Advanced AHDL

Kevin Schaffer
Kent State University
Spring 2006

Identifiers

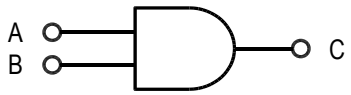
- Composed of legal characters
  - Letters (a–z, A–Z)
  - Digits (0–9)
  - Slash (/)
  - Underscore (_)
- Can begin with a digit
- Must not be a reserved word

Boolean Operators

Operator	Alternate	Description
!	NOT	Inverter
&	AND	AND
! &	NAND	AND with Inverted Output
#	OR	OR
! #	NOR	OR with Inverted Output
$	XOR	Exclusive OR
! $	XNOR	Exclusive OR with Inverted Output

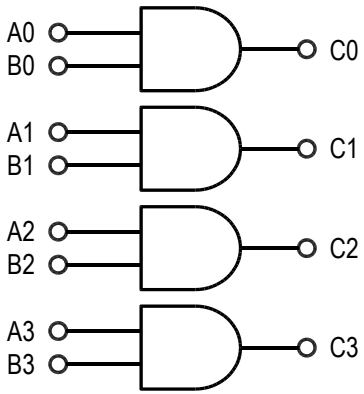
Using Boolean Operators (1)

- `C = A & B`
- If both operands are single nodes, applies the operation to those nodes
- Constants **GND** and **VCC** act as single nodes



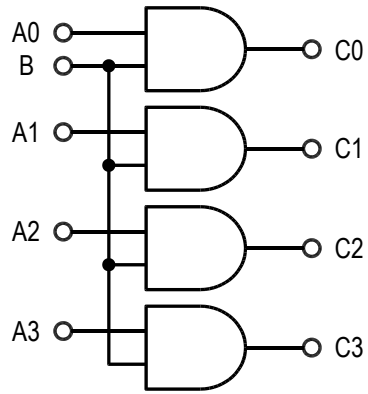
Using Boolean Operators (2)

- `C[] = A[] & 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[] & B`
- If one operand is a group and the other a single node, the single node is replicated to form a group



Arithmetic Operators

Operator	Description
+ (unary)	Identity
− (unary)	Two's Complement Negation
+	Unsigned/Two's Complement Addition
−	Unsigned/Two's Complement Subtraction

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

Operator	Type	Description
==	Logical	Equal to
!=	Logical	Not equal to
<	Arithmetic	Less than
<=	Arithmetic	Less than or equal to
>	Arithmetic	Greater than
>=	Arithmetic	Greater than or equal to

Operator Precedence

Precedence	Operators
1 (Highest)	Identity (+), Negation (–), NOT (!)
2	Addition (+), Subtraction (–)
3	Comparison (==, !=, <, <=, >, >=)
4	AND (&), NAND (!&)
5	XOR ($), XNOR (!$)
6 (Lowest)	OR (#), NOR (!#)

Primitives

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

Flip-Flops and Latches

Primitive	Description
LATCH	Data (D) Latch
DFF/DFFE	Data (D) Flip-Flop
JKFF/JKFFE	JK Flip-Flop
SRFF/SRFFE	Set/Reset (SR) Flip-Flop
TFF/TFFE	Toggle (T) Flip-Flop

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`**)

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 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

Multiphase (Include Files)

- 4-bit Counter (4count.inc)
  - `FUNCTION 4count(clk, clrn, setn, ldn, cin, dnup, 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]);`

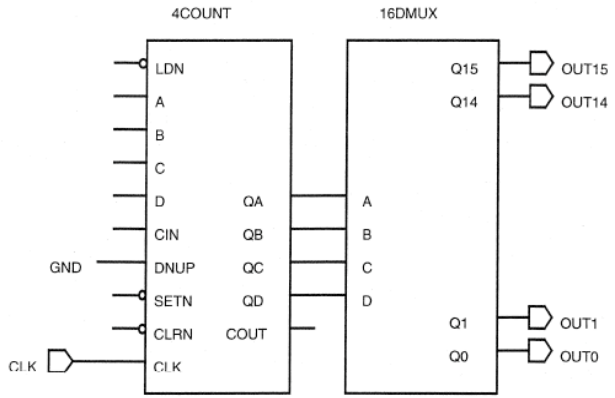
Multiphase (Subdesign)

```
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)



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;
)
```


Clock Divider (Logic)

```
VARIABLE
  counter[(WIDTH - 1)..0]: DFF;
BEGIN
  counter[].clk = clk;
  IF counter[].q == 0 THEN
    counter[].d = DIVISOR - 1;
    clk_out = 1;
  ELSE
    counter[].d = counter[].q - 1;
    clk_out = 0;
  END IF;
END;
```

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

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

LPM Gates

Name	Description
`lpm_and`	Parameterized AND gate
`lpm_bustri`	Parameterized tristate buffer
`lpm_clshift`	Parameterized combinational logic shifter
`lpm_constant`	Parameterized constant generator
`lpm_decode`	Parameterized decoder
`lpm_inv`	Parameterized inverter
`lpm_mux`	Parameterized multiplexer
`lpm_or`	Parameterized OR gate
`lpm_xor`	Parameterized XOR gate

LPM Arithmetic

Name	Description
`lpm_abs`	Parameterized absolute value function
`lpm_add_sub`	Parameterized adder/subtractor
`lpm_compare`	Parameterized comparator
`lpm_counter`	Parameterized counter
`lpm_mult`	Parameterized multiplier

LPM Storage

Name	Description
lpm_ff	Parameterized flip-flop
lpm_latch	Parameterized latch
lpm_ram_dq	Parameterized RAM with separate input and output ports
lpm_ram_io	Parameterized RAM with a single I/O port
lpm_rom	Parameterized ROM
lpm_shiftreg	Parameterized shift register

Multiplier (Subdesign)

```
INCLUDE "lpm_mult";

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


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;
```