CPE 487: Digital System Design

Spring 2018

Lecture 7

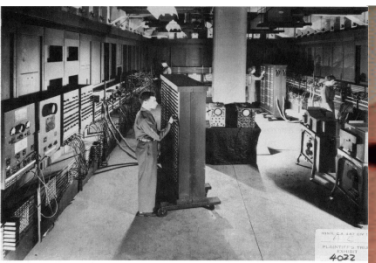
Behavioral Modeling: Variables

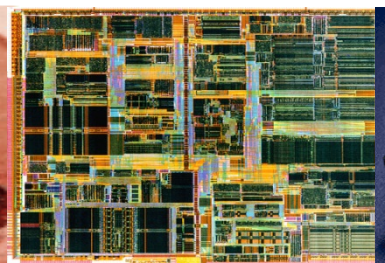
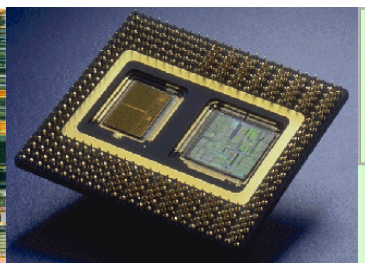
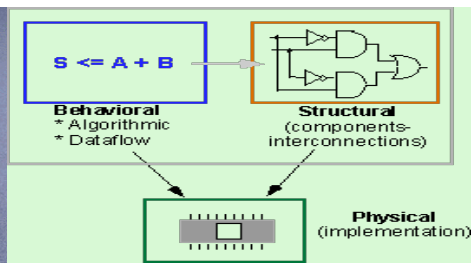
Bryan Ackland

Department of Electrical and Computer Engineering

Stevens Institute of Technology

Hoboken, NJ 07030



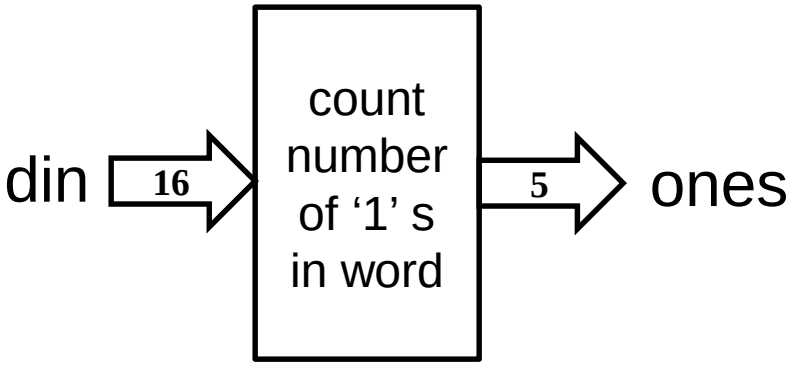
Variables

- In addition to signals, VHDL supports variables
  - Variables in VHDL are similar to variables in conventional programming languages
- Like signals, each variable has a type
- Like signals, variables have a present value
- Unlike signals, variables have no concept of future time
  - easier to use when calculating algorithmic result
  - "cheaper" to implement in simulator
  - no events associated with variables
- Variables are defined within a process and are not visible outside of the process
- Signals represent physical interconnect in circuits
- Variables are local values used to simplify process of calculating a result

Example: Count number of “ones”

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.STD_LOGIC_arith.all;

entity count1s is
   port (
     din: in  std_logic_vector (15 downto 0);
      ones: out std_logic_vector(4 downto 0));
end count1s;
```




```
architecture A1 of count1s is
begin
      p1: process (din)
      variable count: integer;
      begin
           count:=0;
           for i in 0 to 15 loop
              if din(i)=‘1’ then
                    count:= count+1;
              end if;
           end loop;
        ones<=conv_std_logic_vector(count,5)
                  after 5ns;
   end process;
end A1;
```

Variable Assignment Statement

Variable-object := *expression;*

- Expression may include both variables and signals. The present value of a signal is used in the computation
- Computation is performed in zero time (no delta delay)
  - Order of variable assignment statements is important
- Can only occur within process

Variable Assignment Statement: Examples

```
architecture RTL of VASSIGN is
     signal A, B, J  : bit_vector(1 downto 0);
     signal E, F, G  : bit;
begin
      p0 : process (A, J)
      variable C, D, H, Y : bit_vector(1 downto 0);
      variable W, Q       : bit_vector(3 downto 0);
      variable X          : bit;
     begin
           C       := "01";
           X       := E nand F;
           Y       := H or J;
           W       := C & D;
           Q       := (not A) & (A nor B);
           W       := (2 downto 1 => B, 3 => '1', others => '0');
           Y       := (others => '0');
    end process;
end RTL;
```

Review Architecture & Process

```
architecture RTL of OVERALL is
-- signals and constants can be declared here
-- variables CANNOT be declared here
begin
    -- concurrent signal assignment statements here
    -- NO variable assignment statements
    P1: process (SENSITIVITY_LIST)
    -- variables and constants can be declared here
    -- signals CANNOT be declared here
    begin
        -- sequential variable assignment statements here
        -- sequential signal assignment statements here
    end process P1;
end architecture RTL;
```

Wait Statement

- When process has sensitivity list, process is suspended until there is an event on one of the sensitive signals
- Alternatively, process can be suspended with use of wait statements:

wait for *time expression*;

```
wait for 25ns;
```

wait on *signal*;

```
wait on clk, reset;
```

wait until *condition*;

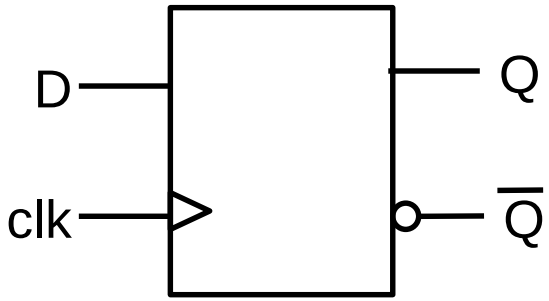
```
wait until index=0;
```

```
wait;              -- means wait forever;
```

- When using wait statements, the process does not suspend at the last statement in the process code, but continues executing from the top of the process.

Example: D Flip-Flop

```
entity Dff 2 is
   port (
     clk,D: in  std_logic;
     Q,Qb: out std_logic);
end entity Dff2;
```




```
architecture  DB of Dff2 is
begin
     ff2_pr: process
     begin
          wait until clk'event and clk='1' ;
          Q<=D after 7 ns;
          Qb<= not D after 7 ns;
     end process ff2_pr;
end architecture DB;
```

Wait Statement

- Possible to use multiple conditions, e.g:

  wait on X,Y **until** Z=0 **for** 100ns;

  means: wait for a maximum of 100ns for an event on X or Y when Z=0

- tpr: **process** (a,b)
  begin
  …..
  end process;

  is the same as

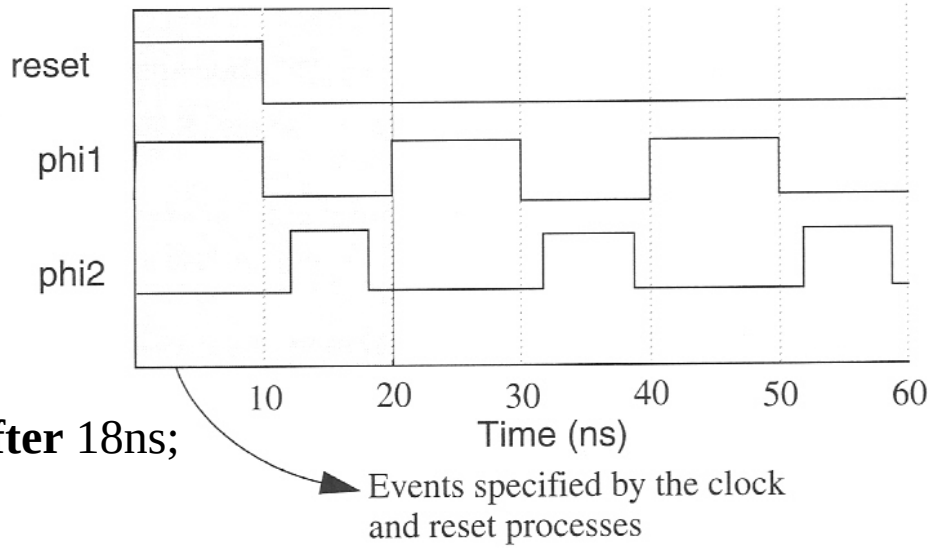
  tpr: **process**
  begin
  …..
  wait on a,b**;**
  end process;

- A process ***must have*** (a sensitivity list) ***or*** (one or more wait statements) ***but not*** both
- Regardless of whether a process uses a sensitivity list or wait statements, **all processes run at time=0**

Generating Periodic Waveforms

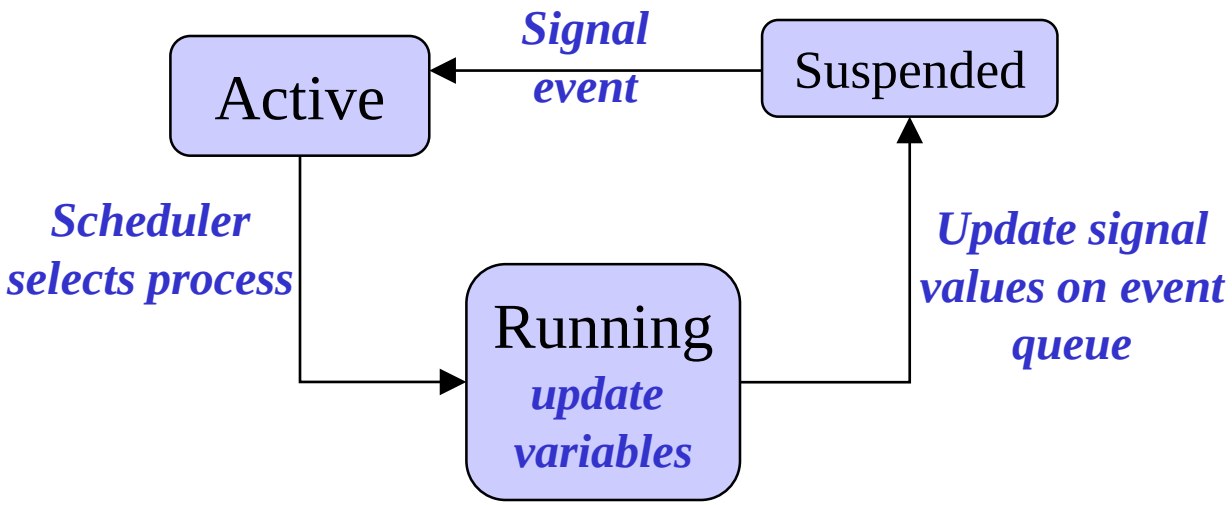
- Process with wait statement
  - Useful for test-bench inputs
- e.g: two-phase non-overlapping clock with reset:

```
architecture tpa of two_phase is
signal phi1, phi2, reset;
begin
reset_pr: reset<= ‘1’, ‘0’ after 10ns;
clk_pr: process
        begin
                phi1<=‘1’,’0’ after 10ns;
                phi2<=‘0’,‘1’ after 12 ns; ‘0’ after 18ns;
                wait for 20ns;
        end process;
end architecture tpa;
```



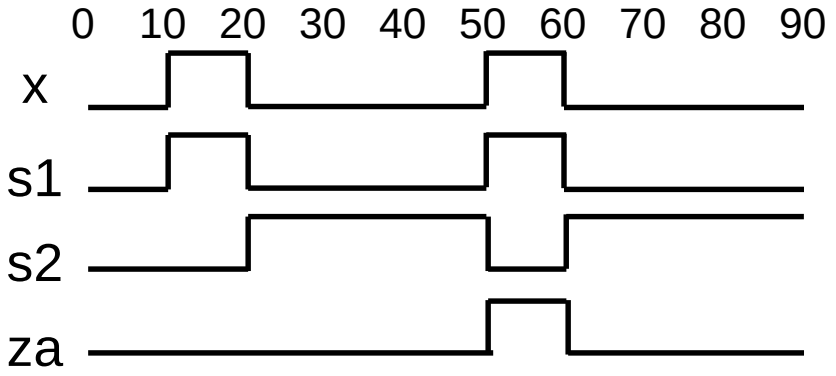
Timing of Variable and Signal Assignments

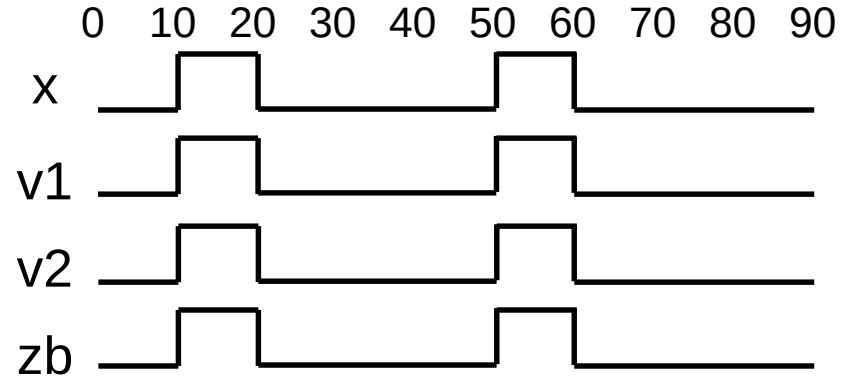
- Variables are assigned at the same time that the variable assignment is executed (zero-time)
  - Order of sequential variable assignment statements is important!
- Signals are assigned when the process is suspended
  - Can specify inertial, transport or zero delay
  - Zero delay signal assignment occurs at present time + Δ
  - How about order of sequential signal assignment statements?



Variable and Signal Timing Example

```
architecture A1 of sig_var is
signal s1, s2, x, za, zb: std_logic;
begin
    x<='0','1' after 10ns, '0' after 20ns, '1' after 50ns, '0' after 60 ns;
    pa: process (x)
    begin
        s1<=x;
        s2<=s1;
        za<=s2;
    end pa;
    pb: process(x)
    variable v1,v2:std_logic;
    begin
        v1:=x;
        v2:=v1;
        zb<=v2;
    end pb;
end A1;
```

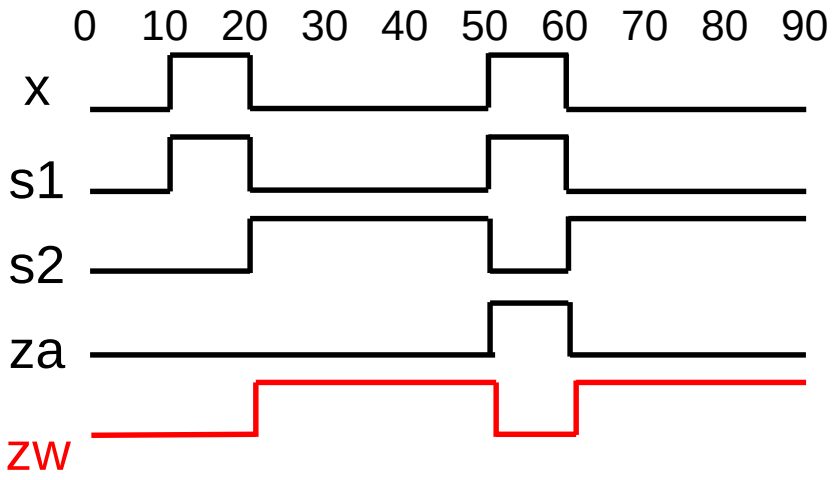




Wait for 0

- "Wait for 0" suspends process and then allows it to restart after a delay of only Δ
- Allows signal assignment to take effect before next statement is executed

```
architecture A1 of sig_var is
signal s1, s2, x, za, zb, zw: std_logic;
begin
    x<='0','1' after 10ns, '0' after 20ns, '1' after 50ns, '0' after 60 ns;
    pa: process
    begin
        wait on x;
        s1<=x;
        s2<=s1;
        za<=s2;
        wait for 0ns;
        zw<=s2;
    end pa;
end A1;
```



Loop Statement: While Iteration

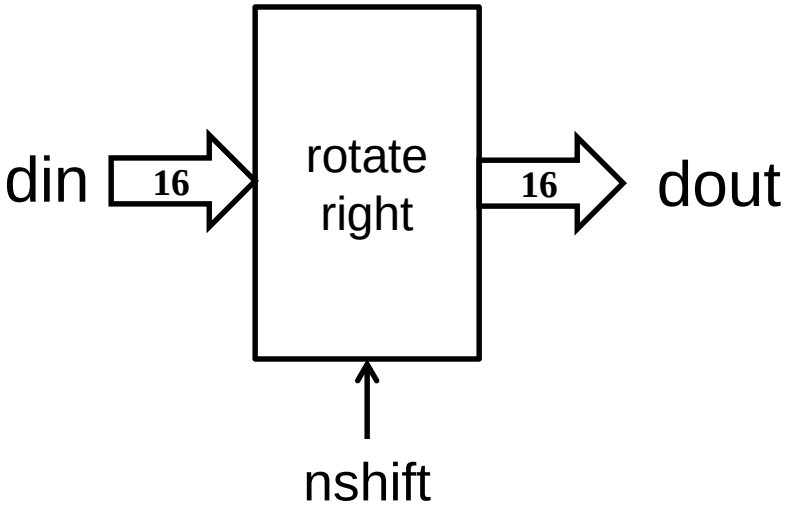
[*loop-label* :] **while** *condition* **loop**
sequential-statements
end loop [*loop-label*];

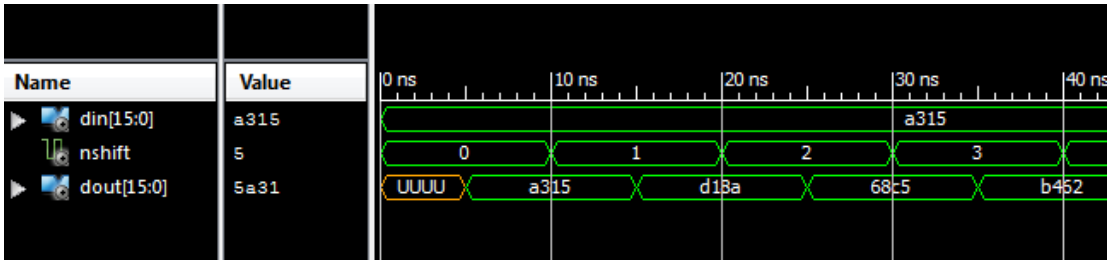
- *condition* is expression using previously declared signals and/or variables
- These signals and variables can be modified within the loop

While Example: Rotate Right Shift

```
entity rrs is
port(din:in std_logic_vector(15 downto 0);
     nshift:in integer;
     dout:out std_logic_vector(15 downto 0));
end rrs;

architecture beh of rrs is
begin
sh_pr: process
     variable nv:integer;
     variable dvar: std_logic_vector(15 downto 0);
     begin
          wait on din, nshift;
          nv:=nshift;
          dvar:=din;
          while nv>0 loop
             dvar:=dvar(0)&dvar(15 downto 1);
             nv:=nv-1;
          end loop;
          dout<=dvar after 5 ns;
     end process;
end beh;
```





Other Useful Sequential Control Instructions

- **exit** [*loop label*] [**when** *condition*];
  - Exit from loop (like C-language *break*). Must be enclosed by a loop statement with the same loop label. If the loop label is not specified, the *exit* always applies to the innermost loop

- **next** [*loop label*] [**when** *condition*];
  - Skip remaining statements in current iteration of the loop (like C-language *continue*). If the loop label is not specified, the *next* always applies to the innermost loop

Loop Examples: Factorial Calculation

```
factorial := 1;
FLP:    for number in 2 to N loop
            factorial := factorial *number;
end loop;
```

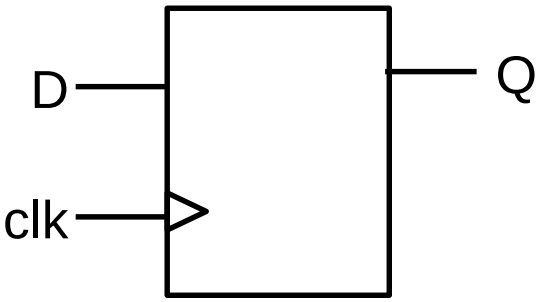
```
j := 2;
factorial := 1;
WLP :   while j<=N loop
            factorial := factorial *j;
            j:= j +1;
end loop;
```

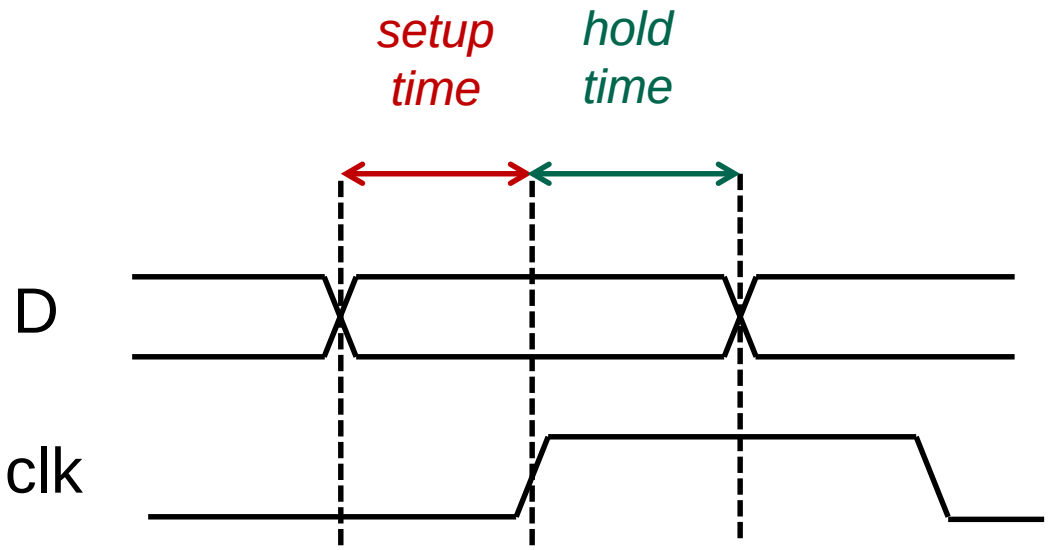
```
k := 1;
factorial :=1;
NLP :   loop
            factorial := factorial *k;
            k:= k +1;
            exit when k > N;
end loop;
```

These are all equivalent

Sequential Assert Statement

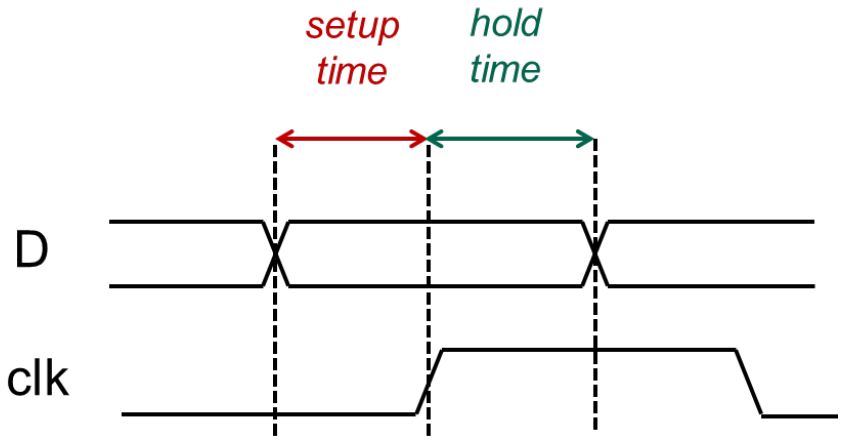
- Assert statement can also be used inside a process
- Assertion check will only be performed when statement is sequentially executed.
- For example: checking minimum required setup and hold times on D Flip-flop





Example: Setup & Hold Time

```
architecture behave of  DFF is
constant SETUP : time := 3 ns;
constant HOLD  : time := 2 ns;
begin
    setup_check : process (clk)
    begin
        if (clk = '1') then
            assert D‘stable(SETUP)
            report "D setup error" severity WARNING;
            Q <= D;
        end if;
    end process;
    hold_check: process (D)
    begin
        if (clk = '1') then
            assert (clk‘last_event > HOLD)
            report "D hold error" severity WARNING;
        end if;
    end process;
end behave;
```



Multiple Assignment Inside a Process

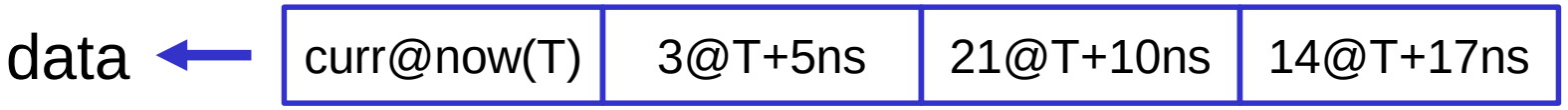
- Outside of a process, assignment statements are executed concurrently
  - multiple assignments to same signal are either illegal or invoke a resolution function

- Inside a process, assignment statements execute sequentially
  - An initial assignment can be overwritten by a subsequent assignment (much like a regular programming language)

Signal Drivers

- A single driver is created for every signal that is assigned a value in a process
  - The driver holds its current value and all its future values
  - All transactions on a driver are ordered in increasing time

```
signal integer data;
p1: process
begin
    data<= 3 after 5ns, 21 after 10ns, 14 after 17ns;
```



Multiple Assignment: Transport Delays

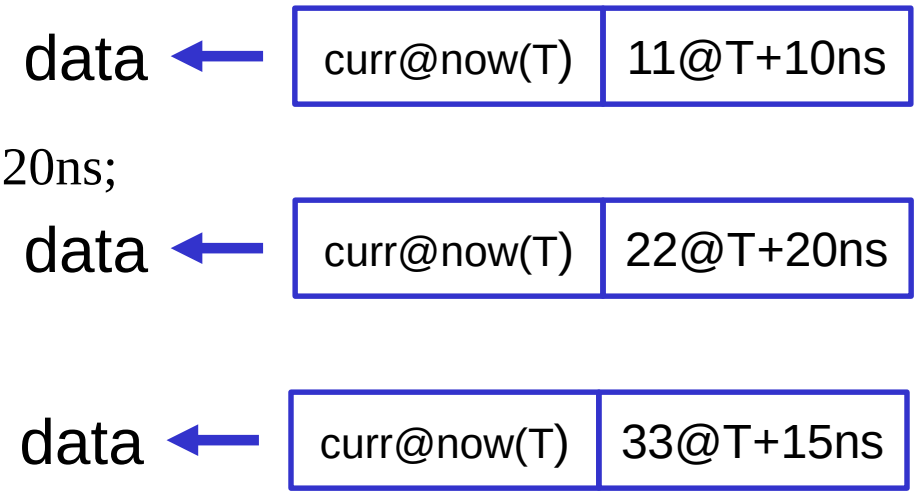
- Within a process, multiple assignments update driver according to the order in which they are executed.
- Transport delay rules:
  1. All transactions that occur at or after the delay time of the first new transaction are deleted.
  2. All the new transactions are added at the end of the driver

```
data<= transport 11 after 10ns;
       …
       …
data<= transport 20 after 22ns;
       …
       …
data<= transport 35 after 18ns;
```

Multiple Assignment: Inertial Delays

- Inertial delay rules:
  1. All transactions that occur at or after the delay time of the first new transaction are deleted.
  2. Add all the new transactions to the driver
  3. Delete old transactions that occur within pulse rejection limit of first new transaction if value is different to value of first new transaction

```
data<= 11 after 10ns;
       …
       …
data<= reject 15ns inertial 22 after 20ns;
       …
       …
data<= 33 after 15ns;
```



Sidebar: Signed & Unsigned Vectors

- We frequently use multi-bit digital words to represent integer values on which we would like to perform arithmetic and relational operations
- The std_logic_vector type is simply an array of bits with no implied digital value
  - Only logical operators (nand, xor, not etc.) are defined in the *IEEE.std_logic_1164* library
  - No arithmetic (+, - etc.) or relational (>, <= etc.) because these would require understanding of meaning of vector
  - Does it represent signed, unsigned, signed-magnitude, floating etc. ?

Operations on Unsigned, Signed Numbers

- USE ieee.numeric_std.all
  and
  signals of the type UNSIGNED, SIGNED
  and conversion functions:
  std_logic_vector(), unsigned(), signed()

OR

- USE ieee.std_logic_unsigned.all
  and
  signals of the type STD_LOGIC_VECTOR
  - ***all*** STD_LOGIC_VECTOR objects will be treated as unsigned
  - approach used in Yalamanchili

- There is also an ieee.std_logic_signed.all
  - ***all*** STD_LOGIC_VECTOR objects will be treated as signed
  - do not use both!

Unsigned Arithmetic Example

Suppose we want to add two 8-bit unsigned std_logic_vectors v1 and v2 to produce an 8-bit unsigned result v3 plus a carry-out

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

signal v1, v2, v3:
            std_logic_vector (7 downto 0);
signal  carry: std_logic;
signal u1, u2: unsigned (7 downto 0);
signal u3: unsigned (8 downto 0);

u1 <= unsigned (v1);
u2 <= unsigned (v2);
u3 <= ('0'&u1) + u2;
v3 <= std_logic_vector(u3(7 downto 0));
carry <= u3(8);
```

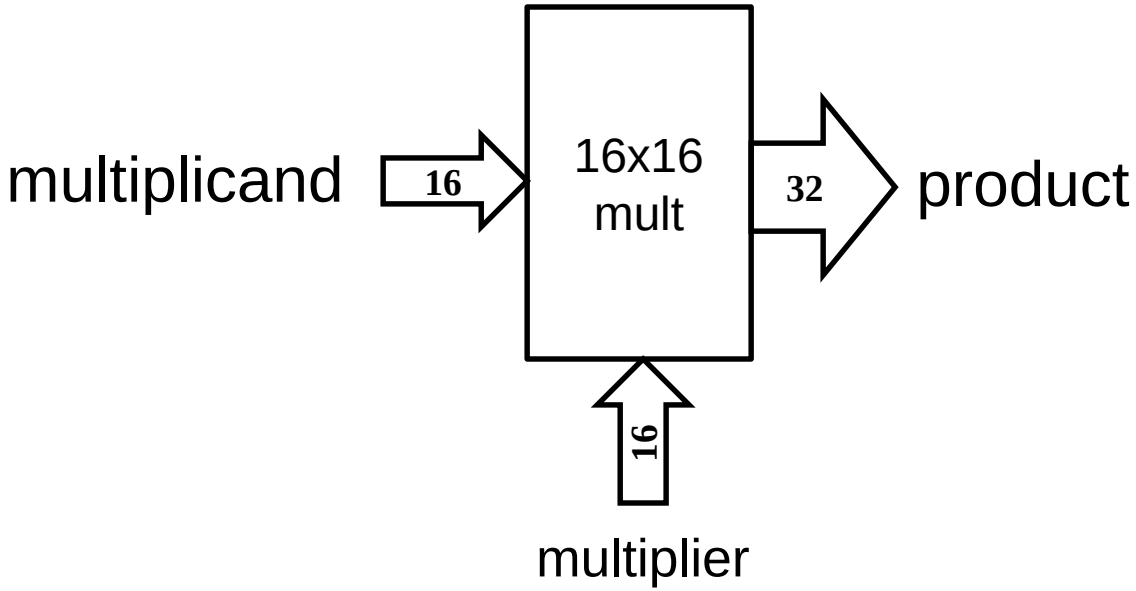
```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

signal v1, v2, v3:
            std_logic_vector (7 downto 0);
signal vtemp:
            std_logic_vector (8 downto 0);
signal  carry: std_logic;

vtemp <= ('0'&v1) + v2;
v3 <= vtemp(7 downto 0));
carry <= vtemp(8);
```

Example: 16-bit unsigned multiplier

- Construct a “shift and add” behavioral model of a 16x16 bit unsigned multiplier using a process and variables



Exercise: Waveform Evaluation

- Sketch the waveforms A, B, C, D, X and Y

```
entity waves is
end waves;

architecture Behavioral of waves is
signal A, B, C, D, X, Y: bit;
begin
p1:   process is
      begin
            A <= '0', '1' after 10 ns, '0' after 20 ns;
            wait for 25 ns;
      end process;
      B <= '0', '1' after 15 ns, '0' after 40 ns;
p2:   process (A,B) is
      begin
            C <= transport A xor B after 10 ns;
            D <= A xor B after 10 ns;
            X <= not C;
            Y <= not X;
      end process;
end Behavioral;
```

Exercise: (Cont.)