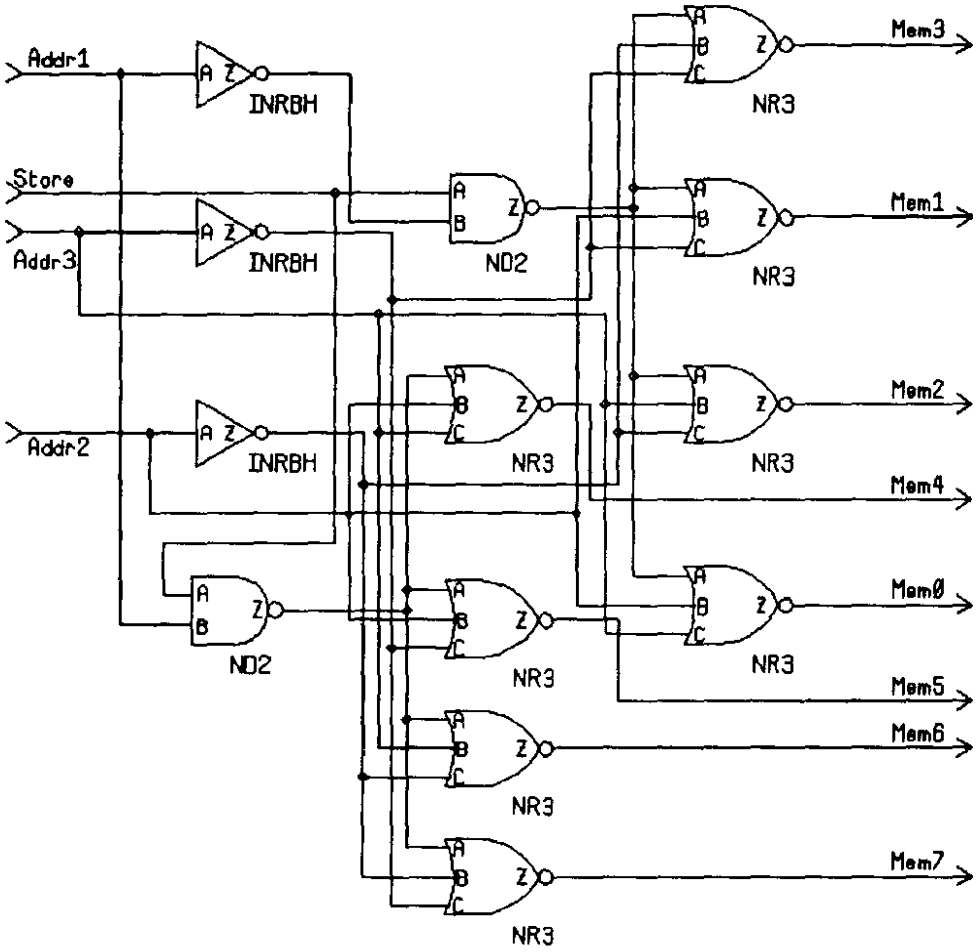


Non-constant bit-select generates a multiplexer

Non-constant Index in Target

```
module NonComputeLeft (Mem, Store, Addr) ;
output [7:0] Mem;
input Store;
input [1:3] Addr;
    assign Mem [Addr) = Store;
endmodule
```

Non-constant Index in Target



References

- Chapter 2, Verilog HDL Synthesis by J. Bhaskar

- Disclaimer: I don’t own all the slides, these slides are copied and adopted from various publicly resources available on the internet.

Thank you