Improved Handling of the Decoding Operation in the Presto Compiler

by

Wenyan Dong

Submitted to the Department of Electrical Engineering and Computer Science

in partial fulfillment of the requirements for the degrees of

Bachelor of Science in Electrical Engineering and Computer Science

and

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2004

Author ....

Department of Electrical Engineering and Computer Science

July 20, 2004

Certified by ....

Karen Pieper

VI-A Company Thesis Supervisor

Thesis Supervisor

Certified by ....

Arvind, Johnson Professor

M.I.T. Thesis Supervisor

Thesis Supervisor

Accepted by ....

Arthur C. Smith

Chairman, Department Committee on Graduate Theses

Improved Handling of the Decoding Operation in the Presto Compiler

by

Wenyan Dong

Submitted to the Department of Electrical Engineering and Computer Science
on July 20, 2004, in partial fulfillment of the
requirements for the degrees of
Bachelor of Science in Electrical Engineering and Computer Science
and
Master of Engineering in Electrical Engineering and Computer Science

Abstract

This thesis presents a research project on decoder related optimizations in HDL designs. The goal of the reasearch is to improve design synthesis quality-of-result, mainly in terms of area; this involves sharing decoders driven by related inputs, and map decoders using fewer number of boolean gates. Algorithms presented in this thesis were implemented in the Presto HDL compiler. A series of tests were conducted using real-world HDL designs in order to determine how effective these optimizations are.

Thesis Supervisor: Karen Pieper
Title: VI-A Company Thesis Supervisor

Thesis Supervisor: Arvind, Johnson Professor
Title: M.I.T. Thesis Supervisor

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Overview

In any VLSI industry, electronic design automation (EDA) tools that synthesize custom silicon from abstract hardware description languages (HDL) are used to automate and accelerate the intricate design process. Over the years, EDA industry has provided tremendous value to chip designers with its automated software to create and validate electronic designs, and has helped raise the level of abstraction in the design process. The unrelenting drive to produce smaller and more complex electronic components and microprocessors has fueled the reliance of chip designers upon EDA tools, and more importantly, the need for ultra-powerful tools that achieve good quality of results (QoR) in terms of area, timing, power, etc.

HDLs describe the architecture and behavior of discrete electronic systems. A typical HDL supports a mixed-level description in which gate and netlist constructs are used with functional descriptions. This mixed-level capability enables a designer to describe system architectures at a very high level of abstraction and then incrementally refine a design's detailed gate-level implementation. HDL descriptions plays an important role in modern design methodology for three reasons [10]:

1. A design written as an HDL description can be simulated immediately. Design simulation at this higher level, before implementation at the gate level, allows

designers to evaluate architectural and design decisions.

2. HDL compiler can automatically convert an HDL description to a gate-level implementation in a target technology. This eliminates the former gate-level design bottleneck.

3. HDL descriptions provide technology-indenpendent documentation of a design and its funtionality. An HDL description is easier to read and understand than a netlist or a schematic description. A technology-independent documentation can be reused to generate the design in a different technology, without having to translate from the original technology.

As a result, HDL compilers has become an important part of the EDA tools.

Synthesis is a process that generates a technology-dependent, gate-level design for an IC design that has been defined using HDL languages. The Design Compiler (DC) comprises tools that synthesize HDL designs. The Presto Compiler – the default HDL compiler in DC – translates Verilog or VHDL descriptions to the Synopsys internal design format, as illustrated in Figure 1-1. Presto first profiles the HDL files to obtain a full intermediate level (IL) description of the design, and performs various architectural optimizations at the same time. During the second step, Presto takes the IL description and translate it into the *ilnet* representation in which every operational component of the design and their interconnections are identified. In the final stage of the compiler, designs are stored using the Synopsys database (DB) format. The DB files are structural representations of the design using cells from the Generic Technology (GTECH) library, which is a technology-indepedent library. The DB files are then passed to the Design Compiler, which can then further optimize the design and map it to a spectific ASIC technology library. The entire synthesis flow [11] is depicted in Figure 1-2.

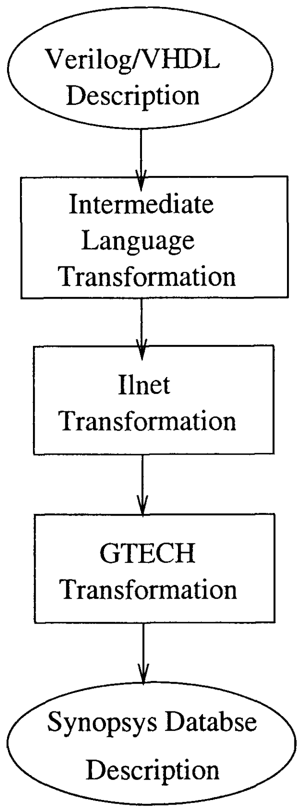


Figure 1-1: Presto Compiler Flow

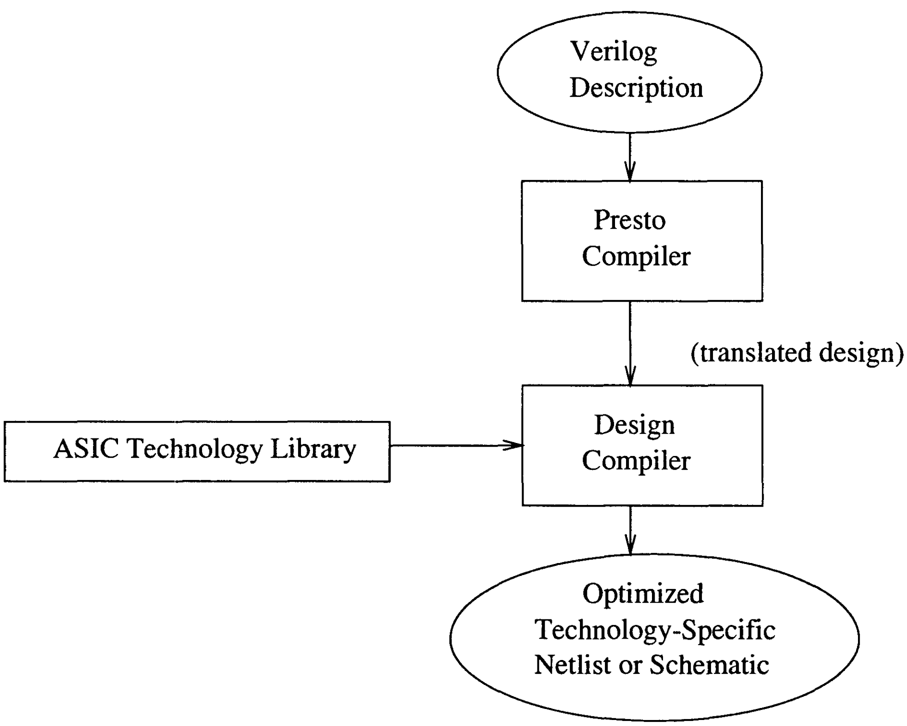
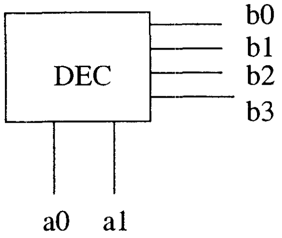


Figure 1-2: Design Synthesis Flow



a0	a1	b0	b1	b2	b3
0	0	1	0	0	0
1	0	0	1	0	0
0	1	0	0	1	0
1	1	0	0	0	1

Figure 1-3: A 2-to-4 Decoder

1.2 Motivation and Goals

A decoder can be thought of as a converter from binary to a one-hot encoding. A 2-to-4 decoder is shown in Figure 1-3, in which A are the inputs, and B are the outputs. In HDL designs, decoders are generated for case variables, as shown in Example 1-1, and left hand side (LHS) variable array subscripts, as shown in Example 1-2.

Ex 1-1: A 3-to-8 decoder is inferred from the case variable *sel*

```
module decoder (sel, res);
   output [7:0] res;
   input [2:0] sel;
   reg [7:0] res;
   always @ (sel) begin
      case (sel)
         3'b000:  res = 8'b00000001;
         3'b001:  res = 8'b00000010;
         3'b010:  res = 8'b00000100;
         3'b011:  res = 8'b00001000;
         3'b100:  res = 8'b00010000;
         3'b101:  res = 8'b00100000;
         3'b110:  res = 8'b01000000;
         3'b111:  res = 8'b10000000;
      endcase;
   end
endmodule
```

Ex 1-2: A 3-to-8 decoder is inferred from the LHS variable array subscript *sel*

```
module decoder (sel, res);
   output [7:0] res;
   input [2:0] sel;
   reg [7:0] res;
   always @ (sel or res) begin
      res[sel] = 1;
   end
endmodule
```

For an n-bit input decoder, there are 2^n output bits, and thus the width of a decoder grows exponentially with its number of input bits. Hence decoders can contribute a significant portion of the design area, and the new generation of HDL compilers need to efficiently synthesize this particular hardware component, in order to achieve better QoR. In the Presto Compiler, decoders are synthesized in two steps: first decoders are generated from case variables and LHS variable subscripts during the ilnet stage; Presto then maps the generated decoders using GTECH library cells during the GTECH tranformtion stage. As a result, decoder related optimizations in the Presto Compiler can also be divided into two steps: 1) generate fewer decoders by sharing those that are driven by common inputs, referred to as *decoder sharing*, and 2) map decoders using fewer number of GTECH cells, referred to as *decoder mapping*. Currently, there is little decoder-related optimizations in the Presto compiler, and this research project will implement algorithms for both decoder sharing and decoder mapping. By generating fewer decoders and reducing their sizes, it is hoped that the number of cells in the final design, and more importantly, the design area would improve thereafter.

1.3 Thesis Organization

Chapter Two provides a detailed description of the decoder sharing problem, considers alternative methods, and discusses in detail the actual algorithms implemented within the Presto framework. Chapter Three provides a detailed description of the decoder mapping problem, and gives mapping algorithms for various kinds of decoders. Chapter Four demonstrates the results obtained using these optimization techniques and discusses possible future improvements.

Chapter 2

Decoder Sharing

2.1 Problem Description

2.1.1 Sharing Decoders and MUXes

For HDL example:

Ex 2-1:
```
input x;
input[1:0] b
output[1:0] a
b[x] = a[x];
```

Currently, Presto detects the variable array subscript on the left hand side and blindly generates a decoder. It also detects the variable array subscript on the right hand side and blindly generates a multiplexer. A multiplexer (MUX) is a digital circuit in which the correct data bit is selected according to the address input. A two-input

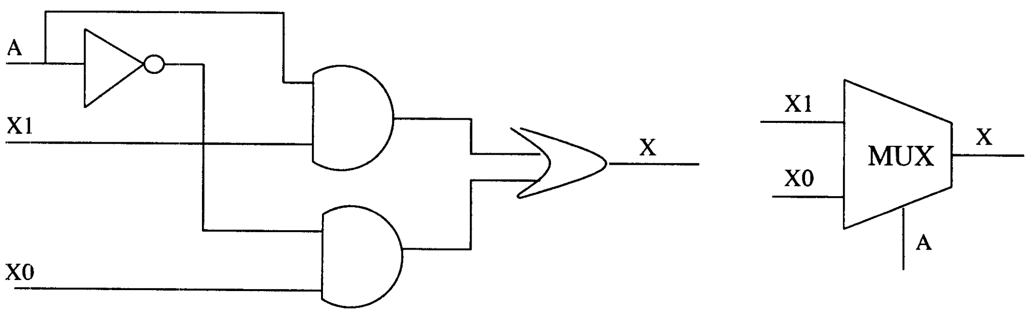


Figure 2-1: A two-input Multiplexer

MUX is shown in Figure 2-1. Input A is the addressing input that controls which of the two data inputs, X0 or X1, will be transmitted to the output.

Before actually synthesizing any cells, the Presto compiler rewrites the above example to the following:

```
decoder_0 = DECODER(x);
mux_0 = MUX(a, x);
case (1'b1)
   decoder_0[0]:  b[0] = mux_0;
endcase
case (1'b1)
   decoder_0[1]:  b[1] = mux_0;
endcase
```

The case construct in the rewritten statements is then synthesized into a *select* cell, which can be thought of as a series of multiplexers with individual control inputs and data inputs. Figure 2-2 shows the selector constructed in this particular example, in which decoder outputs are used as control inputs and the mux output is used as the data input. Figure 2-3 shows the complete circuit for this particular example.

A better approach is to calculate directly, based on the decoder output alone, the addresses for both arrays, which are then inserted directly into the optimized case statements. The optimized rewrite is as follows:

```
decoder_0 = DECODER(x);
case (1'b1)
   decoder_0[0]:  b[0] = a[0];
endcase
case (1'b1)
   decoder_0[1]:  b[1] = a[1];
endcase
```

In the optimized case statements, the addresses for both arrays are transformed into constants based on the value of `DECODER(x)`, without the need for a mux from $a[x]$. When synthesizing the above case statements, the wires for individual bits from both arrays are routed directly to the selector. Figure 2-5 illustrates the improved circuit for this particular example.

Since it is long and tedious to write out each case statement, a shorthand is used

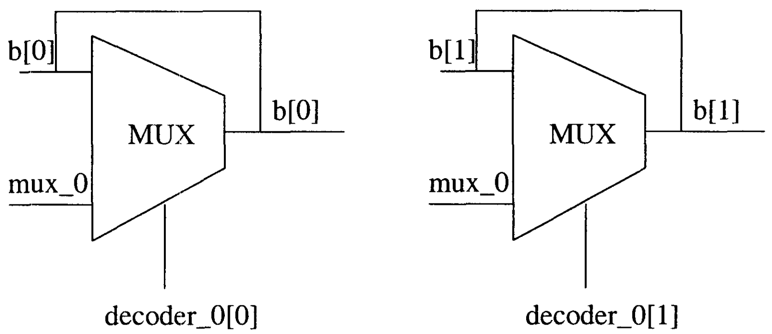


Figure 2-2: A two-input selector

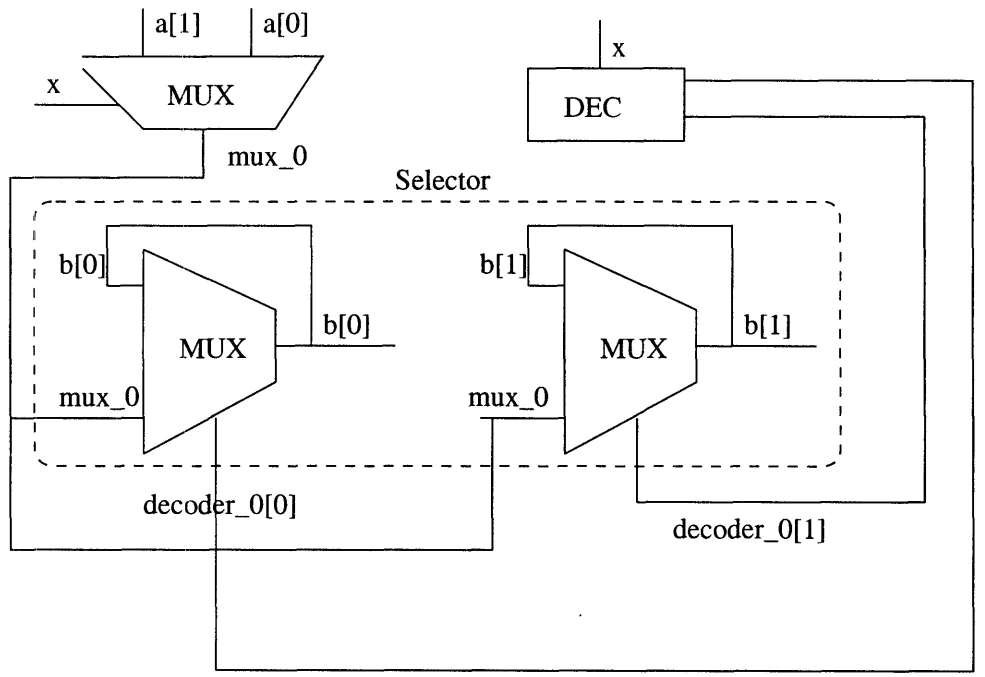


Figure 2-3: The circuit diagram for b[x]=a[x]

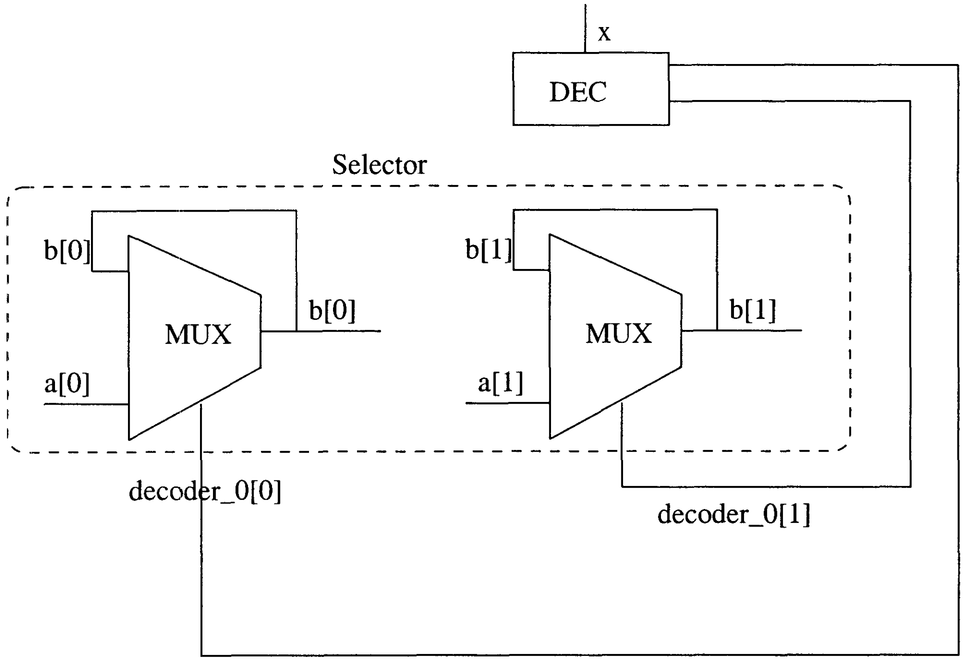


Figure 2-4: Improved circuit for b[x]=a[x] without the mux for a[x]

to replace the case construct, and is shown as follows:

```
decoder_0 = DECODER(x);
b[decoder_0] = a[decoder_0]; -- case construct shorthand
```

Similar simplification is employed throughout the paper.

The example above can be easily generalized to statements in the form of `b[f(x)] = a[g(x)]`, in which indices on both sides are arithmetic expressions that contain a common variable x. Currently, Presto generates a decoder from $f(x)$, and a mux from $g(x)$. A better approach is to identify the common variable x in both indices, and generate a decoder from it. Once the value of x is determined, Presto can calculate both array addresses from $f(x)$ and $g(x)$ respectively, and transform variable subscripts into constants before generating any actual cells. In this case, besides saving one mux, arithmetic cells that would have been generated for $f(x)$ and $g(x)$ are also eliminated. The optimized rewritten statements are as follows:

```
decoder_0 = DECODER(x);
b[f(decoder_0)] = a[g(decoder_0)]; -- case construct shorthand
```

Note that the rewritten case construct includes only cases in which $f(decoder\_0)$ and

$g(decoder\_0)$ are within the range of array a and b. Decoder output bits that lead to out-of-bound index values for either a or b will not be connected to the selector, and are thus left unconnected. This is applied when synthesizing every variable array references.

2.1.2 Sharing Decoders

In the previous section, decoder outputs are shared between left and right hand side array references to eliminate unnecessary muxes. In this section, decoders with common inputs are shared to avoid generating unnecessary decoders.

For the statement `a[x][x][x+y] = 1'b1`, Presto blindly generates one decoder for each dimension – `DECODER(x)`, `DECODER(x)`, `DECODER(x+y)`. Optimally, Presto should be able to detect common variables in all dimensions, and determine a minimum set of decoders that is needed to completely decode the address in every dimension. In this particular example, the optimal set contains only two decoders – one for variable x and one for variable y. `DECODER(x+y)` is considered unnecessary since the value of $x + y$ can be determined from those of x and y. The optimized rewrite is as follows:

```
decoder_0 = DECODER(x);
decoder_1 = DECODER(y);
case (1'b1)
   decoder_0[0] && decoder_1[0]:  a[0][0][0] = 1;
endcase
case (1'b1)
   decoder_0[1] && decoder_1[0]:  a[1][1][1] = 1;
endcase
...
case (1'b1)
   decoder_0[0] && decoder_1[1]:  a[0][0][1] = 1;
endcase
...
```

Note in this case, the control inputs for the synthesized selector are the AND of two output bits from both decoders.

The number of decoders can be further saved by expanding the problem globally, and identify repetitive decoders among different HDL statements. In the following example:

```
input x;
output [1:0] a;
output [1:0] b;
a[x] = 1;
b[x] = 1;
```

Presto currently generates two decoders, one for each assignment. However, it is clear that the two decoders are exactly the same, and thus one of them can be eliminated from the design.

The benefit of decoder sharing is the greatest for designs that are similar to the following:

```
Ex 2-2:     input [4:0] x;
            input [1023:0] a;
            output [1023:0] b;
            int i;
            for (i=0; i<1024; i++) {
               a[x+i] = b[i];
            }
```

Similar snippets of code can be found in signal processing or memory related designs, which often contain large arrays with variable indices inserted in a *for* loop. The ability to extract the variable and its corresponding decoder outside the loop would result in huge savings in number of decoders. Using the example above, Presto currently generates one decoder and one adder for each iteration $(x, x + 1, x + 2...)$, for a total of 1024 decoders and 1023 adders. The original rewrite is as follows:

```
for (i=0; i<1024; i++) {
   decoder_0 = DECODER(x+i);
   a[decoder_0] = b[i]; -- case construct shorthand
}
```

Optimally, variable x should be recognized as a common variable, and the decoder generated from x is then shared among all iterations. The optimized design would save a total of 1023 decoders and 1023 adders. The optimized rewrite is as follows:

```
decoder_0 = DECODER(x);
for (i=0; i<1024; i++) {
   a[decoder_0+i] = b[i]; -- case construct shorthand
}
```

2.1.3 Combining Two Problems

If we treat the right hand side array references the same as those on the left hand side, problems in section 2.1.1 are then essentially the same as those in section 2.1.2. In Example 2-1, if we use a decoder for subscript x in array a on the right hand side instead of a mux, there would be two decoders with the same input x in the design. Hence, one of them would be eliminated, and the same optimized circuit would be synthesized as in Figure 2-4.

Therefore, the problem can be summarized as selecting a minimum set of decoders from which every variable array address can be determined. The number of decoding cells is reduced by reusing decoders with common inputs. This appears to be very similar to that of common subexprssion elimination (CSE) – a well known standard optimization implemented in most traditional compilers. The purpose of CSE is to reduce the runtime of a program through avoiding the repetition of the same computation. Indeed, after rewriting the assignment `b[x] = a[x]` to the following:

```
t1 = decode(x);
t2 = decode(x);
b[t2] = a[t1]; -- case construct shorthand
```

Then *decode(x)* is trivially sharble, and naturally, only one decoder will be generated. However, a direct application of CSE only eliminates decoders with the exact same input, since decode operations with different arguments would not be considered as identical computations. Statement `b[f(x)] = a[g(x)]` becomes

```
t1 = decode(f(x));
t2 = decode(g(x));
b[t2] = a[t1];
```

In this case, *decode*$(f(x))$ and *decode*$(g(x))$ are different and unsharable computations. Hence, a direct application of CSE would still generate unnecssary decoders in the end.

2.2 Implementation Consideration

To effectively reduce the number of decoders, the compiler must first determine which decoders *may* be generated, then choose which ones actually *will* be generated.

2.2.1 Identify Possible Decoders

To produce a list of possible decoders, we need to extract variables in array subscripts and compare them. During the IL stage of the Presto compiler, various architectural rewrites are applied to an HDL design, which may prevent us from correctly identifying all possible decoders. The following design

```
Ex 2-3:  input [1:0] x;
        input [1:0] y;
        a[2x+y+3] = c[2x+y];
        b[x][y] = 1;
```

would be rewritten as

```
        t0 = 2x + y;                       -- CSE
        t1 (32-bits) = t0 (2-bits);        -- type conversion
        a[t1 + 3] = c[t0];
        b[x][y] = 1;
```

First, expression $2x + y$ is a common expression in the design and is renamed to $t0$ as the result of traditional CSE optimization. A constant integer 3, whose type is 32-bits, is contained in the subscript of array a. Hence $t0$, whose type is 2-bits, is renamed to $t1$ when its type is converted to 32-bits. In this case, CSE optimization masqueraded both variables x and y as potential decoding inputs. Moreover, $t0$ and $t1$ are rendered different as a result of type conversion, while the fact is that they both possess the same value, albeit different types, and thus both should be deemed identical.

In order to avoid such reverse effects, one option is to implement decoder sharing during the IL stage of the Presto compiler, but previous to all IL rewrites that may modify any subscript expression. The pros of this option include

1. During the IL stage, an HDL design is represented as a directed graph whose nodes are operations to be performed and their operands. Analyzing any expression, including subscripts, requires graph-traversing, which can be done with relative ease.

2. By analyzing original subscript expressions, a more accurate list of potential decoders can be produced.

The cons of this option include

1. During the IL stage, loop iteration variables are regarded as ordinary variables instead of constants. As a part of an index expression, loop iteration variables would be included as potential decoding inputs, and thus may result in unneccessary decoders. In Example 2-2, if the loop iteration variable i is treated the same as x, i and $x+i$ would be included in the list of potential decoding inputs. However, since i assumes a constant value in each iteration, the list should include x only.

2. Expressions can only be compared by their names, instead of their values. As a result, lexically identical but semantically different variables maybe considered equivalent, or lexically different but semantically identical variables maybe considered different. This can be solved using partial redundancy [3] and value numbering [1] – two classes of algorithms already implemented as a part of CSE in the Presto compiler. However, if decoder sharing is to be implemented before all relevant IL rewrites, including CSE, then these two particlar algorithms would need to be applied twice – during the decoder sharing phase and later again during the CSE phase.

Another option is to implement decoder sharing during the ilnet stage of the Presto compiler, after all the IL rewrites. The ilnet representation for each expression

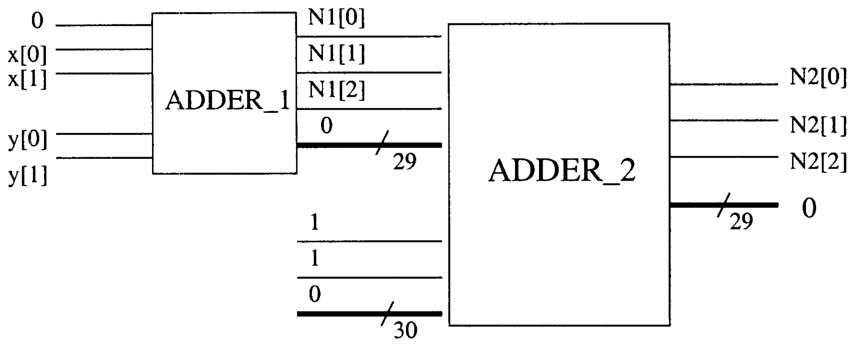


Figure 2-5: Ilnet representation of $t1+3$ $(2x+y+3)$ in example 2-3

consists of nets driven by a certain opearational cell, whose inputs are nets driven by some other operational cells. Once we obtain the ilnet representation for each index, we can extract variables from them by traversing the cell interconnections backwards. Figure 2-5 depicts the ilnet representation of the index expression $t1+3$ in Example 2-3.

The pros of this option include

1. Loop iteration variables are represented by constant nets. The compiler is then able to differentiate them from ordinary variables, and exclude them from the list of potential decoding inputs.

2. The netlist representation is a correct criterion in comparing two expressions. Semantically different expressions are represented with different nets, and semantically identical expressions are represented with the same nets.

The cons of this option include

1. Since each net represents one bit of the expression and most expressions contain multiple bits, multiple nets need to be compared in determining the equivalence of two expressions. This may result in a slight increase in compile time.

2. Arithmetic optimizations may eliminate certain cells, which may in turn disguise potential decoding inputs. In the following example:

```
input [3:0] x;
a[x>>2] = b[x<<2];
```

instead of representing both indices with left shift and right shift cells respectively, the ilnet form for the left index consists of simple nets {0 0 x[3] x[2]} with no driving cells, and the ilnet form for the right index consists of simple nets {x[1] x[0] 0 0} with no driving cells. In this case, it is difficult to extract variable x from both indices. Note that Presto would still produce a correct design, albeit an inferior one, by generating two decoders instead of one.

Comparing the two options described above, the latter provides a better solution. The inability of the first option to identify loop iteration variables may result in unnecessary decoders, which opposites the goal of decoder sharing. Excluding partial redundancy (PR) and value numbering (VN), the first option may have a shorter compile time. This however, would lead to incorrect comparisons and possibly incorrect designs. In terms of correctly comparing two expressions, ilnet comparison provides a much easier and quicker solution than PS and VN.

2.2.2 Choose a Minimum Set of Decoders

After a list of potential decoders is produced, we need to choose from the list, a minimum set of decoders that actually will be generated. Often, the appropriate choice for part of the design may not be appropriate for the whole design. In Example 2-3, to completely decode the address for array a, the compiler may generate two decoders – Decoder(x) and Decoder(y), or just one decoder – Decoder($2x + y$). The same can be said for array c. Thus, when considering the first statement alone, a decoder of $2x + y$ appears to the better option. However, to completely decode the address of array b in the second statement, the compiler must generate two decoders from x and y. Therefore, the opposite is the better option in terms of the whole design. Once Decoder(x) and Decoder(y) are generated, both can be shared among all three arrays to completely decode their addresses.

2.3 Detailed Implementation

2.3.1 Analyze Index Nets

When analyzing each index expression in its ilnet form, the compiler traverses the cell interconnections backwards, identifies the operations that the cells represent, and extracts different combinations of subexpression nets from which potential decoders may be generated.

Before describing the algorithm, several points need to be brought to attention:

1. *eliminate most significant zero nets*

   When decoding index expressions, the compiler is concerned with their values instead of their types. Most significant zeros do not impact the value of an expression and thus those nets can be eliminated without any effects. In other words, any nets in the form of {0 ... 0, n ... n} can be reduced to {n ... n}, with n represnting any non-zero net. In the ilnet representation of the expression $2x + y + 3$, depicted in Figure 2-5, 29 bits of zero nets in the output of the second adder can be eliminated. The same can be said for 30 bits of zero nets in the output of the first adder.

2. *discard constant operands*

   If a variable expression $f(x, y...)$ can be decomposed into – using any arithmetic operation – another variable expression $g(x, y...)$ and a constant, only g needs to be considered as a potential decoding input. The value of f can be easily calculated from Decoder(g). In Figure 2-5, the bottom operand driving the second adder consists of only constant nets, thus nets $t2$ would be dropped from consideration as a possible decoding input. All possible decoding inputs would come from nets $t1$ and its sub-nets.

3. *discard least significant zero nets*

   For nets in the form of {n ... n, 0 ... 0}, only their sub-parts {n ... n} need to be considered as a potential decoding input. Denote z as the number of

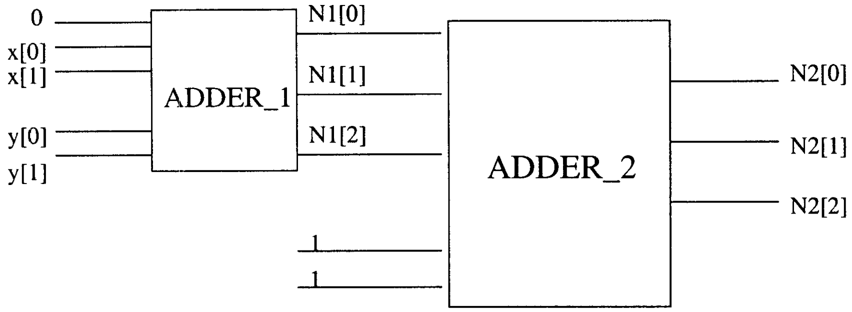


Figure 2-6: Modified ilnet representation of $2x + y + 3$

least significant zero nets. Once the value of the sub-net is determined from the decoder, denoted as v, the value of the entire nets can be easily calculated as $v \cdot 2^z$. In Figure 2-5, $2x$ is represented as {x[1], x[0], 0}, which contains one least significant zero net. Only the sub-nets {x[1], x[0]} will be considered as a possible decoding input.

Note the difference between elimination and discard. When nets are eliminated, not only are they prevented from being part of any decoding input, they are also erased from the original ilnet representation. When nets are discarded, they are prevented from being part of any decoding input, but are kept in the ilnet representation. Since the ilnet representation will later be used to calculate the value of the index once the actual decoders are generated, constant operands and least significant zeros must be kept to ensure the correct calculation. The most significant zeros should be erased from the ilnet representation since they do not impact the value of the index, and if not removed, the compiler would have to evaluate them again.

Function *AnalyzeNets* summarizes the steps for analyzing each index expression. The function input is the ilnet representation of each index with most significant zero nets in the output eliminated (Algorithm 2). The function *AnalyzeNets* calls function *AddToGraph*, which adds the extracted variables to an decoding input graph (Section 2.3.3). Applying the algorithm to the ilnet representation in Figure 2-5, variable nets {N1[0], N1[1], N1[2]}, {x[0], x[1]}, and {y[0], y[1]} would be extracted and added to the graph. By eliminating most significant zero nets, the ilnet representation itself is modified, which is depicted in Figure 2-6.

Algorithm 1: AnalyzeNets(N, parent_nodeID, dir)

```
input  : N – nets to be analyzed with most their significant zeros eliminated
parent_nodeID – the ID of the node in the decoding input graph that
contains the parent of N
dir – whether N is the left or the right child of its parent. '1' represents left
child, and '2' represents right child

cell ← DrivingCell(N);  //cell with N as its output
N' ← DiscardLSBZeros(N);  //N' is N without least significant zeros
if cell == NULL then
    AddToGraph(N', parent_nodeID, dir);
    return ;

left ← InputA(cell);  //the left operand of the cell
left ← EliminateMSBZeros(left);  //eliminate most significant zeros
right ← InputB(cell);  //the right operand of the cell
right ← EliminateMSBZeros(right);

if left is not constant && right is not constant then
    nodeID = AddToGraph (N', parent_nodeID, dir);
    AnalyzeNets(left, nodeID, 1);
    AnalyzeNets(right, nodeID, 2);
else if left is constant then
    //all potential decoding inputs would come from the right operand
    AnalyzeNets(right, parent_nodeID, dir) ;
else
    //all potential decoding inputs would come from the left operand
    AnalyzeNets(left, parent_nodeID, dir) ;
```

Algorithm 2: AnalyzeIndex(index)

```
N ← ilnet representation of the index;

if N is constant then
    return ;

N ← EliminateMSBZeros(N);
AnalyzeNets(N, NULL, NULL);
```

2.3.2 Compare Decoding Input Nets

When adding nets produced from the previous analysis to the decoding input graph, they are compared against all existing nets in the graph in order to prevent duplicates. There are three cases in which two nets are considered the same:

1. The new is exactly the same as the old, e.g.

   old: {x[2], x[1], x[0]}
   new: {x[2], x[1], x[0]}

2. The new is a subset of the old, e.g.

old: {x[2], x[1], x[0]}	old: {x[2], x[1], x[0]}	old: {x[2], x[1], x[0]}
new: {x[2], x[1]}	new: {x[1], x[0]}	new: {x[1]}

   In this case, the values of the two are related through shifting operations. Assume the old contains *old_bits* number of bits, and the new contains *new_bits* number of bits. If the new is exactly the same as $old[t : v]$ with $0 < t < v < (old\_bits - 1)$, then the value of the new can be calculated from the old by left shifting $old\_bits - 1 - v$, and then right shifting $old\_bits - new\_bits$.

   Using the middle example above, the old has 3 bits, and the new has 2 bits. The new is exactly the same as $old[1 : 0]$, thus the left shift number is $2 - 1 = 1$, and the right shift number is $3 - 2 = 1$. The value of the new can be calculated as $(old << 1) >> 1$.

3. The old is a subset of the new, e.g.

   old: {x[2], x[1]}
   new: {x[2], x[1], x[0]}

   This is the exact opposite of the previous case. The value of the old can be calculated from the new, instead of the other way around.

2.3.3 Construct Decoding Input Graph

The decoding input graph is essentially a forest of binary trees that merges all possible decoding inputs. Each node in the graph contains nets that are inputs for potential

Algorithm 3: CompareNets(old, new, & shift)

```
input  : old – existing nets in the decoding input graph
new – nets that need to be added to the decoding input graph
output: 1) An integer indicating which case does the comparison belong to
2) Shift numbers relating the two nets

old_bits ← NumberOfBits(old);
new_bits ← NumberOfBits(new);
case old_bits == new_bits
    Compare the old and the new bit by bit;
    if all bits are the same then
        return 1
    else
        return 0;
case old_bits > new_bits
    n ← new[new_bits - 1]  //n is the most significant bit in the new nets
    //compare n with each bit in the old nets, starting from the most
      significant bit, until one that is the same as n is found in the old nets
    for i ← old_bits-1; i ≥ 0 && n!=old[i]; i- - do {};
    if i < 0 then
        return 0;
    //get the subset from the old that has the same number of bits as the new,
      starting from the ith bit in the old
    sub_old ← old [i : i-(new_bits-1)];
    if CompareNets(sub_old, new) == 1 then
        (shift→left) ← old_bits - 1 - i;
        (shift→right) ← old_bits - new_bits;
        return 2;
    else
        return 0;
case old_bits < new_bits
    if CompareNets(new, old) == 2 then
        return 3;
    else
        return 0;
```

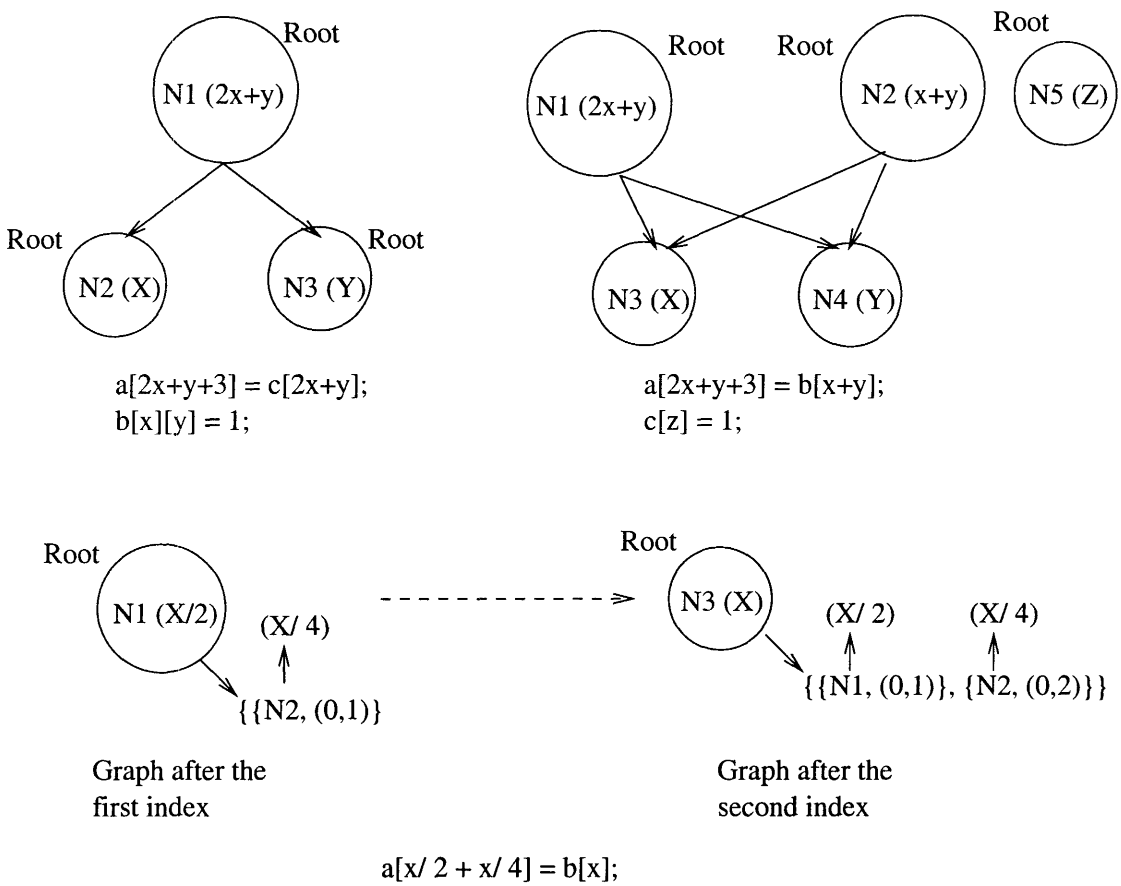


Figure 2-7: Several Decoding Input Graphs.

decoders. Using algorithm *CompareNets*, if the new nets are exactly the same as those in an existing node m, then no nodes will be created and the new nets will not be added to the graph. The ID of node m is returned. If the new nets are a subset of those in node m, then the new nets and the shift numbers relating their values will be added to m's *related-nets* list. No new node is constructed and the ID of node m is returned. If the nets in node m are a subset of the new nets, then a node n containing the new nets is constructed. Node n replaces node m, and takes over m's *related-nets* list. The shift numbers for each element in list is re-calculated. Old nets in node m are also added to node n's *related-nets* list. The ID of node n is returned. The nets in *each* index that do not have a parent are defined as root nets. The node that contains the root nets, whether in itself or its related-nets list, would be marked as "root". Figure 2-7 illustrates several decoding input graphs constructed for different designs.

Algorithm 4: AddToGraph(new, parent_nodeID, dir)

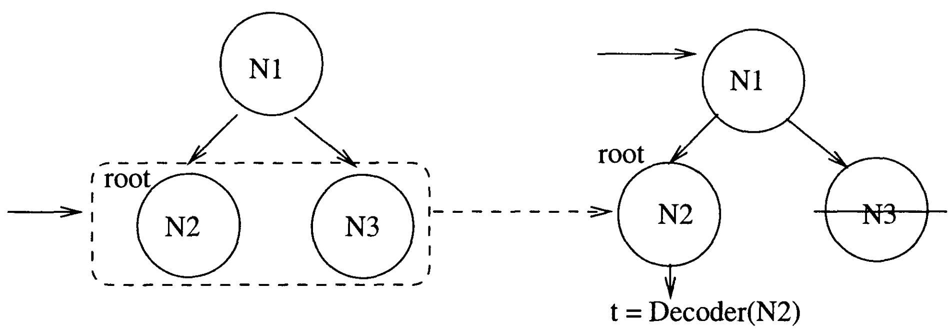
```
input  : new – nets that need to be added to the decoding input graph
parent_nodeID – the ID of the node containing the parent of the new nets
dir – indicates if the new is the left child or the right child of its parent
output: Decoding Input Graph G

repeat
    m ← get a existing node from the graph;
    compare = CompareNets(m.nets, new, & shift);
until compare > 0 OR no more nodes in the graph;
switch compare do
    case 0
        //The new nets do no exist in the graph
        nodeID ← ConstructNode(new);
    case 1
        //The new nets are exactly the same as the existing nets
        nodeID ← m;
    case 2
        //The new nets are a subset of the existing nets
        nodeID ← m;
        m.related_nets.append(new, shift);
    case 3
        //The existing nets are a subset of the new nets
        nodeID ← ConstructNode(new);
        foreach element n in m.related_nets do
            compare = CompareNets(new, n.nets, & new_shift);
            n.shift_number = new_shift;
        nodeID.related_nets.append(m.nets, shift);
if parent_nodeID == NULL then
    mark nodeID as root;
else
    if dir == 1 then
        set the left child of parent_nodeID to node_ID;
    else if dir == 2 then
        set the right child of parent_nodeID to the node_ID;
return node_ID;
```

2.3.4 Generate Decoders from the Graph

Once the decoding input graph has been built, we need to pick a minimum set of nodes from which actual decoders would be built and values of all array subscripts can be determined. This is achieved by traversing the graph reverse topologically. Each interior node in the graph has two children, and those two children must be visited before their parent(s). A node is crossed out when it is eliminated from becoming an actual decoder. Both children will be crossed out except in the following situations:

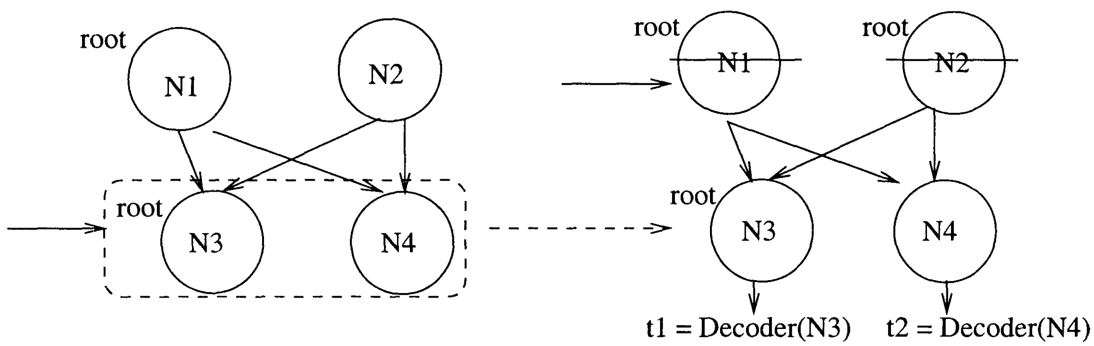
1. One of children is marked as a root, and none of their parents is marked as a root. A decoder is generated from the root child if it has not been crossed out, while the other child node is crossed out.



If an uncrossed node is marked as a root (e.g. node $N2$), it represents a variable expression whose value must be to be decoded, but has yet to be decoded. In this case, a decoder must be generated from the node. Since none of the parent nodes is marked as a root, each of them must be a child of some root node X. If an actual decoder were generated from the non-root child node (e.g. node $N3$), then the values of all parent nodes (e.g. node $N1$) would be naturally determined. However, at least another decoder must be generated in order to determine the value root X. From the standpoint of the root node X, choosing the non-root child node would lead to more than one decoders, which is a worse choice than just generating one decoder from the node X itself. Therefore, the non-root child in this case is crossed out.

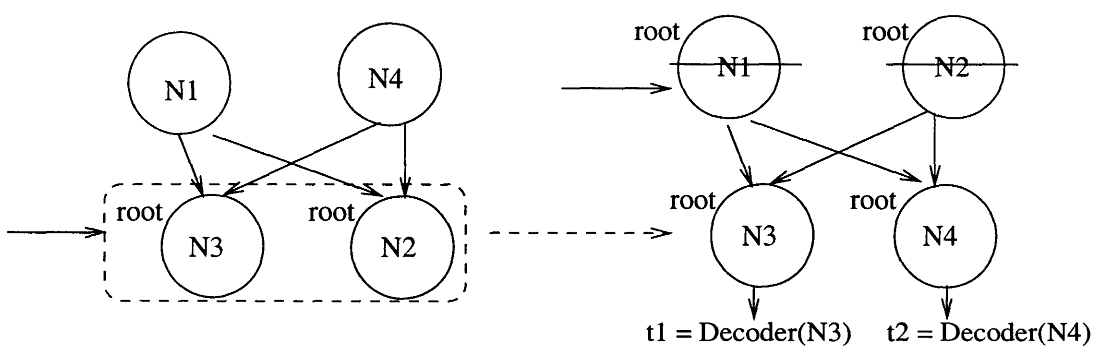
2. One of the children is marked as a root, and at least one of the parents is

also marked as a root. Actual decoders would be generated from both children assuming they not crossed out. As a result, the values of all parent nodes can now be determined based solely on those two decoders. All parent nodes will be marked as "root" and crossed out.

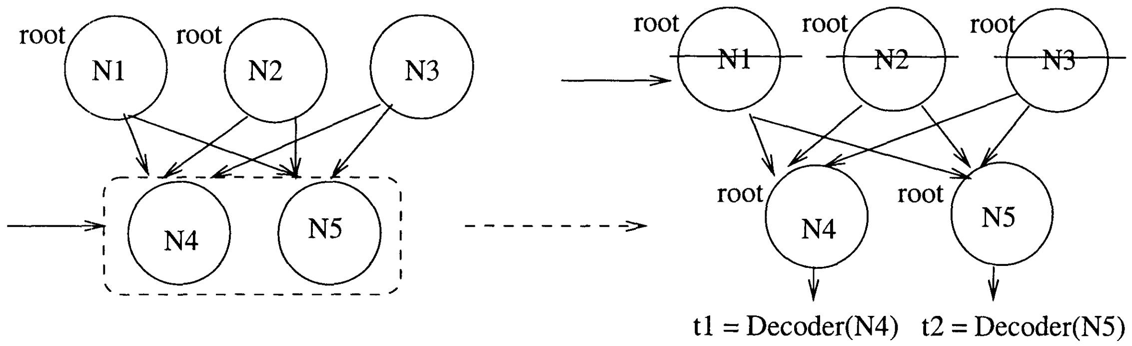


For root node $N3$, a decoder must be generated from it if the node is uncrossed or no decoders are needed if it is crossed. Either way, we do not have any choices when it comes to this particular node. To decode the value of node $N1$, two choices exist – 1) generate one more decoder from $N1$ or 2) generate one more decoder from $N4$. If node $N1$ were chosen, it onyl determines the value of one parent node, namely the node itself. If node $N4$ were chosen, the value of all parent nodes would then be naturaly determined. Hence all parent nodes can be crossed out and marked as roots.

3. Both children are marked as roots. The same actions are taken as in case 2. It does not matter whether any parent is marked as a root or not.



4. Neither child is marked as a root, but at least two of their parents are marked as roots. Same actions are taken as in case 2.



In this case, there are at least two parent nodes that must be decoded, and thus at least two more decoders are needed. By choosing two child nodes, all parent nodes – including those marked as roots – would be naturally determined.

Figure 2-8 illustrates the decoders generated for two simple decoding input graphs from Figure 2-7. Figure 2-9 illustrates the decoder generating process for a relatively complicated graph.

Note that every time a decoder is generated from a node n, its value is also assigned a temporary variable. The attribute of the nets contained in node n is then set to the temporary variable. For nets in node n's related-nets list, their attributes are set to an expression involving the temporary variable and the corresponding shift number. Using the bottom graph in Figure 2-7 as an example, a decoder is generated from nets N3. Assume a temporary variable $t1$ is assigned the value of the decoder. Then the attribute of nets N3 is set to $t1$. The attribute of nets N1 is set to $t1 >> 1$, and the attribute of nets N2 is set to $t1 >> 2$.

2.3.5 Rewrite Array References

Once the actual decoders are generated, each index is rewritten using the values of those decoders. The rewritten expression is deduced by inversely traverse the ilnet representation of each index, and replace it with net attribute expressions.

Again using the bottom example in Figure 2-7, the rewrite is as follows:

```
a[x/2 + x/4] = b[x];  ==>  t1 = Decoder(x);
                           a[t1>>1 + t1>>2] = b[t1];
```

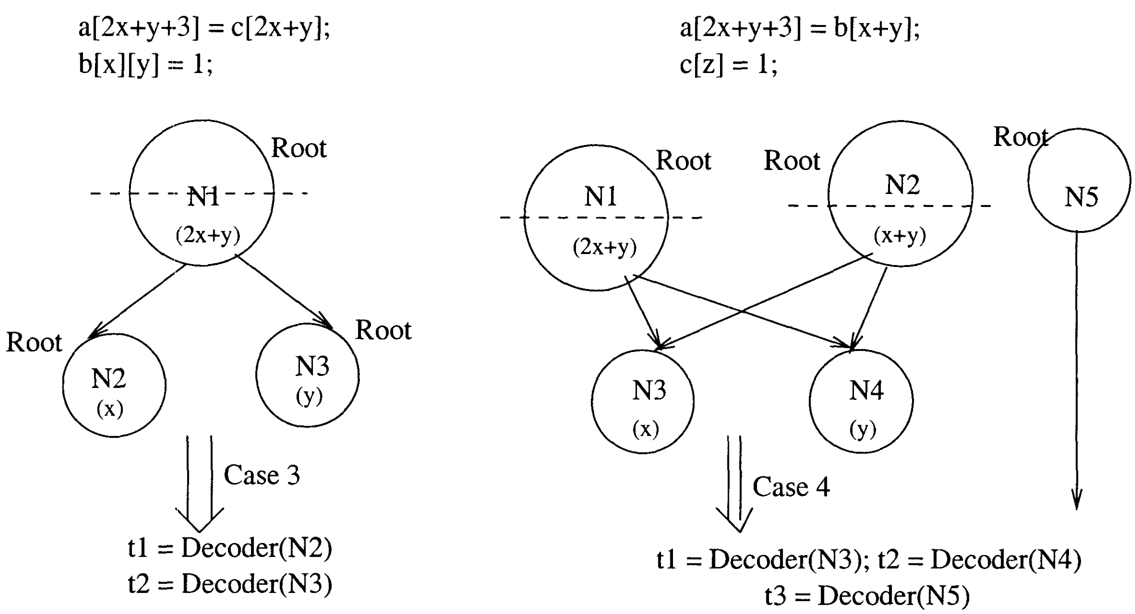


Figure 2-8: Decoders Generated for simple decoding input graphs.

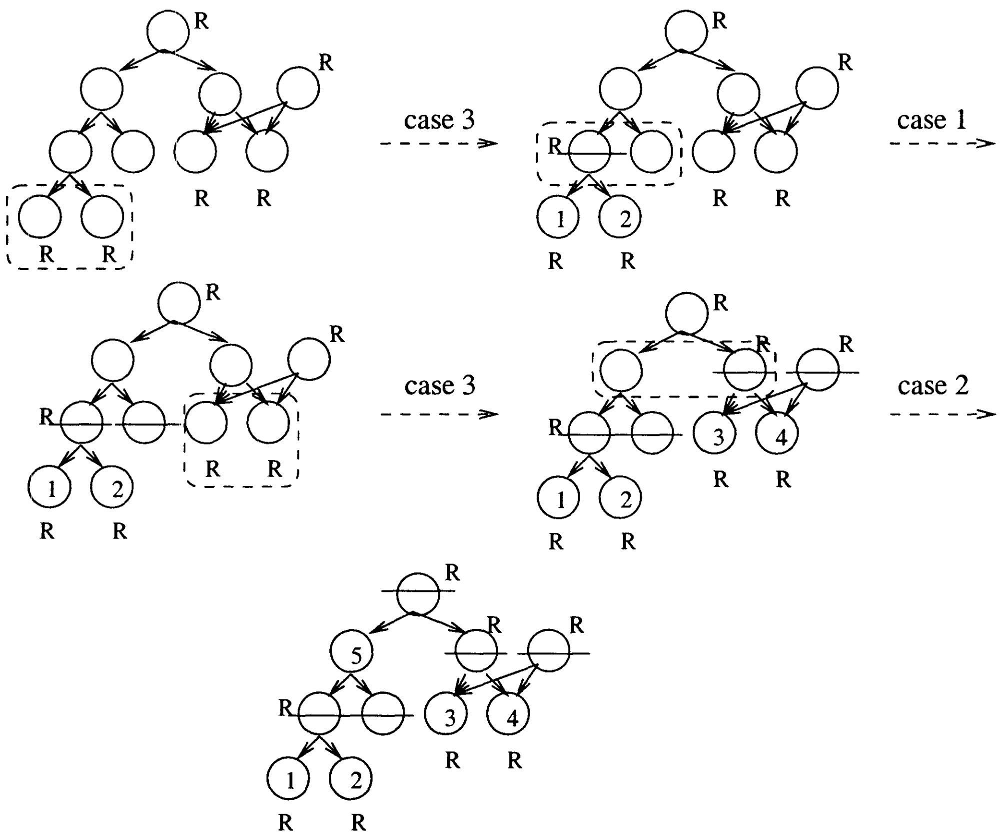


Figure 2-9: Decoder Generating Process for a relatively complicated graph. The first one represents the original decoding input graph. Each graph shows the transformation made after traversing the nodes in the dashed rectangle. Decoders are generated from numbered nodes.

Algorithm 5: RewriteIndex(N)

input : **N** – nets of an index expression

```
if the attribute of N is set then
    expr ← the attribute of N;
    if there are least significant zeros in N then
        expr ← expr ·2^number_of_least_significant_zeros;
    return expr;
else if N is constant then
    expr ← the value of N;
    return expr;
else
    cell ← the driving cell of N;
    op ← the arithmetic operation performed by the cell;
    lhs_net ← InputA(cell);
    rhs_net ← InputB(cell);
    lhs ← RewriteIndex(lhs_net);
    rhs ← RewriteIndex(rhs_net);
    expr ← {lhs op rhs};
    return expr;
```

Using the left example in Figure 2-4, the rewrite is as follows:

```
a[2x+y+3] = c[2x+y]; ==> t1 = Decoder(x); t2 = Decoder(y);
b[x][y] = 1;             a[2t1+t2+3] = c[2t1+t2];
                         b[t1][t2] = 1;
```

After array indices are rewritten using temporary decoder variables, the statements are then rewritten to case constructs described in section 2.1.

Chapter 3

Decoder Mapping

3.1 Problem Description

During the GTECH transformation phase in the Presto compiler, all cells – including decoders – are mapped using cells from the GTECH library. The GTECH library is a generic technology-independent library, which consists of common logic elements [12], which includes:

1. *Boolean Logic Cells*: Buffers, Inverters, AND, OR, NAND, NOR, XOR, XNOR
2. Adder, MUX
3. Flip-Flop, Latch

The goal of the decoder mapping is to map decoders using the fewest number of GTECH cells possible. In doing so, we hope to reduce the size of each decoder.

Define $\overrightarrow{a}$ as the value represented by a binary string $a[n-1:0]$. A decoder with an n-bit input is a combinational circuit specified as follows:

Input: $x[n-1:0] \in \{0,1\}^n$
Output: $y[2^n-1:0] \in \{0,1\}^{2^n}$
Functionality: $y[i] = 1 \Leftrightarrow \overrightarrow{x} = i$

Consider a decoder with whose input x has 3 bits. When $x = 101$, output y equals 00100000. Individual bits in the output can be expressed as a simple boolean function of the decoder input. For a 3-bit decoder,

```
{n7, n6, n5, n4, n3, n2, n1, n0} = DECODER(x[2], x[1], x[0]);
```

each output net is expressed as follows:

```
n7 = AND(x[2], x[1], x[0])
n6 = AND(x[2], x[1], ~x[0])
n5 = AND(x[2], ~x[1], x[0])
     ...
n0 = AND(~x[2], ~x[1], ~x[0])
```

The simplest way to map a decoder is to build a separate circuit for every output bit. This decoder circuit would contain 1) $n \cdot 2^{n-1}$ number of inverters, since each input bit is inverted in half of the 2^n outputs, and 2)$(n-1) \cdot 2^n$ number of AND gates, since each output needs $n-1$ AND gates. Thus, the total cost of this particular brute force design is $\Theta(n \cdot 2^n)$. Assuming the AND gate has a longer delay time than the inverter, the delay of the circuit is $\Theta(t_{pd}(INV) + n \cdot t_{pd}(AND))$. This mapping algorithm is currently employed in the Presto compiler. Intuitively, it is wasteful in terms of the number of GTECH cells used. For example, boolean expressions for every two output bits differ by only one bit, namely $x[0]$, and the corresponding AND tree share all but one single operand. In the current appraoch however, the AND of $x[n-1:1]$ is computed twice.

One solution is to share the same sub-trees among different output bits. Various different sub-trees are available to be shared – an AND-tree of input $x[n-1:1]$ can be shared between outputs $y[2^n-1]$ and $y[2^n-2]$; an AND tree of input $x[n-1:2]$ can be shared between outputs $y[2^n-2]$ and $y[2^n-3]$; an AND tree of input $x[n-1]$ and $x[n-2]$ can be shared among output bits $y[2^n-1:2^{n-2}]$. The choice of common sub-AND trees to be actually shared is crucial to the number of AND gates used.

It seems that the problem can be solved by traditional CSE algorithms. However, several distinct features of the problem renders CSE unsuitable:

1. Expressions for decoder outputs are known in advance. Once the decoder input is determined, so are the expressions for all decoder output bits. More importantly, all available common sub-expressions (sub AND-trees) are natu-

rally determined. Traditional CSE algorithms do not know the expressions in advance, and expressions are much more general than those in decoder circuits. Hence, most CSE algorithms leave the responsibity of finding possible common subexpressions to the vicissitudes of the parser [11], which often leads to obscured common expressions due to the order they appear in the expression tree.

2. To effectively reduce the number of AND gates, the compiler need to pick an optimal set of common sub-AND trees to be shared among different output bits. In traditional CSE algorithms, finding optimal common subexpressions is NP-complete [4], and fast heuristics are deployed. When applied directly on decoding output expressions, many of them yield poor results. For example, if the longest common subexpressions are chosen, then AND-trees from input bit $x[n-1:1]$ would be shared between every two consecutive output bits. This particular set of common sub-AND trees would turn out to be sub-optimal. In fact, due to the symmetric structure possessed by the decoder output expressions, optimal sub-AND trees can be determined in advance, which is described in detail in the following sections.

It is often that some of the decoder outputs do not drive any cells and are left unconnected. Therefore, decoders can be divided into two main categories:

1. *Full Decoder*: A decoder without any unconnected outputs.

2. *Partial Decoder*: A decoder with some unconnected outputs. Output bits are left unconnected because they represent an out-of-bound array index. In the following example:

```
Ex 3-1:  Partial Decoder.  ('U' = Unconnected Nets) :
    input[3:0] x;
    output[9:0] a; ⟹ {U,U,U,U,U,U,n9,n8,...,n0} = Decoder(X)
    a[x] = 1;
```

output bits from 10 to 15 is left unconnected because the range of array a is from 0 to 9.

Partial Decoders can be further categorized according to the following two criteria:

1. *whether the unconnected outputs are continuous:* A *continuous partial decoder* is one whose connected outputs are separate from unconnected outputs. No connected outputs are sandwiched between two unconnected outputs. Otherwise, the decoder is defined as a *discontinuous partial decoder.* The following illustrates a few examples.

```
Ex 3-2:  Continuous Partial Decoders
    {n7, n6, U, U, U, n2, n1, n0} = Decoder(X)
    {U, U, n5, n4, n3, n2, U, U} = Decoder(X)
    {n7, n6, n5, U, n3, n2, n1, n0} = Decoder(X)
Ex 3-3:  Discontinuous Partial Decoder
    {n7, n6, U, n4, n3, U, n1, n0} = Decoder(X)
```

2. *whether the unconnected outputs can be treated as don't cares:* For partial decoders generated from array references, unconnected outputs can be treated as don't cares if the option for dynamic bounds-checking is turned off. In this case, the designer does not care about the behavior of the circuit once the array index is out-of-bound. If the option is turned on, an error should be generated if the index is out-of-bounds. For partial decoders generated from case variables, unconnected outputs are treated as don't cares if they represent cases that are specified as don't cares in the original design.

3.2 Full Decoder

Denote a full decoder with n-input bits as Decoder(n). Let x be its input and y be its output. Decoder(n) can be designed using recursion on n. The circuit of Decoder(1) simply consists of one inverter where $y[0] \leftarrow \mathrm{INV}(x[0])$ and $y[1] \leftarrow x[0]$.

Assume the mappings of decoders with input length less than n are known. Using the divide-and-conquer method, consider a parameter k, where $0 < k < n$. Partition the input bits $x[n-1:0]$ into two parts as follows:

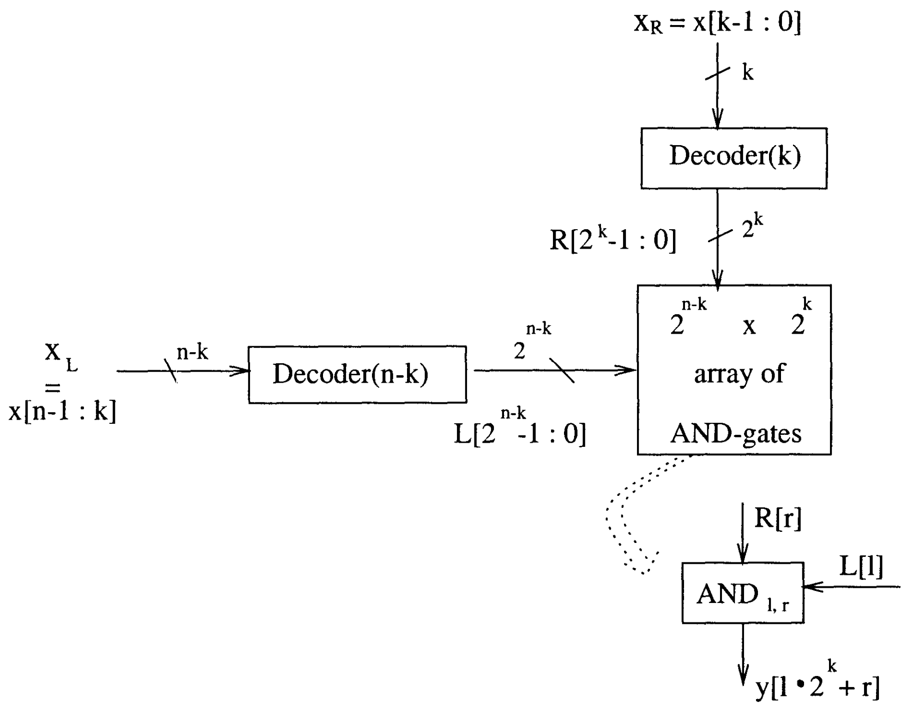


Figure 3-1: A recursive mapping of Decoder(n)

1. The right part, denoted by x_R, including input bits $x[k-1:0]$

2. The left part, denoted by x_L, including input bits $x[n-1,k]$

A recursive mapping of the decoder feeds x_R to Decoder(k). The output of this decoder is denoted by $R[2^k-1:0]$. In a simliar manner, x_L is fed to Decoder($n-k$). The output of this decoder is denoted as $L[2^{n-k}-1:0]$. Both decoder outputs are then fed to a $2^k \times 2^{n-k}$ array of AND gates, in which each AND gate has two inputs from R and L. Denote the AND gate with position (r,l) in the array as $AND_{r,l}$. The inputs of this AND gate are $R[r]$ and $L[l]$. The output of this AND gate is $y[l \cdot 2^k + r]$. Figure 3-1 depicts the recursive implementation.

Let $c(n)$ be the number of gates used in the recursive mapping of Decoder(n). $c(n)$ satisfies the following recurrence equation:

$$c(n) = \begin{cases} 1 & \text{if } n = 1 \\ c(k) + c(n-k) + 2^n & \text{otherwise.} \end{cases}$$

Obviously, $c(n) = \Omega(2^n)$, regardless of k. Let $k = n/2$, or $k = \lfloor n/2 \rfloor$ if n is odd. Assume that n is a power of 2, the recurrence equation is then:

$$\begin{aligned}
c(n) &= 2 \cdot c(n/2) + 2^n \\
&= 4 \cdot c(n/4) + 2^n + 2 \cdot 2^{n/2} \\
&= 8 \cdot c(n/8) + 2^n + 2 \cdot 2^{n/2} + 4 \cdot 2^{n/4} \\
&= n \cdot c(1) + \{2^n + 2 \cdot 2^{n/2} + 4 \cdot 2^{n/4} + ... + n \cdot 2^{n/n}\} \\
&\leq n + \{2^n + 2\log_2 n \cdot 2^{n/2}\} \\
&\leq n + \{2^n(1 + \frac{2\log_2 n}{2^{n/2}})\} \\
&= \Theta(2^n)
\end{aligned} \tag{1}$$

Let d(n), d(INV), and d(AND) be the delay of Decoder(n), Inverter, and AND gate, respectively. $d(n)$ satisfies the following recurrence equation:

$$d(n) = \begin{cases} d(INV) & \text{if } n = 1 \\ max\{d(k), d(n-k)\} + d(AND) & \text{otherwise.} \end{cases}$$

Setting $k = n/2$, it follows that $d(n) = \Theta(\log_2 n \cdot d(AND))$.

The mapping of Decoder(n) described above is asymptotically optimal with respect to both the cost and the delay. A full decoder has 2^n number of outputs, all of which has an unique expression. One new AND gate must be included for each output to differentiate it from all others. Hence the whole circuit must contain at least 2^n number of AND gates. Since $c(n) = \Theta(2^n)$, the recursive mapping is thus asymptotically optimal. In a full decoder, all n input bits must be used to produce an output. Hence a neccessary condition to produce a correct output is that each input bit must appear as a leaf in a binary tree, provided that only 2-input AND gate is used. The height of the tree is $\Omega(\log_2 n)$. Since each node in the tree corresponds to an AND gate, the circuit delay is therefore, $\Omega(\log_2 n \cdot d(AND))$, which is asymptotically the same as $d(n)$.

Not only is $c(n)$ *asymptotically* optimal, it can be proved that it is the *exact* optimal solution as well.

Claim 4.1 For the following recurrence

$$T(n) = \begin{cases} 1 & \text{if } n = 1 \\ min\{T(k), T(n-k)\} + 2^n \quad (0 < k < n) & \text{otherwise.} \end{cases}$$

the minimum is obtained when $k = \lfloor n/2 \rfloor$, and thus $T(n) = T\lfloor n/2 \rfloor + T\lceil n/2 \rceil + 2^n$. To prove this claim, we need to first prove the following:

Claim 4.2 For all n

$$T(n) - T(n-1) \geq T(n-1) - T(n-2) \geq T(n-2) - T(n-3) \geq ... \geq T(2) - T(1)$$

Proof using Induction:

1. **Base Case:**

   $T(3) - T(2) \geq T(2) - T(1) \Longleftrightarrow T(3) \geq 2T(2) - T(1)$

   Since $T(1) = 1$, $T(2) = T(1) + T(1) + 2^2 = 6$ and $T(3) = T(2) + T(1) + 2^3 = 15$, the base case is true.

2. **Assumption:** For all n

   $T(n) - T(n-1) \geq T(n-1) - T(n-2) \geq T(n-2) - T(n-3) \geq ... \geq T(2) - T(1)$

   Rearranging the equation, we have:

   $$T(n) + T(1) \geq T(n-1) + T(2) \geq ... \geq T\lfloor (n+1)/2 \rfloor + T\lceil (n+1)/2 \rceil$$

   which shows that for all $n + 1$:

   $$min\{T(k) + T(n+1-k)\} \text{ must have } k = \lfloor (n+1)/2 \rfloor \text{ for } 0 < k < n+1.$$

3. **Inductive Step:** For $n + 1$, we need to prove

   $$T(n+1) - T(n) \geq T(n) - T(n-1)$$

   $$\Longleftrightarrow T(n+1) \geq 2T(n) - T(n-1)$$

   LHS:

   $$T(n+1) = min\{T(k) + T(n+1-k)\} + 2^{n+1}$$
   $$= T\lfloor (n+1)/2 \rfloor + T\lceil (n+1)/2 \rceil + 2^{n+1}$$

RHS:

$$2T(n)-T(n-1) = 2\{min\{T(k)+T(n-k)\}+2^n\}-\{min\{T(k)+T(n-1-k)\}+2^{n-1}\}$$
$$= 2T\lfloor n/2 \rfloor + 2T\lceil n/2 \rceil + 2^{n+1} - T\lfloor (n-1)/2 \rfloor - T\lceil (n-1)/2 \rceil - 2^{n-1}$$

If $n = 2k$,

$$T(n+1) \geq 2T(n) - T(n-1)$$
$$\iff T(k) + T(k+1) \geq 2T(k) + 2T(k) - T(k-1) - T(k) - 2^{2k-1}$$
$$\iff T(k+1) - T(k) \geq T(k) - T(k-1) - 2^{2k-1}$$

Since $0 < k < n$, we have $T(k+1) - T(k) \geq T(k) - T(k-1)$ based on the inductive assumption. Since $2^{2k-1} > 0$, the above is true.

If $n = 2k+1$,

$$T(n+1) \geq 2T(n) - T(n-1)$$
$$\iff T(k+1) + T(k+1) \geq 2T(k) + 2T(k+1) - T(k) - T(k) - 2^{2k}$$
$$\iff 0 \geq -2^{2k}$$

which is trivially true.

4. **Q.E.D.**

We have proved that for every n,

$$T(n) - T(n-1) \geq T(n-1) - T(n-2) \geq T(n-2) - T(n-3) \geq ... \geq T(2) - T(1)$$

which infers that for every n,

$min\{T(k) + T(n-k)\}$ must have $k = \lfloor n/2 \rfloor$.

Therefore, Claim 4.1 is true, and Decoder(n) uses the exact optimal number of boolean gates.

3.3 Continuous Don't Care Partial Decoders

If the unconnected outputs can be treated as don't cares, expressions for connected outputs in a partial decoder may not require all input bits. Suppose the unconnected outputs in Example 3-1 are don't cares, then output n2 can be expressed as:

```
n2 = AND(~x[2], x[1], ~x[0])
```

instead of

```
n2 = AND(~x[3], ~x[2], x[1], ~x[0]).
```

When $x[3]$ is 0, input x evaluates to 2, and output net $n2$ is rightfully set to 1. If $x[3]$ is 1, then x evaluates to 10, and is out-of-bound. Since the designer does not care what happens in this case, the decoder can output any value. Using the optimal boolean expression, output net $n2$ would be set to 1, and $a[2]$ would be assigned a value of 1. The same can be said for outputs $n3$ to $n7$. For output $n0$, $n1$, $n8$, and $n9$, $x[3]$ cannot be ignored. Otherwise, the design cannot differentiate between output $n0$ and $n8$, and between output $n1$ and $n9$.

Denote a continuous don't care partial decoder with n-input bits as Decoder(n). Let C be the number of connected outputs, C' be the 2's power that is closest to but greater than C, and N be the total number of outputs ($N = 2^n$). If C' is less than N, then less than half the decoder outputs are connected. In this case, $N - C'$ number of unconnected outputs and $n - log_2C'$ number of most significant input bits can be eliminated. In the new and smaller decoder, $C' = N$. In the following example,

Ex 3-4: Partial Decoder with Unnecessary Input and Output bits

```
{n15, n14, ... , n9, U, U, ... , U} = Decoder (x[3:0]);
```

there are only seven connected output bits, $C = 7$ and $C' = 8$. Hence only $log_2C' = 3$ input bits are needed to represent the original decoder. The modified decoder is:

```
{n7, ..., n2, n1, U} = Decoder (x[2:0]);
```

As a result, for every Decoder(n), $C' = N = 2^n$. Since $C' > C$ and C' is the closest 2's power to C, the number of connected outputs must be between $2^{n-1} < C < 2^n$. That is, at least half of the output bits are connected in a continuous don't care decoder. More importantly, due to the continuity of those connected output bits, it is obvious that the least significant $n - 1$ input bits constitute a full decoder.

Figure 3-2 illustrates two examples of 4-bit continuous don't care partial decoders, one of which has outputs $n[10 : 15]$ unconnected, and the other has only $n[0]$ uncon-

	x[3]	x[2]	x[1]	x[0]
n0	0	0	0	0
n1	0	0	0	1
n2	*	0	1	0
n3	*	0	1	1
n4	*	1	0	0
n5	*	1	0	1
n6	*	1	1	0
n7	*	1	1	1
n8	1	0	0	0
n9	1	0	0	1
U	*	*	*	*
⋮				
U	*	*	*	*

(a)

	x[3]	x[2]	x[1]	x[0]
U	*	*	*	*
n1	0	0	0	1
n2	0	0	1	0
⋮				
n8	*	0	0	0
n9	1	0	0	1
n10	1	0	1	0
⋮				
n15	1	1	1	1

(b)

Figure 3-2: Two 4-bit continuous don't care partial decoders. Both input bits x[2:0] constitute a full decoder. In (a), outputs $n[10:15]$ are unconnected, thus the decoder has relatively many unconnected outputs and considered very partial. In (b), only output $n[0]$ is unconnected, thus the decoder has relatively few unconnected output bits and considered close to full.

nected. Let p be the number of connected output pairs that have the same lower $n-1$ input bits. In Figure 3-2(a), $n0$ and $n8$ have the same lower three bits – 000; $n1$ and $n9$ have the same lower three bits – 001. Thus, $p = 2$. In Figure 3-2(b), p is 7.

When the number of connected outputs is close to 2^{n-1}, as in Figure 3-2(a), there are relatively many unconnected outputs, and the decoder is very partial. Intuitively, the first step for mapping such decoder is to build a sub-full decoder from the lower $n-1$ bits, which costs $c(n-1)$. For each pair in p, two AND gates involving the input bit $x[n-1]$ are needed to differentiate one output from the other. The total cost is thus $c(n-1)+2p$. In Figure 3-2(a), a full decoder is built from $x[2:0]$ using the mapping algorithm described in the previous section. Denote y as its output. Then two more AND gates are needed to differentiate $n0 = AND(x[3], y[0])$ from $n8 = AND(\sim x[3], y[0])$, and two more AND gates are needed to differentiate $n1 = AND(x[3], y[1])$ from $n9 = AND(x[3], \sim y[1])$.

When the number of connected outputs is close to 2^n, as in Figure 3-2(b), there are few unconnected outputs, and the decoder is close to full. Intuitively, the first step for mapping such decoder is to build a n-bit full decoder, followed by eliminating unnecessary gates built in the process. Since expressions are unique for each output in the full decoder(n), at least one AND gate is built exclusively for each unconnected output and thus can be munched. There are $2^{n-1} - p$ number of unconnected outputs, thus the same number of gates can be munched in the end. The total cost is $c(n) - (2^{n-1} - p)$. It is possible that more gates can be munched, but the worst mapping cost is obtained by assuming the fewest number of munched gates. In Figure 3-2(b), a 4-bit full decoder is first built from $x[3:0]$. Since the AND gate with inputs $(\sim x[1:0])$ and $(\sim x[3:2])$ appears exclusively in unconnected output $n0$, it would be munched.

Let $s(n)$ be the number of gates used for Decoder(n). Of the two methods described above, we choose the one with the fewer gates, and obtain

$$s(n) = min\{c(n-1) + 2p, c(n) - (2^{n-1} - p)\}$$

To investigate the quality of $s(n)$, we first identify a lower bound on the number of gates needed for Decoder(n) – denoted as $t(n)$. Note $c(n)$ denotes the cost of a n-bit full decoder.

Claim 4.3: A n-bit continuous partial don't care decoder requires at least $c(n-1)+p$ number of AND gates: $t(n) \geq c(n-1) + p$

Proof Using Induction on p:

1. *Base Case*: $p = 1$

   In this case, the decoder has one more output than a $(n-1)$-bit full decoder. At least one extra AND gate is needed for the extra output since it is unique from all outputs in the full decoder. Thus, $t(n) \geq c(n-1) + 1$.

2. *Inductive Assumption*: For all p, we have

   $$t(n) \geq c(n-1) + p.$$

3. *Inductive Step*: For $p+1$,

   comparing two decoders with p and $p+1$ pairs of outputs having the same lower $n-1$ input bits. The latter must have a connected output that was unconnected in the former. Since the new connected output is unique, one more AND gate is needed, and thus $t(n) \geq c(n-1) + (p+1)$.

4. Q.E.D.

Claim 4.4: The proposed heuristic solution $s(n)$ approaches optimum exponentially
Proof: Let ξ be the maximum error between $s(n)$ and $t(n)$.

1. If $c(n-1) + 2p \leq c(n) - (2^{n-1} - p)$, then $s(n) = c(n-1) + 2p$.

   In this case, $c(n-1) + 2p \leq (1+\xi)\{c(n-1) + p\}$

   $$\iff (1-\xi)p \leq \xi c(n-1)$$

   $$\iff p \leq \frac{\xi c(n-1)}{1-\xi} \tag{1}$$

2. Otherwise, $s(n) = c(n) - (2^{n-1} - p)$.

   In this case, $c(n) - (2^{n-1} - p) \leq (1+\xi)\{c(n-1) + p\}$

   $$\iff p \geq \frac{c(n)-(1+\xi)c(n-1)-2^{n-1}}{\xi} \tag{2}$$

Equating (1) and (2), we have:

$$\frac{\xi c(n-1)}{1-\xi} = \frac{c(n)-(1+\xi)c(n-1)-2^{n-1}}{\xi}$$

$$\iff 1-\xi = \frac{c(n-1)}{c(n)-2^{n-1}}$$

$$\iff 1-\xi = \frac{2^{n-1}+c\lfloor (n-1)/2 \rfloor + c\lceil (n-1)/2 \rceil}{2^{n-1}+c\lfloor n/2 \rfloor + c\lceil n/2 \rceil}$$

If n is even, we have

$$\iff 1-\xi = \frac{2^{n-1}+c((n-2)/2)+c(n/2)}{2^{n-1}+c(n/2)+c(n/2)}$$

It was shown in the previous section that $c(n) = \Theta(2^n)$, Thus, we have

$$\iff 1-\xi = \frac{2^{n-1}+2^{(n-2)/2}+2^{n/2}}{2^{n-1}+2^n+2^{n/2}}$$

$$\iff \xi = \frac{1}{2^{n/2}+4}$$

$\Longleftrightarrow \xi = \Theta(\frac{1}{2^{n/2}})$

Similar reasoning can be used to prove that if n is odd, we have:

$\xi = \Theta(\frac{1}{2^{(n-1)/2}})$

In summary, it is proven that ξ decreases exponentially, and the mapping algorithm described above approaches optimum exponentailly fast. $s(n)$ is very close to optimal when n is large.

3.4 Discontinuous Don't Care Parital Decoders

Similar to its continuous counterpart, expressions for connected outputs may not require every input bit due to the existence of don't care unconnected outputs. However, due to the discontinuity of the connected outputs, the lower $n-1$ bits of input may not constitute a sub-full decoder. Therefore, instead of always eliminating the most significant input bit, different combinations of input bits may be eliminated to produce an optimal expression for each output. Figure 3-3 illustrates an extreme example of discontinuous don't care partial decoder in which a different combination of input bits is eliminated for each output, and in the end, each output requires only one input bit and thus no AND gates are needed.

Optimizing output expressions in this case can be viewed as a special case of two-level boolean minimization. A new boolean function F is constructed for each connected output bit, whose ON-set F^{ON} consists of only the output bit in question. The DC-set (don't-care set) F^{DC} consists of every unconnected output bit, and the OFF-set F^{OFF} consists of all other connected outputs. Figure 3-4 illustrates the new boolean function constructed for output $n4$ of the decoder described in Figure 3-3.

Quine-McCluskey (Q-M) method [8] starts with a list of 0-terms, which includes elements in both F^{ON} and F^{DC} sets. Each pair of 0-terms is merged into a single 1-term if they differ by exactly one position. A list of 1-terms is then merged into 2-terms, and so on, until no more merging is possible. If a k-term is formed by merging two (k-1)-terms, then the two (k-1)-terms are not primes and would be discarded later.

{U, U, U, n4, U, n2, n1, U} = Decoder(X[2:0])

	x[2]	x[1]	x[0]
U	*	*	*
U	*	*	*
U	*	*	*
n4	1	0	0
U	*	*	*
n2	0	1	0
n1	0	0	1
U	*	*	*

$\Longrightarrow$

N4 = w[2]

N2 = w[1]

N1 = w[0]

Not a Full Decoder

Figure 3-3: In this extreme example of a discontinuous don't care partial decoder, x[1:0] does not constitute a sub-full decoder. $x[2]$ is sufficient to identify $n4$ from other connected outputs. Similar reasoning can be applied to other two connected outputs. Thus, optimal expression for each output only requires one input bit, and no AND gate is required for this decoder.

ON-set: 100

DC-set: 111, 110, 101, 011, 000 - unconnected outputs

OFF-set: 010, 001 -- other connected outputs (n2, n1)

Figure 3-4: New boolean function constructed for output $n4$ of the decoder described in Figure 3-3.

(1)	100	-- ON-set	1-0	(1,3)		
(2)	111		10-	(1,4)		
(3)	110		**-00**	(1,6)	**1--**	((1 3), (2 4))
(4)	101	-- DC-set	11-	(2,3)		((1 4), (2 3))
(5)	011		1-1	(2,4)		
(6)	000		**-11**	(2,5)		
	0-terms		1-terms		2-terms	

Figure 3-5: Quine-McCluskey method applied to the boolean function in Figure 3-4. The prime terms are indicated in **bold**.

The result of the merging process is a set of prime terms, from whom a minimum subsets that covers F^{ON} is then selected. However, unlike traditional two-level sum-of-products functions, the boolean function constructed for each decoder output contains only one product term (F^{ON} has only one element). Therefore, instead of employing various minimum covering heuristics, we can inspect the prime list and select the one with the least number of literals, which also contains the original term in the ON-set. Figure 3-5 illustrates the Q-M process applied to the boolean function of $n4$ in Figure 3-4. In this case, the prime terms generated by Q-M is $1--$, -00, and -11. The first two prime terms contain the term 100 in the ON-set. Since the first prime has fewer number of literals, it is chosen as the optimal expression for output $n4$. It should be noted that even with the drastic simplification of choosing the minimum cover, the generation of all primes in the merging step still demands large computation time. For a function of n variables, the upper bound on the number of prime implicants is $3^n/n$ – there are nearly 3 million prime implicants for a 16-bit input decoder. Therefore, Q-M is not suited for optimizing decoder output expressions.

Many heuristic algorithms exist that perform the same function as Q-M, but have much shorter execution times. Most of them employ an *expand-reduce* iteration. In the first step, an implicant i is maximally expanded and other implicants covered by i are removed. The prime cover found depends on the ordering in which the variables are taken in the expansion [6]. To minimize the effect of the expansion ordering, the size of each implicant in the cover is then reduced so that it may lead to another cover with smaller cardinality in the next iteration. Unlike Q-M, these heuristic algorithms will not always find the exact optimum for the boolean function.

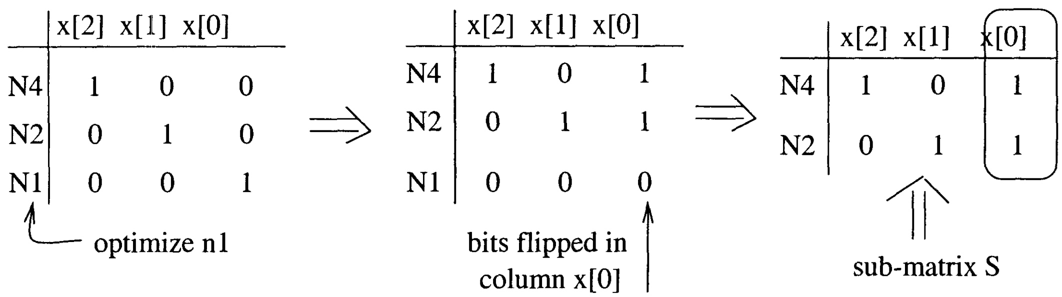


Figure 3-6: Set Cover Transformation.

In practice, minimizers such as MINI [5] and ESPRESSO [9] produce answers that are very close to optimal. Note that those heuristics are complex in order to solve complex two-level functions. Each decoder output expression however, is a simple AND of all input bits. Therefore, the complexity seems a bit unjustified in this case.

Minimization of decoder output expressions can also be transformed into a set-cover problem. Using the matrix that includes every connected outputs, we first inspect row m corresponding to the output bit in question. If the entry $e_{m,n}$ in row m and column n is 1, all entries in column n are then flipped. In the end, every entry in row m is now 0. Let S be the modified matrix without row m, and C be the minimum set of columns such that there is at least one 1 in every row of S. C is the minimum cover of S. The optimal expression for the output in question is then constructed using the input bits in set C only. Figure 3-6 illustrates the process using the decoder depicted in Figure 3-3. When optimizing output $n1$, bits in column $x[0]$ is flipped since $x[0]$ is 1 for $n1$. Removing row $n1$ from the matrix, column $x[0]$ covers the modified sub-matrix. Thus, the optimal expression for $n1$ is $n1 = x[0]$. Similar steps can be applied to other two outputs $n2$ and $n4$.

The validity of such transformation is reasoned as follows: expression minimization for the connected output in row m is equivalent to selecting a minimum set of literals (columns) that covers F^{ON} (row m) and does not intersect F^{OFF} (all rows except m). Denote C to be such sets of columns, $E_{m,C}$ to be entries in row m whose column belongs to C, and $E_{n,C}$ to be entries in row n ($n \neq m$) whose columns belongs to C. Note that after the first step, all elements in row m are 0, including elements in $E_{m,C}$. Since $E_{m,C}$ should not intersect with $E_{n,C}$, elements in $E_{n,C}$ must include at least one 1. The same thing can be said for all rows not equal to m. Thus, C is a minimum set of columns such that there is at least one 1 in every row of matrix S.

A wide range of set-cover algorithms has been well documented. In this case, a simple greedy method - selecting the column with the most 1's - is satisfactory. Even though examples can be shown where the cardinality of the computed cover using such method exceeds twice that of a minimum cover [2], the method itself is simple and fast. Assuming a decoder with n-input bits and C number of connected outputs, the running time is then $O(nC)$.

Denote $c(n)$ be the cost of n-bit full decoder, in this case, there are two choices to map Decoder(n):

1. Build a full n-bit decoder and then eliminate unnecessary boolean gates. The cost is $c(n) - [2^n - C]$, which is the same as its continuous counterpart.

2. Apply the greedy set-cover algorithm to identify optimal expressions for each connected output, and then build a separate circuit for each output. As a result of the boolean minimization, decoder output expressions as a whole have become less structed, and we are not able to know the modified expressions in advance. The literals and the number of literals may be different for each optimal output expression. Moreover, they may also be different for the same output in various decoders that have different sets of connected and disconnected outputs. Hence, it is difficult to construct a cost function that applies across the board. Decoder(n) is mapped using the one that uses few number of boolean gates in the end.

3.5 Continuous Care Partial Decoders

Unlike don't care partial decoders, expressions for connected outputs in this case must require all input bits since unconnected outputs cannot be treated as don't cares. Suppose unconnected outputs in Example 3-1 on page 38 are *not* don't cares, then input bit $x[3]$ cannot be ignored for output $n2$. Otherwise, an input of 10, which is out-of-bounds and produce an error, would drive output $n2$ to true and produce an incorrect design. Using the same reasoning, extra input bits and unconnected outputs cannot be eliminated even if the number of connected outputs is less than half of the total outputs, as did in Example 3-4 on page 44 for continuous don't care partial decoders. Therefore, the number of connected outputs can be anywhere between 0 to 2^n, with each output expression requiring all input bits.

Denote Decoder(n) as a continuous care partial decoder with n-input bits, C as the number of connected outputs, and $c(n)$ as the cost of n-bit full decoder. Due to the continuity of the connected outputs, the least significant $\lfloor \log_2 C \rfloor$ input bits constitute a full decoder. Similar to continuous don't care partial decoders, Decoder(n) can be built in two ways:

1. build the sub-full decoder from the lower $k = \lfloor \log_2 C \rfloor$ input bits, and then add on neccesary gates. Let X be decoder input. Since each output requires all input bits, boolean gates are needed for the most significant $n - k$ input bits – $X[k : n - 1]$. Since connected outputs are continuous, $X[k : n - 1]$ can have at most three different values, in which $X[k : k + 1]$ differ, and $X[k + 2 : n - 1]$ are the same. Otherwise, Decoder(n) would have a $k + 1$ sub-full decoder (see Figure 3-7).

   Hence, for $X[k : n-1]$, we need $n-k-3$ AND gates for $X[k+2 : n-1]$, 3 AND gates for $X[k : k + 1]$, and 3 AND gates to connect these two parts. Moreover, we may need to invert each bit, and a total of $n - k$ inverters may be needed. Therefore, $2n - 2k + 3$ number of gates are needed for the most significant $n - k$ input bits.

   Since each output is unique, we need one AND gate in each output to combine the right k input bits and the left $n - k$ input bits. Therefore, the total cost of this method is $c(k) + (2n - 2k + 3) + C$.

2. build a full decoder from all n input bits, and eliminate unnecessary gates. The total cost of this method is $c(n) - [2^n - C]$.

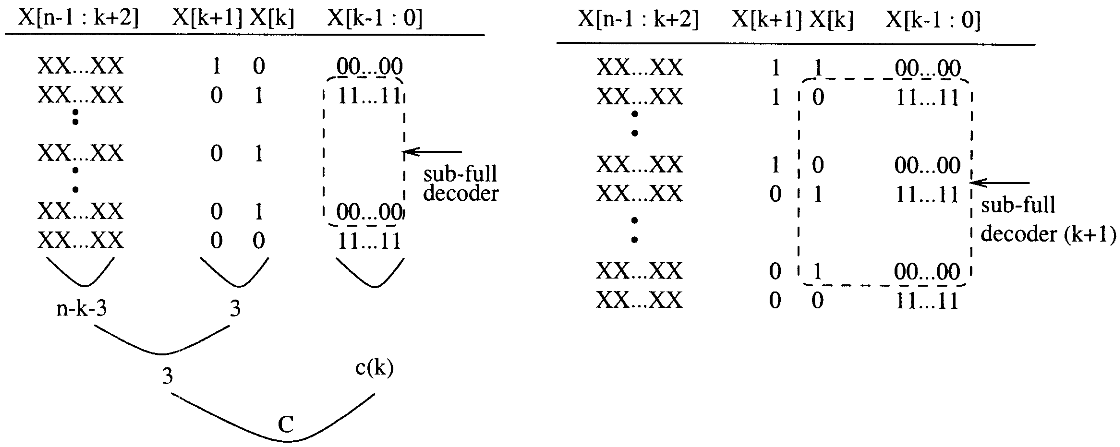


Figure 3-7: In (a), a continuous care partial decoder is built by generate a sub-full decoder and then add on neccessary gates. In (b), if the right k-bits forms a full decoder, then $X[n-1 : k+2]$ must be the same, and $X[k+1 : k]$ has at most threee different values. Otherwise, the decoder would have a sub-full decoder of more than k bits.

Let $s(n)$ denote the cost for Decoder(n), then we have

$$s(n) = min\{c(k) + (2n - 2k + 3) + C, \quad c(n) - [2^n - C]\}, (k = \lfloor \log_2 C \rfloor)$$

To obtain an asymptotic estimate on the range of C in which the first method is better than the second, we set

$$\begin{aligned} & c(k) + (2n - 2k + 3) + C \le c(n) - [2^n - C] \\ \Longleftrightarrow\ & c(k) + (2n - 2k + 3) \le 2c(n/2) \\ \Longleftrightarrow\ & \Theta(2^k) \le 2\Theta(2^{n/2}) \end{aligned}$$

It is obvious that the above inequality always holds when $k \le n/2$, and $C \le 2^{n/2}$. Therefore, when the sub-full decoder constitutes less than or equal to half of the input bits, the first method is applied. Otherwise, the second method is applied.

Claim 4.5: When $k \le n/2$, Decoder(n) requires at least $c(k) + C$ number of gates. Therefore, the first method is *asymptotically* optimal.

Proof: Since each output is unique, no matter how the input bits are split in the beginning, one AND gate is needed in each output to combine two sub-parts in the end – for a total of C number of AND gates. Assuming no gates are needed for input bits $X[n-1:k]$, then it is obvious that the least expensive way to split the input bits is to divide the decoder into two parts: $X[n-1:k]$ and $X[k-1:0]$. Since $X[k-1:0]$ consitutes a full decoder, it costs $c(k)$. Therefore, Decoder(n) costs at least $c(k)+C = \Theta(2^k)+\Theta(2^k) = \Theta(2^k)$. When $k \leq n/2$, the first method is adopted, which costs $c(k) + (2n-2k+3) + C = \Theta(2^k) + (2n-2k+3) + \Theta(2^k) = \Theta(2^k)$ - asymptotically optimal.

Claim 4.6: When $k > n/2$, Decoder(n) requires at least $c(n/2)+C$.

Proof: Based on Claim 4.5, when $k = n/2$, Decoder D requires at least $c(n/2)+C$ number of gates. Decoders in which $k > n/2$ have more unique outputs than D, thus they require at least the same number of gates as D.

Claim 4.7: When $k > n/2$, the proposed solution approaches optimum exponentially.

Proof: Let ξ be the error between $s(n)$ and the least number of gates when $k > n/2$.

$$c(n) - [2^n - C] = (1+\xi)[c(n/2)+C]$$

If n is even:

$$\Longrightarrow 2c(n/2)+C = (1+\xi)[c(n/2)+C]$$

$$\Longrightarrow \xi = \frac{c(n/2)}{c(n/2)+C}$$

$$\Longrightarrow \xi = \Theta(\frac{2^{n/2}}{2^{n/2}+2^k})$$

Since $k > n/2$, we have

$$\Longrightarrow \xi = \Theta(\frac{1}{1+2^{k-n/2}}) = \Theta(\frac{1}{2^{k-n/2}})$$

Therefore, ξ decreases exponentially, and $s(n)$ in this case approaches the minimum exponentially.

3.6 Discontinuous Care Partial Decoders

Similar to its continuous counterpart, expression for each output in this case requires all input bits. However, since its outputs are discontinuous, it is not obvious which part of input bits constitute a sub-full decoder, provided if one does exist. Figure 3-8 illustrates a discontinuous care partial decoder with five connected outputs, in which

x[4]	x[3]	x[2]	x[1]	x[0]
1	1	1	1	1
1	1	1	1	0
1	1	1	0	1
1	1	0	1	1
1	0	1	1	1

Figure 3-8: A discontinuous care partial decoder of 5-input bits with 5 connected outputs. Each outputs requires all input bits. No sub-input bits constitute a sub-full decoder.

no combination of input bits constitute a sub-full decoder. As a result, it only seems appropriate to build such decoders by generate a full decoder from all input bits first, and then eliminate unnecssary gates whose outputs do not drive anything.

Chapter 4

Experimental Results

The impact of decoder related optimizations implemented in the Presto Compiler is evaluated using Synopsys Place and Route Suite (PRS), which is a set of real-world designs used company-wide as a benchmark suite to evaluate the QoR of several design tools.

In the actual implementation, two switches were built for decoder sharing and decoder mapping respectively. By turning on and off the appropriate switches, we are able to exam the separate and combined effects of two optimizations. Table 4-1 summarizes the effect of decoder sharing only; Table 4-2 summarizes the effect of decoder mapping only; Table 4-3 summarizes the effect of both. The statistics measured in each table include:

1. DC # cells, Area: The number of cells in the final synthesized design; The area of the final synthesized design.

2. DC WNS: Worst Negative Slack in the design. Slack is the difference between the actual and expected signal arrival time. A negative slack means a violation of some timing requirement.

3. DC Rule Violate: The number of violations of user-specified design constraints.

4. RTL CPU hours, RTL memory: The CPU hours and memory used by Presto Compiler compiling the design.

5. DC CPU hours, DC memory: The CPU hours and memory used by the Design Compiler.

The first row in each table lists the changes averaged over the whole suite. Note that not all designs in the suite are affected by the optimization, e.g. designs without variable indices. Detailed stats are then listed for each design that was affected by the optimization. For each design, the first line lists stats without the optimization, the second line lists stats with the optimization, and the third line lists the percentage of the improvement achieved. A negative percentage means improvement, and a positive percentage means the opposite.

The goal of the optimization is to decrease the number of cells (DC # cells) and more importantly, the area (DC # Area) of the final design. Examing the results summarized in all three tables, we see that some designs did benefit from the optimaztions. For other designs however, the number of cells and the design area has increased as a result of the optimizations. Moreover, for those that did gain some benefit, the degree of improvement is not as great as expected. *Design6* contains a 9-bit variable index appeared three times. Three full decoders were built in the old design while only one was built in the new design. When measuring decoder sharing alone, decoders are mapped using the brute-force method, and hence a 9-bit full decoder costs 6400 number of gates. Since two decoders are saved in this case, the number of cells (DC #Cells) should decrease by 12800. However, according to Table 4.1, the actual difference is only 2569. *Design7* has the exact same situation, and its actual difference is only 1760. *Design8* contains a 8-bit full decoder. When measuring decoding mapping alone, the old design costs 2816 and the new design costs 312. While theorectically 2504 number of gates should be saved, the actual saving is only 580 according to Table 4.2.

It is believed that the less-than-expected improvement is mainly caused by high fan-out gates produced as a result of decoder optimizations. When a decoder is shared among multiple arrays, its outputs would be driving multiple *select* cells instead of one. Assume n is the number of decoder input bits. In the recursive mapping of the full decoder, each output from the left half of the input needs to be ANDed

	DC #Cells	DC Area	DC WNS	DC Rule Violate	RTL CPU Hours	DC CPU Hours	RTL (MB) Memory	DC (MB) Memory
Average:								
	-0.14%	-0.45%	+0.26%	-4.17%	1.0029X	0.9823X	0.9841X	0.9903X
DESIGN1:								
	27537	221296	-2.75	1	0.00789	1.45	175	393
	27466	223642	-2.82	1	0.0068	1.35	159	381
	-0.26%	+1.06%	+1.13%	+0.00%	0.8621X	0.9299X	0.9119X	0.9691X
DESIGN2								
	42901	531777	-0.43	0	0.0193	2.91	329	756
	43399	534085	-0.53	0	0.0171	2.57	315	747
	+1.16%	+0.43%	+0.99%	+0.00%	0.8873X	0.8851X	0.9582X	0.9882X
DESIGN3								
	87642	247935	-1.3	0	0.0106	0.786	130	531
	88155	248343	-1.24	0	0.00979	0.743	129	517
	+0.59%	+0.16%	-1.27%	+0.00%	0.9231X	0.9457X	0.9926X	0.9738X
DESIGN4								
	184255	3.21e+06	-1.2	1	0.0316	2.45	505	1.3e+03
	184300	3.21e+06	-1.16	0	0.0313	2.48	505	1.3e+03
	+0.02%	-0.21%	-0.79%	-100.00%	0.9914X	1.0106X	1.0000X	0.9953X
DESIGN5								
	6855	260521	-0.25	0	0.00136	0.11	81.7	173
	6781	259067	-0.32	0	0.00136	0.105	81.7	171
	-1.08%	-0.56%	+3.85%	+0.00%	1.0000X	0.9530X	1.0000X	0.9876X
DESIGN6								
	112501	1.09e+06	-1.66	0	0.0264	2.55	241	774
	109932	1.03e+06	-1.61	0	0.0337	2.72	219	727
	-2.28%	-5.84%	-0.94%	+0.00%	1.2784X	1.0675X	0.9061X	0.9396X
DESIGN7								
	112683	1.09e+06	-1.59	0	0.0264	2.76	241	774
	110923	1.03e+06	-1.71	0	0.0297	2.04	219	727
	-1.56%	-5.76%	+2.29%	+0.00%	1.1237X	0.7384X	0.9062X	0.9389X

Table 4.1: Optimization Results for Decoder Sharing Only

DC #Cells	DC Area	DC WNS	DC Rule Violate	RTL CPU Hours	DC CPU Hours	RTL (MB) Memory	DC (MB) Memory
-0.08%	-0.17%	+0.25%	+4.17%	1.0082X	0.9918X	0.9825X	0.9957X
DESIGN1							
27537	221296	-2.75	1	0.0068	1.3	174	393
27900	218715	-2.75	1	0.00735	1.07	165	384
+1.32%	-1.17%	+0.00%	+0.00%	1.0800X	0.8253X	0.9449X	0.9762X
DESIGN2							
42901	531777	-0.43	0	0.0193	2.97	329	756
41678	522110	-0.48	0	0.0193	2.88	311	744
-2.85%	-1.82%	+0.50%	+0.00%	1.0000X	0.9692X	0.9470X	0.9843X
DESIGN3							
87642	247935	-1.3	0	0.0106	0.782	130	53
87642	247935	-1.3	0	0.0106	0.788	130	530
+0.00%	+0.00%	+0.00%	+0.00%	1.0000X	1.0077X	0.9993X	1.0000X
DESIGN4							
184255	3.21e+06	-1.2	1	0.0318	2.41	505	1.3e+03
184288	3.21e+06	-1.16	1	0.0313	2.46	504	1.3e+03
+0.02%	-0.07%	-0.79%	+0.00%	0.9829X	1.0215X	0.9984X	0.9965X
DESIGN5							
6855	260521	-0.25	0	0.00136	0.108	81.8	173
6786	260251	-0.25	0	0.00136	0.101	81.7	173
-1.01%	-0.10%	+0.00%	+0.00%	1.0000X	0.9320X	0.9988X	1.0008X
DESIGN6							
112501	1.09e+06	-1.66	0	0.0258	2.53	241	774
112825	1.09e+06	-1.68	0	0.0253	2.86	215	761
+0.29%	+0.15%	+0.38%	+0.00%	0.9789X	1.1315X	0.8904X	0.9832X
DESIGN7							
112683	1.09e+06	-1.59	0	0.0261	2.71	241	774
113450	1.09e+06	-1.77	0	0.025	2.35	215	761
+0.68%	-0.49%	+3.44%	+0.00%	0.9583X	0.8645X	0.8905X	0.9829X
DESIGN8							
207064	5.28e+06	-0.33	13	0.0106	1.59	235	1.3e+03
206484	5.25e+06	-0.38	14	0.0109	1.66	233	1.3e+03
-0.28%	-0.68%	+1.48%	+100.00%	1.0256X	1.0450X	0.9879X	0.9967X

Table 4.2: Optimization Results for Decoder Mapping Only

DC #Cells	DC Area	DC WNS	DC Rule Violate	RTL CPU Hours	DC CPU Hours	RTL (MB) Memory	DC (MB) Memory
-0.82%	-0.93%	-0.04%	+4.35%	1.0101X	0.9902X	0.9752X	0.9882X
DESIGN1							
27537	221296	-2.75	1	0.00789	1.47	175	393
27922	218928	-2.69	1	0.00653	1.08	157	378
+1.40%	-1.07%	-0.97%	+0.00%	0.8276X	0.7346X	0.8990X	0.9620X
DESIGN2							
42901	531777	-0.43	0	0.0193	2.97	329	756
38141	501656	-0.29	0	0.0185	2.53	306	741
-11.10%	-5.66%	-1.39%	+0.00%	0.9577X	0.8501X	0.9318X	0.9809X
DESIGN3							
87642	247935	-1.3	0	0.0106	0.788	130	531
88155	248343	-1.24	0	0.0103	0.821	129	517
+0.59%	+0.16%	-1.27%	+0.00%	0.9744X	1.0418X	0.9932X	0.9739X
DESIGN4							
184255	3.21e+06	-1.2	1	0.0324	2.47	505	1.3e+03
184297	3.21e+06	-1.16	1	0.0313	2.51	505	1.3e+03
+0.02%	-0.23%	-0.79%	+0.00%	0.9660X	1.0162X	1.0000X	0.9934X
DESIGN5							
6855	260521	-0.25	0	0.00136	0.109	81.7	173
6781	259067	-0.32	0	0.00136	0.113	81.7	171
-1.08%	-0.56%	+3.85%	+0.00%	1.0000X	1.0376X	1.0000X	0.9876X
DESIGN6							
112501	1.09e+06	-1.66	0	0.0258	2.48	241	774
108468	1.02e+06	-1.61	0	0.0332	2.4	206	721
-3.58%	-6.87%	-0.94%	+0.00%	1.2842X	0.9683X	0.8529X	0.9319X
DESIGN7							
112683	1.09e+06	-1.59	0	0.0258	2.7	241	774
107395	1.02e+06	-1.51	0	0.0324	3.04	206	721
-4.69%	-6.79%	-1.53%	+0.00%	1.2526X	1.1263X	0.8530X	0.9314X
DESIGN8							
207064	5.28e+06	-0.33	13	0.0109	1.66	235	1.3e+03
206484	5.25e+06	-0.38	14	0.0106	1.65	232	1.3e+03
-0.28%	-0.68%	+1.48%	+100.00%	0.9750X	0.9933X	0.9877X	0.9967X

Table 4.3: Optimization Results for Decoder Sharing and Mapping Combined

with every output from the right half of the input, and vice versa. Therefore, every AND gate in the first recursive step has a fan-out of $2^{n/2}$, every AND gate in the second recursive step has a fan-out of $2^{n/4}$, and so on. While the saving on number of gates increases as n increases, the fan-out of AND gates in the mapped decoder also increases exponentially. In order to lessen the loads for wires with high fan-outs, the Design Compiler inserts buffers between them and the cells that they are driving. The extra buffers add to the cell numbers and design area, directly counters the benefit provided by the implemented optimizations.

Furthermore, without decoder sharing, a new decoder is generated for each array reference, which can then be placed right next to the *select* cell driven by the decoder. However, when one decoder is shared among multiple arrays, it is inevitable for the decoder to placed further away from some of the *select* cells, which results in longer wires. When the new decoder mappings, the output of one AND gate is shared among many other AND gates, and thus long wires can also be generated for the same reason. Longer wires leads to longer delays and possibly longer critical path, hence producing worse WNS.

Note that it is difficult to lessen the revese effects described above. Once the Design Compiler takes over, it restructures parts of the design as it sees fit, and Presto has no way of predicting which parts of design will be modified. However, what we really care about is the final design area. From Table 4.3, we see that the area was improved for all designs when both decoder optimizations are turned on.

In conclusion, the design areas are improved somewhat at a low cost of circuit timing (WNS). The effects of the optimizations described in this paper are very design-specific. If a design contains a lot of sharable variable array subscripts, then decoder-sharing would improve the design QoR greatly. If a design contains a lot of large decoders, then decoder-mapping would improve the design QoR greatly.

Bibliography

[1] John Cocke. Global common subexpression elimination. *Proceedings of a symposium on Compiler optimization*, 1970.

[2] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms.* The MIT Press, 2001.

[3] E.Morel and C.Renvoise. Global optimization by suppression of partial redundancies. *Communications of the ACM*, 22:96–103, February 1979.

[4] Michael D. Ernst. Serializing parallel programs by removing redundant computation. Technical report, MIT, 1994.

[5] S. Hong, R. Cain, and D. Ostapko. Mini: A heuristic approach for logic minimization. *IBM Journal of Research and Development*, 1974.

[6] Giovanni De Micheli. *Synthesis and Optimization of Digital Circuits.* McGraw-Hill, 1994.

[7] Stenven S Muchnick. *Advanced Compiler Design Implementation.* Morgan Kaufman, 1997.

[8] MIT 6.371 Class Notes. Two-level boolean minization.

[9] R. Rudell and A. Sangiovanni-Vincentelli. Exact minimization of multiple-valued functions for pla optimization. *IEEE TCAD*, 1987.

[10] Synopsys. *HDL Compiler for Verilog*, May 2000.

[11] Synopsys. *Design Compiler User Guide*, June 2002.

[12] Synopsys. *DesignWare GTECH Library Databook*, June 2004.