Using *edif2blif* Version 1.0
(Draft)

Paul Leventis

June 30, 1998

Contents

1 Overview

This document describes *edif2blif* , a tool that converts netlists from the industry standard Electronic Data Interchange Format (EDIF) to the academic Berkeley Logic Interchange Format (BLIF). Section 2 explains how the tool is typically used and what command line options are available. Section 3 gives a brief introduction to the EDIF file format. Section 4 describes the translation table file format. Section 5 gives application-specific instructions on how to perform a complete translation.

2 Command Line Options

The general usage of *edif2blif* is as follows:

```
edif2blif [flags] <edif_file> <blif_file> [tab_file]
```

2.1 EDIF file

This is the path and filename of the EDIF file to be translated. *edif2blif* has only been tested on files that adhere to the EDIF 2 0 0 specification and use keywordLevel 0. Not all tags are supported; if a tag is not supported you will get an error or warning message, depending on the tag. All buses in the EDIF file should be flattened, as port arrays are not supported.

2.2 BLIF File

This is the path and filename where the translated circuit will be written. *edif2blif* uses the .model, .inputs, .outputs, .names, .latch, and .end tags. All latches have timing "re 2," and constant generators (such as VCC and GND) are declared using a 0-input .names declaration.

2.3 TAB File

This is the path and filename of the translation table to use. This argument is needed if logic functionality is desired. There are a few translation tables provided with the *edif2blif* distribution in the "tab" directory; see the README file for details. Please see section 3 for a detailed description of the TAB file format.

Note: When translating sequential circuits, be sure to use at least a minimal TAB file containing the definitions for all latches. Any undefined cell is treated as a NAND gate.

2.4 Flags

There are a number of the flags that alter the behaviour and user interaction of *edif2blif*. Each flag can be turned off by placing a '-' in front of it, or turned on with a '+'. The only flag that requires some explanation is "func"; for a full list of flags run *edif2blif* with no arguments.

The "func" flag turns on or off the assignment of logic functionality to cells. The use of this option requires that every external cell have a definition in the translation table. If any cells do not match a provided definition, the translator will exit with an error. Turning off this option will result in a circuit with all undefined elements converted to NAND gates. Note: it is still useful to provide a translation table so that flip-flops will be recognised as such.

3 The EDIF File Format

EDIF is an industry-standard file format that allows EDA tools to communicate with each other, including the ability to transfer netlists, timing parameters, graphical representations, and any other data the vendors wish. The standard specifies the grammar and structure of the EDIF file, but leaves how this information is interpreted to the vendors.

EDIF is a hierarchical file format. Each EDIF file consists of a set of libraries, each of which can be of type "external," meaning that it contains declarations of cells and their interfaces that are used later in the EDIF file, but are not defined in the file, or of type "library", containing any newly defined cells. A "cell" is a circuit element, such as a gate, that has one or more named interfaces. Each "interface" has a set of "ports", and may have "contents" consisting of "nets" that "join" instantiated cells.

This is an example of how a half-adder would be described in EDIF:

```
(edif netlist
  (edifVersion 2 0 0) (edifLevel 0) (keywordMap (keywordLevel 0))
  (status
    (written
      (timeStamp 1998 2 0 16 45 4)
      (author "Paul Leventis")
      (program "vi :)"))
  )
  (external EXTLIB (edifLevel 0)
    (technology (numberDefinition)
      (simulationInfo
        (logicValue H (booleanMap (true)))
        (logicValue L (booleanMap (false)))))
    (cell AND2 (cellType GENERIC)
      (view INTERFACE
        (viewType NETLIST)
        (interface
          (port out (direction output))
          (port in0 (direction input))
          (port in1 (direction input)))
      )
    )
    (cell XOR2 (cellType GENERIC)
      (view INTERFACE
        (viewType NETLIST)
        (interface
          (port out (direction output))
          (port in0 (direction input))
```

```
        (port in1 (direction input)))
      )
    )
  )
  (library USER_LIB (edifLevel 0)
    (technology
      (numberDefinition)
      (simulationInfo
        (logicValue H (booleanMap (true)))
        (logicValue L (booleanMap (false))))
    )
    (cell HALFADD (cellType GENERIC)
      (view NETLIST
        (viewType NETLIST)
        (interface
          (port sum (direction output))
          (port carry (direction output))
          (port op0 (direction input))
          (port op1 (direction input)))
        (contents
          (instance AND2_1
            (viewRef INTERFACE (cellRef AND2 (libraryRef EXTLIB))))
          (instance XOR2_1
            (viewRef INTERFACE (cellRef XOR2 (libraryRef EXTLIB))))
          (net N_N0 (joined
              (portRef op0)
              (portRef in0 (instanceRef AND2))
              (portRef in0 (instanceRef XOR2))))
          (net N_N1 (joined
              (portRef op1)
              (portRef in1 (instanceRef AND2))
              (portRef in1 (instanceRef XOR2))))
          (net N_N2 (joined
              (portRef sum)
              (portRef out (instanceRef XOR2))))
          (net N_N3 (joined
              (portRef carry)
        )
      )
    )
  )
)
```

This example demonstrates the important tags used to describe a netlist. Notice the two external cells, **AND2** and **XOR2**, declared in the external library **EXTLIB**. Nowhere in the EDIF file is the functionality of these cells defined. In this example, the "status" and "technology" are not important — these tags, as well as many others, are ignored by *edif2blif*. The definition of the half-adder itself occurs in the library **USER_LIB**. First, the interface for this new cell is defined. Next, all cells used in the netlist are instantiated and given unique names. Finally, the pins of the half-adder and those of its cells are connected.

From this example it is clear that a number of problems may arise in converting EDIF to BLIF. First of all, the **AND2** and **XOR2** cells have no specified logic function. The *edif2blif* translator allows you to specify cell functions or to replace all unknown cells with NANDs. Secondly, there may be many user libraries with many new cells defined in each, and it is not mandatory to specify which is the top-level cell. Unless the file contains a "design" tag, the translator assumes that the last cell defined is the top-level cell. Finally, there are many different tags in EDIF, and it is possible that some important ones may be ignored or some that may be safely ignored trigger an error message.

4 Translation Tables

As mentioned above, EDIF parsers use external libraries to determine cell functionality. *edif2blif* uses an external translation table to supply definitions for all imported cells in an EDIF file. When an external cell is declared, or a library cell is specified without any contents, the supplied translation table is searched for a cell of the same name. If this cell is found, then its interface is compared to that of the unresolved cell, and if it matches, the cell is used in the place of the unresolved cell.

A translation table consists of a set of .DEFINE, .MAP, .ALIAS, and .LATCH statements followed by a .END statment. Between any two statements there can be a comment starting with a '#', as well as blank lines. For example, here is the translation table that would be used to convert the half-adder example of section 1:

```
# My Translation Table
# For circuits written by hand in vi

# A 2-input and gate
.DEFINE AND2(in0, in1; out)
11 1

# A 2-input xor gate
.DEFINE XOR2(in0, in1; out)
10 1
01 1

.END
```

Please see the "tab" directory for more comprehensive examples.

4.1 DEFINE Statement

```
.DEFINE cellName(input1, input2, !ignored_input, input3; output_name)
<truth table follows>
```

The DEFINE tag defines a new cell with a set of named inputs and an output. It may have anywhere from 0 to 30 inputs, which must be listed in the same order as they appear in the EDIF cell. An input may be ignored by placing a '!' before its name. Any nets that connect to an ignored input will have that pin removed. A DEFINE tag is followed by zero or more product terms that define the truth table. Each term is of the format:

```
bbb...b 1
```

Where b is '0' for a logic value of low, '1' for a logic value of high, or '-' for a don't care condition. There should be one b per non-ignored input in the define.

An example where ignored inputs are useful is in defining a tri-state buffer:

```
# Defines a Tri-State buffer
.DEFINE TRIBUF(in1, !enable; out)
1 1
```

Note that even though the enable port is not used it must be present in the cell definition in order to match the interface present in the EDIF file. In the above example, all connections to the enable input of the tri-state buffer will be ignored, thus the tri-state will be treated like a single input AND gate. This is necessary since BLIF does not support tri-state buffers. Though not useful for tri-states that drive buses, this technique allows "dummy" tri-states that have been inserted for timing analysis to be removed. This is the case with the EDIF output from Altera MaxPlus2.

4.2 MAP Statement

```
.MAP newCellName oldCellName(newInputName1, 0, 1; newOutputName)
```

The MAP statement allows a new cell to be defined in terms of an already defined cell. The new cell will have ports with the names specified in the argument list. If a constant, either a '0' or a '1', is given as a port name, then the orginal cell's truth table will be reduced with the given logic value on that port. For example:

```
# A 4-input AND gate
.DEFINE AND4(and4_a, and4_b, and4_c, and4_d; and4_out)
1111 1

# A 2-input AND gate
.DEFINE AND2(and2_a, and2_b; and2_out)
11 1

# The same 2-input AND gate defined using a MAP statement
.MAP AND2MAPPED AND4(and2_a, 1, and2_b, 1; and2_out)
```

4.3 ALIAS Statement

```
.ALIAS newCellName oldCellName
```

The ALIAS statement creates a new cell that is functionally equivalent to an already defined one. This new cell has an unnamed set of ports — that is, its ports remain unnamed until a cell with a matching name is declared in an external library of an EDIF file. For example:

```
(cell PASSTHROUGH
  (cellType GENERIC)
  (view INTERFACE)
    (viewType NETLIST)
    (interface
      (port theinput (direction INPUT))
      (port theoutput (direction OUTPUT))
    )
  )
)
```

The tab file could look like this:

```
# A cell which does nothing
.DEFINE PASSTHROUGH(theinput; theoutput)
1 1
```

or, using the .ALIAS statement, it could look like this:

```
# A 1-input AND gate
.DEFINE AND1(a; q)
1 1

# A cell which does nothing and has an unnamed interface
.ALIAS PASSTHROUGH AND1
```

Note that all inputs and outputs will be mapped in the order that they appear in the EDIF file.

4.4 .LATCH Statement

```
.LATCH CellName(!ignored_input, data_input, !ignored_input2, clk_input; output)
```

The LATCH statement defines a D-type flip-flop. The first non-ignored input is taken as the data input, and the second is the clock. Currently there is no facility for supporting preset, clear, or enable inputs, so these inputs must be ignored. For example:

```
# A DFF w/Enable
.LATCH DFFE(D, CLK, !CLRN, !PRN, !ENA; Q)
```

Note: This rather clumsy LATCH statement will be revised or replaced in the next revision of *edif2blif*.

5 Limitations

Though I have chosen not to fix the following limitations right now, I will be willing to do so if there is enough user interest. Please e-mail me (leventi@eecg.utoronto.ca) if any of these items are causing you difficulties. *edif2blif:*

- Cannot model preset, clear, and enable inputs to flip flops. This problem can be worked around by changing the definition of the flip flop in the EDIF file so that it simulates a synchronous preset/clear/enable by adding logic before the D input.
- Cannot handle EABs in MaxPlus2. A solution is to change all I/Os to the instantiated memories to external pins. This may be automated in future versions of *edif2blif* by creating a new statement type in the TAB file to declare a cell be external to the design.
- Cannot handle bi-directional pins. BLIF doesn't support bi-directional pins, but provided that there are not multiple drivers on a pin, a bi-directional pin could be split into an input and an output pin. This will work for MaxPlus2 EDO files.

6 Performing Conversions

This section contains application-specific instructions on how to perform conversions to BLIF. As more source applications are tested, additional instructions will be added.

6.1 Converting from Altera MaxPlus2 8.x

Any MaxPlus2 project that does not make use of a EABs can be translated into blif with relative ease. The only limitations are that presets, clears and enables will be ignored, and bidirectional pins will cause conversion to fail.

MaxPlus2 EDIF is intended for use in timing analysis. Many **DELAY** and **TRIBUF** cells are inserted into the EDIF file in order to accurately model the delay. As a result, the BLIF file produced by *edif2blif* should be minimized by a logic optimizer.

To convert a project into BLIF under MaxPlus2 8.x:

1. In the Compiler, turn "EDIF Netlist Writer" on. This can only be done with "Timing SNF Extraction" turned on.
2. In "EDIF Netlist Writer Settings," select "EDIF Version 2 0 0", and select "Exemplar" as the target.
3. Compile the design; a .edo file will be created.
4. Run *edif2blif* on the .edo file with the command "edif2blif project.edo project.blif altera.tab." You will need to specify a path to the altera.tab file if it is not in the current directory.