Synthesis Of VHDL Code

Outline

1. Fundamental limitation of EDA software
2. Realization of VHDL operator
3. Realization of VHDL data type
4. VHDL synthesis flow
5. Timing consideration

1. Fundamental limitation of EDA software

- Can “C-to-hardware” be done?
- EDA tools:
  - Core: optimization algorithms
  - Shell: wrapping
- What does theoretical computer science say?
  - Computability
  - Computation complexity

Computability

- A problem is computable if an algorithm exists.
- E.g., “halting problem”:
  - can we develop a program that takes any program and its input, and determines whether the computation of that program will eventually halt?
- any attempt to examine the “meaning” of a program is uncomputable

Computation complexity

- How fast an algorithm can run (or how good an algorithm is)?
- “Interferences” in measuring execution time:
  - types of CPU, speed of CPU, compiler etc.

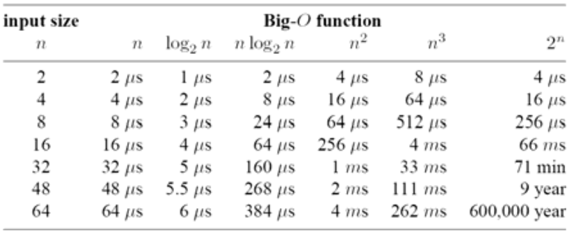
Big-O notation

- $f(n)$ is $O(g(n))$: if n_0 and c can be found to satisfy: $f(n) < cg(n)$ for any n, $n > n_0$
- $g(n)$ is simple function: $1, n, \log_2 n, n^2, n^3, 2^n$
- Following are $O(n^2)$:
  - $0.1n^2$
  - $n^2 + 5n + 9$
  - $500n^2 + 1000000$

Interpretation of Big-O

- Filter out the “interference”: constants and less important terms
- n is the input size of an algorithm
- The “scaling factor” of an algorithm: What happens if the input size increases

E.g.,



input size	Big-O function					
n	n	$\log_2 n$	$n \log_2 n$	n^2	n^3	2^n
2	2 μs	1 μs	2 μs	4 μs	8 μs	4 μs
4	4 μs	2 μs	8 μs	16 μs	64 μs	16 μs
8	8 μs	3 μs	24 μs	64 μs	512 μs	256 μs
16	16 μs	4 μs	64 μs	256 μs	4 ms	66 ms
32	32 μs	5 μs	160 μs	1 ms	33 ms	71 min
48	48 μs	5.5 μs	268 μs	2 ms	111 ms	9 year
64	64 μs	6 μs	384 μs	4 ms	262 ms	600,000 year

- Intractable problems:
  - algorithms with $O(2^n)$
  - Not realistic for a larger n
  - Frequently tractable algorithms for sub-optimal solution exist
- Many problems encountered in synthesis are intractable

Theoretical limitation

- Synthesis software does not know your intention
- Synthesis software cannot obtain the optimal solution
- Synthesis should be treated as transformation and a “local search” in the “design space”
- Good VHDL code provides a good starting point for the local search

- What is the fuss about:
  - “hardware-software” co-design?
  - SystemC, HardwareC, SpecC etc.?

2. Realization of VHDL operator

- Logic operator
  - Simple, direct mapping
- Relational operator
  - =, /= fast, simple implementation exists
  - >, < etc: more complex implementation, larger delay
- Addition operator
- Other arith operators: support varies

- Operator with two constant operands:
  - Simplified in preprocessing
  - No hardware inferred
  - Good for documentation
  - E.g.,

```
constant OFFSET: integer := 8;
signal boundary: unsigned(8 downto 0);
signal overflow: std_logic;
. . .
overflow <= '1' when boundary > (2**OFFSET-1) else
            '0';
```

- Operator with one constant operand:
  - Can significantly reduce the hardware complexity
  - E.g., adder vs. incrementor
  - E.g

    y <= rotate_right(x, y); -- barrel shifter
    y <= rotate_right(x, 3); -- rewiring
    y <= x(2 downto 0) & x(7 downto 3);
  - E.g., 4-bit comparator: x=y vs. x=0

$$(x_3 \oplus y_3)' \cdot (x_2 \oplus y_2)' \cdot (x_1 \oplus y_1)' \cdot (x_0 \oplus y_0)'$$

$$x_3' \cdot x_2' \cdot x_1' \cdot x_0'$$

An example 0.55 um standard-cell CMOS implementation

width	VHDL operator									
	nand	xor	$>_a$	$>_d$	=	$+1_a$	$+1_d$	$+_a$	$+_d$	mux
	area (gate count)									
8	8	22	25	68	26	27	33	51	118	21
16	16	44	52	102	51	55	73	101	265	42
32	32	85	105	211	102	113	153	203	437	85
64	64	171	212	398	204	227	313	405	755	171
	delay (ns)									
8	0.1	0.4	4.0	1.9	1.0	2.4	1.5	4.2	3.2	0.3
16	0.1	0.4	8.6	3.7	1.7	5.5	3.3	8.2	5.5	0.3
32	0.1	0.4	17.6	6.7	1.8	11.6	7.5	16.2	11.1	0.3
64	0.1	0.4	35.7	14.3	2.2	24.0	15.7	32.2	22.9	0.3

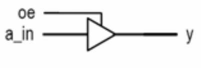
---

3. Realization of VHDL data type

- Use and synthesis of ‘Z’
- Use of ‘-’

Use and synthesis of ‘Z’

- Tri-state buffer:
  - Output with “high-impedance”
  - Not a value in Boolean algebra
  - Need special output circuitry (tri-state buffer)



oe	y
0	Z
1	a_in

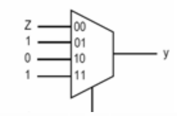
- Major application:
  - Bi-directional I/O pins
  - Tri-state bus
- VHDL description:

  y <= 'Z' **when** oe='1' **else**
  a_in;
- ‘Z’ cannot be used as input or manipulated

  f <= 'Z' **and** a;
  y <= data_a **when** in_bus='Z' **else**
  data_b;

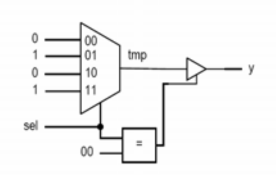
- Separate tri-state buffer from regular code:
  - Less clear:

with sel **select**
  y <= 'Z' **when** "00",
      '1' **when** "01"|"11",
      '0' **when others**;

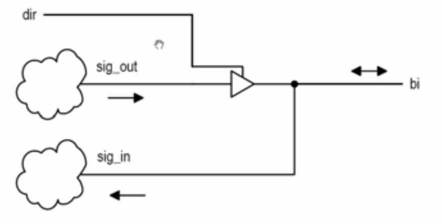


  - better:

with sel **select**
  tmp <= '1' **when** "01"|"11",
      '0' **when others**;
 y <= 'Z' **when** sel="00" **els**
    tmp;

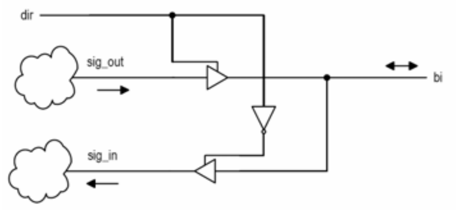


Bi-directional i/o pins



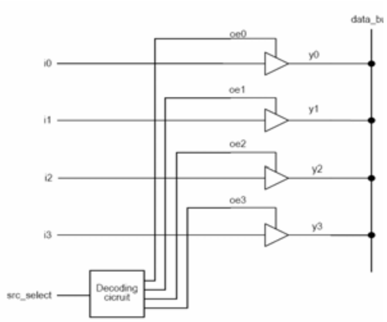
```
entity bi_demo is
   port(bi: inout std_logic;
 . . .

begin
   sig_out <= output_expression;
    . . .  <= expression_with_sig_in;
   . . .
   bi <= sig_out when dir=’1’ else ’Z’;
   sig_in <= bi;
   . . .
```



```
sig_in <= bi when dir=’0’ else ’Z’;
```

Tri-state bus



```
-- binary decoder
with src_select select
   oe <= "0001" when "00",
         "0010" when "01",
         "0100" when "10",
         "1000" when others; -- "11"
-- tri-state buffers
y0 <= i0 when oe(0)=’1’ else ’Z’;
y1 <= i1 when oe(1)=’1’ else ’Z’;
y2 <= i2 when oe(2)=’1’ else ’Z’;
y3 <= i3 when oe(3)=’1’ else ’Z’;
data_bus <= y0;
data_bus <= y1;
data_bus <= y2;
data_bus <= y3;
```

- Problem with tri-state bus
  - Difficult to optimize, verify and test
  - Somewhat difficult to design: “parking”, “fighting”
- Alternative to tri-state bus: mux

```
with src_select select
   data_bus <= i0 when "00",
               i1 when "01",
               i2 when "10",
               i3 when others;  -- "11"
```

Use of ‘-’

- In conventional logic design
  - ‘-’ as input value: shorthand to make table compact
  - E.g.,

input	output
req	code
100	10
101	10
110	10
111	10
010	01
011	01
001	00
000	00

input	output
req	code
1--	10
01-	01
001	00
000	00

- ‘-’ as output value: help simplification
- E.g.,
  ‘-’ assigned to 1: a + b
  ‘-’ assigned to 0: a’b + ab’

input	output
a b	*f*
0 0	0
0 1	1
1 0	1
1 1	-

Use ‘-’ in VHDL

- As input value (against our intuition):
- Wrong:

```
y <= "10" when req="1--" else
     "01" when req="01-" else
     "00" when req="001" else
     "00";
```

- Fix #1:

```
y <= "10" when req(3)='1' else
     "01" when req(3 downto 2)="01" else
     "00" when req(3 downto 1)="001" else
     "00";
```

- Fix #2:

```
. . .
use ieee.numeric_std.all;
. . .
y <= "10" when std_match(req,"1--") else
     "01" when std_match(req,"01-") else
     "00" when std_match(req,"001") else
     "00";
```

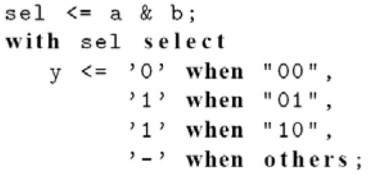
- Wrong:

```
with req select
   y <= "10" when "1--",
        "01" when "01-",
        "00" when "001",
        "00" when others;
```

- Fix:

```
with req select
   y <= "10" when "100"|"101"|"110"|"111",
        "00" when "010"|"011",
        "00" when others;
```

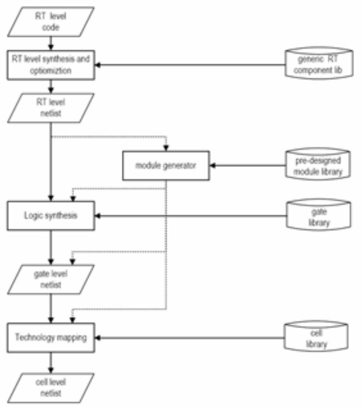
- ‘-’ as an output value in VHDL
- May work with some software



```
sel <= a & b;
with sel select
   y <= '0' when "00",
        '1' when "01",
        '1' when "10",
        '-' when others;
```

4. VHDL Synthesis Flow

- Synthesis:
  - Realize VHDL code using logic cells from the device’s library
  - a refinement process
- Main steps:
  - RT level synthesis
  - Logic synthesis
  - Technology mapping



RT level synthesis

- Realize VHDL code using RT-level components
- Somewhat like the derivation of the conceptual diagram
- Limited optimization
- Generated netlist includes
  - “regular” logic: e.g., adder, comparator
  - “random” logic: e.g., truth table description

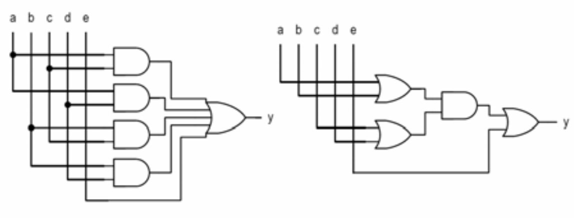
Module generator

- “regular” logic can be replaced by pre-designed module
  - Pre-designed module is more efficient
  - Module can be generated in different levels of detail
  - Reduce the processing time

Logic Synthesis

- Realize the circuit with the optimal number of “generic” gate level components
- Process the “random” logic
- Two categories:
  - Two-level synthesis: sum-of-product format
  - Multi-level synthesis

- E.g.,



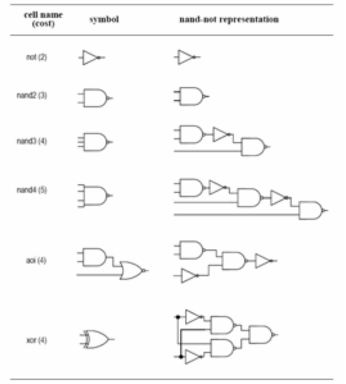
(a) Two-level implementation (b) multi-level implementation

Technology mapping

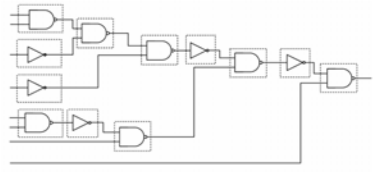
- Map “generic” gates to “device-dependent” logic cells
- The technology library is provided by the vendors who manufactured (in FPGA) or will manufacture (in ASIC) the device

E.g., mapping in standard-cell ASIC

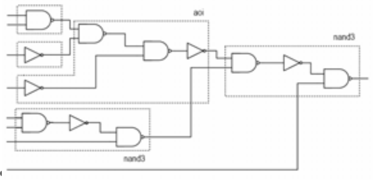
- Device library



- Cost: 31 vs. 17



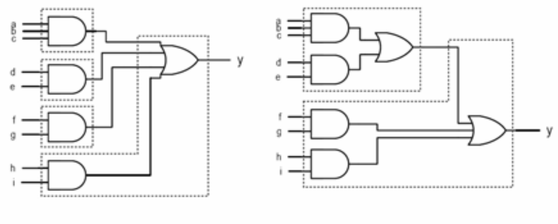
(a) Initial mapping



(b) Better mapping

E.g., mapping in FPGA

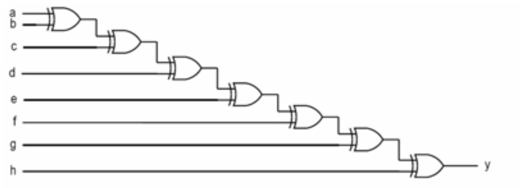
- With 5-input LUT (Look-Up-Table) cells



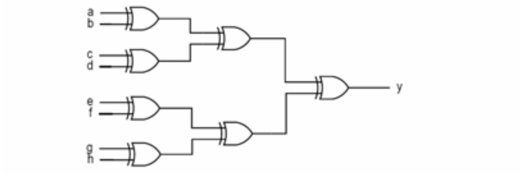
(a) Initial mapping (b) Better mapping

Effective use of synthesis software

- Logic operators: software can do a good job
- Relational/Arith operators: manual intervention needed
- “layout” and “routing structure”:
  - Silicon chip is 2-dimensional square
  - “rectangular” or “tree-shaped” circuit is easier to optimize



(a) Cascading-chain structure



5. Timing consideration

- Propagation delay
- Synthesis with timing constraint
- Hazards
- Delay-sensitive design

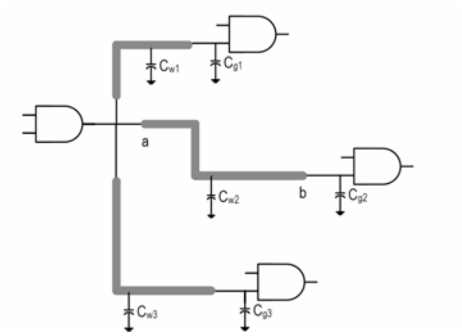
Propagation delay

- Delay: time required to propagate a signal from an input port to a output port
- Cell level delay: most accurate
- Simplified model:

$$delay = d_{intrinsic} + r * C_{load}$$

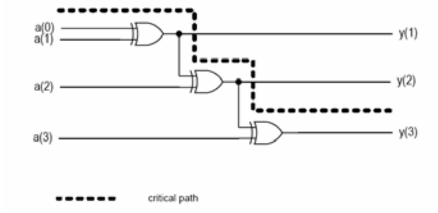
- The impact of wire becomes more dominant

- E.g.

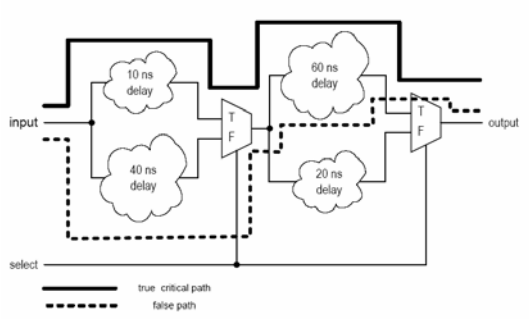


System delay

- The longest path (critical path) in the system
- The worst input to output delay
- E.g



- "False path" may exists:

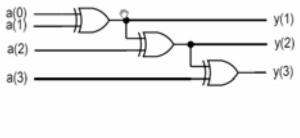


- RT level delay estimation:
  - Difficult if the design is mainly “random” logic
  - Critical path can be identified if many complex operators (such adder) are used in the design.

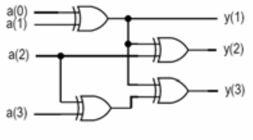
Synthesis with timing constraint

- Multi-level synthesis is flexible
- It is possible to reduce by delay by adding extra logic
- Synthesis with timing constraint
  1. Obtain the minimal-area implementation
  2. Identify the critical path
  3. Reduce the delay by adding extra logic
  4. Repeat 2 & 3 until meeting the constraint

- E.g.,

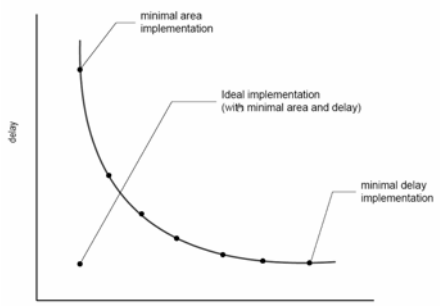


(a) Optimized for area

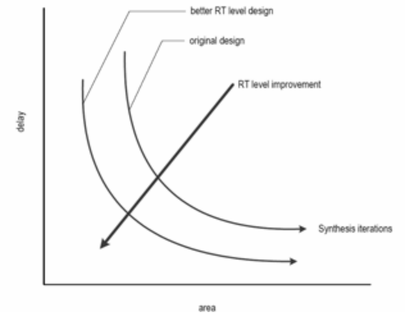


(b) Optimized for delay

- Area-delay trade-off curve



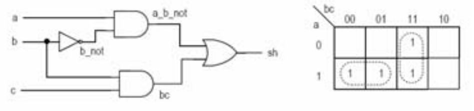
- Improvement in “architectural” level design (better VHDL code to start with)



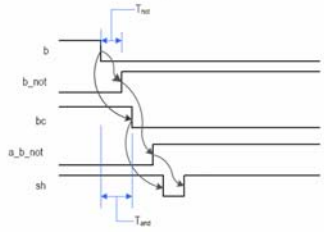
Timing Hazards

- Propagation delay: time to obtain a stable output
- Hazards: the fluctuation occurring during the transient period
  - Static hazard: glitch when the signal should be stable
  - Dynamic hazard: a glitch in transition
- Due to the multiple converging paths of an output port

- E.g., static-hazard (sh=ab'+bc; a=c=1)

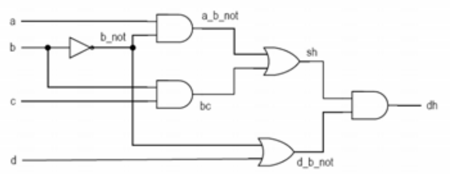


(a) Karnaugh map and schematic

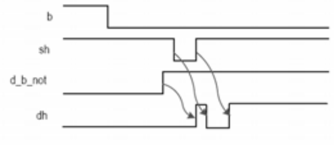


(b) Timing diagram

- E.g., dynamic hazard (a=c=d=1)



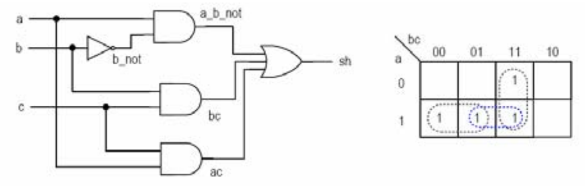
(a) Schematic



(b) Timing diagram

Dealing with hazards

- Some hazards can be eliminated in theory
- E.g.,



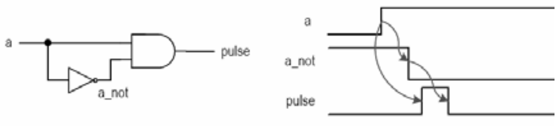
(c) Revised Karnaugh map and schematic to eliminate hazards

- Eliminating glitches is very difficult in reality, and almost impossible for synthesis
- Multiple inputs can change simultaneously (e.g., 1111=>0000 in a counter)
- How to deal with it?
  Ignore glitches in the transient period and retrieve the data after the signal is stabilized

Delay sensitive design and its danger

- Boolean algebra
  - the theoretical model for digital design and most algorithms used in synthesis process
  - algebra deals with the stabilized signals
- Delay-sensitive design
  - Depend on the transient property (and delay) of the circuit
  - Difficult to design and analyze

- E.g., hazard elimination circuit:
  ac term is not needed
- E.g., edge detection circuit (pulse=a a')



- What’s can go wrong:
  - E.g., pulse <= a **and** (not a);
  - During logic synthesis, the logic expressions will be rearranged and optimized.
  - During technology mapping, generic gates will be re-mapped
  - During placement & routing, wire delays may change
  - It is bad for testing verification
- If delay-sensitive design is really needed, it should be done manually, not by synthesis