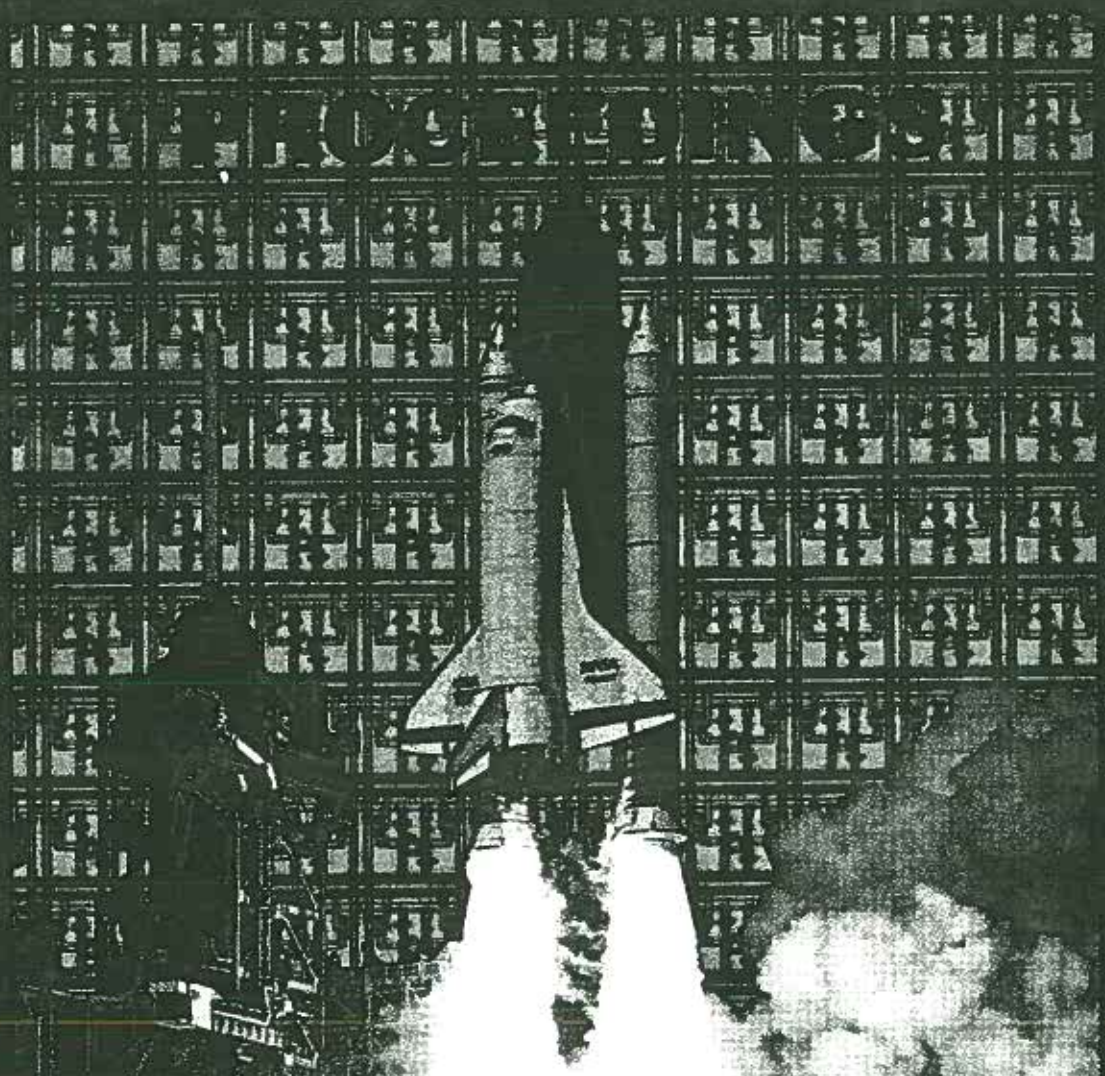
ISCAS '99



The 1999 IEEE International Symposium on Circuits and Systems

May 30 - June 2, 1999

Orlando, Florida

BOARD-LEVEL PROTOTYPE VALIDATION: A BUILT-IN CONTROLLER AND EXTENDED BST ARCHITECTURE

Gustavo R. Alves[1,2], Telmo Amaral[2] and José M. Martins Ferreira[2]

[1]ISEP / DEE
Rua de S. Tome
4200 Porto – PORTUGAL

[2]FEUP / DEEC
Rua dos Bragas
4000 Porto – PORTUGAL

ABSTRACT

Prototype validation is a major concern in modern electronic products design and development. Simulation, structural test, functional and timing debug are all forming parts of the validation process, although very often addressed as dissociated tasks. In this paper we describe an integrated approach to board-level prototype validation, based on a set of mandatory / optional BST instructions and a built-in controller for debug and test, that addresses the late mentioned tasks as inherent parts of a whole process.

1. INTRODUCTION

Increasing complexity and quality demand on electronic products combined with shortening time-to-market is creating a bottleneck on the prototype validation phase. Any sound validation strategy must favour the overlapping of the several verification steps with the design flow and first prototype releases, to prevent delays and augment design celerity. Prototype validation is usually formed by the following steps: *simulation, structural test, functional and timing debug*. These typically address three classes of errors:

- Human errors in the specification or design.
- Technological, implementation or manufacturing errors. These include defective components, soldering problems, broken or short lines, etc.
- Errors related to the tools. These include errors associated with the synthesis, model generation, simulation or layout (at the IC or PCB level) tools.

Simulation provides the first and best platform for detecting and debugging human errors in the prototype specification or design, although this process in itself is also prone to human / tools errors. The values obtained during simulation provide a database of golden vectors that can later be used for the prototype functional and timing debug phase [1]. *ATPG* and *Fault simulation* are done to create the test program and a fault dictionary able to assist on the diagnosis of faults detected during structural test. *Static timing analysis* helps to determine the maximum clock frequency, by revealing the longest paths within the design. Pin-to-pin and other delay types are also calculated during this process. *Structural test* addresses the second class of errors. This step is greatly simplified if the components support BST [2]. Existence of BIST capabilities also helps to diminish ATE requirements. Testing a board proceeds in three main steps: testing the BST infrastructure itself, testing the interconnects, and the components (mainly through the activation of IC-level BIST functions). Other advantages of using BST include simple test interface, assistance on functional debug and test, and availability during field operation debugging [3,4,5]. During *functional debug* the golden vectors extracted from simulation are compared against the values captured on the prototype, thus covering the third class of errors. This process is usually carried out step by step, sometimes with a reduced clock frequency and generally involves a reduced number of vectors. The *timing debug* phase is done with the prototype working on its normal operating speed. Errors not detected during the structural test (due to the fault models used) or functional debug (due to the reduced clock frequency) have to be detected and diagnosed during this last verification step.

This paper describes an integrated solution to board-level prototype validation based on a set of optional BST instructions to be supported by 1149.1-compatible components and a built-in controller for debug and test. The proposed solution addresses the four verification steps as follows: functionality of the optional instructions is included in each IC model and the built-in controller model is included for system level simulation, to reduce differences between simulation / prototype debugging environments. Structural test is covered by mandatory BST instructions, and board-level BIST is supported by the built-in controller. Functional debug is covered by both mandatory and optional BST instructions. Synchronisation between functional and test logic is guaranteed by the built-in controller [3,4]. Timing debug is covered by both optional BST instructions and the built-in controller.

Our requisites included a minimal overhead and interference with the component functional logic for the optional BST instructions, and reusability of a board-level BIST processor [6].

2. PROTOTYPE DEBUG AND TEST REQUIREMENTS

The initial phase of our approach included the identification of the prototype debug and test requirements and the conversion of this requirements into operations implemented by both mandatory and / or optional BST instructions, and / or instructions executed by the built-in controller. Requirements analysis covered characteristics of simulation and debug tools, current debug and test techniques, and debug and test mechanisms accessible through BST [1,3,7,8,9]. The analysis process led to a "simplified" debug and test model with five operation types:

- Control, Observation and Verification (COV)
- Single Step (SS)
- Breakpoint (BP)
- Real Time (RT) analysis, and
- Control of Internal resources and Test program flow (CIT).

0-7803-5471-0/99/$10.00©1999 IEEE

Next, a set of criteria was defined so as to allow an exhaustive dissolution of each operation type in a roll of individual operations. The following list presents the criteria considered for each operation type. Individual operations were obtained by examining minutely each criteria combination. The last phase consisted of analysing the individual operations included in each operation type and converting these into specifications of instructions implemented by the built-in controller or the BST infrastructure. The description of this rather extensive process would go beyond the scope of this paper. For the sake of simplicity and presentation clarity it was decided to omit it.

Due to the requirement of reusing the board-level BIST processor it was decided to design the built-in controller as a dual-processor architecture. One of the processors is responsible for the control of the test logic (the board-level BIST processor), while the other is responsible for the control of the system functional logic, and the synchronisation between the functional and the test logic.

3. A DUAL-PROCESSOR BUILT-IN CONTROLLER FOR DEBUG AND TEST

3.1. Control of test logic

The processor controlling the test infrastructure is an enhanced version of the board-level BIST processor. The original instruction set allowed the control of the low-level TAP operations. A deserializer, an interface to an external dual-port FIFO and a number of new instructions were added, resulting in the instruction set presented in table 1 (new instructions are shadowed).

TAP operations	
SELTAP (0,1)	Selects the BST chain to be controlled by the following instructions.
TRST	Forces an asynchronous reset through the /TRST output.
TMS0, TMS1	Forces a state transition in the internal BST logic of each IC.
NSHF	N bits will be shifted into the selected chain. Bits shifted out of the chain are not compared.
NSHFCP	N bits will be shifted into the selected chain. Bits shifted out of the chain are compared with their expected value (using mask bits).
NTCK	Applies N test clock cycles, while keeping TMS at "0". N represents the contents of the internal 24 bit counter.
NCSHF	Bits shifted out of the selected chain are shifted into the same chain and stored in the selected temporary buffer.
NCSHFCP	Bits shifted out of the selected chain are compared with their expected value, shifted into the same chain and stored in the selected temporary buffer. Mask bits are used to discard don't care bits.
NSHFB2C	N bits stored in the selected temporary buffer will be shifted into the selected chain. Bits shifted out of the chain are not compared.
NSHFCPB2C	N bits stored in the selected temporary buffer will be shifted into the selected chain. Bits shifted out compared with their expected value. Mask bits used to discard don't care bits.
STCK	Applies test clock cycles while synchronism input channel A is at "1", TMS is kept at "0"
Internal control and synchronisation	
STMPB0, STMPB1	Selects the internal 2048 x 1 bit temporary buffer 0 or 1 for storing the values shifted out of the selected chain.
LD C16, N	Loads the internal 16-bit counter with the number of test clock (TCK) cycles to be applied.
LD C24, N	Loads the internal 24-bit counter with the number of test clock (TCK) cycles to be applied.
JPE Address JPNE Address	Conditional jumps based on the state of the internal error flag.
SSA0/1 SSB0/1	Forces a logical value (0,1) on the synchronism output channel A or B.
WSA0/1, WSB0/1	Waits for a logical value (0,1) on the synchronism input channel A or B.
HALT	Terminates test program execution.

Table 1: Instruction set supported by the processor controlling the test infrastructure.

NCSHF and NCSHFCP are used to observe and verify the contents of scan chains without modifying the current value (the value captured on the scan output is placed at the scan input, resulting in a circular shift of the scan chain contents – number of cycles must match the scan chain extension). The values shifted out of the active chain are internally stored in a previously selected temporary buffer (using STMPBx) and deserialized into 8-bit words placed on an external dual-port FIFO for outside observation. NSHFB2C and NSHFCPB2C are used to shift the selected temporary buffer contents into the active scan chain. These instructions enable the debug & test program to return the active scan chain to a former saved state. STCK enables the second processor to control TCK, through synchronism channel A, as during RT operations it is sometimes necessary to supply an unknown number of TCK cycles.

3.2. Control of functional logic

The instruction set of the processor controlling the functional logic is presented in table 2. A first group of instructions implement COV operations on directly accessible pins. A second group of instructions controls the system clock output for SS operations. A third group of instructions controls the system clock output for BP operations. CSTRETCH_N partially implements the cycle stretch technique [1], that consists of selectively stretching the clock-cycle length for isolated cycles prior to a detected failure. The theory is that when the cycle, where a long-path is exercised, is stretched then enough time will be allowed for the correct data to be captured / registered. To accomplish this, the process has to be run iteratively, with successive cycles stretched, to find when the subsequent external failure has indeed been eliminated. When it has, the current stretched cycle is the one that exercises the long-path. The last group of instructions controls synchronism channels and the internal resources. STORE C24 stores the contents of the internal 24-bit counter in an external dual-port FIFO. This counter is used for implementing the cycle stretch technique, so as when the time-related fault is no longer detected, its contents identify the exact cycle where the long-path is exercised.

Instructions supporting COV operations	
RSTOUT [i]	Resets output [i].
SETOUT [i]	Sets output [i].
READ [i]	Selects input [i] which remains connected to the data output
JZ [i], address	Jumps to selected address if input [i] is 0 (or 1).
WAIT_WHL_Z [i]	Remains in this instruction while input [i] is 0 (or 1).
Instructions supporting SS operations	
CLK	Applies a single system clock cycle.
CLK_N	Applies N system clock cycles.
Instructions supporting BP operations	
CLK_WHL_Z [i]	Applies system clock cycles while input [i] is 0 (or 1).
Instructions supporting RT operations	
START_CLK	Initiates the application of system clock cycles.
STOP_CLK	Stops the application of system clock cycles.
CSTRETCH_N	Applies N system clock cycles, stretching a particular clock cycle.
Instructions supporting control of internal resources and synchronism	
DJNZ address	Decrements C24 and jumps to selected address if not 0.
LD C24, N	Loads the internal 24-bit counter with N.
STORE C24	Stores the contents of C24 into external FIFO.
JP address	Unconditional jump to selected address.
SSA0/1, SSB0/1	Forces a logical value (0,1) on the synchronism output channel A or B.
WSA0/1, WSB0/1	Waits for a logical value (0,1) on the synchronism input channel A or B.
HALT	Terminates program execution.

Table 2: Instruction set supported by the processor controlling the functional logic.

4. OPTIONAL BST INSTRUCTIONS

The optional BST instructions defined give support to BP and RT operations. For BP operations the BS register is configured to detect a condition corresponding to values present at input pins or outputs from the IC functional logic. For RT operations the BS register is configured to:

- Store a sequence of two contiguous vectors.
- Store a sequence of two contiguous vectors after a certain condition is found.
- Store a sequence of two contiguous vectors until a certain condition is found.

4.1. Detect condition

The goal is to activate a Condition Detected Output (CDO) pin when the comparison between the vector present at the BS register PIs and the vectors stored at the capture/shift and update stages results true, according to one of eight condition types:

- Equal to expected vector (vector compared through a mask)
- Different from expected vector (compare with mask)
- Greater than limit A (vector > limit A)
- Greater/equal to limit A (vector ≥ limit A)
- Lesser than limit A (vector > limit A)
- Lesser/equal to limit A (vector ≤ limit A)
- Between limit A & B (limit A < vector < limit B)
- Outside limit A or B (vector < limit A or vector > limit B)

The optional instruction SEL_COND places a 3-bit test register between TDI-TDO, which selects the type of condition to be detected. The expected vector (or limit A) is stored in the update stage and the mask (or limit B) is stored in the capture/shift stage. To place the expected vector in the update stage it is necessary to shift the Sample/Preload instruction and then shift in the expected vector. During *Update-DR*, the vector is stored in the update stage. To place the mask in the capture/shift stage it is necessary to shift in the optional instruction DET_COND and then shift in the mask. The values present in the BS register when DET_COND is active are not modified in *Capture-DR* or *Update-DR* states. At the end of the shift process the mask is stored in the capture/shift stage. Condition is evaluated while TAP controller is in *Run-Test/Idle*, where CDO exhibits the result. TCK has no effect on the evaluation process. To support this operation the BS cells have to be modified to the structure illustrated in fig. 1.

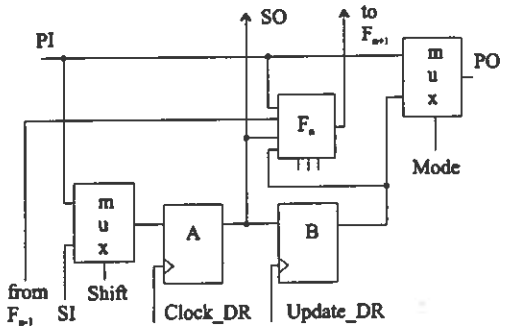


Fig. 1: Modified BS cell supporting instruction DET_COND.

The F_n block evaluates the partial condition at each cell, taking into account the result from the previous cell, and feeds the result to the next cell. CDO is connected to F_n of the BS cell closest to TDO. The F_n block of the BS cell closest to TDI receives the result from a Condition Detected Input (CDI) pin. These two extra pins (CDI and CDO) allow several BST components to be cascade for detecting complex conditions that may include several hundreds of functional pins. CDO is connected to a generic input pin of the built-in controller for BP implementation, using instruction CLK_WHL_Z [i].

4.2. Store sequence

The goal is to store a sequence of two contiguous vectors on the BS register, one vector in the capture/shift stage and the other in the update stage. To store one sequence the optional instruction STORE_SEQ is first shifted in and the TAP controller is placed in *Run-Test/Idle*. While at this state the value present at each BS cell Parallel Input (PI) is captured on the TCK rising edge. On the falling edge the value present in the capture/shift stage is registered on the update stage. To read the stored sequence the TAP controller is placed in *Shift-DR* (contents of the capture/shift stage do not change during *Capture-DR*) and the first vector is shifted out. To read the second vector (stored in the update stage) the TAP controller is first placed in *Exit2-DR*, via *Pause-DR*. During *Exit2-DR* the value stored in the update stage is captured by the capture/shift stage. The TAP controller is placed in *Shift-DR* and the second vector is shifted out. Fig. 2 illustrates a BS cell that implements this instruction.

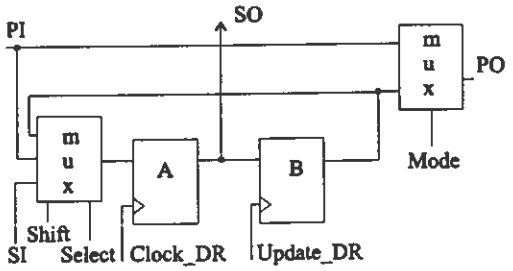


Fig. 2: Modified BS cell supporting instruction STORE_SEQ.

4.3. Store sequence after condition

The goal is to store a sequence of two contiguous vectors after a certain condition. The condition detection and the sequence storing correspond to the functionality defined in the previous optional instructions. To implement the optional STORE_AFTER_COND instruction, a dedicated FSM with states *monitor condition*, *capture sequence I*, *capture sequence II*, and *end of sequence*, was added. The expected vector (or limit A) and the mask (or limit B) are entered the way defined for instruction DET_COND. The FSM is initially at *monitor condition*. Condition is evaluated while the TAP controller is in *Run-Test/Idle*. When the condition results true, the FSM enters (on TCK falling edge) the *capture sequence I* state. While at this state the value present at each BS cell PI is captured by the capture/shift stage and afterwards registered in the update stage. The next TCK falling edge takes the FSM into the *capture sequence II* state. While at this state the value present at the BS cell

PI is captured by the capture/shift stage and the update stage retains its previous value. The next TCK falling edge takes the FSM into the *end of sequence* state and the stored sequence may now be shifted out following the steps defined for instruction STORE_SEQ.

Merging the BS cells illustrated in fig. 1 and fig. 2 allows the implementation of this optional instruction. Fig. 3 presents the time diagram of the STORE_AFTER_COND instruction.

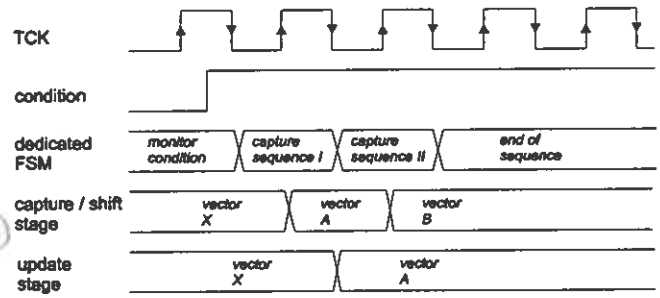


Fig. 3: Time diagram of store after condition.

4.4. Store sequence until condition

The goal is to store a sequence of two contiguous vectors until a certain condition. Condition detection corresponds to detecting a logic '1' at CDI. Two states of the FSM (*monitor condition* and *end of sequence*) are used to implement this optional instruction, named STORE_UNTIL_COND. The FSM is initially at *monitor condition* state. CDI is monitored while TAP controller is in *Run-Test/Idle*. While at this state the value present at each BS cell PI is captured by the capture/shift stage and afterwards registered in the update stage. When CDI is asserted, the FSM enters (on TCK falling edge) *end of sequence* state and the capturing activity is ceased. The stored sequence may now be shifted out following the steps defined for STORE_SEQ. The BS cell illustrated in fig. 2 is also able to implement this optional instruction.

5. IMPLEMENTATION

The built-in controller is implemented in an EPF10K30. The MaxPlus II development system is able to generate a gate-level VHDL description of the design, thus enabling an easy transition to other development systems. The complete set of optional instructions is implemented, together with the mandatory BST infrastructure, in FPGAs emulating the '244 (an 8-bit non-inverting buffer) and the '373 (an 8-bit latch with tri-state outputs). The complete system, including two memories containing the programs for each processor, is now undergoing extensive functional and timing co-simulation. The test programs are initially written in assembly, and an in-house developed application generates the correspondent Memory Initialisation Files (MIFs). These files are read each time a new simulation is performed. The system-level model consists of the individual models of each component (memories, built-in controller, '244, and '373) interconnected for system-level co-simulation. The simulation tool accepts mixed-level modelling, so each component model may either correspond to a behavioural or gate-level model.

6. CONCLUSION

A set of prototype debug and test requirements was initially identified and converted into five basic operation types forming a "simplified" debug and test model. Individual operations included in each operation type were obtained by considering all possible combinations of the gathered criteria. These were then analysed and converted into specifications of instructions implemented by the BST infrastructure or by a board-level built-in controller for debug and test. Mandatory / optional instructions described in the Std. were first considered, and a set of new optional instructions for debug support was then defined. These included: DET_COND - concurrently detecting conditions in RT at the BS register (corresponding to values appearing at IC pins); STORE_SEQ - storing sequences of two contiguous vectors at the BS register; STORE_AFTER_COND - storing sequences of two contiguous vectors after a certain condition; STORE_UNTIL_COND - storing sequences of two contiguous vectors until a certain condition. Estimated overhead for the circuitry needed to implement the optional instructions is approx. 100% in relation to the mandatory BST infrastructure. This number suggests that for an IC where the mandatory BST infrastructure represents an overhead of 2-3%, implementing the optional instructions would raise this value to 4–6%. The built-in controller was implemented as a dual-processor architecture. One of the processors controls the board-level scan chains, while the other controls the system clock, thus guaranteeing synchronisation between system functional and test logic.

The proposed solution is now undergoing extensive system-level functional / timing co-simulation. Small and large programs are being run in the simulation environment, and a database of golden vectors is being extracted for later comparison with values captured during system normal functioning.

7. REFERENCES

[1] Jerry Katz, "A Case-Study in the use of Scan in µSPARC™ testing and debug," in *ITC*, pp. 70-75, 1994

[2] IEEE Std 1149.1. *IEEE Standard Test Access Port and Boundary Scan Architecture*. IEEE Stds. Board, May 1990.

[3] M. Lefébvre, "Functional Test and Diagnosis: A Proposed JTAG Sample Mode Scan Tester," in *ITC*, pp. 294-303, IEEE Computer Society Press, 1990.

[4] R. Sedmak, "Boundary-Scan: Beyond Production Test," in *ITC*, pp. 415-420, IEEE Computer Society Press, 1994.

[5] B. Whittaker and N. Doherty, "Boundary Scan Helps Solve Field Failures," in *Evaluation Engineering*, Vol. 35, n° 10, pp. 26-30, Oct. 1996.

[6] J. Ferreira et al., "Automatic Generation of a Single-Chip Solution for Board-Level BIST of Boundary Scan Boards," *EDAC Proc.*, pp. 154-158, .Mar. 1992.

[7] L. Whetsel, "An IEEE 1149.1 Based Logic/Signature Analyzer in a Chip," *in ITC.*, 1991, pp. 869-878.

[8] T. Blakeslee and J. Liband, "Real-Time debugging techniques: Hardware-assisted real-time debugging techniques for embedded systems," in *Embedded Systems Programming*, vol. 8, n° 4, 1995.

[9] Bruce Erickson, "Selecting the Right Debugging Tool," in *Electronic Design*, pp. 83-98, Oct. 1995.