Introduction to Design Using AHDL

Kevin Schaffer

Kent State University

Spring 2006

HDLs

- For large designs, hardware description languages (HDLs) are usually better than schematic capture
- Permits both behavioral and structural description of logic
- HDLs are concurrent, programming languages are sequential!

AHDL

- Altera Hardware Description Language
- Proprietary HDL for use with Altera's field-programmable logic devices and tools
- Simple syntax, easy to learn
- AHDL files have a TDF extension

Subdesign Section

- Declares component's name
  - Component name and file name must match
- Declares ports
  - Name
  - Input, output, or bidirectional
  - Width (if it is a group)
- Required

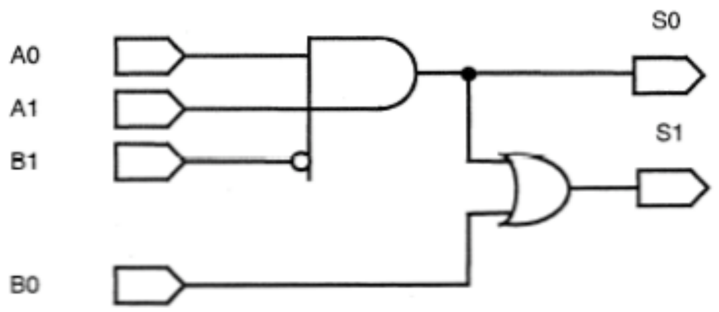
Logic Section

- Defines the logical operation of the component
- Last section in the TDF file
- Required

Combinational Logic (TDF)

```
SUBDESIGN bool1
(
  a0, a1, b0, b1: INPUT;
  s0, s1: OUTPUT;
)
BEGIN
  s0 = a0 & a1 & !b1;
  s1 = s0 # b0;
END;
```

Combinational Logic (GDF)



AHDL Syntax

- Identifiers and keywords are case-insensitive (unlike C/C++)
- Whitespace is not significant
- Statements end with a semicolon (`;`)
- Put comments between percent signs (`%`)

Boolean Expressions

- Operators
  - AND (`&`)
  - OR (`#`)
  - NOT (`!`)
- Constants
  - **VCC**
  - **GND**

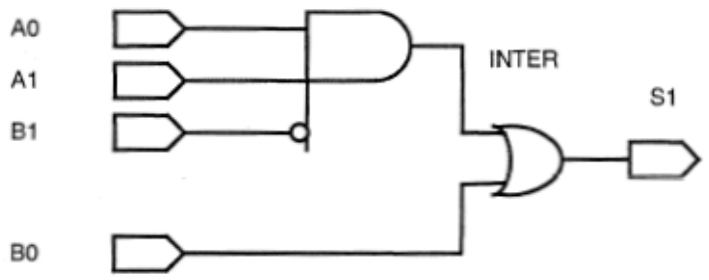
Variables Section

- Declares internal named nodes
- Declares instances of primitives and macrofunctions
- Must follow subdesign section
- Optional

Named Node (TDF)

```
SUBDESIGN bool2
(
  a0, a1, b0, b1: INPUT;
  s1: OUTPUT;
)
VARIABLE
  inter: NODE;
BEGIN
  inter = a0 & a1 & !b1;
  s1 = inter # b0;
END;
```

Named Node (GDF)



Groups

- Multiple bits treated as a single unit
- Range of indexes goes after the group name enclosed in square brackets
  - `a[15..0]`
- Use [ ] as shorthand for the entire group
- Interpreted as unsigned binary numbers in arithmetic operations

Group Operations

- Assignment
  - `a[15..8] = b[7..0];`
  - `a[] = b[];`
- Assignment from a single node
  - `a[4..0] = inter;`
  - `a[] = VCC;`
- Arithmetic
  - `c[15..0] = a[15..0] + b[15..0];`

Address Decoder

```
SUBDESIGN decode
(
  address[15..0]: INPUT;
  chip_enable1, chip_enable2: OUTPUT;
)
BEGIN
  chip_enable1 = (address[15..0] == H"FF30");
  chip_enable2 = (address[15..0] == H"FF50");
END;
```

Constants

- Literals
  - Decimal: **`65328`**
  - Hexadecimal: **`H"FF30"`**
  - Binary: **`B"1111111100110000"`**
- Can be assigned to groups or used with group arithmetic operators
- Use **`CONSTANT`** statement to declare named constants

Named Constants

```
CONSTANT IO_ADDRESS1 = H"FF30";
CONSTANT IO_ADDRESS2 = H"FF50";

SUBDESIGN decode
(
  address[15..0]: INPUT;
  chip_enable1, chip_enable2: OUTPUT;
)
BEGIN
  chip_enable1 = (address[] == IO_ADDRESS1);
  chip_enable2 = (address[] == IO_ADDRESS2);
END;
```

Conditional Logic

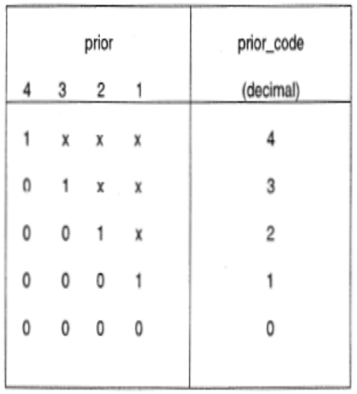
- **`IF`** statement
  - Select operation based on a set of Boolean expressions
  - Similar to **`if`** in C/C++
- **`CASE`** statement
  - Select operation based on the value of a group/node
  - Similar to **`switch`** in C/C++

Multiplexor

```
SUBDESIGN mux
(
  a, b, sel: INPUT;
  result: OUTPUT;
)
BEGIN
  IF sel == 0 THEN
    result = a;
  ELSE
    result = b;
  END IF;
END;
```

Priority Encoder (1)

```
SUBDESIGN priority
(
  prior4: INPUT;
  prior3: INPUT;
  prior2: INPUT;
  prior1: INPUT;
  prior_code[2..0]:
    OUTPUT;
)
```



prior				prior_code
4	3	2	1	(decimal)
1	x	x	x	4
0	1	x	x	3
0	0	1	x	2
0	0	0	1	1
0	0	0	0	0

Priority Encoder (2)

```
IF prior4 THEN
    prior_code[] = 4;
ELSIF prior3 THEN
    prior_code[] = 3;
ELSIF prior2 THEN
    prior_code[] = 2;
ELSIF prior1 THEN
    prior_code[] = 1;
ELSE
    prior_code[] = 0;
END IF;
```

Binary to One-Hot Decoder

```
SUBDESIGN decoder_2_to_4
( inpcode[1..0]: INPUT;
  outcode[3..0]: OUTPUT; )
BEGIN
  CASE inpcode[] IS
    WHEN 0 => outcode[] = B"0001";
    WHEN 1 => outcode[] = B"0010";
    WHEN 2 => outcode[] = B"0100";
    WHEN 3 => outcode[] = B"1000";
  END CASE;
END;
```

Truth Tables

- Map arbitrary input patterns to outputs
- First line declares which inputs and outputs the table maps
- Subsequent lines define the input patterns and corresponding outputs
- Input patterns can contain "don't cares" (x)

Decoder with Truth Table

```
SUBDESIGN decoder
( inp[1..0]: INPUT;
  a, b, c, d: OUTPUT; )
BEGIN
  TABLE
    inp[1..0] => a, b, c, d;
    H"0"      => 1, 0, 0, 0;
    H"1"      => 0, 1, 0, 0;
    H"2"      => 0, 0, 1, 0;
    H"3"      => 0, 0, 0, 1;
  END TABLE;
END;
```

Advanced Address Decoder

```
SUBDESIGN decode2
( addr[15..0], m/io: INPUT;
  rom, ram, print, inp[2..1]: OUTPUT; )
BEGIN
  TABLE
    m/io, addr[15..0] => rom, ram, print, inp[];
    1, B"00xxxxxxxxxxxxxx" => 1, 0, 0, B"00";
    1, B"10xxxxxxxxxxxxxx" => 0, 1, 0, B"00";
    0, B"0000001010000000" => 0, 0, 0, B"00";
    0, B"0000110100010000" => 0, 0, 0, B"01";
  END TABLE;
END;
```

Primitives & Macrofunctions

- Primitives are built-in components
  - Examples: **`TRI`**, **`DFF`**
- Macrofunctions are user-defined components
- Can be instantiated in the variables section or inline in a Boolean expression (like a function)
- Macrofunctions must be declared, typically using an include file

Instantiating Macrofunctions

- Declare the macrofunction instance in the variables section
  - `ff: DFF;`
- Connect the ports in the logic section
  - `ff.d = data_in;`
  - `data_out = ff.q;`

1-bit Register

```
SUBDESIGN reg1
(
  clk, load, d: INPUT; q: OUTPUT;
)
VARIABLE
  ff: DFFE;
BEGIN
  ff.clk = clk;
  ff.ena = load;
  ff.d = d;
  q = ff.q;
END;
```

Inline Macrofunctions

- You can instantiate and connect a macrofunction within an expression
  - **`data_out = DFF(data_in, clk, , );`**
- Unused inputs can be left out, but the commas are required

1-bit Register (Inline)

```
SUBDESIGN reg1b
(
  clk, load, d: INPUT;
  q: OUTPUT;
)
BEGIN
  q = DFFE(d, clk, , , load);
END;
```

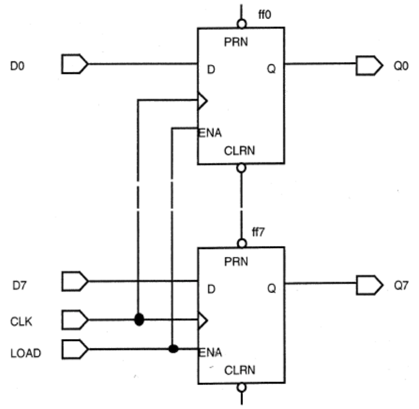
Groups of Macrofunctions

- You can instantiate a group of macrofunctions just like a group of nodes
  - **`ff[7..0]: DFFE;`**
- Then connect all the ports of a group at once
  - **`ff[7..0].clk = clk;`**
  - **`ff[7..0].d = data_in[7..0];`**
- Pay attention to where the [ ]'s go

8-bit Register (TDF)

```
SUBDESIGN register
(
  clk, load, d[7..0]: INPUT; q[7..0]: OUTPUT;
)
VARIABLE
  ff[7..0]: DFFE;
BEGIN
  ff[].clk = clk;
  ff[].ena = load;
  ff[].d = d[];
  q[] = ff[].q;
END;
```

8-bit Register (GDF)



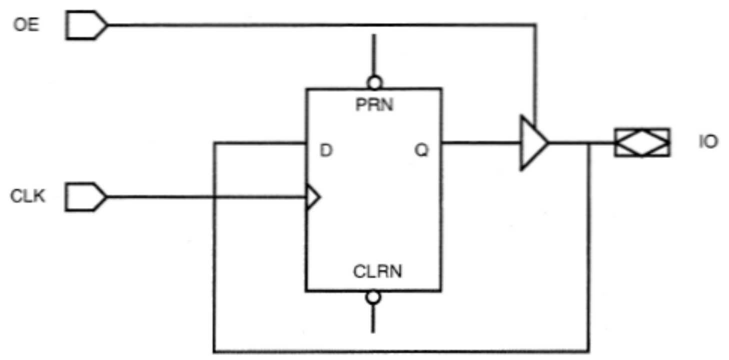
Bidirectional Ports

- Can be used as both inputs and outputs
- Driven by a tristate buffer (**`TRI`**)
  - Tristate buffer has an output enable input
  - When output enable is low, buffer goes into a high-impedance state
- Internal nodes cannot be bidirectional

Bus Register (TDF)

```
SUBDESIGN bus_reg
(
  clk, oe: INPUT;
  io: BIDIR;
)
BEGIN
  io = TRI(DFF(io, clk, , ), oe);
END;
```

Bus Register (GDF)



Declaring Macrofunctions

- Macrofunctions (but not primitives) must be declared before use
  - `FUNCTION bus_reg(clk, oe) RETURNS (io);`
- Typically the declaration is placed in an include (.inc) file and then **`INCLUDE`**'d in a TDF

4-bit Bidirectional Port

```
FUNCTION bus_reg(clk, oe) RETURNS (io);

SUBDESIGN bidir
(
  clk, oe: INPUT; io[3..0]: OUTPUT;
)
BEGIN
  io0 = bus_reg(clk, oe);
  io1 = bus_reg(clk, oe);
  io2 = bus_reg(clk, oe);
  io3 = bus_reg(clk, oe);
END;
```

Registered Outputs

```
SUBDESIGN reg_out
(
  clk, load, d[7..0]: INPUT;
  q[7..0]: OUTPUT;
)
VARIABLE
  q[7..0]: DFFE;
BEGIN
  q[].clk = clk;
  q[].ena = load;
  q[] = d[];
END;
```

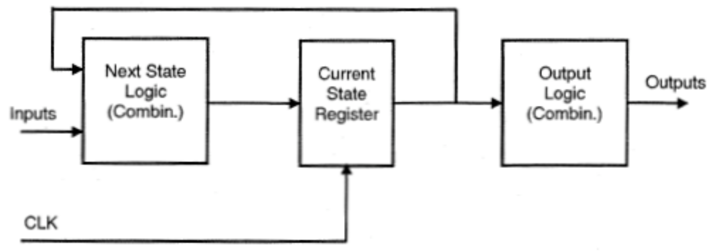
16-bit Loadable Counter (1)

```
SUBDESIGN loadable_counter
(
  clock, load, enable, reset: INPUT;
  d[15..0]: INPUT;
  q[15..0]: OUTPUT;
)
VARIABLE
  count[15..0]: DFFE;
```

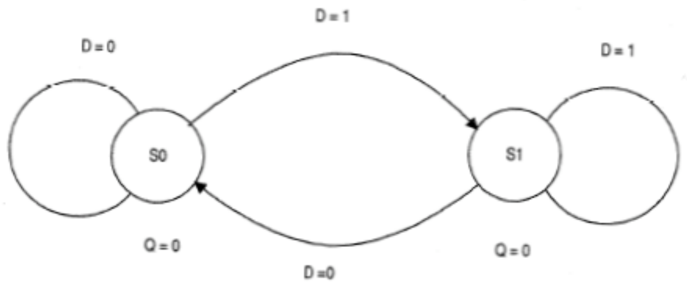
16-bit Loadable Counter (2)

```
count[].clk = clock;
count[].clrn = !reset;

IF load THEN
  count[].d = d[];
ELSIF enable THEN
  count[].d = count[].q + 1;
ELSE
  count[].d = count[].q;
END IF;

q[] = count[].q;
```

State Machines



Flip-Flop as a State Machine



Flip-Flop State Machine (1)

```
SUBDESIGN d_flip_flop
(
  clock, reset, d: INPUT;
  q: OUTPUT;
)
VARIABLE
  ss: MACHINE WITH STATES (s0, s1);
BEGIN
  ss.clk = clock;
  ss.reset = reset;
```

Flip-Flop State Machine (2)

```
CASE ss IS
  WHEN s0 =>
    q = GND;
    IF d THEN
      ss = s1;
    END IF;
  WHEN s1 =>
    q = VCC;
    IF !d THEN
      ss = s0;
    END IF;
END CASE;
```

Flip-Flop State Machine (3)

```
CASE ss IS
  WHEN s0 =>
    IF  d THEN ss = s1; END IF;
  WHEN s1 =>
    IF !d THEN ss = s0; END IF;
END CASE;

CASE ss IS
  WHEN s0 => q = GND;
  WHEN s1 => q = VCC;
END CASE;
```

State Machine Ports

- Clock (**`clk`**)
  - Required
- Asynchronous Reset (**`reset`**)
  - Active-High
- Clock Enable (**`ena`**)

Moore Machine (1)

```
SUBDESIGN manual_state_assignment
(
  clock, reset, ccw, cw: INPUT;
  phase[3..0]: OUTPUT;
)
VARIABLE
  ss: MACHINE OF BITS (phase[3..0])
      WITH STATES (s0 = B"0001",
                   s1 = B"0010",
                   s2 = B"0100",
                   s3 = B"1000");
```

Moore Machine (2)

```
TABLE
  ss, ccw, cw => ss;

  s0, 1,   x  => s3;
  s0, x,   1  => s1;
  s1, 1,   x  => s0;
  s1, x,   1  => s2;
  s2, 1,   x  => s1;
  s2, x,   1  => s3;
  s3, 1,   x  => s2;
  s3, x,   1  => s0;
END TABLE;
```

Mealy Machine (1)

```
SUBDESIGN mealy
(
  clock, reset, y: INPUT;
  z: OUTPUT;
)
VARIABLE
  ss: MACHINE WITH STATES (s0, s1, s2, s3);
```

Mealy Machine (2)

```
TABLE
  ss, y => z, ss;

  s0, 0 => 0, s0;
  s0, 1 => 1, s1;
  s1, 0 => 0, s1;
  s1, 1 => 1, s2;
  s2, 0 => 0, s2;
  s2, 1 => 1, s3;
  s3, 0 => 0, s3;
  s3, 1 => 1, s0;
END TABLE;
```