Automatic Generation of Self-Test Programs - A New Feature of the MIMOLA Design System

Gerd Krüger

Institut für Informatik u. Prakt. Math., Universität Kiel
Olshausenstr. 40-60, D-2300 Kiel 1, W.-Germany

Abstract

A method to automatically generate functional self-test programs for arbitrary processor systems including microprogrammable and custom designed special purpose types is presented. Only commonly available user information is needed, but gate-level details can be utilized as well. The generated self-test programs perform user guided tests for memory function and register decoding, functional or gate-level derived tests for combinational modules, and a machine status check to detect undesired side-effects. The programs are given in the micro- or machine code of the target system, ready for execution. First applications have shown promising results.

1. Introduction

The ever growing complexity of modern digital circuits creates serious problems for the verification of the system's correct function. Formerly successful methods cannot be used any more, because they are based on a gate-level description that if available might cover tens of thousands of gates. Test generation on this level becomes prohibitively costly.

Functional testing offers an alternative. Especially for microprocessors methods have been developed that use only readily available user information to derive a sequence of machine test instructions[1,2,3]. Correct execution of this sequence is supposed to sufficiently validate the circuit's correctness. These methods are fixed upon a given set of standard machine instructions for sequential data processing. They do not take into account the implementation of the instruction, common use of hardware resources, or an underlying microprogramming level. Rigid use of the proposed algorithms might lead to an unwieldy amount of symbolic test instructions[4].

Lai[5] presents a more comprehensive testing methodology. For all digital systems he claims that test generation can be done within four independent steps: functional specification, functional analysis, test case synthesis, and program code synthesis. Only the first two steps of his methodology are implemented. A demonstration on a standard sequential architecture (PDP 11 minicomputer) only covered stuck-at faults on data paths.

This work was supported by the Federal Ministry for Research and Technology (BMFT), W. Germany

A new component of the MIMOLA - Software System (MSS2)[6,7] now offers automatic generation of self-test programs especially for microprogrammable and custom designed special purpose processor systems (Fig. 1). In addition to the test programs, the test code generation software delivers valuable information about the testability of the system under test. Internal modules with poor controllability or observability are reported. As a part of the hardware design process using the MIMOLA design system, self-test programs can be generated on various design stages and for several alternative solutions. With a couple of iterative steps, feeding back the testability information along with other rating data, an optimum of performance, cost, and testability is approached. Hardware design without consideration of testability is obsolete.

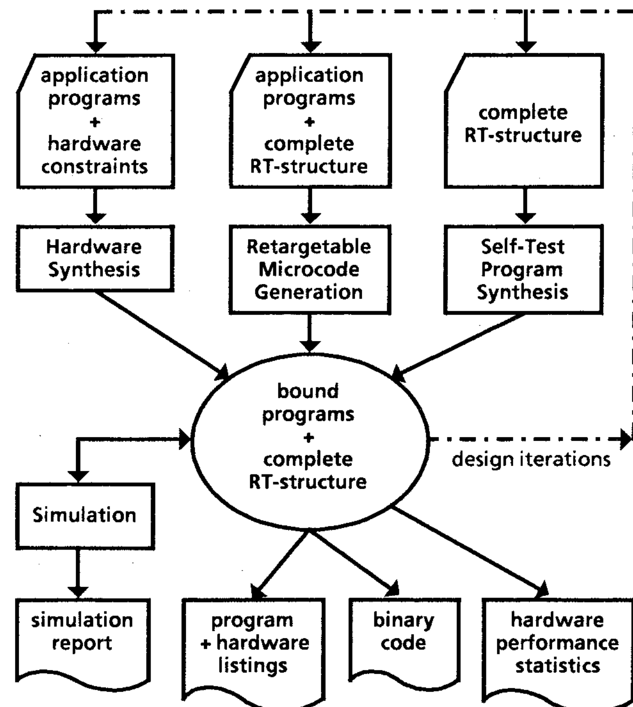

Fig. 1 : The MIMOLA Software System 2 (MSS2)

0738-100X/86/0000/0378$01.00 ©1986 IEEE

For processor systems that have not been designed with MSS2, self-test programs can be generated as well[8]. A functionally equivalent model of the system must be provided, using the computer hardware description language (CHDL) part of MIMOLA (**M**achine **I**ndependent **M**icr**O**programming **LA**nguage).

2. Global Concepts

The test generation part (MSST) of the MIMOLA software system is a tool that automatically produces self-test programs based on user specified test patterns for internal register-transfer level modules. The main features of MSST are:

- Only the common user information about the processor system (instruction formats, operation codes) is absolutely nescessary.
- A register-transfer structure[9] is used to specify the system under test. The original system structure should be preferred, but any functionally equivalent model of interconnected registers, memories, arithmetic-logic units, multiplexers, and busses is acceptable.
- Implementation details down to the gate level can be incorporated in the circuit model and in the fault models, thus being considered for test generation. In this case simple gates are treated like RT-modules.
- In addition to modules and connections along the data paths, address- and control logic are specified. Tests especially tailored for these parts of the system can be generated.
- Expensive external test hardware (a test machine) is not needed. Self-test programs run on the system under test. Monitoring the program counter or program execution time is sufficient to detect and roughly localize a fault.
- A generated self-test program consists of four phases:
  (1) Initialization of registers and memory locations,
  (2) tests for the register decoding and memory addressing function along with functional tests for the storing modules,
  (3) functional or gate level derived tests for the combinational modules,
  (4) verification of the current machine status in order to detect undesired side-effects.
- For any kind of misbehaviour of a system-internal RT-module special testpatterns (input patterns for this module) can be provided in a library file. Default patterns (0101...01 and 1010...10) are used if no entry is found in the library.
- Modules that cannot be controlled or observed via the data path can be checked indirectly by specifying special address or control signals for the test of other modules.
- The generated self-test programs are given directly in the micro- or machine-code of the system under test, ready for execution. For documentation outputs on a higher level (MIMOLA) are available. The amount of details shown (path width, modules traversed, function codes) can be selected.

Self-test Programs

On the way to an economically attractive solution of the testing problem several good reasons call for the use of self-test programs. The obvious restriction to programmable systems is not really significant. Today nearly all of the more complex switching tasks are realized by processors. Integrated circuits that contain only parts of a processor system can be tested in the user environment they are designed for. Besides, a simple test environment with a processor structure is easily constructable. The self-test concept remains applicable.

The minimal configuration for the use of self-test programs must include :

- A comparator, or a test on zero after a subtraction (addition of the two's complement) or an "exclusive OR" operation, and
- a way to change the flow of the program, depending on the result of the comparator (jump conditionally), in order to report the occurrence of an error.

Hardly any processor does not provide these operations.

Two of the fundamental goals of the MSST testing concept in combination are within reach only for self-test programs:

- Total independence of additional test hardware that is not covered by the circuit model specification,
- no limitation to a class of "easily testable" circuits, e.g. circuits that follow rigid design rules or contain built-in test aids (e.g. Scan Path).

For any type of processor system a more or less successful test generation should be possible. The rate of fault coverage achievable by the generated self-test programs depends on the controllability and the observability (as seen from the comparator) of the system under test. Basically it is unquestioned that carefully designed self-test programs show good fault coverage, e.g. for off-the-shelve microprocessors[2,3,10]. This holds especially for tests on the microprogramming level with its extended facilities to control and observe individual modules[11]. Localisation of faults is a far more difficult task and MSST aims at it only secondarily.

Self-test programs are equally suited to be employed by manufacturers and users. The advantages of external test equipment shrink at the rate that system-internal testpoints become inaccessible. When testing a fully integrated system, all the external equipment can do is to supervise the execution of a test program (and check the electrical parameters on external I/O pins). The step to self-test programs then is short and obvious, regarding the cost advantage.

The MSST Approach

According to the MIMOLA philosophy the generation of self-test programs is not supposed to be a fully

automated solution. The human test engineer must be enabled to feed his experience into the test programs. Using MSST it is up to him to decide about fault models, test strategy, fault coverage, and test length. In a test pattern library he can supply MSST with input patterns for all internal modules and functions specified in the circuit model (Fig.2).

```
\Sram (mainram)
 #5555 Addr.: #0000
 #AAAA Addr.: #FFFF
 ...
\Acomp
 #0000 #0001 #0002 #0004 #0008 #0010
 #0020 #0040 #0080 #0100 #0200 #0400
 #0800 #1000 #2000 #4000 #8000 #FFFF
\END
 (* #... denotes hexadecimal numbers *)
```

Fig. 2: Extract from a MSST test pattern library (for the system shown in Fig. 4)

For memory modules the input patterns may cover not only data but address inputs as well. Any functional memory test procedure can be specified by a sequence of data and address input patterns. It is then translated into automatically generated self-test program code.

For combinational modules the specified input patterns correspond to "primitive cubes of a logic fault" for the D-algorithm. Just like these "cubes" represent fault detecting input patterns on the gate level, patterns in the MSST library should be designed to detect faults inside of register-tranfer level modules. The main difference lies in the possibly far more complex module function and in the fact that input patterns now are bit-vectors of varying length instead of single bits.

If the gate level structure of a combinational module is known, then the input patterns to test for faults inside the module can be generated with any one of the gate level test generation procedures (e.g. D-algorithm or derivations). Considering the limited complexity of single RT modules this should be manageable.

Fig. 3 shows the four parts of the self-test programs generated by MSST. Default initialization is omitted if special input patterns for the memory modules are specified in the library. For all memory modules that have been initialized by default a simple check of the address decoding function is carried out. Tests for the memory function follow. One step to test a memory module consists of loading an input pattern, reading it out again and comparing it with the original. On equality the (micro-) program counter is incremented and the test program continues. Otherwise a jump to a user defined "errorexit" label is executed. Loading is not done if the last value written into the memory in an earlier step is equal to the current input pattern.

For combinational modules a test for a certain module function can be performed within one symbolic self-test instruction for each input pattern. The program flow jumps to "errorexit" if the output of the module differs from the expected value while the specified input patterns are applied and the module is controlled to execute the desired function. The machine status is verified by checking all registers and memory cells that have been accessed during test program execution wether they still hold the expected value.

a) <u>Initialization steps:</u>
```
register:=initvalue,
prog.counter:="increment"
(prog.counter);
```

b) <u>memory test steps:</u>
```
register:=input pattern,
prog.counter:="increment"
              (prog.counter);
prog.counter:=
     IF register=inp.pattern
     THEN "increment" (prog.counter)
     ELSE errorexit;
```

c) <u>test steps for combinational modules:</u>
```
prog.counter:=
     IF mod.output
       (mod.function, inp.pattern)
       = expected value
     THEN "increment" (prog.counter)
     ELSE errorexit;
```
or :
```
register:=mod.output
          (mod.function,inp.pattern),
prog.counter:="increment"
              (prog.counter);
prog.counter:=
     IF register=expected value
     THEN "increment" (prog.counter)
     ELSE errorexit;
```

d) <u>machine status verification:</u>
```
prog.counter:=
     IF register=last value loaded
     THEN "increment" (prog.counter)
     ELSE errorexit;
```

Fig. 3: Symbolic samples of the generated self-test program steps

A chance to localize a detected fault is given by simply keeping track of the individual module functions activated so far. A fault should be looked for primarily within those functions of the activated hardware that are executed for the first time. Statistics on module function occurrences (where first) and their frequencies in the generated self-test program can be obtained by MSST and other MSS2 output.

3. Realization

The main task of MSST now is to generate program code for the symbolic test instructions shown in Fig. 3. Depending upon the specification of the processor's control section sequences of micro- or machine code will be generated. In any case the generated program is situated on the lowest programming level in the circuit model. If possible a microprogramming level should always be specified, because it offers extended capabilities for testing and it receives special support from MSS2.

Lai[5] does not mention any approach to realize his concept of test program synthesis. Bellon et al.[3] conclude that "writing test programs is a tedious job" and go back to external test circuitry. In order to produce program code, paths within the hardware leading to the modules to be tested must be found and control signals to activate these paths must be generated. Similar to the D-algorithm implication steps and consistency checks have to be performed. Backtracking will be necessary at least sometimes. Aggravating is the fact that sequential paths (storing modules) must be reckoned with on all branches of the signal paths (data and control).

The task to automatically generate program code for symbolic test instructions is equivalent to a retargetable (micro-) code generation. On the one hand there are a couple of simplifying limitations, on the other hand even stronger requirements occur. The set of symbolic statements is very small and there is no need to sqeeze the elementary operations into a sequence of minimal length or runtime. Higher demands are made on the flexibility of the code generation, exceeding simple transformations (e.g. based on commutativity). It must be possible to use paths crossing modules that alter the data transported. MSST accepts such modules on all paths, e.g. on the path from a data source to an input of a module to be tested (M_{ut}), a shifter with a "shift left logic" function can be passed if the signal expected at the output of the shifter represents an even number.

At the inputs of an altering module and at the data source then no longer the original testpattern p_0 specified for M_{ut} is expected. Instead a pattern p_n must be supplied such that:

① $f_1(f_2(\ldots f_n(p_n)\ldots) = p_0$,

where $f1, f_2, \ldots, fn$ are the functions of all modules $M_1, M_2, \ldots, M_n$ between M_{ut} and a data source. On the way to a comparator the output value of M_{ut} may be altered as well. However, for all mappings on this path an inverse must exist. If p_n now is the output value of M_{ut} and $p_0 = f_1(f_2(\ldots f_n(p_n)\ldots)$ is the pattern that results at the comparator then aditionally ② must hold:

② $f_n^{-1}(f_{n-1}^{-1}(\ldots f_1^{-1}(p_0)\ldots) = p_n$,

where $f_n^{-1}, f_{n-1}^{-1}, \ldots, f_1^{-1}$ are the inverse functions of the modules between M_{ut} and the comparator.

For modules that by no means have access to the specified input patterns, these patterns are altered automatically bit by bit until they fit into the set of accessible patterns. An input pattern that depends on the current state of the program counter is applied to modules on paths that only lead to the program counter and cannot be compared explicitly. This input pattern effects a jump to the address after the next instruction. The next instruction is an unconditional jump to the errorexit label. In this way the program control logic is being tested.

In order to carry out a machine status check at the end of the self-test program all changes of the status must be recorded. The machine status heavily depends on the currently generated program. Sequential detours not foreseeable on the symbolic level result in arbitrary changes. Obviously a test program synthesis independent from test generation will fail in many cases. The MSS2 retargetable microcode compiler[12] therefore could not be used. Experiences gained in this field turned out to be very helpful though, while implementing MSST.

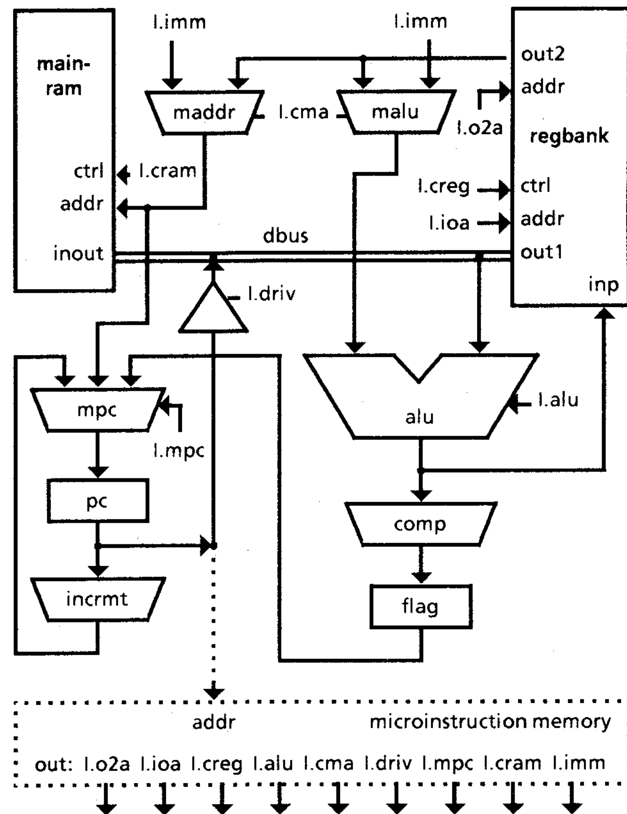

Fig. 4: RT-structure diagram for a simple example processor

4.Example

The strategy of MSST operation can be seen best on an example. Suppose test programs are to be generated for the simple processor system shown in Fig. 4. It consists of RAM, register-file, data bus, ALU, zero flag, some multiplexers, program counter, and incrementer. All is controlled by a 32 bit (horizontal) microinstruction, stored in an instruction memory not explicitly specified.

Inputs to MSST are an RT level structure specification (Fig. 5) written in MIMOLA and translated into MSS2 intermediate code by MSS2 front end software, and optionally a test pattern library (Fig. 2).

For default initialization (no patterns in the library) all memory words are assigned the value of their address. For a register in the register file this can be done by one instruction (Fig. 6). The MSST output shown there represents one MIMOLA program step (one instruction) headed by a Label (L...) and containing two statements to be executed in parallel (separated by a

```
TARGET processor;

STRUCTURE

TYPE word = (15:0); (* 16 bits wide *)

PARTS  (* %... denotes binary numbers *)

mainram : MODULE Sram
  PORT io
  (OUT out : word; IN in : word;
   ADR addr : word; FCT ctrl : (1:0));
  BEGIN
     CASE ctrl OF
     %00: out := TRISTATE;
     %01: out := TRISTATE, Sram(addr):=in;
     %10: out := Sram(addr);
     %11: out := Sram(addr), Sram(addr):=in;
     END
  END;

regbank : MODULE Sreg
  PORT io
  (OUT outl : word; IN inp : word;
   ADR addr : (2:0); FCT ctrl : (1:0));
  BEGIN
     CASE ctrl OF
     %00: outl := TRISTATE;
     %01: outl := TRISTATE, Sreg(addr):=inp;
     %10: outl := Sreg(addr);
     %11: outl := Sreg(addr), Sreg(addr):=inp;
     END
  END;

  PORT o2 (OUT out2 : word; ADR addr : (2:0));
  BEGIN
     out2:=Sreg(addr)
  END;

alu : MODULE Boperator
  (IN datal,data2:word; OUT res:word;
   FCT ctrl:(1:0));
  BEGIN
     res :=   CASE res OF
              %00 : datal;
              %01 : data2;
              %10 : datal "+" data2;
              %11 : datal "-" data2;
              END
  END;

maddr, malu : MODULE N2mux
  (IN dl,d2:word; OUT dat:word; FCT sel:(0));
  BEGIN
     dat :=   CASE sel OF
              0: dl;
              1: d2
              END
  END;

driv : MODULE Adriv
  (OUT dbus: word; IN dat: word; FCT sel: (0));
  BEGIN
     dbus :=  CASE sel OF
              0: TRISTATE;
              1: dat;
              END
  END;

pc : MODULE RP  (*program counter *)
  (OUT iaddr : word; IN dat : word);
  BEGIN
     RP := dat,
     iaddr := RP
  END;

comp : MODULE Acomp  (* zero test *)
  (OUT flag : (0); IN dat : word);
  BEGIN
     flag := "=0" dat
  END;

flag : MODULE Rff  (*cond. code flip-flop *)
  (OUT o : (0); IN i : (0));
  BEGIN
     Rff := i,
     o := Rff
  END;

inrcmt : MODULE Aop  (* incrementer *)
  (OUT res : word; IN dat : word);
  BEGIN
     res := "INCR" dat
  END;

mpc : MODULE N3mux
  (OUT iaddr:word; IN d0:(0);
   IN dl,d2:word; FCT sel:(1:0));
  BEGIN
     iaddr := CASE sel OF
              0: dl;
              1: d2;
              2: IF d0 THEN dl ELSE d2 FI;
              3: IF d0 THEN d2 ELSE dl FI;
              END
  END;

BUS dbus : word;

INSTRUCTION
  I: (o2a(31:29), ioa(28:26), creg(25:24),
      alu(23:22), cma(21), driv(20),
      mpc(19:18), cram(17:16), imm(15:0));

CONNECTIONS

regbank-o2 -> maddr.d2;         mpc    -> pc.dat;
regbank-o2 -> malu.dl;          flag   -> mpc.d0;
regbank-io -> dbus;             maddr  -> mpc.dl;
mainram-io -> dbus;             incrmt -> mpc.d2;
dbus    -> mainram-io.in;       I.mpc  -> mpc.sel;
maddr   -> mainram-io.addr;     driv   -> dbus;
I.cram  -> mainram-io.ctrl;     pc     -> driv.dat;
alu     -> regbank-io.inp;      I.driv -> driv.sel;
I.ioa   -> regbank-io.addr;     comp   -> flag.I;
I.creg  -> regbank-io.ctrl;     alu    -> comp.dat;
I.o2a   -> regbank-o2.addr;     I.cma  -> malu.sel;
malu    -> alu.datal;           I.imm  -> malu.d2;
dbus    -> alu.data2;           I.alu  -> alu.ctrl;
I.cma   -> maddr.sel;           I.imm  -> maddr.dl;
pc      -> incrmt.dat;

END.
```

Fig. 5 : MIMOLA specification of the example processor

```
L0020
(* I  = #1D280005
      = %XXX11101001010000000000000000000101 *)
  regbank-io(%101) := #0005,
  pc := "INCR" (pc);
```

Fig. 6: Default register initialization

comma). The corresponding hexadecimal and binary program code is shown as a commentary.

For a test lateron in the program a sequence of instructions (Fig. 7) is generated, verifying that the initial value is still kept in register 5. This is done for both output ports of the register file, starting with port "io".

```
L00D9
(* I  = #15A4FFFB
      = %XXX10101101001001111111111111011 *)
  flag := "=0"("+"(#FFFB, regbank-io(%101))),
  pc := "INCR" (pc);
L00DA
(* I  = #X3000000
      = %XXXXXX11XX0000000000000000000000 *)
  pc := IF flag  THEN "INCR" (pc)
                 ELSE #0000 FI;
L00DB
(* I  = #1D28FFFB
      = %XXX11101001010001111111111111011 *)
  regbank-io(%111) := #FFFB,
  pc := "INCR" (pc);
L00DC
(* I  = #BD84XXXX
      = %1011110110000100XXXXXXXXXXXXXXXX *)
  flag := "=0"("+"(  regbank-o2(%101),
                     regbank-io(%111))),
  pc := "INCR" (pc);
L00DD
(* I  = #X3000000
      =%XXXXXX11XX0000000000000000000000 *)
  pc := IF flag  THEN "INCR" (pc)
                 ELSE #0000 FI;
```

Fig. 7: Steps to test register 5 (both output ports)

The code generation algorithm proceeds as follows: According to the symbolic statement in Fig. 3 c) a path is traced backwards from the program counter to a conditional multiplexer. At the condition input a sequential path must be used. The zero flag has to be loaded in an immediately preceding instruction. This instruction is inserted now. From "flag" the path is followed to the test-on-zero operator and further on to the "alu". The "+" operation is found suited to realize a comparison. From the right input of the "alu" the output port to be tested is reached via the data bus (non-altering modules and busses not shown in Fig. 7). For a zero result the two's complement of 5 is tracked down from the left "alu" input. The source for this value is the immediate data field of the instruction word. An increment statement for the program counter then is generated and work continues with the conditional assignment in the (now) second instruction (L00DA). When the sources for the incremented value of the program counter and the errorexit jump address have been found, then this test step is completed.

In order to test output port "o2" of the register file three instructions are necessary. The add-up value (-5 = #FFFB) can only be found on the sequential path through port "io" in register 7 (%111).

A desired value at the output of a module is traced back to the inputs and on to a final source (memory, hardwired constant, instruction field or external interface). Required module control signals are traced back first, then address inputs and finally data inputs.

A step to check the function "-" of the combinational module "alu" (with default input patterns) is shown in Fig. 8. The right "alu" input pattern (#0000) is accessed sequentially in register 6. As the output cannot be compared directly, it is stored in register 6. The following comparison and conditional jump is equivalent to a register test step.

```
L017C
(* I  = #19280000
      = %XXX11001001010000000000000000000 *)
  regbank-io(%110) := #0000,
  pc := "INCR" (pc);
L017D
(* I  = #19EC5555
      = %XXX11001111011000101010101010101 *)
  regbank-io(%110) :=
      "-.alu" (#5555,regbank-io(%110)),
  pc := "INCR" (pc);
L017E
(* I  = #19A4AAAB
      = %XXX11001101001001010101010101011 *)
  flag := "=0"("+"(#AAAB, regbank-io(%110))),
  pc := "INCR" (pc);
L017F
(* I  = #X3000000
      = %XXXXXX11XX0000000000000000000000 *)
  pc := IF flag   THEN "INCR" (pc)
                  ELSE #000
```

Fig. 8: One step to test a combinational module

5. Applications

Since and during the development of MSST it has continuously been tried on a variety of digital processors. Apart from simple examples like the one shown here self-test programs have been generated for real-life hardware and commercially used systems, e.g :

- A high speed special purpose processor built up in bipolar bit-slice technology (AMD) and used for high resolution picture processing equipment. Due to very long data and control paths (up to 15 RT modules in a row) and a high degree of encoding the generation of microprograms was most time consuming for this example (up to 3 cpu hours on a Siemens 7.760 mainframe).
- A telephone processor in one-chip CMOS design (Siemens). The architecture and instruction set of this system resembles very much to common microprocessors. Test generation could be done nearly

without problems. A default test (1200 machine instructions) took about 800 cpu seconds.

- A prototype of an innovative architecture (data driven timing) general purpose processor, designed for high throughput by means of massive parallelism on the microprogramming level. The prototype has been built at the Kiel University[13] using standard TTL logic chips. On this example MSST test programs for the first time were brought into action. For different configurations of the system self-tests were generated and ran even on the still incomplete (only one of two ALUs, two of four memory ports etc.) hardware. Some component failures and a couple of wiring faults could be detected and tracked down to the corresponding RT-module with the MSST default test. Errors that could not be detected did not occur so far. A manual generation of test programs for this processor (142 bits horizontal microinstruction) would be quite unreasonable. Probably a lot more errors would be in the programs than in the hardware. MSST generated about 3000 fault free microinstructions within less than 20 minutes (cpu time).

6. Conclusion

An approach to the testing of digital systems especially suited for microprogrammable and special purpose custom designed processors has been presented. MSST is a tool for the test and design engineer that works on the register transfer level and utilizes the natural partition of the system into RT modules. It automatically generates self-test program code that locally applies user provided module input patterns to test for faults in the interior of system internal RT modules. As with all tools the quality of the product still depends on the skills of the user. A test with default patterns makes shure that at least stuck-at faults and shorts between adjacent bit-lines on the module interconnections are detected. All features of MSST are fully implemented and first applications have shown promising results. Experiments to compare the fault coverage achievable with that of other testing methods are under way.

References

[1] S.M.Thatte, J.A.Abraham: "Test Generation for Microprocessors", IEEE Transactions on Computers, Vol.C-29, No.6, June 1980, pp.429-441

[2] A.Hunger: "Neues Verfahren zum Selbsttest von Mikroprozessoren", Verlag TÜV Rheinland, Köln 1982

[3] C.Bellon, A.Liothin, S.Sadier, G.Saucier, F.Grillot, R.Velasco, M.Issenman: "Automatic Generation of Microprocessor Test Programs", 19th Design Automation Conference, 1982, pp. 566-573

[4] A.K.Susskind: "Overview of Microprocessor Testing", IEEE Int. Conference on Computer Design (ICCD) 1983, pp.45-48

[5] K.W.Lai: "Functional Testing of Digital Systems", PhD Diss. CMU-CS-81-148, Dec. 1981, Pittsburgh PA

[6] G.Zimmermann: "The MIMOLA Design System : A Computer Aided Prozessor Design Method", 16th Design Automation Conference, 1979, pp. 53-58

[7] P.Marwedel: "The MIMOLA Design System: Tools for the Design of Digital Processors", 21st Design Automation Conference, 1984, pp.587-593

[8] R.Jöhnk, G.Krüger, P.Marwedel: "MIMOLA Software System 2 User Guide", online documentation 1985

[9] A.C.Parker: "Automated Synthesis of Digital Systems", IEEE Design&Test of Computers, Vol.1, No.4, 1984, pp.75-81

[10] D.Brahme, J.A.Abraham: "Functional Testing of Microprocessors", IEEE Transactions on Computers, Vol.C-33, No.6, June 1984, pp.475-485

[11] C.V.Ramamoorthy, L.C.Chang: "System Modeling and Testing Procedures for Microdiagnostics", IEEE Transactions on Computers, Vol.C-21, No.11, Nov. 1972, pp.1169-1183

[12] P.Marwedel : "A Retargetable Compiler for a High-Level Microprogramming Language", 17th Annual Microprogramming Workshop (MICRO 17), 1984, pp.267-276

[13] L.Nowak :"Entwurf und Realisierung eines neuartigen Rechnerkonzeptes", PhD Diss., Institut für Informatik und Praktische Mathematik, Universität Kiel 1986