Processor Design

José María Arnau
Logic Synthesis

- Convert RTL description of a circuit into a design implementation in terms of logic gates
- Done automatically by a synthesis tool
Levels of Abstraction

• Behavioral
  – Behavior of circuit described using imperative programming
  – always-block in Verilog or process-block in VHDL

• Register Transfer Level (RTL)
  – Design represented by combinatorial datapaths and registers
  – In Verilog: only assign sentences and very restricted always-blocks

• Logic Gate Level
  – Design represented by a netlist single bit cells, including basic logic gates (AND, OR, NOT, XOR...) and registers (D-type flip-flops)
Levels of Abstraction

Behavioral description

```vhdl
always @(posedge clk)
begin
    if (rst) accum <= 26'd0;
    else    accum <= accum + Ai * Bi;
end
```

Register Transfer Level

```vhdl
assign tmp = a + b;
always @(posedge clk)
begin
    y <= tmp;
end
```

Logic Gate Level

```vhdl
wire x, y, w;
wire z0, z1, z2;

and a1(z0, x, y);
or a2(z1, z0, w);
xor a3(z2, z1, z0);
```
Synthesizable code

• Only a subset of an HDL is synthesizable
  – Other constructs of the language are only used for writing testbenches

• Synthesizable subset:
  – RTL + Part of the behavioral constructs
  – Standard that specifies synthesizable Verilog

• Logic synthesis tools support standard synthesizable subset plus some additional behavioral statements
Synthesizable Verilog

• Structural Verilog
  – Constant values
  – Wire and port declarations
  – Static assignment of signals to other signals
  – Module instantiations

• Expressions
  – Arithmetic operators: +, -, *, &, |, ^...
Synthesizable Verilog

- Behavioral modeling
  - Asynchronous or latched logic

```verilog
// asynchronous
always @* begin
  if (add_mode)
    y <= a + b;
  else
    y <= a - b;
end

// latched
always @* begin
  if (!hold)
    y <= a + b;
end
```
Synthesizable Verilog

- Behavioral modeling
  - Synchronous logic (with optional synchronous reset)

```verilog
// counter with synchronous reset
always @(posedge clk) begin
  if (reset)
    y <= 0;
  else
    y <= y + 1;
end
```
Synthesizable Verilog

- Behavioral modeling
  - Synchronous logic with asynchronous reset

```verilog
// counter with asynchronous reset
always @(posedge clk, posedge reset) begin
  if (reset)
    y <= 0;
  else
    y <= y + 1;
end
```
Synthesizable Verilog

- **Conditionals**
  - If-else statements inside always-block

- **Loops**
  - for-loops inside always-block
  - Constant number of iterations
  - Synthesis tool must be able to fully unroll the loop

- **Generate-statements**
  - loops inside generate statements evaluated at synthesis time
Logic Synthesis

```
// 2:1 multiplexer
always @ (a or b) begin
    if (a) z <= b;
    else z <= a;
end
```

Figures by MIT OCW.

Synthesis tools

• Automatically synthesize a logic implementation from a behavioral description in an HDL

• Synthesis programs process HDL:
  – Infer logic and state elements
  – Perform technology-independent optimizations
    • Logic simplification, state assignment…
  – Map elements to target technology
  – Perform technology-dependent optimizations
    • Multi-level logic optimization…
Logic Synthesis

```plaintext
assign z = (a & b) | c;
```

```plaintext
// dataflow
assign z = sel ? a : b;
```
Logic Synthesis

wire [3:0] x, y, sum;
wire cout;
assign {cout, sum} = x + y;

+ operator implemented as a ripple carry adder
module parity(in,p);
    parameter WIDTH = 2;   // default width is 2
    input [WIDTH-1 : 0] in;
    output p;
    // simple approach: assign p = ^in;
    // here's another, more general approach
    reg p;
    always @(in) begin: loop
        integer i;
        reg parity = 0;
        for (i = 0; i < WIDTH; i = i + 1)
            parity = parity ^ in[i];
        p <= parity;
    end
endmodule

...  
wire [3:0] word;
wire parity;
parity #(4) ecc(word,parity);    // specify WIDTH = 4
if boolean_expr_1 then
  if boolean_expr_2 then
    signal_a <= value_expr_1;
  else
    signal_a <= value_expr_2;
  end if;
else
  if boolean_expr_3 then
    signal_a <= value_expr_3;
  else
    signal_a <= value_expr_4;
  end if;
end if;
```vhdl
    case case_expr is
        when c0 =>
            sig_a <= value_expr_a_0;
            sig_b <= value_expr_b_0;
        when c1 =>
            sig_a <= value_expr_a_1;
            sig_b <= value_expr_b_1;
        when others =>
            sig_a <= value_expr_a_n;
            sig_b <= value_expr_b_n;
    end case;
```
Synthesis of Sequential Logic

```verilog
reg q;

// D-latch
always @(g or d) begin
  if (g) q <= d;
end

Synthesis tool will infer a latch!
```

```verilog
reg q;  // this time we mean it!

// D-register
always @(posedge clk) begin
  q <= d;
end
```

Synthesis of Sequential Logic

```plaintext
reg q;
// register with synchronous clear
always @(posedge clk) begin
    if (!reset) // reset is active low
        q <= 0;
    else
        q <= d;
end

reg q;
// register with asynchronous clear
always @(posedge clk or negedge reset) begin
    if (!reset) // reset is active low
        q <= 0;
    else // implicit posedge clk
        q <= d;
end
// warning! async inputs are dangerous!
// there’s a race between them and the
// rising edge of clk.
```

Technology-independent optimizations

- Two-level boolean minimization
  - Reducing the number of products terms and reducing the size of each product term will result in a smaller/faster implementation

- Optimizing finite state machines
  - Find equivalent FSM with fewer states

- Experience has shown it’s advantageous to reduce problem size as much as possible before the technology mapping and technology-dependent optimizations
## Sum-of-products

<table>
<thead>
<tr>
<th>w</th>
<th>x</th>
<th>y</th>
<th>z</th>
<th>out</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

![OR-gate 9-1 diagram]

out

21 / 86
Two-Level Boolean Minimization

- Optimize boolean function expressed as a sum-of-products
- Optimization criteria
  - Number of product terms
  - Number of literals
  - A combination of both
- Minimization steps:
  - Generate the set of prime product-terms for the function
  - Select a minimum set of prime terms to cover the function
- State-of-the-art logic minimization algorithms are based on Quine-McCluskey method
Prime Term Generation

- Express your Boolean function using 0-terms (products with no don’t care entries)
  - Include only those entries where the output of the function is 1
  - Label each term with its decimal equivalent
- Look for pairs of 0-terms that differ only in only 1 bit and merge them in a 1-term
- Next 1-terms are examined in pairs to merge them in 2-terms
- Unmerged terms are prime

<table>
<thead>
<tr>
<th>W</th>
<th>X</th>
<th>Y</th>
<th>Z</th>
<th>label</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>7</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>8</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>9</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>10</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>11</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>14</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>15</td>
</tr>
</tbody>
</table>

| 0 | 8 | 000[A] |
| 5 | 7 | 01-1[B] |
| 7, 15 | -111[C] |
| 8, 9 | 100-|
| 8, 10 | 10-0 |
| 9, 11 | 10-1 |
| 10, 11 | 101- |
| 10, 14 | 1-10 |
| 11, 15 | 1-11 |
| 14, 15 | 111- |

| 2-terms: | 8, 9, 10, 11 | 10- [D] |
|          | 10, 11, 14, 15 | 1-1 [E] |

| 3-terms: | none! |

Prime Term Table

- An “X” in row R and column C means that the 0-term corresponding to row R is contained by the prime corresponding to column C
- Goal: select the minimum set of primes such that there is at least one “X” in every row

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>X</td>
<td>.</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>0101</td>
<td>.</td>
<td>X</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>0111</td>
<td>.</td>
<td>X</td>
<td>X</td>
<td>.</td>
</tr>
<tr>
<td>1000</td>
<td>X</td>
<td>.</td>
<td>.</td>
<td>X</td>
</tr>
<tr>
<td>1001</td>
<td>.</td>
<td>.</td>
<td>X</td>
<td>.</td>
</tr>
<tr>
<td>1010</td>
<td>.</td>
<td>.</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>1011</td>
<td>.</td>
<td>.</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>1100</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>X</td>
</tr>
<tr>
<td>1110</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>X</td>
</tr>
<tr>
<td>1111</td>
<td>.</td>
<td>.</td>
<td>X</td>
<td>.</td>
</tr>
</tbody>
</table>
Two-Level Boolean Minimization

1-terms:

- 0, 8 -000[A]
- 5, 7 01-1[B]
- 7, 15 -111[C]
- 8, 9 100-
- 8, 10 10-0
- 9, 11 10-1
- 10, 11 101-
- 10, 14 1-10
- 11, 15 1-11
- 14, 15 111-

2-terms:

- 8, 9, 10, 11 10-[D]
- 10, 11, 14, 15 1-1-[E]

A B C D E

0000 X . . .
0101 . X . .
0111 . X X .
1000 X . . X
1001 . . X .
1010 . . X X
1011 . . X X
1110 . . . X
1111 . . X . X

A is essential
B is essential
D is essential
E is essential

Dominated Columns

- Some functions may not have essential primes
  - Make arbitrary selection of first prime (e.g. A)
  - A column U of a prime term table dominates V if U contains every row contained in V
  - Delete dominated columns
  - Backtrack our choice and explore different prime
  - Repeat and select best solution

1. Prime table
   
<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
<th>H</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>0001</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0101</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0111</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1000</td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1010</td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1110</td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1111</td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

2. Table with A selected
   
<table>
<thead>
<tr>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
<th>H</th>
</tr>
</thead>
<tbody>
<tr>
<td>0101</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0111</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1000</td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1010</td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1110</td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1111</td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

C dominates B, G dominates H

3. Table with B & H removed
   
<table>
<thead>
<tr>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
</tr>
</thead>
<tbody>
<tr>
<td>0101</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0111</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1000</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1010</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1110</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1111</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

C is essential
G is essential

Selecting C and G shows that only E is needed to complete the cover
Mapping to technology library

- The next step is mapping each equation to the gates available in the target cell library.
- Example: find optimal mapping of this circuit...

...into this library:
Mapping to technology library

- Popular approach: **DAG covering**
  - Represent input netlist in normal form (“subject DAG”)
    - normal form: 2-input NAND gates + inverters
  - Represent each library gate in normal form (“primitive DAGs”)
  - Goal: find a minimum cost covering of the subject DAG by the primitive DAGs
  - If subject and primitive DAGs are trees, efficient algorithm (dynamic programming) for finding optimum
    - Partition subject DAG into a forest of trees (each gate with fanout > 1 becomes root of a new tree)
    - Generate optimal solution for each tree
    - Stitch solutions together
### Primitive DAGs

<table>
<thead>
<tr>
<th>Element/Area Cost</th>
<th>Tree Representation (normal form)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>INVERTER</strong></td>
<td><img src="image" alt="INVERTER" /></td>
</tr>
<tr>
<td>2</td>
<td><img src="image" alt="Tree for INVERTER" /></td>
</tr>
<tr>
<td><strong>NAND2</strong></td>
<td><img src="image" alt="NAND2" /></td>
</tr>
<tr>
<td>3</td>
<td><img src="image" alt="Tree for NAND2" /></td>
</tr>
<tr>
<td><strong>NAND3</strong></td>
<td><img src="image" alt="NAND3" /></td>
</tr>
<tr>
<td>4</td>
<td><img src="image" alt="Tree for NAND3" /></td>
</tr>
<tr>
<td><strong>NAND4</strong></td>
<td><img src="image" alt="NAND4" /></td>
</tr>
<tr>
<td>5</td>
<td><img src="image" alt="Tree for NAND4" /></td>
</tr>
<tr>
<td><strong>AOI21</strong></td>
<td><img src="image" alt="AOI21" /></td>
</tr>
<tr>
<td>4</td>
<td><img src="image" alt="Tree for AOI21" /></td>
</tr>
<tr>
<td><strong>AOI22</strong></td>
<td><img src="image" alt="AOI22" /></td>
</tr>
<tr>
<td>5</td>
<td><img src="image" alt="Tree for AOI22" /></td>
</tr>
</tbody>
</table>

Possible covers

systematic and efficient way to find the optimal answer?

Optimal tree covering

- Optimal cover for a tree consists of a best match at the root of the tree plus the optimal cover for the sub-trees starting at each input of the match
Example

Step 1.

Step 2.

Step 3.

Step 4.

Cover with ND2 or ND3?

1 NAND2 + subtree = 4
Area cost 8

1 NAND3 = 4

Cover with INV or AO21?

1 A021 + subtree 1 = 4
+ subtree 2 = 3
Area cost 13

Area cost 9

Example (III)

[Diagrams and text from the source]

Standard Cell Libraries
Standard Cell Libraries

- Collection of well-defined and appropriately characterized logic gates
- Standard cells must meet predefined specifications to be manipulated by synthesis, place and route tools
- Delivered with a collection of files that provide all the information needed by the various EDA tools
Example: Inverter

- Cell height
- Cell width
- Voltage rails
- Pin placement
- Metal layers
- Well definition

Cell Library Contents

- Combinational logic cells
  - NAND, NOR, INV...
  - Complex cells: AOI...
  - Cells with fan-in <= 4
- Sequential cells
  - Many types of flip-flops
  - Latches
  - Integrated clock gating cells

Multiple Threshold

- Most libraries provide equivalent cells with three VTs:
  - **HVT**: High threshold voltage, less power and higher delay
  - **LVT**: Low threshold voltage, shorter delay but more power
  - **SVT**: Standard threshold voltage, moderate delay and power
- All thresholds can be swapped without any placement/routing

Files in a cell library

- **Behavioral files**
  - Verilog description for simulation and logic equivalence
- **Physical views**
  - Layout of the cells (GDSII format) for DRC, LVS
  - Abstract of the cells (LEF format) for P&R, RC extraction
- **Transistor level**
  - Spice netlist for LVS, transistor-level simulation
- **Timing/power**
  - Liberty files with characterization of timing and power for STA
- **Others**
  - Symbols for displaying the cell in various tools
Liberty Exchange Format (LEF)

- Abstract description of the layout for P&R
  - Readable in ASCII format
  - Contains detailed PIN information for connecting
- LEF files contain information about the technology for use by the place and router
  - Metal layers
  - Design rules

Liberty Exchange Format (LEF)

Layout

Abstract

Physical cell size

Terminals with physical placement

Obstructions

LEF

MACRO IV
CLASS CORE ;
FOREIGN IV 0.000 0.000 ;
ORIGIN 0.00 0.00 ;
SIZE 3.00 BY 12.00 ;
SYMMETRY x y ;
SITE CORE ;
PIN A
DIRECTION INPUT ;
ANTENNASIZE 1.4 ;
PORT
LAYER metall ;
RECT 0.50 5.00 1.00 5.50
;
END
END A
;
OBS
LAYER metall ;
RECT 1.90 6.50 2.60 7.20
;
RECT 0.40 4.90 1.00 5.60
;

Liberty Format (.lib)

- Liberty file provides timing and power for a single corner: process, voltage, temperature...
- Includes information about each standard cell: function, area...
- Timing model for each cell
  - Propagation delay ($t_{pd}$)
  - Output transition ($t_{rise}$, $t_{fall}$)
Open Source Cell Libraries

- Oklahoma State University (OSU) library
  - 500nm, 350nm and 180nm
  - Available in qflow
- FreePDK45 (45nm)
  - https://www.eda.ncsu.edu/wiki/FreePDK45:Contents
- FreePDK15 (15nm)
  - https://www.eda.ncsu.edu/wiki/FreePDK15:Contents
- ASAP7 (7nm)
  - Arizona State University in partnership with ARM
  - http://asap.asu.edu/asap/
Yosys

- **Yosys Open SYNthesis Suite**
- Open-source framework for Verilog synthesis
  - Verilog-2005
  - Logic optimization on netlist
  - Mapping to ASIC standard cell libraries
- Synthesis process guided by script that contains Yosys commands
Yosys Data- and Control-Flow

- Three types of commands:
  - **Frontends**: read input files (Verilog)
  - **Passes**: perform transformation on the design
  - **Backends**: write the design to a file
Program Components and Data Formats

- Verilog Frontend
- VHDL Frontend
- Other Frontends

- AST
- AST Frontend

- RTLIL
- Passes

- Verilog Backend
- ILANG Backend
- Other Backends

Example Synthesis Script

# read design
read_verilog counter.v
hierarchy -check -top counter

# the high-level stuff
proc; opt; fsm; opt; memory; opt

# mapping to internal cell library
techmap; opt

# mapping flip-flops to mycells.lib
dfflibmap -liberty mycells.lib

# mapping logic to mycells.lib
abc -liberty mycells.lib

# cleanup
clean

# write synthesized design
write_verilog synth.v

Command: read_verilog counter.v
Read Verilog source file and convert to internal representation.
Example Synthesis Script

```plaintext
# read design
read_verilog counter.v
hierarchy -check -top counter

# the high-level stuff
proc; opt; fsm; opt; memory; opt

# mapping to internal cell library
technology -opt

# mapping flip-flops to mycells.lib
dfflibmap -liberty mycells.lib

# mapping logic to mycells.lib
abc -liberty mycells.lib

# cleanup
clean

# write synthesized design
write_verilog synth.v
```

Command: hierarchy -check -top counter

Elaborate the design hierarchy. Should always be the first command after reading the design. Can re-run AST front-end.
Example Synthesis Script

# read design
read_verilog counter.v
hierarchy -check -top counter

# the high-level stuff
proc; opt; fsm; opt; memory; opt

# mapping to internal cell library
technology

# mapping flip-flops to mycells.lib
dfflibmap -liberty mycells.lib

# mapping logic to mycells.lib
abc -liberty mycells.lib

# cleanup
clean

# write synthesized design
write_verilog synth.v

Command: proc

Convert “processes” (the internal representation of behavioral Verilog code) into multiplexers and registers.
Example Synthesis Script

# read design
read_verilog counter.v
hierarchy -check -top counter

# the high-level stuff
proc; opt; fsm; opt; memory; opt

# mapping to internal cell library
techmap; opt

# mapping flip-flops to mycells.lib
dfflibmap -liberty mycells.lib

# mapping logic to mycells.lib
abc -liberty mycells.lib

# cleanup
clean

# write synthesized design
write_verilog synth.v

Command: opt
Perform some basic optimizations and cleanups.

http://www.clifford.at/yosys/files/yosys_presentation.pdf
Example Synthesis Script

```
# read design
read_verilog counter.v
hierarchy -check -top counter

# the high-level stuff
proc; opt; fsm; opt; memory; opt

# mapping to internal cell library
techmap; opt

# mapping flip-flops to mycells.lib
dfflibmap -liberty mycells.lib

# mapping logic to mycells.lib
abc -liberty mycells.lib

# cleanup
clean

# write synthesized design
write_verilog synth.v
```

Command: fsm
Analyze and optimize finite state machines.

http://www.clifford.at/yosys/files/yosys_presentation.pdf
Example Synthesis Script

# read design
read_verilog counter.v
hierarchy -check -top counter

# the high-level stuff
proc; opt; fsm; opt; memory; opt

# mapping to internal cell library
technology; opt

# mapping flip-flops to mycells.lib
dfflibmap -liberty mycells.lib

# mapping logic to mycells.lib
abc -liberty mycells.lib

# cleanup
clean

# write synthesized design
write_verilog synth.v

Command: opt
Perform some basic optimizations and cleanups.
Example Synthesis Script

```
# read design
read_verilog counter.v
hierarchy -check -top counter

# the high-level stuff
proc; opt; fsm; opt; memory; opt

# mapping to internal cell library
techmap; opt

# mapping flip-flops to mycells.lib
dfflibmap -liberty mycells.lib

# mapping logic to mycells.lib
abc -liberty mycells.lib

# cleanup
clean

# write synthesized design
write_verilog synth.v
```

**Command: memory**

Analyze memories and create circuits to implement them.

http://www.clifford.at/yosys/files/yosys_presentation.pdf
Example Synthesis Script

# read design
read_verilog counter.v
hierarchy -check -top counter

# the high-level stuff
proc; opt; fsm; opt; memory; **opt**

# mapping to internal cell library
techmap; opt

# mapping flip-flops to mycells.lib
dfflibmap -liberty mycells.lib

# mapping logic to mycells.lib
abc -liberty mycells.lib

# cleanup
clean

# write synthesized design
write_verilog synth.v

**Command: opt**
Perform some basic optimizations and cleanups.
Example Synthesis Script

```
# read design
read_verilog counter.v
hierarchy -check -top counter

# the high-level stuff
proc; opt; fsm; opt; memory; opt

# mapping to internal cell library
techmap; opt

# mapping flip-flops to mycells.lib
dfflibmap -liberty mycells.lib

# mapping logic to mycells.lib
abc -liberty mycells.lib

# cleanup
clean

# write synthesized design
write_verilog synth.v
```

Command: techmap

Map coarse-grain RTL cells (adders, etc.) to fine-grain logic gates (AND, OR, NOT, etc.).
Example Synthesis Script

```bash
# read design
read_verilog counter.v
hierarchy -check -top counter

# the high-level stuff
proc; opt; fsm; opt; memory; opt

# mapping to internal cell library
techmap; opt

# mapping flip-flops to mycells.lib
dfflibmap -liberty mycells.lib

# mapping logic to mycells.lib
abc -liberty mycells.lib

# cleanup
clean

# write synthesized design
write_verilog synth.v
```

Command: opt
Perform some basic optimizations and cleanups.
Example Synthesis Script

```plaintext
# read design
read_verilog counter.v
hierarchy -check -top counter

# the high-level stuff
proc; opt; fsm; opt; memory; opt

# mapping to internal cell library
technology; opt

# mapping flip-flops to mycells.lib
dfflibmap -liberty mycells.lib

# mapping logic to mycells.lib
abc -liberty mycells.lib

# cleanup
clean

# write synthesized design
write_verilog synth.v
```

Command: dfflibmap -liberty mycells.lib
Map registers to available hardware flip-flops.

http://www.clifford.at/yosys/files/yosys_presentation.pdf
Example Synthesis Script

# read design
read_verilog counter.v
hierarchy -check -top counter

# the high-level stuff
proc; opt; fsm; opt; memory; opt

# mapping to internal cell library
techmap; opt

# mapping flip-flops to mycells.lib
dfflibmap -liberty mycells.lib

# mapping logic to mycells.lib
abc -liberty mycells.lib

# cleanup
clean

# write synthesized design
write_verilog synth.v

Command: abc -liberty mycells.lib
Map logic to available hardware gates.

http://www.clifford.at/yosys/files/yosys_presentation.pdf
# read design
read_verilog counter.v
hierarchy -check -top counter

# the high-level stuff
proc; opt; fsm; opt; memory; opt

# mapping to internal cell library
techmap; opt

# mapping flip-flops to mycells.lib
dfflibmap -liberty mycells.lib

# mapping logic to mycells.lib
abc -liberty mycells.lib

# cleanup
clean

# write synthesized design
write_verilog synth.v

Command: clean
Clean up the design (just the last step of opt).
Example Synthesis Script

```
# read design
read_verilog counter.v
hierarchy -check -top counter

# the high-level stuff
proc; opt; fsm; opt; memory; opt

# mapping to internal cell library
techmap; opt

# mapping flip-flops to mycells.lib
dfflibmap -liberty mycells.lib

# mapping logic to mycells.lib
abc -liberty mycells.lib

# cleanup
clean

# write synthesized design
write_verilog synth.v
```

Command: `write_verilog synth.v`

Write final synthesis result to output file.

http://www.clifford.at/yosys/files/yosys_presentation.pdf
Example: Verilog (counter.v)

module counter (clk, rst, en, count);

    input clk, rst, en;
    output reg [1:0] count;

    always @(posedge clk)
        if (rst)
            count <= 2’d0;
        else if (en)
            count <= count + 2’d1;

endmodule
Example: cell library (mycells.lib)

```plaintext
library(demo) {
  cell(BUF) {
    area: 6;
    pin(A) { direction: input; }
    pin(Y) { direction: output;
             function: "A"; }
  }
  cell(NOT) {
    area: 3;
    pin(A) { direction: input; }
    pin(Y) { direction: output;
             function: "A'"; }
  }
  cell(NAND) {
    area: 4;
    pin(A) { direction: input; }
    pin(B) { direction: input; }
    pin(Y) { direction: output;
             function: "(A*B)'"; }
  }
  cell(NOR) {
    area: 4;
    pin(A) { direction: input; }
    pin(B) { direction: input; }
    pin(Y) { direction: output;
             function: "(A+B)’"; }
  }
  cell(DFF) {
    area: 18;
    ff(IQ, IQN) { clocked_on: C;
                 next_state: D; }
    pin(C) { direction: input;
             clock: true; }
    pin(D) { direction: input; }
    pin(Q) { direction: output;
             function: "IQ"; }
  }

http://www.clifford.at/yosys/files/yosys_presentation.pdf
```
Example Synthesis Script

read_verilog counter.v
hierarchy -check -top counter

http://www.clifford.at/yosys/files/yosys_presentation.pdf
Example Synthesis Script

proc; opt; fsm; opt; memory; opt

http://www.clifford.at/yosys/files/yosys_presentation.pdf
Example Synthesis Script

techmap; opt
Example Synthesis Script

dfflibmap -liberty mycells.lib
abc -liberty mycells.lib
clean

http://www.clifford.at/yosys/files/yosys_presentation.pdf
Typical Phases of a Synthesis Flow

- Reading and elaborating the design
- Higher-level synthesis and optimization
  - Converting always-blocks to logic and registers
  - Perform coarse-grain optimizations (resource sharing, const folding…)
  - Handling memories and other coarse-grain blocks
  - Extracting and optimizing FSMs
- Convert remaining logic to bit-level logic functions
- Perform optimization on bit-level logic functions
- Map bit-level logic gates and registers to cell library
- Write results to output file

http://www.clifford.at/yosys/files/yosys_presentation.pdf
Reading and Elaboration

- Reading the design
  
  ```
  read_verilog file1.v
  read_verilog -I include_dir -D enable_foo -D WIDTH=12 file2.v
  read_verilog -lib cell_library.v
  ```

- Design elaboration
  
  - Yosys figures out how the modules are hierarchically connected
  - Re-runs the AST parts of the Verilog frontend to create all needed variations of parametric modules

# simplest form. at least this version should be used after reading all input files
#
# hierarchy

# recommended form. fails if parts of the design hierarchy are missing, removes # everything that is unreachable from the top module, and marks the top module.
#
# hierarchy -check -top top_module
Reading and Elaboration

- **verific** front-end
  - Parses SystemVerilog and VHDL
  - Not open-source, but free for non-commercial use
  - Bindings available in Yosys via **verific** command
  - Load SystemVerilog module
    - `verific {sv2005|-sv2009|-sv2012|-sv} <verilog-file>..
  - Load VHDL module
    - `verific {-vhdl87|-vhdl93|-vhdl2k|-vhdl2008|-vhdl} <vhdl-file>..`
The proc command

- Verilog frontend converts always-blocks to RTL netlists for the expressions and “processes” for the control and memory elements
- The proc command transforms “processes” to netlists of RTL multiplexer and register cells
- proc is the first command after elaboration, other commands cannot work on “processes”

```
proc_clean       # remove empty branches and processes
proc_rmdead      # remove unreachable branches
proc_init        # special handling of "initial" blocks
proc_arst        # identify modeling of async resets
proc_mux         # convert decision trees to multiplexer networks
proc_dff         # extract registers from processes
proc_clean       # if all went fine, this should remove all the processes
```

http://www.clifford.at/yosys/files/yosys_presentation.pdf
Example: proc command

```verilog
read_verilog proc_03.v
hierarchy -check -top test
proc;;
```

```verilog
module test(
    input A, B, C, D, E,
    output reg Y);

    always @* begin
        Y <= A;
        if (B)
            Y <= C;
        if (D)
            Y <= E;
    end
endmodule
```

http://www.clifford.at/yosys/files/yosys_presentation.pdf
The opt command

- Implements a series of optimizations
- Macro command that calls several optimization commands

```
begin
  opt_muxtree       # remove never-active branches from multiplexer tree
  opt_reduce        # consolidate trees of boolean ops to reduce functions
  opt_merge         # merging identical cells
  opt_rmdff         # remove/simplify registers with constant inputs
  opt_clean         # remove unused objects (cells, wires) from design
  opt_expr          # const folding and simple expression rewriting
  while [changed design]
```

http://www.clifford.at/yosys/files/yosys_presentation.pdf
Example: opt command

read_verilog opt_01.v
hierarchy -check -top test
opt

module test(input A, B, output Y);
assign Y = A ? A ? B : 1'b1 : B;
endmodule

http://www.clifford.at/yosys/files/yosys_presentation.pdf
Example: opt command

```plaintext
read_verilog opt_02.v
hierarchy -check -top test
opt
```

```plaintext
module test(input A, output Y, Z);
assign Y = A == A, Z = A != A;
endmodule
```

![Diagram](http://www.clifford.at/yosys/files/yosys_presentation.pdf)
Example: opt command

read_verilog opt_03.v
hierarchy -check -top test
opt

module test(input [3:0] A, B,
            output [3:0] Y, Z);
assign Y = A + B, Z = B + A;
endmodule

http://www.clifford.at/yosys/files/yosys_presentation.pdf
The memory command

- In RTL netlist, memory reads and memory writes are individual cells
- The memory command transforms individual read/write cells to a memory implementation
  - Includes logic for address decoders and registers
- It is also a macro that calls several commands

```plaintext
# this merges registers into the memory read- and write cells.
memory_dff

# this collects all read and write cells for a memory and transforms them
# into one multi-port memory cell.
memory_collect

# this takes the multi-port memory cell and transforms it to address decoder
# logic and registers. This step is skipped if "memory" is called with -nomap.
memory_map
```

http://www.clifford.at/yosys/files/yosys_presentation.pdf
Example: memory command

read_verilog memory_01.v
hierarchy -check -top test
proc;; memory; opt

module test(input  CLK, ADDR,
            input  [7:0] DIN,
            output reg [7:0] DOUT);
    reg [7:0] mem [0:1];
    always @(posedge CLK) begin
        mem[ADDR] <= DIN;
        DOUT <= mem[ADDR];
    end
endmodule

http://www.clifford.at/yosys/files/yosys_presentation.pdf
Example: memory command

module test(
    input WR1_CLK, WR2_CLK,
    input WR1_WEN, WR2_WEN,
    input [7:0] WR1_ADDR, WR2_ADDR,
    input [7:0] WR1_DATA, WR2_DATA,
    input RD1_CLK, RD2_CLK,
    input [7:0] RD1_ADDR, RD2_ADDR,
    output reg [7:0] RD1_DATA, RD2_DATA
);

reg [7:0] memory [0:255];

always @(posedge WR1_CLK)
    if (WR1_WEN)
        memory[WR1_ADDR] <= WR1_DATA;

always @(posedge WR2_CLK)
    if (WR2_WEN)
        memory[WR2_ADDR] <= WR2_DATA;

always @(posedge RD1_CLK)
    RD1_DATA <= memory[RD1_ADDR];

always @(posedge RD2_CLK)
    RD2_DATA <= memory[RD2_ADDR];

endmodule

read_verilog memory_02.v
hierarchy -check -top test
proc;; memory -nomap
opt -mux_UNDEF -mux_BOOL

http://www.clifford.at/yosys/files/yosys_presentation.pdf
The fsm command

- Identifies, extracts, optimizes (re-encodes) and re-synthesizes finite state machines
- It is also a macro that calls several commands

```bash
fsm_detect                      # unless got option -nodetect
fsm_extract

fsm_opt
clean
fsm_opt

fsm_expand                      # if got option -expand
   clean                      # if got option -expand
   fsm_opt                    # if got option -expand

fsm_recode                      # unless got option -norecode

fsm_info

fsm_export                      # if got option -export
fsm_map                         # unless got option -nomap
```

http://www.clifford.at/yosys/files/yosys_presentation.pdf
The techmap command

- Replaces cells with implementations given as Verilog source
- If used without a map file, it uses a built-in map file to map all RTL cells to a generic library of gates and registers

The built-in logic gate types are:

$\_NOT$  $\_AND$  $\_OR$  $\_XOR$  $\_MUX$

The register types are:

$\_SR\_NN$  $\_SR\_NP$  $\_SR\_PN$  $\_SR\_PP$
$\_DFF\_N$  $\_DFF\_P$
$\_DFF\_N0$  $\_DFF\_N1$  $\_DFF\_P0$  $\_DFF\_P1$
$\_DFF\_P0$  $\_DFF\_P1$  $\_DFF\_P0$  $\_DFF\_P1$
$\_DFFSR\_NNN$  $\_DFFSR\_NNP$  $\_DFFSR\_NPN$  $\_DFFSR\_NPP$
$\_DFFSR\_PNN$  $\_DFFSR\_PNP$  $\_DFFSR\_PPN$  $\_DFFSR\_PPP$
$\_DLATCH\_N$  $\_DLATCH\_P$
The abc command

• Provides an interface to ABC, an open source tool for low-level logic synthesis

• The abc command processes a netlist of internal gate types and performs:
  – Logic minimization (optimization)
  – Mapping of logic to a standard cell library (liberty format)
  – Mapping of logic to k-LUTs (for FPGA synthesis)

• dfplibmap command typically called before abc to map internal register cell types to the register types described in the liberty file
The abc command

- abc command can provide estimation of critical path delay
  - abc -D 10000 -constr synth.contr -liberty gscl45nm.lib

- Constraints file synth.constr:
  set_driving_cell INVX1
  set_load 0.015

- abc output:
  - ABC: Current delay (1609.00 ps) does not exceed the target delay (10000.00 ps).
The stat command

- Print some statistics on the selected area of the design

yosys> stat -liberty gscl45nm.lib
15. Printing statistics.

```plaintext
=== dotprod ===

<table>
<thead>
<tr>
<th>Description</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of wires:</td>
<td>687</td>
</tr>
<tr>
<td>Number of wire bits:</td>
<td>1926</td>
</tr>
<tr>
<td>Number of public wires:</td>
<td>6</td>
</tr>
<tr>
<td>Number of public wire bits:</td>
<td>70</td>
</tr>
<tr>
<td>Number of memories:</td>
<td>0</td>
</tr>
<tr>
<td>Number of memory bits:</td>
<td>0</td>
</tr>
<tr>
<td>Number of processes:</td>
<td>0</td>
</tr>
<tr>
<td>Number of cells:</td>
<td>521</td>
</tr>
<tr>
<td>AND2X1</td>
<td>45</td>
</tr>
<tr>
<td>AOI21X1</td>
<td>69</td>
</tr>
<tr>
<td>AOI22X1</td>
<td>9</td>
</tr>
<tr>
<td>BUFX2</td>
<td>22</td>
</tr>
<tr>
<td>DFFPOSX1</td>
<td>26</td>
</tr>
<tr>
<td>INVX1</td>
<td>17</td>
</tr>
<tr>
<td>MUX2X1</td>
<td>1</td>
</tr>
<tr>
<td>NAND2X1</td>
<td>86</td>
</tr>
<tr>
<td>NAND3X1</td>
<td>26</td>
</tr>
<tr>
<td>NOR2X1</td>
<td>27</td>
</tr>
<tr>
<td>NOR3X1</td>
<td>8</td>
</tr>
<tr>
<td>OAI21X1</td>
<td>51</td>
</tr>
<tr>
<td>OR2X1</td>
<td>28</td>
</tr>
<tr>
<td>XNOR2X1</td>
<td>66</td>
</tr>
<tr>
<td>XOR2X1</td>
<td>40</td>
</tr>
</tbody>
</table>

Chip area for module 'dotprod': 1631.286800
Bibliography

- Arto Perttula. VHDL Synthesis Basics.