# Design of Address Generators Using Multiple LUT Cascade on FPGA

Hui Qin and Tsutomu Sasao
Department of Computer Science and Electronics,
Kyushu Institute of Technology
680–4, Kawazu, Iizuka, Fukuoka, 820–8502, Japan

**Abstract— This paper presents multiple LUT cascade to realize an address generator that produces unique addresses ranging from 1 to $k$ for $k$ distinct input vectors. We implemented six kinds of address generators using multiple LUT cascades, Xilinx's CAM (Xilinx IP core), and an address generator using registers and gates on Xilinx Spartan-3 FPGAs. One of our implementations has 76% more throughput, 29.5 times more throughput/slice, and 1.35 times more throughput/memory than Xilinx's CAM.**

## I. Introduction

An ordinary memory produces an output data value given an input address value. On the contrary, an address generator produces an output address value corresponding to the applied input data value. If the input data value is not stored, a special output address is produced (e.g. 0). It is assumed that any data value is stored at exactly one address. Therefore, an input data value produces a unique output address value. The address generator has a broad range of applications, including data compression, routing tables in the internet [1], network switches, and dictionary searching. Although an address generator can be implemented by a content addressable memory (CAM) [2], the CAM dissipates more power than a conventional RAM [3]. Xilinx [4] provides a design for the CAM implemented with block RAMs (BRAMs) of the Xilinx FPGA. Note that Xilinx's CAM is an "IP (intellectual property) core". Another realization of the address generator uses registers and logic gates. In this realization, the interconnections tend to be very complicated.

Recently, Sasao has shown that a multiple-valued input address generator can be realized by an LUT (look-up table) cascade that uses conventional RAMs and gates [7].

In this paper, we propose an extension to an LUT cascade realization for a two-valued input address generator: a *multiple LUT cascade* realization that is easily reconfigured when additional binary vectors are required. In the multiple LUT cascade architecture (Fig. 3), the inputs of each LUT cascade are common with other LUT cascades, and the outputs of each LUT cascade are connected to an encoder. The LUT cascades are used to realize address generation functions and the encoder is used to generate the index from the outputs of cascades. Since both Xilinx's CAM and the multiple LUT cascade use BRAMs, it is interesting to compare the multiple LUT cascade with Xilinx's CAM on the same FPGA. Our basis for comparison is address generation functions with 48 input-variables and 60∼63 registered vectors implemented on a Xilinx Spartan-3 FPGA. First, by using the multiple LUT cascade, we designed six address generators: $r3p12$, $r4p12$, $r4p12or$, $r5p11$, $r5p11or$ and $r6p11$. Then, we used the Xilinx Core Generator tool to produce a CAM. Finally, by using registers and logic gates, we implemented an address generator called Reg-Gate. Reg-Gate has the smallest delay, but requires many slices. In terms of the equivalent throughput, Reg-Gate is lower than other implementations. The multiple LUT cascade produces higher throughput, higher throughput/slice, and higher throughput/memory than Xilinx's CAM. $r6p11$ has 76% more throughput, 29.5 times more throughput/slice, 1.35 times more throughput/memory than Xilinx's CAM. In addition, if the area for one BRAM is less than or equal to the area for 64 slices, $r6p11$ is more efficient than Xilinx's CAM in terms of $delay - area$ product, although it has 97% more delay than Xilinx's CAM.

The rest of the paper is organized as follows: Section 2 describes the multiple LUT cascade. Section 3 shows other realizations for the address generator. Section 4 presents the implementations of the address generator using an FPGA. Section 5 shows the experimental results. And finally, Section 6 concludes the paper.

## II. Multiple LUT Cascade

### A. Address Generators

**Definition 2.1** *Let* $\{\vec{a_1}, \vec{a_2}, ..., \vec{a_k}\}$ *be a set of distinct binary vectors of n bits. An* $n$*-input* **address generation function** *is a mapping* $F(\vec{x}) : \{0,1\}^n \rightarrow \{0, 1, ..., k\}$*, where*

$$F(\vec{x_i}) = \begin{cases} i & \text{if } \vec{x_i} = \vec{a_i} \\ 0 & \text{otherwise.} \end{cases}$$

$k$ *is the* **weight** *of the function (the number of non-zero output values).* $\vec{a_i}$ *is a registered vector. That is, F produces an address ranging from 1 to k for the registered vectors, and produces 0 for all other* ($2^n - k$) *input vectors.*

Typical applications of address generators include:

- Data compression in communications - Used to produce an equivalent, but shorter message. With an address generator, the compressed data represents the same information using fewer bits.
- Routing table in the internet - Used to generate the output of the destination address from the incoming IP (Internet Protocol) address. In the IPv4, an IP address is represented by 32 bits, and the output address is represented by 8 to 16 bits.

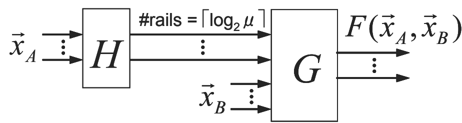



Fig. 1. Decomposition for the function $F$.

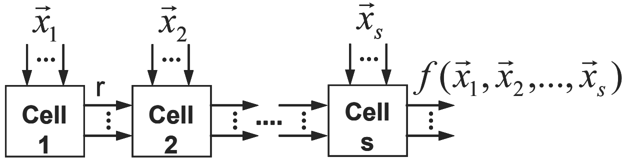



Fig. 2. LUT cascade.

- Network switch - Used to process the address information from the incoming packets. In this application, the output address is usually 8 bits.
- Dictionary searching - Used to generate the indices from the words in a dictionary. In an English word, each alphabetic character can be represented by 5-bit.

In the above applications, the common properties of the address generator include:

- Exactly one element of the domain maps to a non-zero element of the range. Typically many elements of the domain map to 0.
- The number of non-zero outputs is usually much smaller than that of the possible input combinations.
- Data stored in the address generator can be updated.
- A high speed circuit is required.

### B. Address Generators by an LUT Cascade

Functional decomposition [8] is a method where a given function is divided into functions with fewer inputs. For a given function $F(\vec{x})$, let $\vec{x}$ be partitioned as $(\vec{x}_A, \vec{x}_B)$. The decomposition chart of $F$ is a table with $2^{n_A}$ columns and $2^{n_B}$ rows, where $n_A$ and $n_B$ are the number of variables in $\vec{x}_A$ and $\vec{x}_B$, respectively. Each column and row has its unique label with a binary number, and the corresponding element in the table denotes the value of $F$. The column multiplicity, $\mu$, is the number of different column patterns of the decomposition chart. By using functional decomposition, a function $F$ can be decomposed as $F(\vec{x}_A, \vec{x}_B) = G(H(\vec{x}_A), \vec{x}_B)$, as shown in Fig. 1, where the number of rails (signal lines between two blocks $H$ and $G$) is $\lceil \log_2 \mu \rceil$, where $\lceil a \rceil$ denotes the smallest integer greater than or equal to $a$. By iterative functional decompositions, the given function can be realized by an LUT cascade shown in Fig. 2, where each cell consists of a memory [5], [9].

**Theorem 2.1** *[7] An arbitrary $n$-input address generator with weight $k$ can be realized by an LUT cascade, where each cell consists of a memory with $p$ address lines and $r$ outputs. Let $s$ be the necessary number of levels or cells. Then, we have the relation:*

$$s \leq \lceil \frac{n-r}{p-r} \rceil, \tag{1}$$

*where $p > r$ and $r = \lceil \log_2(k+1) \rceil$.*

### C. Address Generators by Multiple LUT Cascade

A single LUT cascade realization often requires many levels. Since the delay is proportional to the number of levels in a cascade, we seek to reduce the number of levels. According to (1), if we increase $p$, then the number of levels $s$ is reduced, but the amount of memory is increased. However, as shown in Fig. 3, we can use the multiple LUT cascade to reduce the levels $s$ when $p$ is fixed. For an $n$-input address generation function with weight $k$, let the number of rails of each LUT cascade be $r$. First, partition the set of vectors into $g = \lceil \frac{k}{2^r-1} \rceil$ groups of $2^r - 1$ vectors each, except the last group, which has $2^r - 1$ or fewer vectors. For each group of the vectors, form an independent address generation function with the same inputs. Then, for each group, realize the corresponding address generator with a LUT cascade. Finally, use a special encoder to produce the correct outputs of the address generator. Let $v_i$ $(i = 1, 2, ..., g)$ be the $i$-th input of the encoder (i.e., $v_i$ is the output value of the $i$-th LUT cascade), and let $v_{out}$ be the output value of the encoder. Then,

$$v_{out} = \begin{cases} 0 & \text{if } v_i = 0 \\ v_i + (i-1)(2^r - 1) & \text{if } v_i \neq 0. \end{cases}$$

**Example 2.1** *For an $n$-input address generation function with weight $k$, for $k = 1000$ and $n = 32$, by Theorem 2.1, we have $r = 10$. Let $p = r + 1 = 11$. When we use a single LUT cascade to realize the function, by Theorem 2.1, we need $\lceil \frac{n-r}{p-r} \rceil = 22$ cells, and the number of levels of the LUT cascade is also 22. Since each cell consists of a memory with 11 address lines and 10 outputs, the total amount of memory is $2^{11} \times 10 \times 22 = 450K$ bits. Note that such cell does not fit in a single block RAM (BRAM) of the Xilinx FPGA, which contains $18K$ bits.*

*However, if we use a multiple LUT cascade to realize the function, we can reduce the number of levels and the total amount of memory, as well as the size of cells to fit in the BRAMs in Xilinx FPGAs. Partition the set of vectors into two groups, and realize each group independently. Then, we need two LUT cascades. For each LUT cascade, the number of vectors is 500, so $r = 9$. Also, let $p = r + 2 = 11$. Then, we need $\lceil \frac{n-r}{p-r} \rceil = 12$ cells in each cascade. Note that the number of levels of the LUT cascades is 12 and is smaller than the single LUT cascade realization. Since each cell consists of a memory with 9 outputs and at most 11 address lines, the total amount of memory is at most $2^{11} \times 9 \times 12 \times 2 = 442K$ bits. Also, note that the size of the memory for a cell is $2^{11} \times 9 = 18K$ bits. This just fits the BRAMs of Xilinx FPGAs.*

*Thus, the multiple LUT cascade not only reduces the number of levels and the total amount of memory, but also adjusts the size of cells to fit into the available memory in the FPGAs.*
*(End of Example)*

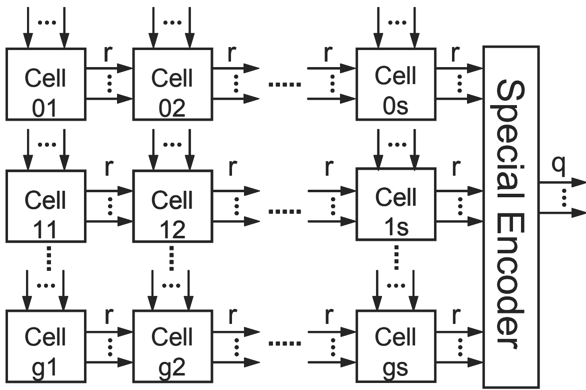



Fig. 3. Multiple LUT cascade architecture.

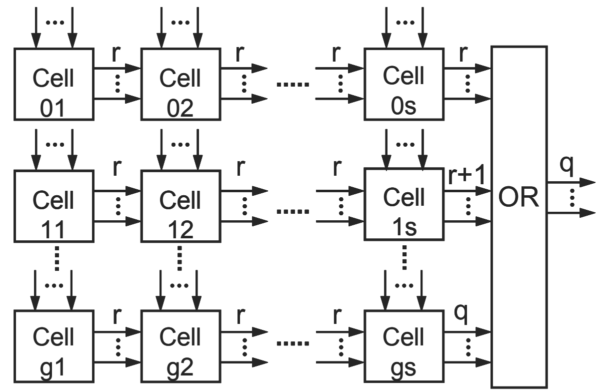



Fig. 5. Multiple LUT cascade OR architecture.

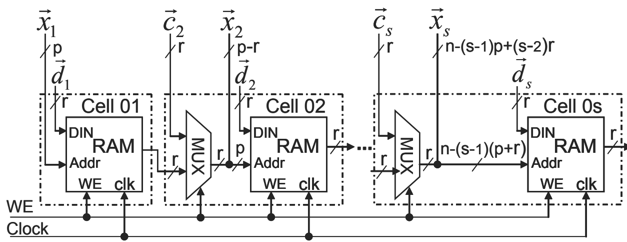



Fig. 4. Detailed design of the LUT cascade.

TABLE I
TRUTH TABLE FOR A 6-INPUT ADDRESS GENERATION FUNCTION

| Inputs | | | | | | Outputs | | |
|---|---|---|---|---|---|---|---|---|
| $x_1$ | $x_2$ | $x_3$ | $x_4$ | $x_5$ | $x_6$ | $out_2$ | $out_1$ | $out_0$ |
| 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 |
| 0 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 0 |
| 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 1 |
| 0 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 0 |
| 0 | 0 | 1 | 0 | 1 | 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 |
| Other values | | | | | | 0 | 0 | 0 |

The **multiple LUT cascade architecture** is shown in Fig. 3, and the realization with this architecture is called the **multiple LUT cascade realization**. It consists of a group of LUT cascades and a special encoder. The inputs of each LUT cascade are common with other LUT cascades, while the outputs of each LUT cascade are connected to the encoder. The LUT cascades are used to realize address generation functions, while the encoder is used to generate the index from the outputs of cascades.

For an $n$-input address generation function with weight $k$, the detailed design of the LUT cascade in the multiple LUT cascade is shown in Fig. 4, where $\vec{x}_i$ $(i = 1, 2, ..., s)$ denotes the primary inputs to the $i$-th cell, $\vec{d}_i$ $(i = 1, 2, ..., s)$ denotes the data inputs to the $i$-th cell and provides the data value to be written in the RAM of the $i$-th cell, $r$ denotes the number of rails and $r \leq \lceil \log_2(k + 1) \rceil$, $\vec{c}_j$ $(j = 2, 3, ..., s)$ denotes the additional inputs to the $j$-th cell and is used to select the RAM location along with $\vec{x}_j$ for write access. Note that $\vec{c}_j$ and $\vec{d}_i$ is represented by $r$ bits, and the RAMs except for the last one have $p$ address lines, but the last RAM has at most $p$ address lines. When $WE$ is high, the $\vec{c}_j$ is connected to the RAM to write the data into the RAMs. When $WE$ is low, the outputs of the RAMs are connected to the inputs of the succeeding RAMs, and the circuit works as a cascade to realize the function.

When the number of groups of the LUT cascades is small, we can use the **multiple LUT cascade OR architecture** to simplify the encoder, as shown in Fig. 5. The realization with this architecture is the **multiple LUT cascade OR realization**. In these two architectures, only the last cells are different. Note that in Fig. 5, the last cell in the LUT cascades except for the first row has more outputs than that in Fig. 3. This is because that the $i$-th LUT cascade produces the values ranging from $(i - 1)(2^r - 1)$ to $i(2^r - 1)$, where $i = (1, 2, ..., g)$, $r$ is the number of rails, and $g$ is the number of groups of the LUT cascades. In this case, we can use OR gates instead of the encoder. Note that the total amount of memory in the multiple LUT cascade OR architecture is larger than the multiple LUT cascade architecture. However, in FPGA implementation, if the memory size of the last cell is smaller than the embedded memory size of the FPGA, the multiple LUT cascade OR architecture is a good choice since it is faster than the corresponding multiple LUT cascade realization.

**Example 2.2** *Table I shows a 6-input address generation function with 6 registered vectors (weight 6).*

**Single Memory Realization:** *The number of address lines is 6 and the number of outputs is 3. Thus, the total amount of memory is* $2^6 \times 3 = 192$ *bits.*

**Single LUT Cascade Realization:** *Since the weight of the function is* $k = 6$*, by Theorem 2.1, the number of rails is* $r = \lceil \log_2 (6 + 1) \rceil = 3$*. Let the number of address lines for the memory in a cell be* $p = 4$*. By partitioning the inputs into three disjoint sets* $\{x_1, x_2, x_3, x_4\}$*,* $\{x_5\}$*, and* $\{x_6\}$*, we have a cascade in Fig. 6, where the additional inputs to the cells are ignored.*

*The total amount of memory is* $2^4 \times 3 \times 3 = 144$ *bits, and the number of levels is* $s = 3$*. Note that it requires smaller amount of memory than the single memory realization.*

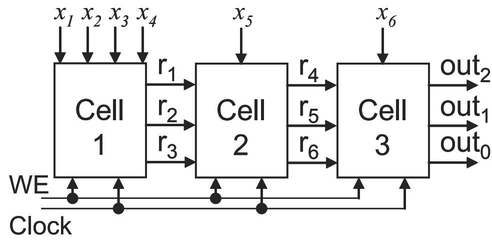
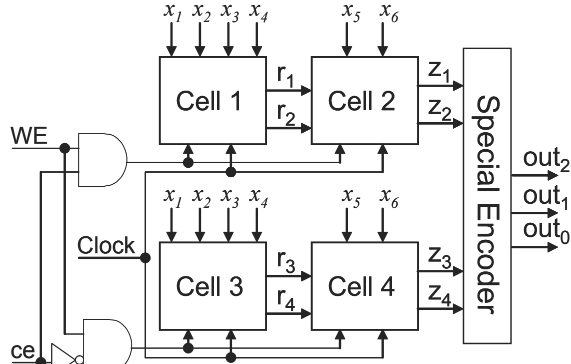



Fig. 6. Single LUT cascade realization.



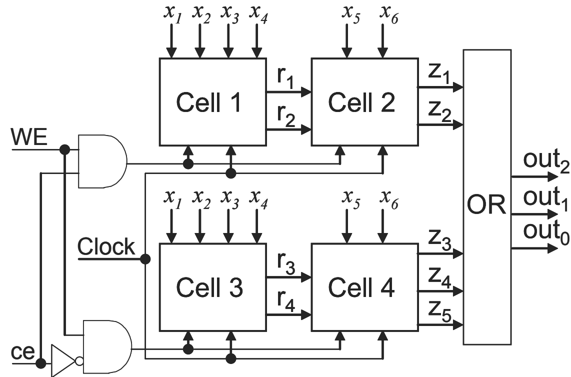




Fig. 7. Multiple LUT cascade realization and multiple LUT cascade OR realization.

**Multiple LUT cascade Realization:** *Partition Table I into two parts. This yields two functions. Since the weight of each function is 3, the number of rails is* $\lceil \log_2(3+1) \rceil = 2$*. Thus, the number of groups of LUT cascades is* $\lceil \frac{6}{2^2-1} \rceil = 2$*. Let the number of address lines for the memory in a cell also be 4. By partitioning the inputs into two disjoint sets* $\{x_1, x_2, x_3, x_4\}$ *and* $\{x_5, x_6\}$*, we obtain the realization in Fig. 7 (a), where the additional inputs to the cells are ignored. The upper LUT cascade realizes the upper part of the Table I, while the lower LUT cascade realizes the lower part of the Table I. The contents of each cell is shown in Table II. The encoder generates the index* $(out_2, out_1, out_0)$ *from the pair of outputs,* $(z_1, z_2)$ *and* $(z_3, z_4)$ :

TABLE II
TRUTH TABLES FOR THE CELLS IN THE MULTIPLE LUT CASCADE REALIZATION

| $x_1$ | $x_2$ | $x_3$ | $x_4$ | $r_1$ | $r_2$ | $x_5$ | $x_6$ | $z_1$ | $z_2$ |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 |
| 0 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 1 | 0 |
| 0 | 1 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 1 |
| Other values | | | | 1 | 1 | × | × | 0 | 0 |
| | | | | Other values | | | | 0 | 0 |

| $x_1$ | $x_2$ | $x_3$ | $x_4$ | $r_3$ | $r_4$ | $x_5$ | $x_6$ | $z_3$ | $z_4$ |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 1 | 0 | 0 | 1 | 1 | 0 | 1 |
| 0 | 0 | 1 | 0 | 0 | 1 | 1 | 1 | 1 | 0 |
| 1 | 1 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 1 |
| Other values | | | | 1 | 1 | × | × | 0 | 0 |
| | | | | Other values | | | | 0 | 0 |

TABLE III
TRUTH TABLES FOR CELL 4 IN THE MULTIPLE LUT CASCADE OR REALIZATION

| $r_3$ | $r_4$ | $x_5$ | $x_6$ | $z_3$ | $z_4$ | $z_5$ |
|---|---|---|---|---|---|---|
| 0 | 0 | 1 | 1 | 1 | 0 | 0 |
| 0 | 1 | 1 | 1 | 1 | 0 | 1 |
| 1 | 0 | 1 | 1 | 1 | 1 | 0 |
| Other values | | | | 0 | 0 | 0 |

$$
\begin{aligned}
out_2 &= z_3 \vee z_4, \\
out_1 &= z_1 \vee z_3 z_4, \\
out_0 &= z_2 \vee z_3 \bar{z}_4.
\end{aligned}
$$

*The total amount of memory is* $2^4 \times 2 \times 4 = 128$ *bits, and the number of levels is 2. Note that the multiple LUT cascade realization uses less memory and fewer levels than the single LUT cascade realization.*

**Multiple LUT cascade OR Realization:** *The design method is similar to the multiple LUT cascade realization. Fig. 7 (b) shows the architecture, where the additional inputs to the cells are ignored. Table III shows the contents of the cell 4 only, since the contents of the other cells are the same as the multiple LUT cascade realization. The output part is simple, i.e.,* $out_2 = z_3$*,* $out_1 = z_1 \vee z_4$*,* $out_0 = z_2 \vee z_5$*. However, the total amount of memory is* $2^4 \times 2 \times 3 + 2^4 \times 3 = 144$ *bits that is larger than the multiple LUT cascade realization. (End of Example)*

## III. OTHER REALIZATIONS

### A. Xilinx's CAM

Xilinx Core Generator system [10] provides two special designs for the CAM. One design only uses the slices called SRL16 Implementation. The other design uses block RAMs called Block SelectRAM Implementation. Both designs for the CAMs

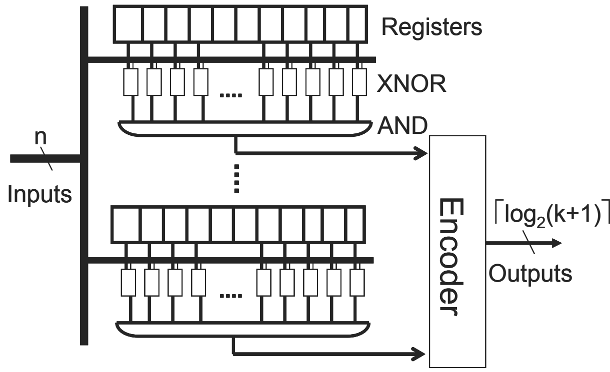



Fig. 8. Realize the address generator with registers and gates.

TABLE IV
FOUR REALIZATIONS USING MULTIPLE LUT CASCADE

| Design | **Vectors** | **r** | **p** | **Groups** | **Levels** |
|---|---|---|---|---|---|
| $r3p12$ | 63 | 3 | 12 | 9 | 5 |
| $r4p12$ | 60 | 4 | 12 | 4 | 6 |
| $r4p12OR$ | 60 | 4 | 12 | 4 | 6 |
| $r5p11$ | 62 | 5 | 11 | 2 | 8 |
| $r5p11OR$ | 62 | 5 | 11 | 2 | 8 |
| $r6p11$ | 63 | 6 | 11 | 1 | 9 |

r: Number of rails
p: Number of inputs of the RAM in a cell
Groups: Number of LUT cascades

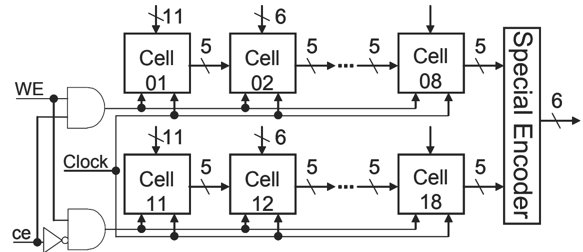



Fig. 9. Architecture for $r5p11$.

are parameterizable IP (intellectual property) cores. The SRL16 Implementation requires no BRAMs, but requires a longer clock cycle time than the Block SelectRAM Implementation [11]. The Block SelectRAM Implementation uses the block RAM as a dual-port RAM [6], while the multiple LUT cascade uses the block RAM just as a single-port RAM. It is interesting to compare the multiple LUT cascade with the Xilinx's CAM implemented by Block SelectRAM Implementation on the same FPGA. We use the following parameters provided by the Xilinx Core Generator to customize the core for the CAM:

- *Block SelectRAM Implementation*.
- *Depth*- Number of words (vectors) stored in the CAM: $k$.
- *Data width*- Width of the data word (vector) stored in the CAM: $n$.
- *Match Address Type*- Three options: Binary Encoded, Single-match Unencoded, and Multi-match Unencoded. We used the Binary Encoded option.

### B. Registers and Gates

The realization in Fig 8 directly implements the address generator by registers and gates. The registers store the registered vectors. The exclusive NOR (XNOR) gates and the AND gate form an equivalence circuit whose output is 1 iff the stored vector is identical to the input vector.

For an $n$-input address generator with one registered vector, we need an $n$-bit register, $n$ copies of XNOR gates, and an $n$-input AND gate. For an $n$-input address generator with $k$ registered vectors, we need $k$ copies of $n$-bit registers, $nk$ copies of XNOR gates, and $k$ copies of $n$-input AND gates. In addition, we need an encoder with $k$ inputs and $\lceil \log_2 (k+1) \rceil$ outputs to generate the output address. This circuit can be considered as a special case of the multiple LUT cascade architecture, where $r = 1$, $p = 2$, and $g = k$.

## IV. FPGA IMPLEMENTATIONS

We implemented the address generators with 48 inputs and 60∼63 registered vectors on a Xilinx Spartan-3 FPGA (Xc3s4000-5) using the multiple LUT cascade, Xilinx Core Generator, and registers & gates. The FPGA device Xc3s4000-5 has 96 BRAMs and 27648 slices. Each BRAM contains 18K bits and each slice consists of two 4-input look-up tables. For each implementation, we described the circuit by Verilog HDL, and then used the Xilinx ISE 7.1i design software to synthesize and place & route.

First, we used the multiple LUT cascade to realize the address generators. To use the BRAMs in the FPGA efficiently, the memory size of a cell in the LUT cascade should not exceed the BRAM size. Let $p$ be number of address lines of the memory in the cell. Since each BRAM contains $2^{11} \times 9$ bits, we have the following relation: $2^p \cdot r \leq 2^{11} \times 9$, where $r$ is the number of rails. Thus, we have the following: $p = \lfloor \log_2 (9/r) \rfloor + 11$, where $\lfloor a \rfloor$ denotes the largest integer less than or equal to $a$.

We designed six kinds of address generators $r3p12$, $r4p12$, $r4p12or$, $r5p11$, $r5p11or$ and $r6p11$ as shown in Table IV, where the column **Vector** denotes the number of registered vectors, the column **r** denotes the number of rails, the column **p** denotes the number of inputs of the RAM in a cell, the column **Groups** denotes the number of LUT cascades, and the column **Levels** denotes the number of levels or cells in the LUT cascade. Among these six designs, $r4p12OR$ and $r5p11OR$ are multiple LUT cascade OR realizations and the others are multiple LUT cascade realizations. Note that $r3p12$ contains 9 groups, and the RAM size of each cell is $2^{12} \times 3$ = 12K bits. We can estimate that if we use the multiple LUT cascade OR architecture for $r3p12$, we need 7 more BRAMs. Thus, $r3p12$ is unsuitable for the multiple LUT cascade OR architecture.

To illustrate Table IV, consider the design of $r5p11$ in Fig 9, where the additional inputs to the cells are ignored. For $r5p11$,

TABLE V
COMPARISONS OF FPGA IMPLEMENTATIONS OF THE ADDRESS GENERATOR

| Design | **Levels** | **Slices** | **Memory** (BRAM) | **F_clk** (MHz) | **tco/tpd** (ns) | **Th.** (Mbps) | **Th./Slice** ($\frac{\text{Mbps}}{\text{slice}}$) | **Th./Memory** ($\frac{\text{Mbps}}{\text{BRAM}}$) | **Delay** (ns) |
|---|---|---|---|---|---|---|---|---|---|
| $r3p12$ (multiple) | 5 | 87 | 45 | 107.227 | 21.831 (tco) | 643 | 7.37 | 14.30 | 68.461 |
| $r4p12$ (multiple) | 6 | 58 | 24 | 102.638 | 19.301 (tco) | 616 | 10.62 | 25.66 | 77.759 |
| $r4p12OR$ (multiple) | 6 | 48 | 24 | 110.205 | 18.203 (tco) | 661 | 13.78 | 27.55 | 72.647 |
| $r5p11$ (multiple) | 8 | 42 | 16 | 127.502 | 19.835 (tco) | 765 | 18.21 | 47.81 | 82.579 |
| $r5p11OR$ (multiple) | 8 | 41 | 16 | 136.147 | 19.429 (tco) | **817** (best) | 19.92 | 51.06 | 78.189 |
| $r6p11$ (multiple) | 9 | 24 | 9 | 131.822 | 15.142 (tco) | 791 | **32.96** (best) | **87.88** (best) | 83.416 |
| Xilinx's CAM | 1 | 414 | 12 | 74.811 | 29.011 (tco) | 449 | 1.08 | 37.41 | 42.378 |
| Reg-Gate | | 3016 | | | 36.529 (tpd) | | | | **36.529** (best) |

since the number of rails is $r = 5$, the number of groups is $\lceil \frac{62}{2^5-1} \rceil = 2$. Thus, we need two LUT cascades and an encoder. Since each LUT cascade consists of 8 cells, the levels of $r5p11$ is 8. To efficiently use BRAMs in the FPGA, the number of inputs of the RAM in the cell is $p = \lfloor \log_2 (9/5) \rfloor + 11 = 11$. Note that the number of the primary inputs to the last cell is 1; this is because $48 - 11 - 6 \times 6 = 1$. Although the RAM in the last cell has 6 inputs, it still requires one BRAM. As for the special encoder, let $v_1$ be the values of the outputs of the upper LUT cascade, let $v_2$ be the values of the outputs of the lower LUT cascade, and let $v_{out}$ be the values of the outputs of the encoder. Then,

$$v_{out} = \begin{cases} v_1 & \text{if } v_1 \neq 0 \\ v_2 + 31 & \text{if } v_2 \neq 0. \end{cases}$$

The special encoder requires 6 slices from the FPGA. After synthesizing and mapping, $r5p11$ required 16 BRAMs and 42 slices.

From Table IV, we can see that decreasing $r$, increases the groups needed to implement the function, but decreases the levels in the cascade.

Next, we used the Xilinx Core Generator system to produce a **Xilinx's CAM** with 48 inputs and 63 registered vectors. After synthesizing and mapping, Xilinx's CAM required 12 BRAMs and 414 slices. Note that Xilinx's CAM requires one clock cycle to find a match.

Finally, we designed the address generator **Reg-Gate** by using registers and gates shown in Fig 8. Note that the number of inputs is $n = 48$ and the number of outputs is $\lceil \log_2 (k+1) \rceil = \lceil \log_2 (63+1) \rceil = 6$. After synthesizing and mapping, it required 3016 slices.

## V. PERFORMANCE AND COMPARISONS

In this section, we evaluate the performance of the multiple LUT cascade realizations and the multiple LUT cascade OR realizations (i.e., $r3p12$, $r4p12$, $r4p12or$, $r5p11$, $r5p11or$ and $r6p11$), and compare them with Xilinx's CAM and Reg-Gate.

In Table V, the column **Levels** denotes the number of levels, the column **Slices** denotes the number of occupied slices, the column **Memory** denotes the amount of utilized memory, the column **F_clk** denotes the maximum clock rate, the column **tco** denotes the maximum time to obtain the outputs after clock, and the column **tpd** denotes the maximum propagation time from the inputs to the outputs. The column **Th.** denotes the maximum throughput. Since the address generator has 6 outputs, it is calculated by:

$$\text{Th.} = 6 \cdot \text{F_clk}.$$

For Reg-Gate, **Delay** denotes the maximum delay from the input to the output and is equal to **tpd**. For the multiple LUT cascade realizations, the multiple LUT cascade OR realizations and Xilinx's CAM, **Delay** denotes the total delay, and is calculated by:

$$\text{Delay} = \frac{1000 \cdot \text{Levels}}{\text{F_clk}} + tco,$$

where 1000 is a unit conversion factor. The column **Th./Slice** denotes the throughput per slice, and the column **Th./Memory** denotes the throughput per memory.

In Table V, the value denoted with *best* shows the best result. Reg-Gate has the smallest delay, but requires many slices. Note that Reg-Gate requires no clock pulses in the address generation operation, while the others are sequential circuits requiring at least one clock pulse. Since delay of Reg-Gate is 36.529 ns, the equivalent throughput is $(1000/36.529) \times 6 = 164.25$ (Mbps); this is lower than all others.

$r5p11OR$ has the highest throughput in Table V. All of the multiple LUT cascade realizations, and the multiple LUT cascade OR realizations have higher throughput, and higher throughput/slice than Xilinx's CAM. In terms of throughput/memory, $r5p11$ and $r6p11$ are better than Xilinx's CAM. $r3p12$ has the smallest delay in the realizations using the multiple LUT cascade, but is slower than Xilinx's CAM.

$r6p11$ has 76% more throughput, 29.5 times more throughput/slice, 1.35 times more throughput/memory, but 97% more

delay than Xilinx's CAM. It is interesting to consider the relation of delay with the area for both the utilized slices and the utilized BRAMs. Let $\alpha$ be an area factor between the BRAM and the slice, i.e., (area for one BRAM) = $\alpha$ (area for one slice). If $\alpha \leq 64$, then $r6p11$ is more efficient than Xilinx's CAM in terms of $delay - area$ product.

## VI. Conclusions

In this paper, we presented the multiple LUT cascade to realize address generators. We illustrated the design methods by address generators with $n = 48$ and $k = 63$. However, it can be extended to any value of $n$ and $k$.

We implemented six kinds of address generators (i.e. $r3p12$, $r4p12$, $r4p12or$, $r5p11$, $r5p11or$ and $r6p11$) on the Xilinx Spartan-3 FPGA (Xc3s4000-5) by using the multiple LUT cascade. For comparison, we also implemented Xilinx's CAM by using the Xilinx Core Generator and Reg-Gate by using registers and gates on the same type of FPGA. Reg-Gate has the smallest delay, but requires many slices. All of the implementations of the multiple LUT cascade have higher throughput, and higher throughput/slice than Xilinx's CAM. In terms of throughput/memory, $r5p11$ and $r6p11$ are better than Xilinx's CAM. $r6p11$ has 76% more throughput, 29.5 times more throughput/slice, 1.35 times more throughput/memory than Xilinx's CAM. In addition, if the area for one BRAM is less than or equal to the area for 64 slices, $r6p11$ is more efficient than Xilinx's CAM in terms of $delay - area$ product although it has 97% more delay than Xilinx's CAM.

## Acknowledgments

This research is partly supported by the Grant in Aid for Scientific Research of JSPS, MEXT, and the Grant of Kitakyushu area innovative cluster project. Discussion with Prof. J. T. Bulter improved presentation of the paper.

## References

[1] M. A. Ruiz-Sanchez, E. W. Biersack, and W. Dabbous, " Survey and taxonomy of IP lookup algorithms," *IEEE Network,* Vol. 15, No. 2, pp. 8-23, Mar.-Apr. 2001.

[2] F. Shafai, K.J. Schultz, G.F.R. Gibson, A.G. Bluschke, and D.E. Somppi, "Fully parallel 30-MHz, 2.5-Mb CAM," *IEEE Journal of Solid-State Circuits,* Vol. 33, No. 11, pp. 1690-1696, 1998.

[3] K. Pagiamtzis and A. Sheikholeslami, "A low-power content-addressable memory (CAM) using pipelined hierarchical search scheme", *IEEE Journal of Solid-State Circuits,* Vol. 39, No. 9, pp. 1512-1519, 2004.

[4] http://www.xilinx.com

[5] 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.

[6] Xilinx, Inc., "Using block RAM for high-performance read/write CAMs," XAPP204 (v1.2), May 2, 2000, available at http://www.xilinx.com/products/design_resources/mem_corner/resource/internal_cam.htm

[7] T. Sasao, "Design method for multiple-valued input address generators," *International Symposium on Multiple-Valued Logic,* May 2006, Singapore, (to be published).

[8] T. Sasao, *Switching Theory for Logic Synthesis,* Kluwer Academic Publishers, pp. 242-246, 1999.

[9] K. Nakamura, T. Sasao, M. Matsuura, K. Tanaka, K. Yoshizumi, H. Qin, and Y. Iguchi, "Programmable logic device with an 8-stage cascade of 64K-bit asynchronous SRAMs," *Cool Chips VIII, IEEE Symposium on Low-Power and High-Speed Chips,* April 20-22, 2005, Yokohama, Japan.

[10] http://www.xilinx.com/products/design_resources/design_tool/grouping/design_entry.htm

[11] Xilinx, Inc., "Content-Addressable Memory v5.1," DS253, Nov. 11, 2004, available at http://www.xilinx.com/ipcenter/catalog/logicore/docs/cam.pdf