The VHDL Hardware Description Language

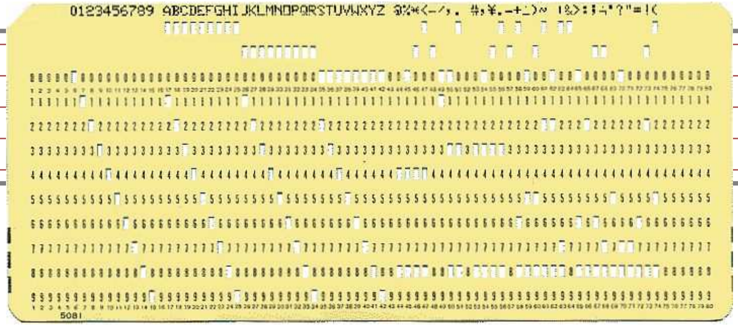
Prof. Stephen A. Edwards

sedwards@cs.columbia.edu

Columbia University

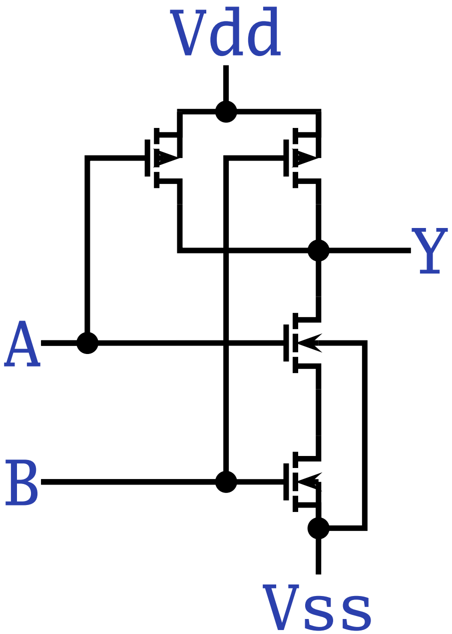
Spring 2007

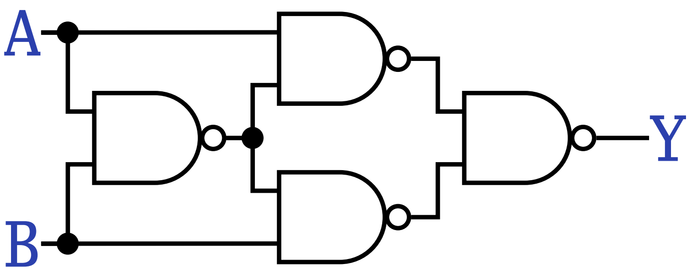
Why HDLs?



1970s: SPICE transistor-level netlists

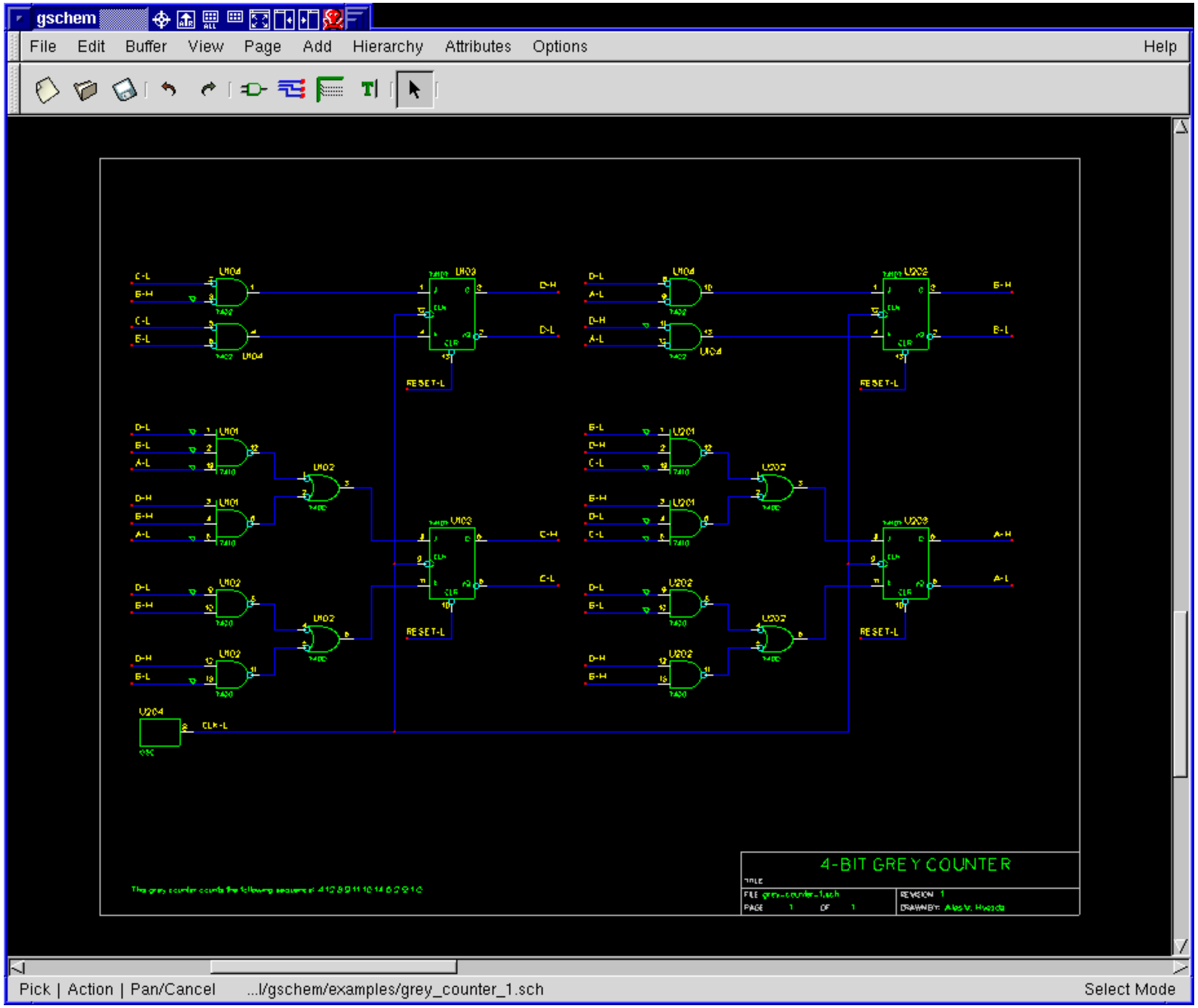
```
An XOR built from four NAND gates

.MODEL P PMOS
.MODEL N NMOS

.SUBCKT NAND A B Y Vdd Vss
M1 Y A Vdd Vdd P
M2 Y B Vdd Vdd P
M3 Y A X   Vss N
M4 X B Vss Vss N
.ENDS

X1 A  B  I1 Vdd 0 NAND
X2 A  I1 I2 Vdd 0 NAND
X3 B  I1 I3 Vdd 0 NAND
X4 I2 I3 Y  Vdd 0 NAND
```





Why HDLs?

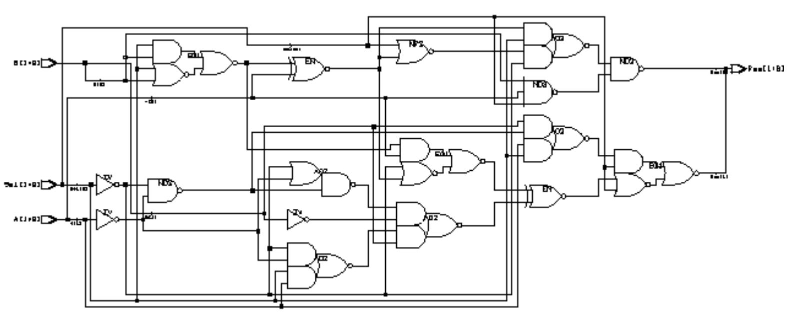
1980s: Graphical schematic capture programs



Why HDLs?

1990s: HDLs and Logic Synthesis

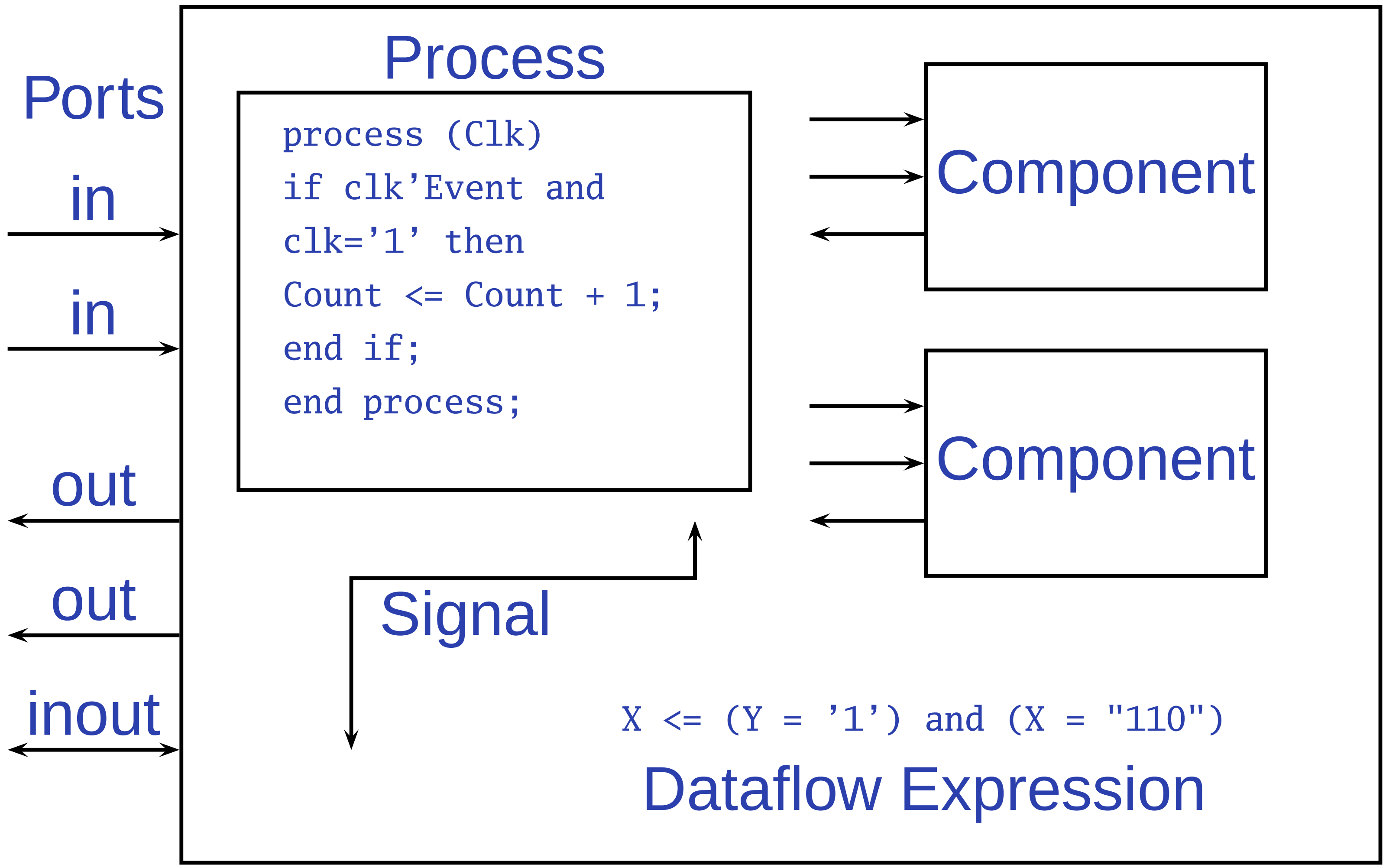
```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.std_logic_arith.all;
entity ALU is
port(   A:      in std_logic_vector(1 downto 0);
        B:      in std_logic_vector(1 downto 0);
        Sel:    in std_logic_vector(1 downto 0);
        Res:    out std_logic_vector(1 downto 0));
end ALU;
architecture behv of ALU is begin
  process(A,B,Sel) begin
    case Sel is
      when "00" => Res <= A + B;
      when "01" => Res <= A + (not B) + 1;
      when "10" => Res <= A and B;
      when "11" => Res <= A or B;
      when others => Res <= "XX";
    end case;
  end process;
end behv;
```



Two Separate but Equal Languages

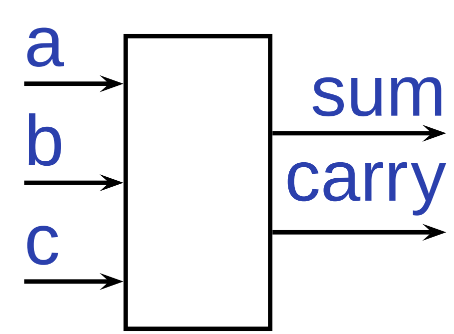
- Verilog and VHDL
- Verilog: More succinct, less flexible, really messy
- VHDL: Verbose, very (too?) flexible, fairly messy
- Part of languages people actually use identical.
- Every synthesis system supports both.

VHDL: Hierarchical Models



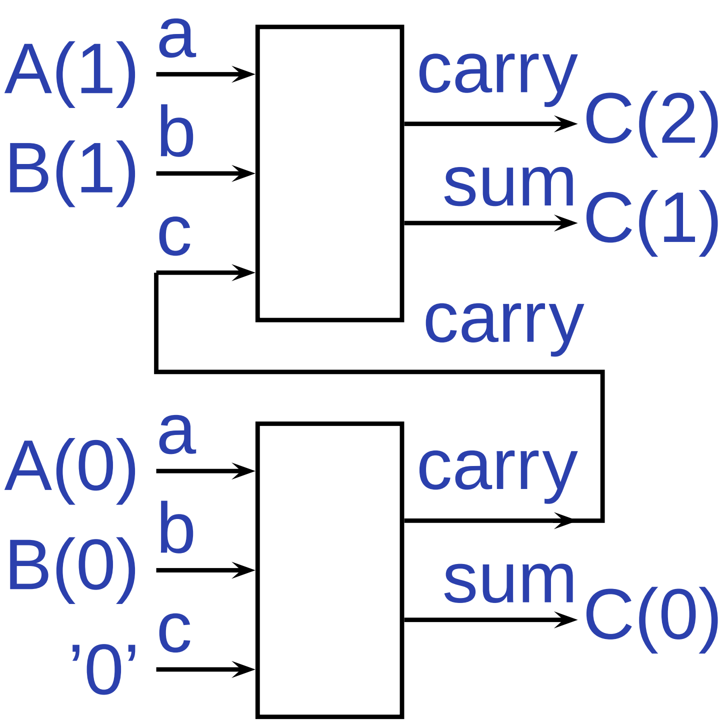
Basic VHDL: Full Adder

```
library ieee;                    -- part of IEEE library
use ieee.std_logic_1164.all; -- includes std_ulogic

entity full_adder is
  port(a, b, c     : in  std_ulogic;
       sum, carry : out std_ulogic);
end full_adder;

architecture imp of full_adder is
begin
  sum   <= (a xor b) xor c;   -- combinational logic
  carry <= (a and b) or (a and c) or (b and c);
end imp;
```



VHDL: Two-bit Counter


```
library ieee;
use ieee.std_logic_1164.all;

entity add2 is
  port (
    A, B : in  std_logic_vector(1 downto 0);
    C    : out std_logic_vector(2 downto 0));
end add2;

architecture imp of add2 is
  component full_adder
    port (
      a, b, c     : in  std_ulogic;
      sum, carry : out std_ulogic);
  end component;
  signal carry : std_ulogic;
begin
  bit0 : full_adder port map (
    a     => A(0),
    b     => B(0),
    c     => '0',
    sum   => C(0),
    carry => carry);
  bit1 : full_adder port map (
    a     => A(1),
    b     => B(1),
    c     => carry,
    sum   => C(1),
    carry => C(2));
end imp;
```



Four-to-one multiplexer: when...else

```
library ieee;
use ieee.std_logic_1164.all;

entity multiplexer_4_1 is
  port(in0, in1 : in  std_ulogic_vector(15 downto 0);
       in2, in3 : in  std_ulogic_vector(15 downto 0);
       s0, s1   : in  std_ulogic;
       z        : out std_ulogic_vector(15 downto 0));
end multiplexer_4_1;

architecture imp of multiplexer_4_1 is
begin
  z <= in0 when (s0 = '0' and s1 = '0') else
       in1 when (s0 = '1' and s1 = '0') else
       in2 when (s0 = '0' and s1 = '1') else
       in3 when (s0 = '1' and s1 = '1') else
       "XXXXXXXXXXXXXXXX";
end imp;
```

Four-to-one mux: with...select

```
library ieee;
use ieee.std_logic_1164.all;

entity multiplexer_4_1 is
  port(in0, in1 : in  std_ulogic_vector(15 downto 0);
       in2, in3 : in  std_ulogic_vector(15 downto 0);
       s0, s1   : in  std_ulogic;
       z        : out std_ulogic_vector(15 downto 0));
end multiplexer_4_1;

architecture usewith of multiplexer_4_1 is
  signal sels : std_ulogic_vector(1 downto 0);
begin
  sels <= s1 & s0;  -- Vector concatenation

  with sels select
    z <=
    in0                when "00",
    in1                when "01",
    in2                when "10",
    in3                when "11",
    "XXXXXXXXXXXXXXXX" when others;
end usewith;
```

Three-to-eight Decoder

```
library ieee;
use ieee.std_logic_1164.all;

entity dec1_8 is
port (
  sel : in  std_logic_vector(2 downto 0);
  res : out std_logic_vector(7 downto 0));
end dec1_8;

architecture imp of dec1_8 is
begin
  res <= "00000001" when sel = "000" else
         "00000010" when sel = "001" else
         "00000100" when sel = "010" else
         "00001000" when sel = "011" else
         "00010000" when sel = "100" else
         "00100000" when sel = "101" else
         "01000000" when sel = "110" else
         "10000000";
end imp;
```

Priority Encoder

```
library ieee;
use ieee.std_logic_1164.all;

entity priority is
  port (
    sel  : in  std_logic_vector(7 downto 0);
    code : out std_logic_vector(2 downto 0));
end priority;

architecture imp of priority is
begin
  code <= "000" when sel(0) = '1' else
          "001" when sel(1) = '1' else
          "010" when sel(2) = '1' else
          "011" when sel(3) = '1' else
          "100" when sel(4) = '1' else
          "101" when sel(5) = '1' else
          "110" when sel(6) = '1' else
          "111" when sel(7) = '1' else
          "---";         -- "-" is "don't care"
end imp;
```

Integer Arithmetic

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity adder is
  port (
    A, B : in  std_logic_vector(7 downto 0);
    CI   : in  std_logic;
    SUM  : out std_logic_vector(7 downto 0);
    CO   : out std_logic);
end adder;

architecture imp of adder is
signal tmp : std_logic_vector(8 downto 0);
begin
  tmp <= conv_std_logic_vector((conv_integer(A) +
                                conv_integer(B) +
                                conv_integer(CI)), 9);
  SUM <= tmp(7 downto 0);
  CO  <= tmp(8);
end imp;
```

A Very Simple ALU

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

entity alu is
  port (
    A, B : in  std_logic_vector(7 downto 0);
    ADD  : in  std_logic;
    RES  : out std_logic_vector(7 downto 0));
end alu;

architecture imp of alu is
begin
  RES <= A + B when ADD = '1' else
         A - B;
end imp;
```

Arithmetic Comparison

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

entity comparator is
  port (
    A, B : in  std_logic_vector(7 downto 0);
    GE   : out std_logic);
end comparator;

architecture imp of comparator is
begin
  GE <= '1' when A >= B else '0';
end imp;
```

Generate: Ripple-carry adder

```
library ieee;
use ieee.std_logic_1164.all;

entity rippleadder is
  port (a, b : in  std_ulogic_vector(3 downto 0);
        cin  : in  std_ulogic;
        sum  : out std_ulogic_vector(3 downto 0);
        cout : out std_ulogic);
end rippleadder;

architecture imp of rippleadder is
  signal c : std_ulogic_vector(4 downto 0);

begin
  c(0) <= cin;

  G1: for m in 0 to 3 generate        -- at compile time
    sum(m) <= a(m) xor b(m) xor c(m);
    c(m+1) <= (a(m) and b(m)) or (b(m) and c(m)) or
              (a(m) and c(m));
  end generate G1;

  cout <= c(4);

end imp;
```

Basic Flip-Flop

```
library ieee;
use ieee.std_logic_1164.all;

entity flipflop is
  port (Clk, D : in  std_ulogic;
        Q      : out std_ulogic);
end flipflop;

architecture imp of flipflop is
begin

  process (Clk)            -- Process sensitive to Clk
  begin
    if (Clk'event and Clk = '1') then  -- Rising edge
      Q <= D;
    end if;
  end process P1;

end imp;
```

Flip-Flop with Synchronous Reset

```
library ieee;
use ieee.std_logic_1164.all;

entity flipflop_reset is
  port (Clk, Reset, D : in  std_ulogic;
        Q             : out std_ulogic);
end flipflop_reset;

architecture imp of flipflop_reset is
begin

  P1: process (Clk)
  begin
    if (Clk'event and Clk = '1') then
      if (Reset = '1') then Q <= '0';
      else Q <= D;
      end if;
    end if;
  end process P1;

end imp;
```

Four-bit binary counter


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

entity counter is
   port(
     Clk, Reset : in  std_logic;
     Q          : out std_logic_vector(3 downto 0));
end counter;

architecture imp of counter is
  signal count : std_logic_vector(3 downto 0);
  begin
    process (Clk)
    begin
      if (Clk'event and Clk = '1') then
        if (Reset = '1') then
           count <= "0000";
        else
           count <= count + 1;
        end if;
      end if;
    end process;

    Q <= count;        -- copy count to output

end imp;
```

Eight-bit serial in/out shift register

```
library ieee;
use ieee.std_logic_1164.all;

entity shifter is
  port (
    Clk : in  std_logic;
    SI  : in  std_logic;
    SO  : out std_logic);
end shifter;

architecture impl of shifter is
  signal tmp : std_logic_vector(7 downto 0);
begin
  process (Clk)
  begin
    if (Clk'event and Clk = '1') then
      for i in 0 to 6 loop -- unrolled at compile time
        tmp(i+1) <= tmp(i);
      end loop;
      tmp(0) <= SI;
    end if;
  end process;

  SO <= tmp(7); -- Copy to output
end impl;
```

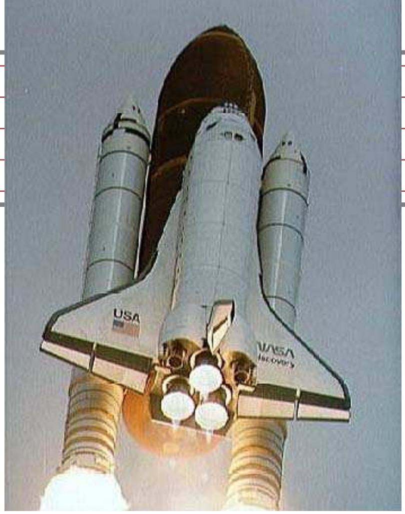
A small RAM

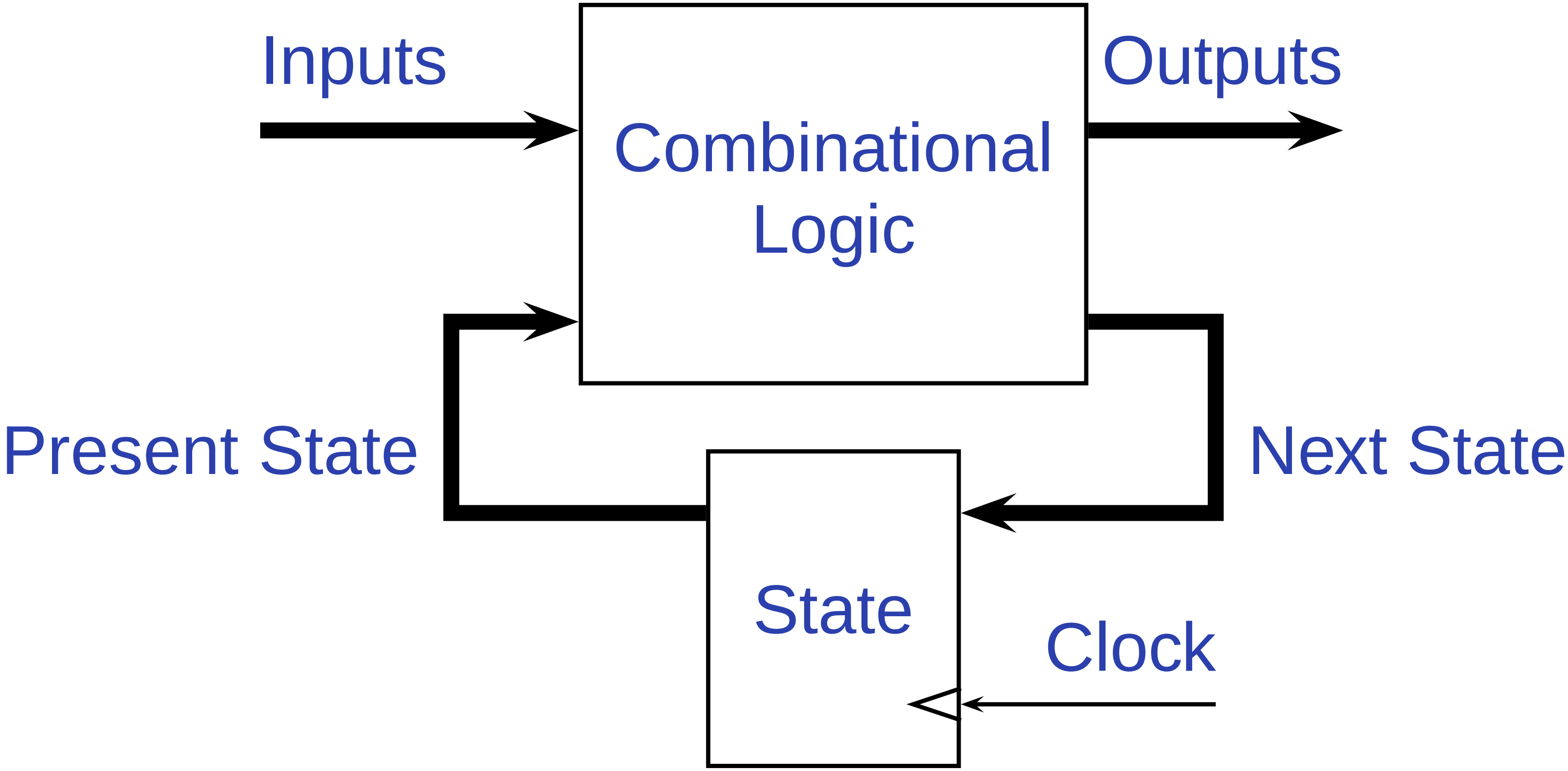
```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity ram_32_4 is
  port (
    Clk  : in  std_logic;
    WE   : in  std_logic;                      -- Write enable
    EN   : in  std_logic;                      -- Read enable
    addr : in  std_logic_vector(4 downto 0);
    di   : in  std_logic_vector(3 downto 0);   -- Data in
    do   : out std_logic_vector(3 downto 0));  -- Data out
end ram_32_4;

architecture imp of ram_32_4 is
  type ram_type is array(31 downto 0) of std_logic_vector(3 downto 0);
  signal RAM : ram_type;
begin
process (Clk)
begin
  if (Clk'event and Clk = '1') then
    if (en = '1') then
      if (we = '1') then
        RAM(conv_integer(addr)) <= di;
        do <= di;
      else
        do <= RAM(conv_integer(addr));
      end if;
    end if;
  end if;
end process;
end imp;
```

A small ROM

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

entity rom_32_4 is
  port (
    Clk  : in  std_logic;
    en   : in  std_logic;                  -- Read enable
    addr : in  std_logic_vector(4 downto 0);
    data : out std_logic_vector(3 downto 0));
end rom_32_4;

architecture imp of rom_32_4 is
type rom_type is array (31 downto 0) of std_logic_vector(3 downto 0);
constant ROM : rom_type :=
  ("0001", "0010", "0011", "0100", "0101", "0110", "0111", "1000",
   "1001", "1010", "1011", "1100", "1101", "1110", "1111", "0001",
   "0010", "0011", "0100", "0101", "0110", "0111", "1000", "1001",
   "1010", "1011", "1100", "1101", "1110", "1111", "0000", "0010");
begin

process (Clk)
begin
  if (Clk'event and Clk = '1') then
    if (en = '1') then
      data <= ROM(conv_integer(addr));
    end if;
  end if;
end process;
end imp;
```

Rocket Science: FSMs

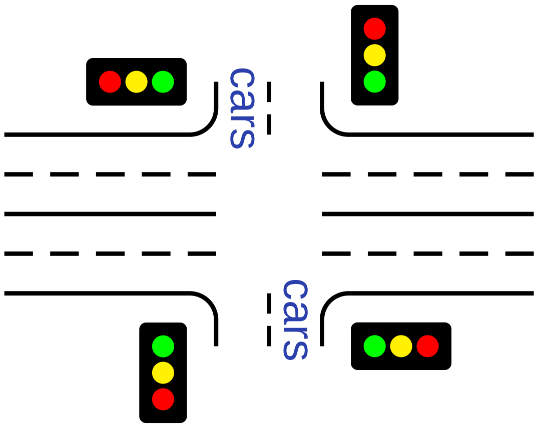




Structure of FSMs in VHDL

```
entity myFSM is
 port( ... );
end myFSM;

architecture imp of myFSM is
  constant STATE1 := "...";
  constant STATE2 := "...";
  signal current_state, next_state : ...;

process (clk)         -- State holding element process
begin
  if (clk'event and clk = '1') then
    current_state <= next_state;
  end if
end process;

process (inputs...)   -- Outputs and next state function
begin
  if (reset = '1') then
    next_state <= STATE1;
  else
    case current_state is
      when STATE1 =>
        output1 <= '1';
        next_state <= STATE2;
      when STATE2 =>
        ...
        next_state <= STATE3;
     end case;
  end if;
end process;

end imp;
```

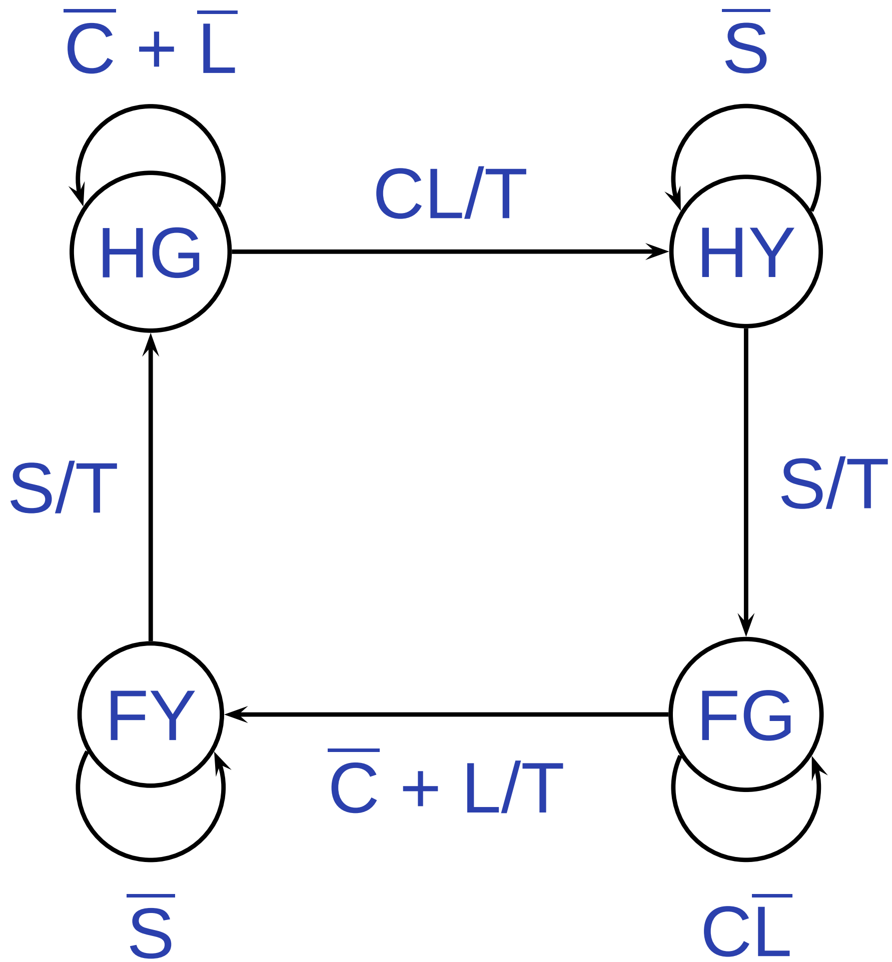
The Traffic Light Controller



This controls a traffic light at the intersection of a busy highway and a farm road. Normally, the highway light is green but if a sensor detects a car on the farm road, the highway light turns yellow then red. The farm road light then turns green until there are no cars or after a long timeout. Then, the farm road light turns yellow then red, and the highway light returns to green. The inputs to the machine are the car sensor, a short timeout signal, and a long timeout signal. The outputs are a timer start signal and the colors of the highway and farm road lights.

Source: Mead and Conway, *Introduction to VLSI Systems*, 1980, p. 85.

FSM for the Traffic Light Controller



C: Car sensor
S: Short timeout
L: Long timeout

T: Start timer

St	**Hwy**	**Farm**
HG	G	R
HY	Y	R
FG	R	G
FY	R	Y

Traffic Light Controller in VHDL (1)

```
library ieee;
use ieee.std_logic_1164.all;

entity tlc is
  port (
    clk            : in  std_ulogic;
    reset          : in  std_ulogic;
    cars           : in  std_ulogic;
    short          : in  std_ulogic;
    long           : in  std_ulogic;
    highway_yellow : out std_ulogic;
    highway_red    : out std_ulogic;
    farm_yellow    : out std_ulogic;
    farm_red       : out std_ulogic;
    start_timer    : out std_ulogic);
end tlc;
```

Traffic Light Controller in VHDL (2)

```
architecture imp of tlc is
signal current_state, next_state : std_ulogic_vector(1 downto 0);
constant HG : std_ulogic_vector := "00";
constant HY : std_ulogic_vector := "01";
constant FY : std_ulogic_vector := "10";
constant FG : std_ulogic_vector := "11";
begin

P1: process (clk)    -- Sequential process
begin
  if (clk'event and clk = '1') then
    current_state <= next_state;
  end if;
end process P1;
```

Traffic Light Controller in VHDL (3)

```
-- Combinational process
--   Sensitive to input changes, not clock

P2: process (current_state, reset, cars, short, long)
begin
  if (reset = '1') then
    next_state <= HG;
    start_timer <= '1';
  else
    case current_state is
      when HG =>
        highway_yellow <= '0';
        highway_red    <= '0';
        farm_yellow    <= '0';
        farm_red       <= '1';
        if (cars = '1' and long = '1') then
          next_state  <= HY;
          start_timer <= '1';
        else
          next_state  <= HG;
          start_timer <= '0';
        end if;
```

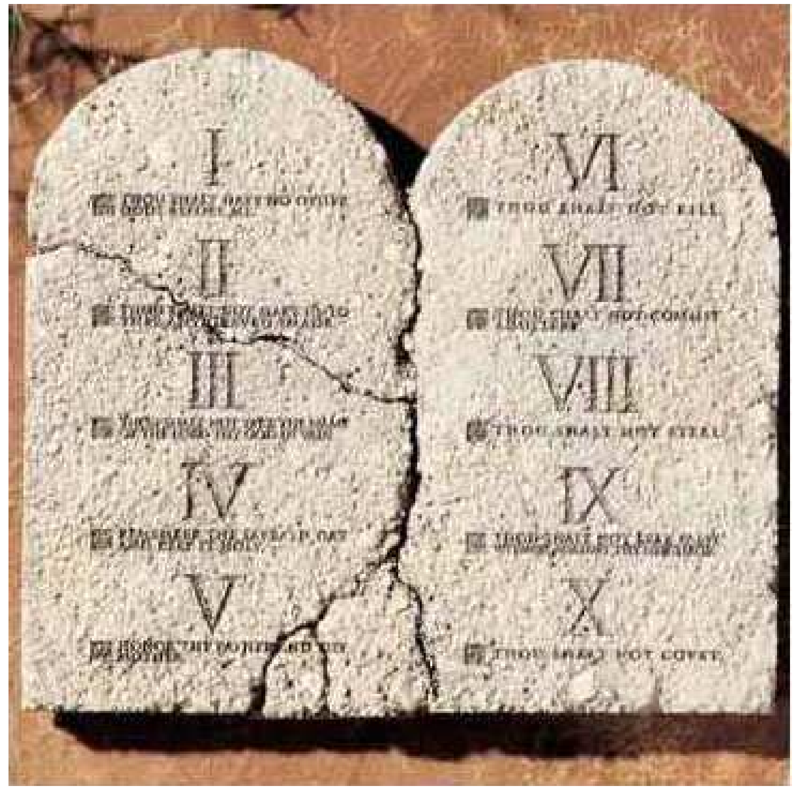
Traffic Light Controller in VHDL (4)

```
when HY =>
  highway_yellow <= '1';
  highway_red    <= '0';
  farm_yellow    <= '0';
  farm_red       <= '1';
  if (short = '1') then
    next_state  <= FG;
    start_timer <= '1';
  else
    next_state  <= HY;
    start_timer <= '0';
  end if;

when FG =>
  highway_yellow <= '0';
  highway_red    <= '1';
  farm_yellow    <= '0';
  farm_red       <= '0';
  if (cars = '0' or long = '1') then
    next_state  <= FY;
    start_timer <= '1';
  else
    next_state  <= FG;
    start_timer <= '0';
  end if;
```

Traffic Light Controller in VHDL (5)

```
      when FY =>
        highway_yellow <= '0';
        highway_red    <= '1';
        farm_yellow    <= '1';
        farm_red       <= '0';
        if (short = '1') then
          next_state  <= HG;
          start_timer <= '1';
        else
          next_state  <= FY;
          start_timer <= '0';
        end if;

      when others =>
        next_state <= "XX";
        start_timer <= 'X';
        highway_yellow <= 'X';
        highway_red    <= 'X';
        farm_yellow    <= 'X';
        farm_red       <= 'X';
    end case;
  end if;
end process P2;

end imp;
```

Ten Commandments of VHDL



I: Thou Shalt Design Before Coding

- Know the structure of what you are designing first.
- Draw a block diagram of the datapath
- Understand the timing (draw diagrams)
- Draw bubble-and-arc diagrams for FSMs
- Only once you have a design should you start coding in VHDL
- VHDL is only a way to ask for component

II: Thou Shalt be Synchronous

- One global clock
- Flip-flops generate inputs to combinational logic, which computes inputs to flip-flops
- Exactly one value per signal per clock cycle
- Do not generate asynchronous reset signals; only use them if they are external
- Edge-triggered flip-flops only. Do not use level-sensitive logic.
- Do not generate clock signals. Use multiplexers to create “load enable” signals on flip-flops.

III: Thou Shalt Be Sensitive

Combinational processes: list all process inputs

```
process (current_state, long)
begin
  if (reset = '1') then
    next_state <= HG;
    start_timer <= '1';
  else
    case current_state is
      when HG =>
        farm_yellow    <= '0';
        if (cars = '1' and long = '1') then
          next_state  <= HY;
        else
          next_state  <= HG;
        end if;
      when HY =>
        farm_yellow    <= '0';
        if (short = '1') then
          next_state  <= FG;
        else
          next_state  <= HY;
        end if;
```

```
process (current_state, reset, cars, short, long)
begin
  if (reset = '1') then
    next_state <= HG;
    start_timer <= '1';
  else
    case current_state is
      when HG =>
        farm_yellow    <= '0';
        if (cars = '1' and long = '1') then
          next_state  <= HY;
        else
          next_state  <= HG;
        end if;
      when HY =>
        farm_yellow    <= '0';
        if (short = '1') then
          next_state  <= FG;
        else
          next_state  <= HY;
        end if;
```

III: Thou Shalt Be Sensitive

Sequential processes: always include the clock. Include reset if asynchronous, and nothing else.

```
process (Clk, D)
begin
  if (Clk'event and Clk = '1') then
    Q <= D;
  end if;
end process;
```

```
process (Clk, D)
begin
  if (reset = '1') then
    Q <= '0';
  else
    if (Clk'event and Clk = '1') then
      Q <= D;
    end if;
  end if;
end process;
```

```
process (Clk)
begin
  if (Clk'event and Clk = '1') then
    Q <= D;
  end if;
end process;
```

```
process (Clk, reset)
begin
  if (reset = '1') then
    Q <= '0';
  else
    if (Clk'event and Clk = '1') then
      Q <= D;
    end if;
  end if;
end process;
```

IV: Thou Shalt Assign All Outputs

Synthesis infers level-sensitive latches if sometimes you do not assign an output.

```
process (current_state, input)
begin
  case current_state is
    when S1 =>
       if (input = '1') then
         output <= '0';
       end if;
    when S2 =>
      output <= '1';
  end case;
end process;
```

```
process (current_state, input)
begin
  case current_state is
    when S1 =>
       if (input = '1') then
         output <= '0';
       else
         output <= '1';
       end if;
    when S2 =>
      output <= '1';
  end case;
end process;
```

“Default” values are convenient

```
-- OK

process (current_state, input)
begin
  case current_state is
    when S1 =>
       if (input = '1') then
         output <= '0';
       else
         output <= '1';
       end if;
    when S2 =>
      output <= '1';
  end case;
end process;
```

```
-- Better

process (current_state, input)
begin
  output <= '1';
  case current_state is
    when S1 =>
       if (input = '1') then
         output <= '0';
       end if;
  end case;
end process;
```

V: Thou Shalt Enumerate States

Better to use an enumeration to encode states:

```
type mystate is (START,RUN,IDLE,ZAPHOD);
signal cst : mystate;
signal nxst : mystate;

process(cst)
 begin
   case cst is
     when START => ...
     when RUN => ...
     when IDLE => ...
   end case;
 end process;
```

Running this produces a helpful error:

```
Compiling vhdl file "/home/cristi/cs4840/lab4/main.vhd" in Library work.
Entity <system> compiled.
ERROR:HDLParsers:813 - "/home/cristi/cs4840/lab4/main.vhd" Line 80.
Enumerated value zaphod is missing in case.
-->
```

VI:

(There is no rule six)

VII: Thou Shalt Avoid Async

Only use asynchronous reset when there is one global signal from outside.

```
-- OK if Reset is from outside

process (Clk, Reset)
begin
  if (Reset = '1') then
    Q <= '0';
  else
    if (Clk'event and Clk = '1') then
      Q <= D;
    end if;
  end if;
end process;
```

```
-- Better

process (Clk)
begin
  if (Clk'event and Clk = '1') then
    if (Reset = '1') then
      Q <= '0';
    else
      Q <= D;
    end if;
  end if;
end process;
```

VIII: Thou Shalt Have One Version

- Never assume signals from the test bench that are not there on the board
- It is hard enough to make simulation match the design; do not make it any harder
- If you must slow down hardware, carefully generate a slower clock and only use that clock globally.

IX: Thou Shalt Not Test For X Or Z

```
architecture behv of ALU is begin
  process(A,B,Sel) begin
    case Sel is
      when "00" => Res <= A + B;
      when "01" => Res <= A + (not B) + 1;
      when "1X" => Res <= A and B;
      when "1Z" => Res <= A or B;
      when others => Res <= "XX";
    end case;
  end process;
end behv;
```

```
architecture behv of ALU is begin
  process(A,B,Sel) begin
    case Sel is
      when "00" => Res <= A + B;
      when "01" => Res <= A + (not B) + 1;
      when "10" => Res <= A and B;
      when "11" => Res <= A or B;
      when others => Res <= "XX";
    end case;
  end process;
end behv;
```

X: Thou Shalt Not Specify Delays

- The `wait` statement can delay for a certain amount of time, e.g., “`wait 10ns;`”
- It is good to use in test benches that are not meant to become hardware
- Do not use them in the design of your hardware