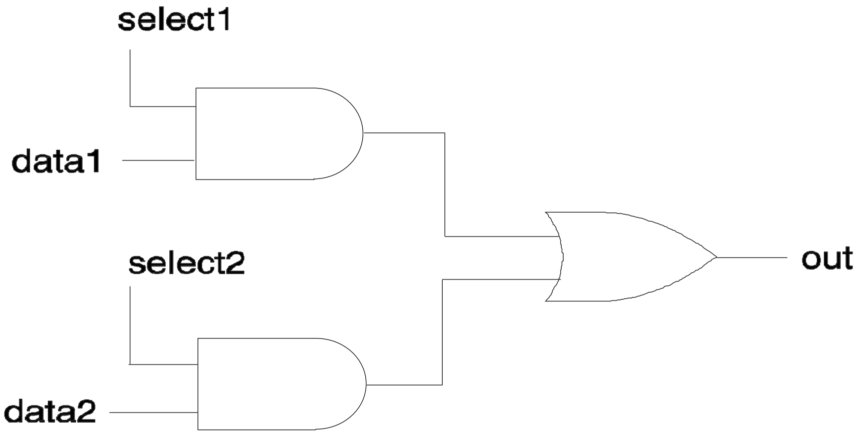
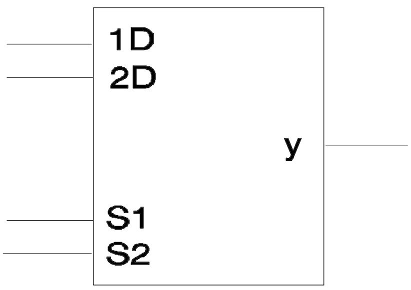
# PART 7

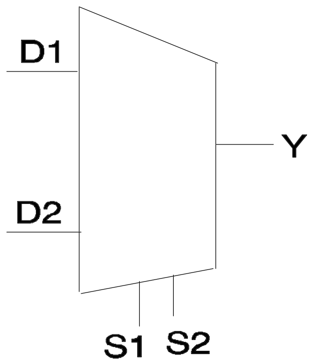
## DATAFLOW

- Constructs for dataflow descriptions
- Multiplexing and clocking
    - Selection constructs
    - Guarded assignments
- Multiple assignments
    - Resolutions
    - Anding, oring, wiring
    - Guarded signals
- State machines
    - Simple sequence detector
    - Multiple active states
- Open collectors
    - Using resolution functions
- A complete dataflow example
- Load dependent timing

# Dataflow.multiplexing



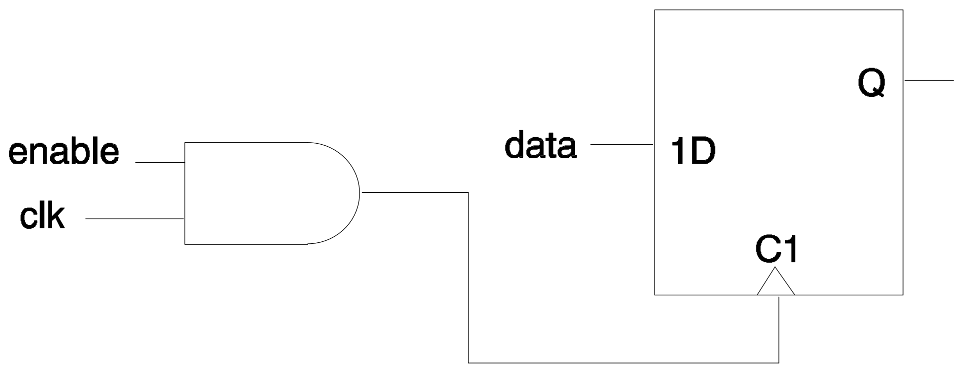



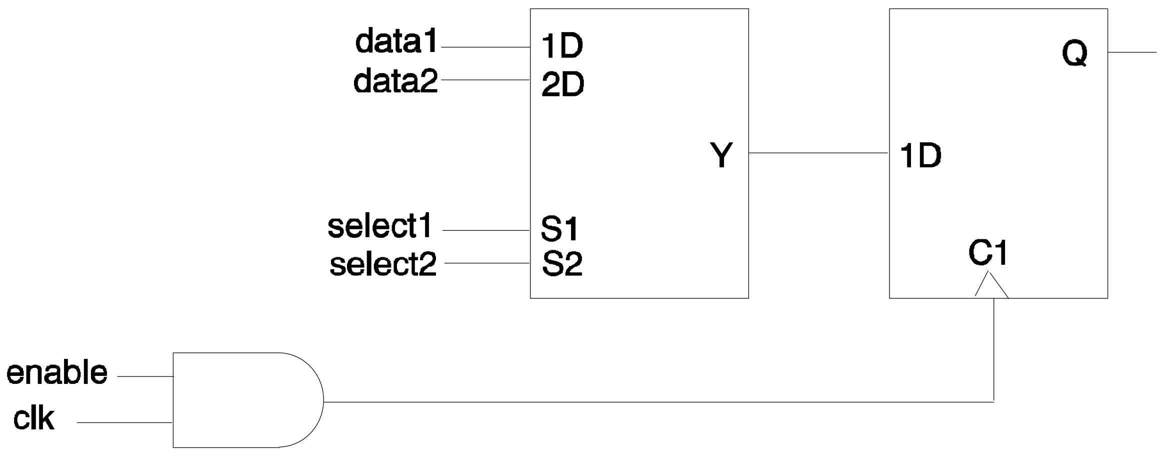

- Multiplexers are used for data selection



© Zainalabedin Navabi, 1994

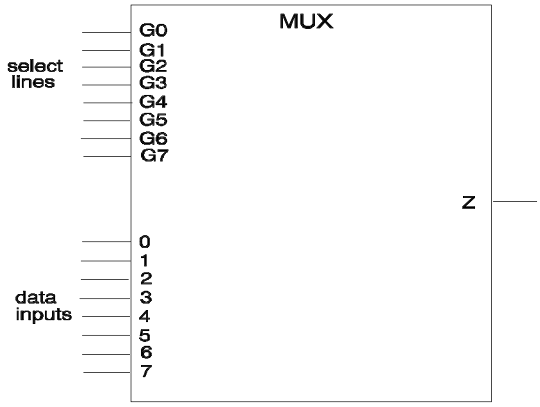
# Dataflow.multiplexing




- Flip flop clocking selects data
- Various forms of data selection may be combined
- Will show language constructs for such selections

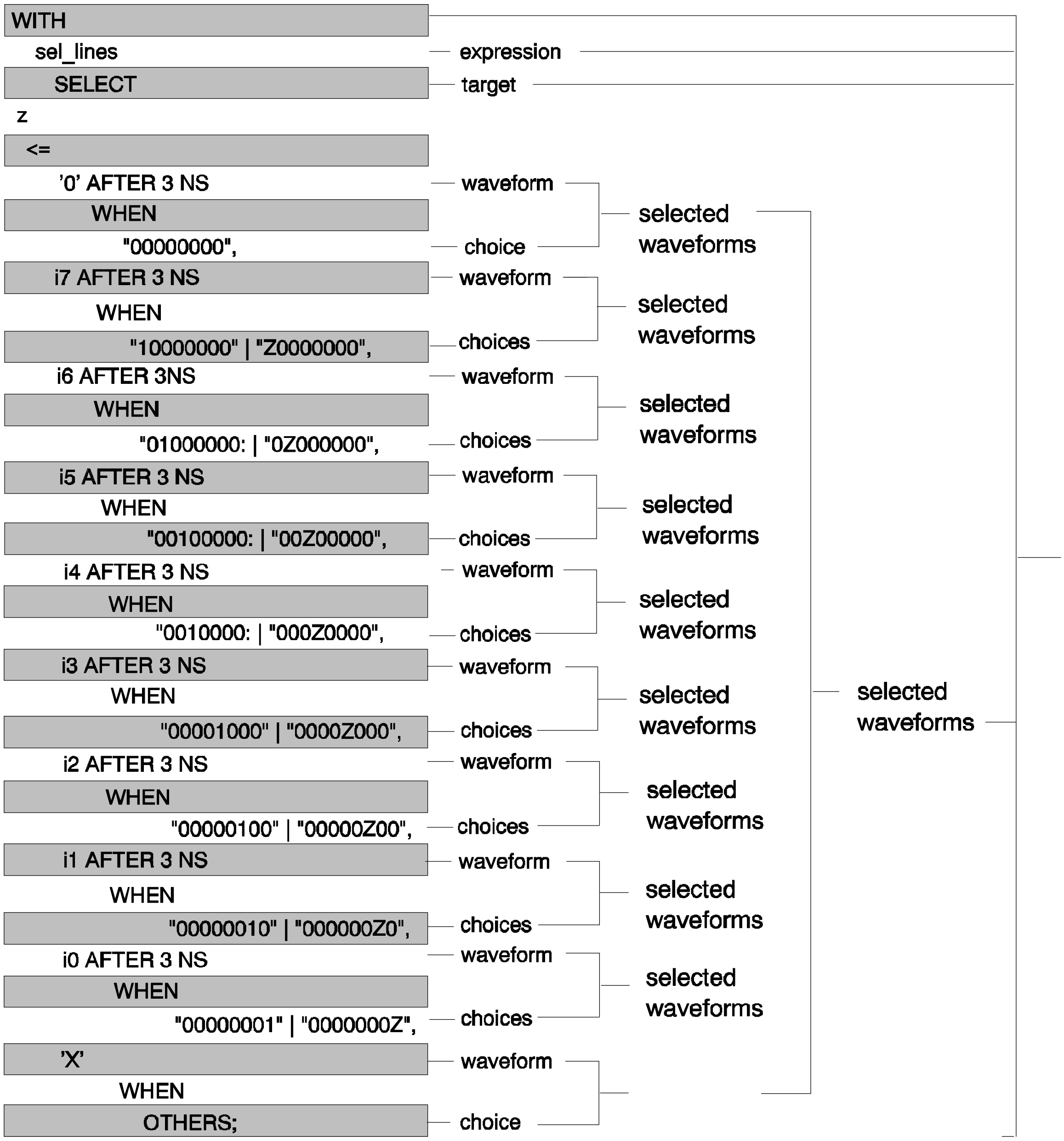
# Dataflow.multiplexing




```
USE WORK.basic_utilities.ALL;
--  FROM PACKAGE USE: qit, qit_vector
ENTITY mux_8_to_1 IS
  PORT (i7, i6, i5, i4, i3, i2, i1, i0 : IN qit;
        s7, s6, s5, s4, s3, s2, s1, s0 : IN qit; z : OUT qit );
END mux_8_to_1;
--
ARCHITECTURE dataflow OF mux_8_to_1 IS
  SIGNAL sel_lines : qit_vector ( 7 DOWNTO 0);
BEGIN
  sel_lines <= s7&s6&s5&s4&s3&s2&s1&s0;
  WITH sel_lines SELECT
    z <= '0' AFTER 3 NS WHEN "00000000",
         i7 AFTER 3 NS WHEN "10000000" | "Z0000000",
         i6 AFTER 3 NS WHEN "01000000" | "0Z000000",
         i5 AFTER 3 NS WHEN "00100000" | "00Z00000",
         i4 AFTER 3 NS WHEN "00010000" | "000Z0000",
         i3 AFTER 3 NS WHEN "00001000" | "0000Z000",
         i2 AFTER 3 NS WHEN "00000100" | "00000Z00",
         i1 AFTER 3 NS WHEN "00000010" | "000000Z0",
         i0 AFTER 3 NS WHEN "00000001" | "0000000Z",
         'X' WHEN OTHERS;
END dataflow;
```

- Description of a simple multiplexer
- Selected signal assignment is used

# Dataflow.multiplexing




- Selected signal assignment
- Selected waveforms use choice or choices

# Dataflow.multiplexing


```
USE WORK.basic_utilities.ALL;
--  FROM PACKAGE USE : qit_vector
ENTITY dcd_3_to_8 IS
  PORT (adr : IN qit_vector (2 DOWNTO 0);
            so : OUT qit_vector (7 DOWNTO 0));
END dcd_3_to_8;
--
ARCHITECTURE dataflow OF dcd_3_to_8 IS
BEGIN
  WITH adr SELECT
    so <= "00000001" AFTER 2 NS WHEN "000",
            "00000010" AFTER 2 NS WHEN "00Z" | "001",
            "00000100" AFTER 2 NS WHEN "0Z0" | "010",
            "00001000" AFTER 2 NS WHEN "0ZZ" | "0Z1" | "01Z" | "011",
            "00010000" AFTER 2 NS WHEN "100" | "Z00",
            "00100000" AFTER 2 NS WHEN "Z0Z" | "Z01" | "10Z" | "101",
            "01000000" AFTER 2 NS WHEN "ZZ0" | "Z10" | "1Z0" | "110",
            "10000000" AFTER 2 NS WHEN "ZZZ" | "ZZ1" | "Z1Z" | "Z11" |
                                                  "1ZZ" | "1Z1" | "11Z" | "111",
            "XXXXXXXX" WHEN OTHERS;
END dataflow;
```

- Decoder description uses selected signal assignment
- All possibilities must be considered

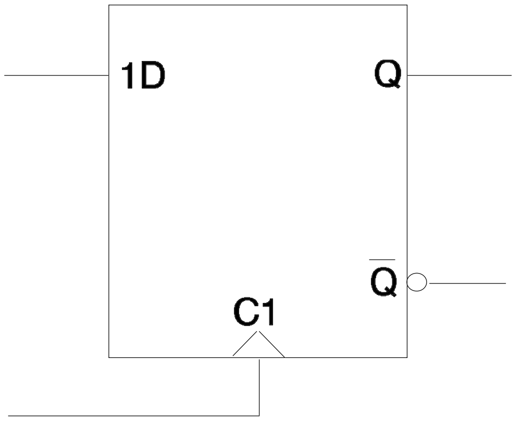
# Dataflow.clocking

```
ENTITY d_flipflop IS
  GENERIC (delay1 : TIME := 4 NS; delay2 : TIME := 5 NS);
  PORT (d, c : IN BIT; q, qb : OUT BIT);
END d_flipflop;
--
ARCHITECTURE assigning OF d_flipflop IS
  SIGNAL internal_state : BIT;
BEGIN
  internal_state <=   d WHEN ( c = '1' AND NOT c'STABLE ) ELSE
                      internal_state;
  q <= internal_state AFTER delay1;
  qb <= NOT internal_state AFTER delay2;
END assigning;
```

- A simple flipflop uses *internal_state*
- On clock edge *d* is transferred to *internal_state*
- Events on *internal_state* cause assignments to *q* and *qb*

# Dataflow.clocking

```
ARCHITECTURE guarding OF d_flipflop IS
BEGIN
  ff: BLOCK ( c = '1' AND NOT c'STABLE )
  BEGIN
    q <= GUARDED d AFTER delay1;
    qb <= GUARDED NOT d AFTER delay2;
  END BLOCK ff;
END guarding;
```




- Better representation of clocking **disconnects** *d* from *q*
- Disconnection is specified by GUARDED
- GUARDED assignments are guarded by guard expression
- Can also guard selected and conditional signal assignments

# Dataflow.clocking

```
ENTITY flipflop_test IS END flipflop_test;
--
ARCHITECTURE input_output OF flipflop_test IS
  COMPONENT
       flop PORT (d, c : IN BIT; q, qb : OUT BIT);
  END COMPONENT;
  FOR c1 : flop USE ENTITY WORK.d_flipflop (assigning);
  FOR c2 : flop USE ENTITY WORK.d_flipflop (guarding);
  SIGNAL dd, cc, q1, q2, qb1, qb2 : BIT;
BEGIN
  cc <= NOT cc AFTER 400 NS WHEN NOW < 2 US ELSE cc;
  dd <= NOT dd AFTER 1000 NS WHEN NOW < 2 US ELSE dd;
  c1: flop PORT MAP (dd, cc, q1, qb1);
  c2: flop PORT MAP (dd, cc, q2, qb2);
END input_output;
```

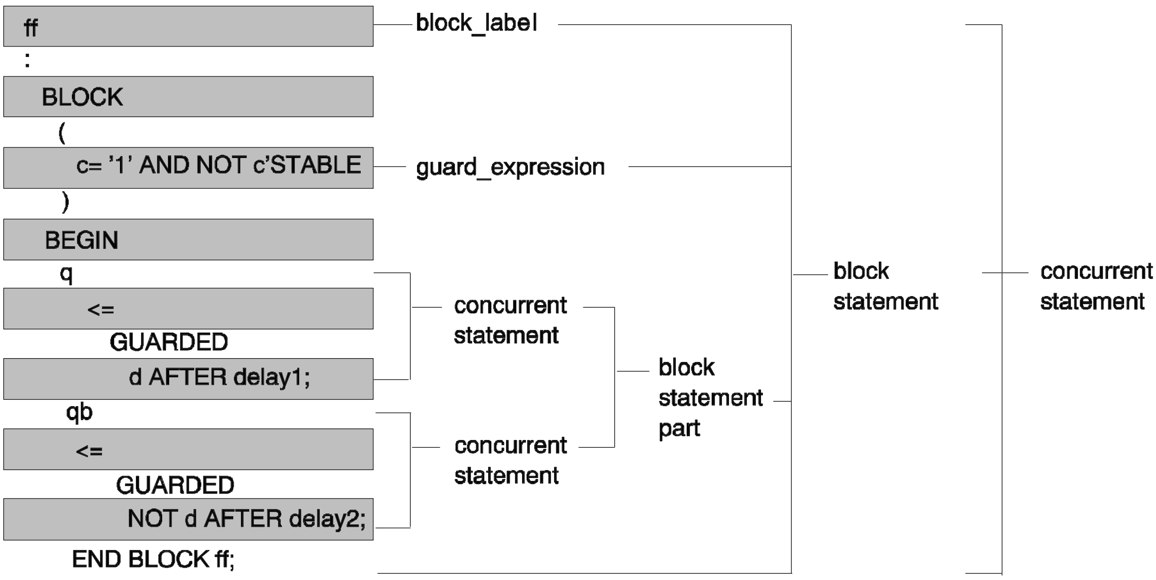
- Testbench tests and verifies both descriptions
- A simple method for generation of periodic signals

# Dataflow.clocking

| TIME (NS) | CC | DD | Q1 | SIGNALS Q2 | QB1 | QB2 | C1: STATE | C2.FF: GUARD |
|---|---|---|---|---|---|---|---|---|
| 0 | '0' | '0' | '0' | '0' | '0' | '0' | '0' | FALSE |
| +*d* | ... | ... | ... | ... | ... | ... | '0' | ..... |
| **4** | ... | ... | **'0'** | ... | ... | ... | ... | ..... |
| 5 | ... | ... | ... | ... | '1' | ... | ... | ..... |
| 400 | '1' | ... | ... | ... | ... | ... | ... | TRUE |
| +*d* | ... | ... | ... | ... | ... | ... | **'0'** | **FALSE** |
| +*d* | ... | ... | ... | ... | ... | ... | **'0'** | ..... |
| 404 | ... | ... | ... | '0' | ... | ... | ... | ..... |
| 405 | ... | ... | ... | ... | ... | '1' | ... | ..... |
| 800 | '0' | ... | ... | ... | ... | ... | ... | FALSE |
| +*d* | ... | ... | ... | ... | ... | ... | '0' | FALSE |
| +*d* | ... | ... | ... | ... | ... | ... | '0' | ..... |
| 1000 | ... | '1' | ... | ... | ... | ... | ... | ..... |
| +*d* | ... | ... | ... | ... | ... | ... | '0' | ..... |
| **1200** | **'1'** | ... | ... | ... | ... | ... | ... | **TRUE** |
| +*d* | ... | ... | ... | ... | ... | ... | '1' | FALSE |
| +*d* | ... | ... | ... | ... | ... | ... | '1' | ..... |
| 1204 | ... | ... | '1' | '1' | ... | ... | ... | ..... |
| 1205 | ... | ... | ... | ... | '0' | '0' | ... | ..... |
| 1600 | '0' | ... | ... | ... | ... | ... | ... | FALSE |
| +*d* | ... | ... | ... | ... | ... | ... | '1' | FALSE |
| +*d* | ... | ... | ... | ... | ... | ... | '1' | ..... |
| 2000 | '1' | '0' | ... | ... | ... | ... | ... | TRUE |
| +*d* | '1' | '0' | ... | ... | ... | ... | '0' | FALSE |
| +*d* | ... | ... | ... | ... | ... | ... | '0' | ..... |
| 2004 | ... | ... | '0' | '0' | ... | ... | ... | ..... |
| 2005 | ... | ... | ... | ... | '1' | '1' | ... | ..... |

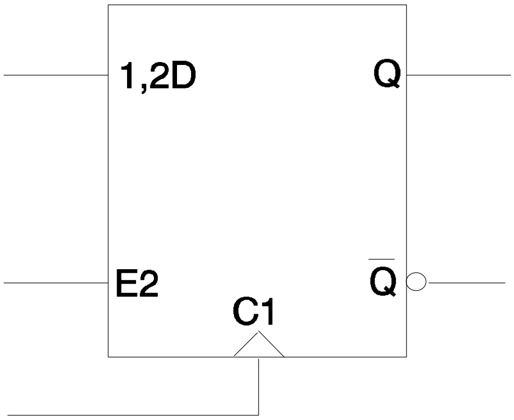
- In assigning:
  Two transactions on *internal_state* for every clock edge
  Transaction on *q1* at time 0004, is due to initialization
- In guarding:
  c2.ff : GUARD sees GUARD inside guarding
  Guard expression is only TRUE for 1 delta

# Dataflow.clocking




- Syntax details of guarded blocks

# Dataflow.clocking




```
ENTITY de_flipflop IS
  GENERIC (delay1 : TIME := 4 NS; delay2 : TIME := 5 NS);
  PORT (d, e, c : IN BIT; q, qb : OUT BIT);
END de_flipflop;
--
ARCHITECTURE guarding OF de_flipflop IS
BEGIN
  edge: BLOCK ( c = '1' AND NOT c'STABLE )
  BEGIN
    gate: BLOCK ( e = '1' AND GUARD )
    BEGIN
      q <= GUARDED d AFTER delay1;
      qb <= GUARDED NOT d AFTER delay2;
    END BLOCK gate;
  END BLOCK edge;
END guarding;
```

| |
|---|
| (e= '1') AND ( c= '1' AND NOT c'STABLE) |

- Can nest block statements
- Combining guard expressions must be done explicitly
- Implicit GUARD signals in each block

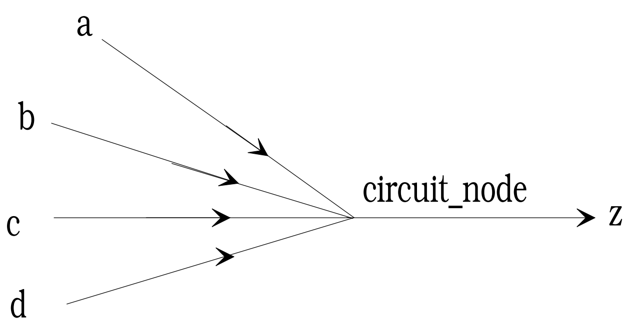
# Dataflow.clocking

```
ENTITY flipflop_test IS END flipflop_test;
--
ARCHITECTURE input_output OF flipflop_test IS
  COMPONENT
       ff1 PORT (d, e, c : IN BIT; q, qb : OUT BIT);
  END COMPONENT;
  FOR c1 : ff1 USE ENTITY WORK.de_flipflop (guarding);
  SIGNAL dd, ee, cc, q1, qb1 : BIT;
BEGIN
  cc <= NOT cc AFTER 400 NS WHEN NOW < 3 US ELSE cc;
  dd <= NOT dd AFTER 1000 NS WHEN NOW < 3 US ELSE dd;
  ee <= '1', '0' AFTER 2200 NS;
  c1: ff1 PORT MAP (dd, ee, cc, q1, qb1);
END input_output;
```

| TIME | SIGNALS | | | | |
|---|---|---|---|---|---|
| (NS) | CC | EE | DD | Q1 | QB1 |
| 0 | '0' | '0' | '0' | '0' | '0' |
| +*d* | ... | '1' | ... | ... | ... |
| 400 | '1' | ... | ... | ... | ... |
| 404 | ... | ... | ... | '0' | ... |
| 405 | ... | ... | ... | ... | '1' |
| 800 | '0' | ... | ... | ... | ... |
| 1000 | ... | ... | '1' | ... | ... |
| 1200 | '1' | ... | ... | ... | ... |
| 1204 | ... | ... | ... | '1' | ... |
| 1205 | ... | ... | ... | ... | '0' |
| 1600 | '0' | ... | ... | ... | ... |
| 2000 | '1' | ... | '0' | ... | ... |
| 2004 | ... | ... | ... | '0' | ... |
| 2005 | ... | ... | ... | ... | '1' |
| **2200** | ... | **'0'** | ... | ... | ... |
| 2400 | '0' | ... | ... | ... | ... |
| 2800 | '1' | ... | ... | ... | ... |
| 3000 | ... | ... | '1' | ... | ... |
| +*d* | ... | ... | '1' | ... | ... |

- Testbench verifies operation of *de_flipflop*
- After 2200 *q1* is disconnected from *d*

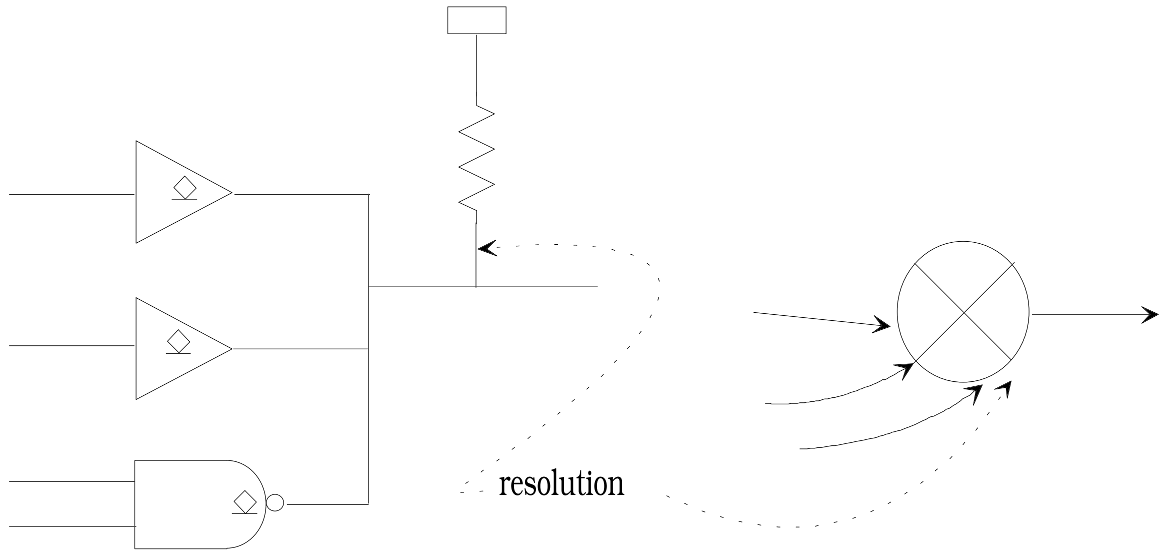
# Dataflow.resolution




```
USE WORK.basic_utilities.ALL;
--  FROM PACKAGE USE: qit
ENTITY y_circuit IS
   PORT (a, b, c, d : IN qit; z : OUT qit);
END y_circuit;
--
ARCHITECTURE smoke_generator OF y_circuit IS
   SIGNAL circuit_node : qit;
BEGIN
   circuit_node <= a;
   circuit_node <= b;
   circuit_node <= c;
   circuit_node <= d;
   z <= circuit_node;
END smoke_generator;
```

- Normally several sources cannot drive a signal
- Real circuits smoke

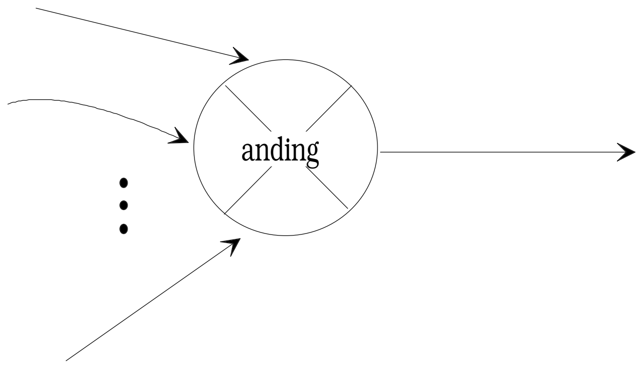
# Dataflow.resolution




- Multiple drivers is possible only if a resolution exists
- Example in hardware is "open collector"
- Pull_up provides resolution
- A convenient symbol is shown

# Dataflow.resolution

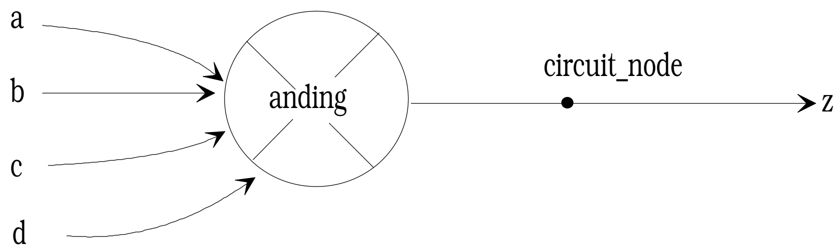
```
--  FROM basic_utilities USE qit, qit_vector, "AND"
FUNCTION anding ( drivers : qit_vector) RETURN qit IS
  VARIABLE accumulate : qit := '1';
BEGIN
  FOR i IN drivers'RANGE LOOP
    accumulate := accumulate AND drivers(i);
  END LOOP;
  RETURN accumulate;
END anding;
```




- VHDL uses resolution functions
- Resolving multiple values to a single value
- Function parameter must be vector of its return type
- All drivers are checked

# Dataflow.resolution

```
USE WORK.basic_utilities.ALL;
--  FROM PACKAGE USE: qit
ARCHITECTURE wired_and OF y_circuit IS
   FUNCTION anding ( drivers : qit_vector) RETURN qit IS
     VARIABLE accumulate : qit := '1';
   BEGIN
     FOR i IN drivers'RANGE LOOP
       accumulate := accumulate AND drivers(i);
     END LOOP;
     RETURN accumulate;
   END anding;
   SIGNAL circuit_node : anding qit;
BEGIN
   circuit_node <= a;
   circuit_node <= b;
   circuit_node <= c;
   circuit_node <= d;
   z <= circuit_node;
END wired_and;
```




- Specify anding for the resolution on *circuit_node*
- Type of *circuit_node* is a subtype of *qit*
- ANDing simultaneously receives all drivers

# Dataflow.resolution

```
FUNCTION oring ( drivers : qit_vector) RETURN qit IS
  VARIABLE accumulate : qit := '0';
BEGIN
  FOR i IN drivers'RANGE LOOP
    accumulate := accumulate OR drivers(i);
  END LOOP;
  RETURN accumulate;
END oring;
```

A subtype:

```
SUBTYPE ored_qit IS oring qit;
. . .
-- The following declarations are equivalent
SIGNAL t : ored_qit;
SIGNAL t : oring qit;
```

For Arrays:

```
TYPE ored_qit_vector IS ARRAY (NATURAL RANGE<>) OF ored_qit;
. . .
SIGNAL t_byte : ored_qit_vector (7 DOWNTO 0);
```

- Another example of resolution function
- Must have subtype for resolved array types

# Dataflow.resolution

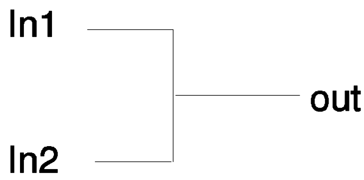
```
ARCHITECTURE multiple_assignments OF mux_8_to_1 IS
  SIGNAL t : ored_qit
BEGIN
  t <= i7 AND s7;
  t <= i6 AND s6;
  t <= i5 AND s5;
  t <= i4 AND s4;
  t <= i3 AND s3;
  t <= i2 AND s2;
  t <= i1 AND s1;
  t <= i0 AND s0;
  z <= t;
END multiple_assignments;
```

- Multiplexer uses implicit ORing on *t*
- AND_OR logic is realized

# Dataflow.resolution

```
FUNCTION wire (a, b : qit) RETURN qit IS
  CONSTANT qit_and_table : qit_2d := (
                          ('0','X','0','X'),
                          ('X','1','1','X'),
                          ('0','1','Z','X'),
                          ('X','X','X','X'));
BEGIN
  RETURN qit_and_table (a, b);
END wire;
```

| In2: \ In1: | 0 | 1 | Z | X |
|---|---|---|---|---|
| 0 | 0 | X | 0 | X |
| 1 | X | 1 | 1 | X |
| Z | 0 | 1 | Z | X |
| X | X | X | X | X |




Out

```
FUNCTION wiring ( drivers : qit_vector) RETURN qit IS
  VARIABLE accumulate : qit := 'Z';  -- wire value with no drivers
BEGIN
  FOR i IN drivers'RANGE LOOP
    accumulate := wire (accumulate, drivers(i));
  END LOOP;
  RETURN accumulate;
END wiring;

FUNCTION wiring ( drivers : qit_vector) RETURN qit;
SUBTYPE wired_qit IS wiring qit;
TYPE wired_qit_vector IS ARRAY (NATURAL RANGE <>) OF wired_qit;
```

- A complete resolution for *wiring*

# Dataflow.resolution

```
--  Declarations, can go in a package declaration

  FUNCTION oring ( drivers : qit_vector) RETURN qit;
  SUBTYPE ored_qit IS oring qit;
  TYPE ored_qit_vector IS ARRAY (NATURAL RANGE <>) OF ored_qit;



--  Function definition, can go in a package body

  FUNCTION oring ( drivers : qit_vector) RETURN qit IS
    VARIABLE accumulate : qit := '0';
  BEGIN
    FOR i IN drivers'RANGE LOOP
      accumulate := accumulate OR drivers(i);
    END LOOP;
    RETURN accumulate;
  END oring;
```

- A complete resolution for *oring*
- OR for BIT is already defined
- If no drivers, '0' is returned

# Dataflow.resolution.application

```
bi : BLOCK (si = '1' OR si = 'z')
BEGIN
   t <= GUARDED ii;
END BLOCK;
    . . .
```

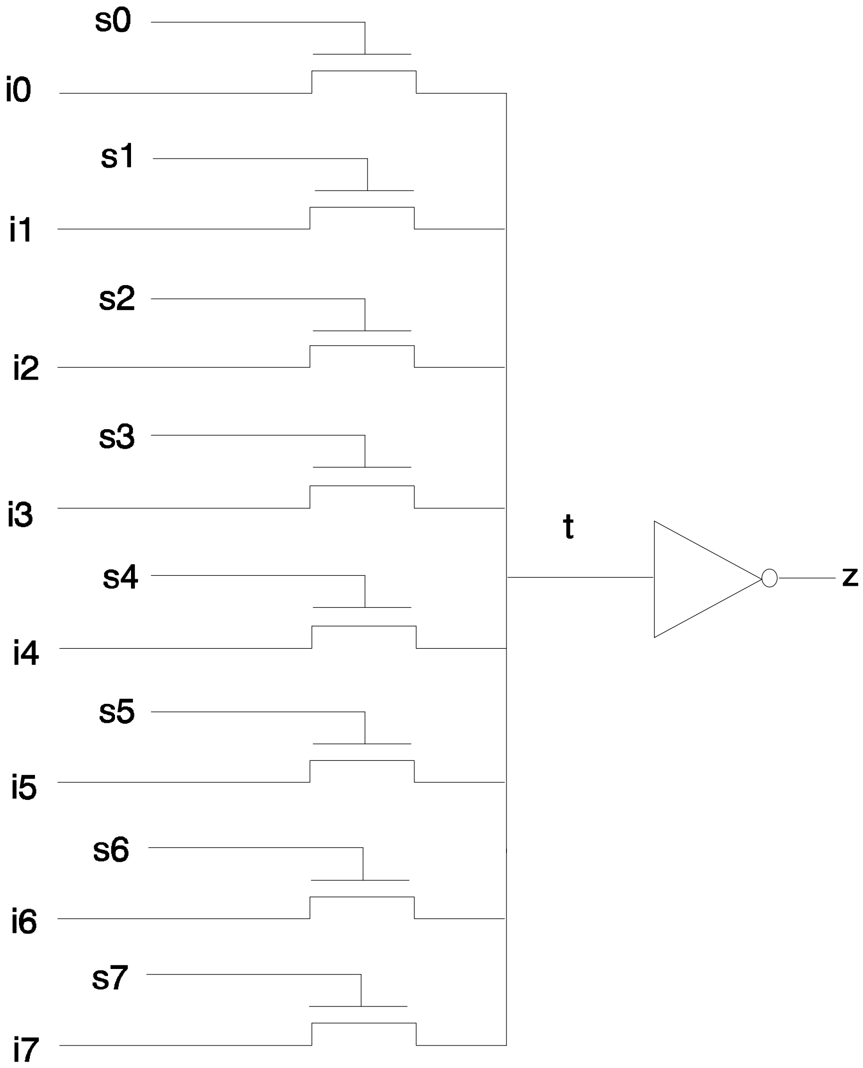
- A mux made of pass transistors
- Each $i_i$ **connects to** $t$ if $s_i$ is '1'
- $i_i$ is disconnected from $t$ if $s_i$ is '0'

# **Dataflow.resolution.application**

```
USE WORK.basic_utilities.ALL;
--  FROM PACKAGE USE: wired_qit
ARCHITECTURE multiple_guarded_assignments OF mux_8_to_1 IS
  SIGNAL t : wiring qit BUS;
BEGIN
  b7: BLOCK (s7 = '1' OR s7 = 'Z') BEGIN  t <= GUARDED i7; END BLOCK;
  b6: BLOCK (s6 = '1' OR s6 = 'Z') BEGIN  t <= GUARDED i6; END BLOCK;
  b5: BLOCK (s5 = '1' OR s5 = 'Z') BEGIN  t <= GUARDED i5; END BLOCK;
  b4: BLOCK (s4 = '1' OR s4 = 'Z') BEGIN  t <= GUARDED i4; END BLOCK;
  b3: BLOCK (s3 = '1' OR s3 = 'Z') BEGIN  t <= GUARDED i3; END BLOCK;
  b2: BLOCK (s2 = '1' OR s2 = 'Z') BEGIN  t <= GUARDED i2; END BLOCK;
  b1: BLOCK (s1 = '1' OR s1 = 'Z') BEGIN  t <= GUARDED i1; END BLOCK;
  b0: BLOCK (s0 = '1' OR s0 = 'Z') BEGIN  t <= GUARDED i0; END BLOCK;
  z <= t;
END multiple_guarded_assignments;
```

- Disconnection is realized by block statements
- If all drivers are disconnected
  - Real hardware returns to 'Z'
  - Use BUS to implement this behavior
- Default in wire is 'Z'
- This default is used if *wiring* is called with *null*
- **Last disconnection causes call to wiring with Null**

# Dataflow.resolution.application




- Circuit shows a register effect
- *t* retains value after last disconnection

# Dataflow.resolution.application

```
USE WORK.basic_utilities.ALL;
--  FROM PACKAGE USE: qit, wired_qit
ENTITY  multiplexed_half_register IS
  PORT (i7, i6, i5, i4, i3, i2, i1, i0 : IN qit;
            s7, s6, s5, s5, s4, s3, s2, s1, s0 : IN qit; z : OUT qit );
END multiplexed_half_register;
--
ARCHITECTURE guarded_assignments OF multiplexed_half_register IS
  SIGNAL t : wired_qit REGISTER;
BEGIN
  b7: BLOCK (s7 = '1' OR s7 = 'Z') BEGIN  t <= GUARDED i7; END BLOCK;
  b6: BLOCK (s6 = '1' OR s6 = 'Z') BEGIN  t <= GUARDED i6; END BLOCK;
  b5: BLOCK (s5 = '1' OR s5 = 'Z') BEGIN  t <= GUARDED i5; END BLOCK;
  b4: BLOCK (s4 = '1' OR s4 = 'Z') BEGIN  t <= GUARDED i4; END BLOCK;
  b3: BLOCK (s3 = '1' OR s3 = 'Z') BEGIN  t <= GUARDED i3; END BLOCK;
  b2: BLOCK (s2 = '1' OR s2 = 'Z') BEGIN  t <= GUARDED i2; END BLOCK;
  b1: BLOCK (s1 = '1' OR s1 = 'Z') BEGIN  t <= GUARDED i1; END BLOCK;
  b0: BLOCK (s0 = '1' OR s0 = 'Z') BEGIN  t <= GUARDED i0; END BLOCK;
  z <= t;
END guarded_assignments;
```

- Use REGISTER to model retaining of last value
- **No call is made to wiring upon last disconnection**

# Dataflow.resolution.application

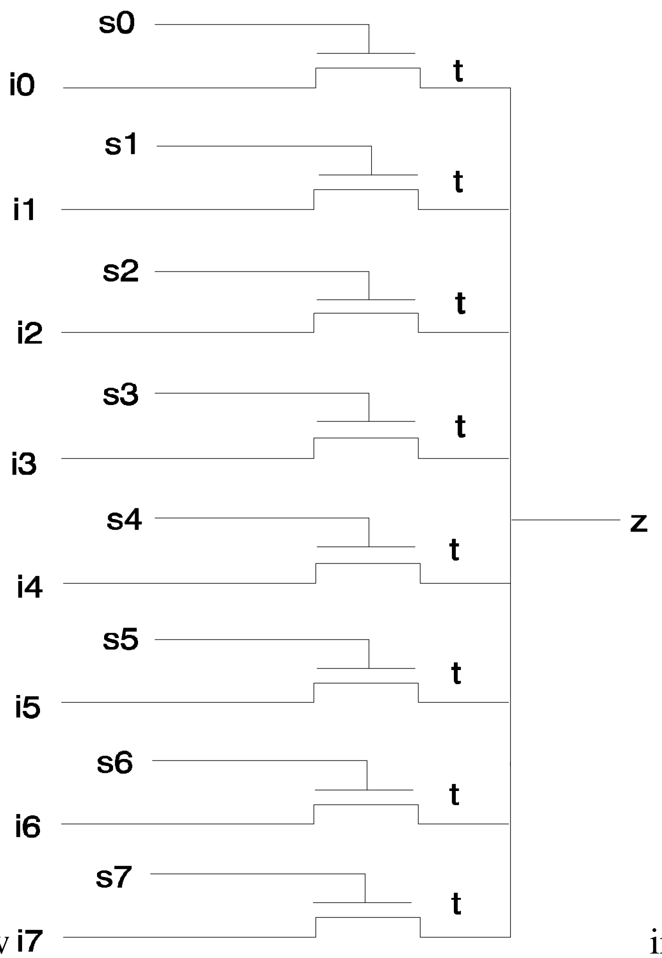
```
USE WORK.basic_utilities.ALL;
--  FROM PACKAGE USE: qit, wired_qit
ENTITY  multiplexed_half_register IS
  PORT (i7, i6, i5, i4, i3, i2, i1, i0 : IN qit;
             s7, s6, s5, s5, s4, s3, s2, s1, s0 : IN qit;
             z : OUT wired_qit REGISTER );
END multiplexed_half_register;


--
ARCHITECTURE guarded_assignments OF multiplexed_half_register IS
  BEGIN
  b7: BLOCK (s7 = '1' OR s7 = 'Z') BEGIN  z <= GUARDED i7; END BLOCK;
  b6: BLOCK (s6 = '1' OR s6 = 'Z') BEGIN  z <= GUARDED i6; END BLOCK;
  b5: BLOCK (s5 = '1' OR s5 = 'Z') BEGIN  z <= GUARDED i5; END BLOCK;
  b4: BLOCK (s4 = '1' OR s4 = 'Z') BEGIN  z <= GUARDED i4; END BLOCK;
  b3: BLOCK (s3 = '1' OR s3 = 'Z') BEGIN  z <= GUARDED i3; END BLOCK;
  b2: BLOCK (s2 = '1' OR s2 = 'Z') BEGIN  z <= GUARDED i2; END BLOCK;
  b1: BLOCK (s1 = '1' OR s1 = 'Z') BEGIN  z <= GUARDED i1; END BLOCK;
  b0: BLOCK (s0 = '1' OR s0 = 'Z') BEGIN  z <= GUARDED i0; END BLOCK;
END guarded_assignments;
```

- Ports can be resolved signals
- Kind of signal (BUS or REGISTER) can also be specified

# Dataflow.disconnection

```
b5: BLOCK (s5='1' OR s5='Z')
    BEGIN t <= GUARDED i5 AFTER 4 NS;
END BLOCK;
```




DISCONNECT t : w ired_qit
AFTER 6 NS;

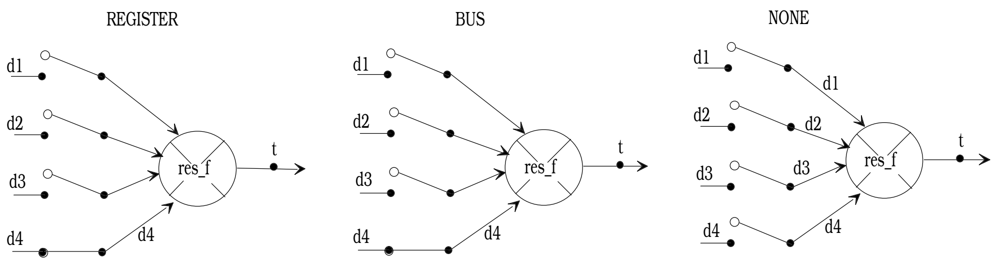
- Signal assignment can specify connection delay
- In guarded signal assignments:

- Use DISCONNECT for disconnection delay
- Above DISCONNECT applies to all assignments to *t*


© Zainalabedin Navabi, 1994

# Dataflow.disconnection

Signal Kind determines effective value




- REGISTER: Last disconnection; *t* retains value
- BUS: Last disconnection; *t* receives values of *res_f(null)*
- none: Upon disconnection, driver value still holds

# Dataflow.general_multiplexer

```
USE WORK.basic_utilities.ALL;
--  FROM PACKAGE USE : qit, qit_vector, wired_qit
ENTITY mux_n_to_1 IS
  PORT (i, s : IN qit_vector; z : OUT wired_qit BUS);
END mux_n_to_1;
--
ARCHITECTURE multiple_guarded_assignments OF mux_n_to_1 IS
BEGIN
  bi: FOR j IN i'RANGE GENERATE
    bj: BLOCK (s(j) = '1' OR s(j) = 'Z')
    BEGIN
      z <= GUARDED i(j);
    END BLOCK;
  END GENERATE;
END multiple_guarded_assignments;
```

- A useful unconstrained multiplexer
- Will know size when instantiated

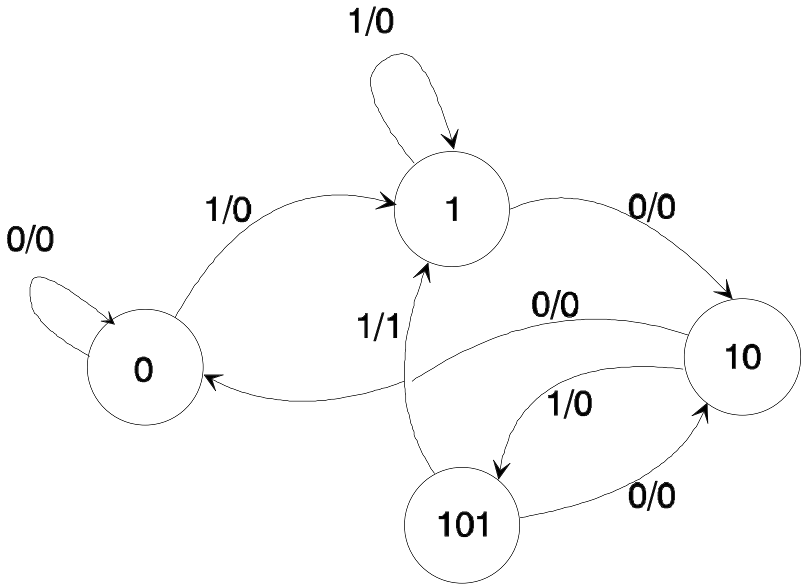
# Dataflow.general_multiplexer

```
USE WORK.basic_utilities.ALL;
ENTITY mux_tester IS  END mux_tester;
--
ARCHITECTURE input_output OF mux_tester IS
  COMPONENT mux PORT (i, s : IN qit_vector; z : OUT wired_qit BUS);
  END COMPONENT;
  FOR ALL : mux
     USE ENTITY WORK.mux_n_to_1 (multiple_guarded_assignments);
  SIGNAL ii, ss : qit_vector (3 DOWNTO 0) := "0000";    SIGNAL zz : qit;
BEGIN
  ii <= "1010" AFTER 10 US, "Z100" AFTER 20 US, "0011" AFTER 30 US;
  ss <= "0010" AFTER 05 US, "1100" AFTER 15 US, "000Z" AFTER 25 US;
  mm : mux PORT MAP (ii, ss, zz);
END input_output;
```

| **TIME (NS)** | **SIGNALS** II(3:0) | **SS(3:0)** | **ZZ** |
|---|---|---|---|
| 0 | "0000" | "0000" | '0' |
| +*d* | ...... | ...... | 'Z' |
| 5000 | ...... | "0010" | ... |
| +*d* | ...... | ...... | '0' |
| **10000** | **"1010"** | ...... | ... |
| +*d* | ...... | ...... | '1' |
| **15000** | ...... | **"1100"** | ... |
| +*d* | ...... | ...... | **'X'** |
| 20000 | "Z100" | ...... | ... |
| +*d* | ...... | ...... | '1' |
| 25000 | ...... | "000Z" | ... |
| +*d* | ...... | ...... | '0' |

- Simulation produces 'X' for two conflincting enabled inputs
- Produces 'Z' when no inputs are enabled

# Dataflow.state_machines




```
--States are named:    reset,  got1,  got10,  got101

clocking : BLOCK (clk = '1' AND NOT clk'STABLE)
BEGIN
  s1: BLOCK ( current = reset AND GUARD )      - - reset
  BEGIN
    current <= GUARDED got1 WHEN x = '1' ELSE reset;
  END BLOCK s1;
  ...
END BLOCK clocking;
```

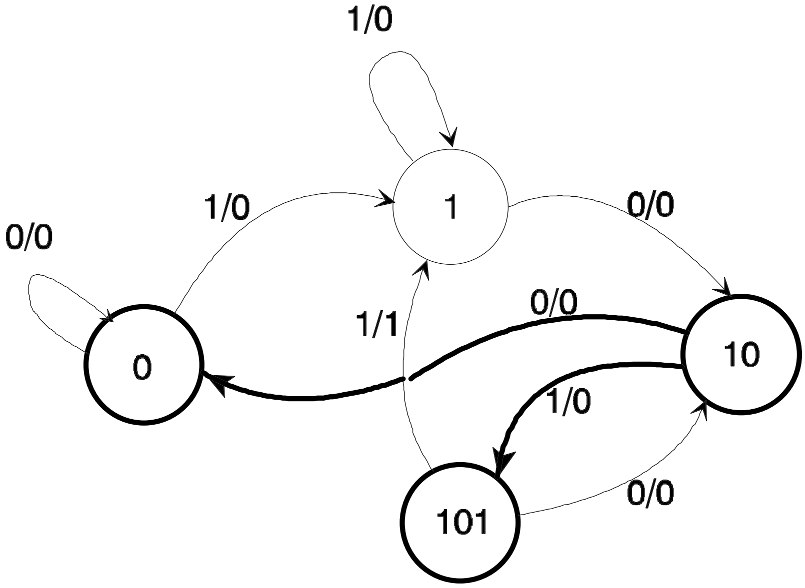
- Use resolutions & guarded blocks
- A simple 1011 Mealy detector
- A block statement for each state

# Dataflow.state_machines

```
ENTITY detector IS  PORT (x, clk : IN BIT; z : OUT BIT); END detector;
--
ARCHITECTURE singular_state_machine OF detector IS
  TYPE state IS (reset, got1, got10, got101);
  TYPE state_vector IS ARRAY (NATURAL RANGE <>) OF state;
  FUNCTION one_of (sources : state_vector) RETURN state IS
  BEGIN  RETURN sources(sources'LEFT);
  END one_of;
  SIGNAL current : one_of state REGISTER := reset;
BEGIN
  clocking : BLOCK (clk = '1' AND NOT clk'STABLE)
  BEGIN
    s1: BLOCK ( current = reset AND GUARD )
    BEGIN
      current <= GUARDED got1 WHEN x = '1' ELSE reset;
    END BLOCK s1;
    s2: BLOCK ( current = got1 AND GUARD )
    BEGIN
      current <= GUARDED got10 WHEN x = '0' ELSE got1;
    END BLOCK s2;
    s3: BLOCK ( current = got10 AND GUARD )
    BEGIN
      current <= GUARDED got101 WHEN x = '1' ELSE reset;
    END BLOCK s3;
    s4: BLOCK ( current = got101 AND GUARD)
    BEGIN
      current <= GUARDED got1 WHEN x = '1' ELSE got10;
      z <= '1' WHEN ( current = got101 AND x = '1') ELSE '0';
    END BLOCK s4;
  END BLOCK clocking;
END singular_state_machine;
```

- Current receives four concurrent assignments
- Current must be resolved; use *one_of*

# Dataflow.state_machines




```
--States are:   1 = reset,   2 = got1,   3 = got10,   4 = got101

clocking : BLOCK (clk = '1' AND NOT clk'STABLE)
BEGIN
  s3: BLOCK (s(3) = '1' AND GUARD)
  BEGIN
    s(1) <= GUARDED '1' WHEN x = '0' ELSE '0';
    s(4) <= GUARDED '1' WHEN x = '1' ELSE '0';
  END BLOCK s3;
END BLOCK clocking;
```

- The last description does not allows multiple active states
- To remedy: use a signal for each state
- State 3 : goes to 1 when x = '0'
  goes to 4 when x = '1'

# Dataflow.state_machines

```
ARCHITECTURE multiple_state_machine OF detector IS
  SIGNAL s : ored_bit_vector (1 TO 4) REGISTER := "1000";
BEGIN
  clocking : BLOCK (clk = '1' AND NOT clk'STABLE)
  BEGIN
    s1: BLOCK (s(1) = '1' AND GUARD)
    BEGIN
      s(1) <= GUARDED '1' WHEN x = '0' ELSE '0';
      s(2) <= GUARDED '1' WHEN x = '1' ELSE '0';
    END BLOCK s1;
    s2: BLOCK (s(2) = '1' AND GUARD)
    BEGIN
      s(3) <= GUARDED '1' WHEN x = '0' ELSE '0';
      s(2) <= GUARDED '1' WHEN x = '1' ELSE '0';
    END BLOCK s2;
    s3: BLOCK (s(3) = '1' AND GUARD)
    BEGIN
      s(1) <= GUARDED '1' WHEN x = '0' ELSE '0';
      s(4) <= GUARDED '1' WHEN x = '1' ELSE '0';
    END BLOCK s3;
    s4: BLOCK (s(4) = '1' AND GUARD)
    BEGIN
      s(3) <= GUARDED '1' WHEN x = '0' ELSE '0';
      s(2) <= GUARDED '1' WHEN x = '1' ELSE '0';
      z <= '1' WHEN (s(4) = '1' AND x = '1') ELSE '0';
    END BLOCK s4;
    s <= GUARDED "0000";
  END BLOCK clocking;
END multiple_state_machine;
```

- S must be resolved vector REGISTER kind
- S <= GUARDED "0000";

  Causes removal of retained value upon last disconnection

# Dataflow.general_state_machines

```
ENTITY detector_m IS  PORT (x,clk : IN BIT; z : OUT BIT);  END detector_m;
--
ARCHITECTURE multiple_moore_machine_1 OF detector_m IS
  FUNCTION oring( drivers : BIT_VECTOR) RETURN BIT IS
    VARIABLE accumulate : BIT := '0';
  BEGIN
    FOR i IN drivers'RANGE LOOP
      accumulate := accumulate OR drivers(i);
    END LOOP;
    RETURN accumulate;
  END oring;
  SUBTYPE ored_bit IS oring BIT;
  TYPE ored_bit_vector IS ARRAY (NATURAL RANGE <>) OF ored_bit;
  TYPE next_table IS ARRAY (1 TO 6, BIT) OF INTEGER;
  TYPE out_table IS ARRAY (1 TO 6, BIT) OF BIT;
  -- Fill in next_val,  out_val, and s arrays
  SIGNAL o : ored_bit REGISTER;
BEGIN
  clocking : BLOCK (clk = '1' AND (NOT clk'STABLE))
  BEGIN
    g: FOR i IN s'RANGE GENERATE
      si: BLOCK (s(i) = '1' AND GUARD)
      BEGIN
        s(next_val(i,'0')) <= GUARDED '1' WHEN x='0' ELSE '0';
        s(next_val(i,'1')) <= GUARDED '1' WHEN x='1' ELSE '0';
        o <= GUARDED out_val(i, x);
      END BLOCK si;
    s (i) <= GUARDED '0';
    END GENERATE;
  END BLOCK clocking;
  z <= o;
END multiple_moore_machine_1;
```

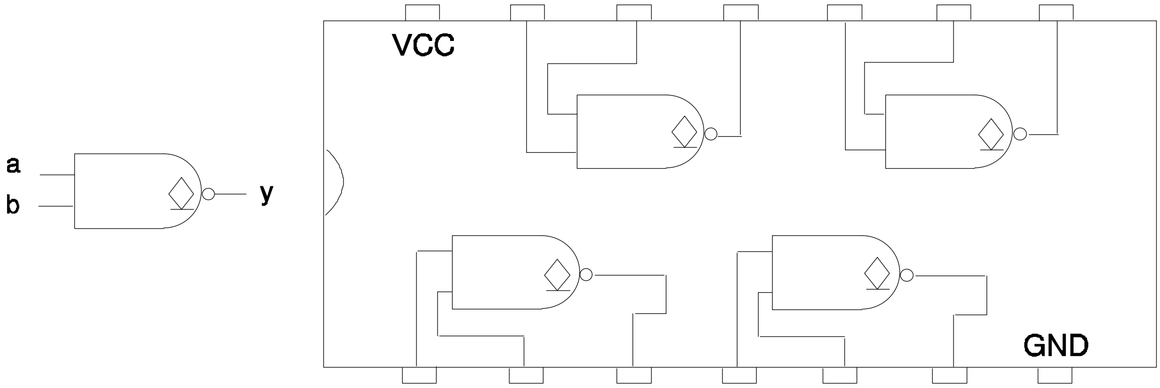
- A Moore  sequence detector
- Specify transisitions & outputing in constant tables
- Allows multiple machines in one

# Dataflow.general_state_machines

```
-------------------------------------------------------------------
--The folowing tables program the general purpose Moore description--
-------------------------------------------------------------------
--                           -- Next States:    -----   x=0, x=1  --
 CONSTANT next_val : next_table := ( (1 , 2),    --S1: -> S1,  S2  --
                                     (1 , 3),    --S2: -> S1,  S3  --
                                     (1 , 4),    --S3: -> S1,  S4  --
                                     (1 , 1),    --S4: -> S1,  S1  --
                                     (5 , 6),    --S5: -> S5,  S6  --
                                     (5 , 6) );  --S6: -> S5,  S6  --
--                                                                --
--                           -- Output Values:  -----   x=0, x=1  --
 CONSTANT out_val : out_table := ( ('0' , '0'),  --S1: == z=0,  0  --
                                   ('0' , '0'),  --S2: == z=0,  0  --
                                   ('0' , '0'),  --S3: == z=0,  0  --
                                   ('0' , '0'),  --S4: == z=1,  1  --
                                   ('0' , '0'),  --S5: == z=0,  0  --
                                   ('1' , '1') );--S6: == z=1,  1  --
                                                                  --
--                           -- Initial Active States:            --
 SIGNAL s : ored_bit_vector (1 TO 6) REGISTER := "100010";        --
-------------------------------------------------------------------
-------------------------------------------------------------------
```

- The *next_val* constant holds next state values
- The *out_val* constant holds the output values on the *z* output
- Initial starting states are set to '1' in the *s* vector

# Dataflow.open_collector




- Resolution functions are used in bussing
- Will use open collector to illustrate

# Dataflow.open_collector

```
USE WORK.basic_utilities.ALL;
--  FROM PACKAGE USE: qit, "AND"
ENTITY nand2 IS
  PORT (a, b : IN qit; y : OUT qit);
  CONSTANT tplh : TIME := 10 NS;  CONSTANT tphl : TIME := 12 NS;
END nand2;
--
ARCHITECTURE open_output OF nand2 IS
BEGIN
  y <= '0' AFTER tphl WHEN (a AND b) = '1' ELSE
       'Z' AFTER tplh WHEN (a AND b) = '0' ELSE
       'X' AFTER tphl;
END open_output;


USE WORK.basic_utilities.ALL;
--  FROM PACKAGE USE: qit
ENTITY sn7403 IS
  PORT (a1, a2, a3, a4, b1, b2, b3, b4 : IN qit; y1, y2, y3, y4 : OUT qit);
END sn7403;
--
ARCHITECTURE structural OF sn7403 IS
  COMPONENT nand2 PORT (a, b : IN qit; y : OUT qit); END COMPONENT;
  FOR ALL : nand2 USE ENTITY WORK.nand2 (open_output);
BEGIN
  g1: nand2 PORT MAP ( a1, b1, y1 );
  g2: nand2 PORT MAP ( a2, b2, y2 );
  g3: nand2 PORT MAP ( a3, b3, y3 );
  g4: nand2 PORT MAP ( a4, b4, y4 );
END structural;
```

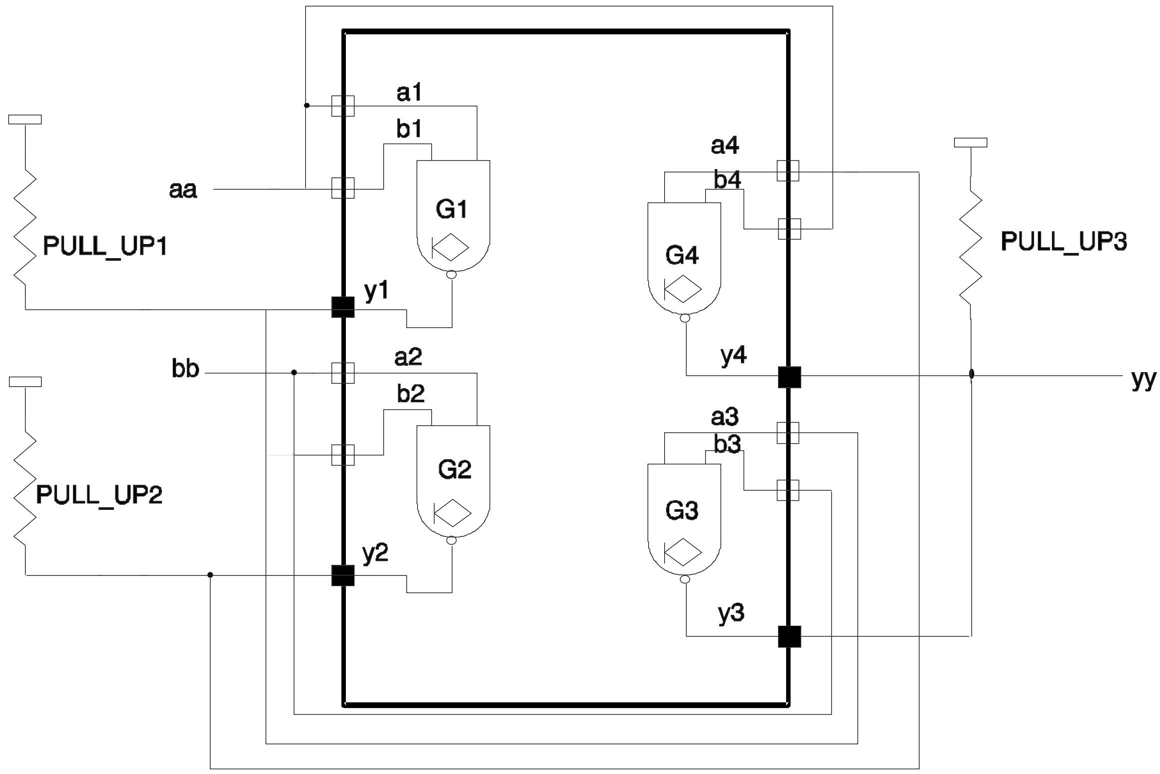
- A two input open collector NAND
- 74LS03 contains 4 open collector NAND gates

# Dataflow.open_collector

| TIME (NS) | SIGNAL AA | BB | YY |
|---|---|---|---|
| 0 | '0' | '0' | '0' |
| 10 | ... | ... | 'Z' |
| 1000 | '1' | ... | ... |
| 1500 | ... | '1' | ... |
| 1512 | ... | ... | '0' |
| 2500 | '0' | ... | ... |
| 2510 | ... | ... | 'Z' |
| 3000 | 'Z' | ... | ... |
| 3012 | ... | ... | '0' |
| 3750 | ... | '0' | ... |
| 3760 | ... | ... | 'Z' |
| 4000 | '0' | ... | ... |
| 4500 | '1' | 'Z' | ... |
| 4512 | ... | ... | '0' |
| 5000 | '0' | ... | ... |
| 5010 | ... | ... | 'Z' |
| 5500 | 'Z' | ... | ... |
| 5512 | ... | ... | '0' |
| 6000 | ... | '0' | ... |
| 6010 | ... | ... | 'Z' |
| 6750 | ... | '1' | ... |
| 6762 | ... | ... | '0' |
| 7500 | ... | '0' | ... |

- Simulation verifies open collector
- Output is never '1'

# Dataflow.open_collector




- Using 74LS03 for implementing an XNOR
- *pull_up3* has two drivers

# Dataflow.open_collector

```
USE WORK.basic_utilities.ALL;
--  FROM PACKAGE USE: qit, anded_qit
ENTITY test_xnor IS END test_xnor;
--
USE WORK.basic_utilities.ALL;
ARCHITECTURE input_output OF test_xnor IS
  COMPONENT sn7403
    PORT (a1, a2, a3, a4, b1, b2, b3, b4 : IN qit; y1, y2, y3, y4 : OUT qit);
  END COMPONENT;
  FOR ALL : sn7403 USE ENTITY WORK.sn7403 (structural);
  SIGNAL aa, bb : qit;
  SIGNAL pull_up_1, pull_up_2, pull_up_3 : anded_qit := 'Z';
BEGIN
  aa <= '1', '0' AFTER 10US, '1' AFTER 30US, '0' AFTER 50US,
           'Z' AFTER 60US;
  bb <= '0', '1' AFTER 20US, '0' AFTER 40US, 'Z' AFTER 70US;
  c1: sn7403 PORT MAP (
                 aa,            bb,            pull_up_1,    pull_up_2,
                 aa,            bb,            bb,           aa,
                 pull_up_1,     pull_up_2 ,    pull_up_3,    pull_up_3 );
END input_output;
```

$$yy = ( aa' . bb)' . (bb' . aa)' = ( aa \oplus bb)'$$

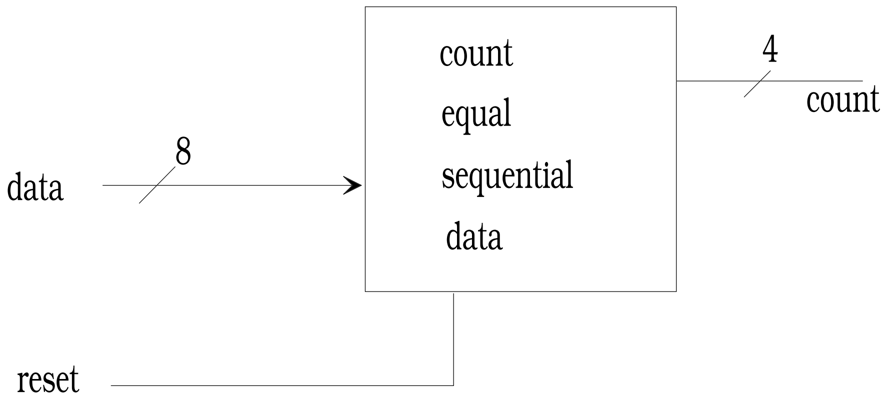
- *pull_up_1* and *pull_up_2* turn 0,Z to 0,1
- *anded_qit* resolution function implements wired logic

# Dataflow.open_collector

| TIME (NS) | SIGNALS AA | BB | PULL_UP_1 | PULL_UP_2 | PULL_UP_3 |
|---|---|---|---|---|---|
| 0 | '1' | '0' | '0' | '0' | '0' |
| 2 | '1' | '0' | '0' | '1' | '0' |
| 4 | '1' | '0' | '0' | '1' | '0' |
| 6 | '1' | '0' | '0' | '1' | '0' |
| 8 | '1' | '0' | '0' | '1' | '0' |
| 10 | '0' | '0' | '0' | '1' | '0' |
| 12 | '0' | '0' | '1' | '1' | '1' |
| 14 | '0' | '0' | '1' | '1' | '1' |
| 16 | '0' | '0' | '1' | '1' | '1' |
| 18 | '0' | '0' | '1' | '1' | '1' |
| 20 | '0' | '1' | '1' | '1' | '1' |
| 22 | '0' | '1' | '1' | '0' | '0' |
| 24 | '0' | '1' | '1' | '0' | '0' |
| 26 | '0' | '1' | '1' | '0' | '0' |
| 28 | '0' | '1' | '1' | '0' | '0' |
| 30 | '1' | '1' | '1' | '0' | '0' |
| 32 | '1' | '1' | '0' | '0' | '1' |
| 34 | '1' | '1' | '0' | '0' | '1' |
| 36 | '1' | '1' | '0' | '0' | '1' |
| 38 | '1' | '1' | '0' | '0' | '1' |
| 40 | '1' | '0' | '0' | '0' | '1' |
| 42 | '1' | '0' | '0' | '1' | '0' |
| 44 | '1' | '0' | '0' | '1' | '0' |
| 46 | '1' | '0' | '0' | '1' | '0' |
| 48 | '1' | '0' | '0' | '1' | '0' |
| 50 | '0' | '0' | '0' | '1' | '0' |
| 52 | '0' | '0' | '1' | '1' | '1' |
| 54 | '0' | '0' | '1' | '1' | '1' |
| 56 | '0' | '0' | '1' | '1' | '1' |
| 58 | '0' | '0' | '1' | '1' | '1' |
| 60 | 'Z' | '0' | '1' | '1' | '1' |
| 62 | 'Z' | '0' | '0' | '1' | '0' |
| 64 | 'Z' | '0' | '0' | '1' | '0' |
| 66 | 'Z' | '0' | '0' | '1' | '0' |

- Simulation shows XNOR implementation
- Pull up resolutions turn gate output 'Z' values to '1'

# Dataflow.dataflow_example




- Seen dataflow primitives
- Use dataflow for system description
- A sequential comparator example

# Dataflow.dataflow_example

```
USE WORK.basic_utilities.ALL;
--  FROM PACKAGE USE: bin2int, int2bin
ENTITY sequential_comparator IS
  PORT (data : IN BIT_VECTOR (7 DOWNTO 0); clk, reset : IN BIT;
        matches : OUT BIT_VECTOR (3 DOWNTO 0));
END sequential_comparator;
--
ARCHITECTURE dataflow OF sequential_comparator IS
  FUNCTION inc (x : BIT_VECTOR) RETURN BIT_VECTOR IS
    VARIABLE i : INTEGER;
    VARIABLE t : BIT_VECTOR (x'RANGE);
  BEGIN
    bin2int (x, i);
    i := i + 1; IF i >= 2**x'LENGTH THEN i := 0; END IF;
    int2bin (i, t);
    RETURN t;
  END inc;
  SIGNAL buff : BIT_VECTOR (7 DOWNTO 0);
  SIGNAL count : BIT_VECTOR (3 DOWNTO 0);
BEGIN
  edge: BLOCK (clk = '0' AND NOT clk'STABLE)
  BEGIN
    buff <= GUARDED data;
    count <= GUARDED "0000" WHEN reset = '1' ELSE
                 inc (count) WHEN data = buff ELSE
                 count;
  END BLOCK;
  matches <= count;
END dataflow;
```

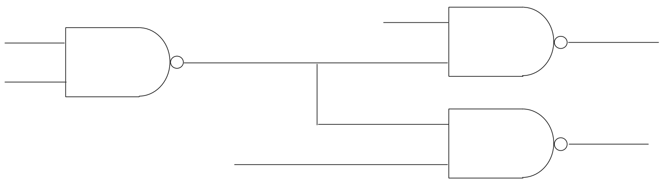
- *inc* function is unconstrained
- Save old data in *buff*
- Compares old and new

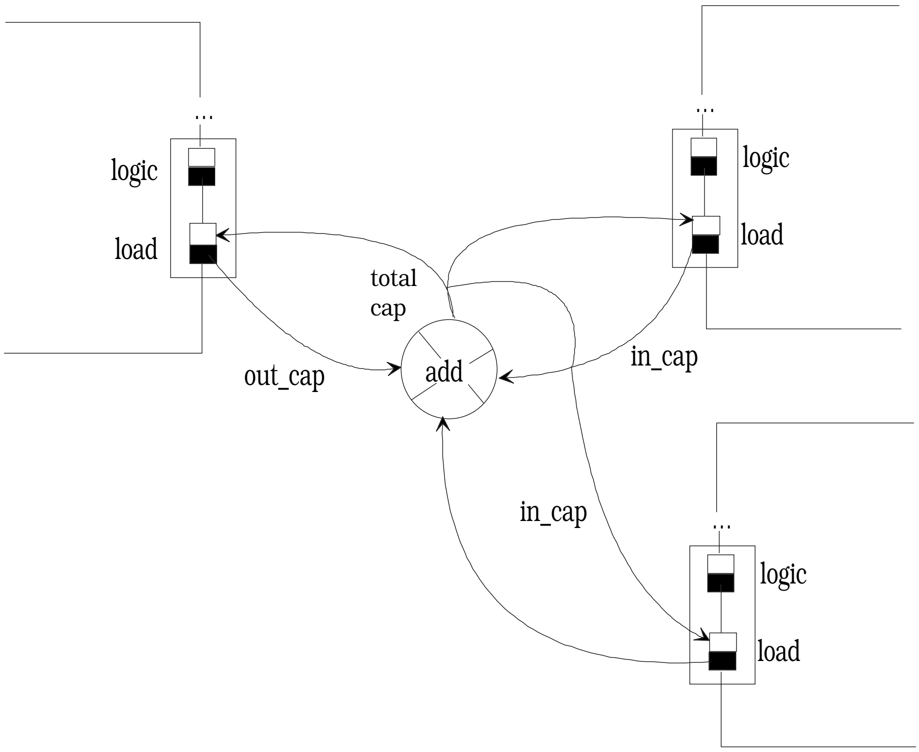
# Dataflow.dataflow_example

| TIMES (NS) | SIGNALS RESET | CLK | DATA(7:0) | UFF(7:0) | COUNT(3:0) | MATCHES |
|---|---|---|---|---|---|---|
| 0 | '0' | '0' | "00000000" | "00000000" | "0000" | "0000" |
| +d | ... | ... | .......... | .......... | ...... | "0000" |
| 200 | ... | ... | "11110101" | .......... | ...... | ...... |
| 500 | ... | '1' | .......... | .......... | ...... | ...... |
| 1000 | ... | '0' | .......... | .......... | ...... | ...... |
| +d | ... | ... | .......... | "11110101" | "0000" | ...... |
| 1200 | ... | ... | "01010110" | .......... | ...... | ...... |
| 1500 | ... | '1' | .......... | .......... | ...... | ...... |
| 1700 | ... | ... | "11111110" | .......... | ...... | ...... |
| 2000 | ... | '0' | .......... | .......... | ...... | ...... |
| +d | ... | ... | .......... | "11111110" | "0000" | ...... |
| 2500 | ... | '1' | .......... | .......... | ...... | ...... |
| 3000 | ... | '0' | .......... | .......... | ...... | ...... |
| +d | ... | ... | .......... | "11111110" | "0001" | ...... |
| +d | ... | ... | .......... | .......... | ...... | "0001" |
| 3200 | ... | ... | "01010100" | .......... | ...... | ...... |
| 3500 | ... | '1' | .......... | .......... | ...... | ...... |
| 3700 | ... | ... | "00010001" | .......... | ...... | ...... |
| 4000 | ... | '0' | .......... | .......... | ...... | ...... |
| +d | ... | ... | .......... | "00010001" | "0001" | ...... |
| 4200 | ... | ... | "10010110" | .......... | ...... | ...... |
| 4500 | ... | '1' | .......... | .......... | ...... | ...... |
| 5000 | ... | '0' | .......... | .......... | ...... | ...... |
| +d | ... | ... | .......... | "10010110" | "0001" | ...... |
| 5500 | ... | '1' | .......... | .......... | ...... | ...... |
| 6000 | ... | '0' | .......... | .......... | ...... | ...... |
| +d | ... | ... | .......... | "10010110" | "0010" | ...... |
| +d | ... | ... | .......... | .......... | ...... | "0010" |
| 6500 | ... | '1' | .......... | .......... | ...... | ...... |
| 7000 | ... | '0' | .......... | .......... | ...... | ...... |
| +d | ... | ... | .......... | "10010110" | "0011" | ...... |
| +d | ... | ... | .......... | .......... | ...... | "0011" |
| 7500 | ... | '1' | .......... | .......... | ...... | ...... |
| 8000 | ... | '0' | .......... | .......... | ...... | ...... |
| +d | ... | ... | .......... | "10010110" | "0100" | ...... |
| +d | ... | ... | .......... | .......... | ...... | "0100" |
| 8500 | ... | '1' | .......... | .......... | ...... | ...... |

- *matches* shows count of matching data

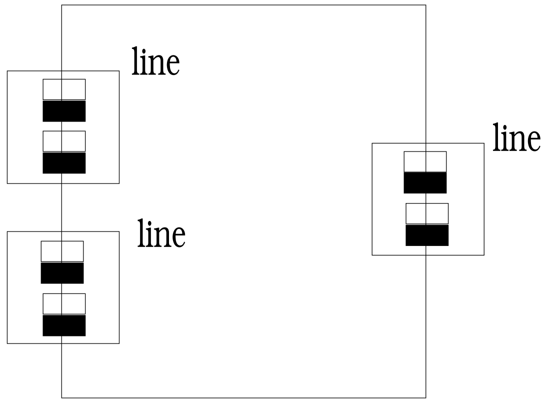
# Dataflow.example.load_dependency






- Model gates for load dependent timing
- Timing depends on capacitance
- A resolution function adds all capacitances
- Internal Timing depends on total capacitance at output

# **Dataflow.example.load_dependency**




```
TYPE line IS RECORD
  logic : wired_qit;
  load  : loading capacitance;
END RECORD;
```

- Each port has:
  1) A Logical value
  2) A Load value
- Logical resolution is *wired*
- Load resolution is *adding*

# Dataflow.example.load_dependency

```
LIBRARY cmos;
USE cmos.basic_utilities.ALL;
PACKAGE added_utilities IS
  FUNCTION loading (drivers : capacitance_vector) RETURN capacitance;
  SUBTYPE loading_capacitance IS loading capacitance;
  TYPE line IS RECORD
    logic : wired_qit;
    load  : loading capacitance;
  END RECORD;
END added_utilities;


PACKAGE BODY added_utilities IS
  FUNCTION loading (drivers : capacitance_vector) RETURN capacitance IS
    VARIABLE load_value : capacitance := 0 ffr;
  BEGIN
    FOR i IN drivers'RANGE LOOP
      load_value := load_value + drivers(i);
    END LOOP;
    RETURN load_value;
  END loading;
END added_utilities;
```

- Use most of *basic_utilities*
- Add new types and resolution functions in the *added_utilities*

# Dataflow.example.load_dependency

```
USE WORK.added_utilities.ALL;
--
ENTITY nand_2 IS
  PORT(in1,in2 : INOUT line; out1 : INOUT line);
END nand_2;
--
ARCHITECTURE load_dependent OF nand_2 IS
  CONSTANT  pullup : resistance := 15 k_o;
  CONSTANT pulldown: resistance := 10 k_o;
BEGIN
  in1.load  <= 10 ffr;   in2.load  <= 10 ffr;   out1.load  <= 2 ffr;
  in1.logic <= 'Z' ;     in2.logic <= 'Z' ;
  outing : PROCESS(in1.logic , in2.logic)
    VARIABLE temp_out : qit;
  BEGIN
    temp_out := in1.logic NAND in2.logic;
    IF temp_out = '0' THEN
      out1.logic <= '0' AFTER 3 * pulldown * out1.load;
    ELSE
      IF temp_out = '1' THEN
        out1.logic <= '1' AFTER 3 * pullup  * out1.load;
      ELSE
        out1.logic <= temp_out;
      END IF;
    END IF;
  END PROCESS outing;
END load_dependent;
```

- *out1.load* is returned by the resolution function
- NAND gate delay depends on total capacitance

# Dataflow.example.load_dependency

```
USE WORK.added_utilities.ALL;
ENTITY  three_nand_system IS  END three_nand_system;
--
ARCHITECTURE test OF three_nand_system IS
  COMPONENT nand_2
    PORT(in1,in2 : INOUT line ; out1 : INOUT line);
  END COMPONENT;
  SIGNAL a1,b1,b2,b3,q1_out,q2_out,q3_out : line;
BEGIN
  b1.logic <= '1';    b2.logic <= '1';    b3.logic <= '1';
  a1.logic <= '0',  '1' AFTER 100 NS,  '0' AFTER 300 NS,
                'Z' AFTER 700 NS,  'X' AFTER 800 NS,  '1' AFTER 1000 NS;
  n1 : nand_2 PORT MAP(a1,b1,q1_out);
  n2 : nand_2 PORT MAP(q1_out,b2,q2_out);
  n3 : nand_2 PORT MAP(q1_out,b3,q3_out);
END test;
```

| fs | a1 | b1 | b2 | b3 | q1_out | q2_out | q3_out |
|---|---|---|---|---|---|---|---|
| 0 | (0, 00) | (0, 00) | (0, 00) | (0, 00) | (0, 00) | (0, 0) | (0, 0) |
| 0 | (0, 10) | (1, 10) | (1, 10) | (1, 10) | (1, 22) | (1, 2) | (1, 2) |
| 60000 | (0, 10) | (1, 10) | (1, 10) | (1, 10) | (1, 22) | (0, 2) | (0, 2) |
| 100000000 | (1, 10) | (1, 10) | (1, 10) | (1, 10) | (1, 22) | (0, 2) | (0, 2) |
| 100660000 | (1, 10) | (1, 10) | (1, 10) | (1, 10) | (0, 22) | (0, 2) | (0, 2) |
| 100960000 | (1, 10) | (1, 10) | (1, 10) | (1, 10) | (0, 22) | (1, 2) | (1, 2) |
| 300000000 | (0, 10) | (1, 10) | (1, 10) | (1, 10) | (0, 22) | (1, 2) | (1, 2) |
| 300000105 | (0, 10) | (1, 10) | (1, 10) | (1, 10) | (1, 22) | (1, 2) | (1, 2) |
| 300065033 | (0, 10) | (1, 10) | (1, 10) | (1, 10) | (1, 22) | (0, 2) | (0, 2) |
| 700000000 | (Z, 10) | (1, 10) | (1, 10) | (1, 10) | (1, 22) | (0, 2) | (0, 2) |
| 700660000 | (Z, 10) | (1, 10) | (1, 10) | (1, 10) | (0, 22) | (0, 2) | (0, 2) |
| 700960000 | (Z, 10) | (1, 10) | (1, 10) | (1, 10) | (0, 22) | (1, 2) | (1, 2) |
| 800000000 | (X, 10) | (1, 10) | (1, 10) | (1, 10) | (0, 22) | (1, 2) | (1, 2) |
| 800000000+ | (X, 10) | (1, 10) | (1, 10) | (1, 10) | (X, 22) | (1, 2) | (1, 2) |
| 800000000+ | (X, 10) | (1, 10) | (1, 10) | (1, 10) | (X, 22) | (X, 2) | (X, 2) |
| 1000000000 | (1, 10) | (1, 10) | (1, 10) | (1, 10) | (X, 22) | (X, 2) | (X, 2) |
| 1000660000 | (1, 10) | (1, 10) | (1, 10) | (1, 10) | (0, 22) | (X, 2) | (X, 2) |
| 1000960000 | (1, 10) | (1, 10) | (1, 10) | (1, 10) | (0, 22) | (1, 2) | (1, 2) |

- *q1_out* is the common node between 3 NAND gates
- Logic values and capacitances at each node are shown
- Timing considers node capacitances

# **Dataflow.conclusions**

**1. Outline:** Introduction, Organization, Outline

**2. Review:** Behavioral description, Using process statements, Top-down design, Using available components, Wiring predefined components, Wiring from bottom to top, Generation of testbench data, Using procedures

**3. VHDL_Timing:** Modeling requirements, Objects & classes, Signals & variables, Concurrent & sequential assignments, Events, transactions & delta delays, Delay modeling, Sequential placement of transactions, Conventions

**4. Structural_Description_of_Hardware:** Wiring parts into larger designs, Start with primitives, Wire gates into general purpose components, Use iterative constructs, Generate testbenches, Show binding alternatives, Use gate-based components for a larger design

**5. Design_Organization:** Subprograms, Packaging, Parameter specification, Parametrization, Top level configuration, Design libraries, A complete example

**6. Utilities_for_high_level_descriptions:** Language aspects, Types, Over loading subprograms operators, Predefined Attributes, User defined attributes

**7. Dataflow: Constructs for dataflow descriptions, Multiplexing and clocking, Multiple assignments, State machines, Open collector gates, A complete dataflow example, Load dependent timing**

**8. Behavioral_Descriptions:** Constructs for sequential descriptions, Assertion for behavioral checks, Handshaking constructs, Timing control, Formatted I/O, MSI parts, A complete MSI based design

**9. STANDARDS:** MVL9: logic value system, Logic type, Operators, Conversions; VHDL'93: Operators, Delay model, Instantiation, Binding, Attributes, Signal assignments, Report, Postponed process

© Zainalabedin Navabi, 1994