Introduction to VHDL -language features

COMP311
Tony McGregor
G.1.05
tonym@cs.waikato.ac.nz

Topics for the rest of VHDL

- type definitions
- type qualification and conversion
- scaler, array and signal attributes
- variables
- loops
- assert
- delays including delta delays
- discreet event simulators
- FSMs
  - traffic light controller
- 4-bit CPU

Types

- VHDL supports user defined types
- types must be declared in a package
  - separate file
  - must be compiled into a library (probably the work library)
- scalar types
- composite types

Scalar Types

- integer types
- floating point types
- physical types
- enumeration types

Composite Types

- array types
- record types

Integer Types

type year **is range** 2000 **to** 2020;

- VHDL has strong typing
  - can not assign a signal (or variable) of one type to another even if they are, for example, both integer types.

type born **is range** 2000 **to** 2020;

signal y : year;
signal x : born;
y <= x; (illegal)

Real Types

- just like integer types but with real numbers

type voltage **is range** 3.2 **to** 16.0;

Physical Types

- include the units of a value

```
type resistance is range 0 to 1E9
  units
       ohm;
   end units resistance;
```

  - example values:
    5 ohm, 22 ohm, 417_000 ohm
  - note the space between the value and the units, this is required.
    - Riviera does not enforce this

Physical Types

- can also define subunits

```
type length is range 0 to 1E9
  units
       um;
       mm = 1000 um;
       m  = 1000 mm;
       km = 1000 m;
       inch = 254000 um;
  end units resistance;
```

Time

- type time is predefined:

```
type time is range something
 units
    fs;
    ps  = 1000 fs;
    ns  = 1000 ps;
    us  = 1000 ns;
    ms  = 1000 us;
    sec = 1000 ms;
    min = 60 sec;
    hr  = 60 min;
end units;
```

Enumerated Types

- defined by a list of values
- for example:

```
type day is (mon, tue, wed, thu, fri,
  sat, sun);

type octal_digit is ('0', '1', '2',
  '3', '4', '5', '6', '7');
```

- only identifiers and character literals are allowed in the list but they may both occur in a single enumerated type.
- the type character is a predefined enumerated type

Subtypes

- subtypes limit the range of values available in a type

```
subtype age is integer range 0 to 120;
subtype weekend_day is day range
        sat to sun;
subtype capital is character range 'A'
  to 'Z';
```

- values of subtypes of the same type can be assigned to one another

A type package

```
package newtype is
  type dow is (mon, tue, wed, thu, fri, sat,
       sun);

  type res is range 1 to 5
    units
      ohm;
    end units;

  subtype weekend is dow range sat to sun;
  subtype capital is character range 'A' to 'Z';

end package newtype;
```

Type Qualification

- Sometimes you can't tell the type of a value

```
type logic_level is (unknown, '0',
  '1');
type system_state is (unknown, running,
  stopped);
```

  - The type of `unknown` is ... unknown
  - A type qualifier resolves this:

    `logic_level'unknown`

Type Conversion

- Similar types can be (explicitly) converted from one type to another:

```
typename(value)

real(123)
integer(3.6)
```

  - real to integer conversions round

- NOTE: this is different to type qualification

Type Attributes

- there are a range of attributes that give information about a type.
  - if we have a type T
    - `T'left` is the first (leftmost) value in T
    - `T'right` is the last
    - `T'low` is the smallest
    - `T'high` is the greatest
    - `T'ascending` is true if the range is ascending
    - `T'image(x)` a string representing the value of x
    - `T'value(s)` the value represented by the string s

Variables

- In addition to signals VHDL supports variables
- defined between a **`process`** statement and its **`begin`**
- assigned with **`:=`** not **<=**
- mostly a constrained version of signals
- variables are not visible outside the process they are defined in good practice to use a variable where it will do
  - just as it's good practice to use a `private` variable
- variables are updated immediately, not at the end of the process loop

Variables Example

```
architecture rtl of xor_gate is
begin
  XOR_GATE: process(a, b) is
  variable result : std_logic;
  begin
    result := a or b;
    if ( (a and b) = '1' )  then
       result := '0';
    end if
    o <= result;
  end process;
end architecture;
```

Broken Signal Example

What is wrong with the following?

```
architecture rtl of xor_gate is
signal result : std_logic;
begin
  XOR_GATE: process(a, b) is
  begin
    result <= a or b;
    if ( (a and b) = '1' )  then
      result <= '0';
    end if
    o <= result;
  end process;
end architecture;
```

Constants

- constants can be defined like variables
- need to have an initial value
  - variables may also have an initial value

```
constant e : real := 2.718281828;

constant prop_delay : time := 3 ns;
```

Control Structures

- VHDL supports control structures
  - loops
    - loop
    - while
    - for
  - conditionals
    - if
    - case
    - assert
    - report

Loop

```
process
begin
   statements;
   loop
      statements;
      wait on signal;
   or
      wait until signal = value;
   or
      wait for time;

   end loop;
end process;
```

Process Statement Loop

- The process statement is actually a **`loop`**
  - If there is a sensitivty list, there is an implict **`wait on`** at the end of the loop
  - If there is no sensitivity list there must be at least one **`wait`** within the process statement

Loop exits

- there are two forms of loop exit
  - **`exit`**;
    - unconditional exit
  - **`exit when`** (*`condition`*);
    - exit when *condition* is true
- **`exit`** may have a label to indicate which loop to exit
    - `exit` *`label`*
    - `exit` *`label`* `when (` *`condition`* `)`

Loop exit

```
process
variable count : integer;
begin
   statements;
   jovan: loop
      statements;
      exit jovan when (count = 10);
      wait for time;
      statements;
   end loop;
end process;
```

Next statement

- similarly there are two forms of next statement
  - cause the loop to restart from the beginning
  - `next;`
    - unconditional return to start
  - `next when ( condition );`
    - return to start when *`condition`* is true
- `next` may also have a label

Next example

```
process(clk)
variable count : integer;
begin
   statements;
   jovan: loop
      statements;
      next jovan when (clk = '0');
      wait for time;
      statements;
   end loop;
end process;
```

While

```
while ( condition ) loop
   statements;
end loop;
```

- loops may also have a label

```
retry: while ( condition ) loop
   statements;
end loop retry;
```

For

```
for identifier in range loop
    statements;
end loop
```

- may have a label
- identifier is only in scope inside the loop
- it can not be modified inside the loop

For loop example

```
hidden_eg process is
   variable a, b : integer;
begin
   a := 10;
   for a in 0 to 7 loop
       b := a;
   end loop;

      --a = 10, b = 7

end process hidden_eg;
```

If

```
if condition then
    statements;
elsif condition then
    statements;
else
    statements;
end if;
```

- may have a label

Case

- an abbreviated form of an if statement with several **elsif** clauses

```
case value is
   when range => statement;
   when range => statement;
     ...
   when others => statement;
end case;
```

Case example

```
case opcode is
  when load|add|subtract =>
    operand := memory_operand;
  when store to branch =>
    operand := address_operand;
  when others
    operand := 0;
end case
```

Case statement

- the value selected on and selection items in a case statement must be a discrete type or a one dimensional array.
- the selection values must be static (determined at analysis time). Constants OK but variables are not.
- may have a label

Assert

```
assert condition
    report expression
    severity expression;
```

- report and severity are optional
- severity is note, warning, error or failure
  - error is the default
- simulator may stop at a given severity (or above)

Assert example

```
assert memory >= low_memory
    report "low on memory"
    severity note;
```

Array Types

- VHDL supports single and multi-dimensional arrays

```
type nibble is array (3 downto 0) of
  std_logic;


type state is (off, warming,
  waitingforwork, running, stopping)
type work_counts is array
  (waitingforwork to running) of
  natural;
```

Multidimensional arrays

```
type input1 is ('a', 'b', 'c');
type input2 is range 0 to 6;

type input_counts is array(input1,
  input2) of natural;

variable input_counter : input_counts;
input_counter('b', 3) := 0;
```

Array Attributes

- if we have an array type A and N, a dimension of the array (between 1 and the number of dimensions)
  - `A'left(N)` is the first (left most) value in Nth index of A
  - `A'right(N)` is the last value in the Nth index
  - `A'low(N)` is the smallest ..
  - `A'high(N)` is the greatest ...
  - `A'ascending(N)` is true if the range for the Nth dimension is ascending
  - `A'range(N)` index range
  - `A'reverse_range(N)` reverse of range
  - `A'length(N)` length of index range

Array attributes -example

```
type A is array (1 to 4, 31 downto 0)
of boolean;
```

- `A'left(1)   = 1`
- `A'right(2)  = 0`
- `A'range(1)  is 1 to 4`
- `A'length(1) = 4`
- `A'asccending(1) = TRUE`

- `A'low = 1`
- `A'length = 4`

- `A'low(1) = 1`
- `A'high(2) = 31`
- `A'reverse_range(2) is 0 to 31`
- `A'length(2) = 32`
- `A'ascending(2) = FALSE`

Unconstrained Arrays

- dimensions supplied in variable (or signal) declaration, not type definition

```
type dag is array (natural range <>) of
  integer;

variable clark : dag(3 downto 0);
```

- `string` is a predefined unconstrained array

```
variable name : string(1 to 32) := “F Dag”;
```

std_logic_vector

- the std_logic_vector type we have used is an unconstrained array

```
signal john : std_logic_vector(3 downto 0);

type sdt_logic_vector is array (natural
   range <>) of std_logic;
```

Unconstrained Array Ports

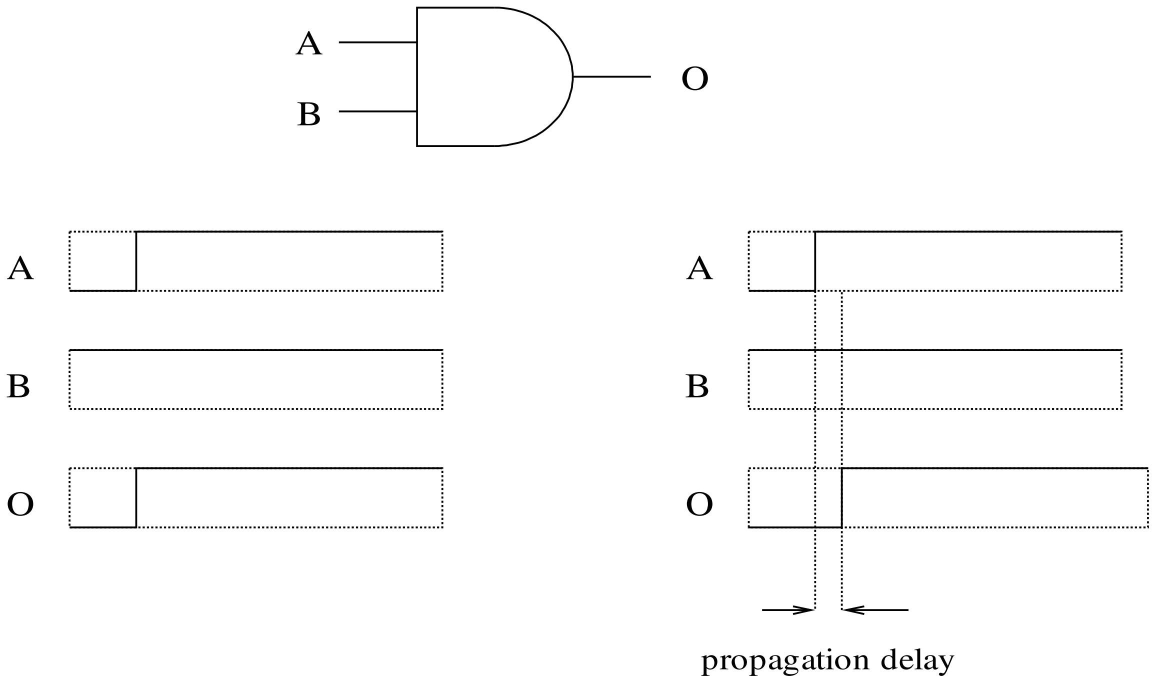
```
entity andn is
  port(i : in  std_logic_vector;
       o : out std_logic);
end entity andn;

architecture behavioural of andn is
begin
  ANDN:process(i) is
    variable result : std_logic;
  begin
    result := '1';
    for index in i'range loop
      result := result and i(index);
    end loop;
    o <= result;
  end process andn
end architecture behavioural;
```

Unconstrained Array ports

```
architecture tb of andn_tb is
    component andn
      port ( i  : in  std_logic_vector(7 downto 0);
             o  : out std_logic);
    end component;
 signal i_i  : std_logic_vector(7 downto 0);
 signal o_i  : std_logic;
begin  -- tb
  DUT: andn port map (
        i => i_i,
        o => o_i);
  test : process
  begin
    i_i <= "00000000";  wait for 10 ns;
    i_i <= "00011100";  wait for 10 ns;
    i_i <= "01111111";  wait for 10 ns;
    i_i <= "11111111";  wait for 10 ns;
    wait;
    end process;
end tb;
```

Propagation Delays

- Gates do not transmit signals instantly
  - they have a propagation delay



Propagation Delays in VHDL

- we can add an 'after' clause to include a propagation delay

```
architecture rtl of half is
begin
  s    <= (a xor b) xor cin after 1 ns;
  cout <= (a xor b) and cin) or
          (a and b) after 1.5 ns;
end rtl
```

4-bit ripple adder with propagation delays

- riviera demo

Signal Attributes

- `S'event` true if there is a change in S in the current process cycle
- `S'active` true if the value of S is set in this cycle
- `S'transaction` a signal of type bit that changes value each time there is a transaction on S
- `S'last_event` the time since the last event on S
- `S'last_active` the time since last transaction on S

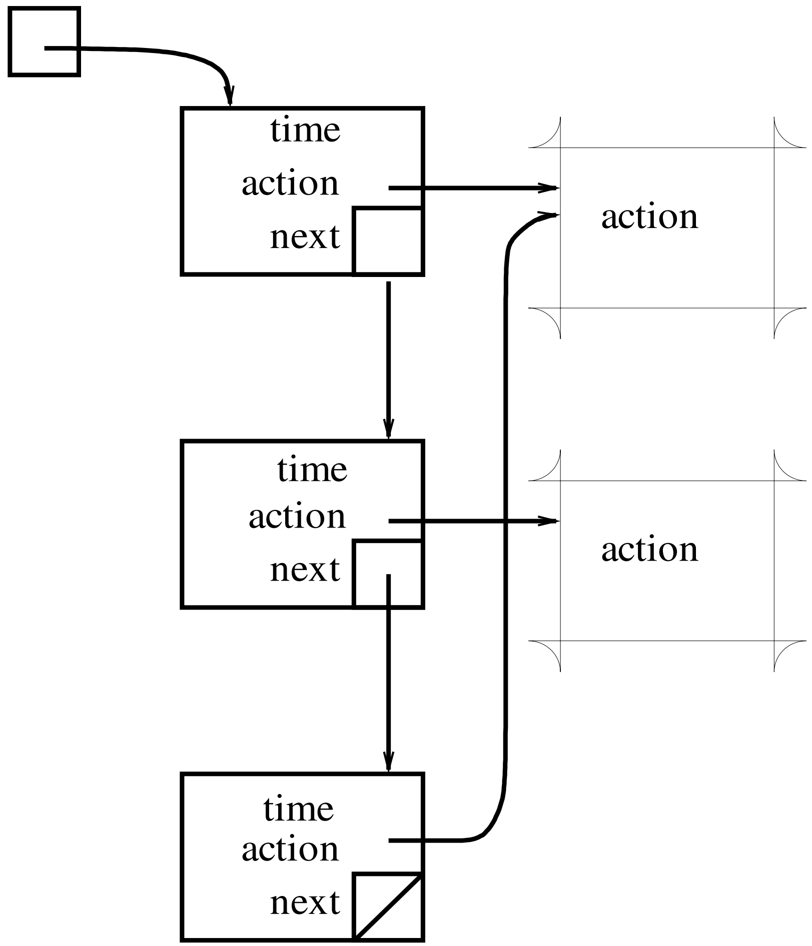
Signal Attributes

- `S'last_value` the value of S before the last event
- `S'quiet(T)` True is there has been no transaction in S in the most recent time interval T
- `S'stable(T)` True if there has been no event (change) in S in the most recent time interval T
- `S'delayed(T)` A signal that has the same value as S but delayed by time T

Discreet event simulation

- A discreet event simulator is based around an ordered list of events
- Each entry in the list contains
  - the time the item is scheduled for
  - the action that should be taken at that time
  - a link to the next item
- Actions often trigger other actions
  - they put new items into the event list.

The Event Chain



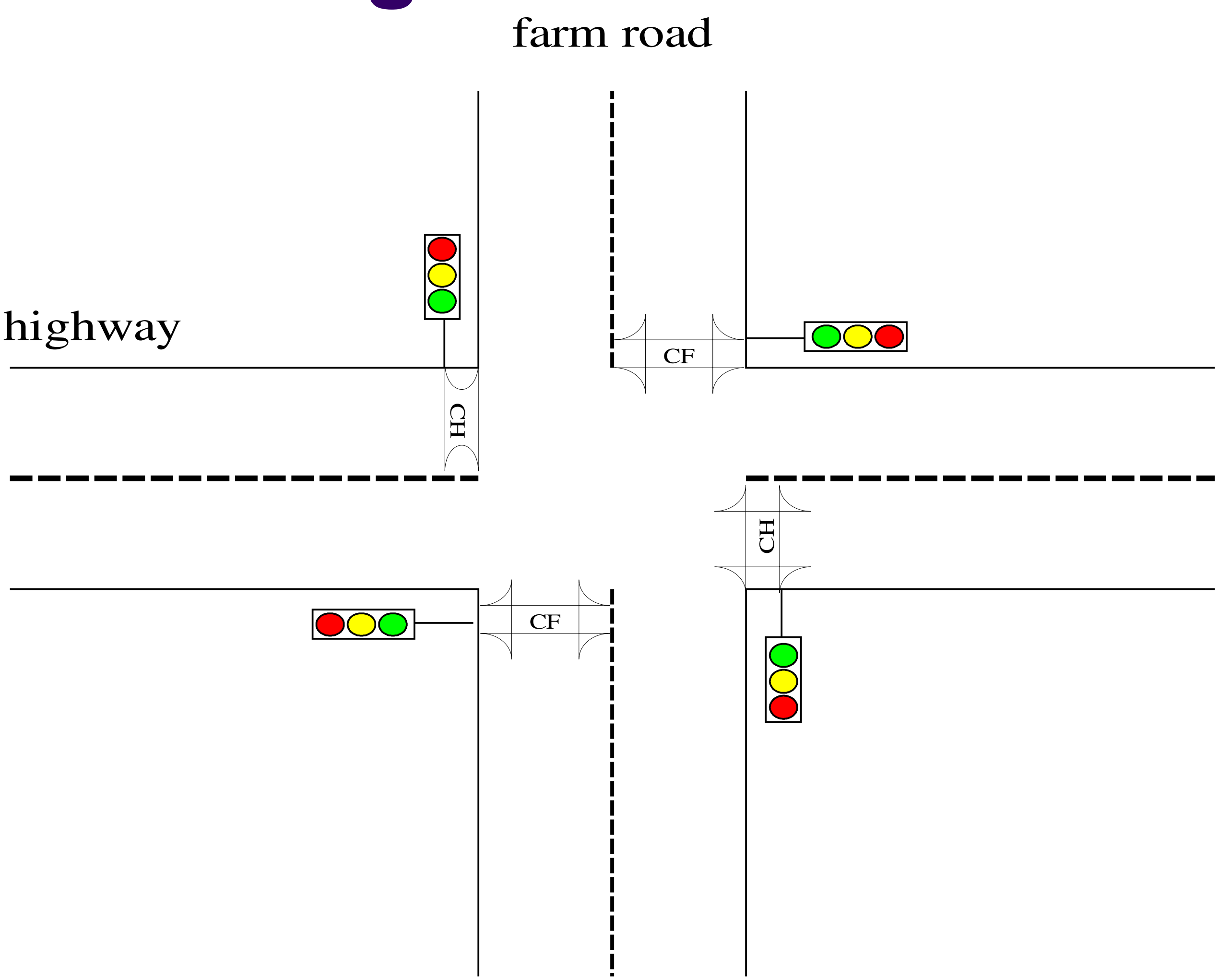
Discreet Event Simulation

- The simulator is a loop that removes the top event off the list and runs the action associated with that event.
- When that event is completed the event that is now at the top of the list is processed
- Simulated time skips from the time in the previous event to the time in the next event
  - no association with real time
- The simulation stops when there are no events left on the list or when a pre-set time limit is exceeded

Delta Time

- When a signal value is changed a new event is added to the event list
- If the event is 'immediate' it goes at the top of the list
- when the current event is complete the next event on the list will be selected
- this explains the notion of 'delta time'
  - signal values don't change until the end of a process block

Traffic Light Controller



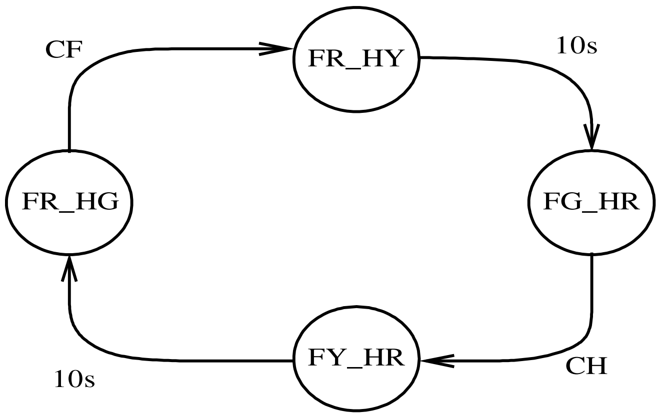
Signals and States

- We have two inputs
  - Car on the Farm road (CF)
  - Car on the Highway (CH)

- Four states that the system can be in
  - Red on the Farm road and Green on the highway (`FR_HG`)
  - Red on the Farm road and Yellow on the Highway (`FR_HY`)
  - Green on the Farm road and Red on the Highway (`FG_HR`)
  - Yellow on the Farm road and Red on the Highway (`FY_HR`)

Finite State Machines

- Commonly used in digital design for control circuits
- Usually described as a state transition diagram



current state	event	next state
FR_HG	CF	FR_HY
FR_HY	delay	FG_HR
FG_HR	CH	FY_HR
FY_HR	delay	FR_HG

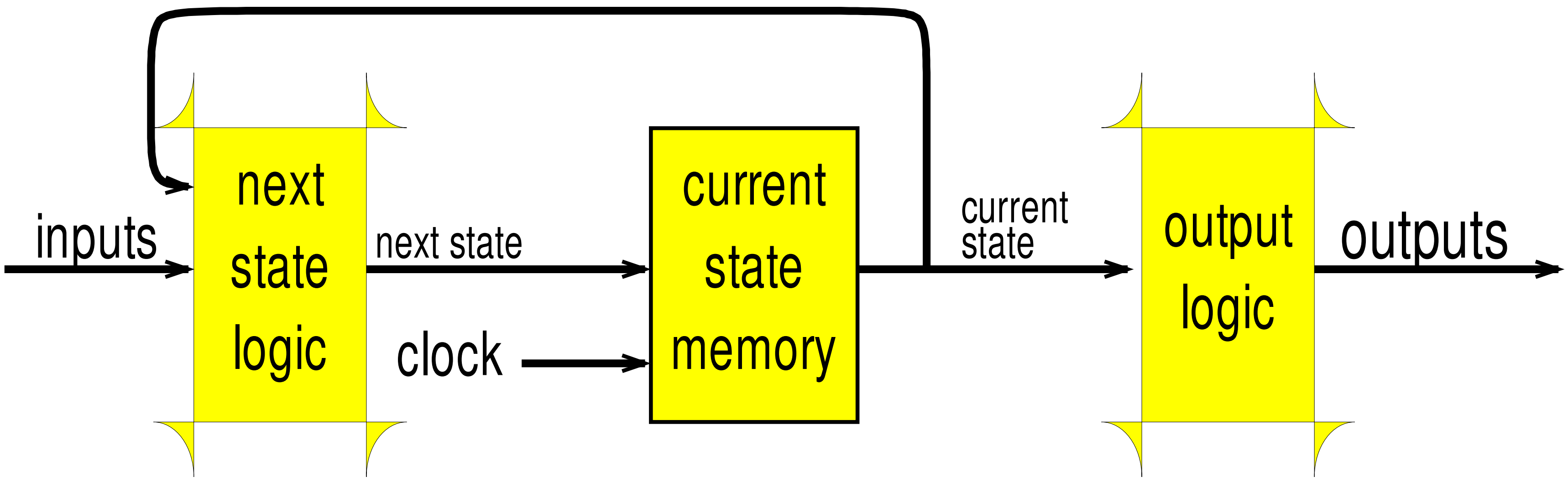
Generating the light signals

- The light signals can be generated from the state of the system:

HGreen **<=** HG_FR
HYellow **<=** HY_FR
HRed **<=** FG_HR **or** FY_HR

FGreen **<=** FG_HR
FYellow **<=** FY_HR
FRed **<=** HG_FR **or** HY-FR

Moore Machines



Moore Machines in VHDL

- Use an enumerated type to represent the states

```
type STATE_TYPE is
                (HG_FR, HY_FR, FG_HR, FY_HR);
```

- use a process for:
  - next state logic
  - state flip-flops
  - output logic

Entity Definition

```
entity traffic is
  port (clock   : in std_logic;
        CF      : in std_logic;
        CH      : in std_logic;
        HGreen  : out std_logic;
        HYellow : out std_logic;
        HRed    : out std_logic;
        FGreen  : out std_logic;
        FYellow : out std_logic;
        FRed    : out std_logic);
end traffic;
```

Architecture outline

```
architecture RTL of traffic is
  type STATE_TYPE is (HG_FR, HY_FR, FG_HR, FY_HR);
  signal current_state, next_state : STATE_TYPE := HG_FR;
begin
  NS: process(current_state, CF, CH)           --next state
  begin
      ...
  end process ns;

  SEQ: process(clock)                          --state memory
  begin
      ...
  end process seq;

  OUTPUTS: process(current_state)              --output logic
  begin
     ...
  end process outputs;
end;
```

Next state logic

```
NS: process(current_state, CF, CH)
begin
  case current_state is
    when HG_FR =>
      if CF = '1' then next_state <= HY_FR;
      else next_state <= HG_FR;
      end if;
    when HY_FR =>  next_state <= FG_HR;
    when FG_HR =>
      if CH = '1' then next_state <= FY_HR;
      else next_state <= FG_HR;
      end if;
    when FY_HR => next_state <= HG_FR;
  end case;
end process;
```

Traffic Light Controller - flip flops

```
SEQ: process(clock)
 begin
   if rising_edge(clock) then
      current_state <= next_state;
   end if;
 end process;
```

Traffic Light Controller - outputs

```
outputs: process(current_state)
begin
   HGreen <= '0'; HYellow <= '0'; HRed <= '0';
   FGreen <= '0'; FYellow <= '0'; FRed <= '0';
   case current_state is
      when HG_FR =>
         HGreen  <= '1'; FRed <= '1';
      when HY_FR =>
         HYellow <= '1'; FRed <= '1';
      when FG_HR =>
         FGreen  <= '1'; HRed <= '1';
      when FY_HR =>
         FYellow <= '1'; HRed <= '1';
   end case;
end process;
```

Traffic Light Controller - timing

- Rivera simulation

FSM Initialisation

- Simulator assumes default value of an object is its left most value
- In traffic light controller want FG_HR as initial state

```
seq: process(clock, reset)
  begin
    if reset = '1' then
      current_state <= FG_HR;
    elsif clock = '1' then
      current_state <= next_state;
    end if;
  end process;
```

4-bit CPU