

# Implementation Complexity of Bit Permutation Instructions

Zhijie Jerry Shi and Ruby B. Lee

Department of Electrical Engineering, Princeton University, Princeton, NJ 08544 USA  
{zshi, rblee}@ee.princeton.edu

**Abstract-** Several bit permutation instructions, including GRP, OMFLIP, CROSS, and BFLY, have been proposed recently for efficiently performing arbitrary bit permutations. Previous work has shown that these instructions can accelerate a variety of applications such as block ciphers and sorting algorithms. In this paper, we compare the implementation complexity of these instructions in terms of delay. We use logical effort, a process technology independent method, to estimate the delay of the bit permutation functional units. Our results show that for 64-bit operations, the BFLY instruction is the fastest among these bit permutation instructions; the OMFLIP instruction is next; and the GRP instruction is the slowest.

## I. INTRODUCTION

Bit permutation operations permute the bits in the operand. They are very effective for achieving diffusion in block ciphers [1], where diffusion dissipates the redundancy in the plain text over the encrypted cipher text. Bit permutation operations are used in many ciphers such as the Data Encryption Standard (DES), Twofish and Serpent. However, arbitrary bit permutations are not directly supported on existing microprocessors, and hence very slow. As a result, many ciphers such as RC5 [2] use data-dependent rotation (DDR) instead. DDR uses only  $\log(n)$  bits to specify the shift amount for  $n$ -bit words. This property of DDR has reduced the strength of the ciphers and makes them vulnerable to cryptanalytic attacks [3].

Several instructions have been proposed recently to do arbitrary bit permutation efficiently. They are GRP [4], OMFLIP [5], CROSS [6], and BFLY [7, 8]. Each instruction has its advantages and disadvantages [9]. For example, GRP can accelerate subword sorting [10] and has good cryptographic properties [11]. OMFLIP needs only four stages regardless of how many bits are to be permuted. But these instructions have not been compared with each other in detail in terms of implementation complexity and latency.

In this paper, we compare the implementation complexity of bit permutation instructions in terms of the latency, or delay, of their respective permutation units. Ideally, when a new instruction is added to a processor, the cycle time of the processor should not be significantly impacted. Knowing the relative delays of these permutation functional units is very helpful when deciding which one to include in a given processor. We use a process technology independent method, *logical effort* [12], to compare the delays of different permutation units. Logical effort is a design methodology that

can be used to estimate the number of stages required to implement the critical path of a given logic function, and hence estimate its delay in a process technology independent way.

In Section II, we briefly describe the logical effort methodology. In Section III, we describe the bit permutation instructions and discuss their implementation. In Section IV, we use logical effort to estimate and compare the delay of different permutation circuits. Section V concludes the paper.

## II. LOGICAL EFFORT

Logical effort [12] is a technology-independent method to estimate the number of stages required to implement a given logic function with CMOS and to determine the maximum possible speed of the circuit. It uses the following concepts:

**logical effort**  $g$ : The total gate capacitance of a logic gate relative to that of a minimum-sized inverter

**electrical effort**  $h$ : The ratio of output capacitance of a gate to its input capacitance

**branching effort**  $b$ : The ratio of total capacitive load on one logic gate's output to the gate capacitance of the next gate on the path examined

**parasitic delay**  $p$ : The total diffusion capacitance on the output node of a gate relative to that of a minimum-sized inverter.

The delay of a single gate can be calculated as:

$$d = gh + p \quad (1)$$

To find the delay along a path, we first calculate the total path effort:

$$F = GBH \quad (2)$$

where  $G = \Pi g$ ,  $B = \Pi b$ , and  $H = \Pi h$ .  $\Pi g$  means the product of the logical effort of all the gates along the path. Similarly,  $\Pi b$  is for the total branch effort and  $\Pi h$  for the total electrical effort. The total electrical effort  $H = \Pi h$  reduces to the ratio of the output capacitance loading the last gate to the gate capacitance of the first gate on the path. Normally, we assume a circuit drives a copy of itself, so  $H = 1$ .

Once the path effort has been calculated, the ideal number of stages required to achieve the logical function can be estimated as:

$$N = \log_{3.6} F. \quad (3)$$

where 3.6 is the stage effort achieving the best performance [12].  $N$  is then rounded to the nearest integer that is reasonable for the path, and the effort delay for each stage can

be calculated as:

$$\alpha = F^{1/N} \quad (4)$$

$\alpha$  can be used to decide the transistor size in each stage along the path. The basic idea is to estimate the number of stages using the ideal stage effort  $\alpha=3.6$ , and then calculate the real  $\alpha$  from the estimated number of stages. Finally, the total delay of the path can be calculated as:

$$D = N\alpha + P, \quad (5)$$

where  $P = \sum p$ . The results in (5) are in the basic time unit used in logical effort, which is independent of process technology. Dividing  $D$  in (5) by five gives the estimated delay in terms of fan-out of four (FO4), the delay of an inverter that drives four identical inverters.

### III. BIT PERMUTATION INSTRUCTIONS

We now describe the permutation instructions CROSS, BFLY, OMFLIP and GRP.

#### A. CROSS

The CROSS instruction defined in [6] is based on the Benes network. A Benes network consists of a butterfly network followed by an inverse butterfly network. An  $n$ -bit butterfly network consists of  $\log(n)$  stages. In each stage,  $n$  bits are divided into  $n/2$  pairs. Two bits in a pair can go to the same position at the output or exchange position with the other one. This is determined by a single control bit. So  $n/2$  control bits are needed for  $n/2$  data pairs at each stage. The stages are differentiated by how bits are paired. If we count stages starting from 1, the distance between two paired bits in stage  $i$  is  $n/2^i$ . Figure 1 shows an example of a 16-bit butterfly network. Each small box is like a 2:1 MUX, where one of two bits in a pair is selected. In the first stage, the distance between two paired bits is  $16/2 = 8$ . In the last stage, the distance is one, i.e., two bits are next to each other.

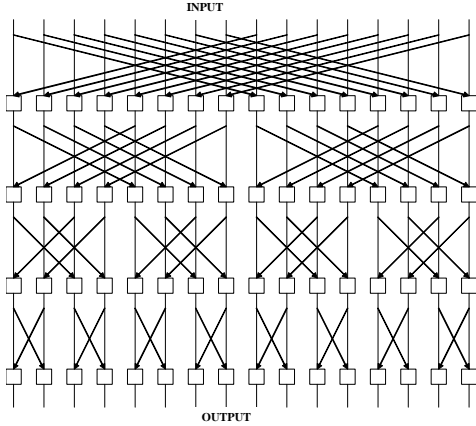


Figure 1: A 16-bit butterfly network

The inverse butterfly network can be constructed by reversing the stages in a butterfly network.

A Benes network is constructed by concatenating a butterfly network with an inverse butterfly network. A CROSS instruction is defined as:

$$\text{CROSS}, m1, m2 \quad R_d, R_s, R_c$$

CROSS permutes the bits in  $R_s$  using any two stages in a Benes network that are specified by  $m1$  and  $m2$ , and stores the permuted bits in  $R_d$ . The two stages specified by  $m1$  and  $m2$  are configured with bits in  $R_c$ ; the lower  $n/2$  bits are used to configure Stage  $m1$ , and higher  $n/2$  bits to configure Stage  $m2$ . A method is given in [6] to configure a Benes network to perform any permutations of the input bits using all stages in a Benes network.  $\log(n)$  CROSS instructions are needed to achieve any one of the  $n!$  permutations of  $n$  bits.

#### B. BFLY

The BFLY instruction is also based on the Benes network. However, BFLY uses the full butterfly network (six stages for 64 bits) to permute input bits while CROSS uses only two stages of the butterfly network or inverse butterfly network per instruction.

To perform arbitrary  $n$ -bit permutations, another instruction IBFLY is required to permute bits with the full inverse butterfly network. In this paper, we focus only on the BFLY instruction. IBFLY will have similar latency as BFLY.

#### C. OMFLIP

The OMFLIP instruction is based on the omega-flip network. A full omega network consists of  $\log(n)$  omega stages, and all omega stages are the same; a full flip network consists of  $\log(n)$  flip stages, and all flip stages are the same. A full omega-flip network, constructed by concatenating a full omega network with a full flip network, is isomorphic to a Benes network. An OMFLIP instruction permutes bits with two stages of the full omega-flip network, and  $\log(n)$  instructions can perform arbitrary  $n$ -bit permutation.

OMFLIP uses only two stages each time, and all omega stages or all flip stages are the same. Hence, only two omega stages and two flip stages are enough to do the OMFLIP instructions. Such a 4-stage network is shown in Figure 2. Unlike CROSS, the number of stages in the functional unit does not depend on the number of bits to be permuted.

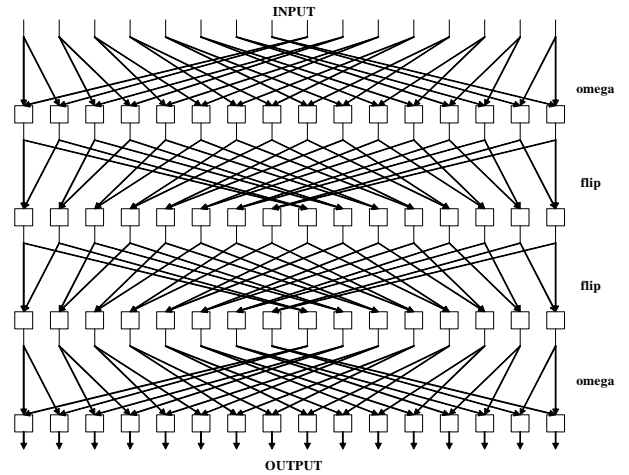


Figure 2: A 4-stage omega-flip network for 16-bit OMFLIP operations

#### D. GRP

The GRP instruction is defined as:

GRP Rd, Rs, Rc

The GRP instruction permutes the data bits in Rs according to the control bits in Rc. The bits in Rs are divided into two groups depending on whether the corresponding bit in Rc is 0 or 1. The two groups of bits are then placed next to each other in Rd. The bits with a control bit of 0 are placed at the left end; the bits with a control bit of 1 at the right end. Figure 3 shows an example of an 8-bit GRP operation. Since the control bit of *b, c, f,* and *h* is 0, these four bits are placed at the left end in Rd. *a, d, e,* and *g* are placed at the right end because their control bit is 1.

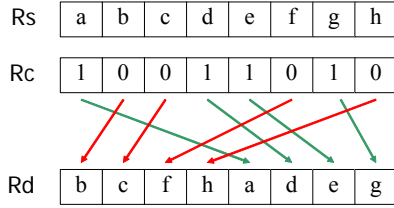


Figure 3: 8-bit GRP operation

There are many ways to implement a GRP operation. Here, we describe a parallel implementation. For convenience, the bits in Rs with a control bit of 0 are referred to as *z bits*, and the bits with control bit of 1 as *w bits*. The GRP operation can be performed in three conceptual steps. Step 1 grabs all *z bits* and sets other bits in the word to 0; Step 2 grabs *w bits* and sets other bits in the word to 0; Step 3 merges the *z bits* and the *w bits* by OR-ing the results generated in the two previous steps. Step 3 is straightforward. And if we can grab *z bits* in Step 1, Step 2 can use the same circuit to grab *w bits* for flipping control bits changes *w bits* to *z bits*.

We use the divide-and-conquer strategy to grab *z bits* in *n* bits, as shown in Figure 4. First, the *n* input bits are divided into two halves. After putting *z bits* at the left end in each half, we combine the *z bits* in both halves, putting all *z bits* at the left end and setting the rest of the bits to 0. For each half of *n/2* bits, we can apply the same method by dividing them into two halves of *n/4* bits. Each set of *n/4* bits can be further divided into smaller sets until every set has only one bit. For sets that has only one bit, the *z bit* is already at the left end if the only bit is a *z bit*. Otherwise, it is set to 0. This can be done with the circuit shown in Figure 5, which we call GRP1Z. In the figure, *i* is the input data bit, and *c* is the corresponding control bit. When *c* = 0, the output *d* = *i* because *i* is a *z bit*. When *c* = 1, *d* is set to 0. ( $k_1, k_0$ ) is one-hot code indicating the number of bits that are set to 0 in (*d*). So  $(k_1, k_0) = (1, 0)$  when *c* = 1 because one bit *d* is set to 0.

A circuit that grabs *z bits* from a *n*-bit set is called GRP<sub>n</sub>Z. GRP1Z is illustrated in Figure 5. GRP2Z consists of two GRP1Zs, and combines their outputs; the circuit that does combination is called GRP2ZD. GRP4Z consists of two GRP2ZDs, and combines their results with a GRP4ZD, and so on. Figure 6 presents a diagram of GRP8ZD, which combines *z bits* from two 4-bit sets. Each small box is the basic cell that is shown in Figure 7. The basic cell has a data input *i*, a data

output *o*, and a select signal *sel*. The output *o* is connected with the input *i* when and only when *sel* = 1. In Figure 6, (*I*<sub>0</sub>, *I*<sub>1</sub>, *I*<sub>2</sub>, *I*<sub>3</sub>) and (*I*<sub>4</sub>, *I*<sub>5</sub>, *I*<sub>6</sub>, *I*<sub>7</sub>) are the outputs of two GRP4Z circuits. In both of them, *z bits* are already placed at the left end and other bits at the right end are set to 0. Those bits that are set to 0 will be referred to as padded 0s. (*S*<sub>4</sub>, *S*<sub>3</sub>, *S*<sub>2</sub>, *S*<sub>1</sub>, *S*<sub>0</sub>) is a one-hot code indicating the number of padded 0s in (*I*<sub>0</sub>, *I*<sub>1</sub>, *I*<sub>2</sub>, *I*<sub>3</sub>). Depending on how many padded 0s are in (*I*<sub>0</sub>, *I*<sub>1</sub>, *I*<sub>2</sub>, *I*<sub>3</sub>), one of (*S*<sub>4</sub>, *S*<sub>3</sub>, *S*<sub>2</sub>, *S*<sub>1</sub>, *S*<sub>0</sub>) is set to 1, and that bit determines at which row the outputs are connected to the inputs. At the output, padded 0s in (*I*<sub>0</sub>, *I*<sub>1</sub>, *I*<sub>2</sub>, *I*<sub>3</sub>) are replaced with bits shifting in from (*I*<sub>4</sub>, *I*<sub>5</sub>, *I*<sub>6</sub>, *I*<sub>7</sub>), and all the *z bits* are located at the left end. For example, when (*I*<sub>0</sub>, *I*<sub>1</sub>, *I*<sub>2</sub>) are *z bits* and *I*<sub>3</sub> is a padded 0, only *S*<sub>1</sub> is set to 1. The inputs and outputs are connected at the second row. The output (*O*<sub>0</sub>, ..., *O*<sub>7</sub>) = (*I*<sub>0</sub>, *I*<sub>1</sub>, *I*<sub>2</sub>, *I*<sub>4</sub>, *I*<sub>5</sub>, *I*<sub>6</sub>, *I*<sub>7</sub>, 0).

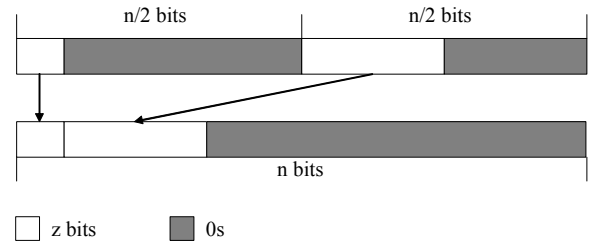


Figure 4: Grab *z bits* recursively

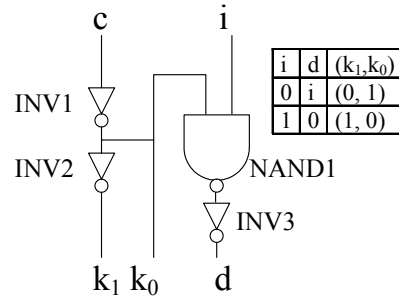


Figure 5: GRP1Z: the first stage in GRP units

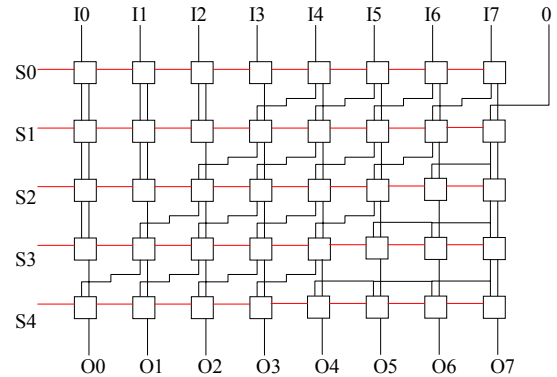


Figure 6: Diagram of GRP8ZD

The circuits generating select signals have a similar structure to that of the data combining circuits shown in Figure 6. These circuits generate the number of padded 0s in

each set of data bits. We call these circuits GRP1ZS, GRP2ZS, GRP4ZS, and so on.

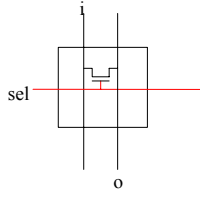


Figure 7: Basic cell

Figure 8 shows the block diagram of the datapath of GRP64, a GRP functional unit for 64 bits. We first use GRP1Z to generate  $z$  bits and  $w$  bits for 1-bit groups. Then, we keep combining the output of smaller sets to generate  $z$  bits and  $w$  bits for a larger set until we get all the  $z$  bits and  $w$  bits for the 64 bits. Then, the  $z$  bits and  $w$  bits are combined with OR gates to get the result of the 64-bit GRP operations.

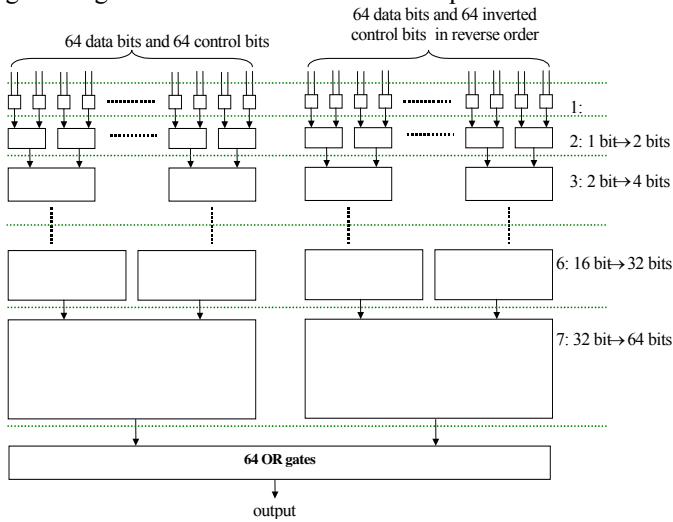


Figure 8: Diagram of GRP64

#### IV. ANALYSIS OF DIFFERENT PERMUTATION CIRCUITS

We now estimate the delay of the 64-bit permutation functional units that performs the BFLY, OMFLIP, and GRP instructions. As mentioned earlier, we assume each permutation unit drives a copy of itself.

In our calculation, only the capacitance of the wires is considered. Wires are converted into a number of inverters, the total input capacitance of which is the same as the capacitance of the wires. We estimate the capacitance of a wire traveling across a cell as equivalent to 1/3 the input capacitance of a minimum-sized inverter [Appendix A].

##### A. BFLY butterfly network latency

In the butterfly network shown in Figure 1, each box can be considered as a 2:1 MUX. In a real implementation, we use 2:1 MUXI shown in Figure 9 instead of MUX. MUXI works similarly to a MUX except that the output of MUXI is inverted. This causes no problem as long as signals are inverted an even number of times.

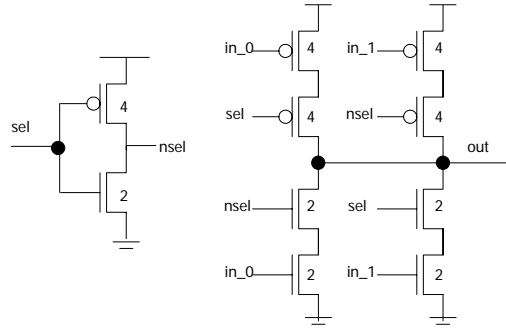


Figure 9: Transistor diagram of a 2:1 MUXI

The numbers in Figure 9 indicate the ratio of the width of transistors to the width of an N-type transistor in a minimum-sized inverter. To achieve the same drive characteristics as a minimum-sized inverter, we double the size of transistors that are connected in series. The parasitic delay of the 2:1 MUXI can be calculated as [12]:

$$p_{muxi} = \left( \frac{\sum w_d}{1+2} \right) p_{inv} = \left( \frac{4+4+2+2}{1+2} \right) p_{inv} = 4p_{inv} \quad (6)$$

The denominator in (6) is the sum of the width of transistors that are connected to the output in a minimum-sized inverter; and the numerator is the sum of width of transistors that connected to the output in the MUXI.

The capacitance of each input is twice that of a minimum-sized inverter. Therefore, the logical effort per data input is 2. The logical effort of the select signal is 4.

The load of the gates in each stage, except for the last stage, consists of wires and MUXIs in the next stage. As mentioned earlier, wires are converted into a number of inverters that have the same capacitance, and then can be modeled as branching effort. Let  $N_{cells}$  be the number of cells that the longest wire travels across in a stage. Since an output in a stage needs to drive the wire and two data inputs of 2:1 MUXIs, we can estimate the branching effort in each stage with the following formula [12]:

$$b = \frac{C_{total}}{C_{useful}} = \frac{C_{wire} + 2C_{muxi}}{C_{muxi}} = \frac{\frac{N_{cells}}{3} + 4}{2} = \frac{N_{cells}}{6} + 2 \quad (7)$$

Table 1 lists the branching effort, logical effort, and the parasitic delay of gates on the critical path of a full 64-bit butterfly network. We use (7) to calculate the branching effort in all stages except for the last stage. The load of the last stage is the wire and the select signals in the first stage because we assume the circuit drives another copy of itself. Suppose the select signal has to cross 32 cells to reach both MUXIs in the first stage. In addition, the select signal needs to drive the select signal for two 2:1 MUXIs; each has a gate capacitance four times as a minimum-sized inverter (See Figure 9). The branching effort before the inverter in the 2:1 MUXIs can be calculated as:

$$(32/3 + 2 \times 4) / 2 = 28/3$$

In Table 1, the  $p$  in Stage 1 is five because of the inverter in 2:1 MUXIs for the select signal.

TABLE 1: LOGICAL EFFORT AND PARASITIC DELAY IN BUTTERFLY NETWORK

Stage	Critical Gate	Load	$b$	$g$	$p$
1	MUXI	Track+ 2 MUXIs	$16/6+2$ $= 14/3$	2	5
2	MUXI	Track+ 2 MUXIs	$8/6+2$ $=10/3$	2	4
3	MUXI	Track+ 2 MUXIs	$4/6+2$ $=8/3$	2	4
4	MUXI	Track+ 2 MUXIs	$2/6+2$ $=7/3$	2	4
5	MUXI	Track+ 2 MUXIs	$1/6+2$ $=13/6$	2	4
6	MUXI	Track+ 2 select signals	$28/3$	2	4
Total			1957.31	64	25

The total effort can be calculated as

$$F = GBH = 64 \times 1957.31 \times 1 = 125267.84$$

The optimal number of stages is:

$$N = \log_{3.6} F = 9$$

There are already seven stages (including the inverter inside the first MUXI). Two inverters can be added along the path to drive long wires. This increases the parasitic delay by two. Therefore, the total delay is:

$$D = N \times F^{1/N} + P = 9 \times 125267.84^{1/9} + (25 + 2) = 60.2$$

Hence, the delay is about 12.0 FO4.

The delay of the inverse butterfly network is estimated as 13.0 FO4 [Appendix B], slightly longer than the delay of the butterfly network.

### B. 4-stage omega and flip network latency

The OMFLIP instruction can be performed with a 4-stage omega-flip network that has two omega stages and two flip stages. Since it uses only two of the four stages each time, data need to pass through the other two stages. Such pass through paths do not exist in omega or flip stages, so they need to be added in the stages. After the pass through paths are added, an output in an omega or a flip stage can choose one from three input bits. Two of them are defined by omega or flip stages (Figure 2) and the third is for the pass through path. So each box in Figure 2 can be implemented with the 3:1 MUXes that is shown in Figure 10. In the figure, data bits either go through two 2:1 MUXI or one inverter and one 2:1 MUXI. MUXI1 chooses one from  $in_0$  and  $in_1$ , the two paired bits defined by the omega or flip network. MUXI2 chooses one from the output of MUXI1 and the pass-through source  $in_p$ . Since MUXI2 inverts the input,  $in_p$  is inverted before going to MUXI2. If  $pass = 1$  in a stage, data take the pass through path. Alternatively, 3:1 MUXIs may be used for shorter delays from the data input to the output. The diagram of a 3:1 MUXI is presented in Figure 11. Figure 11a shows the transistors for selecting data inputs, which have similar structure as those in 2:1 MUXIs. The select signals in Figure 11a are generated in Figure 11b from  $pass$  and  $sel$ . The parasitic delay of 3:1 MUXIs is six; and the logical effort per data input is two, the same as that for 2:1 MUXIs. Since 3:1 MUXIs have shorter delays from the data input to the output, we will use them to implement omega or flip stages except for

the first omega stage, where the delay is dominated by the select signal. In the first omega stage, 3:1 MUXes will be used for they have short delays from  $pass$  to the output.

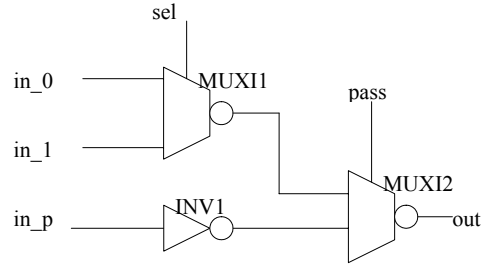
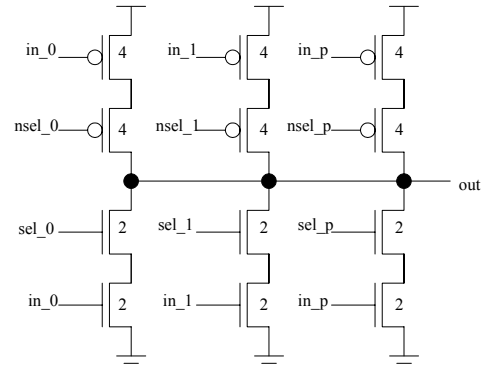
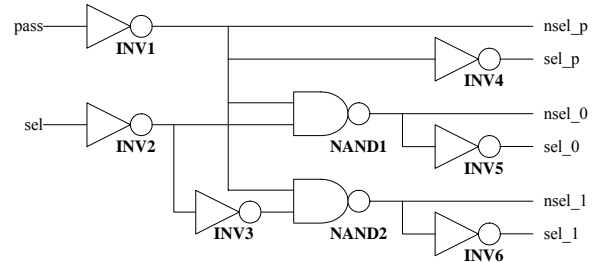


Figure 10: Implementing a 3:1 MUX with 2:1 MUXIs



a)



b)

Figure 11: 3:1 MUXIs for omega or flip stages

TABLE 2: LOGICAL EFFORT AND PARASITIC DELAY FOR 4-STAGE OMEGA-FLIP NETWORK

Stage	Gate	Load	$b$	$g$	$p$
1	MUXI2 in 3:1 MUX	Track + 3 3:1 MUXIs	$(32/3+3*2)/2$ $= 25/3$	2	5
2	3:1 MUXI	Track + 3 3:1 MUXIs	$(32/3+3*2)/2$ $= 25/3$	2	6
3	3:1 MUXI	Track + 3 3:1 MUXIs	$(32/3+3*2)/2$ $= 25/3$	2	6
4	3:1 MUXI	NOT	1	2	6
	NOT	Track + 64 select signals	$(64/3+64*4)/2$ $=416/3$	1	1
Total			80246.91	16	24

The logical effort, branching effort, and parasitic delay for 4-stage omega-flip network are listed in Table 2.

The output of the first three stages needs to drive wires and three data inputs of 3:1 MUXIs. The longest wire in a stage needs to cross 32 cells. Since 3:1 MUXEs are used in the first stage, the data are inverted only three times. As a result, a stage of NOT gate is added after the fourth stage. The NOT gates drive 64 *pass* signal of 3:1 MUXEs, plus a wire crossing 64 cells. Here, we assume each stage has a separate pass signal. The total effort can be calculated as:

$$F = GBH = 80246.91 \times 16 \times 1 = 1283950.61$$

The optimal number of stage is:

$$N = \log_{3.6} F = 10$$

The gates listed in Table 2 already have six stages, including the inverter inside the 2:1 MUXI in the first stage. Four inverters may be added along the path. The delay of such a stage can be estimated as:

$$D = N \times F^{1/N} + P = 10 \times F^{1/10} + (24 + 4) = 68.82$$

The delay is about 13.8 FO4.

### C. GRP implementation latency

The basic cell shown in Figure 7 is slow and reduces the noise margin. We will use the transmission gate (TG) shown in Figure 12 when implementing the GRP unit. Since the load of the inverter INV is one P-type transistor, we use the minimum-sized inverter. Thus, the input load of *sel* is  $(3 + 2) / 3 = 5/3$ . The input load of *i* is 2, i.e., twice as a minimum-sized inverter.

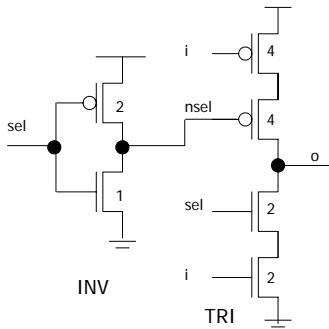


Figure 12: Implementation of the basic cell (TG) in the GRP unit

If the critical path extends from *i* to *o* in a TG, the TG has  $g = 2$  and  $p = 2$ . If the critical path extends from *sel* to *n sel* to *o*, the TG has a logical effort  $g = 4/3 \times 1$ , where 4 is the width of the transistor that *n sel* drives in TG, and 1 is the logical effort of INV. The parasitic delay  $p$  becomes three because of INV.

Since TGs generate the inverted signals, some stages may have the inverted select signals. In such stages, we use ITGs instead of TGs and inverters. Figure 13 shows the diagram of an ITG, which is the same as a TG except that it uses the inverted select signal. In a TG, the input goes to the output when *sel* = 1 while in an ITG, the input goes to the output when *sel* = 0. The input load of the data input *i* in an ITG is the same as in a TG; the input load of *sel* increases to  $7/3$ . If the critical path extends from *sel* to *n sel* to *o*, an ITG has  $g = 2/3$ , and  $p = 3$ .

Table 3 lists the branching effort, logical effort, and the parasitic delay of gates on the critical path of GRP64. In the

table, TG.sel refers to the select input of a TG, and TG.i refers to the data input.

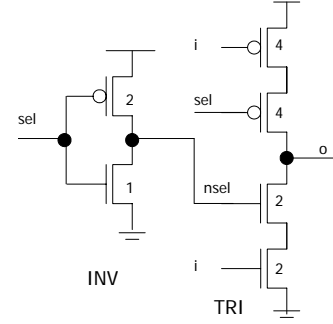


Figure 13: The diagram of ITG that uses the inverted select signal

TABLE 3: LOGICAL EFFORT AND PARASITIC DELAY OF GRP64

Stage	Gate	Load	Track Length	$b$	$g$	$p$
GRP1Z	INV1	INV + NAND + 2 TG.sel + 2 TG.i + TRACK	7	38/3	1	1
GRP1Z	INV2	2 TG.sel + 2 TG.i + TRACK	7	29/6	1	1
GRP2S	TG	4 ITG.sel + 3 ITG.i + TRACK	14	10	2	2
GRP4S	ITG	8 TG.sel + 5 TG.i + TRACK	25	95/6	2	2
GRP8S	TG	16 ITG.sel + 9 ITG.i + TRACK	47	71/2	2	2
GRP16S	ITG	32 TG.sel + 17 TG.i + TRACK	91	353/6	2	2
GRP32S	TG	64 ITG.sel + TRACK	114	562/3	2	2
GRP64D	ITG*	NOR + TRACK	128	133/5	2/3	3
	NOR	INV		1	5/3	2
	INV	INV		2	1	1
Total				$2.02 \times 10^{11}$	35.56	18

\* The critical path extends from *sel* to *o* in this stage.

The total effort can be calculated as:

$$F = GBH = \Pi g * \Pi b = 2.02 \times 10^{11} \times 35.56 = 7.17 \times 10^{12}$$

The optimal number of stages is:

$$N = \log_{3.6} F = 23$$

Since there are two stages in GRP64D, we already have 11 stages shown in Table 3. Twelve inverters need to be added along the path to drive the large load. The delay of the path can be calculated as:

$$D = N \times F^{1/N} + P = 13 \times F^{1/23} + (18 + 12) = 113.30$$

When divided by five, this is about 22.7 FO4.

### D. Comparison and discussion

Table 4 compares the latency of the different 64-bit permutation functional units. We see that GRP is the slowest, and the 6-stage butterfly network (BFLY) is the fastest. The 4-stage OMFLIP is in the middle. Although OMFLIP needs

only four stages because there are only two different types of stages, the 4-stage OMFLIP unit is not as fast as the 6-stage butterfly network for 64-bit operations. This is because one of the control signals has to drive all the gates in a stage; the length of the longest wire does not change between stages; and adding the pass through paths introduces delays. In the butterfly network, no signals have to drive a large number of gates, and the wires become shorter and shorter - the length of the wires reduces by half between stages. In addition, the butterfly network already provides a pass-through path in every stage.

GRP is the most complicated of the bit permutation instructions we investigated. Although the data do not go through a large number of gates, the select signals have a very large load in the later stages. In addition, wires become longer as the combining circuits become larger.

TABLE 4: THE LATENCY OF DIFFERENT FUNCTIONAL UNITS (64 BITS)

Functional Unit	Latency (FO4)
GRP	22.7
OMFLIP (4 stages with pass-throughs)	13.8
BFLY (6-stage Butterfly network)	12.0

Normally, microprocessors have a cycle time of 20-30 FO4 [13]. Aggressive designs may use a cycle time around 16 FO4 [13]. Except for the GRP unit, both the OMFLIP unit and the BFLY unit can finish in one cycle even in aggressive designs. The GRP instruction can finish in one cycle on most microprocessors, but it may affect the cycle time on more aggressively-designed microprocessors or take two cycles.

## V. CONCLUSIONS

In this paper, we use the logical effort method to compare the delay of different bit permutation units for permuting 64 bits. We found that GRP is the slowest, and the butterfly network (BFLY) the fastest, with OMFLIP in the middle. Although OMFLIP only needs four stages, it is not as fast as the 6-stage butterfly network used by BFLY because of its long wires between stages, the large load on one of the control signals, and the overhead for adding the pass through paths. The butterfly network already has a pass-through path in each stage; the length of the wires reduces by half between stages; and no signal has to drive a large number of gates. We present a fast, hierarchical implementation of the GRP operation. This is the most complex implementation of these bit permutation instructions; select signals have a very large load in the later stages, and wires become very long when the combining circuits are large.

For typical processors, GRP, OMFLIP, and BFLY permutation units can all complete in a single cycle. Even for processors with aggressive cycle times around 16 FO4, both OMFLIP and BFLY can finish in one cycle, but GRP would take two cycles or cause the cycle time to increase. Although the GRP instruction is slower than the BFLY or OMFLIP instruction, it has better cryptographic properties than the other two [11], and is more versatile with a variety of applications including fast subword sorting [10], multimedia as well as fast cryptography [9,11]. It is an open question as

to whether a faster implementation of the GRP instruction exists, and this can be investigated in future work.

## ACKNOWLEDGMENT

The authors wish to thank Professor Neil Burgess of Cardiff University for his time and valuable suggestions.

## REFERENCES

- [1] C. E. Shannon, "Communication Theory of Secrecy Systems", *Bell System Tech. Journal*, Vol. 28, pp. 656-715, October 1949
- [2] R.L. Rivest, "The RC5 encryption algorithm", *Fast Software Encryption: Second International Workshop*, volume 1008 of *Lecture Notes in Computer Science*, pp. 86-96, December 1994
- [3] B. Kaliski and Y.L. Yin. *On differential and linear cryptanalysis of RC5*. Lecture Notes in Computer Science 963, Advances in Cryptology -- Crypto'95, pp.171--184, Springer-Verlag, 1995
- [4] Zhijie Shi and Ruby B. Lee, "Bit Permutation Instructions for Accelerating Software Cryptography", *Proceedings of the IEEE International Conference on Application-Specific Systems, Architectures and Processors*, pp. 138-148, July 2000
- [5] Xiao Yang and Ruby B. Lee, "Fast Subword Permutation Instructions Using Omega and Flip Network Stages", *Proceedings of the International Conference on Computer Design*, pp. 15-22, September 2000
- [6] Xiao Yang, Manish Vachharajani and Ruby B. Lee, "Fast Subword Permutation Instructions Based on Butterfly Networks", *Proceedings of Media Processors 1999 IS&T/SPIE Symposium on Electric Imaging: Science and Technology*, pp. 80-86, January 2000
- [7] Ruby B. Lee, Zhijie Shi and Xiao Yang, "How a Processor can Permute  $n$  bits in  $O(1)$  cycles," *Proceedings of Hot Chips 14 - A symposium on High Performance Chips*, August 2002
- [8] Zhijie Shi, Xiao Yang and Ruby B. Lee, "Arbitrary Bit Permutations in One or Two Cycles", *Proceedings of the IEEE International Conference on Application-Specific Systems, Architectures and Processors*, June 2003
- [9] Ruby B. Lee, Zhijie Shi and Xiao Yang, "Efficient Permutation Instructions for Fast Software Cryptography", *IEEE Micro*, Vol. 21, No. 6, pp. 56-69, December 2001
- [10] Zhijie Shi and Ruby B. Lee, "Subword Sorting with Versatile Permutation Instructions", *Proceedings of the International Conference on Computer Design (ICCD 2002)*, pp. 234-241, September 2002
- [11] Ruby B. Lee, Ronald L. Rivest, L. J. B. Robshaw, Z. J. Shi, and Y. L. Yin, "On permutation operations in cipher design", *submitted for publication*
- [12] Ivan Sutherland, Bob Sproull, David Harris, *Logical Effort: Designing Fast CMOS Circuits*, Morgan Kaufmann Publishers, 1999
- [13] R. Ho, K. Mai, and M. Horowitz, "The future of wires," *Special Proceedings of IEEE*, Vol. 89, No. 4, pp. 490-504, April 2001
- [14] MOSIS, "MOSIS Parameter Test Results (0.25  $\mu\text{m}$ )", [http://www.mosis.org/cgi-bin/cgiwrap/umosis/swp/params/tsmc-025/t25t\\_mm\\_non\\_epi\\_mtl-params.txt](http://www.mosis.org/cgi-bin/cgiwrap/umosis/swp/params/tsmc-025/t25t_mm_non_epi_mtl-params.txt), June 2003
- [15] Artisan Components, Inc., TSMC 0.25um Process 2.5-Volt SAGE Standard Cell Library Databook, November 1999.
- [16] MOSIS, "MOSIS Parameter Test Results (0.18  $\mu\text{m}$ )", [http://www.mosis.org/cgi-bin/cgiwrap/umosis/swp/params/tsmc-018/t18h\\_mm\\_non\\_epi-params.txt](http://www.mosis.org/cgi-bin/cgiwrap/umosis/swp/params/tsmc-018/t18h_mm_non_epi-params.txt), June 2003
- [17] Jan M. Rabaey, *Digital integrated circuits: a design perspective*, Prentice Hall, 1996
- [18] Neil Burgess, "New models of prefix adder topologies," *to be published in Journal of VLSI Signal Processing*, 2004

## APPENDIX

### A. The capacitance of wires

This section is based on correspondence with Professor Neil Burgess in July 2002.

The capacitance is estimated for process technologies of 0.25 $\mu\text{m}$  and 0.18 $\mu\text{m}$ , as shown in Table 5. The gate capacitance is taken from [14, 16]. It is assumed that a minimum-size inverter has an nFET ratio of 2:1 and a pFET ratio of 4:1. The capacitance of the wires depends on many factors. The parameters for M1 are taken from [14, 16], assuming typical layout strategies, where M1 does not overlap with polysilicon and M2.

TABLE 5: ESTIMATION OF WIRE CAPACITANCE

1	Process technology	0.25 $\mu\text{m}$	0.18 $\mu\text{m}$
2	Gate capacitance/unit area	6000 aF/ $\mu\text{m}^2$	8300 aF/ $\mu\text{m}^2$
3	Gate capacitance of a minimum-sized inverter	2.25 fF	1.61 fF
4	Area capacitance of M1	70 aF/ $\mu\text{m}^2$	77 aF/ $\mu\text{m}^2$
5	Fringing capacitance of M1	35 aF/ $\mu\text{m}$	38.5 aF/ $\mu\text{m}$
6	Width of M1 tracks	0.45 $\mu\text{m}$	0.30 $\mu\text{m}$
7	Capacitance of M1 track per $\mu\text{m}$	101.50 aF	100.1 aF
8	Number of inverters with equivalent capacitance of 1- $\mu\text{m}$ M1 wire	0.045	0.062
9	Cell width	7.2 $\mu\text{m}$	5.2 $\mu\text{m}$
10	Number of inverters with equivalent capacitance of wire traveling across a cell	$\approx 1/3$	$\approx 1/3$

The gate capacitance of minimum-sized inverter, which is listed in row 3 in the table, is calculated as [17]:

$$g\_len \times (g\_len \times 2 + g\_len \times 4) \times g\_cap$$

where  $g\_len$  is feature size of the process technology, and two and four are the width-to-length ratio of nFET and pFET, respectively.  $g\_cap$  is the gate capacitance per unit area.

The wire capacitance of 1- $\mu\text{m}$  M1 track, which is listed in row 7 in the table, is calculated as [17]:

$$area\_cap \times w\_width + fringing\_cap \times 2$$

where  $w\_width$  is the width of the wire;  $area\_cap$  is the capacitance per unit area and  $fringing\_cap$  is the fringing capacitance per unit length. The length of the wire does not

appear in the formula because it is one here.

Row 8 in the table is the number of minimum-size inverters that have a capacitance equivalent to that of 1- $\mu\text{m}$  wire. It is calculated by dividing row 7 by row 3.

In 0.25 $\mu\text{m}$  technology, the height of 2:1 MUXI is 6.4  $\mu\text{m}$  [15], and the width ranges from 6.3 $\mu\text{m}$  to 7.2 $\mu\text{m}$ , where smaller ones are used to drive small loads. It is safer to choose a larger one.

Row 10 is generated by multiplying row 9 by row 8.

Table 5 shows the capacitance of wire extending across a 2:1 MUXI is approximately one third of the capacitance of a minimum-sized inverter. Burgess uses similar method in [18] to estimate the delay of adders, and shows the estimation matches the simulation results well.

### B. Delay of inverse butterfly network

The following table lists the branching effort, logical effort, and parasitic delay for calculating the delay of the inverse butterfly network. In the table, all MUXIs refer to 2:1 MUXIs.

TABLE 6: LOGICAL EFFORT AND PARASITIC DELAY OF THE INVERSE BUTTERFLY NETWORK

Stage	Gate	Load	$b$	$g$	$p$
1	MUXI	Track +2 MUXIs	2/6+2 = 7/3	2	5
2	MUXI	Track +2 MUXIs	4/6+2 = 8/3	2	4
3	MUXI	Track +2 MUXIs	8/6+2 = 10/3	2	4
4	MUXI	Track +2 MUXIs	16/6+2 = 14/3	2	4
5	MUXI	Track +2 MUXIs	32/6+2 = 22/3	2	4
6	MUXI	Track +2 MUXIs' select signal	28/3	2	4
Total			6624.74	64	25

The total effort can be calculated as:

$$F = GBH = 64 \times 1957.31 \times 1 = 423983.36$$

The optimal number of stage is:

$$N = \log_{3,6} F = 9$$

Since there are already seven stages (including the inverter inside MUXI), we add two inverters on each path. This will increase the parasitic delay by 2. Therefore, the delay is:

$$D = N \times F^{1/N} + P = 9 \times F^{1/9} + 25 + 2 = 65.0$$

When divided by 5, this is about 13.0 FO4.

The inverse butterfly network is slightly slower than the butterfly network because the delay of the stage that has the longest wires can not overlap with the delay of the control signals.