Advanced AHDL

Kevin Schaffer
Kent State University
Spring 2006

Identifiers

- Composed of legal characters
  - Letters (a–z, A–Z)
  - Digits (0–9)
  - Slash (/)
  - Underscore (_)
- Can begin with a digit
- Must not be a reserved word

Boolean Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Alternate</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>!</td>
<td>NOT</td>
<td>Inverter</td>
</tr>
<tr>
<td>&amp;</td>
<td>AND</td>
<td>AND</td>
</tr>
<tr>
<td>! &amp;</td>
<td>NAND</td>
<td>AND with Inverted Output</td>
</tr>
<tr>
<td>#</td>
<td>OR</td>
<td>OR</td>
</tr>
<tr>
<td>! #</td>
<td>NOR</td>
<td>OR with Inverted Output</td>
</tr>
<tr>
<td>$</td>
<td>XOR</td>
<td>Exclusive OR</td>
</tr>
<tr>
<td>! $</td>
<td>XNOR</td>
<td>Exclusive OR with Inverted Output</td>
</tr>
</tbody>
</table>

Using Boolean Operators (1)

- \( C = A \& B \)
- If both operands are single nodes, applies the operation to those nodes
- Constants GND and VCC act as single nodes
Using Boolean Operators (2)

- \( C[] = A[] \land B[] \)
- If both operands are groups, the operation is applied to corresponding group members
- Both groups must be the same size

Using Boolean Operators (3)

- \( C[] = A[] \land B \)
- If one operand is a group and the other a single node, the single node is replicated to form a group

Arithmetic Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ (unary)</td>
<td>Identity</td>
</tr>
<tr>
<td>- (unary)</td>
<td>Two's Complement Negation</td>
</tr>
<tr>
<td>+</td>
<td>Unsigned/Two's Complement Addition</td>
</tr>
<tr>
<td>-</td>
<td>Unsigned/Two's Complement Subtraction</td>
</tr>
</tbody>
</table>

Using Arithmetic Operators

- Operands must be groups or numbers
- If both operands are groups, they must be the same size
- If one operand is a group and the other a number, the number is sign-extended to match the size of the group
Comparison Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>==</td>
<td>Logical</td>
<td>Equal to</td>
</tr>
<tr>
<td>!=</td>
<td>Logical</td>
<td>Not equal to</td>
</tr>
<tr>
<td>&lt;</td>
<td>Arithmetic</td>
<td>Less than</td>
</tr>
<tr>
<td>&lt;=</td>
<td>Arithmetic</td>
<td>Less than or equal to</td>
</tr>
<tr>
<td>&gt;</td>
<td>Arithmetic</td>
<td>Greater than</td>
</tr>
<tr>
<td>&gt;=</td>
<td>Arithmetic</td>
<td>Greater than or equal to</td>
</tr>
</tbody>
</table>

Operator Precedence

<table>
<thead>
<tr>
<th>Precedence</th>
<th>Operators</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 (Highest)</td>
<td>Identity (+), Negation (−), NOT (!)</td>
</tr>
<tr>
<td>2</td>
<td>Addition (+), Subtraction (−)</td>
</tr>
<tr>
<td>3</td>
<td>Comparison (==, !=, &lt;, &lt;=, &gt;, &gt;=)</td>
</tr>
<tr>
<td>4</td>
<td>AND (&amp;), NAND (! &amp;)</td>
</tr>
<tr>
<td>5</td>
<td>XOR ($), XNOR (! $)</td>
</tr>
<tr>
<td>6 (Lowest)</td>
<td>OR (#), NOR (! #)</td>
</tr>
</tbody>
</table>

Primitives

- Buffer primitives
  - Logically do nothing
  - Give hints to synthesis tools
- Tristate (TRI)
- Flip-flops and latches

Flip-Flops and Latches

<table>
<thead>
<tr>
<th>Primitive</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LATCH</td>
<td>Data (D) Latch</td>
</tr>
<tr>
<td>DFF/DFFE</td>
<td>Data (D) Flip-Flop</td>
</tr>
<tr>
<td>JKFF/JKFFE</td>
<td>JK Flip-Flop</td>
</tr>
<tr>
<td>SRFF/SRFFE</td>
<td>Set/Reset (SR) Flip-Flop</td>
</tr>
<tr>
<td>TFF/TFFE</td>
<td>Toggle (T) Flip-Flop</td>
</tr>
</tbody>
</table>
**Data Latch**

- Primitive: **LATCH**
- Inputs
  - Data Input (d)
  - Enable (ena)
- Outputs
  - Data Output (q)

**Data Flip-Flop**

- Primitive: **DFF or DFFE**
- Inputs
  - Data Input (d)
  - Clock (clk)
  - Asynchronous Clear (clrn) and Set (prn)
  - Enable (ena), DFFE only
- Outputs
  - Data Output (q)

**JK Flip-Flop**

- Primitive: **JKFF or JKFFE**
- Inputs
  - Set (j) and Reset (k)
  - Clock (clk)
  - Asynchronous Clear (clrn) and Set (prn)
  - Enable (ena), JKFFE only
- Outputs
  - Data Output (q)

**Set/Reset Flip-Flop**

- Primitive: **SRFF or SRFFE**
- Inputs
  - Set (s) and Reset (r)
  - Clock (clk)
  - Asynchronous Clear (clrn) and Set (prn)
  - Enable (ena), SRFFE only
- Outputs
  - Data Output (q)
**Toggle Flip-Flop**

- **Primitive:** TFF or TFFE
- **Inputs**
  - Toggle (t)
  - Clock (clk)
  - Asynchronous Clear (clrn) and Set (prn)
  - Enable (ena), TFFE only
- **Outputs**
  - Data Output (q)

**Multiphase Clock Example**

- Generates 16 non-overlapping clock signals
- Built from a 4-bit counter and 4-to-16 decoder
- Uses standard macrofunctions provided with MAX+PLUS II

**Grouping Nodes**

- Combine nodes (and other groups) into a group by placing the node/group names in parentheses
  - g[3..0] = (a, b, c, d);
  - (a, b, c, d) = g[3..0];
- Can also be used to assign multiple ports in a single statement
  - comp.(a, b) = B"10";

**Multiphase (Include Files)**

- 4-bit Counter (4count.inc)
  - FUNCTION 4count(clk, clrn, setn, ldn, cin, dnum, d, c, b, a)
    - RETURNS (qd, qc, qb, qa, cout);
- 4-to-16 Decoder (16dmux.inc)
  - FUNCTION 16dmux(d, c, b, a)
    - RETURNS (q[15..0]);
### Multiphase (Subdesign)

```
INCLUDE "4count";
INCLUDE "16dmux";

SUBDESIGN multiphase_clock
(
  clk: INPUT;
  out[15..0]: OUTPUT;
)
```

### Multiphase (Logic)

```
VARIABLE
  counter: 4count;
  decoder: 16dmux;

BEGIN
  counter.clk = clk;
  counter.dnup = GND;
  decoder.(d,c,b,a) = counter.(qd,qc,qb,qa);
  out[15..0] = decoder.q[15..0];
END;
```

### Multiphase (GDF)

```
VARIABLE
  q[3..0]: NODE;

BEGIN
  (q[3..0], ) = 4count(clk,,,,, GND,,,,);
  out[15..0] = 16dmux(q[3..0]);
END
```

### Multiphase (Alternate Logic)

```
VARIABLE
  q[3..0]: NODE;

BEGIN
  (q[3..0], ) = 4count(clk,,,,, GND,,,,);
  out[15..0] = 16dmux(q[3..0]);
END
```
Parameters

- Parameters allow a component to be customized at instantiation
- Can create generic components that can be used in a number of contexts
- Examples
  - n-bit Register
  - n-to-$2^n$ Decoder

n-bit Register (Subdesign)

PARAMETERS
  (  WIDTH = 8  )
)

SUBDESIGN regn
  (  clk, d[(WIDTH - 1)..0]: INPUT;
     q[(WIDTH - 1)..0]: OUTPUT;
  )

n-bit Register (Logic)

VARIABLE
  ff[(WIDTH - 1)..0]: DFFE;

BEGIN
  ff[].clk = clk;
  ff[].d = d[];
  q[] = ff[].q;
END;

Using the n-bit Register

- Prototype
  - FUNCTION regn(clk, d[(WIDTH - 1)..0])
    WITH (WIDTH)
    RETURNS (q[(WIDTH - 1)..0]);
- Instantiate in variable section
  - reg4: regn WITH (WIDTH = 4);
- Instantiate inline
  - q[3..0] = regn(clk, d[3..0])
    WITH (WIDTH = 4);
Extended Arithmetic

- Expressions based on constants and/or parameters can use additional operators and functions
  - Operators: *, ^, DIV, MOD
  - Functions: CEIL, FLOOR, LOG2
- These cannot be applied to nodes or groups
- String comparison is also possible

Bits and Values

- The exponentiation operator (^) and the CEIL and LOG2 functions come in handy when writing parameterized designs
- To determine the number of bits necessary to represent a number N, use:
  - CEIL(LOG2(N + 1))
- The maximum value that can be represented with K bits is 2^K - 1

Clock Divider (Subdesign)

PARAMETERS
(
  DIVISOR = 25000000
);
CONSTANT WIDTH = CEIL(LOG2(DIVISOR));

SUBDESIGN clock_divider
(
  clk: INPUT;
  clk_out: OUTPUT;
)
If Generate

- Conditionally compile a section of the design
- Evaluates the condition at compile time, rather than in the hardware
- As such, condition must be based on constants and/or parameters

IF cond THEN GENERATE
  statements
ELSE GENERATE
  statements
END GENERATE;

If Generate Example

BEGIN
  ...
  IF SIMULATION == "YES" THEN GENERATE
    pb_debounced = pb;
  ELSE GENERATE
    pb_debounced = debounce(clk, pb);
  END GENERATE;
  ...
END;

For Generate

- Repeats a section of the design for each integer in a range
- Range must be resolved at compile time (constants and parameters only)

FOR i IN m TO n GENERATE
  statements
END GENERATE;

Decoder (Subdesign)

PARAMETERS (NUM_INPUTS);
CONSTANT NUM_OUTPUTS = 2 ^ NUM_INPUTS;

SUBDESIGN decoder
{
  data[(NUM_INPUTS - 1)..0]: INPUT;
  result[(NUM_OUTPUTS - 1)..0]: OUTPUT;
}
Decoder (Logic)

BEGIN
  FOR i IN 0 TO NUM_OUTPUTS - 1 GENERATE
    result[i] = (data[] == i);
  END GENERATE;
END;

LPM Gates

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>lpm_and</td>
<td>Parameterized AND gate</td>
</tr>
<tr>
<td>lpm_bustri</td>
<td>Parameterized tristate buffer</td>
</tr>
<tr>
<td>lpm_clshift</td>
<td>Parameterized combinational logic shifter</td>
</tr>
<tr>
<td>lpm_constant</td>
<td>Parameterized constant generator</td>
</tr>
<tr>
<td>lpm_decode</td>
<td>Parameterized decoder</td>
</tr>
<tr>
<td>lpm_inv</td>
<td>Parameterized inverter</td>
</tr>
<tr>
<td>lpm_mux</td>
<td>Parameterized multiplexer</td>
</tr>
<tr>
<td>lpm_or</td>
<td>Parameterized OR gate</td>
</tr>
<tr>
<td>lpm_xor</td>
<td>Parameterized XOR gate</td>
</tr>
</tbody>
</table>

LPM

- Library of Parameterized Modules (LPM)
- Gates
  - AND, OR, NOT
- Arithmetic
  - Adders, subtractors, multipliers
- Storage elements
  - Flip-flops, shift registers
  - RAMs, ROMs

LPM Arithmetic

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>lpm_abs</td>
<td>Parameterized absolute value function</td>
</tr>
<tr>
<td>lpm_add_sub</td>
<td>Parameterized adder/subtractor</td>
</tr>
<tr>
<td>lpm_compare</td>
<td>Parameterized comparator</td>
</tr>
<tr>
<td>lpm_counter</td>
<td>Parameterized counter</td>
</tr>
<tr>
<td>lpm_mult</td>
<td>Parameterized multiplier</td>
</tr>
</tbody>
</table>
LPM Storage

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>lpm_ff</td>
<td>Parameterized flip-flop</td>
</tr>
<tr>
<td>lpm_latch</td>
<td>Parameterized latch</td>
</tr>
<tr>
<td>lpm_ram_dq</td>
<td>Parameterized RAM with separate input and output ports</td>
</tr>
<tr>
<td>lpm_ram_io</td>
<td>Parameterized RAM with a single I/O port</td>
</tr>
<tr>
<td>lpm_io</td>
<td>Parameterized ROM</td>
</tr>
<tr>
<td>lpm_shiftreg</td>
<td>Parameterized shift register</td>
</tr>
</tbody>
</table>

Multiplier (Logic)

VARIABLE
   mult: lpm_mult WITH (LPM_WIDTHA = 8,
                       LPM_WIDTHB = 8,
                       LPM_WIDTHP = 16);
BEGIN
   mult.dataa[] = multiplicand[];
   mult.datab[] = multiplier[];
   product[] = mult.result[];
END;

Multiplier (Subdesign)

INCLUDE "lpm_mult";

SUBDESIGN mult8x8
(    multiplicand[7..0]: INPUT;
    multiplier[7..0]: INPUT;
    product[15..0]: OUTPUT;
)