The FHDL User Manual

Peter M. Maurer

Department of Computer Science
Baylor University
Waco, TX 76798

Table of Contents

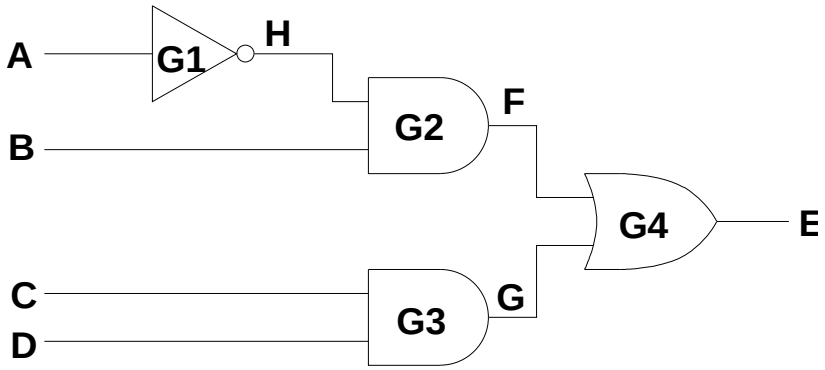
CHAPTER 1

The FHDL Gate-Description Language

1.1 Overview

The Functional Hardware Design Language (FHDL) resembles assembly language in that each statement has a label field, an operation code, and a list of operands. The label field, which is optional, starts at the first character of a statement and ends with a colon (:). The operation code must always be present and must be preceded by one or more spaces or tabs. If a label is present the operation code follows the label. The operand list, which is also optional, follows the operation code, and must be separated from the operation code by one or more spaces or tabs. Every statement must be followed by a newline character (return key) or a semicolon (;).

To translate a logic diagram such as the following into FHDL you must first choose unique names for each of the connections.



In this example the names A-H have been chosen for the connections. Next you must choose a unique name for the circuit (for this example we will choose "example1.") You begin your circuit description with the following statement.

```
example: circuit
```

Next you must make a list of the primary inputs and outputs of the circuit. In the example above, A, B, C, and D are primary inputs while E is a primary output. An "input" statement is used to declare primary inputs, while an "output" statement is used to declare primary outputs. The following statements would be used for our example.

```
inputs:  A, B, C, D
outputs: E
```

Next you must describe each gate using an appropriate statement. In our example, one "not" statement, two "and" statements, and one "or" statement will be required. These statements do not require labels, but it is a good idea to use them anyway. The gates of our example would be written as follows.

```
g1:  not  A,H
g2:  and  (H,B),F
g3:  and  (C,D),G
g4:  or   (F,G),E
```

The order of the statements (including the input and output statements) is arbitrary. The operand list of a statement that describes a gate has two parts. The first part lists the gate's inputs while the second part lists the gate's outputs. If there is more than one input, the list of inputs must be enclosed in parentheses. Similarly if there is more than one output, the list of outputs must be enclosed in parentheses.

Finally you must end your description with an "endcircuit" statement. The entire description is given below.

```
example1: circuit
     inputs    A,B,C,D
     outputs   E
g1:  not  A,H
g2:  and  (H,B),F
g3:  and  (C,D),G
g4:  or   (F,G),E
     endcircuit
```

FHDL "knows" about many different kinds of gates, which are listed in Appendix A. In most cases the order of the inputs and outputs is significant.

1.2 Simulating Circuits

Put your circuit description into a file that ends with the string ".ckt". The rest of the file name should match the name of your circuit. For the example given in the last section, you would use the file name "example1.ckt". Once this file is created, run the following UNIX command

```
fhdl example1.ckt
```

If you haven't made any errors, this will give you a file named "example1.c". You must then run the following command.

```
cc example1.c -o example1
```

The program "example1" will simulate your circuit, but first you must create a file named "example1.vec" which contains the test vectors for your circuit. You simulate your circuit using the following command.

```
example1 <example1.vec
```

Each test vector represents one set of inputs for your circuit. You will get one set of outputs for each set of inputs. Each set of inputs must appear on a separate line. The values for each of the primary inputs must be listed in the same order as they appear on the "input" statement. A ".vec" file for our example is illustrated below.

```
0,1,0,0
1,1,0,1
0,0,0,0
1,1,1,1
```

This file will produce the outputs:

```
1
0
0
1
```

Blank lines and lines beginning with asterisks are treated as comments in a ".vec" file. These lines will be copied into the output file to assist you in reading the output.

More sophisticated users will wish to make use of additional features of FHDL that are described in other memoranda. Features currently available are ROM and PLA preprocessors, the FHDL macro processor, and the FHDL driver language. To invoke all FHDL functions, replace the fhdl command given above with the following command.

```
fhdl -n example1.ckt
```

This command has the added benefit of performing the "cc" command automatically for you. You may use this version of the command even if you do not use any of the additional FHDL features, although compilation time will be somewhat longer.

1.3 Creating and Using Subcircuits.

In addition to the known gate types, you can create new ones by declaring them as circuits. For example, suppose you want to create a two-input "NAND" with one active-low input and one active-high input. You could do this as follows.

```
xnand:    circuit
     inputs    A,B
     outputs   C
     not  A,ABAR
     nand (ABAR,B),C
     endcircuit
```

You may now use xnand as a gate in any other circuit in the same file. When you have several circuits in the same file, the first is treated as the main circuit and all others are treated as subcircuits. To use the xnand subcircuit, include a statement similar to the following in some other circuit.

```
gtest:    xnand     (Q,R),S
```

You may use a subcircuit any number of times.

1.4 Wire Declarations.

A signal may be declared to be active low by using the following statement.

```
wire A,type=active_low
```

The signal A should appear as a connection in some gate of the circuit. Similarly, a signal can be declared as unconnected by using the following statement.

```
wire B,type=no_connect
```

Signals can be declared as permanently one or permanently zero by using one of the following statements.

```
zero A,E,F
one  B,C
```

A signal can be declared to be a bus, which will cause it to be treated as a collection of independent signals. This option makes coding easier and more readable and makes simulations run faster. To declare a signal as a bus, use the following statement.

```
wire A,width=10
```

The width may be anything from 1 to 32. When a bus is fed into an ordinary gate, the gate will be replicated to match the width of the signal. Different gates treat buses differently, so check the appendix first.

The collect, distribute, and expand statements are used to connect signals and other buses to a bus. The collect statement is used to feed a collection of signals into a bus. An example of a collect statement is given below.

```
collect   (a,b,c),d
```

The signals a, b, and c are "collected" together into the bus d. The three signals may have any width, but the sum of the widths of the inputs cannot

be larger than the width of the output. The "position" parameter can be used to place a, b, and c at specific points within d, as illustrated by the following statement.

```
collect   (a,b,c),d,position=(1,3,5)
```

Assuming that the widths of a, b, and c are all 1, this statement inserts a, b, and c into the odd numbered positions of d. Note that the collect statement *creates* the bus d, so you *cannot* use several collect statements to build a bus. You have to do it all in one statement. The "position" parameter treats the leftmost bit of the bus as position zero. No overlap check is done for position parameters, so you must be careful.

The "distribute" statement is the reverse of the collect statement. It is used to distribute the signals in a bus to several output signals. The following statement distributes the signals in a bus a to two outputs x and y.

```
distribute  a,(x,y)
```

The leftmost bits of "a" are fed to "x", and the next bits are fed to "y." The width of "a" must be greater than or equal to the sum of the widths of "x" and "y." The "position" parameter can be

used on the distribute statement in order to specify the location within the input where the bits of an output must start. For example, the following statement extracts the high and low order bits from a 16-bit bus.

```
distribute abus,(highbit,lobit),position=(0,15)
```

The "expand" statement fans a width 1 signal out into every position of a bus. Assume that the width of the signal a is 1 and the width of signal b is 5. Then the following two statements are equivalent.

```
expand    a,b
collect   (a,a,a,a,a),b
```

1.5 The FHDL ROM Specification Language

The FHDL ROM specification language allows one to specify the number of address bits, the number of bits per word, and the contents of each word. See Chapter III for a more sophisticated ROM specification language. The contents of a word must be specified in hexadecimal. If a ROM of constants is to be created, the FHDL ROM specification language may be preferable to the ROM preprocessor language.

In FHDL, a ROM is a circuit containing only "romword" statements and input/output declarations. The following is an example.

```
rom1:     circuit
     inputs    address
     outputs   romout
     wire address,width=8
     wire romout,width=16
     romword   7ffe
     romword   8000
     romword   0001
     romword   07ff
     endcircuit
```

As this example shows, each "romword" instruction supplies the value for one word of the ROM. ROM addresses are assigned consecutively starting with zero. The word values must be specified in hexadecimal, without the "0x" prefix. There must be exactly one input, and the width of the input must be explicitly declared to be the number of address bits in the ROM. There must be at least one output, and each output should have a declared width. The maximum width for any input or output is 32. ROMs with word length greater than 32 may be constructed using multiple outputs. When a ROM has multiple outputs, each "romword" instruction must specify the value of each output as illustrated below.

```
abc: circuit
     inputs    address
     outputs   o1,o2,o3
     wire address,width=8
     wire o1,width=4
     wire o2,width=4
     wire o3,width=4
     romword   a,b,c
     romword   d,e,f
     romword   0,1,2
     romword   3,4,5
     endcircuit
```

As both examples illustrate, it is *not necessary* to specify a value for each word in the ROM. When the FHDL compiler generates simulation code for the rom, the input address is checked against the address of the last word specified, and if the input address is larger, a run-time error message is issued. This provides a convenient method for determining whether invalid ROM addresses are being issued.

1.6 The FHDL PLA Specification Language

The FHDL PLA specification language allows one to specify the number of inputs, the number of outputs, and the contents of the AND and OR plane portion of each wordline. See Chapter IV for a more sophisticated PLA specification language. The contents of the AND plane must be specified in trinary where 0 and 1 represent a bit test against the specified value and x represents don't care. The trinary string is normally enclosed in quotes. The contents of the OR plane must be specified in hexadecimal.

In FHDL, a PLA is a circuit containing only "plaword" statements and input/output declarations. The following is an example.

```
pla1:     circuit
     inputs    a1
     outputs   plaout
     wire a1,width=8
     wire plaout,width=16
     plaword   "0111xxxx",7ffe
     plaword   "x010x01x",8000
     plaword   "011x0xxx",0001
     plaword   "0111101x",07ff
     endcircuit
```

As this example shows, each "plaword" instruction supplies the AND and OR plane values for one wordline of the PLA. OR plane values must be specified in hexadecimal, without the "0x" prefix. There must be at least one input and at least one output, and each input and output should have a declared width. The maximum width for any input or output is 32. PLAs with more than 32 inputs and outputs may be constructed using multiple inputs and outputs. When a PLA has multiple inputs and outputs, each "plaword" instruction must specify the value of each input and each output as illustrated below.

```
abc: circuit
     inputs    a1,a2
     outputs   o1,o2,o3
     wire a1,width=2
     wire a2,width=2
     wire o1,width=4
     wire o2,width=4
     wire o3,width=4
     plaword   "xx","01",a,b,c
     plaword   "0x","x0",d,e,f
     plaword   "1x","x1",0,1,2
     plaword   "11","xx",3,4,5
     endcircuit
```

As both examples illustrate, it is *not necessary* to specify a value for each set of input conditions. When the FHDL compiler generates simulation code for the pla, the input condition is checked against the conditions specified for each wordline. If no wordline is selected by the condition, a run-time error message is issued. This provides a convenient method for determining whether invalid PLA conditions are being issued.

1.7 Known Gate Types.

1.7.1 Simple Gate Types

```
and
or
nand
nor
not
xor
xnor
```

All of these gates may be used with buses as long as the widths of all inputs and outputs are identical. The gates replicate themselves for each element of the bus. Replicated gates typically simulate much faster than individually specified gates. All gates except "not" may have an arbitrary number of inputs, but must have at least two inputs. The "not" gate must have one input. All of these gates must have one output. Since all of these gates represent symmetric functions, the order of the inputs is not significant.

1.7.2 And-or-inverts and Or-and-inverts.

```
aoi...
oai...
```

These actually represent families of gates rather than a single gate. The aoi or oai prefix must be followed by a string of digits, which define the structure of the gate. For an aoi, each digit represents one AND gate, and the value of the digits defines the number of inputs for the AND gate. The outputs of all AND gates are ORed together and the output is inverted. The oai works in a similar fashion. Only digits 1-9 may be used, 0 is unacceptable. There is no

restriction on the number or order of the digits, but in practice one should conform to the conventions of existing cell libraries. The inputs are clustered from right to left in accordance with the order of the digits in the digit string. The following is an example of an aoi gate.

```
aoi212  (a1,a2,b1,c1,c2),out
```

This gate could be translated into ands ors and nots as follows.

```
and  (a1,a2),x1
and  (c1,c2),x2
or   (x1,b1,x2),x3
not  x3,out
```

1.7.3 Flip Flops.

```
rsff
dff
dff1
dff2
dff3
dff4
jkff
jkff1
jkff2
jkff3
jkff4
tff
tff1
tff2
tff3
tff4
```

All flip flops may have one or two outputs. If one output is specified, it is the normal uncomplemented output. If two outputs are specified, the first is the uncomplemented output and the second is the complemented output. The two outputs must have the same width.

The rs flip flop (rsff) has two inputs, set and reset.

The d flip flop (dff) has two inputs, d and clock.

The jk flip flop (jkff) has three inputs, j, k, and clock.

The t flip flop has two inputs toggle and clock.

For these flip flops, the inputs are all active high must be specified in the order given.

Flip-flops of the form dff1, jkff1, and tff1 are CMOS variations on the basic flip-flops. In addition to the inputs already mentioned, each of these must also have an inverted-clock input. Flip-flops ending in 1 have no other inputs in addition to the inverted clock. Those ending in 2 have an active-high asynchronous set, those ending in 3 have an active-low asynchronous reset, and those ending in 4 have both. The additional inputs follow the clock in the following order as appropriate <inverted-clock>,<set>,<reset>.

If the outputs of a flip flop are buses, the gates replicate themselves for each bit in the bus. The width of an input must be 1 or identical to the width of the outputs. If the width of an input is 1, the input is fanned out to all of the replicated gates. Otherwise, the individual bits in the

input are routed to the individual flip flops. This replication of gates is logical not physical, so replicated gates usually simulate much faster than a collection of individually specified gates.

1.7.4 Tristate Gates

tbufi
tgate

These gates are similar in function. The first is an inverting tristate buffer, while the second is a non-inverting transmission gate. When laid out, the first will have amplification, the second will not. The inputs to these gates are identical. The first input is the D input while the next two are clock and inverted clock. The clock is active high. If the clock is active, the tbufi acts as an inverter while the tgate copies its input to its output. If the clock is not active, the output of the gate does not change. (This allows wired-or connections to work properly.)

1.7.5 Special Function Gates

expand
collect
distribute
hlcv

The "expand," "collect," and "distribute" gates are described in detail in the text of this document. The "hlcv" gate is used to convert an active-high signal to an active-low signal and vice versa. No logic function is performed, the value of the output is identical to the value of an input. However, the input can be declared as active-high and the output as active-low, or vice versa. This allows proper functioning of gates that are sensitive to active-high, active-low declarations without introducing unnecessary logic.

1.7.6 Functional blocks.

mux
demux
decoder
encoder
comparator
adder
ram
register
counter
alu

1.7.6.1 Mux format

```
mux  (inputs,control),output
```

The control input may be either a set of n width-1 inputs or a single width-n bus. If the inputs and outputs are all the same width, there must be one output and 2^n inputs. In this case, the MUX is replicated for each bit in the output. If the inputs and outputs are not all the same

width, then the output must be width 1, there must be only one input, and that must be a bus of width 2^n.

1.7.6.2 Demux format

```
demux     (input,control),(outputs)
```

The control input may be either a set of n width-1 inputs or a single width-n bus. If the inputs and outputs are all the same width, there must be one input and 2^n outputs. In this case, the MUX is replicated for each bit in the input. If the inputs and outputs are not all the same width, then the input must be width 1, there must be only one output, and that must be a bus of width 2^n.

1.7.6.3 Decoder format

```
decoder   (control),(outputs)
```

The control input may be either a set of n width-1 inputs or a single width-n bus. If the outputs are all of width one, there must be 2^n of them. Otherwise there must be a single output of width 2^n.

1.7.6.4 Encoder format

```
encoder   (control),(outputs)
```

The control input may be either a set of 2^n width-1 inputs or a single width-2^n bus. If the outputs are all of width one, there must be n of them. Otherwise there must be a single output of width n. This is a priority encoder.

1.7.6.5 Comparator format

```
comparator   (leftin,rightin),(output)
```

Either leftin or right in may be n width-1 signals or a width-n bus. The output must be three width one signals or a width-3 bus. The first output is active for leftin<rightin, the second for leftin=rightin and the third for leftin>rightin.

1.7.6.6 Adder format

```
adder   (leftin,rightin),(output)
```

Either leftin or right in may be n width-1 signals or a width-n bus. The output may also be n width-1 signals or an width-n bus. The default is to have no carry in and no carry out. The presence of a carry-in is specified by the attribute "carry=in" while the presence of a carry out is specified by the attribute "carry=out." The presence of both is specified by the attribute "carry=(in,out)." If carry in is present, it must follow "rightin" in the input list and must have a width of one. Similarly carry out follows "output" in the output list and must have a width of one. The following is an example with both.

```
adder   (leftin,rightin,ci),(output,co),carry=(in,out)
```

1.7.6.7 Ram format

```
ram  (address,datain,readwrite),(dataout)
```

Address, datain and dataout must be buses. Datain and dataout must have the same width. Readwrite must be a width 1 signal which is zero for read and one for write. The size of the ram depends on the width of the address.

1.7.6.8 Register format

```
register  (datain,load,controlsignals,clock),
     (dataout,statussignals),
     options
```

Datain and dataout may be a set of n width-1 signals or one width-n bus. Options are of the form option_name=option or option_name=(opt1,opt2,...). The options determine the content of the controlsignals and statussignals fields. The available options are as follows.

```
control=clear  (include an asynchronous clear)
        right  (include a shift-right signal)
        left   (include a shift-left signal)
        set    (include an asychronous set)
```

These signals appear in the controlsignals field in the order specified. If the control option is not used, there are no signals in the controlsignals field.

```
status=all_zero     (include a zero status output)
       all_ones     (include a set status output)
```

These signals appear in the statussignals field in the order specified. If status is not specified, there are no signals in the statussignals field.

```
clock=yes (create a clocked register)
      no  (create an async-load register)
```

Default is "yes." If "yes" is specified, the clock is always the last input in the input list.

```
serial=right   (Include a right serial input)
       left    (Include a left serial input)
```

This option adds serial inputs to the controlsignals field. These will appear in the order specified, either at the begining of the controlsignals field or at the end, depending on whether the status or control input is specified first. The right serial input is used for left shifts and the left serial input is used for right shifts.

1.7.6.9 Counter format

```
counter   (datain,controlsignals),
     (dataout,statussignals),
     options
```

The datain and dataout, if present, must be a collection of n width-1 signals, or one width-n bus. The options are the same format as those for registers. The available options are as follows.

```
control=set
        clear
        count_up
        count_down
        load
```

Each option causes one control signal to be included in the controlsignals field in the order specified. If "load" is not included, then "datain" is implicitly omitted. If both count_up and count_down are omitted, the counter will count up with each clock pulse (if the clock is present) or with each input vector (if there is no clock).

```
status=all_zero
       all_ones
```

This is the same as for registers.

```
clock=yes
      no
```

This is the same as for registers.

```
dataout=yes
        no
```

If dataout=no is specified, the dataout field will be omitted. Default is dataout=yes.

```
width=<<number>>
```

If both datain and dataout are missing, you must use this parameter to indicate how many bits are in the counter.

```
type=binary
     decade
```

The normal (and default) type of counter is a binary counter. A decade counter counts from zero to nine and back to zero, and must have a width of 4.

1.7.6.10ALU format

```
alu  (Ainput,Binput,control),(output,statussignals),options
```

Ainput, Binput, and output may each be n width-1 signals or one width-n bus. Control may be from four to six width-1 signals or a bus of width from four to six. The size of the control field depends on the options. Options have the same format as for registers and counters. The options are as follows.

```
control=carryin
        cenable
```

These options increase the size of the control field from a default of four to either five or six depending on whether one control option or two has been specified. The exact position of the carryin and carry enable signals really doesn't matter since the ALU simulator considers the control field to be one n-bit field.

```
status=all_zero
       all_ones
       carryout
```

These options add status signals to the statussignals field.

```
function=(number, name , ... )
```

This parameter indicates which function is selected for each value of the control inputs. The control field is treated as an n-bit binary number where n is either 4, 5, or 6. Function name must be one of the following. During simulation, when the control field has the value "number" the function indicated by "name" is performed on "Ainput" and "Binput" and placed into "output." The available functions are listed in the following table.

Specification	Function
one	**1**
zero	**0**
a	**a**
b	**b**
not_a	**a'**
not_b	**b'**
and	**a&b**
or	**a|b**
xor	**(a&b')|(a'&b)**
nand	**a'|b'**
nor	**a'&b'**
xnor	**(a&b)|(a'&b')**
a_and_not_b	**a&b'**
a_or_not_b	**a|b'**
not_a_and_b	**a'&b**
not_a_or_b	**a'|b**
subtract	**a-b**
add	**a+b**
incr_a	**a+1**
incr_b	**b+1**

Specification	Function
decr_a	**a-1**
decr_b	**b-1**
b_minus_a	**b-a**

CHAPTER 2

Algorithmic State Machines

The statements of an algorithmic state machine description have the same format as those of an FHDL statement (see Chapter I, section 1). The first statement of an algorithmic state machine is a "circuit" statement that gives the name of the circuit. The last statement is an "endcircuit" statement. The body of an ASM is declared using the statements "asm_state," "asm_test," and "asm_cond". These statements cannot be mixed with FHDL gate declarations. (However, a file may contain several different types of circuits.) An "asm_state" statement is used to declare each state of a state machine, the "asm_test" statement is used to declare tests for conditional state transfers and conditional outputs, while the "asm_cond" statement is used to declare conditional outputs.

2.1 ASM State Declarations

The format of the "asm_state" statement is as follows.

```
asm_state state_name,next_statement,o=(state_outputs)
```

The field "state_name" gives a unique name to the state. The "state_outputs" field is a list, separated by commas of the outputs that must be unconditionally asserted when the machine is in this state. Any number of outputs may be specified. If only one output is specified, the parens may be omitted. The "next_statement" field is the name of the statement that follows this statement in the flow chart of the state machine. If the machine transfers unconditionally to a new state, then "next_statement" will be the name of the new state. If the state transfers to two or more states depending on certain conditions, then "next_statement" will be the name of an "asm_test" statement.

2.2 ASM Condition Declarations

The "asm_test" statement is used to cause conditional state transfer, and to activate conditional outputs. The format of the "asm_test" statement is as follows.

```
asm_test  test_name,(next_statements),c=(conditions)
```

Any number of conditions may be specified, and each condition may have the value true or false. The list of "next_statements" must contain one statement name for each combination of condition states. That is, if n conditions are specified, then 2^n "next_statement" names must be specified. If one condition is specified, then the "false" "next_statement" name must come first, and the "true" "next_statement" name must come second. In general, you can determine the order of the "next_statement" names by treating the combination of condition values as a string of binary digits, with zero represented by false and one represented by true. The "next_statement" names must be specified in ascending numeric sequence. Thus for two conditions, the "next_statement" names must be specified in the order FF, FT, TF, and TT.

Each statement name in the "next_statement" list must be the name of another statement. An "asm_state" statement is named to create a conditional state transfer, an "asm_cond" statement to create a conditional output, and another "asm_test" statement is named to create a chain of tests.

2.3 ASM Conditional Output Declarations

Conditional outputs are created by using "asm_cond" statements. The format of an "asm_cond" statement is given below.

```
asm_cond  statement_name,next_statement,o=(outputs)
```

The "statement_name" field is a unique name that is assigned to the statement. The "next_statement" field is the name of the statement that follows this one in the flow chart of the circuit. The "outputs" field is a list of outputs that must be activated when the conditional output is activated. This statement should follow one or more "asm_test" statements. The "next_statement" field may name an "asm_state," an "asm_test," or another "asm_cond" statement.

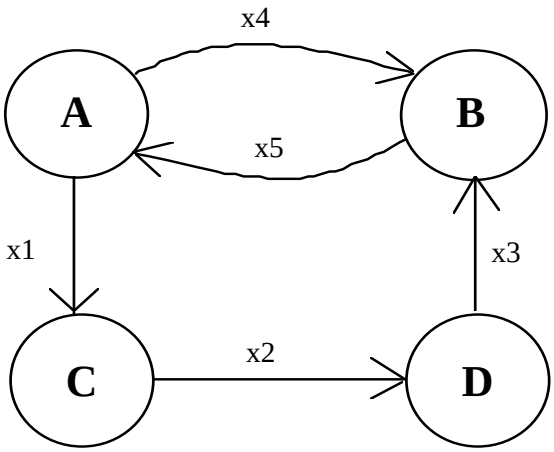
All outputs that appear on "asm_state" and "asm_cond" statements must be declared as primary outputs of the circuit containing them. Furthermore all conditions that appear on an "asm_test" statement must be declared as primary inputs of the circuit. All primary inputs and outputs of an ASM must have width 1, although they may be declared as active-high or active-low. When an active-low output is "activated" its value is set to zero, otherwise it is set to one. When an active-high output is activated, its value is set to one otherwise it is set to zero. If an output is not mentioned in a state, it will be set to its inactive value.

For active-high conditions, zero is treated as false and one is treated as true. For active-low conditions, zero is treated as true and one is treated as false.

You must be careful that each statement belongs to only one state. In other words it must not be possible to go from two "asm_state" statements to a particular "asm_test" or "asm_cond" statement without going through another "asm_state." You will get several error messages if you violate this rule. (If you violate this rule it is impossible to build a circuit corresponding to the ASM specification, even if you could manage to simulate it in software.)

2.4 State Machine Examples

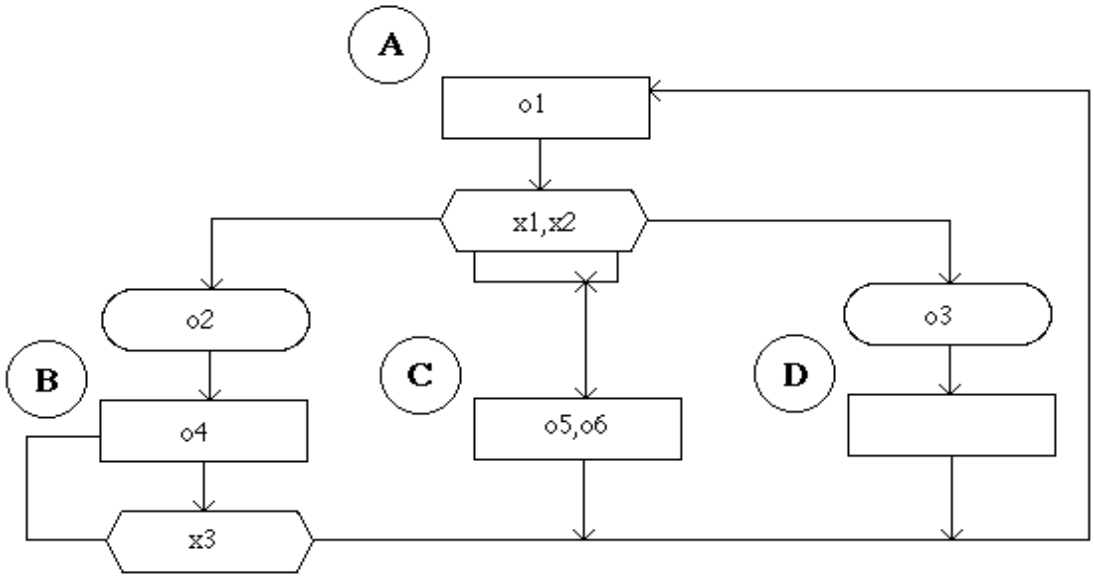
To illustrate the use of FHDL to code simple state machines, consider the following example.



This statemachine would be coded in FHDL as follows.

```
example2: circuit
          inputs    x1,x2,x3,x4,x5
          outputs   o1,o2,o3,o4
          asm_state A,testa,o=o1
          asm_test  testa,(A,B,C,B),c=(x1,x4)
          asm_state B,testb,o=o2
          asm_test  testb,(B,A),c=x5
          asm_state C,testc,o=o3
          asm_test  testc,(C,D),c=x2
          asm_state D,testd,o=o4
          asm_test  testd,(D,B),c=x3
          endcircuit
```

The following is a more complicated example.



This example would be coded in FHDL as follows.

```
example3: circuit
          inputs    x1,x2,x3
          outputs   o1,o2,o3,o4,o6,o6
          asm_state A,test1,o=o1
          asm_test  test1,(cout1,C,C,cout2),c=(x1,x2)
          asm_cond  cout1,B,o=o2
          asm_cond  cout2,D,o=o3
          asm_state B,test2,o=o4
          asm_test  test2,(B,A),c=x3
          asm_state C,A,o=(o5,o6)
          asm_state D,A
          endcircuit
```

If an state machine requires a clock, it must be specified on the "circuit" statement, and in the list of primary inputs. The following is a modified version of the first two statements of example3, to include a clock.

```
example3: circuit   clock=iclk
          inputs    x1,iclk,x2,x3
```

If it is necessary for the current state of the state machine to be visible to other logic in your design, place the name "current_state" in the list of primary outputs. The following is a modification of example3 to include a clock and current_state output.

```
example3: circuit   clock=iclk
          inputs    x1,x2,x3,iclk
          outputs   o1,o2,o3,current_state,o4,o6,o6
```

Once a state machine has been declared, it can be used just like any other subnetwork. The state machine can be used any number of times. Each time the state machine is used, a new (logical) copy of the machine is created.

CHAPTER 3

The ROM Preprocessor

3.1 Overview

The FHDL ROM compiler is a preprocessor that provides a simple but powerful language for specifying the contents of a ROM. The format of the ROM preprocessor statements is modeled after that of FHDL statements. Each statement has a label field, an opcode, and an operand field. The label field begins at the first character of the statement and ends with a colon (:). The label field is optional for some types of statements, and manditory for others. The opcode, which is manditory for all statements, follows the label field, and must be separated from the label field by one or more spaces or tabs. If no label field is present, the opcode must be preceded by one or more spaces or tabs to signify that the label field is omitted. The operand field follows the opcode and must be separated from the opcode by one or more spaces or tabs. The operand field consists of one or more operands separated by commas. The format of an operand depends on the type of statement. The operand field, and the statement, end with a newline (return key) or a semicolon(;). The operand field of a statement can be continued to the next line by ending a line with a comma. Spaces and tabs are not allowed in the operand field, except preceeding or following a comma.

Labels may contain upper and lower case letters, numbers, underlines and periods. Labels may not duplicate a ROM preprocessor keyword, nor may duplicate labels be defined, regardless of type. Case is significant for labels, so Lab1, LAb1, and lAb1 are three different labels. On the other hand, case is *not significant* in keywords, so rom, RoM and ROM are all the same keyword.

Once a ROM description has been completed, it must be run through the ROM preprocessor before being compiled by the FHDL compiler. Although FHDL provides a method for describing the contents of a ROM, the method is not flexible enough for debugging complex microcoded ROMs. Nevertheless, the native FHDL method is probably more convenient for

specifying the contents of ROMs that contain constants and other simple forms of data. Therefore, the final section of this report contains a description of the FHDL native method for specifying ROM contents. The ROM preprocessor converts the preprocessor language into FHDL native mode instructions.

3.2 Specifying Fields.

Each word in a ROM is broken into one or more fields. Fields may contain many different types of data, some examples of which are data, rom addresses, control signal values, and so forth. Each field must be declared using a statement similar to the following.

```
new_addr: field   width=12,position=15
```

This statement declares a field named "new_addr" which has a width of 12 bits and begins at bit 15 of the ROM word. The bits of each word are numbered from the left starting with zero. Each field in the ROM word must be declared, even if it is never used. The order of the "field" statements is not important, since each statement has both a width and position associated with it. If the "width" specification is omitted, a width of one is assumed. If the "position" specification is omitted, a position of zero is assumed. Fields may not overlap.

3.3 Using Equates.

The width and position parameters of the "field" statement can be rather difficult to keep track of if the number and position of your fields changes often. (This is sometimes the case during ROM development.) The "equ" statement can be used to simplify the process of adding new fields, and changing the size of existing ones. Suppose you have the following three fields, declared as in the previous section.

```
flda:  field   width=3,position=0
fldb:  field   width=4,position=3
fldc:  field   width=7,position=7
```

If you want to add a field between flda and fldb, you must change the position of fldb and fldc. The same is true if you change the size of flda. The following is the same three fields coded with equates.

```
flda:  field   width=awid,position=apos
fldb:  field   width=bwid,position=bpos
fldc:  field   width=cwid,position=cpos
awid:  equ     3
bwid:  equ     4
cwid:  equ     7
apos:  equ     0
bpos:  equ     awid+apos
cpos:  equ     bwid+bpos
```

Using this technique, one need only concern oneself with the width and the order of each field. Positions are calculated automatically by the preprocessor.

The right hand side of an equate may be an arbitrary expression involving constants; the names of other equates; the operators +, -, *, and /; and parenthesis. The order of the equate statements does not matter, as long as they do not reference one another cyclically. Expressions may also be used to specify field widths and positions, as illustrated below.

```
flda:  field   width=awid,position=0
fldb:  field   width=bwid,position=bpos
fldc:  field   width=awid+bwid,position=bwid+bpos
awid:  equ     3
bwid:  equ     4
apos:  equ     0
bpos:  equ     awid+apos
```

This example illustrates the rule that any place a number is acceptable, an expression is also acceptable.

3.4 Specifying ROM words.

The "word" statement is used to specify the contents of a ROM word. The label field of a "word" statement is optional. The operands of a "word" statement, which are called commands, specify the contents of each field of the word. The contents of a field is specified using an expression of the following form.

```
expression->field_name
```

To illustrate, consider the following "word" statements, which specify the contents of three words, whose format is described by the "field" statements of the previous section.

```
word 5->flda,017->fldb,0x4c->fldc
word 2->flda,12->fldb,29->fldc
word 0x3->flda,0xa->fldb,0177->fldc
```

This example also illustrates the use of octal and hexadecimal numbers. Numbers that begin with 0x are assumed to be in hexadecimal format. The digits a,b,c,d,e,f,A,B,C,D,E,F are acceptable in such numbers along with the usual 0-9. Numbers beginning with zero are assumed to be in octal format, and only the digits 0-7 are acceptable. Octal and hexadecimal numbers may be used wherever decimal numbers are acceptable.

Since it may not be convenient to specify the contents of every field on every word, the "field" statement allows a default value to be specified, which will be used if a particular "word" statement does not assign a value to a field. The following is an example of fields specified with default values.

```
flda:  field   width=3,position=0,default=4
fldb:  field   width=4,position=3,default=abc+def
fldc:  field   width=7,position=7,default=0x7f
```

If no default value is specified for a field, the field is assumed to have a default value of zero.

A shorthand notation, consisting of just the field name, can be used to assign the value "1" to a field of width 1. (This is normally considered to be activating a control signal.) For example, if abc is a one-bit field, "1->abc" and "abc" will produce the same result.

The "true" parameter can be used to extend this shorthand notation to multi-bit fields. Suppose the following field declaration has been made.

```
xyz:  field   width=10,position=10,default=5,true=15
```

With this declaration, "15->xyz" and "xyz" will produce the same result. There is no default value for the "true" parameter, so multi-bit fields with no "true" parameter must have values explicitly assigned to them, or must be allowed take their default value.

One bit fields can be declared as active-high or active-low. If a field is declared as active-low, the value assigned to it will be inverted before any output is actually done. Thus, if abc is a one bit field that specifies the value of a control signal, the expressions "1->abc" and "abc" will activate the control signal regardless of whether it is active-high or active low. This inversion also applies to the default value of an active-low field. Since active-high is the default for one bit fields, an explicit declaration of active high does not affect the output. Multi-bit fields *may not* be declared as active high or active low. The following is an example of an active high and an active low declaration.

```
abc:  field   position=12,active=low
def:  field   position=15,active=high
```

To reduce the confusion that active-high and active-low fields may cause, the constants "%t" and "%f" can be used to assign values to one-bit fields. The constant "%t" will turn a signal on, while "%f" will turn the signal off, regardless of whether it is active-high or active-low. When used in an expression, "%t" acts like a 1 and "%f" acts like a zero. These constants may be used wherever numbers are acceptable.

3.5 Required Fields

At times it may be desirable for the value of a certain field to specified by every "word" instruction. For example, some microcode sequencers require that a "next address" be specified with each instruction. In such a case it is possible to associate a field with a position in the operand field of the "word" instruction. To clarify this, consider the following instruction.

```
     word   abc,def,ghi
```

The command "abc" is at command-position 0, "def" is at command-position 1, and so forth. One associates a field with a certain command position by specifying the "cmdpos" parameter on the "field" statement. The following is an example.

```
flda:     field     width=5,position=0,cmdpos=0
fldb:     field     width=5,position=5,cmdpos=1
fldc:     field     width=5,position=10,cmdpos=2
```

Once these declarations have been made, each "word" statement must have at least three operands. These three operands must be either numbers or expressions that specify the value of the corresponding fields. The first three operands *must not* be of the form "expression->field_name". The following is a more complete example.

```
     flda:     field     width=5,position=0,cmdpos=0
fldb:  field   width=5,position=5,cmdpos=1
fldc:  field   width=5,position=10,cmdpos=2
       word    3,7,9
       word    a+b,c,0x5
```

A required operand may be forced to its default value by specifying a null value for the operand. The following statement specifies null values for three required operands.

```
     word ,,
```

As this example illustrates, a null value is simply an omitted value, with the requisite commas still in place.

3.6 Complex Commands

At times it will be necessary to specify the value of several fields in order to accomplish a single action. An example is an arithmetic operation in a microprogrammed computer, which usually requires the specification of operand sources, alu control signals, and result destination. The "command" statement can be used to group a set of commands together into a single complex command. The following is an example.

```
add_ab:  command  alu_add->alu,enab_a,enab_b,load_c
         word     add_ab
```

Once the command "add_ab" is defined, it can be used by many different "word" statements. Complex commands and simple commands may be mixed both on "word" statements and "command" statements. The order of the "command" statements and word statements does not matter, but "command" statements may not reference one another circularly.

3.7 ROM addresses.

When a label is used on a "word" statement, the rom address of the word defined by the "word" statement is assigned as the value of the label. The label may be used in an expression exactly as if it were an the label of an equate statement. Furthermore, an asterisk ("*") may be used in an expression to specify the rom address of the current "word" statement (or the next "word" statement if it appears in some other type of statement). The following illustrates the use of labels and rom addresses. in the following it is assumed that the microcode sequencer being used forces a jump on every instruction.

```
flda:  field   position=0
fldb:  field   position=1
jadr:  field   position=3,width=12,cmdpos=0
a:     word    b,flda,fldb
b:     word    c,fldb
c:     word    *-2,flda
```

Normally ROM addresses are assigned consecutively starting from zero. The "org" statement can be used to change the rom address of the next "word" instruction. An example of an "org" statement given below.

```
    org   *+10
```

This statement will leave a ten word "hole" in the ROM. When an expression appears on an "org" statement, all equates and rom addresses that are needed to evaluate the expression must appear before the "org" statement *in the text*. This is the only restriction on statement ordering.

When a hole is left in a ROM, the preprocessor adds null words to fill the hole. A null word is created by assigning every field its default value.

The preprocessor normally will expand a ROM to a size large enough to hold all defined words, including holes. The number of address bits will be enough to address all specified words, but no larger. If it is necessary to limit the rom to a specific number of address bits, the "size" statement must be used. An example of a size statement is given below.

```
    size   256
```

This statement will limit the rom to 8 address bits (no more, no less) and 256 words. The operand of the size statement may be an expression.

3.8 Adding New Opcodes

Some ROM sequencers have several commands that can be used to do conditional and unconditional jumps, subroutine calls, loops and so forth. In order to simplify the creation of microcode for these sequencers, the ROM preprocessor allows these commands to be defined as opcodes. The first step is to define a field as an opcode field. The following is an example of an opcode field.

```
opfld:  field   width=12,position=10,type=opcode
```

Next, each new opcode must be defined using an equate instruction, as illustrated below.

```
jump:   equ   1
cjump:  equ   2
return: equ   3
```

The values assigned to these opcodes must, of course, be meaningful to the microcode sequencer. Now, the defined opcodes may be used in place of the "word" opcode to specify a "word" statement. However when the "word" opcode is used, the opcode field will be assigned its default value. When one of the new opcodes is used, the opcode field will be assigned the value of the new opcode.

For claritly, defined symbols may be used in the place of the "word" opcode, even if no opcode field has been defined. In this case, the values assigned to the opcodes are immaterial.

3.9 ROM Output

Recall that the rom preprocessor prepares data for the FHDL compiler. Since the FHDL compiler cannot handle buses whose width is greater than 32, the output of the rom will be grouped into blocks of 32 bits starting from the left. Of course, the last (or only) block may have fewer than 32 bits. Depending on how the output of the ROM is used by the rest of the circuit, it may be convienient to group the outputs differently. The "output" statement can be used to do this. The operands of the output statement determine how the outputs of the ROM are grouped. One group of outputs will be created for each operand. Each operand must be an expression that gives the size of the group. The value of the expression must range from 1 to 32. An example of an output statement is given below.

```
    output    awid,bwid,cwid
```

The most convenient grouping of outputs is by field. Note that this grouping is logical rather than physical.

3.10 ROMs With Multiple Word Formats

At times it is necessary or useful to be able to specify more than one command format. The most obvious use of multiple formats would be when the ROM words actually have more than one physical format. Some microcode sequencers use different physical formats for jumps, conditional jumps and ordinary microinstructions. Another less obvious use of multiple formats is when you wish to give the illusion of multiple command formats, even though there is only one physical format. To illustrate, consider the case of providing conditional and unconditional

jumps. One way to implement unconditional jumps is to treat them as conditional jumps and use a condition that is always true. The same scheme could be used to implement ordinary micro-instructions as conditional jumps using a condition that is always false. For conditional jumps it would be convenient to have two required fields, condition, and jump address. On the other hand, it would be inconvenient to require explicit specification of conditions on unconditional jumps and ordinary instructions. Similarly on ordinary microinstructions *no* required operands would seem to be most appropriate.

The ROM preprocessor allows multiple formats to be declared, and allows each opcode to be associated with a particular format. Let us continue with the jump/conditional-jump/ordinary example, and assume that every rom word has a condition field and a next address field. The condition field is used by the sequencer to select a particular condition to be tested. Let us assume that 0 will select "always-false" and 1 will select "always-true." The first step is to define the opcodes, as follows.

```
jump:   equ   1
cjump:  equ   2
cont:   equ   3
```

In this example, we will not use an opcode field, so the three symbols could just as easilly be given the same value. We give them different values just in case we change our mind about having an opcode field. The next step is to assign each opcode to a format. The "format" statement is used for this purpose. The following illustrates.

```
fjump:   format   jump
fcjump:  format   cjump
fcont:   format   cont
```

The operand field of the "format" statement is a list of one or more command names. The label of the statement is the name of the format. All of the opcodes in the operand field will be associated with the format named in the label.

As fields are defined, they also must be assigned to a format. The following is the definition of the condition field for each of the three formats.

```
conda:  field  position=0,width=3,cmdpos=1,format=fcjump
condb:  field  position=0,width=3,default=1,
               format=fjump,type=constant
condc:  field  position=0,width=3,default=0,
               format=fcont,type=constant
```

Note that these three fields all occupy the same place in the rom word. When multiple formats are used, the rules for overlapping fields are modified somewhat. There may not be any overlapping fields *in a single format.* Furthermore, there must be a definition for each field *in each format.* The "type=constant" parameter may be used to prevent the user from assigning a value to a field in a particular format. The ROM preprocessor will issue an error message if a value is assigned to a constant field.

The next step in this example is to define the address fields. This field will be constant zero for ordinary microinstructions (cont opcode), it will be a required field for the other two opcodes. It must be specified in command position zero for jumps and command position one for conditional jumps. The following is the definition of these fields.

```
addra:  field  position=3,width=8,default=0,
               format=fcont,type=constant
addrb:  field  position=3,width=8,cmdpos=(0,1),
               format=(fjump,fcjump)
```

Note that only two field descriptions are required. The format parameter can be used to assign a field to more than one format, as illustrated above. When the "cmdpos" parameter is used with a field assigned to more than one format, a single number can be used to specify that the field appears in the same command position in each format, or a list of numbers can be used (one per format) to specify that the field appears at different command positions in each format.

For the sake of illustration, let us assume that all romwords have three one-bit control fields, regardless of format, and that none of these are required fields. We can define these fields as follows.

```
ctl_a:  field  position=11,format=(fcont,fjump,fcjump)
ctl_b:  field  position=12,format=(fcont,fjump,fcjump)
ctl_c:  field  position=13,format=(fcont,fjump,fcjump)
```

To complete the example, here are some statements that define rom words.

```
beg:    cont  ctl_a
        cont
        cont
        cjump  xcond,endit,ctl_c
        jump   beg,ctl_a
endit:  jump   *,ctl_b
```

The first format defined in the text is the default format. All fields without "format" parameters, are assigned to the default format. The default format is used for "word" opcodes and for opcodes that are not explicitly mentioned in a "format" statement. The default format is used for creating dummy words to fill holes left by "org" statements.

3.11 Include Statements

ROM coding can become quite tedious if every ROM description had to specify a complete set of formats. This would especially be true if you needed to create several different ROMS for use with the same sequencer and the same (or very nearly so) hardware. It may also be the case that you don't really want your microprogrammers to know all of the details of the field definitions and format definitions. The "include" statement provides a means for getting around these problems. An example of an "include" statement is given below.

```
    include   "/usr/fhdl/rom/rom1"
```

Note that the file name is enclosed in quotes. The quotes prevent the slashes from being interpreted as division signs. A full path name, of course, is taken to be the name of the file to be included. Something other than a full path name can be interpreted in two ways. If the name of an include library is placed on the command line, all include file names are assumed to be relative to the include library. (The include library must be a directory.) If no include directory is supplied, all include file names are assumed to be relative to the current directory.

3.12 Running the preprocessor

When you create your ROM you may include the ROM preprocessor code in the same file as your FHDL code. The ROM preprocessor code must begin with a statement of the following form.

```
my_rom:   rom
```

The ROM preprocessor code must end with the following statement.

```
          endrom
```

The "rom" and "endrom" statements replace the "circuit" and "endcircuit" statements used in FHDL code. The ROM preprocessor expects all input to be supplied from the standard input and produces all output on the standard output. Furthermore, all non-rom code is passed unchanged from the input to the output, so ROM preprocessor code can be mixed with FHDL in the same file. More than one ROM may be specified in the same file. The following command invokes the ROM preprocessor. (This command is invoked automatically if the fhdl command with the "-n" option is used.)

```
romasm  <testit.rom  >testit.ckt
```

The output of the rom preprocessor may also be directly piped into the FHDL compiler as follows.

```
romasm  <testit.rom  |  fhdl  >testit.c
```

If an include library is required, specify it as follows (rom.stuff is the name of a directory containing your include files).

```
romasm rom.stuff  <testit.rom  |  fhdl  >testit.c
```

3.13 Using ROMs

As stated above, the ROM preprocessor converts the preprocessor language into FHDL ROM specifications. One uses the ROM by calling it just as if it were an ordinary circuit. The name of the ROM is given by the label on the "rom" statement. To create microcode you must supply both the ROM specifications and the ROM sequencer specifications. The ROM preprocessor provides no microcode sequencing hardware, you must provide all sequencer hardware in FHDL. In addition you must provide a microinstruction register, and route the control signals from this register to the appropriate points in your design.

CHAPTER 4

The PLA Preprocessor

4.1 Overview

The FHDL PLA compiler is a preprocessor that provides a simple but powerful language for specifying the contents of a PLA. The format of the PLA preprocessor statements is modeled after that of FHDL statements. Each statement has a label field, an opcode, and an operand field. The label field begins at the first character of the statement and ends with a colon (:). The label field is optional for some types of statements, and manditory for others. The opcode, which is manditory for all statements, follows the label field, and must be separated from the label field by one or more spaces or tabs. If no label field is present, the opcode must be preceded by one or more spaces or tabs to signify that the label field is omitted. The operand field follows the opcode and must be separated from the opcode by one or more spaces or tabs. The operand field consists of one or more operands separated by commas. The format of an operand depends on the type of statement. The operand field, and the statement, end with a newline (return key) or a semicolon(;). The operand field of a statement can be continued to the next line by ending a line with a comma. Spaces and tabs are not allowed in the operand field, except preceeding or following a comma.

Labels may contain upper and lower case letters, numbers, underlines and periods. Labels may not duplicate a PLA preprocessor keyword, nor may duplicate labels be defined, regardless of type. Case is significant for labels, so Lab1, LAb1, and lAb1 are three different labels. On the other hand, case is *not significant* in keywords, so pla, Pla and PLA are all the same keyword.

Once a PLA description has been completed, it must be run through the PLA preprocessor before being compiled by the FHDL compiler. Although FHDL provides a method for describing the contents of a PLA, the method is not flexible enough for specifying and debugging complex PLAs. Nevertheless, the final section of this report contains a description of

the FHDL native method for specifying PLA contents. The PLA preprocessor converts the preprocessor language into FHDL native mode instructions.

4.2 Specifying Fields.

Each PLA wordline consists of two sections, the AND plane section and the OR plane section. Each section is broken into one or more fields. AND plane fields contain inputs that will be tested by the PLA. OR plane fields may contain many different types of data, some examples of which are data, control signal values, and so forth. Each field must be declared using a statement similar to the following.

```
new_state:  field   width=12,position=15
```

This statement declares an OR plane field named "new_addr" which has a width of 12 bits and begins at bit 15 of the OR plane. The bits of each word are numbered from the left starting with zero. Each field in the OR plane must be declared, even if it is never used. The order of the "field" statements is not important, since each statement has both a width and position associated with it. If the "width" specification is omitted, a width of one is assumed. If the "position" specification is omitted, a position of zero is assumed. Or plane fields may not overlap.

AND plane fields are declared in a similar fashion as the following declaration illustrates.

```
cond_one:  field   width=1,position=15,type=input
```

The only difference between an AND plane declaration and an OR plane declaration is the presence of the parameter "type=input". The "position" parameter indicates a position within the AND plane. All AND plane fields must be declared even if they are always "don't care." AND plane fields may not overlap.

4.3 Using Equates.

The width and position parameters of the "field" statement can be rather difficult to keep track of if the number and position of your fields changes often. (This is often the case during PLA development.) The "equ" statement can be used to simplify the process of adding new fields, and changing the size of existing ones. Suppose you have the following three OR plane fields, declared as in the previous section.

```
flda:     field     width=3,position=0
fldb:     field     width=4,position=3
fldc:     field     width=7,position=7
```

If you want to add a field between flda and fldb, you must change the position of fldb and fldc. The same is true if you change the size of flda. The following is the same three fields coded with equates.

```
flda:  field   width=awid,position=apos
fldb:  field   width=bwid,position=bpos
fldc:  field   width=cwid,position=cpos
awid:  equ     3
bwid:  equ     4
cwid:  equ     7
apos:  equ     0
bpos:  equ     awid+apos
cpos:  equ     bwid+bpos
```

Using this technique, one need only concern oneself with the width and the order of each field. Positions are calculated automatically by the preprocessor.

The right hand side of an equate may be an arbitrary expression involving constants; the names of other equates; the operators +, -, *, and /; and parenthesis. The order of the equate statements does not matter, as long as they do not reference one another cyclically. Expressions may also be used to specify field widths and positions, as illustrated below.

```
flda:  field   width=awid,position=0
fldb:  field   width=bwid,position=bpos
fldc:  field   width=awid+bwid,position=bwid+bpos
awid:  equ     3
bwid:  equ     4
apos:  equ     0
bpos:  equ     awid+apos
```

This example illustrates the rule that any place a number is acceptable, an expression is also acceptable.

4.4 Specifying the Value of AND and OR Plane Fields.

The "word" statement is used to specify the contents of one or more PLA wordlines. The label field specifies the contents of the AND plane while the operand field specifies the contents of the OR plane. The labels of "word" statements are called "conditions" while the operands are called commands. A simple condition is specified using an expression of the following form.

```
field_name=expression
```

The field_name must be the name of an AND plane field, while the expression must supply the value against which the field will be tested. A simple command is specified using an expression of the following form.

```
expression->field_name
```

The field_name must be the name of an OR plane field, while the expression supplies the value that will be stored in the field. To illustrate, let us add the following AND plane field declarations to the three OR plane declarations specified in the last section.

```
cnda:  field   position=0,width=1,type=input
cndb:  field   position=1,width=1,type=input
```

The following "word" statements specify the contents of three wordlines.

```
cnda=1:  word  5->flda,017->fldb,0x4c->fldc
cndb=0:  word  2->flda,12->fldb,29->fldc
cnda=0:  word  0x3->flda,0xa->fldb,0177->fldc
```

This example also illustrates the use of octal and hexadecimal numbers. Numbers that begin with 0x are assumed to be in hexadecimal format. The digits a,b,c,d,e,f,A,B,C,D,E,F are acceptable in such numbers along with the usual 0-9. Numbers beginning with zero are assumed to be in octal format, and only the digits 0-7 are acceptable. Octal and hexadecimal numbers may be used wherever decimal numbers are acceptable.

Since it may not be convenient to specify the contents of every OR plane field on every word, the "field" statement allows a default value to be specified, which will be used if a particular "word" statement does not assign a value to a field. The following is an example of fields specified with default values.

```
flda:  field   width=3,position=0,default=4
fldb:  field   width=4,position=3,default=abc+def
fldc:  field   width=7,position=7,default=0x7f
```

If no default value is specified for an OR plane field, the field is assumed to have a default value of zero.

No default value may be specified for AND plane fields. An AND plane field is assumed to have a default value of "don't care". The only way to specify a "don't care" value for an AND plane field is to allow it to take its default value.

A shorthand notation, consisting of just the field name, can be used to assign the value "1" to an OR plane field of width 1. (This is normally considered to be activating a control signal.) For example, if abc is a one-bit field, "1->abc" and "abc" will produce the same result.

The "true" parameter can be used to extend this shorthand notation to multi-bit OR plane fields. Suppose the following field declaration has been made.

```
xyz:  field   width=10,position=10,default=5,true=15
```

With this declaration, "15->xyz" and "xyz" will produce the same result. There is no default value for the "true" parameter, so multi-bit OR plane fields with no "true" parameter must have values explicitly assigned to them, or must be allowed take their default value.

One bit OR plane fields can be declared as active-high or active-low. If a field is declared as active-low, the value assigned to it will be inverted before any output is done. Thus, if abc is a one bit field that specifies the value of a control signal, the expressions "1->abc" and "abc" will activate the control signal regardless of whether it is active-high or active low. This inversion also applies to the default value of an active-low field. Since active-high is the default for one bit fields, an explicit declaration of active high does not affect the output. Multi-bit OR plane fields and AND plane fields *may not* be declared as active high or active low. The following is an example of an active high and an active low declaration.

```
abc:   field   position=12,active=low
def:   field   position=15,active=high
```

To reduce the confusion that active-high and active-low fields may cause, the constants "%t" and "%f" can be used to assign values to one-bit fields. The constant "%t" will turn a signal on, while "%f" will turn the signal off, regardless of whether it is active-high or active-low. When used in an expression, "%t" acts like a 1 and "%f" acts like a zero. These constants may be used wherever numbers are acceptable.

4.5 Required OR Plane Fields

At times it may be desirable for the value of certain OR plane fields to specified by every "word" instruction. It is possible to associate a field with a position in the operand field of the "word" instruction and thereby force it to be specified for every wordline. To clarify this, consider the following instruction.

```
a=1:  word  abc,def,ghi
```

The command "abc" is at command-position 0, "def" is at command-position 1, and so forth. One associates a field with a certain command position by specifying the "cmdpos" parameter on the "field" statement. The following is an example.

```
flda:  field   width=5,position=0,cmdpos=0
fldb:  field   width=5,position=5,cmdpos=1
fldc:  field   width=5,position=10,cmdpos=2
```

Once these declarations have been made, each "word" statement must have at least three operands. These three operands must be either numbers or expressions that specify the value of the corresponding fields. The first three operands *must not* be of the form "expression->field_name". The following is a more complete example.

```
flda:  field   width=5,position=0,cmdpos=0
fldb:  field   width=5,position=5,cmdpos=1
fldc:  field   width=5,position=10,cmdpos=2
b=2:   word    3,7,9
c=0:   word    a+b,c,0x5
```

A required operand may be forced to its default value by specifying a null value for the operand. The following statement specifies null values for three required operands.

```
a=1:  word   ,,
```

As this example illustrates, a null value is simply an omitted value, with the requisite commas still in place.

4.6 Complex Commands

At times it will be necessary to specify the value of several fields in order to accomplish a single action. An example is an arithmetic operation in a microprogrammed computer, which usually requires the specification of operand sources, alu control signals, and result destination. The "command" statement can be used to group a set of commands together into a single complex command. The following is an example.

```
add_ab:  command  alu_add->alu,enab_a,enab_b,load_c
a=1:     word     add_ab
```

Once the command "add_ab" is defined, it can be used by many different "word" statements. Complex commands and simple commands may be mixed both on "word" statements and "command" statements. The order of the "command" statements and word statements does not matter, but "command" statements may not reference one another circularly.

4.7 Complex Conditions

At times it will be necessary to specify the value of several fields in order to detect a certain condition. There are two ways to specify complex conditions. First, the condition on a "word" statement may contain the logical operators "&" (AND) and "|" (OR). AND has precedence over OR, but parentheses may be used to override the precedence. The second method is to use a "condition" statement to define a condition, and then use the name of the condition on the "word" statement. The following is an example.

```
cnd_abc:  condition  a=1&b=2&c=0
cnd_abc:  word       0x1->flda
```

Once the condition "cnd_abc" is defined, it can be used by many different "word" statements. Defined conditions such as "cnd_abc" and simple conditions such as "a=1" may be combined in a single logical expression both on a "word" statement and on a "condition" statement. The order of the "condition" statements and the "word" statements does not matter, but "condition" statements may not reference one another circularly.

When a complex condition using the OR connective appears on a "word" statement, the condition is reduced to sum-of-products form and one wordline is generated for each product term. This must be done because the AND plane is incapable of executing the OR function. This procedure pushes the "OR" connective processing into the OR plane. Thus a single "word" statement may generate many wordlines.

4.8 Limiting the Number of Wordlines

If you must limit the number of wordlines in your PLA, include the following statement in your PLA description.

```
    size  <expression>
```

The expression may be an explicit number, or it may be a complex expression. In either case an error message will be issued if the number of wordlines exceeds the expression on the "size" statement. Only one size statement per PLA description is allowed.

4.9 Adding New Opcodes

For clarity the PLA preprocessor allows the "word" opcode to be replaced by one or more user defined opcodes. The opcode can be used to specify the value of an OR plane field. The first step is to define an OR plane field as an opcode field. The following is an example of an opcode field.

```
opfld:  field   width=12,position=10,type=opcode
```

Next, each new opcode must be defined using an equate instruction, as illustrated below.

```
add1:   equ   1
subt1:  equ   2
noop:   equ   3
```

Now, the defined opcodes may be used in place of the "word" opcode to specify a "word" statement. When the "word" opcode is used, the opcode field will be assigned its default value. When one of the new opcodes is used, the opcode field will be assigned the value of the new opcode.

Defined symbols may be used in the place of the "word" opcode, even if no opcode field has been defined. In this case, the values assigned to the opcodes are immaterial.

4.10 The Begin and End Statements

There are cases where a group of wordlines must all specify the same condition and the same set of commands. The "begin" and "end" statements allow conditions and commands to be specified for sets of wordlines without repetitive coding. To illustrate consider the following example.

```
a=1:  begin   ADD->next_state
b=2:  word    1->sub
c=0:  word    2->sub
d=3:  word    SUBT->next_state,0->sub
      end
```

Without the "begin" and "end" statements this example would be written as follows.

```
a=1&b=2:  word   ADD->next_state,1->sub
a=1&c=0:  word   ADD->next_state,2->sub
a=1&d=3:  word   SUBT->next_state,0->sub
```

The condition on a "begin" statement is ANDed with the conditions on the "word" statements contained between the "begin" and "end" statements. Conditions on the "word" statements *may not* override conditions on the "begin" statement. Commands on the "begin" statement are combined with commands on the "word" statement. The commands on the "word" statement *may* override the commands on the "begin" statement. "Begin" statements may be nested arbitrarily.

4.11 Grouping Input and Output Fields

Recall that the PLA preprocessor prepares data for the FHDL compiler. Since the FHDL compiler cannot handle buses whose width is greater than 32, the inputs and the outputs of the PLA will be grouped into blocks of 32 bits starting from the left. Of course, the last (or only) block may have fewer than 32 bits. Depending on how the inputs and outputs of the PLA are used by the rest of the circuit, it may be convienient to group them differently. The "input" and "output" statements can be used to do this. The operands of the "input" and "output" statements determine how the inputs and outputs of the PLA are grouped. One group of inputs or outputs will be created for each operand. Each operand must be an expression that gives the size of the group. The value of the expression must range from 1 to 32. An example of an input and an output statement is given below.

```
output  awid,bwid,cwid
input   condawid,condbwid
```

The most convenient grouping of inputs and outputs is by field. Note that this grouping is logical rather than physical.

4.12 Multiple OR Plane Formats

At times it may be convienient to be able to specify more than one OR-plane format. The most obvious use of multiple formats would be when the OR plane actually has more than one

physical format. In this case one OR plane field is being used as a format indicator for the remainder of the OR plane portion of the wordline. Another less obvious use of multiple formats is when you wish to give the illusion of multiple command formats, even though there is only one physical format. To illustrate, consider the case of providing a set of one-operand commands, a set of two-operand commands and a set of three-operand commands in the same PLA description. Let us assume that there are three OR plane fields and two AND plane fields. The three-operand commands will supply values for all three fields, while the one- and two-operand fields will cause default values to be assigned to the unspecified fields.

The PLA preprocessor allows multiple formats to be declared, and allows each opcode to be associated with a particular format. The first step is to define the opcodes, as follows.

```
oneop:    equ   1
twoop:    equ   2
threeop:  equ   3
```

In this example, we will not use an opcode field, so the three symbols could just as easilly be given the same value. We give them different values just in case we change our mind about having an opcode field. The next step is to assign each opcode to a format. The "format" statement is used for this purpose. The following illustrates.

```
f1:  format   oneop
f2:  format   twoop
f3:  format   threeop
```

The operand field of the "format" statement is a list of one or more opcodes. The label of the statement is the name of the format. All of the opcodes in the operand field will be associated with the format named in the label.

As fields are defined, they also must be assigned to a format. The following is the definition of the field that is used only by the three operand format.

```
fld3a:  field  position=0,width=3,cmdpos=2,format=f3
fld3b:  field  position=0,width=3,default=1,
               format=f2,type=constant
fld3c:  field  position=0,width=3,default=0,
               format=f1,type=constant
```

Note that these three fields all occupy the same place in the OR plane. When multiple formats are used, the rules for overlapping fields are modified somewhat. There may not be any overlapping fields *in a single format.* Furthermore, there must be a definition for each field *in each format.* The "type=constant" parameter may be used to prevent the user from assigning a value to a field in a particular format. The PLA preprocessor will issue an error message if a value is assigned to a constant field.

The next step in this example is to define the field that will be shared by two operand and three operand commands. This field will be constant zero for one-operand opcodes. It must be specified in command position zero for two-operand opcodes and command position one for three-operand opcodes. The following is the definition of these fields.

```
fld2a:  field  position=3,width=8,default=0,
               format=f1,type=constant
fld2b:  field  position=3,width=8,cmdpos=(0,1),
               format=(f2,f3)
```

Note that only two field descriptions are required. The format parameter can be used to assign a field to more than one format, as illustrated above. When the "cmdpos" parameter is used with a field assigned to more than one format, a single number can be used to specify that the field appears in the same command position in each format, or a list of numbers can be used (one per format) to specify that the field appears at different command positions in each format. Now let us define the field that is shared between all three formats.

```
fld3:  field  position=11,width=1,cmdpos=(0,1,0),
              format=(f1,f2,f3)
```

The following is the definition of the two AND plane fields. Note that AND plane fields have a single format.

```
ca:  field    position=0,width=3,type=input
cb:  field    position=3,width=4,type=input
```

To complete the example, here are some statements that define wordlines.

```
ca=1:       oneop    1
ca=2:       twoop    5,0
cb=3:       threeop  0,8,3
ca=4&cb=2:  threeop  1,9,2
cb=4:       oneop    0
cb=7:       twoop    6,1
```

The first format defined in the text is the default format. All fields without "format" parameters, are assigned to the default format. The default format is used for "word" opcodes and for opcodes that are not explicitly mentioned in a "format" statement.

4.13 Include Statements

PLA coding could become quite tedious if every PLA description had to specify a complete set of formats. This would especially be true if you needed to create several different PLAs to control the same hardware. It may also be the case that you don't really want your PLA coders to know all of the details of the field definitions and format definitions. The "include" statement provides a means for solving these problems. An example of an "include" statement is given below.

```
    include   "/usr/fhdl/pla/pla1"
```

Note that the file name is enclosed in quotes. The quotes prevent the slashes from being interpreted as division signs. A full path name, of course, is taken to be the name of the file to be included. Something other than a full path name can be interpreted in two ways. If the name of an include library is placed on the command line, all include file names are assumed to be relative to the include library. (The include library must be a directory.) If no include directory is supplied, all include file names are assumed to be relative to the current directory.

4.14 Running the preprocessor

When you create your PLA you may include the PLA preprocessor code in the same file as your FHDL code. The PLA preprocessor code must begin with a statement of the following form.

```
my_pla:   pla
```

The PLA preprocessor code must end with the following statement.

```
          endpla
```

The "pla" and "endpla" statements replace the "circuit" and "endcircuit" statements used in FHDL code. The PLA preprocessor expects all input to be supplied from the standard input and produces all output on the standard output. Furthermore, all non-rom code is passed unchanged from the input to the output, so PLA preprocessor code can be mixed with FHDL in the same file. More than one PLA may be specified in the same file. The following command invokes the PLA preprocessor. (This command is invoked automatically when the fhdl command with the "-n" option is used.)

```
plasm <testit.pla >testit.ckt
```

The output of the PLA preprocessor may also be directly piped into the FHDL compiler as follows.

```
plasm <testit.pla | fhdl >testit.c
```

If an include library is required, specify it as follows (pla.stuff is the name of a directory containing your include files).

```
plasm pla.stuff <testit.pla | fhdl >testit.c
```

4.15 Using PLAs

As stated above, the PLA preprocessor converts the preprocessor language into FHDL PLA specifications. One uses the PLA by calling it just as if it were an ordinary circuit. The name of the PLA is given by the label on the "pla" statement. To create a microcoded PLA you must supply both the PLA specifications and the PLA sequencer specifications.

CHAPTER 5

The MACRO Preprocessor

5.1 Overview

The FHDL macro processor was designed to provide a convenient method for expanding the FHDL language. Currently the FHDL compiler recognizes functional blocks such as registers and ALUs. The FHDL macro processor provides a method for defining new functional blocks and for implementing other language extensions. The macro processor can also be used to extend the capabilities of the FHDL ROM and PLA preprocessors.

The syntax of an FHDL macro statement is identical to that of an ordinary FHDL statement. Each statement contains a label field, an opcode, and an operand field. The label field starts at the first character of the statement and ends with a colon (:). The label field is optional for most macro statements. The opcode, which is mandatory for all statements, must be separated from the label field by one or more spaces or tabs. If there is no label field, the opcode must be preceded by one or more spaces or tabs to signify that the label field is missing. The operand field, which must be separated from the opcode by one or more spaces or tabs, consists of one or more operands separated by commas. The operand field is optional for some statements. Every statement must be terminated by a newline character (return key) or a semicolon (;). A statement normally occupies only one line, but if the operand field ends with a comma, the operand field is assumed to be continued on the next line.

All language extensions are implemented in the form of macro instructions. Macro definitions may be included with the FHDL text that uses them, or they may be placed in one or more macro libraries. Placement of macro instructions is discussed in detail in the following sections.

5.2 A Simple Macro Definition

Suppose you want to define a functional block that will invert three signals simultaneously. An example of the functional block, as it would appear in the FHDL text, is given below.

```
abc: not3 (a,b,c),(abar,bbar,cbar)
```

The following macro definition could be used to define the "not3" functional block. (This block is simple enough to be defined using FHDL subnetworks, but we will quickly move to more complicated examples.)

```
not3:  'macro
       'input   'a,'b',c
       'output  'd,'e,'f
       not      'a,'d
       not      'b,'e
       not      'c,'f
       'endmacro
```

This example contains several instances of macro keywords and variables. All macro keywords begin with the character ', as do all macro variables. This example also contains occurrences of the two types of statements recognized by the macro processor. Any statement that has a macro keyword for an opcode is a preprocessor statement. Any other statement is a text statement. Preprocessor statements are used to define macros and variables, and to control the processing of text statements. Text statements are used to generate text. In the preceding example, the first three statements are preprocessor statements, the next three are text statements, and the last statement is a preprocessor statement.

All macro definitions must begin with a 'macro statement and end with an 'endmacro statement. The label on the 'macro statement gives the name of the macro. Macro definitions may not be nested, and a macro *may not* generate the definition of another macro. The following statement, which invokes the "not3" macro, is an example of a *macro call*.

```
qed:   not3   (x,y,z),(q,e,d)
```

The macro preprocessor removes all macro calls and replaces them with the text generated by the text statements of the macro definition. In this case the result will be as follows.

```
not   x,q
not   y,e
not   z,d
```

Note that the label "qed" has been thrown away. The two macro statements 'input and 'output define macro variables. The value of these variables is taken from the macro call. The value of variable 'a will be the *text* of the first element of the input list of the macro call, while the value of the variable 'f will be the *text* of the third element of the output list. Variables defined using 'input and 'output statements are of the type "string." As will be explained below, macro variables may be of three different types, string, integer, and list. The scope of macro variables is the macro definition. When the name of a macro variable is encountered in a text statement, the value of the variable replaces the name. Only one scan of each text statement is done.

The number of inputs and outputs in the macro call need not match the number of inputs and outputs declared in the macro definition. Extra inputs and outputs are ignored, while variables corresponding to missing inputs and outputs contain the null string.

5.3 A More Complicated Example

If it is necessary to guarantee that "not3" actually has three inputs and three outputs, the macro definition must explicitly perform the test, as illustrated in the next example.

```
not3: 'macro
      'inputs  'a,'b'c
      'outputs 'd,'e,'f
      'if      'count('ilist)'!=3
      'error   s,"not3 should have 3 inputs"
      'exit
      'endif
      'if      'count('olist)'!=3
      'error   s,"not3 should have 3 outputs"
      'exit
      'endif
      not      'a,'d
      not      'b,'e
      not      'c,'f
      'endmacro
```

This macro definition illustrates the use of several new features. First is the conditional statement 'if. An 'if statement must have one operand, and must be followed in the text by an 'endif statement. The operand is usually a conditional expression as shown in the example. The operators "'!=", "'==", "'<=", "'>=", "'<", and "'>" may be used to form conditional tests. (Note that each of these operators begins with an apostrophe.) The operators "&&", "||", and "!" (and, or, not) may also be used to form complex conditions. Of course, parentheses are also acceptable. If the condition is true, then the statements between the 'if statement and the 'endif statement are processed, otherwise they are skipped. A general arithmetic expression may be used as the operand of an 'if statement. When this is done, zero represents false and nonzero represents true. 'if statements may be nested arbitrarily. Each 'if statement must have a corresponding 'endif statement. As will be explained below, an 'if statement can be combined with an 'else statement and/or one or more 'elif (else-if) statements.

WARNING!! The operators "!=", "==", "<=", ">=", "<", and ">" are also treated as legal operators by the macro processor. HOWEVER, these operators are considered to be comparison operators used by other FHDL parsers. These operators will be passed through unchanged to the output. Check each of your comparison operators carefully to be sure that you have not used the wrong type. To guard against misuse of comparison operators, the macro processor will recognize the operators "$NE", "$EQ", "$LE", "$GE", "$LT" and "$GT" as valid comparison operators. Exclusive use of these operators will guard against problems with missing apostrophes.

The variables 'ilist and 'olist are built-in list-type variables whose values are the input list and output list of the macro call. The built-in function 'count may be applied to any list-type variable to obtain the number of items in the list.

The 'error statement is used to send error messages to the user. The first operand gives the severity of the message (i=information only, w=warning, s=severe(output terminated), t=terminal(immediate termination)). Anything other than these four letters will be interpreted as "t". The second operand is the text of the message to be sent to the user. The macro processor

will add line numbers and file names to the message. Note that the message must be enclosed in quotes because it contains spaces. The 'exit statement causes processing of the current macro instruction to terminate. Processing of the input continues.

Now suppose we wish to enhance the not3 macro so that it will accept an arbitrary number of inputs and outputs, as long as the number of inputs and outputs is the same. We will call the new macro "not.m".

```
not.m:  'macro
        'if      'count('ilist)'!='count('olist)
        'error   s,"not.m inputs and outputs don't match"
        'exit
        'endif
        'int     'i
        'assign  'i,0
        'while   'i'<'count('ilist)
        not      'ilist('i),'olist('i)
        'assign  'i,'i+1
        'endwhile
        'endmacro
```

This macro illustrates several new features. First is the declaration of work variables. The 'int statement, which may have several arguments, declares one or more variables of type integer. Work variables may be of type integer, or of type string (declared with an 'str statement), or of type list (declared with a 'list statement). The 'assign statement can be used to assign new values to work variables.

The 'while statement, which must be followed by an 'endwhile statement, is used to repeatedly process a collection of statements. The operand of a 'while statement follows the same rules as the operand of an 'if statement. The statements between the 'while statement and its corresponding 'endwhile are processed repeatedly as long as the condition remains true. The rule stated above, that each statement is scanned for variables only once, must be modified for loops. Each statement in a loop is scanned only once *each time it is processed*. Processing a text statement does not change the statement, it merely causes output to be produced according to the instructions contained in the statement. While loops may be nested arbitrarily and may be combined arbitrarily with 'if statements.

The 'assign statement causes a value to be assigned to a variable. An 'assign statement has two operands, the first of which is the name of the work variable whose value will be changed. The second operand is an expression that, when evaluated, will produce the new value of the variable. The assign operator "<-" may be used instead of the 'assign statement. Thus the following two statements are equivalent.

```
'assign  'i,'i+1
'i<-'i+1
```

An expression may be a constant, such as 0, or the name of a variable such as 'i, or a complex expression involving constants, variables, operators, and built-in functions. A complete list of all operators and built-in functions will be found in the appendices. Among these operators are the arithmetic operators "+", "-", "*", and "/", and the comparison operators discussed above. The comparison operators return a 1 for true and a 0 for false when used as arithmetic expressions.

The built-in variables 'ilist and 'olist may also be used as built-in functions. When they are used as functions, they take one numeric operand that indicates a position in the input (or output) list. Positions are numbered from zero. The function returns the text of the operand in the specified position in the input (or output) list of the macro call. If the operand is greater or equal to the number of items in the input (or output) list, the null string is returned.

A 'while loop such as that illustrated in the previous example can be expressed more simply as a 'for loop, as illustrated in the following example.

```
not.m:  'macro
        'if      'count('ilist)'!='count('olist)
        'error   s,"not.m inputs and outputs don't match"
        'exit
        'endif
        'int     'i
        'for     'i<-0,'i'<'count('ilist),'i<-'i+1
        not      'ilist('i),'olist('i)
        'endfor
        'endmacro
```

The format of a 'for statement is given below.

```
    'for    exp1,exp2,exp3
    <body>
    'endfor
```

This statement is functionally equivalent to the following.

```
    exp1
    'while    exp2
    <body>
    exp3
    'endwhile
```

The first expression initializes loop values, the second is the condition under which the loop will continue to iterate, and the third expression is used to update loop variables for the next iteration. Any of the three expressions may be parenthesized lists of expressions, as the following "two-variable" loop illustrates.

```
  'for  ('i<-0,'j<-0),
        'i'<'ilim&&'j'<'jlim,
        ('i<-'i+1,;j<-'j+1)
```

Loops can be terminated early by use of the 'break and 'continue statements. Neither of these statements takes any arguments. Executing the 'break statement causes the innermost loop to be immediately terminated.

Executing the 'continue statement causes the *current iteration* of the innermost loop to be immediately terminated. If the innermost loop is a 'for loop, the loop variables are updated. If the continuation condition of the 'for or 'while loop is still true, the next iteration of the loop begins.

5.4 Accessing The Argument List

The statement format accepted by the macro preprocessor is more flexible than that accepted by the FHDL compiler, in that a text statement may have any number of arguments. In particular, a *macro call* may have an arbitrary number of arguments. For example, let us assume that when the "not.m" macro defined in section 3 is used, the outputs are all of the form <name>.bar where <name> is the name of the corresponding input. To save coding time, we could eliminate the need to specify the outputs, since the name of the output can be deduced from the name of the input. Again, to save coding time, we will allow the user to specify just the input list, *without the parentheses*. Thus the user would code the following statement.

```
not.m    a,b,c
```

This statement would produce the following output.

```
not      a,a.bar
not      b,b.bar
not      c,c.bar
```

The definition of "not.m" would be modified as illustrated below.

```
not.m: 'macro
       'int     'i
       'assign  'i,0
       'while   'i'<'count('args)
       not      ''i,''i#".bar"
       'assign  'i,'i+1
       'endwhile
       'endmacro
```

The built-in variable 'args is a list-type variable whose value is the entire argument list of the macro call. The individual arguments are accessed using variables of the form '0, '1, '2, and so forth. The value of the variable '0 is the text of the first argument in the argument list, the value of '1 is the text of the second argument, and so forth. The variables '0 and 'ilist are identical, as are '1 and 'olist. Although all argument variables are technically of type list, the coercion rules of the preprocessor will cause these variables to assume the type string or the type list, as appropriate to their value and usage.

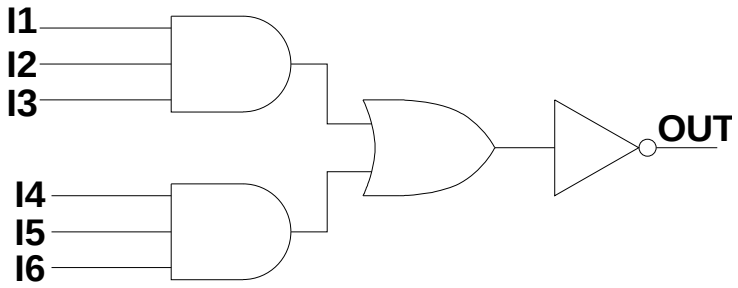
The operator ' is the indirection operator. When the indirection operator is applied to an expression, the value of the expression is used as a variable name and the value of the indirectly referenced variable is returned. The ' character is prepended to the value of the expression, so the expression *must not* include the leading ' character. Although this looks like multipass substitution, it is not.

Finally, the "#" operator is the concatenation operator. Unlike some macro processors, the FHDL macro processor *does not* provide automatic concatenation of consecutive expressions.

5.5 Generating Net Names

In the examples given so far, all net names were either supplied as operands of the macro call, or constructed from values supplied by the macro call. In a sense, these names are the "primary inputs and outputs" of the macro. If additional net names are required beyond those specified as primary inputs and outputs, great care must be taken to avoid duplicating a net name.

This is not of concern when one uses an FHDL subnet, because the FHDL circuit flattener renames these nets in such a way as to produce a unique name each time the subnet is called. The macro preprocessor does not provide such a mechanism, although several different mechanisms can be explicitly programmed. This section will give examples of several different methods for providing unique net names. Each of these examples will focus on the problem of creating a AOI32 gate out of AND, OR, and NOT gates. The logic diagram of the AOI32 gate is given below.



The macro to generate gates of this type is given below.

```
AOI32: 'macro
       'inputs   'i1,'i2,'i3,'i4,'i5
       'outputs  'out
       and       ('i1,'i2,'i3),("tempa"#'calln)
       and       ('i4,'i5),("tempb"#'calln)
       or        (tempb#'calln,tempb#'calln),(tempc#'calln)
       not       tempc#'calln,'out
       'endmacro
```

The value of the built-in variable 'calln is the number of macro calls that have been processed so far, not counting those that were encountered during the processing of the current macro. The variable 'calln remains constant throughout the processing of the current macro. The value of 'calln is guaranteed to be different for each macro call.

This example also illustrates two different methods for specifying strings. A string may be enclosed in quotes thus: "this is a string". If a string is enclosed in quotes it may contain any character except the null character (zero character). If a string contains only upper and lower case letters, digits, periods and underlines, the quotes may be omitted.

The next method of handling internal signals emulates the method used by the FHDL circuit flattener.

```
AOI32:'macro
      'inputs  'i1,'i2,'i3,'i4,'i5
      'outputs 'out
      and      ('i1,'i2,'i3),('label#.tempa)
      and      ('i4,'i5),('label#.tempb)
      or       ('label#.tempb,'label#.tempb),('label#.tempc)
      not      'label#.tempc,'out
      'endmacro
```

The the value of the built-in variable 'label is the text of the label field of the macro call. If there is no label the value of 'label is the null string. This, of course, can cause problems, so it might be wise to make the following change to the above macro.

```
AOI32:  'macro
        'inputs  'i1,'i2,'i3,'i4,'i5
        'outputs 'out
        'str     'tlab
        'if      'label'==""
        'assign  'tlab,X#'calln
        'else
        'assign  'tlab,'label
        'endif
        and      ('i1,'i2,'i3),('tlab#.tempa)
        and      ('i4,'i5),('tlab#.tempb)
        or       ('tlab#.tempb,'tlab#.tempb),('tlab#.tempc)
        not      'tlab#.tempc,'out
        'endmacro
```

This example illustrates the use of work variables of type string. The next example makes use of global variables as well as string variables.

```
AOI32:  'macro
        'inputs   'i1,'i2,'i3,'i4,'i5
        'outputs  'out
        'str      't1,'t2,'t3
        'gblint   'label_number
        'assign   't1,L#'label_number
        'assign   't2,L#'label_number+1
        'assign   't3,L#'label_number+2
        'assign   'label_number,'label_number+3
        and       ('i1,'i2,'i3),('t1)
        and       ('i4,'i5),('t2)
        or        ('t1,'t2),('t3)
        not       't3,'out
        'endmacro
```

This example illustrates the use of global variables. The integer variable 'label_number is declared to be global. Global variables retain their values from call to call, while local variables are initialized at the beginning of each call. Local variables are declared using 'int, 'str, and 'list statements, while global variables are declared using 'gblint, 'gblstr, and 'gbllist statements. All macros that declare the same global variable share access to the variable. A macro has no knowledge of undeclared global variables, so it is permissible for one macro to declare a local variable with the same name as a global variable declared by another macro. Global integers are initialized to zero, global strings are initialized to the null string, and global lists are initialized to the null list. These initializations are also done for local variables at the beginning of each call.

.The second feature illustrated by this example is the coercion between integers and strings. When an integer variable is used as a string, it is automatically converted to a string of decimal digits without leading zeros (unless the value is zero). Negative numbers have a leading minus

sign. A string that contains only digits can be used as a number. It will be coerced to an integer before use. The assumed radix is 10, regardless of leading zeros.

5.6 Function Calls

Because the FHDL macro processor provides variables, if statements and while statements, it has the full power of a general purpose programming language. One important feature of general purpose programming languages is the ability to define subroutines. The ability to nest macro calls is a type of subroutine feature, but there are times when the ability to define a function that can be used in an expression is helpful. Any macro may be used as a function by following the macro name with a parenthesized list of arguments. The following macro computes the "factorial" function recursively.

```
factorial: 'macro
           'if      '0'==0||'0'==1
           'assign  'factorial,1
           'else
           'assign  'factorial,'0*factorial('0-1)
           'endif
           'endmacro
```

A macro that is used as a function should not contain text statements. As this example illustrates, a macro returns a value by assigning a value to a variable whose name is identical to the macro name. This variable, which is of type string, is a local variable that is created at the beginning of the function call. If the macro is called normally, the variable will not exist, therefore it is a good idea to avoid using a macro both as a function and as an ordinary macro. If, for some reason, you absolutely must do this, you can use the built-in variable 'callt to determine whether the macro has been called as a function or as a normal macro. When used as a string variable, 'callt has the value "FUNC" for a function call and the value "STMT" for a normal call. When used as a condition it has the value true for function calls and false for normal calls. When used in an arithmetic expression, it has the value one for function calls and zero for normal calls.

In the example above it *is not* possible to replace the function name "factorial" with an arbitrary expression. However, the "apply" operator, (@) allows you to use an arbitrary expression as a function name. The following is an example of the "apply" operator.

```
"fact"#"orial"@5
```

When the apply operator is evaluated, the expression on the left is evaluated and treated as a function name. The expression on the right is evaluated and treated as the argument list of the function. The value of the expression is the value returned by the function.

5.7 Generating Partial FHDL Statements

The primary method for generating FHDL statements is the text statement. Each text statement causes a complete FHDL statement to be generated. At times you may wish to generate only a part of an FHDL statement. To illustrate, let us return to the example of Section 3. Recall that this example generated a NOT gate for each input/output pair. Let us modify this example to produce a subnetwork rather than generating the not gates "in line." Because a text statement has a fixed number of operands, a single text statement cannot be used to generate the

input and output lists. However the 'disp statement can be used to get around this problem, as the following example illustrates.

```
not.m:  'macro
        'if         'count('ilist)'!='count('olist)
        'error      s,"not.m inputs and outputs don't match"
        'exit
        'endif
        circuit
        'int        'i
        'disp       "\tinputs\t"#'ilist(1)
        'assign     'i,1
        'while      'i'<'count('ilist)
        'disp       ","#'ilist('i)
        'assign     'i,'i+1
        'endwhile
        'disp       "\n\tinputs\t"#'olist(1)
        'assign     'i,1
        'while      'i'<'count('olist)
        'disp       ","#'ilist('i)
        'assign     'i,'i+1
        'endwhile
        'disp       "\n"
        'assign     'i,0
        'while      'i'<'count('ilist)
        not         'ilist('i),'olist('i)
        'assign     'i,'i+1
        'endwhile
        endcircuit
        'endmacro
```

The 'disp statement is used to generate unformatted text. The statement has one operand, which is evaluated and placed in the output. Note that opcodes, separators and statement terminators must be explicitly specified. This example also illustrates the use of special characters in character strings. In general, anything that is acceptable in the C language is acceptable to the macro preprocessor, except the null character (\0). The 'disp statement can be used to create non-FHDL output.

5.8 The else-if Construct

There are times when one must create a multi-way conditional depending on several different conditions. One way to do this is to use nested 'if statements, that is, place a second 'if statement in the 'else part of the first statement. This relatively common construct can be expressed more compactly using the 'elif statement. For example, suppose you want to generate three different types of statement, depending on whether the variable 'x has the value 1, 2 or 3. The following sequence of statements will accomplish this.

```
'if      'x'==1
and      (a,b),c
'elif    'x'==2
or       (a,b),c
'elif    'x=3
xor      (a,b),c
'else
'error   s,"Invalid value of x"
'endif
```

Note that a single 'endif statement is used to terminate the 'if construct. The 'else portion of the statement will be executed if none of the preceding conditions are true. The body of an 'elif portion will be executed if the corresponding condition is true and all preceding conditions are false. The 'elif statement has a single operand which follows the same rules as the operand of the 'if statement.

5.9 Accessing Attributes

In addition to ordinary operands, many FHDL statements have attributes which are of the form <name>=<value> or <name>=(<value1>, ...). These types of operands are also acceptable to the macro processor. When an attribute is used on a macro call, both the name of the attribute and the value may be specified as expressions. In the macro definition, the name and the value of the attribute may be accessed separately using the 'aname and 'avalue built-in functions. Suppose the following macro call has been used.

```
stmt5:  newmac  (a,b,c),(d,e,f),type=large
```

Within the definition of "newmac" the expression 'aname('2) will return the value "type" and the expression 'avalue('2) will return the value "large". The expression 'aname('1) will return the null string, and the expression 'avalue('1) will return the list (d,e,f). In general if the operand of 'aname is not of the form <name>=<value> then 'aname will return the null string. If the operand of 'avalue is not of the form <name>=<value> then 'avalue will return the operand itself.

The 'aname and 'avalue functions may also be used to extract the components of expressions of the form <value>-><field-name>, which are used by the ROM and PLA preprocessors. For expressions of this form, 'aname returns the field name while 'avalue returns the value.

5.10 Arithmetic and Logical Expressions.

As mentioned above, arithmetic expressions may contain the operators +, -, *, and /. In addition unary plus and minus are allowed. Note that the division operator is integer division. The operator "%" can be used to obtain remainders. Thus 5%2 would give the value 1. Exponentiation can be performed by using the exponentiation operator, **.

The comparison operators '<, '>, '<=, '>=, '==, and '!= can be used in arithmetic expressions. These operators return 0 for a false condition and 1 for a true. The operators &&, || and ! may also be used in arithmetic expressions. The binary operator && gives the value 0 if either of its operands is zero, and the value 1 otherwise. The binary operator || gives the value 0 if both of its operands are zero, and the value 1 otherwise. The unary operator ! gives the value 1 if its operand is zero and the value 0 otherwise.

All numbers and integer variables are represented as 32-bit binary numbers. There are a number of bit-level operations that can be performed on numbers and integer variables. The operator "&" returns the bitwise AND of its two operands, while the operator "|" returns the bitwise OR. The unary operator "~" returns the bitwise complement (ones complement) of its operand. The binary operator "^" returns the bitwise EXCLUSIVE OR of its operands. The binary operators ">>" and "<<" perform right and left shifts respectively. The left-hand operator is shifted the number of bits specified by the right-hand operator. Examples are 'x>>2 and 'y<<3. Either operand may be an expression.

The normal operator precedence for arithmetic operators is given below in low to high order. Operators listed on the same line are of equal precedence. Normal precedence can be overridden with parenthesis.

```
||, |, ^
&&, &
!, ~
'<, '>, '==, '<=, '>=, '!=
<<, >>
+, -
*, /, %
**
unary +, unary -
```

The evaluation of expressions containing the operators +, -, *, /, &, |, and ! may sometimes give unexpected results. Because these operators are also used by other FHDL parsers, the macro preprocessor makes an effort not to evaluate these operators. Placing an apostrophe ahead of one of these operators will force its evaluation by the macro processor. To prevent the evaluation of an operator, enclose the entire expression in quotation marks.

5.11 String Handling Functions.

The macro processor provides several string handling functions. The concatenation operator (#) mentioned above can be used to concatenate strings. In addition, the 'substr function can be used to extract substrings of a given string. The 'substr function requires three operands. The first is the string from which the substring is to be extracted. The second operand is the starting position of the substring. The first character of the string is in position zero. The third operand is the length of the substring. If the length is specified as zero, the substring from the starting position to the end of the subject string is returned. Some examples of this function are given in the following table.

`'substr("abcdef",0,3)`	`returns "abc"`
`'substr("abcdef",3,1)`	`returns "d"`
`'substr("abcdef",2,3)`	`returns "cde"`
`'substr("abcdef",2,0)`	`returns "cdef"`

A negative starting position is treated as zero. A starting position greater than or equal to the length of the string causes the null string to be returned. A negative length is treated as a zero. If there are not enough characters in the string to create a substring of the specified length, the substring from the starting position to the end of the string is returned. If the 'substr function is specified with fewer than three arguments, the omitted arguments are treated as zeros. Any of the three arguments may be expressions.

The 'len function can be used to count the characters in a string. The 'len function requires one argument which should be a string. It returns the number of characters in a string. The following macro fragment can be used to process the characters of a string one at a time.

```
'int      'i
'str      'char
'assign   'i,0
'while    'i'<'len('0)
'assign   'char,'substr('0,'i,1)
/*... process 'char ...*/
'assign   'i,'i+1
'endwhile
```

The macro preprocessor provides several "read only" built-in string variables. The variable 'null can be used in place of "" to represent the null string. The variables 'callt and 'label have already been discussed. In addition, the variable 'opcode contains the opcode of the macro call. (As will be explained below, a macro can be given more than one name, so this variable does have some use.) The value of the variable 'run can be set from the UNIX command line. Its default value is "alogic".

5.12 List Handling Features

The macro preprocessor provides features for declaring and processing lists. A list variable is declared as in the following example.

```
'list   'listvar
```

The 'gbllist statement can be used to declare global list variables. The built-in list variables 'ilist, 'olist and 'args have already been discussed. A list constant is simply a list of elements separated by commas and enclosed in parentheses. Thus to assign a three-element list to the variable declared above, use the following statement.

```
'assign  'listvar,(a,bbb,qed)
```

The first element of a list can be accessed using the 'first function, so the function call 'first((a,b,c)) returns the string "a". The 'rest function can be used to remove the first element of the list and return the rest. Thus 'rest((a,b,c)) returns the list (a,b), and 'rest((a,b)) returns the string "b". If the list contains only one element, 'rest will return the null string. The 'select function can be used to select a particular element of a list. This function requires two arguments, a list and a number indicating which element to select. Elements are numbered starting with zero, so 'select((a,b,c),0) returns "a" while 'select((a,b,c),2) returns "c". If the second argument is greater or equal to the number of elements in the list, the null string is returned.

Lists may be nested, so the following is a valid example of a list constant.

(a,b,(c,(d,dd),e),f)

Because of list nesting, the functions 'first, 'rest and 'select may return either a list or a string. To determine whether the object returned by one of these functions is a string or a list, use the 'atom function. The function 'atom('first((a,b,c))) will return true, while 'atom('rest((a,b,c))) will return false. As for other conditionals, if these true and false values are used in arithmetic expressions they are treated as one and zero respectively. You can test for the null string (or null list, which is the same thing) by using the function 'nil. The function 'nil returns true if its argument is the null string (or null list) and false otherwise.

Lists can be constructed one element at a time by using the functions 'cons and 'consr. These functions take two arguments, the second of which must be a list, and the first of which must be an element to be added to the list. If the second argument is not a list, it is coerced to a single-element list. The first argument may be an integer, a string, or another list. The function 'cons adds the element to the beginning of the list, while the function 'consr adds the element to the end of the list.

When a list is used as a string it is coerced to a string. If it is a single-element list, the result is the value of the single element. If it is a multi-element list, the result is a string that begins with "(" and ends with ")" and contains the values of the elements separated by commas. Thus the list (a,b,c) is coerced to the string "(a,b,c)". When a list is used as an integer, it is first coerced to a string then the string is coerced to an integer. When an integer or a string is used as a list, it is coerced to a single-element list.

5.13 Type Conversion Functions

The macro preprocessor provides automatic type conversion between lists, strings, and integers. To summarize, numeric strings are converted to integers using a standard (character) decimal-to-binary conversion. Non-numeric strings are converted to zeros. Integers are converted to strings by using a standard binary-to-decimal conversion that produces no leading zeros. Conversions between lists and the other two types are detailed in Section 12.

Automatic type conversions are supplemented by several functions that do explicit type conversions. Integers can be converted to fixed-length strings using the 'itos function. The 'itos function requires two arguments, the first of which is the integer to be converted, and the second of which is the width of the number to be produced. If the converted string is shorter than the specified width, it is padded on the left with zeros. If it is longer than the specified width, characters on the *left* are truncated.

In addition to the decimal conversions, the functions 'xtoi, 'otoi, 'btoi, 'hex, 'octal, and 'binary can be used to do conversions using the bases 16, 8, and 2. The function 'xtoi converts a string containing digits and the letters a-f (or A-F) into an integer. The characters are assumed to represent base-16 digits. Similarly the functions 'otoi and 'btoi can be used to convert strings containing the characters 0-7 (for 'otoi) or 0-1 (for 'btoi) into integers. The characters of the string are assumed to be octal (for 'otoi) or binary (for 'btoi) digits. The functions 'hex, 'octal, and 'binary perform integer to string conversions. The function 'hex produces a base-16 number using the additional digits a-f, while 'octal and 'binary produce base-8 and base-2 numbers respectively.

Integer constants may be specified in decimal, octal, or hexadecimal. Decimal numbers other than zero begin with a non-zero digit and contain only the digits 0-9. Octal numbers begin with a zero and contain only the digits 0-7. Hexadecimal numbers begin with the two characters "0x" or "0X" and thereafter contain only the characters 0-9, a-f, and A-F. Note that the operand of

'xtoi *must not* contain the leading "0x" or "0X" characters. Furthermore, the operand of 'otoi need not start with a zero. Furthermore, the function 'hex *will not* prepend "0x" or "0X" to its output, nor will the output of 'octal necessarily start with a zero.

If the functions 'hex, 'octal, and 'binary are supplied with a single operand, the result string will begin with a non-zero digit, and will be just long enough to contain all significant digits of the converted value. If a second operand is supplied, it must be an integer specifying the number of digits to be produced. The result string will be padded with zeros or truncated *on the left* to produce a string of the specified length.

The function 'ctoi can be used to convert a character to an integer whose value is the binary value of the character in the underlying character representation. This is ascii for most systems, so 'ctoi(" ") usually returns the number 32. The operand of 'ctoi is a string. All characters but the first are ignored.

5.14 Redirecting Output

Generating logic all inline is the natural mode of operation for the macro processor. At times it may be desirable to generate a subnetwork and place references to it inline. This complicates the data generation problem, because the subnetwork and the reference must be generated at different places in the code, and will be separated by an unpredictable number of statements.

The output redirection facility of the macro processor was designed to solve this problem. The output of the macro processor is divided into three sections, the standard output or inline section, the network section, and the subnetwork section. For the examples given above, only the inline section is used. When output is generated the inline section appears first, the network section second, and the subnetwork section third. Output may be added to these sections in arbitrary order. For example consider the following macro definition of a full adder.

```
fulladd:  'macro
          'inputs      'a,'b,'c
          'outputs     'sum,'carry
          'gblint      'fa_done
          'if          'fa_done'==0
          'subnetwk
fa:       circuit
          inputs       a,b,c
          outputs      sum,carry
          xor          (a,b),i1
          xor          (i1,c),sum
          and          (a,b),i2
          and          (a,c),i3
          and          (b,c),i4
          or           (i1,i2,i3),carry
          endcircuit
          'assign      'fa_done,1
          'endif
          'stdout
'label:   fa           ('a,'b,'c),('sum,'carry)
          'endmacro
```

The statement 'subnetwk changes the current output section to be the subnetwork section. The statement 'stdout changes the current output section to the inline section. The 'network statement changes the current output section to the network section. In this example, the definition of the circuit "fa" will appear after the all calls to "fa".

The output redirection feature also allows several circuits to be constructed simultaneously in piecemeal fashion. This is done by creating named subnetwork and network sections. The following example demonstrates the use of named subnetwork sections.

```
'exam:  'macro
        'subnetwk   a
a:      circuit
        'subnetwk   b
b:      circuit
        'subnetwk   a
        input       aa
        output      aaa
        not         aa,aaa
        endcircuit
        'subnetwk   b
        input       bb
        output      bbb
        not         bb,bbb
        endcircuit
        'stdout
        'endmacro
```

In this example, two named subnetwork sections, a, and b are defined. Named subnetwork sections appear in the order defined in the subnetwork section of the output. If the unnamed subnetwork section is used in conjunction with named subnetwork sections, it is treated as a named subnetwork section whose name is the null string. The first 'subnetwk statement encountered that contains a particular name defines a named subnetwork section. All other subnetwork sections containing the same name add to the named subnetwork section. The name may be a complex expression. Named network sections are handled similarly.

5.15 Creating Macro Libraries

Macros may appear inline with the text of a circuit, or they may be placed in macro libraries, and accessed automatically. A macro library is a directory whose files contain macro definitions. Each macro must be a separate file, and the file name must exactly match the name of the macro. A macro may be given more than one name by using the UNIX "ln" command to link macro names.

A macro library is often used to define a coordinated set of macros for a particular application. When this is done, it is sometimes necessary to have special initialization and termination macros. One way to use such macros is to force all users to place the initialization and termination macros at appropriate points in the text of their circuits. Another way is to explicitly define initialization and termination macros in the library. The initialization macro must be named "$START" and the termination macro must be named "$END". If an initialization macro is defined, it will be invoked before the first statement of the user's circuit is

interpreted. If a termination macro is defined it will be invoked after the last statement of the user's circuit is interpreted.

Several macro libraries may be used in a single run. It is permissible for the same macro-name to appear in several libraries. In this case the form of the UNIX command that invokes the preprocessor determines which library to use for a particular macro. See section 18 for details.

5.16 Including Text

It may be necessary to define a large number of global variables, that are shared by a number of macro definitions. It may also be necessary to replicate other text in each macro. Explicit replication of text in each macro makes that body of text next to impossible to change. The *include* feature of the macro processor can be used to circumvent this problem. The following example illustrates the use of this feature.

```
exam: 'macro
      'include   global_defs
      ...
      'endmacro
```

This include statement causes the file "global_defs" from the macro library to be included in the text of the macro "exam". The file name *may not* be an expression. It must be a string either with or without quotes. If the name is not found in any macro library, it will be treated as an ordinary file name. If the file can be found using the ordinary rules for locating files, (i.e. full path name, relative to current directory) that file will be included in the text, otherwise a "nonexistent file" message will be issued.

The 'include statement may be nested arbitrarily, but beware of circular references. Circular references *always* cause a non-terminating expansion of the text. This is because the 'include statement is executed as the text is *read* from the source files, not when the code is *interpreted*. Every 'include statement is executed *unconditionally* regardless of where in appears in the text. Thus the following example will cause *two* include statements to be executed.

```
exam:  'macro
       'if       'run'=='abc'
       'include  abc
       'else
       'include  xyz
       'endif
       'endmacro
```

For convenience, all include files may be gathered into an include directory which will be treated as a macro library by the preprocessor. It is important that the names of include files and macros be distinct. See section 18 for more information.

5.17 A Word on Format

.All portions of a statement may be specified as expressions. The examples given in preceding sections show, for the most part, constants in the label field and opcode field. In fact, both of these fields may be specified as complex expressions. Preprocessor statements, however, can be specified *only* with constant opcodes. An expression *cannot* be used to generate a preprocessor keyword.

Statements may be assigned more than one label, as illustrated in the following examples.

```
a:
b:
c:   'macro
```

```
a;b:;c:  'macro
```

When more than one label is used on a macro call, the 'label variable contains the value of the *first* label. The built-in list variable 'llist can be used to access the entire list of variables attached to the macro call. Also, the built-in function of the same name can be used to access various portions of the label list.

Macros may be assigned more than one name by placing several labels on the 'macro statement, or by placing additional names in the operand field of the 'macro statement. (If only one name is used, it may appear either in the label field or in the operand field.) When a macro is given more than one name, any of the assigned names may be used as an opcode to invoke the macro. The actual name used can be determined in the macro definition by using the 'opcode variable. If "function call" macros are given more than one name, the 'opcode variable *must* be used to return the function value. This is because the name of the "return" variable always matches the name used to invoke the function. If one of the macro's other names is used, an "undefined variable" message will be issued. The precise method for doing this is explained below.

The six statements used to define variables may have label fields. If labels are used on these statements, the labels are taken to be the names of additional variables to be defined. Thus the following statement defines six integer variables.

```
a:
b:
c:  'int  d,e,f
```

When a label of a 'macro statement or one of the six variable-defining statements contains commas, the label is treated as a list of names. Thus the following statement assigns three names to a macro.

```
x,y,z:   'macro
```

And the following statement defines three integer variables.

```
a,b:  'int   c
```

The previous examples also illustrate another syntactic principle of the macro processor. The expression 'x means "the value of x". The expression x means "the variable x". Although it is less confusing to use the form 'x everywhere, it is not syntactically correct to do so. HOWEVER, the macro processor permits the syntactically incorrect form to be used in variable definitions and in assignments. The syntactically correct way to increment the value of the variable 'i is as follows.

```
    'assign   i,'i+1
```

However, the preprocessor accepts the following statement as incrementing the value of 'i.

```
    'assign   'i,'i+1
```

The syntactically correct meaning of this statement is "obtain the value of i and use the value as the name of the variable to be assigned." Since indirect assignment tends to be quite rare, this is taken to be a miswritten direct assignment. It is actually good practice to use the syntactically *incorrect* forms, since this will tend to lessen the frequency with which you forget to use the leading ' mark. When it is necessary to do an indirect assignment this can be done by enclosing the name of the variable in parentheses as the following statement illustrates.

```
'assign   ('indvar),'i+1
```

Anything other than a *user defined variable* as the first operand of an 'assign statement will cause the syntactically correct rules to be applied. Thus, returning a value from a function-call macro with more than one name can be done using a statement similar to the following.

```
'assign   'opcode,"return value"
```

In the examples given above, the error message on the 'error statement was specified as a single string. In fact an arbitrary expression may be used, as illustrated below.

```
'error   s,"input count="#'count('ilist)#" should be 3"
```

The severity code may also be supplied as a complex expression. Only the first character of the operand is significant. Therefore a severity code of "severe" or "superfluous" is the same as "s".

Comments may be included in macros. There are two types of comment statements. If the opcode "comment" is used (without the leading ' mark) the comments will be included in the generated text. If the opcode 'comment is used, the comments *will not* appear in the generated text. The first type of comments are used to include informational messages in the generated text. The second type are used to comment the macro itself. In either case the body of the comment may contain any printable character. A semicolon or a newline character marks the end of a comment.

Within a macro the order of variable definition statements is irrelevant. The first use of a variable may precede its definition. Similarly the first use of a macro may precede the definition of the macro. It is necessary, however, for all macro definitions to precede all non-macro text in an input file. The non-macro text in an input file is treated as an unnamed macro, so preprocessor statements may appear in the non-macro text. In particular 'include statements may appear in non-macro text.

5.18 Executing the Preprocessor

The preprocessor is executed using the "alogic" command. The simplest form of this command is to use it as a filter for FHDL input, as the following illustrates. (The alogic command is executed automatically if the fhdl command with the "-n" option is used.)

```
alogic input.ckt | fhdl >input.c
```

A macro library can also be used with this form as the following command illustrates.

```
alogic -m maclib input.ckt | fhdl >input.c
```

The name of the macro library is "maclib". File names for both macro libraries and input files follow the usual UNIX rules for file name formation. The flag "-m" must precede the name of *each* macro library. The "-m" flag must be separated from the macro library name by one or more spaces or tabs. If more than one macro library is specified, and the same macro is defined

in more than one library, the definition encountered *last* takes precedence. This is also true for the initialization and termination macros. Only the last encountered initialization and termination macros will be executed. The following is a command that specifies two macro libraries "mac1" and "mac2" and an include directory "include1".

```
alogic -m mac1 -m mac2 -m include1 input.ckt | fhdl >input.c
```

In most cases, complex applications such as this will have shell files that supply the library names. File names without the preceding "-m" flag are input files. There may be several input files specified as in the following example.

```
alogic -m mac1 input1.ckt input2.ckt input3.ckt | fhdl >input.c
```

Input files are processed one at a time in the order specified. After the processing of a file is complete, the macros defined in the file are retained and made available to all succeeding files. This may cause problems if several files define the same macro (in this case, break into several runs). The main use of this feature is to allow all non-library macros to be placed in a header file that precedes the circuit file on the command line.

A file name of "-" (a single dash preceded and followed by one or more spaces or tabs) indicates the standard input. The standard input is read (possibly several times) whenever the file "-" is processed.

The value of the variable 'run may be set using the "-r" flag on the command line. The "-r" flag must be preceded and followed by one or more spaces or tabs. The (command line) argument following the "-r" flag is assigned to the 'run variable. There must be only one "-r" flag on the command line. All input files in a particular run share the same value of the 'run variable. The following is an example of setting the 'run variable.

```
alogic -m mlib -r old input.ckt | fhdl >input.c
```

Normally the output of the macro preprocessor is directed to "stdout". (Error messages are directed to "stderr".) This output can be redirected in the usual fashion, or it can be explicitly directed to an output file by using the "-o" flag on the command line. The "-o" flag must be preceded and followed by one or more spaces or tabs. The output of the preprocessor will be placed in the file name following the "-o" flag. There may be at most one "-o" flag on the command line. If several output files are needed, break into several runs.

5.19 Macro Statement Summary

5.19.1 Operand-Type Designators

Designator	Meaning
-	none allowed
A	Arbitrary
E	Expression
F	File Name
L	i,w,s, or t
V	Variable Name
VL	A list of variable names separated by commas
X	an expression or omitted

5.19.2 Statements

Opcode	Operands	Function
'assign	V,E	Evaluate E and assign to V
'break	-	Terminate innermost loop
comment	A	Pass comment through to output
'comment	A	Comment disappears from output
'continue	-	Terminate the current iteration of the innermost loop
'disp	E	Output a portion of an FHDL statement
'elif	E	Introduce an alternative condition into an IF statement
'else	-	Introduce the ELSE portion of an IF statement
'endfor	-	End of a 'for loop
'endif	-	End of an IF statement
'endmacro	-	End of a macro definition
'endwhile	-	End of a 'while loop
'error	L,E	Print an error message E of severity L
'eval	E	Evaluate the expression E
'exit	-	Terminate evaluation of the current macro
<expression>	-	Evaluate the expression
'for	E,E,E	Begin a for loop. Operands are init, cond, incr
'gblint	VL	Define global integer variables
'gbllist	VL	Define global list variables
'gblstr	VL	Define global string variables
'if	E	Begin an if statement with condition E
'include	F	Include the file named F in the current macro or input file
'input	VL	Define input-signal operand names
'int	VL	Define local integer variables
'list	VL	Define local list variables
'macro	VL	Begin a macro definition

Opcode	**Operands**	**Function**
'network	X	Begin or resume a (possibly named) network section
'output	VL	Define output-signal operand names
'stdout	X	Begin or resume a (possibly named) standard-output section
'str	VL	Define local string variables
'subnetwk	X	Begin or resume a (possibly named) subnetwork section
'while	E	Begin a while loop with continuation condition E

5.20 Macro Function and Builtin Variable Summary

5.20.1 Operand-Type Designators

Designator	Meaning
-	none, this is a variable
A	Macro Argument, or a part thereof
N	Numeric Expression
L	List Expression
S	String Expression
C	Single-Character String Expression
E	Arbitrary Expression

5.20.2 Functions and Variables

Opcode	Operands	Function
'aname	A	Returns the name for expressions of the form <name>=<value> or <value>-><name>.
'args	-	Returns a list containing all macro arguments for the current call.
'args	N	Returns the Nth argument of the current macro call.
'atom	E	Returns 0 if E is a list, 1 otherwise.
'avalue	A	Returns the value for expressions of the form <name>=<value> or <value>-><name>.
'binary	N	Converts N to a binary string and returns the result.
'binary	N,N	Converts arg1 to a fixed-length binary string (arg2) and returns the result.
'btoi	S	Treats S as a string of binary digits, and converts it to an integer.
'calln	-	The sequential number of the macro call that invoked the current macro.
'callt	-	Type of current macro call. True for functions, false for statements. Strings "STMT" or "FUNC" if evaluated as a string.
'cons	E,L	Append E to the head of the list L, and return the result.
'consr	E,L	Append E to the tail of the list L, and return the result.
'count	L	Count the elements of list L and return the result.
'ctoi	C	Return the integer value in the underlying character set of the character C.
'first	L	Return the first element of list L.
'hex	N	Convert the number N to a string of hexadecimal digits.
'hex	N,N	Convert the first argument to a fixed-length string of hexadecimal digits, length equal to second argument.
'ilist	-	For FHDL gate-type macro calls, return the list of input names.
'ilist	N	For FHDL gate-type macro calls, return the Nth input name.
'itos	N,N	Convert the first argument to a string of decimal digits whose

Opcode	Operands	Function
		length is equal to the second argument. (Automatic coercion takes care of variable-length strings.).
'label	-	The label of the current macro call.
'len	S	Returns the length of the string argument.
'llist	-	Returns the list of labels for the current macro call.
'llist	N	Returns the Nth label of the current macro call.
'nil	L	Returns 1 if the argument is the null list, 0 otherwise.
'null	-	The null string or the null list.
'octal	N	Converts N to a string of octal digits.
'octal	N,N	Converts the first argument to a fixed-length string of octal digits. The length is equal to the second argument.
'olist	-	For FHDL gate-type macro calls, return the list of output names.
'olist	N	For FHDL gate-type macro calls, return the Nth output name
'opcode	-	The opcode of the current macro call.
'otoi	S	Treats S as a string of octal digits and returns the integer value.
'rest	L	Deletes the first element from a list and returns the rest. For single element lists, returns 'null.
'run	-	The value of the -r option from the alogic command line. "alogic" is default.
'select	L,N	Select the Nth element from list L.
'substr	S,N,N	Return the substring of S that starts at the character denoted by the first argument (0 is the first char of S) and spans a number of characters equal to the second argument.
'substr	S,N	Return the substring of S that starts at the character denoted by the first argument (0 is the first char of S) and extends through the end of the string.
'xtoi	S	Treat S as a string of hexadecimal digits, and return the integer equivalent.

5.21 Macro Operator Summary

5.21.1 Operand-Type Designators

Designator	Meaning
N	Numeric Expression
L	List Expression
S	String Expression
E	Arbitrary Expression
V	A Variable Name

5.21.2 Operands

Opcode	Operands	Function
@	S@L	Evaluate the first argument as a string, and the second as a list. Treat the string obtained from the first argument as a function name and apply it to the second argument treated as an argument list. Does not work with builtin functions.
,	E,E	Creates a list containing both expressions. May be used to create multi-element lists.
<-	V<-E	Evaluates E and assigns the value to V.
<-	S<-E	Evaluates E and S, and assigns the value of E to the variable whose name matches the value of S.
=	E=E	Outputs as "E=E". If used in a macro argument, may be the subject of 'aname and 'avalue functions.
->	E->E	Outputs as "E->E". If used in a macro argument, may be the subject of 'aname and 'avalue functions.
#	S#S	Concatenates the two strings.
&&	N&&N	Logical AND of the arguments treating zero as false non-zero as true.
&	N&N	Bitwise AND of the two (32-bit integer) arguments.
&	E&E	Outputs as "E&E" if one argument is not numeric.
'&	E'&E	Bitwise AND of the two (32-bit integer) arguments, regardless of type.
\|\| | N\|\|N | Logical OR of the arguments treating zero as false non-zero as true. |
\| | N\|N | Bitwise OR of the two (32-bit integer) arguments. |
\| | E&E | Outputs as "E\|E" if one argument is not numeric. |
'\| | E'\|E | Bitwise OR of the two (32-bit integer) arguments, regardless of type. |
^ | E^\|E | Bitwise EXCLUSIVE OR of the two (32-bit integer) arguments, regardless of type. |
! | !N | Logical NOT of the argument, treating zero as false non-zero as true. |

Opcode	Operands	Function
~	~N	Bitwise NOT of the (32-bit integer) arguments. Ones Complement.
<	E<E	Outputs as "E<E"
>	E>E	Outputs as "E>E"
==	E==E	Outputs as "E==E"
<=	E<=E	Outputs as "E<=E"
>=	E>=E	Outputs as "E>=E"
!=	E!=E	Outputs as "E!=E"
'<	E'<E	Less than comparison. Returns 1 for true, 0 for false.
'>	E'>E	Greater than comparison. Returns 1 for true, 0 for false.
'==	E'==E	Equal to comparison. Returns 1 for true, 0 for false.
'<=	E'<=E	Less than or equal to comparison. Returns 1 for true, 0 for false.
'>=	E'>=E	Greater than or equal to comparison. Returns 1 for true, 0 for false.
'!=	E'!=E	Not equal to comparison. Returns 1 for true, 0 for false.
$LT	E$LTE	Same as '<.
$GT	E$GTE	Same as '>.
$EQ	E$EQE	Same as '==.
$LE	E$LEE	Same as '<=.
$GE	E$GEE	Same as '>=.
$NE	E$NEE	Same as '!=.
<<	N<<N	Shifts the first (32-bit integer) argument left the number of bits indicated by the second argument.
>>	N>>N	Shifts the first (32-bit integer) argument right the number of bits indicated by the second argument.
+	N+N	Returns the sum of its arguments.
+	E+E	Outputs as "E+E" if one argument is not numeric.
'+	N'+N	Returns the sum of its arguments, regardless of type.
-	N-N	Returns the difference of its arguments.
-	E-E	Outputs as "E-E" if one argument is not numeric.
'-	N'-N	Returns the difference of its arguments, regardless of type.
*	N*N	Returns the product of its arguments.
*	E*E	Outputs as "E*E" if one argument is not numeric.
'*	N'*N	Returns the product of its arguments, regardless of type.
/	N/N	Returns the quotient of its arguments.
/	E/E	Outputs as "E/E" if one argument is not numeric.
'/	N'/N	Returns the quotient of its arguments, regardless of type.
**	N**N	Raises the first argument to the power designated by the second argument. This is done iteratively.
unary +	+N	Returns N.
unary +	+E	Outputs as "+E" if the argument is not numeric.
unary '+	'+E	Returns E regardless of type.
unary -	-N	Returns the negation of N. Two's Complement.

Opcode	Operands	Function
unary -	-E	Outputs as "-E" if the argument is not numeric.
unary '-	'-E	Returns the negation of E regardless of type.
'	'E	Evaluates its argument, and treats the result as a variable name, and then returns the value of the named variable. If the expression is numeric and evaluates to n, the value of the nth argument of the current macro call is returned.

5.22 Macro Processor Keywords

'args
'assign
'atom
'avalue
'binary
'break
'btoi
'calln
'callt
comment
'comment
'cons
'consr
'continue
'count
'ctoi
'disp
'elif
'else
'endfor
'endif
'endmacro
'endwhile
'error
'eval
'exit
'first
'for
'gblint
'gbllist
'gblstr
'hex
'if
'ilist
'include
'input
'int
'itos
'label
'len
'list
'llist
'macro
'network

'nil
'null
'octal
'olist
'opcode
'options (reserved for future use)
'otoi
'output
'rest
'run
'select
'stdout
'str
'subnetwk
'substr
'while
'xtoi

5.22 Macro Operator Precedence

All preprocessor operators are listed from lowest priority to highest. If more than one operator is listed on a line, all operators on the same line have equal priority. The first column indicates the associativity of the operator.

Precedence	**Associativity**	**Operator**
Low	Right	@
	Right	, (Yes, comma is treated as an operator)
	Right	<-
	Left	= ->
	Left	#
	Left	|| | ^ '|
	Left	&& & '&
	Right	! ~
	Non assoc.	< > == <= >= !=
	Non assoc.	'< '> '== '<= '>= '!= $LT $GT $EQ $LE $GE $NE
	Non assoc.	<< >>
	Left	+ - '+ '-
	Left	* / % '* '/
	Right	**
	Right	unary + unary - unary '+ unary '-
High	Right	' (indirection)

CHAPTER 6

The Test Driver Language

The FHDL Test Driver Language is designed to simplify the process of creating and running functional tests. It is capable of running tests, suppressing all but the "interesting" results, checking for errors, generating the same vector repeatedly until a condition becomes true, and generating error messages. Driver-language statements interact dynamically with the simulation to produce results that cannot be achieved in a static vector environment.

6.1 Introduction

The FHDL Test-Driver Language is intended to simplify the testing of circuits specified using the Functional Hardware Design Language. The features described in this chapter are automatically provided when the fhdl command with the "-n" option is used. These features are not available otherwise. The Driver language can be used to specify the format of tests, and the manner in which tests are applied to the circuit. Feedback from the circuit under test can be used to control the testing process. Figure 1 illustrates the relationship between the driver created using the FHDL driver language, and the circuit under test.

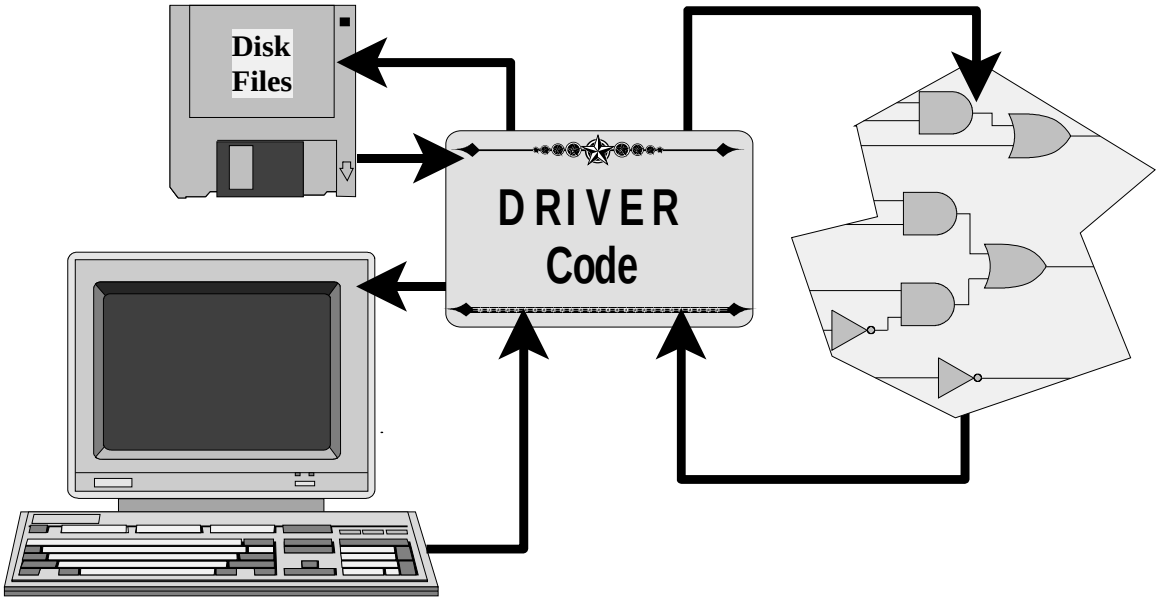


Figure 1. The Relationship of the Driver and the CUT.

The Driver can be used to save time in generating test vectors, and can be used to compare expected results with the real results and report discrepancies.

6.2 The Format of the Language

The format of driver language statements is given below. The format is similar to that of the other FHDL languages.

```
<label>:   <opcode>   <operands>
```

Each statement contains a label field, and op-code field and an operands field. The label field begins at the first character of a statement and ends with a colon. A label may contain any combination of letters, digits, underlines and periods. The label field is separated from the op-code field by one or more spaces or tabs. If the label is omitted, the colon must also be omitted and the statement must begin with a space or a tab to signify that the label is omitted. Labels are ignored on most driver language statements. The opcode field must be separated from the operands field by one or more spaces or tabs. The operands field is optional for some statements, but the op-code is required for all statements. A statement is terminated by a carriage return or a semicolon. If a line ends in a comma, the statement is assumed to be continued on the next line.

The driver provides local variables and allows the nets of the Circuit Under Test (CUT) to be accessed directly. Variable names and net names consist of an arbitrary string of letters, digits, underlines and periods. Case *is* significant for variable names and net names so Abc, ABc, and abC are three different variable or net names. Case *is not* significant for driver-language keywords, so the keywords Driver, DRIVER, and DrIvEr are all the same.

A set of driver specifications must begin with a "driver" statement and end with an "enddriver" statement. These statements must be included with the rest of your FHDL specifications. At the present time, only one set of driver statements is allowed in any FHDL specification. Figure 2 illustrates the form of the driver specifications.

```
main: driver
      <driver statements>
      enddriver
```

6.3 Expressions

Most driver statement operands may be expressions. Expressions are formed just as they are in most programming languages. The simplest expressions are variable names, net names, and numbers. Numbers may be specified in decimal, octal, or hexadecimal. Decimal numbers begin with a non-zero digit and contain only the digits 0-9. Octal numbers begin with a zero and contain only the digits 0-7. Hexadecimal numbers begin with the characters 0x or 0X and contain only the digits 0-9 and a-f (A-F may also be used).

The operators +, -, *, and / may be used to perform addition, subtraction, multiplication and division. Unary + and - may also be used. (Warning though, all driver variables and all nets are treated as unsigned integers.)

The operator "->" may be used to assign a value to a variable or to a net. The form of this expression is <value>-><variable> or <value>-><net>. The assignment expression has a value equal to the value assigned to the variable or net.

The operators ==, !=, <, >, <=, and >= may be used to do equal, not equal, less than, greater than, less than or equal to, and greater than or equal to comparisons. Thus to compare a variable xyz equal to 5, use the expression xyz==5. A comparison operator will produce the value 1 if the comparison is true and 0 if the comparison is false. This value may be used, in an arbitrary way, in other expressions. The operator "==" may be abbreviated as "=".

The operators &, |, and ! (the AND, OR, and NOT functions) may be used to create complex conditions. These operators treat any non-zero value as true and zero as false. They produce 1 for a true result and 0 for a false result. This result may be used arbitrarily in other expressions.

A set of expressions may be evaluated by separating them with commas as in the following.

```
0x64->a,2->b,3->c
```

If an expression of this form is used in another expression, its value is the value of the last expression.

Operator priority is given from low to high in the following table. The table also indicates whether a particular operator is left-associative, right-associative, or non-associative. Operators of equal precedence are listed on the same line.

Prededence	Associativity	Operator
Low	right	, (comma)
	left	->
	right	!
	left	|
	left	&
	non assoc.	== != < > <= >=
	right	Unary + Unary -
	left	+ -
High	left	* /

6.4 Statements

6.4.1 The variable statement

The "variable" statement is used to declare variables. The following is an example.

```
variable        a,b,abc,z,xyz
```

Variable statements may appear anywhere in the set of specifications. There is no limit to the number of variables that may be declared. Variable names are arbitrary strings of letters, digits, underlines and periods. Case *is* significant. Any variable name that is used but not declared is assumed to be a net of the circuit under test. Thus undeclared variables will normally be reported as non-existent nets. If a variable has the same name as a net in the circuit under test, the variable name overrides the net name. There is no limit on the length of variable names. All variables are treated as unsigned 32-bit integers.

6.4.2 The go statement

The "go" statement causes the functional simulator to be executed. One set of inputs is supplied and one set of outputs is obtained. For clocked circuits, the "go" statement causes the circuit to be executed for one phase of the clock. The format of the "go" statement is given below.

```
go
```

To execute the functional simulator for more than one phase, or to execute repeatedly with the same set of inputs, a numeric operand may be placed on the go statement as illustrated below. This operand may be an expression.

```
go   30
go   abc+2
```

The "go" statement may cause the values of certain variables and nets to be updated automatically. See the "on," "clock," and "count" statements.

6.4.3 The expression statement

The expression statement is used to assign values to variables and nets. An arbitrary expression is used as the op-code of the statement. In order to be meaningful, the expression should contain at least one assignment operator. The following illustrates the use of assignments to create a test vector for a circuit.

```
inputa->1,inputb->0,clk1->0
go
clk2->1
go
```

The names "inputa" and "inputb" are assumed to be primary inputs of the circuit under test. Note that the values of these inputs is identical for both "go" statements. The SET statement is a variation of the expression statement. This statement is illustrated below.

```
set  inputa->1,inputb->0,clk1->0
go
set  clk2->1
go
```

6.4.4 The read statements

The read statement is used to read a vector of values from an external file and assign the values to a set of variables or nets. The first operand of a read statement must be a number (or an expression) that specifies the file to be read. The following is an example of a read statement.

```
read 0,inputa,inputb,xyz
```

Each vector consists of a number of values separated by commas and terminated by an end of line character. If the vector contains more values than there are variables (or nets) on the read statement, the extra values are discarded. If there are fewer values than variable names (or net names) the extra variables (or nets) will be assigned the value zero. The values must be specified in hexadecimal without the leading 0x or 0X.

The file number must be (or evaluate to) a number between zero and ten inclusive. A file number of zero causes the standard input to be read. (Recall that a set of FHDL specifications compiles into a UNIX program. The zero file refers to the standard input of that program, which defaults to the terminal.) File numbers between one and ten refer to the arguments of the command used to invoke the simulator. A file number of 1 causes the first argument of the command to be treated as a file name, and the program reads the input vector from that file.

If specifying input values in hexadecimal is inconvenient, the "readd" statement can be used to read decimal, octal, or hexadecimal data. The format of the "readd" statement is identical to that of the "read" statement. When the "readd" statement is used, input values are assumed to be in decimal if they begin with a non-zero digit, octal if they begin with zero, and hexadecimal if they begin with the characters 0x or 0X.

The "get" command can be used to abbreviate reads from the standard input. The following two commands are equivalent.

```
read  0,inputa,inputb,xyz
get   inputa,inputb,xyz
```

The command "getd" can be used to read decimal, octal, and hexadecimal data from the standard input. Its format is identical to that of the "get" statement.

The expression "eof(<file number>)" can be used to test a file for end-of-file. The expression eof(0) returns 1 if file zero (stdin) is at end of file, and zero otherwise.

6.4.5 The write statements

The write statement is used to display the current values of variables and nets. For example, the following statement causes the current value of the net "outputa" and the variable "xyz" to be displayed on the standard output.

```
write  0,xyz,outputa
```

The first operand of a write statement is a file number. Just as in a read statement, the file number must be or evaluate to a number from zero through 10 inclusive. The number zero is used to specify the standard output of the UNIX command used to invoke the simulator. The numbers from 1 through 10 refer to the operands of the UNIX command used to invoke the

simulator. If the number 5, say, is used as a file number, the fifth operand of the command will be treated as a file name. The file will be opened for output and the data from the write command will be written on the file.

Each write statement produces one line of output. The values of the specified nets and variables are written in hexadecimal with no leading or trailing spaces, and separated by commas. All hexadecimal values for a particular net or variable will have the same number of digits, with leading zeros if necessary. Variations of the write command can be used to specify data in different formats.

The writed statement can be used to write values in decimal rather than hexadecimal. The format of the writed statement is identical to that of the write statement. When data is written in decimal, leading zeros are omitted. In all other respects, the output format is identical to that of the write statement.

The writex statement can be used to include the variable or net name in the output. When writex is used, the variable or net name is prepended to the the value, and separated from the value by an equal sign. The writex statement outputs values in hexadecimal, with leading zeros to maintain a constant width.

The writexd statement can be used to output named data in decimal. The decimal output will not contain leading zeros.

The display statement can be used as shorthand for writing named hexadecimal data on the standard output. The following two statements are identical.

```
write     0,xyz,outputa
display   xyz,outputa
```

The displayd statement can be used to display named decimal data on the standard output. It is equivalent to using a writexd statement with file zero.

6.4.6 The monitor statements

The monitor statement can be used to cause a write statement to be executed following every "go" statement. It is assumed that the output caused by a monitor statement will be formatted by a post processing program of the user's choice. The monitor statement is executable, so monitoring can be turned on or off at various points in the simulation. An example of a monitor statement is given below.

```
monitor   0,xyz,outputa
```

When monitoring is active, all monitor statements for a particular file number will be gathered into a single "write" statement that is executed after each go statement. The file to which monitoring is directed will contain one line of output per go statement, regardless of how many monitor statements have been used to specify the variables or nets to be monitored. Furthermore, every time a monitor statement is executed, a list of variable names is written to the specified file. This list of names identifies all variables that are currently being monitored on the file, and the order in which their values appear. The variable names have two leading characters prepended to specify the format of the monitored value. The first character is an x or a blank to specify whether the monitored value is printed with or without a name respectively. The second character is a d or a blank to specify whether the value is printed in decimal or hexadecimal respectively. The name will be followed by a comma which will in turn be followed by an integer that specifies the width of the variable being monitored. This number may vary for signals, it will be 32 for all variables defined in the driver description. A comma separates the

following variable name from the width of the previous variable. These lines allow a post processing program to determine the names of the variables or nets being monitored, their formats, and their positions in the output. The lines containing variable names will begin with the characters "T," (the capital letter T followed by a comma) to distinguish them from lines containing data. Similarly, the lines containing data will begin with the characters "D,", the capital letter D followed by a comma. Any messages that are written to a monitor file will begin with an asterisk ("*") to distinguish them from data lines.

The monitord statement can be used to cause a "writed" statement to be executed after every go instead of a "write" statement. Similarly, the monitorx and monitorxd statements can be used to cause writex and writexd statements to be executed. Only one line of output will be produced per go statement per file, regardless of whether mixed types of monitor statements have been used to initiate output. Execution of monitord, monitorx, and monitorxd statements also causes the line containing the variable names to be output.

The demonitor statement can be used to turn off the monitoring of a variable or a net. The following statement causes monitoring to be turned off for the net outputb, and the variable xyz.

```
demonitor   xyz,outputb
```

The demonitor statement will cause lists of variable names to be written to any affected file, making the new monitoring status available to a post-processing program.

If a monitor monitorx, monitord, or monitorxd statement is executed for a variable or net that is already being monitored, the variable or net is automatically demonitored before the monitor statement is executed. A variable or net can be monitored to at most one file at a time.

6.4.7 The if statement

The if statement can be used to conditionally execute statements. The following is an example of an if construct.

```
if      a==b
        display   a,b
        c->d
        monitor   d
endif
```

In this example, the statements between the if and endif statements will be executed only if a is equal to b. The operand of an if statement may be an arbitrary expression. If the expression evaluates to non-zero, it is considered to be a true result, and the statements between the if and endif statements will be executed. If the expression evaluates to zero, it is considered to be a false result and the statements will be skipped. Every if statement must have a corresponding endif statement. If statements may be nested arbitrarily.

The else statement can be used to cause statements to be executed when a condition is false. The following example illustrates.

```
if    a==b
      display   a,b
      c->d
      monitor   d
else
      display   c,d
      x->y
```

```
     monitor   z
endif
```

In this example a and b will be displayed if a and b are equal, and c and d will be displayed otherwise.

The elif statement can be used to construct multi-way conditionals. It can be used to simplify the construction of else-if conditionals. The following two constructs will produce the same results.

```
if   a==b
     display   a,b
else
     if   a==c
          display   a,c
     else
          display   x,y
     endif
endif
```

```
if   a==b
     display   a,b
elif a==c
     display   a,c
else
     display   x,y
endif
```

Any number of elif statements may be included in a single if construct. The final else statement is optional. A single endif statement terminates the entire construct.

6.4.8 The while statement

The while statement can be used to execute statements repeatedly. The following is an example.

```
while  done==0
       0->clk
       go
       1->clk
       go
endwhile
```

In this example, a feedback symbol "done" is used to control the application of vectors to the simulator. Assume that clk is a clock input to the circuit under test. This set of statements would be suitable for driving a microprogrammed arithmetic circuit such as a divider or a multiplier.

Every while statement must have a corresponding endwhile. The statements between the while and endwhile are executed until the specified condition becomes false (equal to zero). It is possible for the body of the loop to be executed zero times. While statements may be nested arbitrarily both with other while statements and with if statements.

6.4.9 The for statement

The for statement is used to execute statements repeatedly, and at the same time specify the values of loop variables. The following are two examples.

```
for  1->a,a<10,a+1->a
     go
endfor

for  (1->a,b->0),a<10,(a+1->a,b+5->b)
     go
endfor
```

The first example illustrates the use of a single loop variable, while the second illustrates the use of two loop variables. Both of these "loop variables" are assumed to be input nets of the circuit under test. The format of a "for" construct is given below.

```
for  <init>,<test>,<incr>
     <statement list>
endfor
```

The expressions <init>, <test>, and <incr> are three arbitrary expressions separated by commas. If any of these expressions contain commas, they must be enclosed in parenthesis. The expression <incr> is executed once before the loop begins. The <test> is executed prior to each iteration of the loop and if it is false (equal to zero), the loop terminates. The <incr> is executed prior to every iteration of the loop except the first, but only if the <test> expression is true (non zero). It is possible for the statement list to be executed zero times. For statements may be nested arbitrarily with other for statements, while statements, and if statements.

6.4.10 Break and continue statements

Early termination of a loop may be accomplished using the break statement. This statement terminates the innermost loop in which it is contained. If it is executed inside of a for loop, the loop variables will retain the values they had when the break statement was executed.

The continue statement terminates only the current iteration of the innermost loop. Execution of the loop continues with the next iteration (if any). If the continue statement is executed inside of a for loop, the next iteration begins with the execution of the <incr> expression.

The following is an example.

```
if   a==b
     break
else
     continue
endif
```

6.4.11 The message statement

The message statement can be used to include arbitrary messages in the output of the driver. The format of a message statement is given below.

```
message   <file number>,"<message>",<expression list>
```

The file number is the same as that found in the read and write statements. It must be an expression that evaluates to a number from zero through ten inclusive. If zero is specified, the

message is directed to the standard output of the UNIX command used to invoke the simulator. File numbers from 1 through 10 refer to the arguments of the command used to invoke the simulator. The specified argument is treated as a file name and opened for output. The message is written on this file.

The message is arbitrary text enclosed in quotes. The message will appear on a separate line in the specified file. It will be preceded by the characters "* " (asterisk, blank) to distinguish it from monitor output. If the message contains the character "%" substitution of expression values will be performed in the same manner as for the C function "printf" (s.v.). Expression values are used in left-to-right order just as for printf. Any variable conversions acceptable to printf are acceptable to the message statement. The following is an example.

```
message   0,
   "The values are %d and %d (%8.8x and %8.8x in hex)",
   a+1,b,a+1,b
```

6.4.12 The error statement

The error statement is used display error messages on the standard error output of the UNIX command used to invoke the simulator. Its format is identical to that of the message statement, except the file number is omitted. The following is an example.

```
if (a*b)!=result
   error "Invalid product.  %d*%d SB %d, received %d",
      a,b,a*b,result
endif
```

As in the message statement, the error message will appear on a separate line in the standard error output.

6.4.13 The clock statement

The clock statement is used to cause clock variables to be automatically updated after every "go" statement. Clock statements are declarations and are not executable. The following is an example of a clock statement.

```
   clock     abc,0,1,2,4
```

The name "abc" may refer to either a variable or a net. The initial value of the variable or net will be 0. After the first "go" statement is executed the value of "abc" will be 1. After the second "go" statement its value will be 2, and so forth. The value changes only after the "go" statement is completely executed.

6.4.14 The count statement

The count statement is used to increment a counter after every "go" statement. By default the initial value of the counter is zero, there is no final value, and the increment is 1. The count statement is a declaration and is not executable. The format of the count statement is given below.

```
   count     <net or variable name>
```

The count statement may be supplied with from 1 to 3 operands in addition to the net or variable name.. If one operand is supplied, it is assumed to be the inital value of the counter. If two operands are supplied, they are assumed to be the initial value of the counter followed by the

increment. If three operands are supplied they are assumed to be the initial value followed by the final value followed by the increment. When a final value is supplied, and the value of the counter exceeds the final value, its value will be set back to its inital value. Otherwise a counter is incremented without bound.

6.4.15 On conditions

The on statement is used to test a condition after the execution of each "go" statement. If the condition is true, a set of specified actions will be taken. An "on" block is a declaration, and is not executable. The format of an "on" block is given below.

```
<label>:  on   <expression>
               <arbitrary statements>
          endon
```

The "on" statement acts as if it were an if statement that followed every "go" statement. When the expression evaluates to true (non zero) the statements between the on and the endon statements are executed. The on conditions are tested *before* updates due to clock and count statements are performed.

The "deactivate" statement may be used to disable an "on" block. The format of the deactivate statement is given below.

```
deactivate     <on block label>
```

When a deactivate statement is executed, checking of the "on" block's condition is suspended, and the statements between the "on" and the "endon" statements will not be executed regardless of the value of the expression.

The "activate" statement may be used to negate the effect of a previous "deactivate" statement. Its format is given below.

```
activate  <on block label>
```

The label of the "on" statement is optional if "activate" and "deactivate" statements are not used.

The initial state of all "on" blocks is "activated". A deactivate statement may be used to deactivate the on block before the first "go" statement. It is also possible to define an "on" block that is initially deactivated by replacing the "on" keyword with the "xon" keyword as illustrated below. An "xon" block may be terminated with either an "endon" statment or an "endxon" statement.

```
<label>:  xon  <expression>
               <arbitrary statements>
          endon
```

6.4.16 The include statement

The include statement can be used to include a file of pre-written driver commands in the current driver. The include statement is illustrated below.

```
include "my.dir/my.file"
```

The single operand of this command is a file name, which should be enclosed in quotes. (If the file name is a legal driver-language name the quotes may be omitted.) The all driver

commands listed in the file are compiled as part of the current driver. Include commands may be nested arbitrarily.

6.4.17 Invoking the Interactive Command Interpreter

The interactive command interpreter is invoked using the following command.

```
interactive
```

When this command is executed, a prompt will appear on the user's terminal, and the user may begin entering commands interactively. See section 5 for a discussion of the interactive command interpreter.

The interpret command is used to interpret command files. A command file is a file containing a collection of commands for the interactive command interpreter. This file is passed to the driver in the usual fashion and is executed using the following command.

```
interpret <file-number>
```

6.4.18 Dynamic Output Processors

A dynamic output processor can be used to display the data produced by a monitor command. A dynamic output processor may be attached to any file number from 1-10. The file number must be larger than the number of operands supplied to the command that invokes the simulator. The attach command is used to invoke a dynamic output processor, as illustrated below.

```
attach <file-number>,<command-name>
```

The second argument of this command must be the name of a UNIX command for processing the output. The output of the simulator will appear on the standard input of the command (for example, the UNIX commands "cat" and "pr" can be used as output processors.) When output is directed to a dynamic output processor, the output buffer is flushed at the end of every line, to guarantee that the output processor is synchronized with the simulator.

The detach command can be used to reverse the effect of an attach command. This command sends an end-of-file indicator to the output processor, but does not force the termination of the command. However, the output processor (which runs as a separate process) must terminate before this command will complete its execution. The following is an example of the detach command.

```
detach <file-number>
```

6.4.19 6. The quit statement

The quit statement causes immediate termination of the simulation. Its format is given below.

```
quit
```

6.5 6.5 The Interactive Command Interpreter

The interactive command interpreter is invoked using the "interactve" command in the compiled version of the language. One may intermix "interactive" commands arbitrarily with other commands, or one may test exclusively with the interactive interface using the following driver definition.

```
driver
interactive
enddriver
```

The interactive command interpreter provides a language which is virtually identical to the compiled driver language. There are, however, certain differences which are outlined in this section. Interactive commands are entered in response to prompts. The prompt may have several forms depending on the way commands are entered. The basic prompt is a greater-than sign ">". If a line ending with a comma is entered, the greater-than sign changes to a plus-sign, "+", to indicate that a continuation of the same line is being entered. If a complex command, such as an "if" statement or a "for" statement is entered, the prompt will be preceded by a number to indicate the nesting level. Any partially entered command may be deleted by pressing control-C. Control-C may also be used to prematurely terminate the execution of a loop. (Control-| may be used to kill the simulation in an emergency.) As with the compiled language, a semi-colon may be used to place more than one command on a line.

Interactive commands may not be labeled (except in certain cases where labels are required). Furthermore, it is not necessary to precede the op-code of a command with a space or a tab, although it is *permitted* to do so. Alogic macros may not be used with the interactive language. A native macroing facility is provided instead.

A few commands change function when they are used interactively. The quit command is used to exit interactive mode rather than to terminate the simulation. The interpret command is identical to the include command. The function of the include command does not change. The interactive command may be used within a command file to allow commands to be entered interactively. It is equivalent to "include /dev/tty". Control-D must be used rather than "quit" if it is necessary to exit from the interactive mode back to the command file. The "quit" command terminates the command interpreter, regardless of nesting levels. Control-D may always be used to exit from interactive mode, regardless of how it was invoked. Labels are *required* on all "on" and "xon" commands. With these exceptions, all commands described in section 4 above may be used interactively.

There are also a few new commands that enhance the utility of the interactive interface. These are described below.

6.5.1 The help command

The help command displays a list of available command names. It is illustrated below

```
help
```

6.5.2 The show commands

The show commands are used to list the names of variables, signals, macros, and on conditions. They may also be used to show the contents of interactively entered "on" blocks and interactively entered macros. The "showv" command, or its alias "ls" is used to display the names of all variables, on conditions and interactively entered macros. Each variable will be displayed along with its current value. The name of each "on" block will also displayed with its trigger condition. If the on block is deactivated, it will be so indicated. This list includes both compiled variables and on conditions as well as those that were defined using the interactive interface. (Unlabeled "on" blocks will not be listed.)

The "shows" command is used to display the names of all signals defined in the circuit description. The output of the "shows" and "showv" commands appears in dictionary order,

which is not necessarily alphabetic. This problem will be corrected in the future. The showv and showon commands are illustrated below.

```
showv
shows
```

ls

The "showon" statement is used to display the contents of an interactively specified on block. (The contents of compiled on blocks cannot be displayed.) It is specified as follows.

```
showon <on-block-name>
```

Similarly the "showm" statement is used to display the contents of an interactively specified macro.

```
showm <macro-name>
```

The output of these commands is the list of statments that comprise either the on-block or the macro. For complex commands such as "while," "if," and "for" "goto" statements inserted by the parser will appear in this list. The parser breaks "for" statements into two "set" statements and a "while" statement (and several "goto" statements), so "for" statements will look peculiar in this list.

6.5.3 Interactively specified macros

For the compiled version of the language, macros are specified using the FHDL macro processor. Since this macro processor is not available at run time, the interactive command interpreter provides a native macro facility. A macro is defined as follows.

```
<label>: macro <argument-names>
     ... <macro body> ...
     endmacro
```

The macro body may contain any interactive statement except "on" or "xon." (Macro definitions may not appear inside of an "on" block.) The list of argument names is optional. An example of a macro is given below.

```
cycle: macro
       go;go
       endmacro
```

Once a macro is defined, its name may be used as a command, so the following command would cause two "go" statements to be executed.

```
cycle
```

When argument names are specified, each name defines a local variable whose value is supplied from the command line that invoked the macro. (The name may duplicate that of an existing object such as a variable or on block.) Furthermore, new local variables are generated for each call, so recursive macros will work properly. The following is an example of a macro with arguments.

```
test1: macro a,b
     for 0->bus1,bus1<a,bus1+1->bus1
     for 0->bus2,bus2<b,bus2+1->bus2
     go
     endfor
     endfor
     endmacro
```

This macro would be invoked as follows.

```
test1 5,10
```

The number of arguments on the invoking command line does not have to match the number of arguments in the definition. Extra arguments are ignored, while missing arguments are assigned the value zero. If a variable or signal name is used as an argument, it will be passed by address, so macros may perform side-effects on their arguments. Constants and expressions are passed by value, so an argument of "(x)" will cause the variable x to be passed by value.

If a "variable" statement appears within the body of a macro, it defines local variables that exist only while the macro is being executed. Local variables are created and initialized to zero when the execution of the macro begins. If a local variable has the same name as some other object, the local definition will replace the former definition of the name while the macro is executing. The former definition is restored after the macro terminates, and its value is unaffected. Thus local variables will work properly in recursive macros, and it is not necessary for different macros to use unique local variable names, even if they invoke each other.

When a macro is invoked it inherits the environment of its caller. Any variable or signal that was accesible to the caller is also accessible to the macro. It is possible for a nested macro to perform side-effects on the local variables of its caller, although this practice is not recommended.

6.5.4 The remove statement

Any object that is defined through the interactive interface may be "undefined" using the "remove" statement. This statement may be used to remove variables, on blocks, and macros. Variables that are the object of "count" or "clock" statements should not be removed, although this restriction will itself be removed in the future. The following is an example of a remove statement.

```
remove a,b,c
```

CHAPTER 7

Downloading, Installation, and Use

7.1 Prerequisites

Before compiling this software you must make sure the following packages are available and installed on your computer.

yacc (Byacc aliased to yacc.)
flex (plain lex is OK, but you must change some things in the project files and Make files.)
gcc (this is necessary even if you compile under Visual C++)

The package gcc must be in your path. The packages yacc and flex must be available as programs to Visual C++ if you are compiling under Visual C++.

The software is can be compiled using either gcc or Visual C++. (Version 8 for sure, I don't know about the others.) A make file and Visual C++ project file is available for each item. I have compiled these items under Linux, Windows NT and Windows Vista. Dev-C++ should be no problem either, but I haven't included project files for it. If you wish to use Dev-C++, use the Makefile as a guide.

7.2 Downloading the FHDL Components

FHDL is available as a set of software packages on the Baylor Computer Science Technical Report Site.

https://beardocs.baylor.edu/handle/2104/4824//browse-title

This URL may not be persistent. If it doesn't give you a list of technical reports, then go to the following site.

http://beardocs.baylor.edu

Click on the link for the "Engineering and Computer Science" technical reports, then "Computer Science" and then "Computer Science Technical Reports" From there you can browse by author, or title, or search for a specific report. These reports have no numbers, so if you need to reference them, just use the name.

Create a new directory (or *folder* if you prefer) for the software. Each downloaded element should be placed in this directory. I will call this your FHDL directory.

Before getting started, create three directories "include" "bin" and "lib" under your FHDL directory. The bin directory should be in your path and should also be listed under Visual C++'s list of executable files directories, if you are using Visual C++. The lib and include directories should be listed under Visual C++'s list of library and include directories. See below for how to do this.

Download the packages "mkwhat" and "tokdec" and unpack them into your FHDL director. The directories "mkwhat" and "tokdec" should appear in your FHDL directory. You can throw the zip files away. Make sure that you didn't double up the directories. For example, if the only directory under "mkwhat" is another "mkwhat" directory, this is wrong. Move all files from the lower directory to the upper one, and delete the lower one.

Compile both packages. If you're working under Linux, do a "make install" command. Under Visual C++ select the "release" configuration and recompile the entire package. Under Visual C++, compiling the project will create a "release" directory under the project directory. This will contain the .exe file. This file must be moved to your "bin" directory before proceeding. Under Linux, nothing more needs to be done.

The additional packages are:

connlib
Fhdl
Alogic
Romasm
Plasm
Driver

Not all of these are currently available, but they will be released as soon as they have been cleaned up and repackaged.

With the exception of connlib, all packages create an executable file. These files should be placed in your bin directory. The "make install" command will do this automatically if you are running under Linux. Under Visual C++ you will have to move the files yourself. (If you're good

at hacking Visual C++ projects, you can make this automatic.) The executable files will appear in either the debug or release directories under your project directory, depending on which project configuration you use. I recommend using the “release” option, unless you’re having trouble and need to debug the software.

The output of the connlib project is a library that will be used by other packages. It should be placed in your “lib” directory. Again, under Linux, this will be done automatically. For both Linux and Visual C++, go to the connlib project directory and *copy* DON’T MOVE the three .h files to your include directory.

To use connlib in Visual C++, you must add the file “connlib.lib” to your project as a linker input file. (Go to project settings, and select the input option under linker options.)

7.3 How to make stuff available to Visual C++

Start Visual C++, go to the “tools” menu and select “Options …”. You may have to open a project to make this menu appear. Any project will do.

Go to the “Projects and Solutions” section, and click on the + to make the sub-menu appear. From the sub-menu, select “VC++ Directories”.

A drop-down box will appear on the right containing several options, including “Executable Files”, “Include Files” and “Library Files”. Add your bin directory to the “Executable Files” list, add your include directory to the “Include Files” list, and your lib file to the “Library Files” list.

Once you make these changes, they are effective for all projects.