TSTE12 Design of Digital Systems Lecture 8

Kent Palmkvist

LINKÖPING UNIVERSITY

Agenda

- Practical issues
- Hardware description
  - FPGA
- HDL based design

TSTE12 Deadlines Y,D,ED

- Final version of design sketch and project plan this week
  - Show implementation ideas, show sequence of implementation and task partitioning between group members
- Weekly meetings should start
  - Internal weekly meeting with transcript sent to supervisor
- Lab 2 results will be checked after the project ends

TSTE12 Deadlines MELE, erasmus

- Final requirement specification this week
- Wednesday 21 September 21.00: Lab 2 soft deadline
  - Lab 2 results will be checked after project completed

Handin (homework), Individual!

- 1st handin deadline Monday 19 September 23:30
- Use only plan text editor (emacs, vi, modelsim or similar) for code entry.
- Solve tasks INDIVIDUALLY
- Submit answers using Lisam assignment function
  - 4 different submissions for code, one for each code task
  - 1 submission for all theory question answers
- Use a special terminal window when working with handins

  module load TSTE12 ; TSTE12handin

Hardware overview

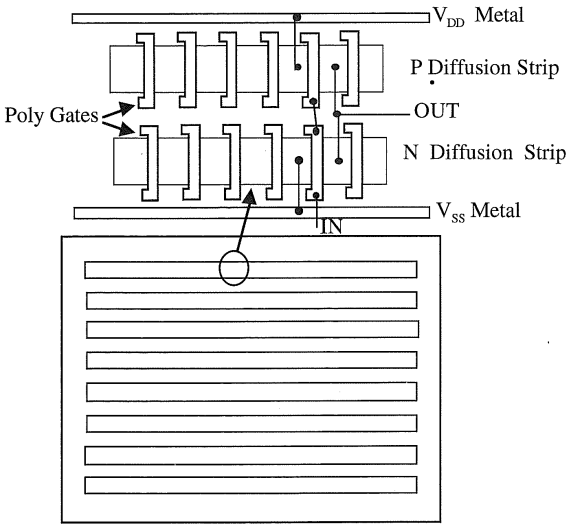
- Detailed description of Hardware
- Standard parts
  - TTL (SSI, MSI LSI)
  - Memories, microprocessors, I/O
- ASIC (Application Specific Integrated Circuit)
  - Integrated circuit that has been produced for a specific application and (often) produced in small numbers
  - Memories and microprocessors are general application devices

ASIC technologies

- May use different technologies for ASIC: PLD, Gate array, FPGA, Standard cell, custom. ASIC is however limited to Standard cell and gate array. Custom design is also used.
- CMOS switch. Power consumption: $P \sim CV^2f$
  - Use low power supply, reduce clock, reduce area
- Transistor channel length (old measure of chip manufacturing process) shorter than 0.01 um (so called 5 nm used today, e.g. TSMC N5 process in Ryzen 7000 series CPUs, 4 nm in Apple A16 Bionic in Iphone 14)

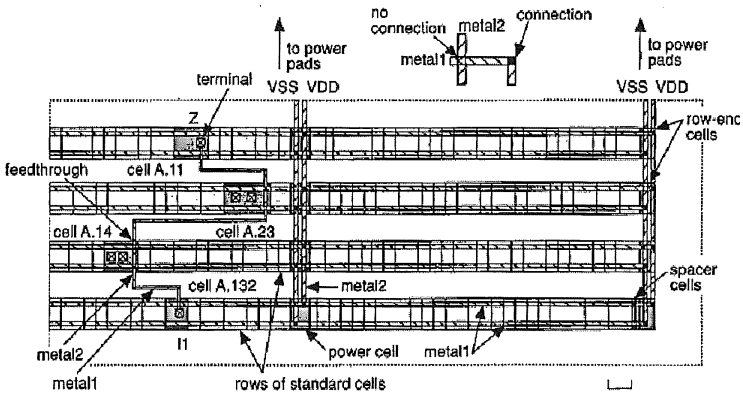
Gate array, mask programmable

- Predefined pattern of transistors
  - Add interconnect metal for each design
  - Fast manufacture (weeks)
  - No transistor sizing
- Example shows inverter design
- Combined with library of existing cells
  - Basic gates, flipflops etc.



Standard cell

- Transistor placement and metal layers unique for each design, needs to be manufactured
- Limited number of layout cell types (Cell library)
- Cells already characterized
- Slow manufacture (month)

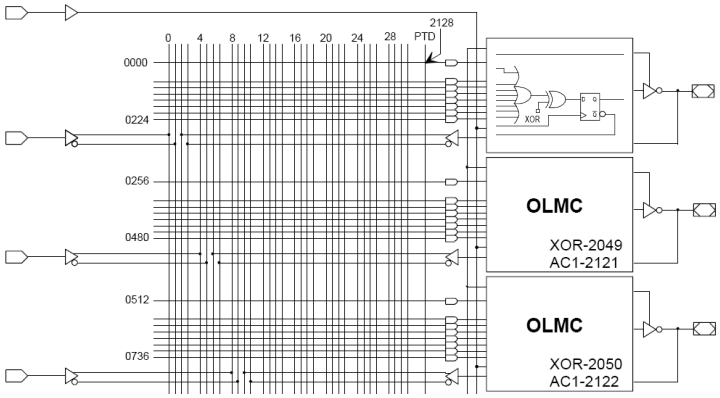


Full custom layout

- Full custom
  - Individual placement and scaling of transistors
  - Full control of wires and connections
  - Maximum control, maximum effort
- Complete freedom to place and route transistors
  - Not limited to existing logic style/library
  - Slow manufacture (months)
  - Higher performance than standard cells
- Requires more testing (simulation)

Programmable devices: PLD

- Input variables forming AND-OR arrays, flipflops at the outputs.
  - Crosspoint can connect to create products
- Inputs and outputs on chip edge
- Only small circuit designs



Programmable devices: CPLD

- Combine PLD structure with additional onchip interconnect
  - Support larger designs

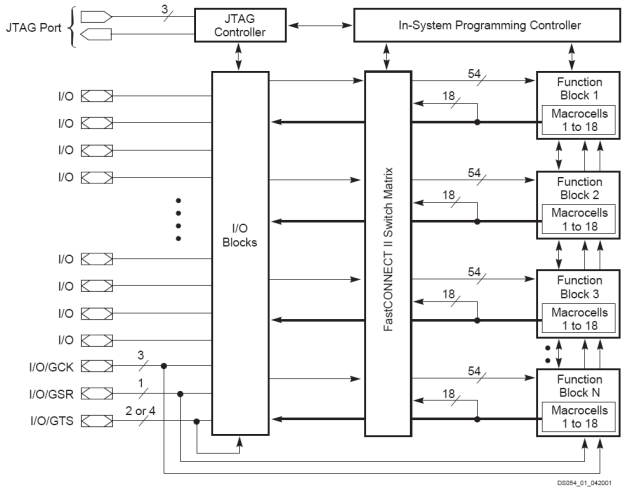
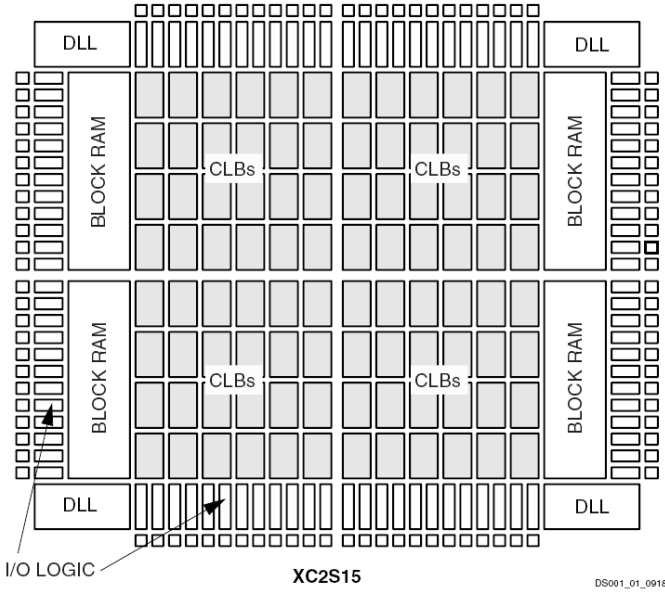


Figure 1: **XC9500XL Architecture**

FPGA structure

- Field Programmable Gate Array
- Cells in an array, special I/O blocks around the edges.
- Between cells (CLBs or LEs) are routing wires located (interconnect)

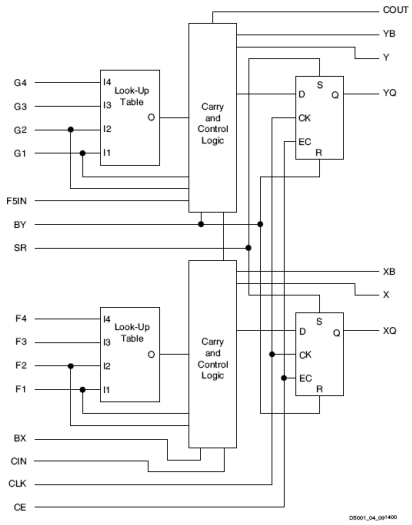


FPGA building blocks

- CLB/Logic Element
  - Different name in different manufacturers designs
  - In many cases are they based on lookup tables (i.e., no simple gates, instead more advanced functions) => less need for routing channels (that are expensive). Lookup table can be viewed as a small RAM or a MUX with fixed inputs.
  - Trade off between big lookup tables and utilization. Optimal around 4-6 bits address.
  - Often a flip-flop included in the CLB/LE

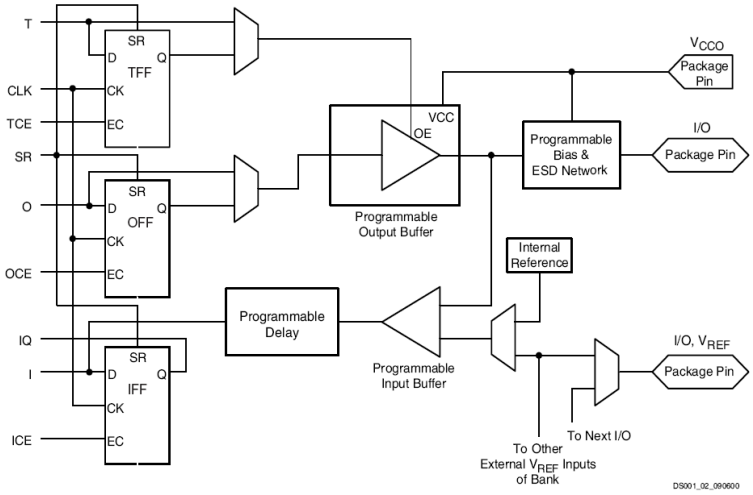
CLB Example: Xilinx (AMD) Spartan II

- Choose positive or negative clock edge
- May combine lookup tables
- CLB may be rearranged into a memory or shift register



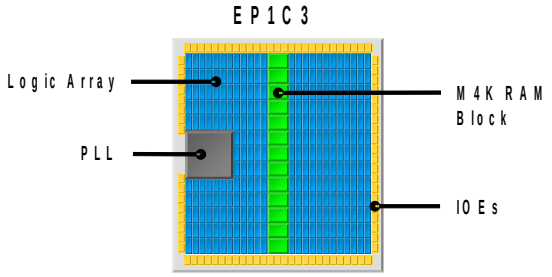
Xilinx Spartan II I/O logic

- Support multiple I/O standards
  - 3.3V, 1.8V, 1.2V etc.
  - Differential
- Flipflops located close to pin
  - Reduce delays due to routing signal to pin
- Different drive strength



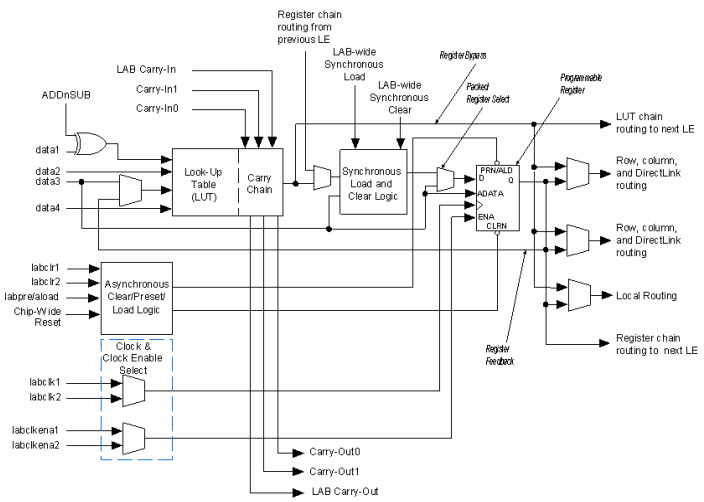
Other FPGA example: Altera (intel) Cyclone

- Same basic structure
  - Logic Array
  - Separate I/O blocks
  - Dedicated memory
  - PLL for internal clock generation



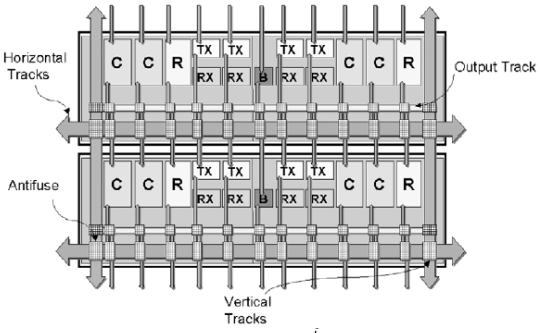
Altera (Intel) Cyclone

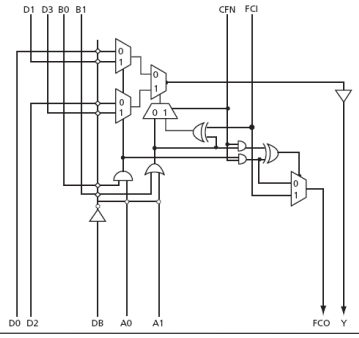
- Logic element structure
  - More detailed description
  - One Lookup table + one flipflop

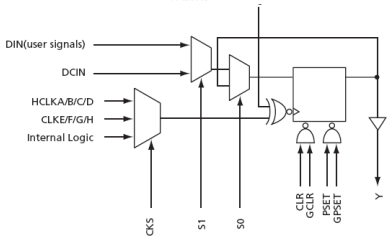


Alternative FPGA structure: Actel (Microsemi)

- Smaller functions
- Separate register from combination logic

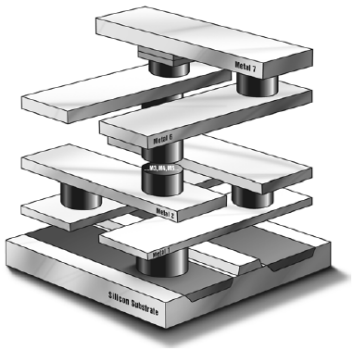






Programming of FPGA

- Two types: reprogrammable or one-time programmable
- Control a CMOS-switch using a RAM/EPROM/EEPROM-cell. The CMOS switch is slow (compared to the alternative)
- The alternative is fuse/Antifuse (burn together two wires by using high voltage)



Important aspects

- Speed of the switched (impedance, capacitans). Many switches in series ruins the performance
- Reprogrammable? Needs any design changes to be done?
- Volatile designs? What happens at power failure? How is the design put into the chip? How long delay from power on to working design?
- Area of the switches? Needs many switches?

Technology comparison table

- Most common
  - SRAM
  - EEPROM/FLASH
- Xilinx/Altera
  - SRAM/EEPROM
- Actel
  - Antifuse
  - Do also create classic FPGA (SRAM/EEPROM based)
- SRAM based FPGA usually support automatic configuration from serial flash memory at power-on

	SRAM	Antifuse	EPROM	EEPROM/FLASH
Volatile	yes	no	no	no
Re-programmable	yes	no	yes	yes
Chip area	large	small	small	average
R (routing nets)	large	small	large	large
C	large	small	large	large

How to configure the FPGA

- Non-volatile technology
  - FLASH, EEPROM, PROM, etc.
- External programmer
  - Software on PC to program device
- External ROM/FLASH
  - Standard FLASH
  - Serial FLASH
- Embedded microcontroller
  - Boot application configures FPGA
  - Not possible if flash needed for CPU operation

FPGA configuration, cont.

- Large volumes may use non-programmable devices based on FPGA
  - Resynthesize: may give different behavior
  - Strip FPGA: Remove configuration logic

FPGA hardware options

- Multipliers
- DSP blocks
  - Multiply-ackumulate
  - Common operation in DSP
  - High precision (> 20 bits)
- Optimized I/O support
  - Differential signaling
  - Low swing/current steering

FPGA hardware options, cont.

- Clock circuits
  - Phase locked loops (PLL), Delay locked loops (DLL)
  - Clock tree distribution
- Serializer/deserializer
  - Support modern PC bus standards such as PCI Express
  - Dedicated block to send/recieve high speed (> Gbit/s) serial data
  - Reduce number of I/O pins

FPGA hardware options, cont.

- A/D and D/A converters
- Memory units
- CPU
  - e.g., physical powerpc or ARM core inside FPGA
  - Usually combined with external memory interfaces and CPU-based I/O support (e.g. wired ethernet, SD-card reader etc.)
- Alternative to dedicated CPU hardware: soft cpu
  - VHDL design of a processor
  - Allows for modification of processor structure

ASIC vs FPGA

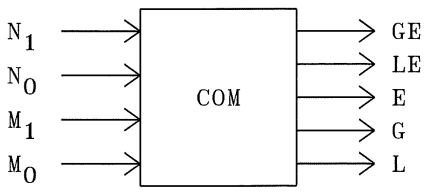
- ASIC have a large NRE cost
  - Non-Recurring Engineering cost, price of 1st unit
- FPGA have large per unit cost
- Selection of technology depend on
  - Performance requirements
  - Number of units
  - Time to market

HDL based design

- Structured design using HDL
- FSM descriptions

Example: Combination logic

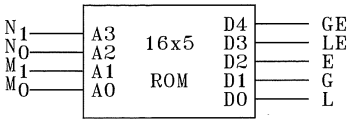
- Binary comparison. Compare two two-bit words
  - GE: Greater or equal
  - LE: Less or equal
  - E: Equal
  - G: Greater
  - L: Less




```
entity COM is
  generic (D:time);
  port (N1, N0, M1, M0: in BIT;
        GE, LE, E, G, L: out BIT);
end COM;
```

Descriptions

- ROM
  - Table lookup



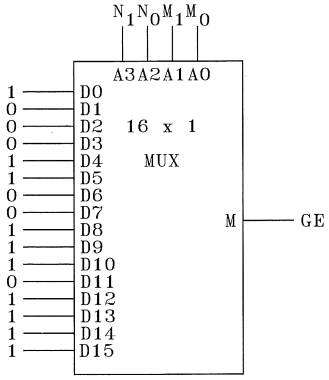

```
use work.TRUTH4x5.all;
architecture TABLE of COM is
begin
  process (N1,N0,M1,M0)
    variable INDEX: INTEGER;
    variable WOUT: WORD;
  begin
    INDEX := INTVAL (N1&N0&M1&M0);
    WOUT := TRUTH (INDEX);
    GE <= WOUT(4) after D;
    LE <= WOUT(3) after D;
    E  <= WOUT(2) after D;
    G  <= WOUT(1) after D;
    L  <= WOUT(0) after D;
  end process;
end TABLE;
```

```
package TRUTH4x5 is
  constant NUM_OUTPUTS: INTEGER:=5;
  constant NUM_INPUTS: INTEGER:=4;
  constant NUM_ROWS: INTEGER:= 2 ** NUM_INPUTS;
  type WORD is array(NUM_OUTPUTS-1 downto 0) of BIT;
  type ADDR is array(NUM_INPUTS-1 downto 0) of BIT;
  type MEM is array (0 to NUM_ROWS-1) of WORD;
  constant TRUTH: MEM :=
              ("11100", "01001", "01001", "01001",
               "10010", "11100", "01001", "01001",
               "10010", "10010", "11100", "01001",
               "10010", "10010", "10010", "11100");
  function INTVAL(VAL:ADDR) return INTEGER;
end TRUTH4x5;

package body TRUTH4x5 is
  function INTVAL(VAL: ADDR) return INTEGER is
    variable SUM: INTEGER:=0;
  begin
    for N in VAL'LOW to VAL'HIGH loop
      if VAL(N) = '1' then
        SUM := SUM + (2 ** N);
      end if;
    end loop;
    return SUM;
  end INTVAL;
end TRUTH4x5;
```


Descriptions, CASE statement

- One multiplexer for each output




```
architecture MUX of COM is
begin
  process(N1,N0,M1,M0)
  begin
    case N1&N0&M1&M0 is
      when "0000" => GE <= '1' after D; LE <= '1' after D;
        E <= '1' after D; G <= '0' after D; L <= '0' after D;
      when "0001" => GE <= '0' after D; LE <= '1' after D;
        E <= '0' after D; G <= '0' after D; L <= '1' after D;
      when "0010" => GE <= '0' after D; LE <= '1' after D;
        E <= '0' after D; G <= '0' after D; L <= '1' after D;
      when "0011" => GE <= '0' after D; LE <= '1' after D;
        E <= '0' after D; G <= '0' after D; L <= '1' after D;
      when "0100" => GE <= '1' after D; LE <= '0' after D;
        E <= '0' after D; G <= '1' after D; L <= '0' after D;
      when "0101" => GE <= '1' after D; LE <= '1' after D;
        E <= '1' after D; G <= '0' after D; L <= '0' after D;
      when "0110" => GE <= '0' after D; LE <= '1' after D;
        E <= '0' after D; G <= '0' after D; L <= '1' after D;
      when "0111" => GE <= '0' after D; LE <= '1' after D;
        E <= '0' after D; G <= '0' after D; L <= '1' after D;
      when "1000" => GE <= '1' after D; LE <= '0' after D;
        E <= '0' after D; G <= '1' after D; L <= '0' after D;
      when "1001" => GE <= '1' after D; LE <= '0' after D;
        E <= '0' after D; G <= '1' after D; L <= '0' after D;
      when "1010" => GE <= '1' after D; LE <= '1' after D;
        E <= '1' after D; G <= '0' after D; L <= '0' after D;
      when "1011" => GE <= '0' after D; LE <= '1' after D;
        E <= '0' after D; G <= '0' after D; L <= '1' after D;
      when "1100" => GE <= '1' after D; LE <= '0' after D;
        E <= '0' after D; G <= '1' after D; L <= '0' after D;
      when "1101" => GE <= '1' after D; LE <= '0' after D;
        E <= '0' after D; G <= '1' after D; L <= '0' after D;
      when "1110" => GE <= '1' after D; LE <= '0' after D;
        E <= '0' after D; G <= '1' after D; L <= '0' after D;
      when "1111" => GE <= '1' after D; LE <= '1' after D;
        E <= '1' after D; G <= '0' after D; L <= '0' after D;
    end case;
  end process;
end MUX;
```

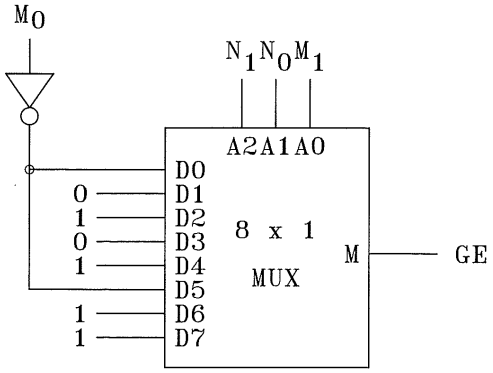
Descriptions, improved CASE

- Remove one variable in the selection of the case statement
- Use the removed variable as output value or its inverse
- More variables can be removed
  - Increase logic in front of multiplexer

```
architecture MUX3 of COM is
begin
 process (N1, N0, M1, M0)
 begin
  case N1&N0&M1 is
   when "000" => GE <= not M0 after D; LE <= '1' after D;
    E <= not M0 after D; G <= '0' after D; L <= M0 after D;
   when "001" => GE <= '0' after D; LE <= '1' after D;
    E <= '0' after D; G <= '0' after D; L <= '1' after D;
   when "010" => GE <= '1' after D; LE <= M0 after D;
    E <= M0 after D; G <= not M0 after D; L <= '0' after D;
   when "011" => GE <= '0' after D; LE <= '1' after D;
    E <= '0' after D; G <= '0' after D; L <= '1' after D;
   when "100" => GE <= '1' after D; LE <= '0' after D;
    E <= '0' after D; G <= '1' after D; L <= '0' after D;
   when "101" => GE <= not M0 after D; LE <= '1' after D;
    E <= not M0 after D; G <= '0' after D; L <= M0 after D;
   when "110" => GE <= '1' after D; LE <= '0' after D;
    E <= '0' after D; G <= '1' after D; L <= '0' after D;
   when "111" => GE <= '1' after D; LE <= M0 after D;
    E <= M0 after D; G <= not M0 after D; L <= '0' after D;
  end case;
 end process;
end MUX3;
```


Hardware, improved Case statement

- One mux plus inverter
- Every output have its own multiplexer (same as for non-improved case statement)



Partitioning

- Rewrite expressions, sharing common subexpression
  - E= GE AND LE
  - G = GE AND NOT LE
  - L = LE AND NOT GE
- That is, two expressions followed by simple generation of E, G, and L
- Designer makes logic synthesis instead of tool
  - Synthesis tool may still modify description

Two-level logic

- Many different choices
- Can be described as structure

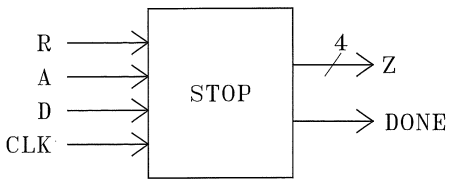
```
architecture POSDF of COM is
  signal Z1,Z0: BIT;
begin
  Z1 <= (not N0 or M1 or M0) and (not N1 or M1) and
        (not N1 or not N0 or M0);
  Z0 <= (N1 or N0 or not M0) and (N1 or not M1) and
        (N0 or not M1 or not M0);
  LE <= Z1 after D;
  GE <= Z0 after D;
   E <= Z1 and Z0 after D;
   G <= Z0 and not Z1 after D;
   L <= Z1 and not Z0 after D;
end POSDF;
```

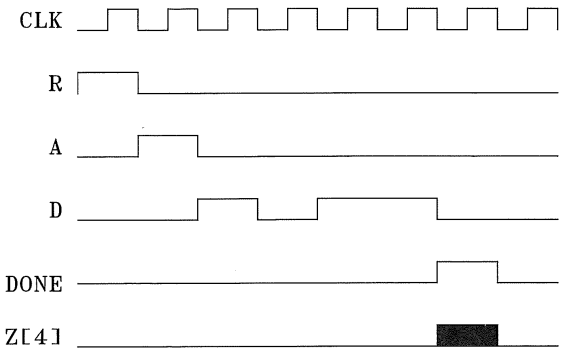
Structural description

```
architecture TWO_LEVEL_OR_AND of COM is
  signal Z10,Z11,Z12,Z00,Z01,Z02: BIT;
  signal N0BAR,N1BAR,M0BAR,M1BAR: BIT;
  signal Z0,Z1,Z0NOT,Z1NOT: BIT;
  component NOT2G
    generic (D: TIME);
    port (I: in BIT; O: out BIT);
  end component;
  for all: NOT2G use entity NOT2(BEHAVIOR);
  component AND2G
    generic (D: TIME);
    port (I1, I2: in BIT; O: out BIT);
  end component;
  for all: AND2G use entity AND2(BEHAVIOR);
  component AND3G
    generic (D: TIME);
    port(I1,I2,I3: in BIT; O: out BIT);
  end component;
  for all: AND3G use entity AND3(BEHAVIOR);
  component OR2G
    generic (D: TIME);
    port(I1,I2: in BIT; O: out BIT);
  end component;
  for all: OR2G use entity OR2(BEHAVIOR);
  component OR3G
    generic (D: TIME);
    port (I1,I2,I3: in BIT; O: out BIT);
  end component;
  for all: OR3G use entity OR3(BEHAVIOR);
  component WIREG
    port (I: in BIT; O: out BIT);
    end component;
  for all: WIREG use entity WIRE(BEHAVIOR);
begin
C1: NOT2G
    generic map (2 ns)
    port map (N0, N0BAR);
 C2: NOT2G
    generic map (2 ns)
    port map (N1, N1BAR);
 C3: NOT2G
    generic map (2 ns)
    port map (M0, M0BAR);
 C4: NOT2G
    generic map (2 ns)
    port map (M1, M1BAR);
 C5: OR3G
    generic map (2 ns)
    port map (N0BAR, M1, M0, Z10);
 C6: OR2G
    generic map (2 ns)
    port map (N1BAR, M1, Z11);
 C7: OR3G
    generic map (2 ns)
    port map (N1BAR, N0BAR, M0, Z12);
 C8: AND3G
    generic map (2 ns)
    port map (Z10, Z11, Z12, Z1);
 C9: OR3G
    generic map (2 ns)
    port map (N1, N0, M0BAR, Z00);
C10:OR2G
    generic map (2 ns)
    port map (N1, M1BAR, Z01);
C11:OR3G
    generic map (2 ns)
    port map (N0, M1BAR, M0BAR, Z02);
C12:AND3G
    generic map (2 ns)
    port map (Z00, Z01, Z02, Z0);
C13:NOT2G
    generic map (2 ns)
    port map (Z1, Z1NOT);
C14:NOT2G
    generic map (2 ns)
    port map (Z0, Z0NOT);
C15:AND2G
    generic map (2 ns)
    port map (Z0, Z1, E);
C16:AND2G
    generic map (2 ns)
    port map (Z0, Z1NOT, G);
C17:AND2G
    generic map (2 ns)
    port map (Z1, Z0NOT, L);
C18:WIREG
    port map (Z0, GE);
C19: WIREG
    port map (Z1, LE);
end TWO_LEVEL_OR_AND;
```


Finite state machines (FSM)

- Example: serial/parallel converter
  - A indicates start of data
  - Output Z only during one clock cycle






```
entity STOP is
  port (R, A, D, CLK: in BIT;
        Z: out BIT_VECTOR(3 downto 0);
        DONE: out BIT);
end STOP;
```

FSM design, cont.

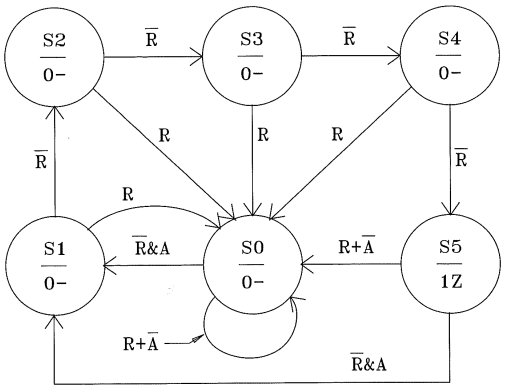
- First: Select type of state machine (Moore, Mealy)
  - Moore machine have stable output after a few gate delays
  - Moore machine can not produce output dependent on current input values
  - Moore machine may require more states than Mealy machines
  - Mealy machine may sometimes be required due to direct respons from FSM on input signal change

FSM Design, cont.

- Second: Create a state diagram. Good start is reset-state S0.
  - S1: First data on D, Done=0, Z unspecified
  - S2: Second data on D, Done =0, Z unspecified
  - S3: Third data on D, Done = 0, Z unspecified
  - S4: Fourth data on D, Done = 0, Z unspecified
  - S5: Output on Z, Done= 1
  - In S5 can A also be 1 (indicating new data)
    - Next clock cycle must take care data, i.e., use S1 without passing through S0

FSM state diagram

- Some tools can translate state diagram automatically to VHDL (e.g., HDL Designer)



Alternate desription

- Transition list
  - Textual description of the FSM
  - Useful for large state diagrams
  - Graphs become hard to understand when number of states increase
  - Possible to cope with complexity by use of hierarchy

Current State	Transition Expression	Next State	Data Transfers	Output
S0	R+Ā	S0	None	DONE=0, Z unspecified
S0	R̄ & A	S1		
S1	R̄	S2	Store bit 1	DONE=0, Z unspecified
S1	R	S0		
:	:	:	:	:
S5	R̄ & A	S1	None	DONE=1, Z=parallel data out
S5	R + Ā	S0		

FSM description in VHDL

```
architecture FSM_RTL of STOP is
  type STATE_TYPE is (S0, S1, S2, S3, S4, S5);
  signal STATE: STATE_TYPE;
  signal SHIFT_REG: BIT_VECTOR (3 downto 0);
begin

STATE: process (CLK)
  begin
    if CLK='1' then
      case STATE is
        when S0 =>
          -- Data Section
          -- Control Section
            if R='1' or A='0' then
              STATE <= S0;
            elsif R='0' and A='1' then
              STATE <= S1;
            end if;
        when S1 =>
          -- Data Section
            SHIFT_REG <= D & SHIFT_REG(3 downto 1);
          -- Control Section
            if R='0' then
              STATE <= S2;
            elsif R='1' then
              STATE <= S0;
            end if;
```

```
when S2 =>
  -- Data Section
    SHIFT_REG <= D & SHIFT_REG(3 downto 1);
  -- Control Section
    if R='0' then
      STATE <= S3;
    elsif R='1' then
      STATE <= S0;
    end if;
when S3 =>
  -- Data Section
  -- Shift in the third bit
    SHIFT_REG <= D & SHIFT_REG(3 downto 1);
  -- Control Section
    if R='0' then
      STATE <= S4;
    elsif R='1' then
      STATE <= S0;
    end if;
when S4 =>
  -- Data Section
  -- Shift in the fourth bit
    SHIFT_REG <= D & SHIFT_REG(3 downto 1);
  -- Control Section
    if R='0' then
      STATE <= S5;
    elsif R='1' then
      STATE <= S0;
    end if;
```

```
        when S5 =>
          -- Data Section
          -- Control Section
            if R='0' and A='1' then
              STATE <= S1;
            elsif R='1' or A='0' then
              STATE <= S0;
            end if;
      end case;
    end if;
  end process STATE;

  OUTPUT: process (STATE)
  begin
    case STATE is
      when S0 to S4 =>
        DONE <= '0';
      when S5 =>
        DONE <= '1';
        Z <= SHIFT_REG;
    end case;
  end process OUTPUT;
end FSM_RTL;
```


State machine partitioning

- State machines partitioned into multiple processes
  - Updating (clocked), i.e., the state register
  - Next state calculation
  - Output calculation
- May find different combinations of these
  - Single process
  - Two processes (nextstate + output, state update)
  - Three processes (nextstate, output, state update)
- Multiple processes to avoid creating Mealy instead of Moore machine

State assignment

- States are not coded in VHDL
  - Use enumeration
    - Allows synthesis tools do a better work
  - Powerful computer algorithms usually find better state assignment
  - Possible to control state minimisation and assignment in synthesis tool
    - E.g. one-hot encoding may be more suitable in same cases

Alternative description: table based

- Small statemachine, one input X and one output Z
- Code the state table as an array with nextstate and output

```
entity TWO_CONSECUTIVE is
  port(CLK,R,X: in BIT; Z: out BIT);
end TWO_CONSECUTIVE;
--
architecture FSM of TWO_CONSECUTIVE is
  type STATE is (S0,S1,S2);
  signal FSM_STATE: STATE := S0;
  type TRANSITION is record
    OUTPUT: BIT;
    NEXT_STATE: STATE;
  end record;
  type TRANSITION_MATRIX is array(STATE,BIT) of TRANSITION;
  constant STATE_TRANS: TRANSITION_MATRIX :=
    (S0 => ('0' => ('0',S1), '1' => ('0',S2)),
     S1 => ('0' => ('1',S1), '1' => ('0',S2)),
     S2 => ('0' => ('0',S1), '1' => ('1',S2)));
begin
  process(R,X,CLK,FSM_STATE)
  begin
    if R = '0' then -- Reset
      FSM_STATE <= S0;
    elsif CLK'EVENT and CLK ='1' then -- Clock event
      FSM_STATE <= STATE_TRANS(FSM_STATE,X).NEXT_STATE;
    end if;
    if FSM_STATE'EVENT or X'EVENT then -- Output Function
      Z <= STATE_TRANS(FSM_STATE,X).OUTPUT;
    end if;
  end process;
end FSM;
```

2022-09-14 14:59