On Designs of Radix Converters Using Arithmetic Decompositions — Binary to Decimal Converters —

Yukihiro Iguchi [1] Tsutomu Sasao [2] Munehiro Matsuura [2]

[1] Dept. of Computer Science, Meiji University, Kawasaki 214-8571, Japan
[2] Dept. of Computer Science and Electronics, Kyushu Institute of Technology, Iizuka 820-8502, Japan

Abstract

In digital signal processing, radixes other than two are often used for high-speed computation. In the computation for finance, decimal numbers are used instead of binary numbers. In such cases, radix converters are necessary. This paper considers design methods for binary to q-nary converters. It introduces a new design technique based on weighted-sum (WS) functions. The method computes a WS function for each digit by an LUT cascade and a binary adder, then adds adjacent digits with q-nary adders. A 16*-bit binary to decimal converter is designed to show the method.*

1. Introduction

Arithmetic operations of digital systems are usually done by binary numbers [8]. However, for high-speed digital signal processing, p-nary $(p > 2)$ numbers are often used [1, 5]. Computation for finance usually uses decimal numbers instead of binary numbers. In such cases, conversions between binary numbers and p-nary numbers are necessary. Such operation is **radix conversion** [2, 7].

Various methods exist to convert p-nary numbers into q-nary numbers, where $p \geq 2$ and $q \geq 2$. Many of them require large amount of computations. Radix converters can be implemented by table lookup. That is, to store the conversion table in the memory. This method is fast, but requires a large amount of memory. When the number of inputs is large, the memory is too large to implement. Thus, more efficient methods have been developed.

In [6], ROMs and adders are used to implement binary to decimal converters.

In [10], LUT cascades [9] are used to implement binary to ternary converters, ternary to binary converters, binary to decimal converters, and decimal to binary converters.

In [13], weighted-sum functions (WS functions) is introduced to design radix converters by LUT cascades.

In [3], LUT cascade and arithmetic decomposition [12] are used to implement p-nary to binary converters that require smaller memory.

In this paper, we consider a design method of p-nary to q-nary $(q > 2)$ converters by using LUT cascades and arithmetic decompositions. To explain the concepts, we use examples of binary to decimal converters, but the method can be easily modified to any numbers of p and q. A 16-bit binary to decimal converter is designed to show the method.

2. Radix Converter

2.1. Radix Conversion

Definition 2.1 *Let $\vec{x} = (x_{n-1}, x_{n-2}, \ldots, x_0)_p$ be a p-nary number of n-digit, and let $\vec{y} = (y_{m-1}, y_{m-2}, \ldots, y_0)_q$ be a q-nary number of m-digit. Given the vector $\vec{x}$, the* **radix conversion** *is the operation that obtains $\vec{y}$ satisfying the relation:*

$$\sum_{i=0}^{n-1} x_i p^i = \sum_{j=0}^{m-1} y_j q^j, \tag{2.1}$$

where $x_i \in P$, $y_j \in Q$, $P = \{0, 1, \ldots, p-1\}$, and $Q = \{0, 1, \ldots, q-1\}$.

When p (q) is not 2, they are represented by binary coded p (q)-nary numbers.

Definition 2.2 *Let i be an integer. $BIT(i, j)$ denotes the j-th bit of the binary representation of i, where the LSB is the 0-th bit.*

Example 2.1 *Note that an integer 6 is represented by the binary number* $(1, 1, 0)$=$(BIT(6, 2)$, $BIT(6, 1)$, $BIT(6, 0))$. *Thus,* $BIT(6, 2) = 1$, $BIT(6, 1) = 1$, *and* $BIT(6, 0) = 0$. (End of Example)

Example 2.2 *In the case of a binary to decimal converter, a decimal number is represented by the binary-coded-decimal (BCD) code. That is,* 0 *is represented by* (0000), 1 *is represented by* (0001), . . ., 8 *is represented* (1000), *and* 9 *is represented by* (1001). *Note that* (1010), (1011),. . ., (1111)

Table 2.1. Truth table of a binary to decimal converter.

Binary				Decimal		Binary Coded Decimal							
x_3	x_2	x_1	x_0	y_1	y_0								
0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	1	0	1	0	0	0	0	0	0	0	1
0	0	1	0	0	2	0	0	0	0	0	0	1	0
0	0	1	1	0	3	0	0	0	0	0	0	1	1
0	1	0	0	0	4	0	0	0	0	0	1	0	0
0	1	0	1	0	5	0	0	0	0	0	1	0	1
0	1	1	0	0	6	0	0	0	0	0	1	1	0
0	1	1	1	0	7	0	0	0	0	0	1	1	1
1	0	0	0	0	8	0	0	0	0	1	0	0	0
1	0	0	1	0	9	0	0	0	0	1	0	0	1
1	0	1	0	1	0	0	0	0	1	0	0	0	0
1	0	1	1	1	1	0	0	0	1	0	0	0	1
1	1	0	0	1	2	0	0	0	1	0	0	1	0
1	1	0	1	1	3	0	0	0	1	0	0	1	1
1	1	1	0	1	4	0	0	0	1	0	1	0	0
1	1	1	1	1	5	0	0	0	1	0	1	0	1

are unused codes. Table 2.1 is the truth table of the 4*-digit binary to decimal converter.*

In Table 2.1, the inputs in the binary representation are denoted by $\vec{x} = (x_3, x_2, x_1, x_0)$*. The outputs in the decimal representation are denoted by* $\vec{y}$= (y_1, y_0)*, and in the binary-coded-decimal representation are denoted by* $(BIT(y_1, 3)$*,* $BIT(y_1, 2)$*,* $BIT(y_1, 1)$*,* $BIT(y_1, 0)$*,* $BIT(y_0, 3)$*,* $BIT(y_0, 2)$*,* $BIT(y_0, 1)$*,* $BIT(y_0, 0))$*.* (End of Example)

2.2. Conventional Realization

2.2.1 Random Logic Realization

Radix converters can be implemented by using comparators, subtracters, and multiplexers.

Example 2.3 *Consider the 8-digit binary to decimal converter (***8bin2dec***) shown in Figure 2.1. It converts 8-digit binary numbers* $x[7:0]$ *into* 3*-digit BCD. The output range is 0 to* 255*.*

First, the subtracter S_1 *calculates* $x - 200$*. The comparator* C_1 *compares the input* x *with* 200*. When* $x \geq 200$ *(*$x < 200$*),* $o2[1] = 1(0)$*, and the multiplexer* M_1 *passes* $t_1 = x - 200$ *(*$t_1 = x$*) to the next stage. Next, the subtracter* S_2 *calculates* $t_1 - 100$*. The comparator* C_2 *compares the value* t_1 *with* 100*. When* $t_1 \geq 100$ *(*$t_1 < 100$*),* $o2[0] = 1(0)$*, and the multiplexer* M_2 *passes* $t_2 = t_1 - 100$ *(*$t_2 = t_1$*) to the next stage. In this way, the circuit converts the binary number into the decimal number.* (End of Example)

2.2.2 Single Memory Realization

Figure 2.2 shows a single memory realization that stores the truth table of the radix converter. It converts n-digit binary-coded p-nary numbers into m-digit binary-coded q-nary numbers.

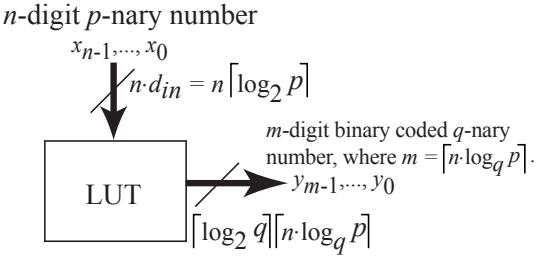

Figure 2.2. n**-digit** p**-nary to** q**-nary converter: Single memory realization.**

Let d_{in} be the number of bits to represent an input digit. Then, $d_{in} = \lceil \log_2 p \rceil$. Since the number of input digits is n, the total number of bits in the input is nd_{in}. A p-nary number with n digits takes values from 0 to $p^n - 1$.

Let d_{out} be the number of bits in an output digit. Then, $d_{out} = \lceil \log_2 q \rceil$. The number of output digits is $m = n \cdot \lceil \log_q p \rceil$. Note that the most significant digit requires only $d_{out_{m-1}} = \lceil \log_2([p^{n-1}/q^{m-1}]) \rceil$ bits. Thus, the size of the memory is $2^{nd_{in}}((m-1) \cdot d_{out} + d_{out_{m-1}})$ bits.

This method is simple and fast, but, when the number of digits for the radix converter is large, the memory will be huge.

Example 2.4 *The* 16*-digit binary to decimal converter (***16bin2dec***) takes the range* $[0, 2^{16} - 1] = [0, 65535]$*. The output is represented by* $m = \lceil \log_{10} 65535 \rceil = 5$ *digits.* $d_{out} = 4$*, and* $d_{out_{m-1}} = 3$*. When the converter is realized by a single memory, its size is* $2^{16} \cdot ((5-1)4 + 3) = 1,245,184$ *bits.* (End of Example)

2.2.3 LUT Cascade Realization

In a single memory realization of a p-nary to q-nary converter, the size of memory tends to be too large.

To reduce the amount of hardware, LUT cascades realizations are used in [10], where outputs are partitioned into groups. By using the functional decomposition theory [8], we can predict such a realization is feasible or not. To perform functional decompositions, a Binary Decision Diagram for Characteristic Function (BDD_for_CF) [10] is used as the data structure.

Figure 2.3 shows the 16-digit binary to decimal converter [10], where the LUT cascade with three cells realizes all the outputs. LUTs are implemented by memories. The total memory size is $73,728$ bits. This method uses only memory, and the interconnections are limited only to adjacent memories. However, since the logic synthesis uses BDD_for_CFs, when the number of digits is large, the computation time tends to be excessive.

On the other hand, the design method in [3] finds LUT cascades without using BDDs_for_CF. It produces circuits

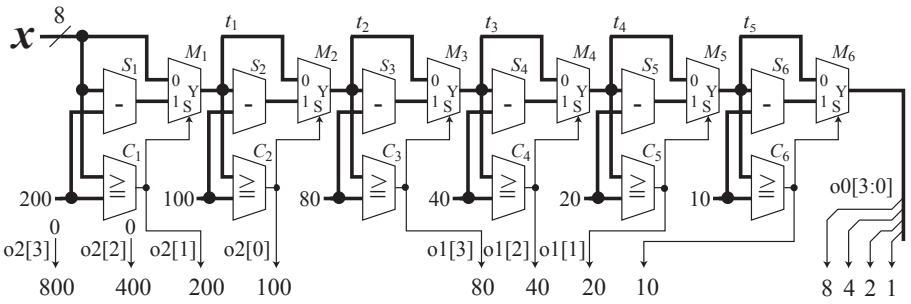

Figure 2.1. 8-digit binary to decimal converter: Random Logic Realization.

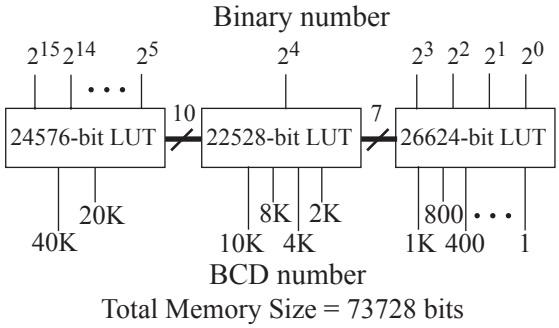

Figure 2.3. 16-digit binary to decimal converter: LUT Cascade Realization.

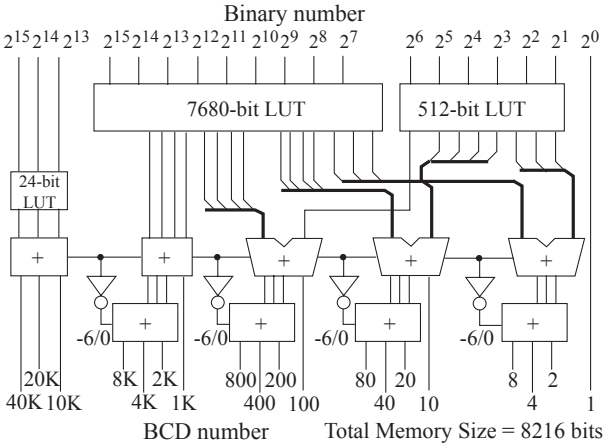

Figure 2.4. 16-digit binary to decimal converter: Realization by memories and BCD adders

with a smaller amount of memory than [10] by using arithmetic decompositions. Unfortunately, this method can design only p-nary ($p > 2$) to binary converters, and is not applicable to binary to q-nary ($q > 2$) converters.

2.2.4 Realization by Memories and q-nary Adders

Design methods of binary to decimal converters using memories and adders are shown in [6]. Figure 2.4 shows a 16-digit binary to decimal converter [6]. The features of this circuit are as follows:

1. In the binary to decimal converter, the input 2^0 is directly connected to the least significant bit of the output (1).
2. Other 15 inputs are divided into two: The upper 9 bits and the lower 6 bits. For all possible inputs, each LUT stores corresponding BCD numbers.
3. The most significant digit (40K, 20K, and 10K) is obtained by a 3-input LUT (originally, which was implemented by gates) and a binary adder.
4. The total amount of memory is $8,216$ bits.
5. The middle LUT stores BCD values in **excess**-6-code.
6. BCD additions are implemented by binary adders and special subtracters. When the carry is added to the next higher order digit, 6 is subtracted from the BCD digit.

The original circuit in [6] used a 1-of-16 decoder which generates memory selection signals for 16 small-scale memories. Since larger memories are available today, we replaced them by a single LUT.

With this method, a binary to decimal converter is implemented by using LUTs and q-nary ($q = 10$) adders. This method reduces the total amount of memory by partitioning the inputs into two.

3. WS Function

The weighted sum function (WS function) is a mathematical model of radix converters, bit-counting circuits, and convolution operations [12, 13]. In this section, we show some properties of WS functions, and give a design method of radix converters by using them.

Definition 3.1 *An n-input WS function [13] is defined as*

$$WS(\vec{x}) = \sum_{i=0}^{n-1} w_i \cdot x_i, \tag{3.1}$$

where $\vec{x} = (x_{n-1}, x_{n-2}, \ldots, x_1, x_0)$ is the input vector,

$\vec{w} = (w_{n-1}, w_{n-2}, \ldots, w_1, w_0)$ *is the* **weight vector** *consisting of integers.*

In this paper, we represent p-nary to q-nary conversions with WS functions. From here, unless otherwise noted, w_i and x_i denote non-negative integers.

Definition 3.2 *Let MIN_n (MAX_n) be the minimum (maximum) number represented by a WS function $WS(\vec{x}) = \sum_{i=0}^{n-1} w_i x_i$, where $w_i \geq 0$, $x_i \in \{0, 1, \ldots, p-1\}$, and $p \geq 2$.*

When all the input x_i are 0's, a WS function takes its minimum value, $MIN_n = WS(0, 0, \ldots, 0) = 0$. When all the input x_i are $p-1$, the WS function takes its maximum value, $MAX_n = WS(p-1, p-1, \ldots, p-1) = \sum_{i=0}^{n-1}\{w_i \cdot (p-1)\} = (p-1)\sum_{i=0}^{n-1} w_i$.

Definition 3.3 *For $i < j$, $[i, j]$ denotes the set of integers, $\{i, i+1, \ldots, j\}$.*

Next, we will consider the range of WS functions.

Definition 3.4 *$Range(f(x))$ denotes the range of a function $f(x)$.*

Example 3.1 *For $WS_1(\vec{x}) = x_0 + 3x_1$, and $x_i \in \{0, 1, 2\}$, we have $Range(x_0) = \{0, 1, 2\}$, and $Range(3x_1) = \{0, 3, 6\}$. Thus, $Range(WS_1(\vec{x})) = \{0, 1, 2, 3, 4, 5, 6, 7, 8\} = [0, 8]$. On the other hand, for $WS_2(\vec{x}) = x_0 + 4x_1$, and $x_i \in \{0, 1, 2\}$, we have $Range(x_0) = \{0, 1, 2\}$, and $Range(4x_1) = \{0, 4, 8\}$. Thus, $Range(WS_2(\vec{x})) = \{0, 1, 2, 4, 5, 6, 8, 9, 10\} \neq [0, 10]$. For $WS_3(\vec{x}) = x_0 + 2x_1$, and $x_i \in \{0, 1, 2\}$, we have $Range(x_0) = \{0, 1, 2\}$, and $Range(2x_1) = \{0, 2, 4\}$. Thus, $Range(WS_3(\vec{x})) = \{0, 1, 2, 3, 4, 5, 6\} = [0, 6]$.* (End of Example)

Example 3.1 shows that $Range(WS(\vec{x})) = [0, MAX_n]$ holds in some cases and does not in other cases. It depends on the combinations of values of w_i. In the case of $WS_2(\vec{x}) = x_0 + 4x_1$, $MIN = 0$, and $MAX = 2(1 + 4) = 10$. Because the number of values for x_i is 3, $WS_2(\vec{x})$ takes at most $3^2 = 9$ different values. Therefore, $Range(WS_2(\vec{x})) \neq [0, 10]$.

Lemma 3.1 *The number of values produced by a WS function $WS(\vec{x})$, is at most p^n, where $WS(\vec{x}) = \sum_{i=0}^{n-1} w_i x_i$, $w_i \geq 0$, $x_i \in \{0, 1, \ldots, p-1\}$, and $p \geq 2$.*
(Proof: Omitted)

The next theorem shows the necessary and sufficient condition for $Range(WS(\vec{x})) = [0, MAX_n]$.

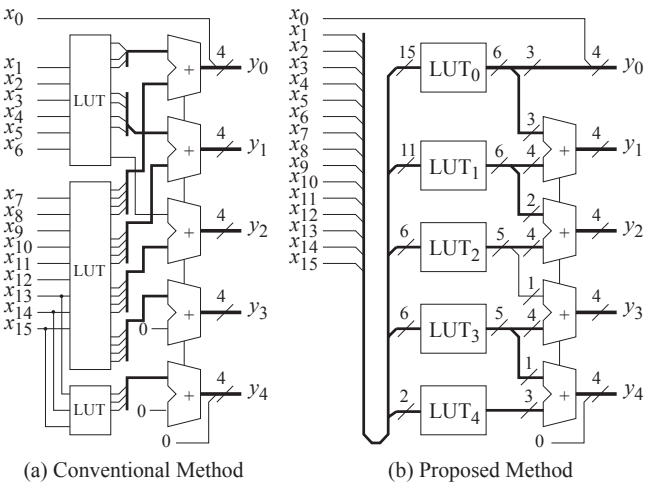

Figure 4.1. Realization Using Memories and q-nary Adders.

Theorem 3.1 *Consider a WS function $WS(\vec{x}) = \sum_{i=0}^{n-1} w_i x_i$, $w_i \geq 0$, $x_i \in \{0, 1, \ldots, p-1\}$, and $p \geq 2$. $Range(WS(\vec{x})) = [0, MAX_n]$ iff $w_0 = 1$ and $w_i \leq MAX_{i-1} + 1$.*
(Proof: Omitted)

Corollary 3.1 *$WS(\vec{x}) = \sum_{i=0}^{n-1} p^i x_i$ takes all the value in $[0, p^n - 1]$.*
(Proof: Omitted)

4. Realization Using LUT Cascades and Arithmetic Decompositions

In this part, we consider design methods of radix converters by using LUT cascades and arithmetic decompositions.

Figure 4.1(a) shows the conventional circuit which is redrawn from Fig. 2.4 with fewer blocks. This circuit is implemented by using only three LUTs and five decimal adders. In Fig. 4.1(a), outputs of the middle LUT are connected to q-nary adders.

On the other hand, Fig. 4.1(b) shows the proposed method in this paper. In this method, each LUT stores BCD values, and feeds to a digit and a carry out. Consequently, outputs from each LUT blocks are connected only to the corresponding q-nary adder and the adjacent q-nary adder. However, this method requires m LUTs. Furthermore, the numbers of inputs of LUTs for lower digits are large. From here, we are going to reduce the total amount of memory for LUT cascades using the concept of WS functions.

Table 4.1. Decimal Number Representations for Power of Two

x	2^{15}	2^{14}	2^{13}	2^{12}	2^{11}	2^{10}	2^{9}	2^{8}	2^{7}	2^{6}	2^{5}	2^{4}	2^{3}	2^{2}	2^{1}	2^{0}
10^4	3	1														
10^3	2	6	8	4	2	1										
10^2	7	3	1	0	0	0	5	2	1							
10^1	6	8	9	9	4	2	1	5	2	6	3	1				
10^0	8	4	2	6	8	4	2	6	8	4	2	6	8	4	2	1

4.1. LUT Cascade Realizations of WS Functions

Here, we consider the logic function realized by each LUT in Fig. 4.1(b). Table 4.1 shows decimal numbers for 2^i $(i = 15, 14, \ldots, 0)$. The rows show digits for 10^4, 10^3, 10^2, 10^1 and 10^0. We can obtain functions of LUTs in Fig. 4.1(b) from Table 4.1. For example, the value of the digit for $10000 = 10^4$ can be computed with x_{14}, x_{15}, and the carry propagation signal from the lower digit. We obtain the following equations:

$$z_4 = x_{14} + 3x_{15}, \quad (4.1)$$

$$z_3 = x_{10} + 2(x_{11} + x_{15}) + 4x_{12} + 6x_{14} + 8x_{13}, \quad (4.2)$$

$$z_2 = (x_7 + x_{13}) + 2x_8 + 3x_{14} + 5x_9 + 7x_{15}, \quad (4.3)$$

$$z_1 = (x_4 + x_9) + 2(x_7 + x_{10}) + 3x_5 + 4x_{11} + 5x_8 + 6(x_6 + x_15) + 8x_14 + 9(x_{12} + x_{13}), \quad (4.4)$$

$$z_0 = x_0 + 2(x_1 + x_5 + x_9 + x_{13}) + 4(x_2 + x_6 + x_{10} + x_{14}) + 6(x_4 + x_8 + x_{12}) + 8(x_3 + x_7 + x_{11} + x_{15}), \quad (4.5)$$

$$y = 10^4 z_4 + 10^3 z_3 + 10^2 z_2 + 10 z_1 + z_0, \quad (4.6)$$

where $z_i (i = 0, 1, 2, 3, 4)$ represents the logic function for LUT_i.

Since equations (4.1) – (4.5) represent WS functions, we can use the properties in Section 3 to construct an LUT cascade having small amount of memory. Also, we can reduce the number of levels for the LUT cascade.

The LUT cascade shown in Fig. 4.2(a) realizes equation (4.5). Each LUT has only one external input x_i. As shown in Table 4.1, weight coefficients are non-negative integer. Equation (4.5) has only non-zero weight coefficients. Note that equation (4.3) representing the digit 10^2, has zero weight coefficients w_{12}, w_{11}, w_{10},w_6, ..., w_0 which are multiplied by x_{12}, x_{11}, x_{10}, x_6, ..., x_0.

In the circuit realization, terms with zero coefficients are omitted, and the input variables are reordered so that the weights are arranged in increasing order.

To find the minimum cascades, we consider the cases where adjacent LUTs are merged and not. Let s be the number of LUTs in the LUT cascade. The number of different combinations is 2^{s-1}. In this case, the input variable x_i incidents to LUT_i, and each LUT has $d = \lceil \log_2 p \rceil$ two-valued inputs. With these constraints, we have 2^{s-1} different combinations for all LUT cascades realizations.

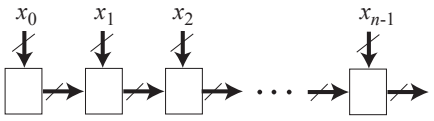

Figure 4.3. LUT Cascade.

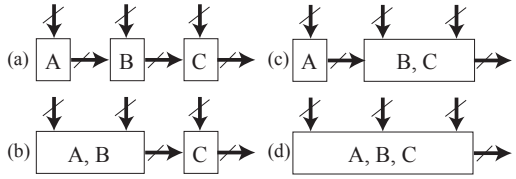

Figure 4.4. All Possible LUT Cascade Realizations for the LUT cascades with 3 cells.

Theorem 4.1 *Consider the LUT cascade in Fig. 4.3, where the variable ordering and assignment to the cells are fixed. In this case, the LUT cascade with the minimum memory size can be found among 2^{s-1} different combinations.*
(Proof: Omitted)

Example 4.1 *Figure 4.4(a)–(d) show the all possible LUT cascade realizations for the LUT cascades with 3 cells shown in (a).* (End of Example)

The next lemmas show methods to detect mergeable LUTs in an LUT cascade.

Lemma 4.1 *Consider the LUT cascade in Fig. 4.5, where LUT H has k inputs and k outputs. In this case, without increasing the amount of memory, two LUTs can be merged into one.*
(Proof: Omitted)

By using Lemma 4.1, we can find mergeable LUTs. In Fig. 4.5, we can reduce the LUT H and one level.

Lemma 4.2 *Consider the LUT cascade shown in Fig. 4.6, where LUT H has k-input and $(k-1)$-output, and LUT G has k-input $(k-1)$-output. In this case, without increasing the amount of memory, two LUTs can be merged into one.*
(Proof: Omitted)

Lemma 4.2 shows a method to reduce one levels without increasing total amount of memory. For other cases, the merge of adjacent LUTs will increase the total amount of memory.

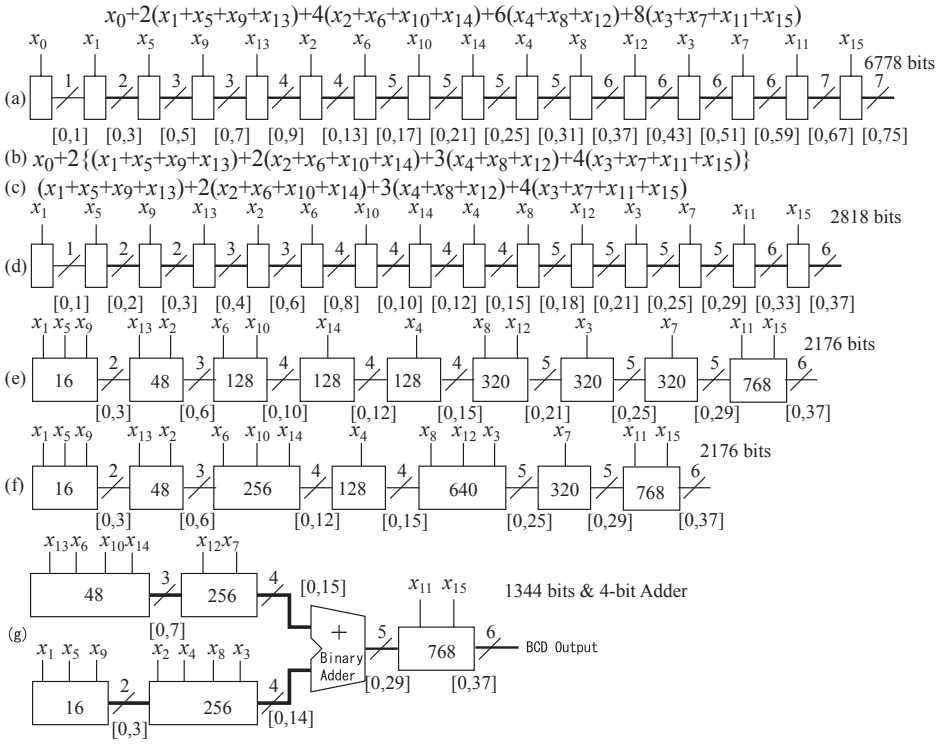

Figure 4.2. LUT Cascade for z_0.

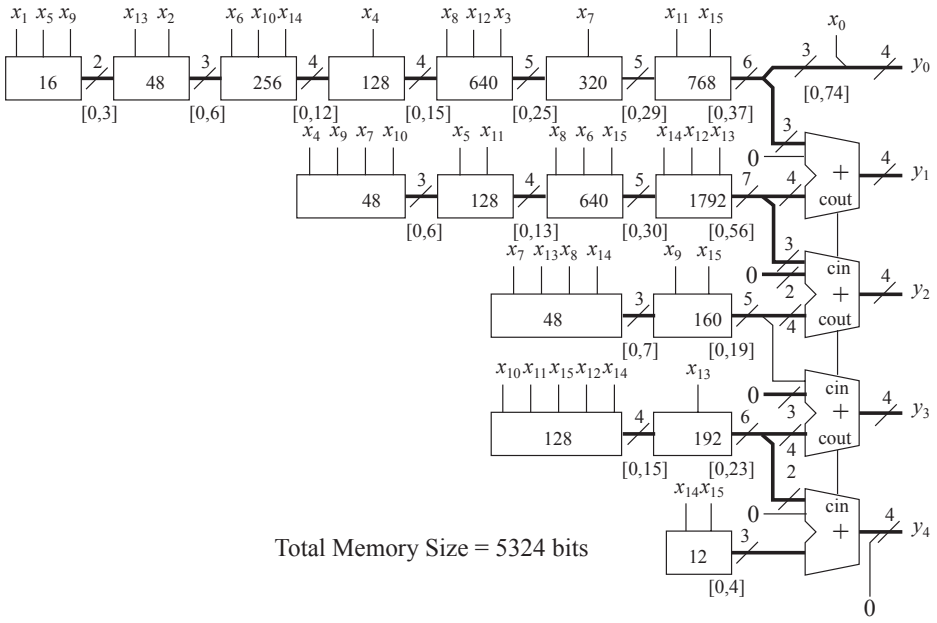

Figure 4.7. Realization of 16-digit Binary to Decimal Converter using LUT cascades and BCD adders

Algorithm 4.1 *(Merge of LUTs)*

1. *Obtain the LUT cascade for the WS function where each LUT_i has only one external input x_i, $(i = 0, 1, \ldots, s-1)$.*
2. *Apply Lemma 4.1 to the LUT cascade repeatedly.*
3. *Apply Lemma 4.2 to the LUT cascade repeatedly.*

Theorem 4.2 *Algorithm 4.1 produces the LUT cascade with the minimum memory size.*
(Proof: Omitted)

Offen, we need the LUT cascade with fewer levels by increasing the amount of memory. This can be done as follows: Let s be the number of LUTs in the LUT cascade

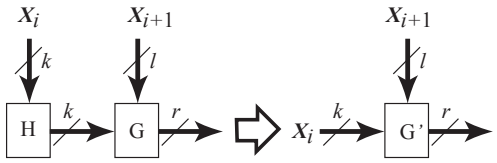

Figure 4.5. Mergeable LUTs.

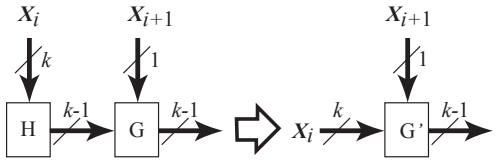

Figure 4.6. Mergeable LUTs with Same Size.

obtained by Algorithm 4.1. The number of combinations to merge adjacent LUTs or not is 2^{s-1}. Among these cascades, select the LUT cascade that satisfies required specification.

Example 4.2 *Figure 4.2 shows the design process of the LUT cascade for the equation (4.5) by using Algorithm 4.1. The integer in each LUT in (e) and (f) denotes the size of the memory.*

- *Figure 4.2(a) shows the LUT cascade for the equation (4.5), where each LUT has only one external input x_i. The total memory size is $6,778$, and the number of levels is 16.*
- *In the binary to decimal conversion, the input x_0 directly represents the output $BIT(y_0, 0)$ [6]. Thus, we can remove the term x_0 from the equation (4.5). Next, factor it by two, and we have the formula shown in (b).*
- *(d) shows the LUT cascade that corresponds to the equation (c). The total memory size is $2,818$, and the number of levels is 15.*
- *(e) shows the LUT cascade that is obtained by applying Lemma 4.1 to (d) repeatedly. The total memory size is $2,176$, and the number of levels is 9.*
- *(f) shows the LUT cascade that is obtained by applying Lemma 4.2 to (e) repeatedly. The total memory size is $2,176$, and the number of levels is 7.*

(End of Example)

Note that the outputs of the final LUT are represented by the BCD code, since they are inputs of BCD adders.

Example 4.3 *By applying Algorithm 4.1 to equations (4.1) – (4.5), we have LUT cascades. Then, we add the outputs by BCD adders. Figure 4.7 is the resulting 16-digit binary to decimal converter, where the total memory size is $5,324$ and uses four decimal adders. Note that the circuit in Fig. 2.4 requires $8,216$ bits, and uses five BCD adders.*

(End of Example)

4.2. Realization Using Arithmetic Decompositions of WS Functions

To further reduce the size of memory, we can use arithmetic decompositions.

Theorem 4.3 *A WS function can be represented as a sum of two WS functions as follows:*

$$WS(\vec{x}) = \sum_{i=0}^{n-1} w_i x_i = \alpha WS_A(\vec{x}) + WS_B(\vec{x}), \quad (4.7)$$

where $WS_A(\vec{x}) = \sum_{i=0}^{n-1} a_i x_i$, $WS_B(\vec{x}) = \sum_{i=0}^{n-1} b_i x_i$, and α is an integer. This is the **arithmetic decomposition**, *where α is the* **decomposition coefficient**.

α can be an arbitrary integer, where $1 \leq \alpha < \sum_{i=0}^{n-1} w_i x_i$. In this part, we consider only the case, where $\alpha = 1$, $X_A \cap X_B = \Phi$, and $X_A \cup X_B = X$.

From here, we will consider to reduce the total amount of memory of LUT cascades by arithmetic decompositions. When we realize a binary to decimal converter by LUT cascades and BCD adders, the last LUTs have to represent BCD codes. That is, the last LUT need to have the decoding function. Thus, we will decompose the LUT cascade except for the last LUT by a binary adder.

When the numbers of bits in two inputs of a binary adder are balanced, we can use a smaller adder. So, we try to partition the input variables so that the sum of weight coefficients will balance.

Algorithm 4.2 *(Partition of the inputs into two for arithmetic decomposition)*

1. $X_A \leftarrow \phi$; $X_B \leftarrow \phi$; $IN \leftarrow \{w_0, w_1, w_2, \ldots, w_{n-1}\}$;
2. *If* $IN = \phi$ *then goto step (7);*
3. *Let* w_i *be the minimum element in* IN;
4. $IN \leftarrow IN - \{w_i\}$;
5. *If* $(|X_A| \leq |X_B|)$, *then* $X_A \leftarrow X_A + \{x_i\}$, *else* $X_B \leftarrow X_B + \{x_i\}$;
6. *Go to step (2);*
7. *Sets* X_A *and* X_B *represent a partition of the inputs.*

Example 4.4 *Figure 4.2(g) is obtained by the arithmetic decomposition of (f). The total memory size is $1,344$ bits, and it uses a 4-bit binary adder.* (End of Example)

Example 4.5 *Figure 4.8 shows the realization of 16bin2dec by applying Algorithm 4.2 to Fig. 4.7. It uses $3,788$ bits, three binary adders, and four decimal adders.* (End of Example)

Note that Algorithm 4.2 produced a balanced decomposition.

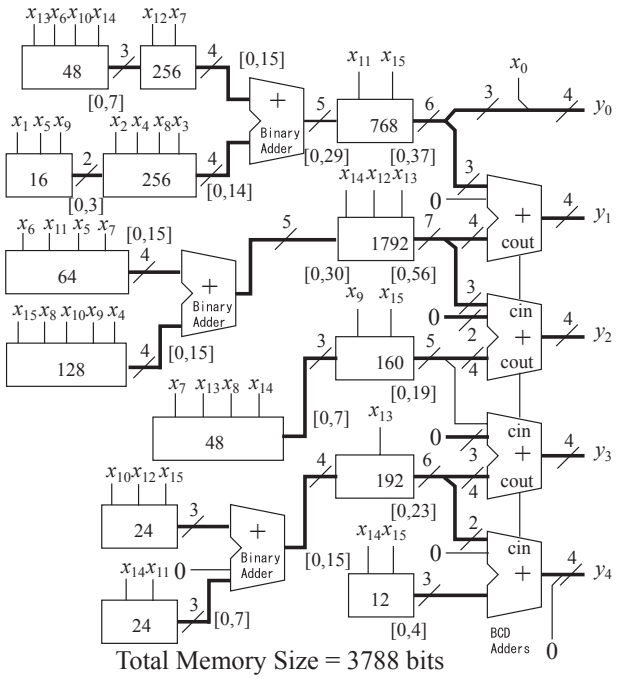

Figure 4.8. 16-digit binary to decimal converter using LUT cascades, binary adders, and BCD adders.

5. Conclusion

In this paper, we presented design methods of p-nary to q-nary converters. For readability, we showed the concept by using the examples for $p = 2$ and $q = 10$. However, in Table 4.1, by replacing 2^i by p^i in the top row and 10^j by q^j in the leftmost column, the method can be easily modified to any radix converters. In this case, a p-nary to q-nary converters are implemented by using LUT cascades, binary adders, and q-nary adders.

6. Acknowledgements

This research is supported in part by the Grant in Aid for Scientific Research of MEXT, the Kitakyushu Area Innovative Cluster Project of MEXT, and Special Research of Meiji University.

References

[1] T. Hanyu and M. Kameyama, "A 200 MHz pipelined multiplier using 1.5 V-supply multiplevalued MOS current-mode circuits with dual-rail source-coupled logic," *IEEE Journal of Solid-State Circuits 30, 11*, (1995), 1239-1245.

[2] C. H. Huang, "A fully parallel mixed-radix conversion algorithm for residue number applications," *IEEE Trans. Comput.*, vol. 32, pp. 398-402, 1983.

[3] Y. Iguchi, T. Sasao, and M. Matsuura, "On design of radix converters using arithmetic decompositions," *36th International Symposium on Multiple-Valued Logic*, Singapore, May 17-20, 2006, p. 3

[4] K. Ishida, N. Homma, T. Aoki, and T. Higuchi, "Design and verification of parallel multipliers using arithmetic description language: ARITH," *34th International Symposium on Multiple-Valued Logic*, Toronto, Canada, May 2004, pp.334-339.

[5] I. Koren, *Computer Arithmetic Algorithms, 2nd Edition*, A. K. Peters, Natick, MA, 2002.

[6] S. Muroga, *VLSI System Design*, John Wiley & Sons, 1982, pp. 293-306

[7] D. Olson, and K. W. Current, "Hardware implementation of supplementary symmetrical logic circuit structure concepts," *30th IEEE International Symposium on Multiple-Valued Logic* Portland, Oregon, May 23-25, 2000.

[8] T. Sasao, *Switching Theory for Logic Synthesis*, Kluwer Academic Publishers, 1999.

[9] T. Sasao, M. Matsuura, and Y. Iguchi, "A cascade realization of multiple-output function for reconfigurable hardware," *International Workshop on Logic and Synthesis(IWLS01),* Lake Tahoe, CA, June 12-15, 2001, pp. 225-230.

[10] T. Sasao, "Radix converters: Complexity and implementation by LUT cascades," *35th International Symposium on Multiple-Valued Logic*, Calgary, Canada, May 19-21, 2005, pp.256-263.

[11] T. Sasao, "Analysis and synthesis of weighted-sum functions," *International Workshop on Logic and Synthesis*, Lake Arrowhead, CA, USA, June 8-10, 2005, pp.455-462.

[12] T. Sasao, Y. Iguchi, and T. Suzuki, "On LUT cascade realizations of FIR filters," *DSD2005, 8th Euromicro Conference on Digital System Design: Architectures, Methods and Tools*, Porto, Portugal, Aug. 30 - Sept. 3, 2005, pp.467-474.

[13] T. Sasao, "Analysis and synthesis of weighted-sum functions," *IEEE Trans. on CAD*, Vol. 25, No. 5, May 2006, pp. 789-796..