Using the Connlib Package to Obtain Parsed Netlist Data

Peter M. Maurer
Department of Computer Science
Baylor University
Waco, TX 76798

1 Abstract

The connlib package can be used to obtain parsed netlist data from ".ckt" files. These files must be created using the Functional Hardware Description Language FHDL. This data can be used in any way you choose. The ISCAS85 and ISCAS89 packages are available as .ckt files, so these benchmarks can be used to test the software you create.

2 Running the Package

To run the package, you must include file "conndefs.h" in your program. This file is part of the connlib package, and should be placed in one of your include directories. To parse a file, you must first open the file using the stdio package, and then pass the open file to the parser. This is done in the following manner:

```
FILE *MyFile;
char * FileName = "Something.ckt";
MyFile = fopen(FileName,"r");
if (MyFile==NULL)
{
     fprintf(stdout,"Can't open %s\n",FileName);
     exit(100);
}
parse(MyFile,1);
fclose(MyFile);
nwfixwire(nwhead);
if (cktbad)
{
     fprintf(stdout,"Circuit is bad\n");
     exit(100);
}
```

The connlib function "parse" is used to parse the file. The second parameter of 1 indicates that this is a new compilation. You can parse several files together by setting this parameter to 0 for all but the first file. Once the file is parsed, the circuit is not yet completely built. You must call the function "nwfixwire" with the global variable "nwhead" to complete the construction of the circuit. Once both of these steps are complete, you must check the global variable "cktbad" to

determine if there were any compile errors. If there were, cktbad will be equal to 1 and error messages will have been issued. If cktbad is zero, then there were no compile errors.

After you execute this code, the parsed content of the .ckt file is in the global variable “nwhead” and you’re on your own.

3 Using the Parsed Data.

After a successful parse, the variable “nwhead” points to a linked list of NETWORK structures. The final NETWORK in the list will have a “next” pointer equal to NULL.

3.1 The NETWORK Structure

The NETWORK structure is defined as follows.

```
typedef struct network
{
   struct network *next;
   DICTE *id;
   int type;
   int gate_count;
   int net_count;
   int input_count;
   int output_count;
   int io_count;
   int attr_count;
   struct attrhold *attr;
   int fault_count;
   struct fault *faults;
   long user;
   int  ref;
   char *gname;
   NET **netlist;
   GATE **gatelist;
   NET **inputs;
   NET **outputs;
   NET **ioputs;
}
NETWORK;
```

The variables are defined as follows.

next	Used to create linked lists of NETWORK structures.
id	Points to a dictionary entry. The dictionary entry, in turn, contains the name of the circuit. Unnamed circuits have generated names.
type	This variable will contain one of the following defined constants SUBNET, ROM, RAM, PLA, or ASM. In most cases the type will be SUBNET, but if

	you have other kinds of circuits in your .ckt file, you will see some of the other types.
gate_count	Gives the number of gates in the circuit.
net_count	Gives the number nets in the circuit
input_count	Gives the number of primary inputs.
output_count	Gives the number of primary outputs.
io_count	IOPUT count. Always zero for the current version of FHDL
attr_count	Gives the number of attributes of the network
attr	The user can add attributes to the network statement like this: X: network Attr1=A,Thing2=(a,b,c) Each attribute is stored in an attr structure, and these are formed into a linked list, the head of which is pointed to by this variable. None of these attributes mean anything to FHDL, so use them as you wish. See below for the structure of an attribute.
fault_count	Gives the number of faults in the circuit. Faults can be associated with a net or with the circuit itself. They mean the same thing regardless.
faults	Points to a linked list of fault structures. See below for the format.
user	Not used by FHDL.
ref	Used by the circuit flattener. Not used otherwise. Indicates whether the circuit is referenced by another.
gname	Used by the code generator. Not used otherwise. This is the function name of the function created to simulate the circuit.
netlist	Points to an array of pointers to NET structures. The array will contain one structure for each NET in the circuit. This array MAY NOT be sorted.
gatelist	Points to an array of pointers to GATE structures. There is one gate structure per gate in the circuit. This list may be sorted if you wish.
inputs	Points to an array of pointers to NET structures. There will be one NET structure per primary input, listed in the order that the primary inputs were specified. This actually points to the “netlist” array.

outputs — Points to an array of pointers to NET structures. There will be one NET structure per primary output, listed in the order that the primary outputs were specified. This points into the “netlist” array.

ioputs — Not used in the current implementation of FHDL.

3.2 The DICTE structure

The DICTE structure is represents an entry in the parser’s hash-table. These structures play double-duty in supplying the names for NETWORKs, NETs, and GATEs. The DICTE structure is as follows.

```
typedef struct dicte
{
   struct dicte *next;
   char *name;
   short type;
   short hashp;
   ELEM elem;
}
DICTE;

typedef union dicteu
{
   struct net *n;
   struct gate *g;
   struct network *w;
}
ELEM;
```

The variables are defined as follows.

next — Pointer to the next element. The hash table is a 256-element array of linked lists. The index is the hash of the name.

name — The name of the element. Null-terminated string.

type — ISGATE, ISNET, or ISNETWK

hashp — Hash of the variable name.

elem — Back-pointer to the named element. (See union type above.)

3.3 The ATTRHOLD structure.

The ATTRHOLD structure is defined as follows. NETs, GATEs, and NETWORKs can have attributes. Each attribute is contained in one ATTRHOLD structure. All attributes for a particular element are collected into a null-terminated list pointed to by the element.

```
typedef struct attrhold
{
   struct attrhold *next;
   char *name;
   int valcount;
   char *val[1];
}
ATTRHOLD;
```

The variables are defined as follows.

next	Pointer to the next item in the list. NULL if this is the last.
name	The string on the left-hand side of the equal sign.
valcount	The number of values specified on the right-hand side of the equal sign.
val	An array (NOT always of size 1) containing each value specified on the right.

3.4 The FAULT structure

Although the FHDL system has never been used for fault simulation, there is no reason why it couldn't be. You can specify faults in the FHDL language, and if you do, they will be stored in the following structures.

```
typedef struct fault
{
   struct fault *next;
   GATE *fgate;
   short dir;
   short pos;
   int type;
   char *type2;
}
FAULT;
```

Faults are specified using the FHDL "fault" statement. It works like this.

Name: fault MyGate,I3,SA0;
Or
Name: fault MyGate,O0,SA1,Zorro

The field “Zorro” is an extra type field for future expansion.

The variables are defined as follows.

next	This variable is used to create linked lists of faults. The final element in the list has this variable set to NULL.
fgate	Pointer to the gate that the fault is associated with. Obtained from the first parameter.
dir	0=input, 1=output, obtained from the first character of the second parameter.
pos	Position in the input or output of the affected net. Obtained from the second parameter.
type	Type of fault: SA0, SA1, WFN, WIS, MII, EIS, WIL, SHO, EIR, WIR SA0 and SA1 are stuck at zero and stuck at 1. I forget what the others are. I think they are Wrong Function, Wrong Input Signal, Missing Input Signal, Extra Input Signal, and … ? Hmmm.
type2	Extra type field, stored as a string.

3.5 The NET Structure

The net structure is used to hold information about nets. The basic connectivity information is gathered from the input and output lists of gates. Additional information can be added using the “wire” statement.

```
typedef struct net
{
   struct net *next;
   DICTE *id;
   unsigned short type;
   unsigned short width;
   int index;
   int scc_index;
   int input_count;
   int output_count;
   int attr_count;
   struct attrhold *attr;
   int value;
   long user;
   char *gname;
   struct gate *gates[1];
}
NET;
```

The variables are defined as follows.

next	This is used to create a linked list of NET structures. By the time you get this structure, this variable will be unused, so you can use it for whatever you wish.
id	Points to a dictionary entry giving the name of the net.
type	Type is PI, PO, PIO (not currently used), GI, GO, CZERO, CONE, ACTIVE_LOW, NO_CONNECT. These types can be combined, so nets typically have several type codes. Use AND OR and NOT to check for specific codes.
width	Normal value is 1. Can be greater than 1 if the net is to be considered a bus. This can have any value, but most of our code-generators won't accept anything larger than 32. What you do is up to you.
index	Unused by FHDL. Many connlib applications use this as a back-index into the netlist array. Use it for anything you wish.
scc_index	Unused by FHDL. This was added for a specific application that partitioned circuits into fanout-free networks, and by another application that partitioned sequential circuits into strongly connected components. Use it for anything you wish.
input_count	Number of gates that use this net as an input. (Roughly. If the net appears twice in the input list of a gate, this counts as two gates.)
output_count	Number of gates that use this net as an output. Same rule about duplicate usage.
attr_count	The number of net attributes. These come from the WIRE statement.
attr	A linked list of ATTRHOLD structures, one per attribute.
value	Not used by FHDL. Used by interpretive simulators to store the value of the net. You may use it for anything you wish.
user	Unused by FHDL. Use it for anything you wish.
gname	Used by code generators to generate variable names to hold the net values. If you don't use our code generators, you can use it for anything you wish.
gates	An array of pointers to GATE structures, usually larger than 1. The size of the array is equal to input_count+output_count. The gates that use the net as an input are listed first, then the gates that use the net as an output. The gates are listed in no particular order.

3.6 The GATE Structure

Every gate in the circuit is assigned to a GATE structure. The GATE structure has the following format

```
typedef struct gate
{
   struct gate *next;
   DICTE *id;
   char *ctype;
   int type;
   long type2;
   int index;
   int scc_index;
   int input_count;
   int output_count;
   int attr_count;
   struct attrhold *attr;
   struct fault *faults;
   long user;
   char *gname;
   struct net *nets[1];
}
GATE;
```

The variables are defined as follows.

next — Used to create linked lists of gates. By the time you get this structure, this variable will be unused, so use it for whatever you wish.

id — Pointer to a dictionary entry giving the name of the gate. Unnamed gates will be given a generated name before they get to you.

ctype — The character-string version of “type”. If the type of a gate is not recognized, it is assumed to be a subnetwork reference. “type” will contain SUBNET and “ctype” will contain the name of the network being referenced.

type — One of the following codes: SUBNET AND OR NOT NAND NOR XOR XNOR AOI OAI HLCV BUF BUFI TGATE TGATEI TBUF TBUFI SCC SUBCKT DFF1 DFF2 DFF3 DFF4 RSFF JKFF1 JKFF2 JKFF3 JKFF4 TFF TFF1 TFF2 TFF3 TFF4 DEMUX DECODER ENCODER ALU ADDER COMPARATOR MULTIPLIER DIVIDER REGISTER COUNTER ASM_STATE ASM_TEST ASM_COND_OUT EXPAND DISTRIBUTE COLLECT ROMWORD PLAWORD
BUF is not the same as BUFF. BUFF gates are non-inverting amplifiers with a single input and a single output. They are converted to type HLCV before you ever see them. Remember HLCV is a BUFF gate. BUF is a tristate buffer with a

	data input and a control input. AOI and OAI gates never are specified as bare AOI and OAI. The ctype must be examined to find the true gate type. In cytpe, the AOI or OAI prefix will be followed by a non-ascending sequence of digits that give the input groupings for the gate. The number of gate inputs will equal the sum of these digits. Thus AOI4321 is an AOI gate with four input groupings, one of size 4, one of size 3, one of size 2 and one of size 1. These groups appear in the input list in the order specified. The function of an AOI4321 is: $(abcd + efg + hi + j)'$. AOI is a negated sum of products, OAI is a negated product of sums.
type2	This variable is used when "type" contains SUBNET. This gives the type of the subnet. It will be one of SUBCKT, ROM, PLA, or ASM.
index	Unused by FHDL. Use this variable for whatever you wish.
scc_index	Unused by FHDL. Use this variable for whatever you wish.
input_count	The number of gate inputs.
output_count	The number of gate outputs.
attr_count	The number of gate attributes.
attr	Linked list of gate attributes, one structure per attribute. Note that gate-types DEMUX through COUNTER in the list above have pre-defined attributes, but nothing else does. You can use attributes to specify gate delays, or anything else you wish. Even in the case of pre-defined attributes, these attributes are used ONLY by the code generator, so you can use them however you wish.
faults	See the discussion of faults under the NETWORK structure. Faults can be attached to the GATE structures or to the NETWORK structure, but usually not both.
user	Unused by FHDL. Use this for whatever purpose you wish.
gname	Used by the code generator to generate a legal C++ name for the gate, should it be necessary that the gate have one. Sometimes this is necessary, sometimes it isn't. If you are not using one of our code-generators, then you can use this variable for whatever you wish.
nets	An array of NET structure pointers. The input list is specified first, in the order the y appear on the gates input list. The output list follows the input list. Again, nets are listed in the order they appear on the gate statement.

4 Other connlib Functions

Memory elements will not be set up correctly unless you call some additional functions. The code given above assumes that you have a single netlist type circuit that you want to play with. If you want to create more complex types of circuits with ASMs, ROMs, and PLAs, you must add the following function calls after the call to “nwfixwire”.

```
asmaud1(nwhead,&asmptr,&hdwptr1);
asmaud2(asmptr);
memaud(hdwptr1,&memptr,&hdwptr);
```

The first function call filters out all ASMs into a separate linked list headed by “asmptr”. Everything else is dumped into a separate list headed by hdwptr1. (Both variables are pointers to a NETWORKs.) The second function call verifies that the ASM specification is semantically correct. The third function call filters out ROM and PLA circuits and places them on a separate linked list headed by “memptr”. Everything else (standard netlists) is placed on the list hdwptr. Both variables are pointers to NETWORKs. After calling these three functions, you should test the global variable “cktbad” to see if any errors have occurred. If this variable is non-zero, there have been errors and you should stop. If it is zero, everything is OK and you can continue.

If you’ve gone to all this trouble, you’ll almost certainly want to call the next three functions.

```
sgtype2(hdwptr);
expand(hdwptr);
audit(hdwptr,0);
```

The first verifies that the type fields of all structures are set properly. The second flattens the circuit. ROMs, PLAs, and ASMs are not subject to flattening, that’s why the other functions filter them out. The final function call does a final semantic check on the flattened circuit to flag any remaining errors. The zero parameter says that IOPUTS are not allowed.

When the circuit is flattened, the main circuit is assumed to be the first one in the list. Any time a circuit reference is found, the entire list is searched for a matching name. Thus the order of the specifications is arbitrary. Circuit references may not be circular. This would lead to a circuit of infinite size and is definitely considered to be an error.

You should check the global variable “cktbad” after calling expand and after calling audit.

If you have gotten this far, you may wish to call one of our code generators. They are all named “makec”. The connlib version of makec generates a C function that can be used to simulate the circuit. The code is generated into an open output file (of stdio variety) that must be supplied by you. Here is a sample function call:

```
FILE *cfile;
cfile = fopen("MyCFile.c","w");
makec(hdwptr,cfile);
```

The function makec does not generate a main routine. You can, of course, supply your own, but I’d recommend using our main routine generator, genmain. We have many versions of makec. Each one requires its own version of genmain. To call genmain add the following lines after the makec call.

```
genmain(hdwptr,cfile);
fclose(cfile);
```

Most of our compiled simulators will now execute gcc to compile the generated code. What you do at this point is up to you.

5 ROMs, PLAs and ASMs.

ROMs, PLAs, and ASMs all act as if they were a single combinational gate. To be used they must be referenced from a conventional circuit. They cannot be simulated in isolation. The content of these types of circuits is highly restricted. ROMs may contain only ROMWORD type gates. PLAs may contain only PLAWORD type gates. ASMs may contain only ASM_TEST, ASM_STATE, and ASM_COND_OUT type gates.

5.1 ROMs

ROMWORD gates appear in the ROM circuit in ascending order by address, starting with address 0. Addresses may not be skipped, but it is not necessary to fill out the entire address space. The missing words will be treated as zeros. There may be many output fields of differing sizes, or a single bus-type output field. This information is taken from the primary output list of the ROM NETWORK structure. Each ROMWORD gate must have a value for each output field. These fields are in null terminated character string format and are stored in the net array of the GATE structure in the order specified. Each element of the net array contains a pointer to a character string. The character strings are assumed to be in hexadecimal format without the leading 0x.

5.2 PLAs

PLAWORD gates appear in the PLA circuit in the order specified, although this order is not particularly important. Each PLAWORD gate must have an input operand followed by one or more output operands. These operands are stored in the net array in null-terminated character string format. The input count of the PLAWORD will be 1 and the output count will equal the number of output fields of the PLA. The number of output fields is determined by the output list declaration of the PLA NETWORK. The input string is in trinary format containing the characters 0, 1, and x, where x represents “don’t care”. This is the “address” of the PLAWORD. The output fields are assumed to be in hexadecimal format without the leading 0x.

5.3 ASMs

ASM gates are linked pretty much like ordinary gates. (Read the FHDL manual.) Things that look like inputs and outputs are treated as input and output links. Additional linkage information is supplied through the attributes. (Yes, those things that look like attributes are stored as attributes.) Even though the ASM has an internal state, it is treated as if it were an ordinary combinational gate.