RAN WG1 meeting #5 **TSGR1#5(99) 539**

Place : Corea
Date : 1-4 June
Title : Shuffle multiplexing definition and complexity
Source : Mitsubishi Electric
Paper for : Discussion

Table of content:

1 Introduction

In [1] Nortel has hinted at the use of a shuffle multiplexing. In [2], Nortel gave a definition of this shuffle multiplexing, but for multiplexing only 2 blocks. In section 3 we give reasons why considering only the case of 2 blocks is not sufficient.
The complexity of the interleaving and multiplexing scheme must be evaluated on the whole : not only for interleaving, but also for multiplexing. In order to evaluate the complexity of the scheme proposed by Nortel we therefore need the definition of the shuffle multiplexing for greater number of multiplexed blocks, along with a study of its complexity. This is what is intended in this paper.

2 References

[1] **TSGW1#2(99)106** Discussion on channel interleaver for 3GPP selection **(Source Nortel)**
[2] **TSGW1#4(99)466** On the Algebraic Channel Interleaver Design **(Source Nortel)**

3 Motivation for cases where more than 2 blocks are to be multiplexed together

Currently, there are several reasons why there might be more than 2 blocks multiplexed together at the input of the shuffle multiplexer:

- **The number of blocks is depending of the number of QoS.**
  In the case of the plain telephony terminal we have only 2 QoS : one for speech, and one for associated control signalling. However there are surely cases with more than 2 QoS.
- **The number of blocks per QoS can be greater than one.**
  The channel coder is inputted every TTI a transport block set + CRC, and conversely outputs every TTI a code block set. Our understanding is that the number of blocks in the code block set can be zero, one or more depending both on the possible transport block sets, and of the channel encoder technology. For instance, in the case of a convolutional encoder it might be better to have only zero or one code block, that is to say the encoding would be spanning on several transport blocks when needed. As a matter of fact, there would be no additional complexity in decoding all the transport block set in one pass.

Contrary to the convolutional encoder, the turbo encoder might need restriction of the number of transport blocks over which the encoding is spanning. This is because the complexity of the decoder is increasing with the size of the decoded block : more memory is needed, and also several turbo interleavers would be needed in the case of variable rate.

- **The number of blocks can be increased in the case of unequal protection of source bits**
  In order to achieve unequal protection of the several class of bits at the output of the source encoder, a proposed solution was to encode them on separate transport channels. If this solution is retained, then this means that the number of blocks to multiplex for this source encoder is one block for each class of bits.

In conclusion to this section, considering only the case of 2 blocks multiplexed, as in [2] might be slightly too restrictive, even in the case of the plain speech only terminal.

4 General definition of shuffle multiplexing

The general algorithm, for any number p of blocks to be multiplexed is given on table 1. The block selection step ("find j such that e_j = min $\{e_1, e_2,\ldots, e_p\}$") is defined on table 2. The block selection step is the step that makes the complexity depends on the number of blocks.

An example of 3 blocks multiplexed by the algorithm below is shown on figure 5.

input data :

X_1: **array**[0..N_1-1] **of symbols** -- 1st block to be multiplexed

… …

X_i: **array**[0..N_i-1] **of symbols** -- i^{th} block to be multiplexed

… …

X_p: **array**[0..N_p-1] **of symbols** -- p^{th} block to be multiplexed

Y : **array**[0..N-1] **of symbols** -- multiplex block, $N = N_1+N_2+\ldots+N_p$

The algorithm is as follows :

for i=1 **to** p **do**

  $\delta_i := N-N_i$; -- initialise error criteria increment

  $e_i := \delta_i$; -- initialise error criteria

  $x_i := 0$; -- initialise position in all the blocks to be multiplexed

end_do

x := 0 -- initialise position in the multiplex

```
while x< N do
    find j such that e_j = min {e_1, e_2,…, e_p}
    Y[x] := X_j[x_j]          -- multiplex one symbol
    x := x+1
    x_j := x_j+1
    if x_j = N_j then
        e_j := +∞             -- block j will not be selected again
    else
        e_j := e_j + 2·δ_j
    end_if
end_do
```

Table 1 Shuffle Multiplexing

```
j := 1 ;
i := 2 ;
while i≤ p do
    if e_i < e_j then j := i ;
end_do
```

***Table 2 Block selection step* "find j such that $e_j = \min\{e_1, e_2,\ldots, e_p\}$"**

4.1 Dynamic of error criteria

For $x = N_i$ the maximum value of the error criteria e_i is :

$$f(x) = N-x + 2x(N-x)$$

$df/dx = -1 +2(N-x) -2x = (2N-1) – 4x$
then $f(x)$ is maximum for $x = x_{max} = (2N-1)/4$
Now if we take $N = 81376$, we get that :
$x_{max} = 40687.75$
$N- x_{max} = 40688.25$
maximum of $f(x) = \quad 3311067376.125 < \ 4294967296 = 2^{32}$

Then e_i can be held on a 32 bit register.

Note : the $+\infty$ value can be easily represented as the "all one" value (4294967295).

5 Definition of shuffle multiplexing in case of 2 blocks.

In the case of only 2 block multiplexed, the definition is the one given by Nortel in [2]. In this definition we have $e = e_2-e_1$. We remind the definition in the table 3 below. On this table the statement "e := $+\infty$" and "e := $-\infty$" can be equivalently respectively replaced by "e := $2\cdot N^2$" and "e := $-2\cdot N^2$"

```
input data :
    X_1: array[0..N_1-1] of symbols     -- 1st block to be multiplexed
    X_2: array[0..N_2-1] of symbols     -- 2nd block to be multiplexed
    Y : array[0..N-1] of symbols        -- multiplex block,  N = N_1+N_2
```

The algorithm is as follows :

```
e_i := N1-N2;                    -- initialise error criteria
x1 := 0;                         -- initialise position in the 2 blocks to be multiplexed
x2 := 0;
x := 0                           -- initialise position in the multiplex

while x< N do
    if e ≤ 0 then
        Y[x] := X2[x2]           -- multiplex one symbol
        x2 := x2+1
        if x2 = N2 then
            e := +∞              -- block 2 will not be selected again
        else
            e := e + 2·N1
        end_if
    else
        Y[x] := X1[x1]           -- multiplex one symbol
        x1 := x1+1
        if x1 = N1 then
            e := -∞              -- block 1 will not be selected again
        else
            e := e - 2·N2
        end_if
    end_if
    x := x+1
_do
```

3 Shuffle

6

When several blocks have the same size, the algorithm of table can be optimised so that the block selection step is less stringent. The case of several blocks with the same size will

transport block set contain a varying number of equal size transport blocks.

The optimisation consists in grouping the blocks by sizes.

multiplexer. That is to say, there are q blocks of size N_1 $_2$ blocks of size N , …, q_p N_p

Instead of carrying the block selection over the **Error!** sizes.
This is shown below :

```
input data:
        1,1:        [0..N1      of symbols      -- 1   block to be multiplexed
                                …
        X^1_p1: array[0..N1-1] of symbols       -- q1^th block to be multiplexed
        X2,1: array [0..Ni-1] of symbols        -- (q1+1)^th block to be multiplexed
              …             …
        X^p_pp: array [0..Np-1] of symbols      -- Error! block to be multiplexed
        Y : array [0..N-1] of symbols           -- multiplex N = q1·N1+q2·N2+…+ qp·Np

The algorithm is as follows :
        for i=1 to p do
                δi := N-Ni;             -- initialise error criteria increment
                ei := δi;               -- initialise error criteria
                xi := 0;                -- initialise position in all the blocks to be multiplexed
        end_do
        x := 0                          -- initialise position in the multiplex

        while x< N do
                find j such that ej = min {e1, e2,…, ep}        -- see table 2
                for i:=1 to qj do
                        Y[x] := Xj,i[xj]        -- multiplex one symbol
                        x := x+1
                end_do
                xj := xj+1
                if xj = Nj then
                        ej := +∞        -- blocks of size Nj won't be selected again
                else
                        ej = ej + 2·δj
                end_if
        end_do
```

Table 4 Shuffle multiplexing optimised when several blocks have the same sizes

7 Use of a hardware accelerator

Now, when the number of blocks to be multiplexed is great, another way to speed up the algorithm of table 1 is to carry out the block selection step by a HW accelerator. The hardware accelerator we propose is shown on figure 4. This HW accelerator is e.g. accessed by a processor that is carrying out the rest of the algorithm.

The HW accelerator is holding a list of the couples $(e_i,i-1)$. This list is ordered by a bubble sort so that the least couple $(e_i,i-1)$ is placed in front of the list. Then the only thing is therefore to read the value of i for the first element of the list in order to know which block is selected for next multiplexed symbol.

Each couple $(e_i,i-1)$ is kept in a register of L bits. Among this L bits 32 bits are used to carry the value of e_i and $\lceil \log_2(p) \rceil$ bits are used to carry the value of i-1. Then $L = 32+\lceil \log_2(p) \rceil$.

In the case p = 8 the register format is shown on figure 3 : the value of i is kept in the least significant bits, whereas the value of e_i is kept on the most significant bits. This way by

sorting the registers with the canonical order on integers we get the least valued register is the (e_i,i) couple to be selected by the algorithm of table 2.

The sorting is performed by an elementary sort gate shown on figure 1. This gate has two input ports of L bits D0 and D1 and two output port O0 and O1 the gate performs as follows :

```
if D1 greater than D0 then
        O1 ⇐ D1
        O0 ⇐ D0
else
        O1 ⇐ D0
        O0 ⇐ D1
end if
```

The SELECT2 signal alternately takes 0 and 1 values : there is an even sort phase (SELECT2 = 0) and an odd sort phase (SELECT2 = 1). This is shown on figure 2.
Note that the D0 and D1 input ports of MUX gates are in reverse order from a MUX gate to the next one. This way, when SELECT2 = 1, then only sort gates at odd positions are operating a sort. That is to say only the 1st, the 3rd, etc. gates are working.
Respectively, when SELECT2 = 0 only the sort gates at even position are working. That is to say only the 2nd, 4th, etc. gates are working.

With such an arrangement, if besides SELECT1 = 1, ENABLE1 =1 and ENABLE2 = 1, and RESET = 0, then the contents of the registers will end in being sorted in increasing order from the left to the right of the figure3.

The SELECT1 port can be set to 0 only when SELECT2 is also null. When SELECT1 is set to zero the value of the first register is overwritten by the value provided by port DATA_IN.

The algorithm is operated as follows :

- Initialisation
  - First of all the register are reset by maintaining RESET port to 1 as long as needed. Then RESET is kept to 0 until the completion of the algorithm.
  - The next step is to initialise all the registers. This is done very simply by writing a new value (e_i,i-1) in the first register, and by doing this p times (p=8 here). Because the all-zero values that have been set by means of the RESET are necessarily not greater than the initial values (e_i,i-1), there is necessary before each write an all-zero value from the RESET that is overwritten in the first register.
- Multiplexing loop
  - Then, for all multiplexing iteration, all there is to do is to read (e_j,j-1) on the DATA_OUT port. The value of j gives which block is selected. This way the next symbol can be multiplexed.
  - Then the processor increments the read e_j like in the instruction "$e_j := e_j + 2\delta_j$" in the algorithm of table 1. Then the processor overwrites the first register by the value of (e_j,j-1) by puting it on port DATA_IN and by setting SELECT1 to 0 at the same time for one clock cycle when SELECT2 = 0.

Note with this scheme a symbol can be multiplexed **every 2 clock cycles.** The crux of this design is that **the list needs not to be completely sorted for symbol to be multiplexed.** Instead, the sorting and the multiplexing can be carried out **in parallel**. As a matter of fact,

the only thing to be sure is that the first element of the is hold the minimum value. Whether the remainder of the list is sorted is of no importance.

When the HW accelerator is operated at maximum speed, then SELECT1 and SELECT2 always have the same value, that is to say they are both alternating 0 and 1. Every time SELECT1 is 1 the next selected block is read on DATA_OUT port, and every time SELECT1 is 0 the error criteria is updated for the just selected block.

7.1 Hardware accelerator complexity estimate

In the following spreadsheet we find a complexity of around 4 kgates for the case of 8 block multiplex HW accelerator.

We have the following spread sheet:

constants names	constant value		comment
L	35		L = 32+ceiling(log p)
p	8		

component name	gate nb per component	acronym	comment
latch	6	LATCH	
2to1 MUX	4	MUX	
L bit comparer	274	GE	8*L-6
elementary sort	554	SORT	1*GE + 2*L*MUX
total	**3958**		(p-1)*SORT + p*LATCH + p*MUX

Complexity of comparer :
EQ_n : comparer on n bits for "=" comparison
GE_n : comparer on n bits for "≥" comparison

comparator expression		complexity
$GE_n(x_{n-1}, \ldots, x_0, y_{n-1}, \ldots, y_0)$	$\Leftarrow GE_{n-k}(x_{n-1}, \ldots, x_k, y_{n-1}, \ldots, y_k)$ or $(EQ_{n-k}(x_{n-1}, \ldots, x_k, y_{n-1}, \ldots, y_k)$ and $GE_k(x_{k-1}, \ldots, x_0, y_{k-1}, \ldots, y_0))$	$GE_{n-k} + GE_k + EQ_{n-k} + 2$
$GE_n(x_{n-1}, \ldots, x_0, y_{n-1}, \ldots, y_0)$	$\Leftarrow EQ_{n-k}(x_{n-1}, \ldots, x_k, y_{n-1}, \ldots, y_k)$ and $EQ_k(x_{k-1}, \ldots, x_0, y_{k-1}, \ldots, y_0)$	$EQ_{n-k} + EQ_k + 1$
$GE_1(x_i,y_i)$	$\Leftarrow x_i$ or not y_i	2 gates
$EQ_1(x_i,y_i)$	$\Leftarrow$ (not(x_i or y_i)) or (x_i and y_i)	4 gates

By making a cascade architecture (k = n-1) we have :
$GE_n = GE_1 + GE_{n-1} + EQ_1 + 2 = 2 + GE_{n-1} + 4 + 2 = GE_{n-1} + 8$
So $GE_n = 8n - 6$

8 Conclusion

In this paper we have given a complete description of the shuffle multiplexing as we can understand it.
The complexity, whether it is hardware or software complexity is clearly depending on the number of blocks that are multiplexed together. Then in order to carry out fair comparison we would need more insight on the actual number of blocks that need to be multiplexed.

Especially, it seems that when the number of blocks is great, the shuffle multiplexing is more complex than a simple concatenation multiplexing followed by NTT DoCoMo's modified FS-MIL proposal.

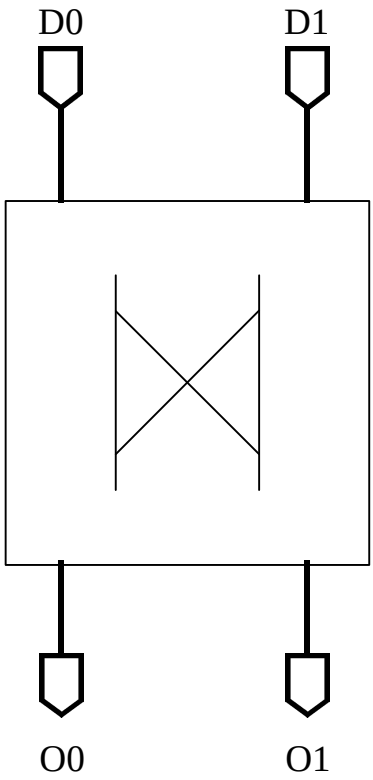

Figure 1 Elementary sort gate

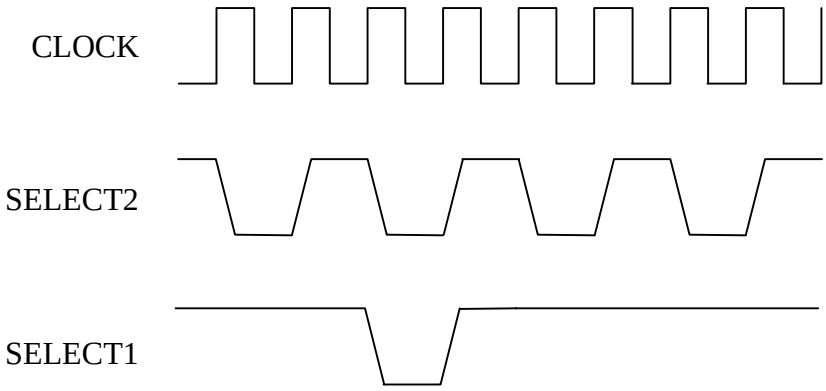

Figure 2 Waveforms for CLOCK, SELECT2 and SELECT1 signals

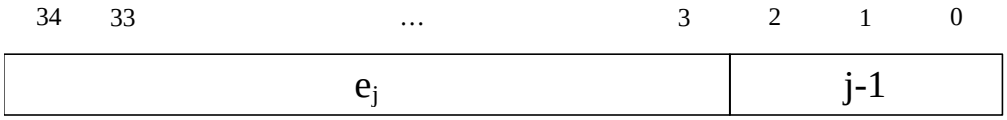

Figure 3 format of registers, when p≤8

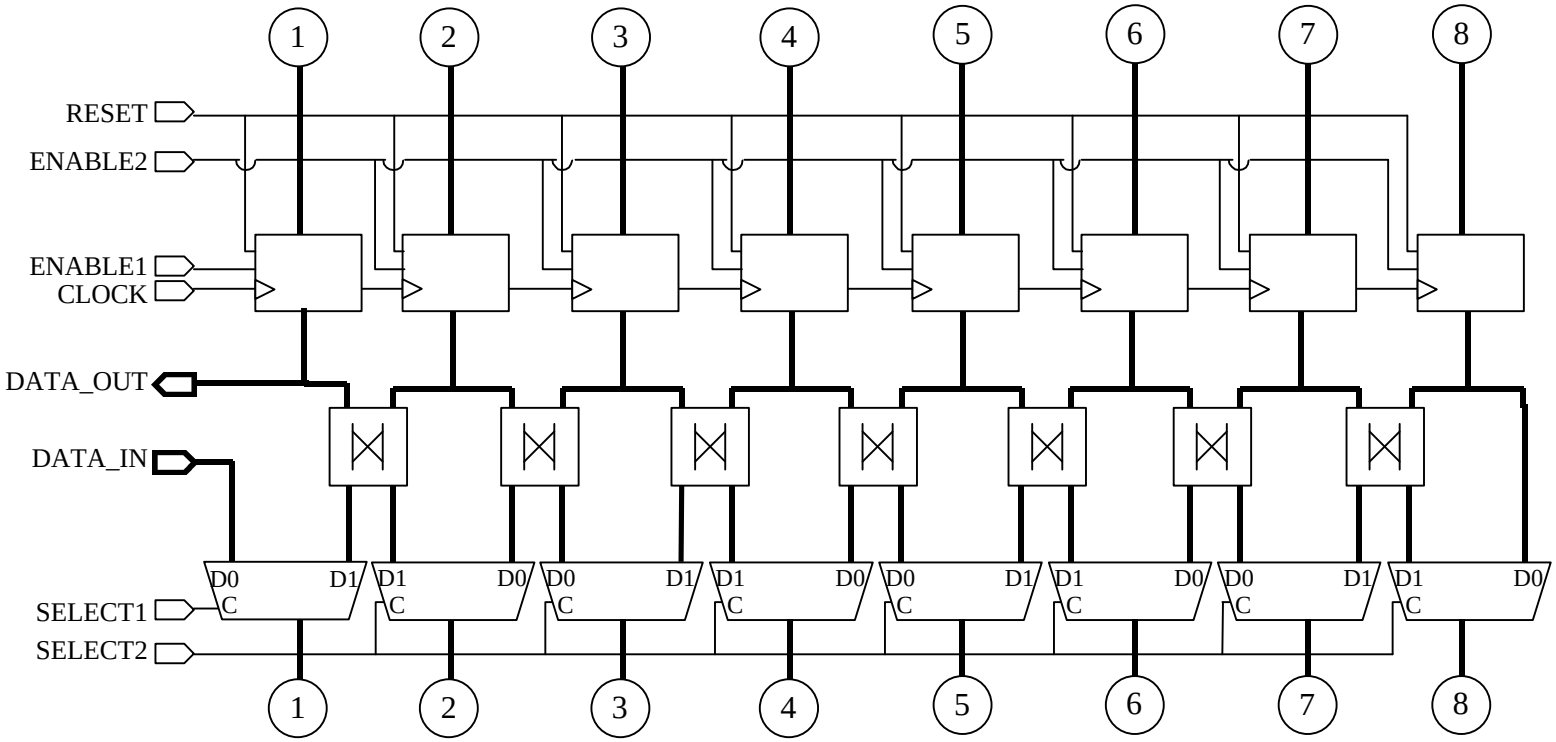

Figure 4 Hardware accelerator for fonction « find j such that e_j = min {e_1, e_2…, e_8} »

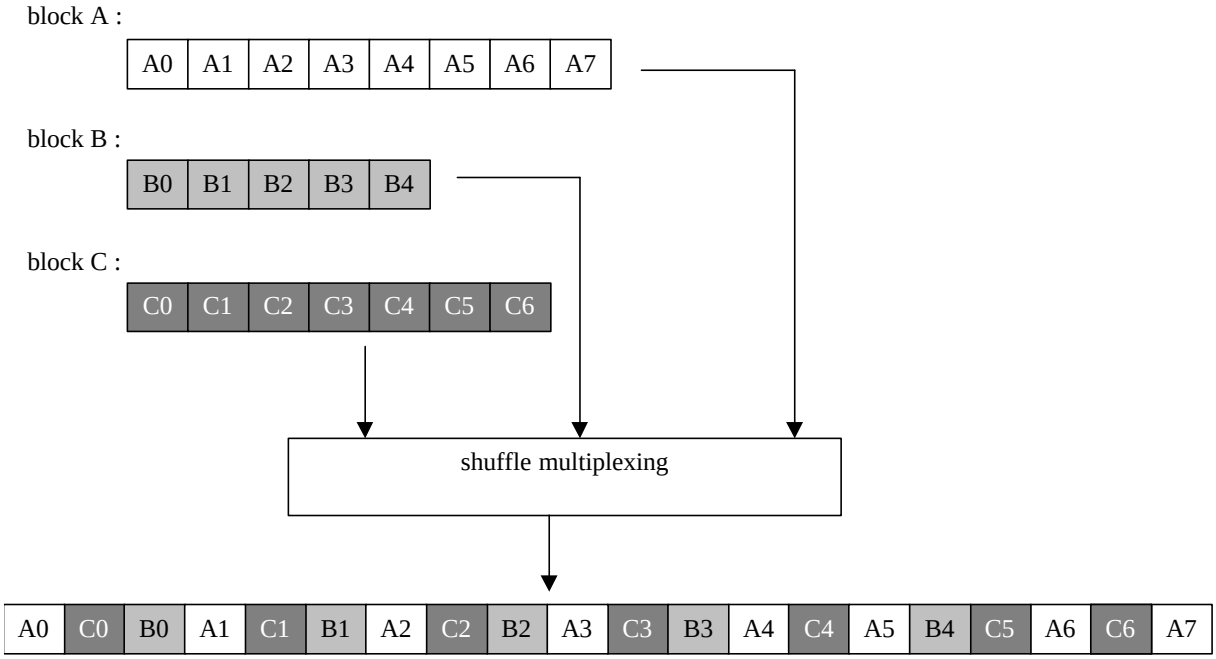

Figure 5 Shuffle Multiplexing