Sharif University of Technology
Department of Computer Engineering

# Digital System Design
# Dataflow modelling

Siavash Bayat-Sarmadi

## Outline

- Continuous assignment statement
  - **assign**
- Delays in dataflow modelling
- Operators

## Introduction

- Gate-level modeling
  - Netlist of gates
  - Suitable for small designs
- Next level up: Dataflow modeling
  - Continuous assignment
    - The **assign** keyword

```
module and2(output out, input in1,in2);
 assign out = in1 & in2;
endmodule
```

## Rules

- LHS of assignment
  - Scalar net

```
assign out = in1 & in2;
```

  - Vector of net

```
assign addr[15:0]= addr1[15:0]&addr2[15:0];
```

  - Concatenation of nets

```
assign {c_out,sum[3:0]}=a[3:0]+b[3:0]+c_in;
```

    - Discussed later in this lecture
  - reg is illegal

## Rules (Cont'd.)

- RHS of assignment:
  - Any data type
    - reg
    - nets
    - …
- Always active
  - Change of any RHS variables, evaluates LHS data

## Shortcuts

- The ordinary way

```
wire out;
reg in1, in2;
assign out = in1 & in2;
```

- Implicit Continuous Assignment

```
reg in1, in2;
wire out = in1 & in2;
```

- Implicit Net Declaration

```
reg in1, in2;
assign out = in1 & in2;
```

## Delays

- Three ways to specify
  1. Regular assignment delay

```
assign #10 out = in1 & in2;
```

  2. Implicit continuous assignment delay

```
wire #10 out = in1 & in2;
```

## Delays (Cont’d.)

- Three ways to specify
  3. Net declaration delay
     - Any value change applied to the net is delayed accordingly
     - Also possible in gate-level modelling

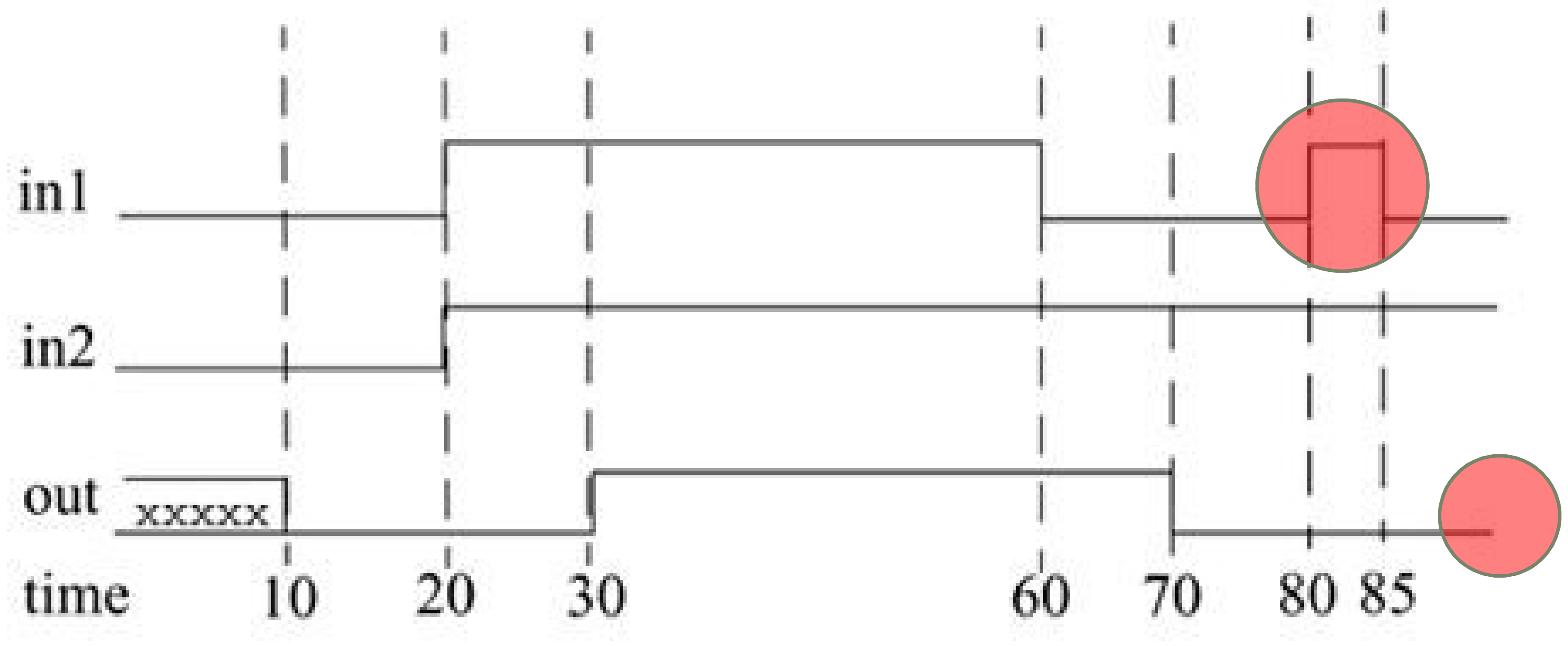
```
wire #10 out;
assign out = in1 & in2;
```

## Delays (Cont'd.)

- Control the time between:
  - The change in RHS & assignment to LHS
- All delays are inertial
  - Input pulses shorter than the delay of the assignment statement do not propagate to the output

# Delays (Cont'd.)

- Example
  - `assign #10 out = in1 & in2`



# Operators

| Operator category | Operator symbol |
|---|---|
| Arithmetic | * / + - % ** |
| Logical | ! && \|\| |
| Relational | < > <= >= |
| Equality | == != === !=== |
| Bitwise | ~ & \| ^ ^~ ~^ |
| Reduction | & ~& \| ~\| ^ ~^ ^~ |
| Shift | >> << >>> <<< |
| Concatenation | { } |
| Replication | { { } } |
| Conditional | ? : |

## Arithmetic Operators

- Unary
  - -4
  - +5
- Binary
  - Add +
  - Subtract -
  - Multiply *
  - Divide /
  - Power **
  - Modulus %

## Logical

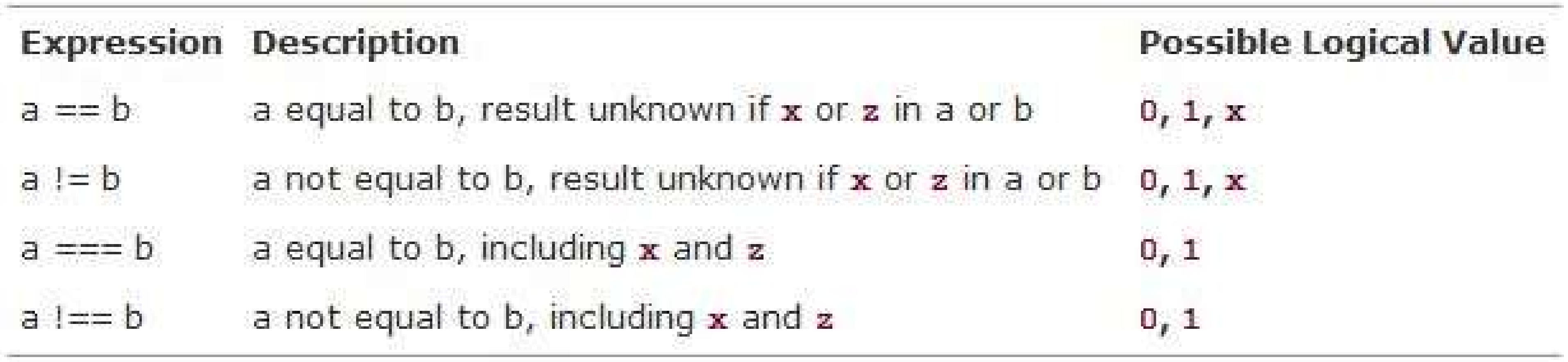
- Always evaluate to 0, 1 or x
- x value usu. treated as false
- Negation !
- And &&
- Or ||

# Relational operators

- >, <, >=, <=
- 0 if the relation is false
- 1 if the relation is true
- x if any of the operands has unknown x bits
- Example

| Operand1 | Operator | Operand2 | Result |
|---|---|---|---|
| 3'b0xz | < | 3'b111 | x |
| 3'b011 | > | 3'b110 | 0 |
| 3‘b100 | >= | 3'b011 | 1 |
| 3'b01z | <= | 3'b101 | x |

# Equality Operators



| Expression | Description | Possible Logical Value |
|---|---|---|
| a == b | a equal to b, result unknown if **x** or **z** in a or b | 0, 1, x |
| a != b | a not equal to b, result unknown if **x** or **z** in a or b | 0, 1, x |
| a === b | a equal to b, including **x** and **z** | 0, 1 |
| a !== b | a not equal to b, including **x** and **z** | 0, 1 |

Example

| Operand1 | Operator | Operand2 | Result |
|---|---|---|---|
| 3'b1xz | == | 3'b1xz | x |
| 3'b1xz | == | 3'b0xz | 0 |
| 3'b1xz | === | 3'b1xz | 1 |
| 3'b1xz | === | 3'b1zz | 0 |

## Bitwise Operators

- Negation ~
- And &
- Or |
- Xor ^
- Xnor ^~, ~^

## Reduction Operators

- Unary
- Perform on a single vector
- Produce one-bit result
- And &
- Nand ~&
- Or |
- Nor ~|
- Xor ^
- Xnor ^~, ~^

Example

| Operator | Operand | Result |
|---|---|---|
| ~& | 4'b1011 | 1 |
| ~\| | 4'b0110 | 0 |
| ^ | 4'b1011 | 1 |

## Shift Operators

- Logical
  - >>
  - <<
- Example
  - 4’b1100 >> 1
    - Result is 4’b0110

- Arithmetic
  - >>>
  - <<<
- Example
  - 4’b1100 >>> 1
    - Result is 4’b1110

## Concatenation Operator

- Appends multiple operands
- Sizes should be known

```
//A = 1'b1, B = 2'b00, C = 2'b10
Y = {A, B, C, 3'b001};//8'b10010001
Y = {A, B[0], C[1]};  //3'b101
```

## Replication Operator

- A number expressing replication

```
//A = 1'b1, B = 2'b00, C  = 2'b10
Y = { 4{A} }              //4'b1111
Y = { 4{A}, 2{B} }        //8'b11110000
Y = { 4{A}, 2{B}, C }/10'b1111000010
```

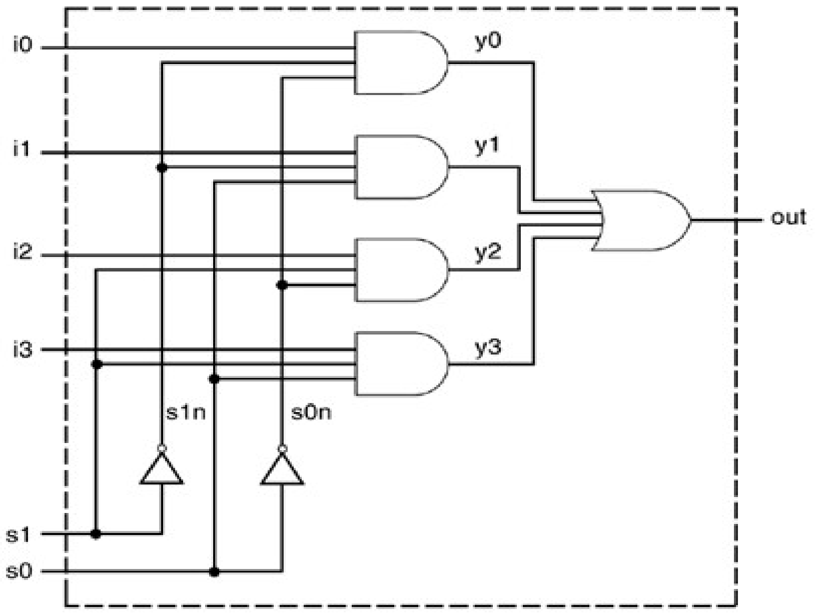
## Conditional Operator

- Ternary
- Usage
  - condition_expr ? true_expr : false_expr
- Produces 1-bit result
- Example: functionality of a 2-to-1 mux
  - `assign out = control ? in1 : in0;`

# Gate level vs Dataflow

- Dataflow

```
module mux4_to_1(output
out,input i0, i1, i2, i3,
s1, s0);
assign out = s1?(s0?i3:i2):
        (s0 ? i1 : i0);
endmodule
```



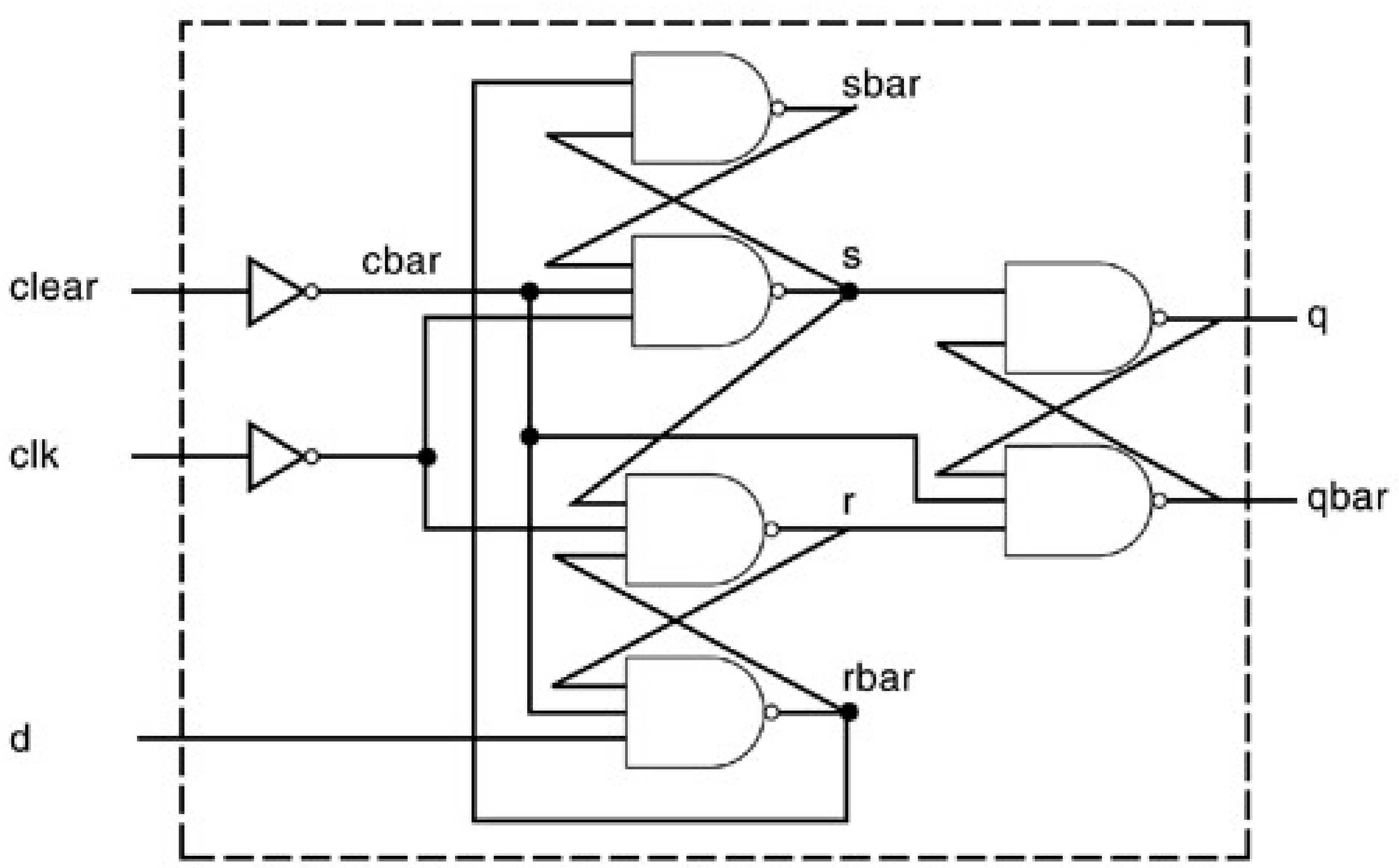
- Gate level

```
module mux4_to_1(output
out,input i0, i1, i2, i3,
s1,s0);
wire s1n, s2n;
wire y0, y1, y2, y3;
not (s1n, s1);
not (s0n, s0);
and (y0 , i0 , s1n, s0n);
and (y1, i1, s1n, s0);
and (y2 , i2 , s1, s0n);
and (y3, i3, s1, s0);
or (out, y0, y1, y2, y3);
endmodule
```

# Operator Precedence

| Operators | Operator symbols | Precedence |
|---|---|---|
| Unary | + - ! ~ | Highest precedence |
| Multiply, Divide, Modulus | * / % | |
| Add , Subtract | + - | |
| Shift | << >> | |
| Relational | < <= > >= | |
| Equality | == != === !== | |
| Reduction | & ~&<br>^ ^~<br>\| ~\| | |
| Logical | &&<br>\|\| | |
| Conditional | ? : | Lowest precedence |

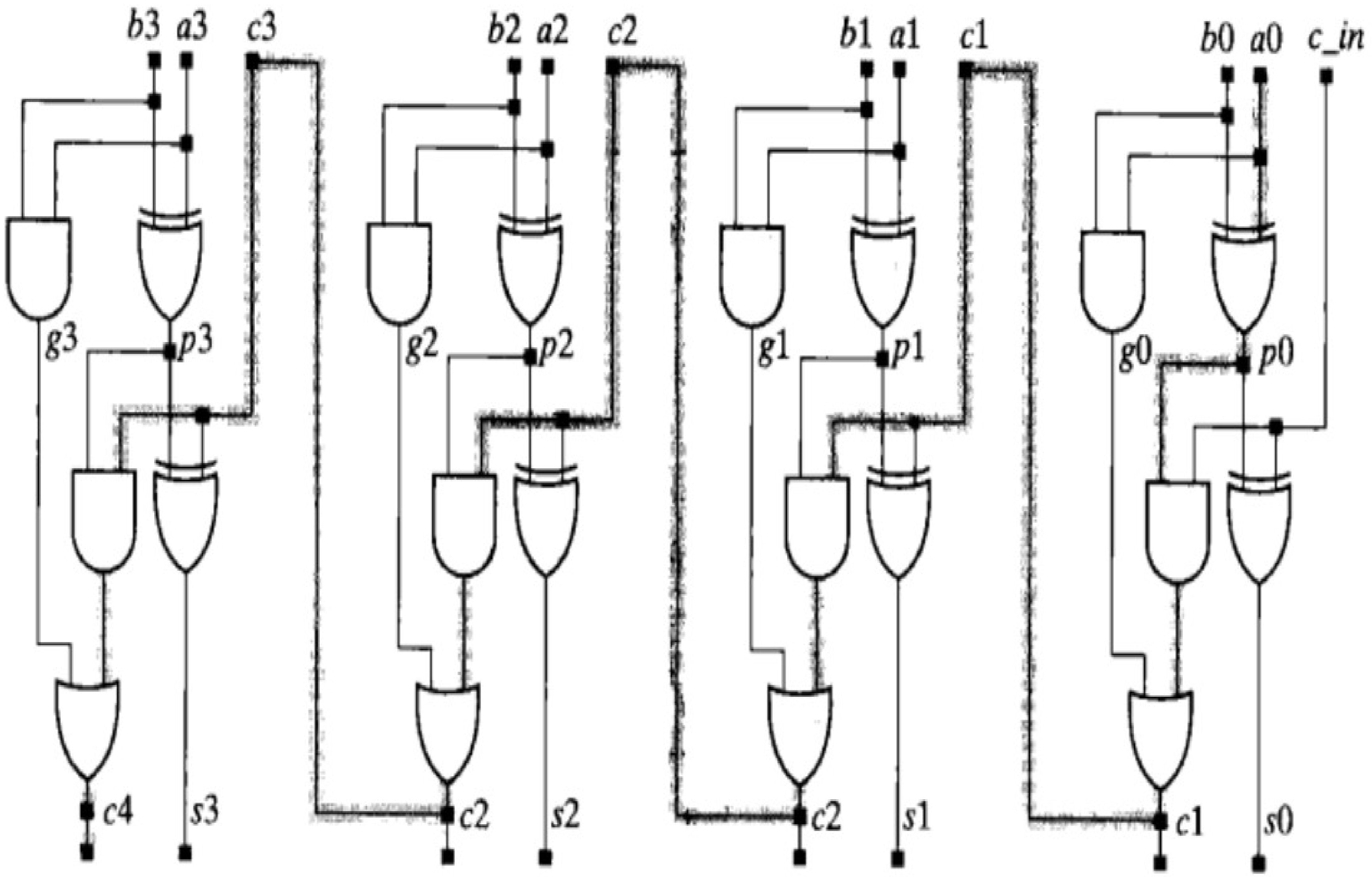
# Example: D-FF



# Example: Dataflow D-FF

```
module Dff(q, qbar, d, clear, clk);
output q, qbar;
input d, clear, clk;
wire cbar, s, sbar, r, rbar;
assign cbar = ~clear,
     s = ~(sbar & cbar & ~clk),
     r = ~(rbar & s & ~clk),
     sbar = ~(rbar & s),
     rbar = ~(r & cbar & d);
assign q = ~(s & qbar) ,
qbar = ~(q & r & cbar);
endmodule
```

# Example: Carry Ripple Adder



# Example: Carry Ripple Adder

```
module Add_prop_gen(output [3:0] sum,output
c_out, input [3:0] a, b, input c_in);
wire [3:0] carrychain;
wire [3:0] g = a & b;//carry generate
wire [3:0] p = a ^ b;//carry propagate
wire [4:0] shiftedcarry ={carrychain,c_in};
assign sum = p ^ shiftedcarry[3:0];
assign c_out = shiftedcarry[4];
assign carrychain[0] = g[0] | (p[0]&c_in);
```

## Example: Carry Ripple Adder

```
genvar i;
generate for(i = 1 ; i < 4 ; i = i + 1 )
assign carrychain[i] =
g[i] | (p[i] & carrychain[i-1]);
endgenerate
endmodule
```

Part-4

# 16-bit RISC Processor

DataFlow ALU