AREA-TIME OPTIMAL ADDER WITH RELATIVE PLACEMENT GENERATOR

Aamir A. Farooqui [1], Vojin G. Oklobdzija [2], Sadiq M. Sait[3]

[1]Synopsys® Inc.
Synopsys Module Compiler
700 Middlefield Road,
Mountain View CA 94043
USA
aamirf@synopsys.com

[2]ACSEL Laboratory
Electrical Engineering Dept.
University of California
Davis, CA 95616
USA
vojin@ece.ucdavis.edu

[3] Department of Computer Engineering
KFUPM Box 673
King Fahd University of Petroleum & Minerals
Dhahran-31261, Saudi Arabia
sadiq@kfupm.edu.sa

1 ABSTRACT

This paper presents design of an adder generator, for the production of area-time-optimal adders. A unique feature of the proposed generator is its integrated synthesis and layout environment achieved by providing relative placement information to the synthesis tool. Adders produced by this generator are dynamically configured for a given technology library, wire-load model, delay, and area goal. The adder architecture used in this generator is a hybrid of Brent & Kung, carry select, and ripple carry adders. When compared with standard cell fast adders, a 20%-50% reduction in area with comparable delays is achieved. The reduction comes from a judicious selection of ripple carry or carry select adders based on computation of delays. When performance is being met, the carry select adders are replaced with ripple carry adders. The proposed generator has been integrated into a commercially available high-performance datapath design tool.

2 INTRODUCTION

Addition is an important operation affecting the speed and performance of several digital systems. High-speed adders can be realized with the widely used carry look-ahead [1], carry select [2], or binary carry look-ahead [3, 4] techniques. Due to the high design cost of full custom adders, standard-cell based designs are widely used [5][6][7][8] but their performance is also limited by a specific cell library and design constraints.

There is another class of adders that are obtained using software programs called generators [9][10][11][12]. Using these, adders can be generated for a variety of design goals, bit-widths, and process technologies. First attempt to produce area-time optimal adders using software programs was reported by B. Wei et al., in [9]. The proposed adder architecture was based on Ladner and Fischer's parallel prefix computation model [13], which is essentially a look-ahead addition. The design was formulated as a dynamic programming problem and optimized for area and delay. In [10], Chan et al., proposed a multi-dimensional programming paradigm to control the block sizes of carry skip adders and carry look-ahead adders (CLA) for minimizing the delay. In the generator proposed by Becker et al. [11] automatic generation of conditional sum adders, with included testability features, is presented. All the above generators have a major drawback; they do not consider the wiring effects, which impacts the performance of the final product. With shrinking VLSI dimensions, especially in today's deep sub-micron technologies, interconnect delays play a dominant role in overall performance of the circuit [[14]. The delay due to interconnects in some cases exceeds the switching delays of gates/circuits. Moreover the digital design paradigm has moved from logic domain to physical domain, but none of the above mentioned generators provide any information for the placement or tiling of adder cells.

In this paper, we present an efficient generator to generate area-time optimal adders, with relative placement (RP) information. Relative placement (RP) is the placement of cells/gates with reference to each other. RP information is used by the placement tool for fast, efficient and structured placement of instances in layout. RP aligns cells horizontally in rows and vertically in columns. Each row corresponds to a single bit of a data-bus. The cells making up a function span multiple bits (rows) and are arranged vertically in a column. A function can be composed of one or more such columns.

The proposed generator takes into consideration the wire-load models and other parameters such as delay, area, and operating conditions (temperature and voltage). The adders generated are targeted for low area, with speeds comparable to Brent & Kung [7][9] and Koggie Stone adders [15]. The generator has been integrated with a commercially available datapath synthesis tool, which allows further enhancement such as, pipelining, retiming, GUI interface, and adder instantiation in a Verilog like language called MCL (Module compiler language).

The adder architecture used in this generator is a hybrid of Brent & Kung [4], carry select [2], and ripple carry adders. An integral feature of the generator is the intelligent use of both ripple carry and carry select adders. In order to achieve low area with high-speed, area optimized ripple carry adders [16] are used along with carry select adders (CSA) and carry generation logic. The whole adder is composed of 4-bit ripple carry and CSA blocks. The 4-bit ripple carry adder blocks are used for fast arriving carry signals (in the beginning of the adder) and CSA are used for late arriving signals. The carry generation logic (carry chain) has been implemented using the '*o*' operator as described by Brent in [3,4] with a modification that four-bit groups are used, instead of two. By using four-bit groups, the number of wires and hardware is reduced by half while keeping the adder delay proportional to O(*log n*) [16], where '*n*' is the number of bits of the adder. In the proposed generator, timing analysis is used to equalize the computation of the entire sum thereby employing a maximum number of ripple carry adders, resulting in further reduction of hardware and thereby the layout area. The adders produced by the generator are similar in topology to the one proposed earlier [16]. However better quality adders are generated due to timing analysis incorporated using different technology wire load models. The paper is organized as follows: In Section 3 the architecture of the adder produced by the generator is described. In Section 4 the generator algorithm is described in detail. In Section 5 experimental results are reported to demonstrate the effectiveness of the proposed generator. Finally, Section 6 concludes this paper.

3 ADDER ARCHITECTURE

Let, $A = a_{n-1}, a_{n-2} \ldots\ldots a_1, a_0$, and $B = b_{n-1}, b_{n-2}, \ldots\ldots b_1, b_0$ be the two input operands, with a_{n-1} and b_{n-1} be the most significant bits. The

generate and propagate signal at bit position "*i*" are given by; $g_i = a_i \bullet b_i$, and $p_i = a_i + b_i$, (where: $\bullet$ = AND operation and + = OR operation). The Carry out from bit position "*i*" is given by; $C_i = g_i + g_{i-1} \bullet p_i$ provided $C_0 = 0$. The "*o*" operator as defined by [3] is given as follows:

$$(g, p)\ o\ (g', p') = (g + (p \bullet g'), p \bullet p') \quad (1)$$

The group Generate (G) and Propagate (P) are given by:

$$(G_i,P_i) = (g_0,p_0) \text{ if } i=0 \ \&\ (g_i,p_i)\ o\ (g_{i-1},p_{i-1}) \text{ if } 0<i<n \quad (2)$$

In [7, 9, 3], using (1), the generate and propagate signals for each level (*k*) of the adder are generated using the following combination:

$$(G_{i+2k},P_{i+2k}) = (g_{i+2k},p_{i+2k})\ o\ (g_i,p_i) \text{ for } 0 < k < \log n \quad (3)$$

Fig. 1 shows two carry generation schemes using (3). The carry tree is made up of 3-input AND-OR and 2-input AND gates, implementing $g_{i.i-1} = g_i + g_{i-1} \bullet p_i$ and $p_{i.i-1} = p_{i-1} \bullet p_i$ (small circles) respectively, at each level of the tree. It can be observed from Fig. 1 that the number of wires and area of Kogge-Stone (KS) or Brent's adder (BK) [7, 16] increases exponentially with the increase in the number of bits. In case of Ladner Fischer adder the fan-out of gates increase with the depth of the tree [13].

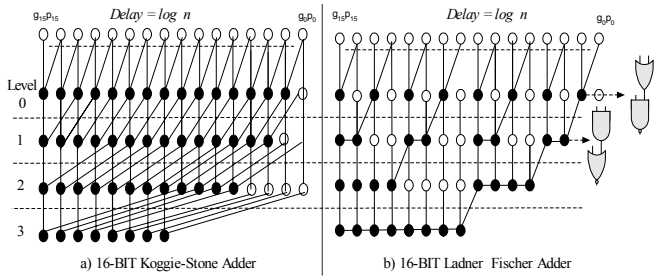

Fig. 1. Parallel prefix adder architectures.

In the proposed implementation, for '*n*' bits, at $k = 0$ (first level) *n/2* generate and propagate signals are produced using the following combination:

$$(G_{2i+1}, P_{2i+1}) = (g_{2i+1}, p_{2i+1})\ o\ (g_{2i}, p_{2i}) \text{ for } 0 < i < n/2 \quad (4)$$

At the second level *n*/4 signals are produced (by grouping the signals generated at the first level) using (4) but limiting *i* to *n/4*. These signals are the four-bit group generate and propagate signals. Their value for 4-bit case is given below, and their grouping is shown in Fig. 2.

$$(g_{10}, p_{10}) = (g_1, p_1)\ o\ (g_0, p_0) \text{ and}$$

$$(g_{32}, p_{32}) = (g_3, p_3)\ o\ (g_2, p_2) \text{ at k = 0 (at first level)} \quad (5)$$

$$(G_{30}, P_{30}) = (g_{32}, p_{32})\ o\ (g_{10}, p_{10}) \text{ (at second level)} \quad (6)$$

In this realization, only four bit group generate and propagate signals are generated ($G_{3+4i,4i}$, $P_{3+4i,4i}$ for $0 < i < n/4$); the rest are generated within the conditional sum adders. Once we have the 4-bit group generate and propagate signals, the carries in multiplies of 4 are generated using (2). This technique results in minimum wiring and area. For n bits, approximately n/k signals are generated at each level of the adder, in contrast to 2(n-2k) signals required in [7] and [9]. Four-bit groups offer lesser wiring and area as compared to binary, keeping the delay equivalent to Brent's adder [3].

Fig. 2 shows the block diagram of the proposed 32-bit adder. In this architecture, the carry tree generates the carry inputs in multiples of four (C_{30}, C_{70}….) Since inverted logic is faster than non-inverted logic, therefore the proposed generator alternates the polarities of *g* and *p* at each level to produce fast carry generation logic. The carries generated by the tree are then used to generate the final sum using either a ripple carry adder or a CSA. In the following sections we explain the design of the area-delay-optimized ripple and CSA.

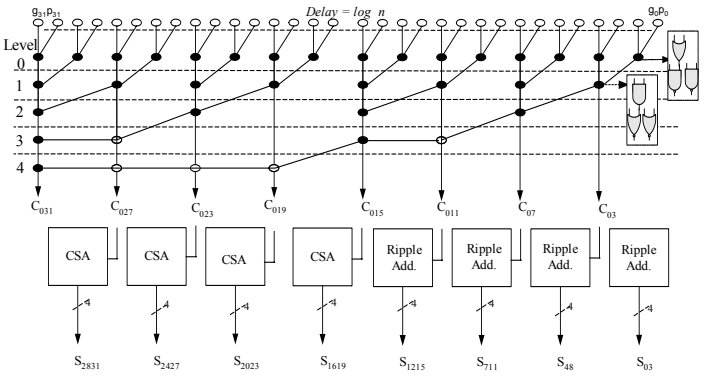

Fig. 2. 32-Bit adder block diagram.

Fig. 3 shows the area optimized 4-bit adder. The area of the adder is optimized by utilizing generate (*g*) and propagate (*p*) signals produced in the carry tree. This adder requires only 9-gates (counting shaded AND-OR pair as one gate) for 4-bits.

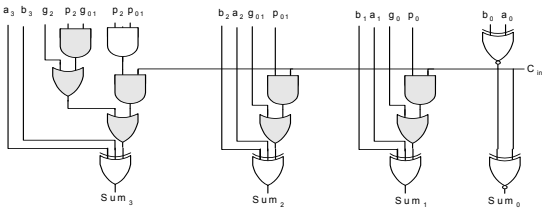

Fig. 3. Four bit area optimized ripple carry adder

Since, the four bit carry select adders are not on the critical path, they could be designed using two sets of four bit ripple carry adders, with $C_{in} = 0$ for one set, and $C_{in} = 1$ for the other. However, we have found that it is possible to reduce the hardware by merging the two adders together. Fig. 4 shows the schematic diagram of the 4-bit merged carry select adder (CSA). The hardware of this merged CSA is approximately 40% smaller than the hardware required by two separate four-bit ripple carry adders. In VLSI implementation, the 4-bit CSA is combined with the 4-bit carry generate circuit. In the following sections, we explain the Generator architecture.

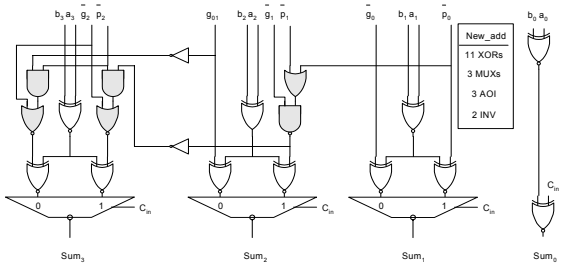

Fig. 4. Four bits carry select adder.

4 GENERATOR

In this section we present the generator design. The generator is described in C++ and contains the following modules:

a) Logic generator, and b) Adder optimizer.

Functions, for obtaining technology related information, delay computation, parsing of input for the determination of adder characteristics (bitwidth, delay, area goals), and GUI interface are carried out by the datapath synthesis engine.

4.1 Logic generator

The main function of the logic generator is to instantiate, place, and interconnect logic cells and generate relative placement (RP) information. RP assigns an instance/gate to a row and column position. Each row corresponds to a single bit of a bus, and each column corresponds to a function or an operation. The instances making up a function span multiple bits (rows) and are arranged vertically in a column. A function can be composed of one or more columns. Relative placement does not specify absolute (X, Y) locations for instances, as would full ASIC place and route. The benefits of working with relative placement rather than absolute placement are efficiency and speed. By working at a higher level of abstraction, one can quickly explore the design space. The generator provides the RP information along with gate level netlist in a file that provides an efficient starting point for placement tools.

The datapath synthesis engine parses the input and then passes a data structure (*add_db*) to the logic generator. This data structure contains information regarding the adder type, number of bits, and pointers to input/output operands, desired area and delay. The same data structure contains the pointer to synthesized adder output data.

The logic generator creates a data structure *gp_tree* consisting of '*n*' link lists containing input operand bits and their associated delay in ascending order. Following this, it first synthesizes the carry tree, as explained below.

Carry Tree Generation

Generate the g and p terms for each bit starting from 0 to n

During this operation the generator, instantiates one row of NOR and NAND gates, to produce *g_n* (inverted generate), and *p_n* (inverted propagate) terms respectively, for each bit.

Generate a binary tree using Equation 6 for log(n-1) levels

This operation instantiates AND-OR and AND gates or each black node of Fig. 2. Since inverted logic is faster then non-inverted logic therefore alternate polarities of *g* and *p* are produced at each level of the GP tree (using AND-NOR and NAND gates), in order to get full advantage of inverting logic cells.

Generate intermediate g and p terms of the GP tree, which is not power of 2.

Finally, instantiate AND-OR and AND (AND-NOR and NAND gates) for white nodes of Fig. 2, in a manner similar to step 2.

During each of the operations (steps 1-3), nets corresponding to each instantiated gate are placed in the link list *gp_tree*, and sorted for delays. After each cell instantiation, RP data structure is updated with the bit/column number and *GP* tree level. Following step-3, carries in multiples of 4n-1 are generated. These carries are then applied to the four bit adders as shown in the following steps.

Adder Generation

The adder generator only instantiates the ripple carry and CSA as shown below:

Generate the first ripple carry adder (Fig.2) for the first four bits.

This operation instantiates a ripple carry adder. Each instantiation requires one AND, four AND-OR and four XOR gates as shown in Fig. 3. No further optimization for this part of the adder is possible. Therefore, the *add_db* and RP data structures are updated for the first four sum bits.

Generate CSAs (Fig. 3) for rest of the bits starting from 4 to n-bits.

This operation instantiates *n/4*-1 CSAs. Each instantiation of a CSA requires the instantiation of 2 inverters, 3 AND-OR (INVERT) gates, 11 XOR gates and 3 MUXs as shown in Fig. 4. Since, it is possible to replace fast arriving carry CSAs with ripple carry adders, therefore do not update the *add_db* and RP data structures.

Once all the instantiations are completed, the final step is the adder optimization using timing analysis. Following section explains this procedure and the algorithm involved.

4.2 Adder optimizer

The first step in adder optimization is to calculate the maximum adder delay (*maxDelay*) in pico second (ps) at the output of the CSAs (instantiated above).

The procedure below illustrates the adder optimization.

```
Calculate maximum adder delay:
    int numBits = n;
    int MaxLevel = log(n-1);
    int maxDelay = 0;
    for (i=4; i < numBits; i++)
    {
      delay[i]=gp_tree[MaxLevel][i].delay;
      if (delay[i] > maxDelay)
        maxDelay = delay[i];
    }
```

In this procedure *maxDelay* is obtained, by probing the total path delay at each output bit of the CSA and comparing it with *maxDelay*. If the delay at any bit is greater than *maxDelay* then update the *maxDelay* with the delay at that bit position. Final step in adder optimization is to replace each fast carry path CSA, with an area optimized ripple carry adder. This is performed by first replacing a 4-bit CSA with a ripple carry adder and calculating the 4-bit group delay (*maxGroupDelay)*. If the delay does not increase (*maxGroupDelay* < *maxDelay*) then commit the move and synthesize ripple carry adder, else synthesize CSA as shown below.

```
Area optimization
    int maxGroupDelay = 0;
    for (i=4; i<numBits; i = (i + 4))
    {
      Instantiate_Ripple_Carry();
      for (j=i; j < i+4; j++)
      {
       delay[j] =  gp_tree[MaxLevel][j].delay;
        if (delay[j] > maxGroupDelay)
           maxGroupDelay = delay[j];
      }
      if(maxGroupDelay > maxDelay)
        Synthesize_Ripple_Carry();
      else
        Synthesize_CSA();
    }
```

In the above procedure the *Instantiate_Ripple_Carry* only instantiates a ripple carry adder without updating the *add_db* and RP data structures. *Synthesize_Ripple_Carry* and *Synthesize_CSA* routines not only instantiate the adders but also update the *add_db* and RP data structures and produce the output sum.

5 PERFORMANCE COMPARISON

The performance of the adders synthesized by the generator is compared with the BK adders [7] [9], and KS [15]. The BK and KS adders are synthesized using Synopsys Design Compiler (DC). While the new adders are first synthesized using the proposed generator and the generated netlist is post processed by the DC. Each adder is synthesized using ten different technology libraries under worst case operating conditions and different wire-load models.

Fig. 5 and Fig. 6–show the delay and area comparison between different adders using a 0.18u standard cell library under worst-case conditions and library18_conservative wire-load model. It is clear from the area-delay reports generated by DC, that the adders generated by the proposed generator are comparable in terms of delay (good in some cases and slightly slow in others) to BK, and KS. In terms of area, adders generated by this proposed generator are always 20%-50% better. Fig. 7 shows the RP layout of a 64-bit adder using 0.18u technology. The adder bits increase from right (LSB) to left (MSB), inputs are at the top, while the sum is at the bottom. In this figure, one can easily observe the placement of cells in rows and columns positions to get a rectangular layout.

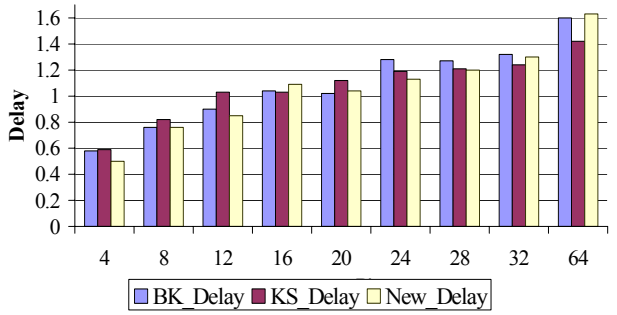

Fig. 5. Delay comparison of BK, KS, and NEW adder in 0.18u Tech.

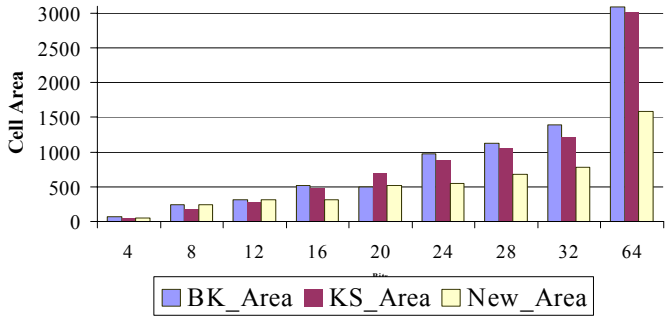

Fig. 6. Area comparison of BK, KS, and NEW adder in 0.18u Tech.

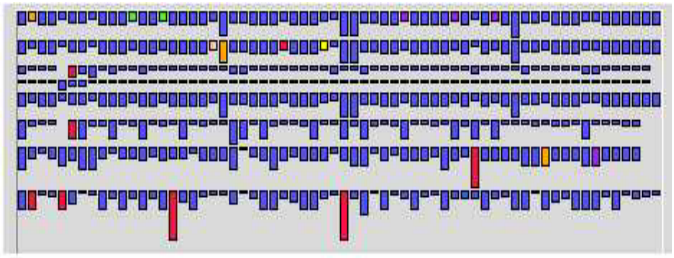

Fig. 7. Graphical view of the relatively placed gates of a 64-bit adder**.**

6 CONCLUSIONS

In this paper we presented a generator for the design of area-time optimal adders with layout information. Adders of up to 1024 bits are generated and are compared with those generated by commercially available synthesis tools. The proposed adders are found to be 20% to 50% smaller and with comparable delays. The reduction comes from a judicious selection of ripple carry or carry select adders based on the computation of delays. When performance is being met the faster carry select adders are replaced with slower ripple carry adders. Accurate computation of delays using the specific technology library, wire-load models and delay/area goals are employed in making the above decision. Further, information from the generator is used to determine the relative location of cells in the generated netlists. This generator has been integrated into a commercially available high-performance datapath design tool.

7 ACKNOWLEDGEMENTS

The authors acknowledge the support of Synopsys Corporation and the King Fahd University of Petroleum & Minerals for this work.

8 REFERENCE:

1. Se-Joong Lee et al., “480ps 64-bit race logic adder”, Digest of Technical Papers Symposium on VLSI Circuits, 2001 Page(s): 27 –28.
2. Sklansky, J., “Conditional-Sum Addition Logic”, IRE Trans. EC-9, No. 2, June 1960, pp. 226-231.
3. Mathew, S.K. et al.; “Sub-500-ps 64-b ALUs in 0.18u SOI/bulk CMOS: design and scaling trends”, Solid-State Circuits, IEEE Journal of , Volume: 36 Issue: 11 , Nov. 2001 Page(s): 1636 -1646
4. Brent, R.P.; Kung, H.T., “A regular layout for parallel adders”, IEEE Transactions on Computers, vol.C-31, (no.3), March 1982. p.260-4.
5. R. Zimmermann, “Efficient VLSI implementation of modulo(2n+1) addition and multiplication”, Proc. 14th IEEE Symposium on Computer Arithmetic, Apr. 1999pp. 158-167.
6. Beaumont-Smith, A.; Lim, C.-C., “Parallel prefix adder design”, Proc. 15th IEEE Symposium on Computer Arithmetic, 2001, Page(s): 218 –225.
7. Dozza, D.; Gaddoni, M.; Baccarani, G. "A 3.5 ns, 64 bit, carry-lookahead adder", Proc. 1996 IEEE International Symposium on Circuits and Systems. Circuits and Systems, vol.2, p.297-300.
8. V. G. Oklobdzija and E. R. Barnes, "Some Optimal Schemes For ALU Implementation In VLSI Technology," Proceedings of the 7th Symposium on Computer Arithmetic, pp. 2-8.
9. Wei, B.W.Y.; Thompson, C.D., “Area-time optimal adder design”, IEEE Trans. on Computers, Volume: 39 Issue: 5 , May 1990 Page(s): 666 –675.
10. Chan, P.K.; Schlag, M.D.F.; Thomborson, C.D.; Oklobdzija, V.G., “Delay optimization of carry-skip adders and block carry-lookahead adders“, Proc. 10th IEEE Symposium on Computer Arithmetic, 1991 Page(s): 154 –164.
11. Becker, B.; Drechsler, R.; Molitor, P. , “On the generation of area-time optimal testable adders”, Computer-Aided Design of Integrated Circuits and Systems, IEEE Trans. on , Vol: 14: 9 , Sept. 1995 Page(s): 1049 –1066.
12. Hsu, J.; Bair, O., “A Compiler For Optimal Adder Design”, Proc. IEEE Custom Integrated Circuits Conference,1992 Page(s): 25.6.1 -25.6.4
13. R. E. Ladner and M. J. Fischer, “Parallel Prefix Computation”. Journal of the ACM, vol. 27, no. 4, *October 1980, 831-838*
14. V. G. Oklobdzija, "High-Performance System Design: Circuits and Logic",Book, IEEE Press, July, 1999.
15. Knowles S.,” A Family of Adders”, Proc. of the 14th Symposium on Computer Arithmetic, 1999 pp. 30-34
16. A. A Farooqui, V. G. Oklobdzija, F. Chehrazi, "Multiplexer based adder for media signal processing", Proc. 1999 International Symposium on VLSI Technology, Systems, and Applications, Taipei, Taiwan, R.O.C., Page(s): 100 -103, 8-10 June 1999.