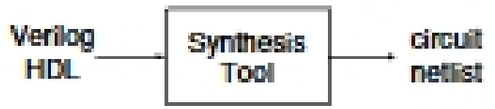
EECS150 - Digital Design
Lecture 6 - Synthesis

February 3, 2005
John Wawrzynek

Logic Synthesis

- Verilog and VHDL started out as simulation languages, but quickly people wrote programs to automatically convert Verilog code into low-level circuit descriptions (netlists).



- Synthesis converts Verilog (or other HDL) descriptions to implementation technology specific primitives:
  - For FPGAs: LUTs, flip-flops, and RAM blocks
  - For ASICs: standard cell gate and flip-flop libraries, and memory blocks.

Why Logic Synthesis?

1. Automatically manages many details of the design process:
   - → Fewer bugs
   - → Improved productivity
2. Abstracts the design data (HDL description) from any particular implementation technology.
   - Designs can be re-synthesized targeting different chip technologies. Ex: first implement in FPGA then later in ASIC.
3. In some cases, leads to a more optimal design than could be achieved by manual means (ex: logic optimization)

Why Not Logic Synthesis?

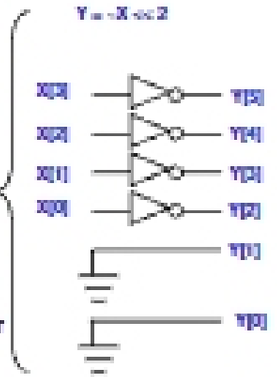
1. May lead to non-optimal designs in some cases.

How does it work?

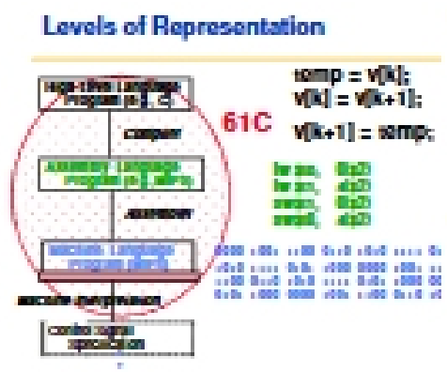
- A variety of general and ad-hoc (special case) methods:
  - **Instantiation**: maintains a library of primitive modules (AND, OR, etc.) and user defined modules.
  - **"macro expansion" / substitution**: a large set of language operators (+, -, Boolean operators, etc.) and constructs (if-else, case) expand into special circuits.
  - **Inference**: special patterns are detected in the language description and treated specially (ex: inferring memory blocks from variable declaration and read/write statements, FSM detection and generation from "always @ (posedge clk)" blocks).
  - **Logic optimization**: Boolean operations are grouped and optimized with logic minimization techniques.
  - **Structural reorganization**: advanced techniques including sharing of operators, and retiming of circuits (moving FFs), and others?

Operators

- Logical operators map into primitive logic gates
- Arithmetic operators map into adders, subtractors, ...
  - Unsigned 2s complement
  - Model carry: target is one-bit wider that source
  - Watch out for *, %, and /
- Relational operators generate comparators
- Shifts by constant amount are just wire connections
  - No logic involved
- Variable shift amounts a whole different story — shifter
- Conditional expression generates logic or MUX



Synthesis vs Compilation

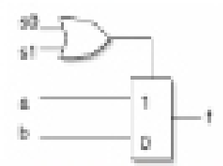


- Compiler
  - recognizes all possible constructs in a formally defined program language
  - translates them to a machine language representation of execution process
- Synthesis
  - Recognizes a target dependent subset of a hardware description language
  - Maps to collection of concrete hardware resources
  - Iterative tool in the design flow

Simple Example

```
module foo (a,b,s0,s1,f);
input [3:0] a;
input [3:0] b;
input s0,s1;
output [3:0] f;
reg f;
always @ (a or b or s0 or s1)
   if (!s0 && s1 || s0) f=a; else f=b;
endmodule
```

- Should expand if-else into 4-bit wide multiplexor and optimize the control logic:



Module Template

Synthesis tools expects to find modules in this format.

```
module [top_module_name] (input list);
// Port declarations: followed by wire, reg, integer, task and function declarations //
// Describe hardware with zero or more continuous assignments, always blocks, module
     instantiations and gate instantiations //
// Continuous assignment
wire [result] [signal_name];
assign [result] [signal_name] = [expression];
// always block
always @( [event expression] )
begin
// Procedural assignments
// if statements
// case, casex, and casez statements
// while, repeat and for loops
// user task and user function calls
end
// Module instantiation
[module_name] [instance_name] (input list);
// Instantiation of built in gate primitive
[gate_type_keyword] (input list);
endmodule
```

- The order of these statements is irrelevant, all execute concurrently.
- The statements between the begin and end in an always block execute sequentially from top to bottom. (However, beware of blocking versus non-blocking assignment!)
- Statements within a fork-join statement in an always block execute concurrently.

Procedural Assignments

- Verilog has two types of assignments within always blocks:
- Blocking procedural assignment "="
  - The RHS is executed and the assignment is completed before the next statement is executed. Example:

    Assume A holds the value 1 ... A=2; B=A; A is left with 2, B with 2.
- Non-blocking procedural assignment "<="
  - The RHS is executed and assignment takes place at the end of the current time step (not clock cycle). Example:

    Assume A holds the value 1 ... A<=2; B<=A; A is left with 2, B with 1.
- The notion of the "current time step" is tricky in synthesis, so to guarantee that your simulation matches the behavior of the synthesized circuit, follow these rules:

i. Use blocking assignments to model combinational logic within an always block.
ii. Use non-blocking assignments to implement sequential logic.
iii. Do not mix blocking and non-blocking assignments in the same always block.
iv. Do not make assignments to the same variable from more than one always block.

Supported Verilog Constructs

- Net types: wire, tri, supply1, supply0; register types: reg, integer, time (64 bit reg); arrays of reg
- Continuous assignments.
- Gate primitive and module instantiations.
- always blocks, user tasks, user functions.
- Inputs, outputs, and inouts to a module.
- All operators (+, -, *, /, %, <, >, <=, >=, ==, !=, ===, !==, &&, ||, !, ~, &, ~&, |, ~|, ^, ~^, ^~, <<, >>, ?:, {,}, {{}}) (Note: / and % are supported for compile-time constants and constant powers of 2.)
- Procedural statements: if-else-if, case, casex, casez, for, repeat, while, forever, begin, end, fork, join.
- Procedural assignments: blocking assignments =, nonblocking assignments <= (Note: <= cannot be mixed with = for the same register).
- Compiler directives: `define, `ifdef, `else, `endif, `include, `undef
- Miscellaneous:
  - Integer ranges and parameter ranges.
  - Local declarations to begin-end block.
  - Variable indexing of bit vectors on the left and right sides of assignments.

Unsupported Language Constructs

Generate error and halt synthesis:

- Net types: trireg, wor, trior, wand, triand, tri0, tri1, and charge strength;
- register type: real.
- Built-in switches and bidirectional switches, and pull-up, pull-down.
- Procedural statements: assign (different from the "continuous assignment"), deassign, wait.
- Named events and event triggers.
- UDPs (user defined primitives) and specify blocks.
- force, release, and hierarchical net names (for simulation only).

Simply ignored

- delay, delay control, and drive strength.
- scalared, vectored.
- initial block.
- Compiler directives (except for `define, `ifdef, `else, `endif, `include, and `undef, which are supported).
- Calls to system tasks and system functions (they are only for simulation).

Combinational Logic

CL can be generated using:

1. primitive gate instantiation:

   &and, &or, etc.
2. continuous assignment (assign keyword), example:

```
Module: adder_8 (result, sum, a, b, sin);
output: result;
output: [7:0] sum;
input: sin;
input: [7:0] a, b;
assign: {result, sum} = a + b + sin;
endmodule
```

3. Always block:

```
always @ (event_expression)
begin
  // procedural assignment statements, if statements,
  // case statements, while, repeat, and for loops.
  // Task and function calls
end
```