IE1204 Digital Design

F11: Programmable Logic, VHDL for Sequential Circuits

Elena Dubrova
KTH/ICT/ES
dubrova@kth.se
This lecture

• BV pp. 98-118, 418-426, 507-519
Programmable Logic Devices

• *Programmable logic devices (PLDs)* were introduced in the 1970s
• They are based on a structure with an AND-OR array that makes it easy to implement a sum-of-products expression
Structure of a PLD

Input buffers and inverters

AND plane

OR plane

\[ x_1, x_2, \ldots, x_n \]

\[ x_1, \bar{x}_1, \ldots, x_n, \bar{x}_n \]

\[ P_1, P_k \]

\[ f_1, f_m \]
Programmable Logic Array (PLA)

- Both AND and OR arrays are programmable

IE1204 Digital Design, HT14
Programmable Array Logic (PAL)

- Only the AND array is programmable
Combinatorial and register outputs

• In earlier PLDs there were
  – combinatorial outputs
  – register outputs (outputs with a flip-flop)
• For each circuit the number of combinational and register outputs was fixed
• To increase flexibility, *macrocells* were introduced
  – one can choose if an output is combinatorial or has a flip-flop
A programmable multiplexer can be used to select the type of output.
Programming of PLDs
Complex PLD's (CPLD)

- PLDs were quite small (PALCE 22V10 had 10 flip-flops)
- To program larger functions, structures consisting of several PLD-like blocks were developed
CPLD Structure

Interconnection wires
Programming of CPLDs via the JTAG interface

- Modern CPLDs (and FPGAs) can be programmed by downloading circuit description (programming information) via a cable
- Download usually uses a standard port called *JTAG port* (Joint Test Action Group)
You can program the chips when they are soldered to the circuit board - using the programmer you can select which chip you want to program through the JTAG port.

(a) CPLD in a Quad Flat Pack (QFP) package

(b) JTAG programming
Field Programmable Gate Arrays

- CPLDs are based on the AND-OR array
- It is difficult to make really large functions using CPLDs
- FPGAs use a different concept based on *logic blocks*
Structure of an FPGA
Look-up-tables (LUT)

A LUT with $n$ inputs can realize all combinational functions with up to $n$ inputs. The usual size of LUT in an FPGA is $n = 4$
Example: XOR-Gate

Programmed values

<table>
<thead>
<tr>
<th>$x_2$</th>
<th>$x_1$</th>
<th>$f$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Two-input LUT

Multiplexer
A logic block in an FPGA often consists of a LUT, a flip-flop and a multiplexer to select register output.
Programming the LUT's and the connection matrix in an FPGA

- Blue cross: switch is programmed
- Black cross: switch is not programmed
DE2 University Board

- DE2 Board
  - Cyclone II EP2C35 FPGA (Datorteknik-course)
  - 4 Mbytes of flash memory
  - 512 Kbytes of static RAM
  - 8 Mbytes of SDRAM
  - Several I/O-Devices
  - 50 MHz oscillator
# Cyclone II Family

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>LEs</td>
<td>4,608</td>
<td>8,256</td>
<td>14,448</td>
<td>18,752</td>
<td>33,216</td>
<td>50,528</td>
<td>68,416</td>
</tr>
<tr>
<td>M4K RAM blocks (4 Kbits plus 512 parity bits)</td>
<td>26</td>
<td>36</td>
<td>52</td>
<td>52</td>
<td>105</td>
<td>129</td>
<td>250</td>
</tr>
<tr>
<td>Total RAM bits</td>
<td>119,808</td>
<td>165,888</td>
<td>239,616</td>
<td>239,616</td>
<td>483,840</td>
<td>594,432</td>
<td>1,152,000</td>
</tr>
<tr>
<td>Embedded multipliers (3)</td>
<td>13</td>
<td>18</td>
<td>26</td>
<td>26</td>
<td>35</td>
<td>86</td>
<td>150</td>
</tr>
<tr>
<td>PLLs</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>Maximum user I/O pins</td>
<td>158</td>
<td>182</td>
<td>315</td>
<td>315</td>
<td>475</td>
<td>450</td>
<td>622</td>
</tr>
</tbody>
</table>

(3) Total Number of 18x18 Multipliers
# Stratix III Family

## Table 1–1. Stratix III FPGA Family Features

<table>
<thead>
<tr>
<th>Device/Feature</th>
<th>ALMs</th>
<th>LEs</th>
<th>M9K Blocks</th>
<th>M144K Blocks</th>
<th>MLAB Blocks</th>
<th>Total Embedded RAM Kbits</th>
<th>MLAB RAM Kbits (2)</th>
<th>Total RAM Kbits (3)</th>
<th>18×18-bit Multipliers (FIR Mode)</th>
<th>PLLs</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Stratix III Logic Family</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EP3SL50</td>
<td>19K</td>
<td>47.5K</td>
<td>108</td>
<td>6</td>
<td>950</td>
<td>1,836</td>
<td>297</td>
<td>2,133</td>
<td>216</td>
<td>4</td>
</tr>
<tr>
<td>EP3SL70</td>
<td>27K</td>
<td>67.5K</td>
<td>150</td>
<td>6</td>
<td>1,350</td>
<td>2,214</td>
<td>422</td>
<td>2,636</td>
<td>288</td>
<td>4</td>
</tr>
<tr>
<td>EP3SL110</td>
<td>43K</td>
<td>107.5K</td>
<td>275</td>
<td>12</td>
<td>2,150</td>
<td>4,203</td>
<td>672</td>
<td>4,875</td>
<td>288</td>
<td>8</td>
</tr>
<tr>
<td><strong>EP3SL150</strong></td>
<td>57K</td>
<td>142.5K</td>
<td>355</td>
<td>16</td>
<td>2,850</td>
<td>5,499</td>
<td>891</td>
<td>6,390</td>
<td>384</td>
<td>8</td>
</tr>
<tr>
<td>EP3SL200</td>
<td>80K</td>
<td>200K</td>
<td>468</td>
<td>36</td>
<td>4,000</td>
<td>9,396</td>
<td>1,250</td>
<td>10,646</td>
<td>576</td>
<td>12</td>
</tr>
<tr>
<td>EP3SE260</td>
<td>102K</td>
<td>255K</td>
<td>864</td>
<td>48</td>
<td>5,100</td>
<td>14,688</td>
<td>1,594</td>
<td>16,282</td>
<td>768</td>
<td>12</td>
</tr>
<tr>
<td>EP3SL340</td>
<td>135K</td>
<td>337.5K</td>
<td>1,040</td>
<td>48</td>
<td>6,750</td>
<td>16,272</td>
<td>2,109</td>
<td>18,381</td>
<td>576</td>
<td>12</td>
</tr>
<tr>
<td><strong>Stratix III Enhanced Family</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EP3SE50</td>
<td>19K</td>
<td>47.5K</td>
<td>400</td>
<td>12</td>
<td>950</td>
<td>5,328</td>
<td>297</td>
<td>5,625</td>
<td>384</td>
<td>4</td>
</tr>
<tr>
<td>EP3SE80</td>
<td>32K</td>
<td>80K</td>
<td>495</td>
<td>12</td>
<td>1,600</td>
<td>6,183</td>
<td>500</td>
<td>6,683</td>
<td>672</td>
<td>8</td>
</tr>
<tr>
<td>EP3SE110</td>
<td>43K</td>
<td>107.5K</td>
<td>639</td>
<td>16</td>
<td>2,150</td>
<td>8,055</td>
<td>672</td>
<td>8,727</td>
<td>896</td>
<td>8</td>
</tr>
<tr>
<td><strong>EP3SE260 (1)</strong></td>
<td>102K</td>
<td>255K</td>
<td>864</td>
<td>48</td>
<td>5,100</td>
<td>14,688</td>
<td>1,594</td>
<td>16,282</td>
<td>768</td>
<td>12</td>
</tr>
</tbody>
</table>

DE3 Board

IE1204 Digital Design, 23
HT14
Multiple processors can be implemented on an FPGA

- Nine II is a so-called 'soft-processor' (32-bit) that can be implemented on Altera’s FPGAs
- Today's FPGAs are so large that multiple processors can fit on a single FPGA chip

Very powerful multiprocessor systems can be created on an FPGA!
An ASIC (Application Specific Integrated Circuit) is a circuit which is manufactured at a semiconductor factory.

In a full custom integrated circuit, the entire circuit is customized.

In an ASIC, some design steps have already been made to reduce design time and cost.

There are several types of ASICs:
- Gate array ASICs
- Standard cell ASIC
• In a gate array ASIC, gates (or transistors) are already on silicon
We only need to create only the links between the inputs and outputs of gates.
## Comparison
### FPGA, Gate Array, Standard Cell

<table>
<thead>
<tr>
<th></th>
<th>Initial Cost</th>
<th>Cost per part</th>
<th>Performance</th>
<th>Fabrication Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>FPGA</td>
<td>Low</td>
<td>High</td>
<td>Low</td>
<td>Short</td>
</tr>
<tr>
<td>Gate Array (ASIC)</td>
<td>High</td>
<td>Low</td>
<td>High</td>
<td>Long</td>
</tr>
<tr>
<td>Standard Cell (ASIC)</td>
<td>High</td>
<td>Low</td>
<td>High</td>
<td>Long</td>
</tr>
</tbody>
</table>
Design Trade-Offs

- Microprocessor
- Programmable Logic
- Gate Array
- Standard Cell
- Full Custom
VHDL: Sequential circuits
In a Moore-type machine output signals depend only on the current state.
In a Moore machine, we have three blocks:
- Next state decoder
- Output decoder
- State register

These blocks are executed in parallel.
Quick question

• Which logic gate is represented by the following VHDL code?

\[ q \leftarrow a \text{ and } \neg b; \]

Alt: A

Alt: B

Alt: C
Quick question

- Which logic gate is represented by the following VHDL code?

```vhdl
if (a /= b) then
  q <= '1';
else
  q <= '0';
end if;
```

Alt: A

Alt: B

Alt: C
Processes in VHDL

• A *architecture* in VHDL can contain multiple processes
• Processes are executed in parallel
• A process is written as a sequential program
Moore-machine processes

- For a Moore machine, we create three processes
  - Next state decoder
  - Output decoder
  - State register
Internal signals

• Moore machine contains internal signals for
  – Current state
  – Next state

• These signals are declared in the architecture description
We use bottle dispenser vending machine as an example

We describe its system controller in VHDL
Bottle dispenser

• Bottle dispenser consists of several parts
  – COIN RECEIVER
  – DROP BOTTLE
  – COIN RETURN
• Machine accepts only the following coins: 1 Euro, 50 Cent, 10 Cent
• The vending machine only returns 10 Cent coins
Signal Properties

- DROP_READY is active for one clock cycle after the bottle has been ejected
- CHANGER_READY is active for one clock cycle after a 10 Cent coin is ejected
- Because of the mechanical properties, the following signals are active and inactive for several clock cycles:
  - COIN_PRESENT (active for several clock cycles after the coin drop)
  - DROP_READY (active for several clock periods after bottle drop)
  - CHANGER_READY (inactive for several clock periods after coin return)
Flow diagram of control system

- Coin
  - 10 Cent, 50 Cent, 1 Euro
- Coin return
  - 10 Cent
- Bottle Price
  - 1 Euro

1. Coin registered?
   - No: Total < 1 €
   - Yes:
     - Total < 1 €: Eject bottle
     - Total = 1 €: Return 10 Cent
     - Total > 1 €: Decrease sum
2. Reset sum
3. Return 10 Cent
4. Decrease sum
5. Eject bottle
State diagram (Moore)

- Upon entry into the state signal becomes active
- When exiting the state signal becomes inactive

(a) Wait for coin
(b) Register coin
(c) Coin is registered (3 cases)
(d) Drop bottle
(e) Reset sum
(f) Return 10 Cent
(g) Decrease sum with 10 Cent
State diagram

- State assignment with no claim for optimality (Ad hoc)
  - (a) next to (b)
  - (b) next to (c)
  - (d) next to (e)
  - (f) next to (g)

- For all these cases, only one variable changes

<table>
<thead>
<tr>
<th>AB</th>
<th>00</th>
<th>01</th>
<th>11</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>a</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>b</td>
<td>c</td>
<td>e</td>
<td>g</td>
</tr>
</tbody>
</table>

(“-“ = don’t care)
The state diagram contains all information required to generate an implementation.

- **Assumption**: D flip-flops are used as state register.

The state variable order is ABC, i.e. state (c) is A = 0, B = 1, C = 1.
Construction of next-state and output decoders

At next step, we develop the logic for the next state \((D_A, D_B, D_C)\) and outputs
Decoder: Next state - $D_A$

\[
D_A = \overline{AB}(=) + \overline{AB}(>) + AC
\]
Variable-Entered Mapping (VEM)

- Variable-Entered Mapping can help to draw and minimize Karnaugh diagrams with many variables.
- Instead of opening an "extra dimension" we write a variable into the Karnaugh map.
- You must be extra careful when drawing circuits so that you do not forget a variable combination!

<table>
<thead>
<tr>
<th>C</th>
<th>DA</th>
<th>AB</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>00 01 11 10</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0 - 1 1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>(=) + (&gt;0</td>
</tr>
</tbody>
</table>
Decoder: Next state - $D_B$

$$D_B = \overline{AB(=)} + BC \overline{C} + BC(CP) + ABC$$
Decoder: Next state- $D_c$

$$D_c = \overline{A}C(CP) + B\overline{C}(DR) + \overline{AB}(CR) + \overline{BC}$$

<table>
<thead>
<tr>
<th>C</th>
<th>00</th>
<th>01</th>
<th>11</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>CP</td>
<td>-</td>
<td>DR</td>
<td>CR</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

CP : COIN_PRESENT
DR: DROP_READY
CR: CHANGER_READY
Decoder: Output signals

- Output decoder is trivial, since its value is directly dependent on the current state

DROP = \overline{ABC}
CLR_ACC = ABC
RETURN_10_CENT = \overline{ABC}
DEC_ACC = \overline{ABC}
Implementation

- The logical expressions are then implemented with logic gates as usually
ENTITY Vending_Machine IS
  PORT (  
    -- Inputs
    coin_present  : IN std_logic;
    gt_1_euro     : IN std_logic;
    eq_1_euro     : IN std_logic;
    lt_1_euro     : IN std_logic;
    drop_ready    : IN std_logic;
    changer_ready : IN std_logic;
    reset_n       : IN std_logic;
    clk           : IN std_logic;
    -- Outputs
    dec_acc        : OUT std_logic;
    clr_acc        : OUT std_logic;
    drop           : OUT std_logic;
    return_10_cent : OUT std_logic);
END Vending_Machine;

• Entity describes the system as a 'black box'
• Entity describes the interface to the outside world
• All inputs and outputs are described
• Apart from the input and output signals, block diagram needs signals for
  – Clock
  – Reset (active low)
The architecture describes the function of the machine

We define

– internal signals for the current and next states
– three processes for next-state decoder, output decoder and state register
Upon entry into the state signal becomes active
When exiting the state signal becomes inactive

(a) Wait for coin
(b) Register coin
(c) Coin is registered (3 cases)
(d) Bottle drop
(e) Reset sum
(f) Return 10 Cent
(g) Decrease sum with 10 Cent
- We need to create a type for internal signals
- Since we describe the states, we use an enumerated type with the values a, b, c, d, e, f, g
- We declare one variable for the current state (current_state) and one for the next state (next_state)

ARCHITECTURE Moore_FSM OF Vending_Machine IS 
   
   TYPE state_type IS (a, b, c, d, e, f, g);
   SIGNAL current_state, next_state : state_type;
   BEGIN -- Moore_FSM
   ...

Vending machine
Internal Signaler
If we do not specify a state assignment, synthesis tool will select it
We can force a certain encoding using attributes (NOTE: Attributes are dependent on synthesis tool and thus are not portable!)

ARCHITECTURE Moore_FSM OF Vending_Machine IS
  TYPE state_type IS (a, b, c, d, e, f, g);
  -- We can use state encoding according to BV 8.4.6
  -- to enforce a particular encoding (for Quartus)
  ATTRIBUTE enum_encoding : string;
  ATTRIBUTE enum_encoding OF state_type : TYPE IS "000 001 011 110 111 100 101";
  SIGNAL current_state, next_state : state_type;
BEGIN  -- Moore_FSM
  ...
Vending machine Block diagram

- Signals A, B, C describe current state
- Signals $D_A$, $D_B$, $D_C$ describe next state
Quick question

- Which state machine is represented by this VHDL code?

```vhdl
case state is
  when 0 =>
    if (k = '1') then
      nextstate <= 1;
    else
      nextstate <= 2;
    end if;
  when 1 => nextstate <= 2;
  when others => nextstate <= 0;
end case;
```
Vending machine
Next-State Decoder

- Next-State-Decoder is described as a process
- Sensitivity list contains all the inputs that 'activate' the process

\[
\text{NEXTSTATE : PROCESS (current\_state, coin\_present, gt\_1\_euro, eq\_1\_euro, lt\_1\_euro, drop\_ready, changer\_ready) -- Sensitivity List}
\]
\[
\text{BEGIN -- PROCESS NEXT\_STATE}
\]
\[
\text{...}
\]
We now use a CASE statement to describe for each state conditions for transitions to the next state

... 

CASE current_state IS 
  WHEN a => IF coin_present = '1' THEN 
    next_state <= b; 
  ELSE 
    next_state <= a; 
  END IF; 
  WHEN b => IF coin_present = '0' THEN 
    next_state <= c; 
  ELSE 
    next_state <= b; 
  END IF;
Vending machine
Next-State Decoder

- We can simplify the description by specifying a default value for the next state

... 

next_state <= current_state;
CASE current_state IS
  WHEN a => IF coin_present = '1' THEN
    next_state <= b;
    END IF;
  WHEN b => IF coin_present = '0' THEN
    next_state <= c;
    END IF;

...

It is important to specify all options for next_state signal. Otherwise we may implicitly set next_state <= next_state which generates a loop.
We terminate the CASE statement with a WHEN OTHERS statement. Here we specify that we should go to the state \( a \) if we end up in an unspecified state

\[
\begin{align*}
&\text{WHEN } g \Rightarrow \text{next}_\text{state} \leq c; \\
&\text{WHEN OTHERS } \Rightarrow \text{next}_\text{state} \leq a; \\
&\text{END CASE;}
\end{align*}
\]

END PROCESS NEXTSTATE;
Vending machine Output decoder

- Output decoder is described as a separate process
- Sensitivity list contains only the current state because the outputs are directly dependent on it
Vending machine Block diagram

- Signals A, B, C describe current state
- Signals $D_A$, $D_B$, $D_C$ describe next state
Vending machine
Output decoder

OUTPUT : PROCESS (current_state)
BEGIN
  -- PROCESS OUTPUT

  drop <= '0';
  clr_acc <= '0';
  dec_acc <= '0';
  return_10_cent <= '0';

  CASE current_state IS
    WHEN d   => drop <= '1';
    WHEN e   => clr_acc <= '1';
    WHEN f   => return_10_cent <= '1';
    WHEN g   => dec_acc <= '1';
    WHEN OTHERS => NULL;
  END CASE;
END PROCESS OUTPUT;
State register is modeled as a synchronous process with asynchronous reset (active low)

CLOCK : PROCESS (clk, reset_n)
BEGIN
  IF reset_n = '0' THEN -- asynchronous reset (active low)
    current_state <= a;
  ELSIF clk'EVENT AND clk = '1' THEN -- rising clock edge
    current_state <= next_state;
  END IF;
END PROCESS CLOCK;
In a Mealy machine, output signals depend on both the current state and inputs.
Mealy machine in VHDL

• A Mealy machine can be modeled in the same way as the Moore machine
• The difference is that output decoder is also dependent on the input signals
• Process which models outputs needs to have input signals in the sensitivity list as well!
More on VHDL

- The sample code for bottle dispenser available on the course website
- Look at the study of "VHDL synthesis" on the course website
- Both Brown/Vranesic- and Hemert-book includes code samples
Summary

• PLD, PAL, CPLD
• FPGA
• ASIC – gate array and standard cell
• Modeling sequential circuits with VHDL
• Next lecture: BV pp. 584-640