

Technology Mapping in Circuit Design Aids

CDA IS A COLLECTION of small computer-aided design programs used by about a dozen designers for board-level medium- and large-scale hardware. These programs have evolved continuously over the last 15 years to keep pace with the increasing variety and diversity of chips.

Fitter programs attempt to convert logical equations to specific target devices or technologies. We say "attempt" because it's possible that the input specification will not fit onto the device; either because the specification is too large or possibly because the device lacks a feature required in the specification. (An example would be asking for a register from a combinatorial device.)

The flow graph in Figure 1, next page, illustrates the use of the fitters in the CDA context. The symbolic product terms in the figure are the output format of several programs, but we don't describe them here.

CDA (Circuit Design Aids) contains six device-specific fitters: Part, Npart, Act, Act2, XNF, and XIL. Part targets common first-generation programmable logic device (PLD) parts commonly called PALs. Npart targets the newer Advanced Micro Devices MACH series. MACH parts are a two-level logic es-

MARK KAHRS
Rutgers University
BART N. LOCANTHI
ROBERT C. RESTRICK III
AT&T Bell Laboratories

CDA (Circuit Design Aids) is a collection of tools for the design of digital systems. These tools transform product term input into fuse maps for a variety of programmable parts including programmable logic devices (commonly called PALs) and field-programmable gate arrays.

entially composed of 22V10 input blocks interconnected through a network. Trimberger¹ calls this device class complex PLDS (CPLDs). Both Part and Npart produce partitioned product terms that the Xpal fuse compiler converts to JEDEC² fuse format.

Act produces a variety of output forms including ADL format for Actel's place-and-route software. Act2 is a similar program for the newer ACT 2 chips. The XNF program translates the sum of products to Xilinx Netlist Format; Xilinx software for the 3000 series does its own

minimization. XIL is something of a blend between XNF and Act, performing some optimization (such as tree factorization) in preparation for the Xilinx 4000 series place-and-route tool. Act produces a format suitable for the Actel place-and-route software.

A simple model

Whatever the target device, we use the same simple objects to specify the desired behavior. These objects have a combinatorial section followed by a control section, as shown in Figure 2a-f. The control section may be simply a buffer, an inverter, a clocked device, a tristate driver, or combinations of these (one such example is shown in

Figure 2f). A clocked device can include D flip-flops, toggle flip-flops, set-reset latches, or whatever the input language of the equation translator can syntactically describe.

A simple PAL like the 16L8 (16 inputs, 8 inverted outputs) requires a control section that is just a simple inverter followed by a tristate driver, while a 16R8 (16 inputs, 8 registered outputs) would use the control sections shown in Figure 2f. More sophisticated devices allow the selection of arbitrary configurations via additional fuses. Even though the mod-

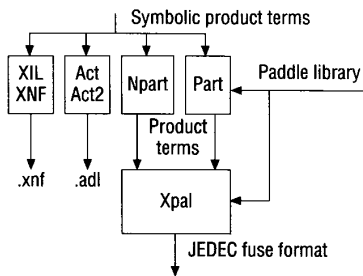


Figure 1. CDA fitter usage.

el is simple, designers have used it for random logic sections of ICs as well as the programmable parts we describe here.

Consider the simple four-bit counter shown in Figure 3. The *x* output nodes are D flip-flops clocked by the clock input *clk*. The *a* input variables are on the load side of the counter, and the load pin loads the counter synchronously. The symbolic product term description of this counter is

```
.o x0 load a0 x0
  0:5 3:3
.o x1 load a1 x0 x1
  3:3 4:13 8:13
.o x2 load a2 x0 x1 x2
  3:3 12:29 16:25 16:21
.o x3 load a3 x0 x1 x2 x3
  3:3 28:61 32:49 32:41 32:37
.o x0@d clk
  1:1
.o x1@d clk
  1:1
.o x2@d clk
  1:1
.o x3@d clk
  1:1
```

This takes the form

```
.o output_node(optional attribute)
  input_nodes
```

followed by product terms (in decimal [sic]) as value:mask. Value and mask are bit vectors with the least significant

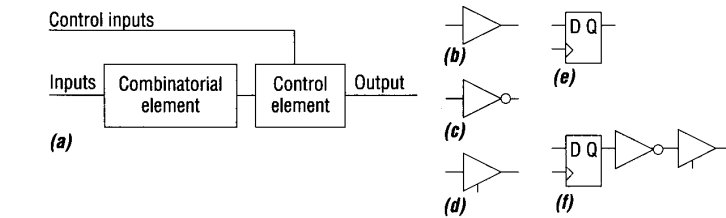


Figure 2. Programmable device having a combinatorial element followed by a control section (a). The control section may be a buffer (b), inverter (c), tristate device (d), clocked driver (e), or a combination (f).

bit corresponding to the first input_node and the most significant bit corresponding to the last rightmost input_node. A zero in the mask portion indicates a don't-care option; a zero in the value portion means "complement that input." The mask:value product terms are natural for manipulation such as using Quine-McCluskey³ minimization, Shannon factorization, or if-then-else partitioning.^{4,5} Even so, the sum-of-products representation for a counter gets out of hand very rapidly for large counter sizes.

Optional attributes pass information (such as clocked element declarations) to the other fitters. For example, @d declares a D flip-flop while @t declares a toggle flip-flop. The use of toggle flip-flops often reduces the number of terms required for counter-like sequential logic. Figure 4 displays the output for a 4-bit counter using toggle flip-flops.

Note how the number of x3 terms has been reduced from five to three. The fitter interprets the @t operator (and any other optional attributes).

The job of a fitter is to take the symbolic product term input and produce product terms acceptable for any one of numerous devices:

- Part, for common programmable devices
- Npart, for the AMD MACH devices
- Act and Act2, to compile into Act1 library gates for the ACT 1 and ACT 2 families

Four-bit counter			
Input 1	a0	x0	Output 1
Input 2	a1	x1	Output 2
Input 4	a2	x2	Output 4
Input 8	a3	x3	Output 8
Load	Load		
Clock	Clk		

Figure 3. A 4-bit counter description.

- XNF and XIL, to compile into Xilinx intermediate form for the 3000 and 4000 series parts.

Part. This program tries to squeeze product terms into a set of programmable parts of one particular variety. Part starts with a single instance of the target device and attempts to assign output functions to the outputs. If it encounters an output function that can't be assigned, Part considers a second device. It continues assigning outputs and, when necessary, adding devices until all output functions are assigned.

Part uses the Assign function to return Fit, a measure of how well a set of output functions fit on a particular device: the smaller its value, the better the fit. Assign first assigns the output signals by calling a procedure Find(terms, type) that looks for pins of a given type and a given number of terms. The bits of type map to functional requirements such as input, output, inverting, noninverting, clocked,

```

.o x0 load a0 x0          # x0 depends on load, a0, x0
  0:1 2:6 4:6           # product terms
.o x1 load a1 x0 x1
  3:11 4:5 9:11
.o x2 load a2 x0 x1 x2
  3:19 12:13 17:19
.o x3 load a3 x0 x1 x2 x3
  3:35 28:29 33:35
.o x0@t clk              # x0 '@t' (toggle ff clock) attribute
  1:1
.o x1@t clk
  1:1
.o x2@t clk
  1:1
.o x3@t clk
  1:1

```

Figure 4. Four-bit counter with toggle flip-flops.

and so on. Declarations supply the corresponding properties of the pins of a particular device. Find selects a pin of the right type that uses the least number of available product terms. For an output-only pin, Find will prefer an output-only node over one that can be an input and/or an output. Often Find sends both the product terms for a function and those for its complement to the fitter. In this case Assign selects from the possible output polarities. If an output cannot be assigned to the device at all, Assign returns a special large value.

Each output signal carries with it inputs that must also be assigned (via Find). These inputs can feed into the combinatorial (AND/OR) array or serve as special-function inputs (clock, enable, clear, or set). If it can't assign all the input pins, Assign returns a positive number proportional to the number of missing inputs. Otherwise, Assign returns a negative number decreasing proportionally with the number of extra pins. By weighing deficits more highly than surpluses, the minimization of the total fit of the device results in a search for an acceptable configuration.

Consider two outputs o1(i0,i1,i2) and o2(i0,i2). Assigning them to separate de-

vices would require a total of five input pins. If they were both on the same device, i0 and i2 could be shared, requiring only three input pins. If the initial placement leaves input pins unassigned, Part uses the Kernighan-Lin^{6,7} min-cut graph-partitioning algorithm to minimize the number of unassigned pins. The algorithm successively selects and swaps pairs of outputs on separate devices so that the total fit is reduced.

If Part is successful, it generates several files. Each device contains a pin assignment file and a product term file. Part also generates a graphic schematic file showing the connectivity of the devices in the case of multiple devices.

The output of Part for the 4-bit counter on a 22V10 would be

```

.x 22V10
.o 14 6 5 14
  3:3 0:5 2:6
.o 15 6 3 14 23 15
  3:3 12:29 16:21 16:25 14:30
  18:22 18:26
.o 22 6 2 14 23 15 22
  3:3 28:61 32:37 32:41 32:49
  30:62 34:38 34:42 34:50
.o 23 6 4 14 23
  3:3 8:13 4:13 10:14 6:14

```

Notice how the pin numbers have been assigned to each of the output product terms where the symbolic node names used to be. In the case of inverting outputs, some devices provide feedback to the combinational array *before* the inversion, so that the product term feedback must be inverted. Noninverting outputs are left alone.

Npart. Many common programmable devices fit within the scope of Part and the Paddle description language. Some devices have multiple blocks that look like conventional programmable devices connected through programmable, on-chip interconnection network devices called complex PLDs. Typical of these devices is the MACH series. The many trade-offs between speed and complexity limit the interconnection network in size. Therefore the particular signal assignment dictates whether or not a device will fit.

On one hand, these devices look like multiple, simple programmable devices and fit into the scheme of Part particularly well. On the other hand, the constraints of the interconnection network prohibit a simple language description. A new program Npart supports such devices. It accepts that the blocks are part of a larger group (the devices themselves) and, like Part, assigns outputs to blocks, minimizing the number of inputs and hence the interconnections between the blocks. Doing this first step well profoundly affects the success of the succeeding signal assignment/interconnection step.

Once the outputs have been assigned to their respective banks, Npart must assign the signals to actual pins or internal nodes and configure the interconnection network. As with Part, each signal receives a bitmapped value, with the more constrained requirements being assigned to higher order bits. For example, the most significant bit denotes frozen signals (that is, preassigned pins). Clock inputs that must be as-

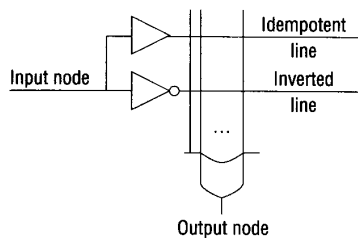


Figure 5. An AND/OR array.

signed before regular inputs (because they must be assigned to dedicated pins) use the next highest order bit. Npart sorts the signals by the bitmap value in decreasing order and then assigns them, ensuring that the most difficult pins are assigned first.

Npart assigns a signal by considering each unassigned node of matching type. The heuristic used in selecting a particular node follows: Each output node may be connected to a set of input nodes distributed over the banks of the switch. These sets of nodes may intersect since output nodes may share inputs. This implies that connecting an output to some inputs will reduce the size of any set containing those inputs. The number of degrees of freedom of an output node is defined as the number of input nodes that may be reached by a given output node. The total number of degrees of freedom of all the output nodes serves as a metric, and nodes are assigned such that this metric is maximized.

As an example, suppose we have n outputs, each capable of being connected to m inputs. Initially the number of degrees of freedom would be $n*m$. Selecting any output reduces this number by n plus the number of instances where the inputs to which the output is to be connected appear as possible destinations for the other outputs. If possible when the next pin is selected, it will come from one of the outputs whose input set has been reduced.

As nodes are assigned, if a signal can-

not be connected, Npart backtracks by removing assigned nodes one by one, trying to reassign the problem signal. If successful, Npart reassigns the removed signals. This backtracking may be recursive to a depth limited by a command line argument. Though the default search depth is set to two, difficult cases may use values of three. Search depths larger than three have long runtimes and don't seem to be very effective.

The previously generated pin assignments can be read into both Part and Npart. These pins are frozen so that wiring changes are not required. Both tools attempt to partition the product terms within these constraints, assigning unfrozen pins and buried outputs. It is possible that additional functions would not fit into the device with the new pin restraints. This would be a disaster since pins will either have to be moved (thereby resulting in numerous wiring changes) or new devices introduced (into an already full circuit board).

Xpal. Given the output of fitters like Part and Npart, the next step is the compilation of logical terms into a map of fuses that describes the connections of the AND/OR arrays. The fuse numbers are output in JEDEC standard fuse format, the input accepted by most device programmers.

Since the world of programmable parts is ever expanding, we designed the Paddle (Programming Array Device Description Language) language as a side input to Xpal. Paddle's underlying model combines a set of AND/OR arrays and an optional array of fuses. Figure 5 illustrates an AND/OR array.

The input to an AND/OR array is a set of nodes; the output is also a set of nodes. A node can be externally visible (such as a pin), or it may be an internal node. The buffered or inverted version of a node is called a line; a fuse controls a line's intersection with the input term of the OR. The collection of fuses over a set

of input and output nodes is an array.

A device could have multiple AND/OR arrays. Examples of such devices include the MACH parts, the Cypress 7C361 finite-state machine controller, and the Signetics Macrologic (PLS 501, 601, and 701) parts. Every array declaration therefore must be followed by a symbolic name, which also helps Xpal to give useful error messages. Each array must also have an offset declaration, which permits the array to be placed anywhere in the fuse space. Next, the declaration of input and output nodes must be given. Note that in most PLDs, input lines come in both complement and idempotent flavors and are provided courtesy of the buffer on the input pin. Paddle therefore contains a declaration for doubling a given set of nodes, with either the complement or idempotent line first.

Fuse arrays let Paddle declare special fuses that some programmable parts offer. For example, the 22V10 contains fuses that set the polarity of the output pin, whether the output is latched or combinatorial and so forth. Since these fuses are not organized in an AND/OR array, Paddle just permits a one-to-one mapping between artificial pin (external node) numbers and fuse numbers.

Compilation of product terms to fuse numbers. Given a product term for a particular output pin, Pal verifies that the number of input terms is less than the maximum number of terms on that output pin. If Part or Npart processed the input to Xpal, this is not a problem since Part generates properly partitioned equations.

Next, Pal locates the beginning of the AND/OR array in fuse space. Figure 6 shows a compilation of a given output pin.

Pal calculates the fuse number for each implicant by adding the pin number to the term number of the output pin. This is multiplied by the maximum number of input nodes per line to get

```

foreach term do {
  output_line = (term number + line number of output pin) *
    max width of fuses per line;
  value = implicant[term].value;
  foreach bit_number do {
    bit_mask = 1 << bit;
    if don't care, then loop;
    pin = input_pin[bit_number];
    input_line = input_line[pin];
    pin_flags = flags[pin]; /* per pin flags */
    if (value & bit_mask) { /* don't complement */
      if (pin_flags & COMPLEMENT_1ST) /* complement on 1st line */
        input_line++; /* idempotent on 2nd! */
    } else /* complement */ {
      if (!(pin_flags & COMPLEMENT_1ST))
        input_line++;
    }
    zero_fuse(input_line + output_line + array_offset);
  } /* end foreach bit_number */
} /* end for term */

```

Figure 6. Compilation of an output pin.

the starting fuse number of the first line in the output node.

Each input variable has a binary value and a mask that says whether the variable is used in the product term. Assuming the pin is really an input pin, Pal checks the input for a complement and the device for an available input line. Assuming an input line is available, Pal accepts the fuse number to be the array offset plus the first line fuse (just calculated) plus the input line number. Pal sets this fuse to the default value (don't blow).

Note that the fuse value is not output at this time; it is saved. The final act of Xpal is to write and output the array of changed fuses. This minimizes the output size by concatenating adjacent altered fuses and consequently reduces downloading time to the programmer over the slow serial line.

Xpal at work. Using the 4-bit counter example, Xpal takes the just-mentioned Part output and Figure 7's Paddle definition of a 22V10 device (22 inputs, 10

"versatile" outputs) and generates a fuse map. (Readers interested in a sample Paddle description should contact Mark Kahrs at the address at the end of this article.)

This definition omits the polarity and architectural fuses. The name following the Array declaration is unique (in this case, it is `and_or`). The notation `Complement+` means that an input line to the `and_or` array is complemented with the idempotent line first. The outputs take the form `node:maximum`, where `Node` is the node number, and `Maximum` is the maximum number of inputs to the OR gate. The fuse number of a line can be set to an arbitrary value with an `=`; otherwise the line number is set to the maximum value from the previous line plus the fuse number of the first line of the previous node.

In the 4-bit counter example, consider the fuses for output pin 14, which has three inputs, pins 6, 5, and 14. The line number of the output pin is 123, the maximum number of inputs of the fuse array is 22 (the number of inputs) \times 2

```

NS22V10=AM22V10=22V10 {
  package "DIP2403"
  macrocell typical {
    enable : 100,
    clocked : 300,
    invert : 200,
    reset : 400,
    set : 500,
    clock : 600
  }
  declare {
    internal {
      reset { 25 }
      set { 26 }
    }
    external {
      inputs { 1..11, 13 }
      clock { 1 }
      clocked enabled macrocell
        typical { 14..23 }
      ground { 12 }
      supply { 24 }
    }
  }
  array and/or {
    inputs complement+ {
      1, 23,
      2, 22,
      .
      .
      .
      10, 14,
      11, 13
    }
    outputs {
      25:1, % asynchronous
        reset
      123:1,
      23:8,
      .
      .
      .
      114:1,
      14:8,
      26=131 % synchronous
        preset
    }
  }
}

```

Figure 7. Paddle definition of a 22V10.

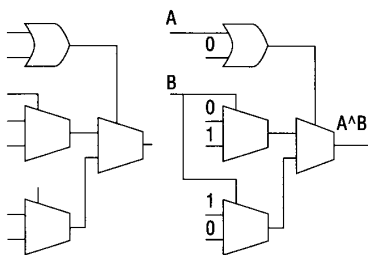


Figure 8. Primitive Actel cell and XOR gate.

	Load	a2	x0	x1	x2	Value mask
0	0	0	0	1	1	
	0	0	0	1	1	
1	0	1	1	0	0	
	1	1	1	0	1	
2	1	0	0	0	0	
	1	1	0	0	1	
3	1	0	0	0	0	
	1	0	1	0	1	

Figure 9. Internal binary value and mask fields.

(including complementary inputs). The initial output fuse line will be $123 \times 22 = 2,706$. The first term has a mask of 3 and a value field of 3. For bit 0, the selected input line would belong to pin 14 (a feedback term). The input line for pin 14 is 19, therefore the fuse number will be $2,706 + 19 = 2,725$. This fuse will be set to the "connect" value (that is, don't blow the fuse), and the next bit in the value field (for pin 5) will be checked. Pin 6 is omitted since its mask bit is zero.

Actel FPGAs. Actel ACT 1 FPGAs contain up to 547 primitive elements consisting essentially of three 2:1 multiplexers. After configuration, antifuses that can only be programmed once interconnect these primitive cells. By comparison, the Xilinx FPGAs are SRAM based (and therefore reprogrammable), and more expensive. Figure 8 shows one possible configuration of a primitive

cell to form an exclusive OR gate.

Rather than allow customers to deal directly with the primitive cells and internal routing, Actel provides a netlist interface to a library of preconfigured cells such as gates, flip-flops, multiplexers, and buffers. Actel proprietary software automatically places and routes the cells, and for each library primitive, chooses an appropriate configuration of the actual primitive cells (often among several possibilities).

This last point is very important for two reasons. First, placement of the cell and the routing of nodes to it influence the choice of configuration for a primitive cell. Second, the load presented by an input to a library cell is nontrivially related to the number of connections one would use to create such a cell out of a primitive.

For example, while a two-input XOR built from the primitive cell needs more than one connection to at least one of the inputs, the Actel library XOR function specifies only a single load for each input. So at least we get something in return for not having complete control of the device.

Tree building. Act's first task is building data structures and handling cross-references. For example, the following four lines define the x2 bit of the counter in the toggle flip-flop example:

```
.o x2 load a2 x0 x1 x2
3:19 12:13 17:19
.o x2@t clk
1:1
```

These two definitions are linked by their mention of the same output variable; the parser duly notes this and assigns the second set of product terms to the clock input of the toggle flip-flop called for by the @t operator. Figure 9 shows the internal binary value and masks fields.

Factoring. From an array of product

terms it is a simple matter to discover which if any inputs are common among them. This set is the AND of all of the masks in the array.

While computing the composite mask, the code also determines whether both senses of a common input appear in the array. This result is the difference between factoring the array as an AND or as a MUX of the selected input. At present, Act factors out non-multiplexed inputs first.

If no input is common to all product terms, Act chooses the most frequently referenced one to partition the terms into two sets. Note that the general case result $OR(d, MUX(s, a, b))$ is equivalent to the if-then-else DAGs (directed acyclic graphs) described by Karplus.⁴⁵

The factoring process proceeds in divide-and-conquer steps. After an input x is selected for factoring, Act divides the array into multiples of x and \bar{x} . Before recursing to factor the resulting subarray(s), Act checks them to see if one is the complement of the other. If so, Act returns XOR, and only the smaller subarray needs to be factored.

The XOR test is exhaustive yet inexpensive. When two expressions are not mutual complements, they are typically discovered almost immediately, usually during comparison of the input sets (the ORs of all product term masks). Complete walks of the state space only occur on successful tests. Discovering all the XORs in a 20-bit ripple-carry counter takes about one second on a 25-MHz R3000.

Balancing. Once the procedure encounters a solitary term or an array of terms that it can't factor, it builds the AND/OR tree. If necessary, Act builds OR trees from subtrees of four or less elements, and AND trees from subtrees of three or less elements. This ensures that the root of any subtree can be represented as at most one Actel cell.

Act builds AND trees in a canonical form. The Actel library defines, for ex-

ample, four variations of three-input AND gates: AND3, AND3A, AND3B, and AND3C. These gates have 0, 1, 2, and 3 inverted inputs (bubbles, in the language of logic design). AND trees are built with bubbles starting from the left.

Height balancing and canonicalization are both properties of the trees as they are built. Once constructed, trees are never rewritten.

Common subexpressions can be detected during the tree construction process by checking to see if a node of the right type with its parameter list already exists before creating a new node. If it exists, Act returns a pointer to the existing node. Canonicalization and luck serve to identify the vast majority of common subexpressions. Very complex expressions reduce this effect.

Before matching, Act again scans each tree from bottom up to compute an approximate fan-out for each node. Every time a node is visited, Act increments its fan-out count and decrements the fan-outs of each of its child nodes. If a node's fan-out exceeds some threshold, Act will insert a buffer tree between it and its uses. Fan-out counts are approximate because Actel library gate inputs do not always have unit loads, and gate assignment follows the fan-out computation.

Matching. Act uses tree matching^{8,9} to select macros from the Actel gate library. Like Dagon,¹⁰ Act defines the target architecture in Twig,¹¹ a language for generating code generators. However, unlike Dagon, Act automatically generates the Twig specification from a list of one-line definitions of Actel library modules. For example, the AX1 primitive is

```
AX1 = (!A & B) ^ C
```

compiled into

```
.o AX1 A B C
2:7 4:6 5:5
```

from which Act constructs

```
AX1 = xor(C,and(not(A),B))
```

and a simple Awk program turns into a Twig pattern/action definition

```
e: xor(and(not(e),e),e)
{TOPDOWN;}
={
  tDO(%3$);
  tDO(%2$);
  tDO(%1$);
  func($$, "AX1",3,"Y","A",
    "B","C");
};
```

Briefly, the Topdown directive tells the pattern matcher to begin at the top; Do follows the children, and Func emits the call to the Actel cell. This attempt to express Twig patterns in the same way that incoming trees will be built has been moderately successful. A stronger sense of canonicalization might help here.

Once a tree is matched, Act walks the tree top-down, maintaining a stack of identifiers and Actel gate descriptors. After all the leaf nodes have been traversed, at least the proper number of tree identifiers will have been either pushed or left on the stack. If netlist output is desired, Act prints the results and pops the stack, leaving the tree identifier of the matched subtree on the stack. If human-readable output is desired, Act pushes the tree identifier on the stack without popping the tree identifiers, and a recursive function prints a list representation of it when the matcher returns.

Act at work. In the earlier 4-bit counter example (not using toggle flip-flops), Act constructs the following tree from the input terms:

```
x0 = mux(load,!x0,a0)
x1 = mux(load,xor(x1,x0),a1)
x2 = mux(load,xor(x2,and
```

```
(x1,x0)),a2)
x3 = mux(load,xor(x3,and
(x2,x1,x0)),a3)
```

Note the MUXes and XORs discovered from the sum of products input form. Including I/O buffers and flip-flops, the complete description for this circuit in an Actel device is

```
clk = CLKBUF(clk)
x0 = BIBUF(DFM4(load,INV
(x0),a0,clk,0),1)
load = INBUF(load)
a0 = INBUF(a0)
x1 = BIBUF(DFM4(load,XOR
(x1,x0),a1,clk,0),1)
a1 = INBUF(a1)
x2 = BIBUF(DFM4(load,XOR(x2,
AND2(x0,x1)),a2,clk,0),1)
a2 = INBUF(a2)
x3 = BIBUF(DFM4(load,XOR
(x3,AND3(x0,x1,x2)),a3,
clk,0),1)
a3 = INBUF(a3)
```

The DFM4 macro cell includes both a multiplexer and a D flip-flop. Twig's dynamic programming algorithm naturally finds such combinations. At present, however, the ordering of the costs do not reflect the discovery of common subexpressions or the subsequent fan-out computation.

The Act2 library defines classes of combinational and sequential modules that can be combined into a single module. Modifying the Twig specification to take advantage of this property took a couple of hours, and the resulting Act2 program worked the first time.

XII. XIL and XNF generate the Xilinx Netlist Format from symbolic product terms. In both cases, the majority of work goes into matching product terms of flip-flops and buffering with that of the Xilinx Netlist Format. In addition, XIL does Act-like tree factorization before output.

Table 1. Act fitter results.

Circuit	Inputs	Outputs	Terms	Act2	Mis_pga	Act(blocks)	Act(buffers)
Duke2	22	29	87	164	198	349	50
Bw	5	28	87	64	80	100	32
Clip	9	5	167	62	62	113	13
Rd84	8	4	256	63	72	74	11
5xp1	7	10	75	48	53	44	15
Misex1	8	7	32	24	—	26	14
Misex2	25	18	29	42	—	65	42
Apex6	135	99	452	302	—	408	372
Apex7	49	37	176	108	—	121	106
Rot	135	107	691	427	—	414	264

After implementing a variant of Woo's visible edge algorithm¹² for packing four-input lookup tables, we discovered that Xilinx tools for the 4000 series parts did quite an excellent job of bin packing. They worked within the additional constraints of the arrangement of four- and three-input lookup tables in these parts, so XIL internal trees are output as binary trees.

Discussion and future work

Xpal and Paddle are a success. Users can define new devices in the time it takes to decipher the manufacturer's fuse maps and type them in. The hardest task is extracting the fuse maps from the manufacturers in the first place, since manufacturers regard fuse maps as proprietary knowledge. Paddle's main fault is that it lacks any knowledge about the internal semantics of the devices it describes. It assumes that the designer knows how to map a design onto the internal fuse numbers and leaves this to earlier programs in the design chain. It also accepts node numbers numerically instead of symbolically. This is partly historical and partly related to the limited knowledge Xpal requires from the programs upstream. In any event, it has proven to be of little practical consequence.

Before the introduction of Part (and Npart), designers were often confront-

ed with designs that didn't fit into the given devices. Part demonstrated that a min-cut algorithm could execute effective distribution of equations to parts. Part's extension to Npart lets designers combine numerous equation files into one file and thereby minimize board space.

Most current work in logic synthesis is based on multilevel optimization.¹³ The two-level logic form used by Act suffers by comparison as the logic width increases. Table 1 lists the results when the described fitters are run using the LGSynth89 benchmarks.¹⁴


Compared with the number of gates reported in Ercolani and DeMicheli,¹⁵ Act can be a factor of two larger. At least two things are at work here. 1) Input to Act is not processed by a multilevel optimizer such as MIS.¹³ 2) Act doesn't try very hard to find common subexpressions. Any common expressions it finds are usually localized (for example, an expression for a clock enable common to an entire register) and any more effort put into sharing subexpressions might adversely affect routing. Note that Ercolani and DeMicheli does not consider the effect of loading or fan-out.

Additionally, Act runs extremely fast. All of the examples here have been run in less than one second on an R3000-based multiprocessor. For most of the

applications encountered thus far in our research environment (mostly random or interconnection logic), the width of typical logic is narrow, and the output code from Act appears to match well with the Actel architecture. Unfortunately, the well-known Logic Synthesis benchmarks are wide (for example, the alu4 circuit has 131 product terms!) and do not show our tools at their best.

WE'VE FOUND TWO-LEVEL LOGIC to be convenient in this application, and we've been pleasantly surprised to achieve such utility from such a well-worn tool set. However, the use of hierarchy in the source language would reduce the pressure on the code-generation problem as well as allow access to macro libraries supplied by FPGA manufacturers.

Most current structured hardware description languages like Verilog and VHDL emphasize control structure instead of expressions involving fields of bits. Until such languages include constructs for bit-field manipulation, interesting bit manipulations will be difficult.

The FPGA industry is in constant flux. Compared with table lookup architectures like Xilinx and two-level PAL derivatives such as MACH, the Actel architecture is a relatively fine-grained one. These architectures all seem well in hand and are subjects of ongoing work. However, at least one upcoming technology, Quicklogic, has an even finer grain and will further point out the need for hierarchy and effective code generation. 

References

1. S. Trimberger, "Guest Editor's Introduction: Field-Programmable Gate Arrays," *IEEE Design & Test of Computers*, Vol. 9, No. 3, Sept. 1992, pp. 3-5.
2. *JEDEC Fuse Standard #3A*, Joint Electron

- Device Engineering Council, Washington, D.C.
3. E.J. McCluskey, "Minimization of Boolean Functions," *Bell System Tech. J.*, Vol. 35, 1956, pp. 1417-1444.
 4. K. Karplus, "Xmap: A Technology Mapper for Table Lookup Field-Programmable Gate Arrays," *Proc. IEEE/ACM 28th Design Automation Conf.*, Association for Computing Machinery, New York, 1991, pp. 240-243.
 5. K. Karplus, "Amap: A Technology Mapper for Selector-Based Field-Programmable Gate Arrays," *Proc. IEEE/ACM 28th Design Automation Conf.*, ACM, 1991, pp. 244-247.
 6. B.W. Kernighan and S. Lin, "An Efficient Heuristic Procedure for Partitioning Graphs," *Bell Sys. Tech. J.*, Vol. 49, No. 2, 1970, pp. 291-308.
 7. B.W. Kernighan and S. Lin, "Method of Minimizing the Interconnection Cost of Linked Objects," US Patent 3,617,714, Nov. 2, 1971.
 8. D.C. Schmidt and G. Metze, "Modular Replacement of Combinational Switching Circuits," *IEEE Trans. Computers*, Jan. 1975, pp. 29-47.
 9. D.C. Schmidt, "Gate for Gate Modular Replacement of Combinational Switching Circuits," *Proc. Ninth Ann. Design Automation Workshop*, ACM, June 1972, pp. 331-340.
 10. K. Keutzer, "DAGON: Technology Binding and Local Optimization by DAG Matching," *Proc. IEEE/ACM 24th Design Automation Conf.*, ACM, June 1987, pp. 331-340.
 11. A.V. Aho, M. Ganapathi, and S.W.K. Tjiang, "Code Generation Using Tree Matching and Dynamic Programming," *ACM Trans. Programming Languages and Systems*, Vol. 11, No. 4, Oct. 1989, pp. 491-516.
 12. N. Woo, "A Heuristic Method for FPGA Technology Mapping Based on the Edge Visibility," *Proc. 28th Design Automation Conf.*, ACM, 1991, pp. 248-251.
 13. R. Brayton et al., "MIS: A Multiple-Level Logic Optimization System," *IEEE Trans. Computer-Aided Design of Integrated Cir-*

cuits and Systems, Nov. 1987, pp. 1062-1081.

14. *Logic Synthesis and Optimization Benchmarks User Guide*, S. Yang, ed., MCNC (Microelectronics Center of North Carolina), Jan. 1991, available by any-

mous ftp from mcnc.org.

15. S. Ercolani and G. DeMicheli, "Technology Mapping for Electrically Programmable Gate Arrays," *Proc. IEEE/ACM 28th Design Automation Conf.*, ACM, June 1991, pp. 234-239.



Mark Kahrs, Robert C. Restrck III, and Bart N. Locanthi

Mark Kahrs, an assistant professor of electrical and computer engineering at Rutgers University, teaches courses in digital signal processing, audio engineering, and computer system architecture. He has worked for Xerox PARC, the Center for Computer Research in Music and Acoustics at Stanford University, the Institute de Recherche et Coordination Acoustique Musique in Paris, and AT&T Bell Laboratories. Kahrs holds a PhD in computer science from the University of Rochester, New York.

Robert C. Restrck III is a member of the technical staff in the Computing Sciences Research Center and a Fellow at AT&T Bell Laboratories. His research includes networking and hardware design. Restrck received a PhD in electrical engineering from the University of Michigan.

Bart N. Locanthi helped to found Silicon Graphics, Inc., in time off from his job as a member of the technical staff at AT&T Bell Laboratories. His interests include computer architecture, information display, programming languages, computer-aided design, and technologies for telecommuting. He designed the first asynchronous windowing terminal as well as a follow-on that became the basis for Plan 9 from Bell Labs. Currently, he is most active in the emerging field of personal communications systems. Locanthi holds a PhD from the California Institute of Technology, Pasadena.

Direct questions concerning this article to Mark Kahrs, Rutgers University, Dept. of Electrical and Computer Engineering, College of Engineering, PO Box 909, Piscataway, NJ 08855-0909; kahrs@winlab.rutgers.edu.