
COMPLETION DETECTION AND ACKNOWLEDGE CIRCUITRY

This chapter provides information about **ack_logic** types and how to use them to connect registers. Topics are given in the following sections:

To Locate Information About....	Go to Page
Ack_logic	36
Coding Acknowledge Signals	41
Register Usage Notes	47

Ack_logic

Types

The **ncl_logic** type is a 3-valued logic {'0', '1', 'N'} used in the data and control paths. However, acknowledge and reset signals need only two data values and a single wire in the netlist. For these signals use the **ack_logic** type declared in the package **ack_logic**. The base type is **ack_uologic**.

```
TYPE ack_uologic IS ( 'U', -- Uninitialized
                    'X', -- Forcing Unknown
                    '1', -- Forcing 1
                    '0', -- NULL
                    'Z', -- High Impedance
                    '-' -- Don't care
                    );
```

As with **std_logic**, there are only two data values {'0', '1'}. The netlist generated by synthesis will have only one wire per **ack_logic** signal. The other types based on **ack_uologic** are as follows:

- **ack_uologic_vector**
- **ack_logic**
- **ack_logic_vector**

The two resolved subtypes for coding are **ack_logic** and **ack_logic_vector**. They are defined in the same way the two **std_logic** types are defined from **std_uologic**.

- `subtype ack_logic is resolved ack_uologic;`
- `type ack_logic_vector is array (natural range <>) of ack_logic;`

As with **std_logic**, there are some resolved subtypes based on **ack_uologic** (**X01N**, **X01NZ**, **UX01N**, **UX01NZ**). Here are some example declarations:

```
signal a : ack_logic;
signal b : ack_logic_vector( 7 downto 0);
```

Library and Use Statements

Include the following lines before each entity that uses **ack_logic**:

```
library ncl;
use ncl.ack_logic.all;
```

Currently, only positive logic functions are supported (AND, OR, and **std_and**). The AND implements a 2-of-2 threshold gate, while **std_and** implements a Boolean AND.

Connecting Registers using Acknowledge Signals

Registers use acknowledge signals for handshaking. The acknowledge signals are single bit, so **ack_logic** is used in the source code. In this section, we show how DATA and NULL waveforms pass through a single register. Then, we demonstrate the process to combine registers in three simple combinations: a pipeline, a fork, and a join.

As shown in [Figure 12](#), each 2NCL register has a data input, ($a0$, $a1$), a data output, ($z0$, $z1$), and an acknowledge input, ki , and an acknowledge output, ko . The acknowledge input is like the clock input of a flip-flop; it enables data to be loaded into the register. The acknowledge output detects the state of the output and requests the next state.

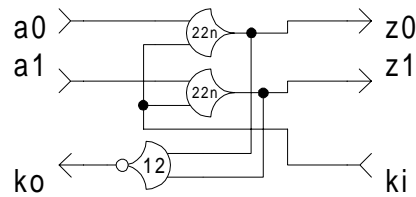


Figure 12 Single-bit 2NCL Register (reset not shown)

How the register goes from an initial NULL state to a DATA state to a NULL state is shown in [Figure 13](#).

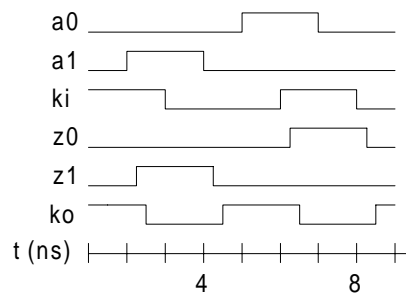


Figure 13 Register Behavior from NULL to DATA to NULL

The protocol (behavior) that the registers use to communicate is as follows:

0 ns – Assume that during reset, all data values are '0' and all acknowledge signals are '1'.

2 ns – After reset is released, data begins to flow, and a '1' arrives on $a1$. Because ki is '1', the data is loaded, so $z1$ asserts '1', which causes ko to assert '0' (request NULL). That is, the register has received DATA and can now load NULL.

4 ns – To complete the cycle, NULL arrives on a ($(a0, a1) = ('0', '0')$), and NULL is loaded into z . This causes ko to assert '1' (request DATA). That is, the register has received a NULL and can now load DATA.

5 - 9 ns – A second DATA/NULL cycle is shown from 5 to 9 ns. This time a switches before ki .

How registers can be connected is explained next. [Figure 14](#) illustrates the simplest type of configuration, a pipeline, which functions as follows:

- On the left and right are two registers, and in the middle is combinational logic. The data signals use `ncl_logic` signals.
- At the bottom, the acknowledge circuit is shown.

Acknowledge circuits use **ack_logic** signals.

- When the register on the right has DATA, it requests NULL and waits until NULL is presented to its inputs.

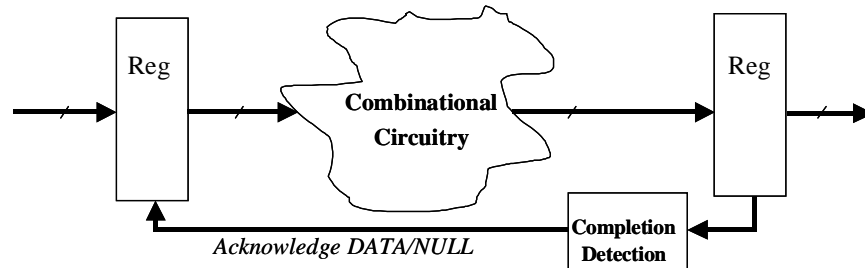


Figure 14 Pipeline Configuration

The circuit below shows how this architecture can be coded. Two-bit registers are used, and the combinational logic is an AND and OR gate.

Example: Coding for a Pipeline Configuration

```

library ncl;
use ncl.ncl_logic.all;
use ncl.ack_logic.all;
use ncl.ncl_components.all;

entity regv is port (
    datain    : in  ncl_logic_vector(1 downto 0);
    ackin     : in  ack_logic;
    rst       : in  ack_logic;
    ackout    : out ack_logic;
    dataout   : out ncl_logic_vector(1 downto 0)
);
end regv;

architecture arch of regv is
    signal kright : ack_logic;
    signal com_in, com_out : ncl_logic_vector(1 downto 0);
begin

    gleft: ncl_register_ss
        generic map (stages => 1, initial_value => -1, width => 2)
        port map (datain, kright, rst, com_in, ackout);

    combinational_logic:
        process (com_in)
        begin
            com_out(0) <= com_in(0) OR com_in(1);
            com_out(1) <= com_in(0) AND com_in(1);
            hysteresis(com_in, com_out);
        end process;

    gright: ncl_register_ss
        generic map (stages => 1, initial_value => -1, width => 2)

```

```

    port map (com_out, ackin, rst, dataout, kright);

    end arch;

```

When a register is not in a pipeline, two related rules should be followed:

Forks—A signal must be acknowledged by all latching registers.
 Join—A register must acknowledge all its input signals.

The fork rule is illustrated in [Figure 15](#) and in the code example below and functions as follows:

- The output of register *a_reg* is sent to two registers, *y_reg* and *z_reg*.
- With a fork, two or more acknowledge signals must be combined into one using *n* of *n* threshold gates.

Here, the *ko* of *y_reg*, *ky*, and *ko* of *z_reg*, *kz*, are connected with a th22 gate.

- With a th22 gate, the register on the left, *a_reg*, will wait until both registers on the right have latched the data. If an AND gate was used, *a_reg* would proceed when only one register was ready. If *y_reg* was unconnected and *z_reg* was connected to *a_reg*, then *a_reg* would proceed regardless of the status of *y_reg*.

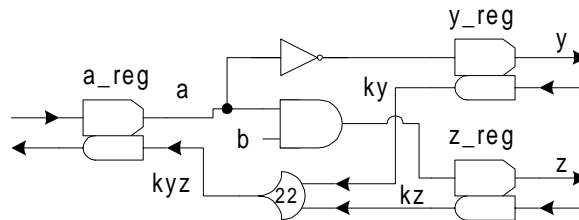


Figure 15 Circuit with a Fork

Example: Coding for the Fork Rule

```

kyz <= ky and kz ;
a_reg: ncl_register
  port map (datain => data_a, ki => kyz, rst => rst,
           dataout => a, ko => ka );

```

The join rule is the fork rule restated from the perspective of the register receiving data. The join rule is illustrated in [Figure 16](#) and in the code example below and functions as follows:

- The outputs of *a_reg* and *b_reg* are sent to register *y_reg*.
- With a join, the acknowledge signal from *y_reg* must be sent to both registers.

This is done by connecting the output acknowledge from *y_reg*, *ky*, to the acknowledge inputs of *a_reg* and *b_reg*. If the acknowledge input of a register is not connected, the circuit freezes, so this error is easy to detect.

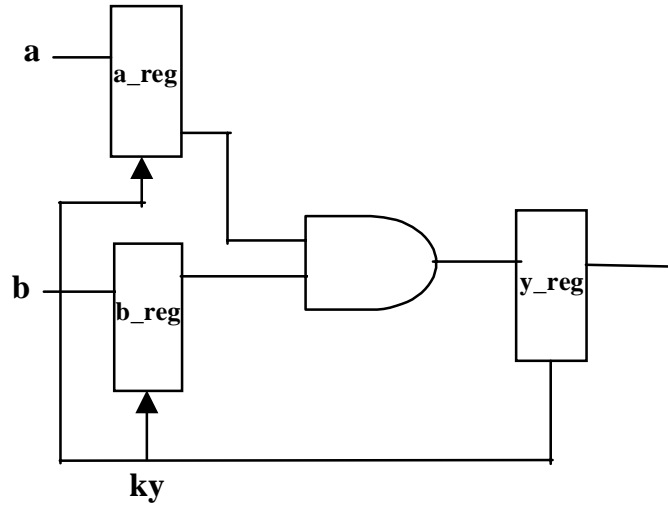


Figure 16 Join Rule

Example: Coding for the Join Rule

```
a_reg: ncl_register
  port map (datain => a, ki => ky, rst => rst,
            dataout => ao, ko => ka );
b_reg: ncl_register
  port map (datain => b, ki => ky, rst => rst,
            dataout => bo, ko => kb );
```

Coding Acknowledge Signals

How acknowledge circuits should be used to synchronize registers has been discussed in the previous sections. Coding these circuits is explained in this section.

The three ways you can code the circuits are as follows:

- using components
- using process statements
- using concurrent signal assignment statements

Using Components

Several components already contain acknowledge circuitry. The most important is the completion component that implements a completion detection tree with n -of- n threshold gates. The following is the definition for a **completion** component:

Definition: Complete Statement

```

component completion
  generic ( width : integer := 2 ) ;
  port ( a : in ack_logic_vector( 0 to width-1 ) ;
        z : out ack_logic ) ;
end component ;

```

This is an example instantiation of a completion component:

Example: Implementing a Completion Detection Tree

```

reg_comp : completion
  generic map ( width => 9 )
  port map    ( a => ko, z => kout ) ;

```

In this example, the signal *ko* is a 9-bit input of type **ack_logic_vector**. The signal *kout* is an output of type **ack_logic**. The generic map is used to specify the width of the vector. A 9-bit completion tree is generated by this statement. [Figure 17](#) shows how this circuit might be synthesized.

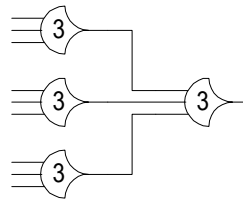


Figure 17 Nine-bit Completion Tree Example

The **cmpd** component used for inferred registers also has built-in completion. The **ncl_register_ss** contains built-in completion as well. [Figure 18](#) illustrates a three-bit, two-to-one multiplexer with a register; the code used to create this logic follows the figure. By using this component, the acknowledge tree for the register is built in.

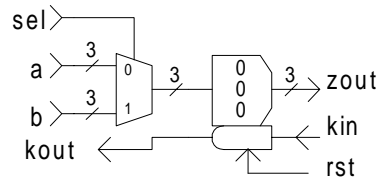


Figure 18 Three-Bit, Two-to-One Multiplexer with Register

Example: Three-Bit, Two-to-One Multiplexer with a Register

```

entity mux_2to1_3b is
  port ( a, b : input ncl_logic_vector(2 downto 0);
        sel : input ncl_logic;
        kin, rst : input ack_logic;
        z : output ncl_logic_vector(2 downto 0);
        kout : output ack_logic
  );
end mux_2to1_3b;

architecture ncl of mux_2to1_3b is
  signal zout : ncl_logic_vector(2 downto 0);
  signal sel : ncl_logic;
begin
  comb : process (sel, a, b)
  begin
    if sel = '0' then
      zout <= ain;
    else
      zout <= bin;
    end if;
    hysteresis(sel & a & b, zout);
  end process comb;

  ireg: ncl_register_ss
    generic map (width => 3; initial_value => 0; stages => 1 );
    port map ( datain => zout, ki => kin, rst => rst,
              dataout => z, ko => kout );
end ncl;

```

The registers and completion units described are sufficient if you are using full completion acknowledge circuits (that is, all signals are acknowledged) and combinational logic that is not combined with the control or acknowledge circuits. However, you may need to instantiate logic explicitly for your acknowledge path. In particular, an inversion may be required. Currently, all circuits generated by the NCL flow remove all inversions, creating monotonic circuits. Thus, the only way to force an inversion is to instantiate it. The gates you can instantiate are listed below.

Components Available		
ack_buf	ack_inv	ack_t22
ack_th12	ack_th13	ack_th14
ack_th12b	ack_th13b	ack_th14b
ack_th22	ack_th33	ack_th44
ack_th22d	ack_th22n	
ack_th33d	ack_th33n	
ack_th44d	ack_th44n	
ack_mutex		

These gates are declared in the **ncl_components** package and can be simulated. During the second pass, they are mapped into the appropriate gate.

You should not connect data signals to these components. The components are intended only for acknowledge signals. If you use a data signal, problems will occur.

A function **std_and** that implements the t22 gate is also available. This is an example of its use:

```
signal ka, kb, kz : ack_logic;
kz <= std_and(ka, kb);
```

Using Process Statements

Acknowledge circuits use hysteresis differently than **ncl_logic** circuits. With **ncl_logic**, a signal always has a NULL wavefront separating successive DATA wavefronts. To implement this, all functions need hysteresis. Circuits using **ack_logic** are not required to follow this protocol. The simplest rule to handle hysteresis in acknowledge circuits is

Use hysteresis for threshold gates with hysteresis

Thus, OR and **std_and** functions do not need hysteresis statements, while the AND function requires hysteresis.

When acknowledge circuits are placed inside processes, a hysteresis procedure provides the hysteresis functionality. The hysteresis declarations are the same as the **ncl_logic** procedures:

```
procedure hysteresis( ain : in    ack_logic_vector;
                    signal zout : inout ack_logic);
procedure hysteresis( ain : in    ack_logic_vector;
                    signal zout : inout ack_logic_vector);
```

Each output variable requires a separate hysteresis statement. The input to the statement is the concatenation of all the required input variables. Let's examine several cases to see which inputs need hysteresis. To simplify the use of hysteresis, we will limit each statement to one type of function. That is, we won't mix ANDs with ORs and **std_and**s.

The th22 gate needs hysteresis.

```
process (ka, kb, kc)
```

```
begin
  kz <= ka and kb and kc;
  hysteresis( ka & kb & kc, kz );
end process;
```

The OR gate and `std_and` gate need no hysteresis.

```
process (ka, kb, kc)
begin
  kz <= ka or std_and(kb, kc);
end process;
```

To reduce mistakes, implement acknowledge circuits one function at a time; for example,

```
process (ka, kb, kc)
  variable kt : ack_logic;
begin
  kt := ka or kb;
  kz <= kt and kc;
  hysteresis( kc & kt, kz);
end process;

process (ka, kb, kc)
begin
  kt <= ka and kb;
  hysteresis( ka & kb, kz);
  kz <= kt or kc;
end process;
```

The above hysteresis statements assume that all inputs switch from all '0's, to a number of '1's, then back to all '0'. Thus, inverted signals should not be combined with uninverted signals. Inversions should be instantiated.

Using Concurrent Signal Assignments

Acknowledge logic does not have to be placed inside of processes. The NCL flow supports unconditional concurrent signal assignments with the hysteresis function:

```
function hysteresis( ain : ack_logic_vector) return boolean;
```

This function is used with threshold gates outside of a process. Follow the hysteresis rules described under [“Using Process Statements”](#). The function's input is the concatenation of all the variables that were used in the left-hand side of the assignment:

```
z <= a and b when hysteresis(a & b) else unaffected;
```

The statement includes three parts, the assignment, the hysteresis function, and the **else unaffected** clause. The last clause is necessary to implement the hysteresis.

The hysteresis function is not required. Without hysteresis, the code generates the proper netlist and identical source code simulation, with one difference. The AND statement outputs an 'X' when the inputs are '0' and '1', as shown in the truth table below.

Truth Table for concurrent assignment $z \leq a$ and b ;

a	b	z(t-1)	z(t)
'0'	'0'	'-'	'0'
'0'	'1'	'0'	'X'
'0'	'1'	'1'	'X'
'1'	'0'	'0'	'X'
'1'	'0'	'1'	'X'
'1'	'1'	'-'	'1'

Compare this to the truth table for the same assignment placed inside a process statement with a hysteresis procedure.

Truth Table for concurrent assignment $z \leq a$ and b when `hysteresis(a & b)` else unaffected;

a	b	z(t-1)	z(t)
'0'	'0'	'-'	'0'
'0'	'1'	'0'	'0'
'0'	'1'	'1'	'1'
'1'	'0'	'0'	'0'
'1'	'0'	'1'	'1'
'1'	'1'	'-'	'1'

When these inputs are connected to registers, the registers behave identically, since the registers will switch only on '0' or '1', not 'X'. When all inputs are '1', the acknowledge input to the register is '1'. The same is true when all signals go from '1' to '0'.

For example, the following code will simulate properly after synthesis, since a 2-of-2 threshold gate is generated.

```
signal ka, kb, kcon : ack_logic;
kcon <= ka and kb ;
```

However, the source code simulation will be different. The architecture in the example below shows the difference between the two coding styles.

Example: Acknowledge Signals using Processes and Concurrent Statements

```
architecture arch of th22 is
  signal inter : ack_logic;
begin
  -- concurrent signal assignment
  kcon <= ka and kb ;

  -- process
  process(ka, kb)
  begin
    inter <= ka and kb ;
```

```
        hysteresis(ka & kb, inter);  
    end process;  
    kpro <= inter;  
end arch;
```

Simulation of this code, shown in [Figure 19](#), shows that the concurrent acknowledge signal, *kcon*, goes to 'X' at 2 ns (as soon as one input is '0' and the other input is '1'). The acknowledge signal in the process, *kpro*, stays at '0'. However, both signals go to '1' at 3 ns. For designs with straightforward completion, where all acknowledge signals always go to '0' before starting the next DATA cycle and all signals go to '1' before starting the next NULL cycle, using this approach makes coding easier.

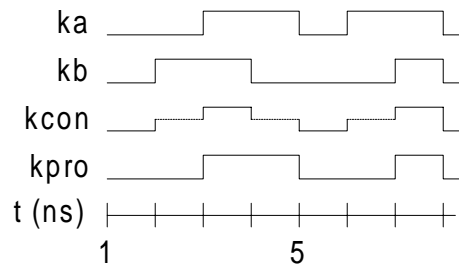


Figure 19 Source Code Simulation Showing Hysteresis Behavior

Register Usage Notes

These sections explain how to use registers in a design, including initialization, single-stage registers, and multi-stage registers.

Initialization

The following is the design practice you should follow:

- All registers driving combinational logic should be reset to NULL.
- All registers driven by combinational logic should be reset to NULL.

You also should follow these standards when creating your test benches. When all registers driving combinational logic output NULL, the combinational logic is forced to NULL, as illustrated in [Figure 20](#). You can see that inputs from the test bench (on the left) are also NULL.

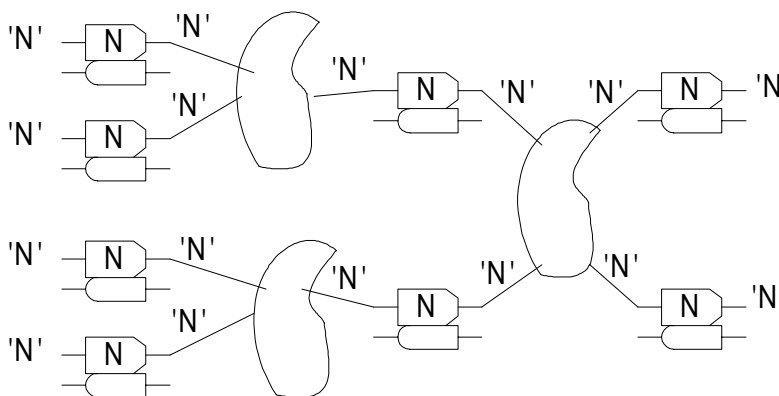


Figure 20 Initial State of Data Signals

When all registers driven by combinational logic are reset to NULL, then all acknowledge signals between logic will be '1', as shown in [Figure 21](#). Thus, with all the data and acknowledge signals in known states, the circuit will come out of reset in a known state and be ready to run. Note the application of the fork and join rules in the acknowledge path.

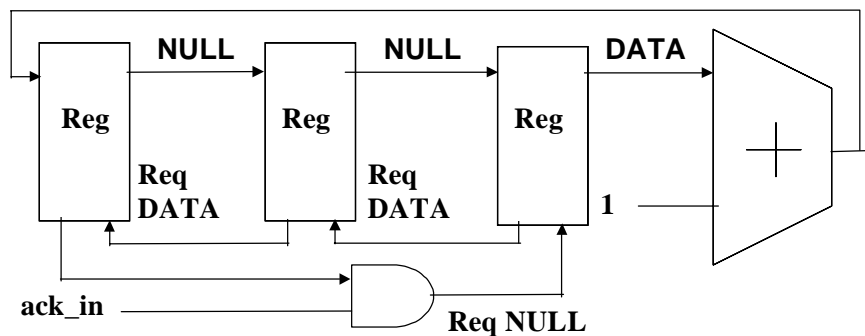


Figure 23 Counter with Three Register Stages

Three-stage registers are needed in filter-type designs, as shown in [Figure 24](#). During each data cycle, the middle register on the top will read data from the left and write data to both registers on the right. To do this, the design must have a three-stage register.

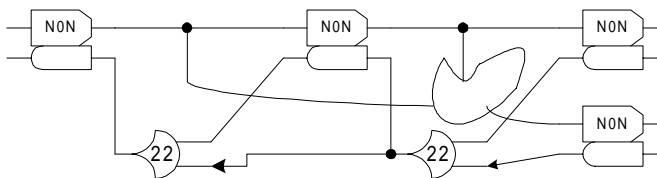


Figure 24 Filter Tap with Three-Stage Registers

