# Architectural Timing Verification and Test for Super Scalar Processors

Pradip Bose

IBM Corporation, Risc System/6000 Division, Zip 4441
11400 Burnet Road, Austin, TX 78758

## Abstract

*We address the problem of verification and testing of super scalar processors, from the point of view of correctness of program execution time. Trace–driven architectural simulation methods are commonly used in current industrial practice to estimate cycles–per–instruction performance of a candidate processor organization, prior to actual implementation. We present a novel set of strategies for testing the timing correctness of processors as represented in an architectural timing model ("timer"). We focus on two main aspects of the theory: (a) deriving architectural test sequences to cover possible failure modes, defined in the context of a pipeline flow state transition fault model; (b) deriving loop test kernels to verify steady–state (periodic) behavior of pipeline flow, against analytically predicted signatures. We develop the theory in the context of an example super scalar processor and its timer model.*

## I. Introduction:

Functional testing and verification of processor architectures is a critical aspect of the overall verification methodology used in advanced (VLSI) CPU design. In current industrial practice, (e.g., [1]), random architectural test program generation and simulation methods are commonly used to "verify" functional correctness of implemented instructions. The investment in manpower and computer time required to complete such architectural verification prior to (and in between) chip tape–outs, is very high and is constantly on the rise. Also, there is a lack of adequate formalism and theory, in terms of failure models and coverage criteria; thus, the fact that a given processor simulation model has "passed" millions of Architecture Verification Programs (AVPs) (see [1], section IV) certainly boosts confidence in the implemented correctness, but what percentage of the global implementation space has been exercised, is not known. Classical fault–model based functional test generation methods (e.g. [2,3]) are occasionally used in essence, if not in full, to "bias" the architectural test sequence generation towards a few typical failure modes. Formal verification techniques (e.g. [4,5]) seem to show promise, but in many cases are yet to cross the "toy examples" barrier and be used in real–world VLSI CPU chips.

With the current growth in super scalar and super pipelined processors (RISC or CISC), the need for pre–hardware architectural *timing verification* is also of utmost importance. Thus, for example, it is not enough to verify that the mapped (equivalent) test case for the assignment statement: A = B + C "works" correctly in the functional sense; we must also be able to verify the correctness of the exact cycle count for its execution. In initial generations of microprocessor design, such timing or performance verification requirements were minimal because of the lack of concurrent (pipelined) structures. In modern VLSI processors, due to multiple dispatch modes and concurrent pipeline execution with out–of–order execution modes, the "science" of deriving Performance Verification Programs (PVPs) in addition to the traditional AVPs is becoming increasingly important. In this paper, we present our approach to PVP generation as used in our current super scalar design and verification.

In prior work [6,7,9], we briefly described alternate methods used in current industrial practice, for evaluating cycles–per–instruction (CPI) performance for super scalar machine models. A workload or benchmark–driven **timer,** is a cycle–by–cycle (timing) simulator of a candidate processor organization, with a program workload as the driving input. If this input is a *dynamic* execution trace, we refer to the tool as a *dynamic* timer. On the other hand, *static* timers evaluate program execution time by analyzing a static program listing (high–level, intermediate or assembly/machine code). In either case, timers do not carry out actual functional simulation of the workload: only the cycle–by–cycle timing behavior of the concurrent pipeline structures is simulated. A **functional simulator,** on the other hand, is a sequential (i.e., one instruction at a time) simulation of an idealized von–Neumann machine implementing the candidate (instruction set) architecture. One of the outputs of such a functional simulator, is in fact a dynamic trace which can be used to drive a dynamic timer. The parameters which can be set and changed easily for such a timer, are usually limited to processor organizational parameters, e.g. queue/buffer lengths, pipeline depths, cache latency parameters, branch prediction–related controls, bus/dispatch bandwidths, various context–sensitive

0363-8928/94 $3.00 © 1994 IEEE

processing switches (on/off or boolean flag parameters), etc. In some recent, truly programmable timers, (e.g. the BRAT timer in [10]), the instruction set architecture itself is made available to the user as a global parameter; and, a whole range of similar processor chips can be modeled by "programming" a range of such parameters. The effect of alternate compiler optimization strategies on processor performance, can be studied using such dynamic timers, only by regenerating the trace for alternatively compiled modules. In the case of a static timer, on the other hand, the compiler optimization parameters can (at least in part) be made a subset of the timer parameters, for the chip designer to experiment with [1]. However, even in infinite cache mode, such static estimators tend to have less accuracy than trace–driven dynamic timers, but are usually much faster [6,7].

In this paper, we restrict ourselves to dynamic timers only; from this point onward, we omit the "dynamic" qualifier when referring to timers. In the early or intermediate stages of processor design, timers are useful for making design trade–offs and parameter sizing. Later, when the machine and timer models have stabilized, accurate pre–hardware projections are made using benchmark–driven timer runs. A crucial problem in this context, is that of testing or validating the timer model against a "gold" processor model.

The sources of inaccuracies in a timer model are: (1) modeling errors due to programming mistakes; (2) errors due to misinterpretation of informally specified execution semantics; (3) data–sensitive execution semantics which are usually considered to be beyond the range of modeling capability of timers. An example of the third category of inaccuracies, is a case where alternate paths within a staged pipeline data path are followed, depending on the value range of the operand (register or memory). Such instances, are usually infrequent enough, that they may be ignored in analyzing performance behavior for realistic program benchmarks. We therefore limit our attention, in this paper to (1) and (2) above. Since dynamic traces can be millions (and possibly billions) of instructions long, it is impractical for a designer to go over the entire cycle–by–cycle listing to identify such defects. A robust test/verification methodology, based on a tailored, test case suite, generated from higher–level application kernels (loops), is described in this paper. We present this method in terms of an example RISC super scalar processor, patterned after the RS/6000, but with an added organizational feature: the instruction completion buffer mechanism. We refer to this machine as the ERISC (extended RISC) machine. The machine organization for ERISC was defined specifically for the purposes of this paper; it is not based on an actual product. The trace–driven timer for ERISC was derived by extending the TRISC timer [8] used earlier [6,7] to study the accuracy of static timers.

## II. The ERISC machine and its timer:

The example RISC machine (ERISC) used for the purposes of this paper has a typical super scalar organization (Figure 1), implementing the POWER architecture [12]. All functional ops (FIX or FLT) are register–to–register, with two sources and one destination, explicitly specified. The instruction dispatch buffer can hold up to d instructions, which is a timer parameter. Every cycle, up to three instructions can be dispatched, one to each of the functional units: BRN, FIX and FLT. Floating point load and store instructions are processed by FIX for address generation, prior to cache request. The instruction ids for such instructions are dispatched to FLT as well, to aid synchronization. Actual synchronization is effected via register renaming. Both FIX and FLT (source and destination) registers are subject to dynamic renaming. FIX and FLT each has its own set of physical (rename) buffers, the sizes of which are timer parameters. The architected register file is updated during actual instruction completion. Instruction execution can be out–of– order, but instruction dispatch and actual completion is in–order. The in–order dispatch and completion mechanism is managed by using a completion buffer or queue, maintained and controlled by the branch–and–condition unit, BRN. This mechanism is similar to the completion buffer scheme described for the PowerPC 603 machine [10]; the in–order completion mechanism facilitates implementation of precise interrupts, with out–of–order execution modes present in the overall processor.

The organization parameters mainly considered in this paper are:

1. Effective number of (1–cycle) pipeline stages, $p_1$ and $p_2$ respectively, in the FIX and FLT units. (The BRN unit has, effectively, a 1–stage pipe).

2. The sizes of the instruction queues q_brn, q_fix and q_flt, associated with the three functional units.

3. The cache access latency, c , in machine cycles. (Infinite cache model is assumed).

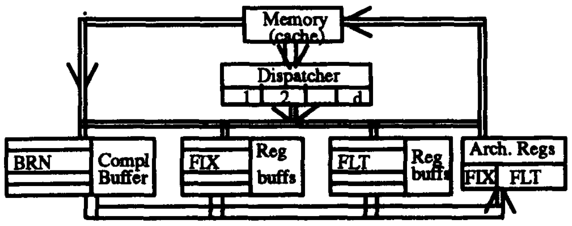

Figure 1. ERISC processor organization.

4. The dependent bubble parameters, b1 and b2, respectively, where , b1 (b2) is the number of pipeline bubbles in cycles caused by a consecutive dispatch sequence of two *dependent* fixed (float) instructions, where the second instruction is data dependent – via register interlocks – on the first one.

5. The number of FIX and FLT rename buffer registers, R_fix and R_flt.

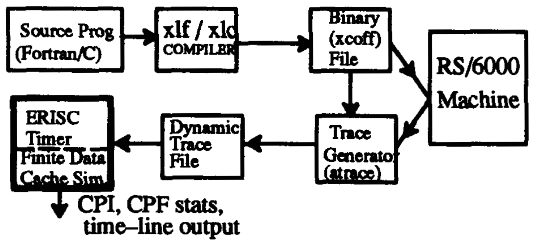

Figure 2. ERISC timer tools complex

<u>The ERISC timer:</u>

As stated in Section I, a **timer** is a cycle–by–cycle simulator of a candidate machine organization. Its main purpose is to print out an overall cycles–per–instruction (CPI) performance figure for a given instruction trace. As a side benefit, the detailed timer outputs are useful in identifying compiler deficiencies and organizational bottlenecks. In pre–hardware evaluations, dynamic instruction traces are usually generated by a separate **instruction set simulator.** Figure 2 shows the software organization of the tools used to drive the ERISC timer. Since the instruction set architecture assumed is that of an RS/6000 [11,12], we are able to use available compilers [13] and trace generators [14] for generating traces using an existing hardware platform. The actual ERISC timer program is written in Pascal (with an alternate implementation in C) and runs on an RS/6000. For the purposes of this study, we have used an **infinite cache** timer model, in which memory reference paths are pipelined, resulting *always* in cache hits, with a fixed latency of c cycles.

In analogy with hardware (architectural) testing of processor chips (e.g., [2]), we formulate a functional performance fault model to diagnose the model defects referred to in section I.

## III. Functional performance fault model:

We define the fault model from two different aspects: (a) functional unit level and (b) instruction level. We later assert that the two are essentially equivalent for our purpose.

**(a): Unit–Level Fault Model:**

<u>Instruction Fetch/Dispatch Unit:</u>

The normal ("gold", or fault–free) model of the instruction–issue logic (IIL) may be stated formally as follows:

```
Procedure IIL_Gold:
begin
  i:= 1;
  num_disp:= 0;
  while (i <=3) and (num_disp <=3) do begin
    case inst_buf(i).typ of
      fix: if (not fix_ren_buf_full) and (not compl_buf_full)
        and (not fix_busy) then begin
              issue_to_fix(i);
              num_disp:= num_disp+1;
              i:= i+1;
        end
        else return; (* no more dispatches *)
      flt: if (not flt_ren_buf_full) and (not compl_buf_full)
        and (not flt_busy) then begin
              issue_to_flt(i);
              num_disp:= num_disp+1;
              i:= i+1;
        end
        else return; (* no more dispatches *)
    brn: if (not compl_buf_full) and (not brn_busy)
          then begin
              issue_to_brn(i);
              num_disp:= num_disp+1;
              i:= i+1;
        end;
    end; (* case *)
  if (not instr_buf_full) then fetch_new_instrs;
end; (* end, procedure IIL_Gold *)
```

**Execution sub–processes:**

*Dispatch decoder:*

The instruction (pre)–decode process (manifested by the CASE statement, above), which classifies instruction ops, according to the unit to which they must be dispatched.

*Dispatch inhibitors:*

The inhibitors to the process of instruction dispatch to individual functional units, as evident from the logic above, are:

- FIX: The logical predicates: (a) fix_ren_buf_full, whose truth implies that all available rename (backup) buffers in FIX are taken; (b) compl_buf_full, which flags the condition when the dispatch process has run out of issuable instruction id's; (c) fix_busy, which is true when the FIX decode stage is busy and the associated unit queues (if present) are full.
- FLT: The logical predicates: (a) flt_ren_buf_full, analogous to fix_ren_buf_full (above); (b) compl_buf_full, as in FIX; (c) flt_busy, analogous to fix_busy (above).
- BRN: The logical predicates: (a) compl_buf_full, as above; (b) brn_busy, analogous to fix_busy or flt_busy (above).

*Dispatch executors:*

The actual dispatch sub-processes which result in transfer of instructions from the dispatch unit to the individual functional units, are: (a) issue_to_fix; (b) issue_to_flt; and (c) issue_to_brn;

*Dispatch sequencers:*

The "counting" logic: (a) increment the index variable i, the process which ensures that the same instruction is not multiply dispatched, and that the very *next* instruction in sequence is dispatched; (b) increment the count of the number of instructions, num_disp, dispatched in the current cycle: fault-free counting here is necessary to ensure that the instruction buffer filling procedure, fetch_new_instrs brings in the correct number of instructions and "stitches" them into the instruction buffer at the correct point.

The following *functional* defect model is assumed for the IIL timer module, based on the notion of faulty *decoding, inhibitor-control, execution* and *sequencing*. One or more of the following faults, detectable as architectural timing errors, can be present.

1. In spite of the absence of all inhibitors, one or more of the dispatch execution mechanisms are blocked.
2. In the presence of one or more dispatch inhibitors, an illegal dispatch execution path is enabled.
3. An instruction is decoded (classified) erroneously and dispatched to the wrong functional unit.
4. Due to faulty sequencing, an instruction is dispatched in successive cycles to the same functional unit.
5. The head-pointer of the instruction buffer is updated (incremented) erroneously, after completion of dispatch.

The FIX or FLT Execution Pipe:

The normal ("gold", or fault-free) model of the FIX(FLT) functional execution pipe (FEP) can be stated formally as follows:

```
Procedure FEP_Gold:
begin
  if  pipe_last_stage.busy and putaway_bus_avail then begin
      putaway_bus:= pipe_last_stage.instr;
      pipe_last_stage.instr:= null;
      pipe_last_stage.busy:= false;
      free_reg_interlocks;
      send_finish_report_to_BRN;  (* the "finish" bit in the completion
      buffer in BRN, for this instruction, gets set *)
  end;
  if  (not pipe_last_stage.busy) then
     for i:= last_stage downto 2 do pipe_stage[i]:= pipe_stage[i-1];
       (* the above iteration implements the single-cycle advance of
          instructions in the linear pipe stages *)
  if  pipe_stage(1).instr <> null then begin
     if (not pipe_stage(2).busy) and (not reg_interlock) then begin
        set_register_interlocks;
        pipe_stage(2):= pipe_stage(1);
        pipe_stage(1).instr:= null;
        pipe_stage(1).busy:= false;
     end;
  end;
  if (not FEP_queue_empty) then cycle_FEP_queue;
  (* the above step forwards an instruction from the queue,
     if present, to the decode stage, pipe_stage(1) of the FEP *)
end; (* procedure FEP_Gold *)
```

In a manner similar to the IIL_Gold case, we can define the **Decode, Inhibit, Execute and Sequence** sub-processes for the FEP_Gold process. We omit the detailed definition and proceed directly to the assumed functional defect model for this case. One or more of the following faults may be present:

1. Under unblocked pipe condition, an instruction gets held in a given stage of the pipeline for one or more cycles longer than it should; — a pipe sequencing fault.
2. Under blocked mode, (e.g. dependence-based register interlocking), an instruction advances illegally into the next stage; — a pipe inhibitor control fault.
3. Under unblocked mode, an instruction advances along the pipeline stages by an incorrect (larger) amount; — a pipe sequencing fault.
4. One or more register interlock bits are set or released incorrectly, or at the wrong stage; — a pipe execution fault.
5. The "finish" signal transmission to BRN does not happen, or happens in the wrong cycle; — a pipe execution fault.
6. One or more registers are decoded erroneously in the decode stage; — a pipe decode fault.

The BRN Unit: The instruction completion logic (ICL) is the most crucial aspect of the BRN timing functionality. This mechanism, as stated earlier, keeps track of live, in-order-dispatched instructions, and ensures in-order completion of instructions. Thus, instruction id's are issued to the completion buffer tail on dispatch and taken off from the head of the queue, in order, on completion. Up to three "finished" instructions may complete per cycle, to match the maximum instruction issue rate of three per cycle. Instruction "finish" bits, maintained in the completion buffer, are set by individual functional units, on instruction execution "finish". This mechanism allows out-of-order execution within the window of live instructions, while ensuring the convenience of in–order "completion", which facilitates implementation of precise interrupts.

The ICL_Gold formalism is pretty straightforward, as apparent from the description above. We omit the specification and fault model definition. Completion buffer directed testing concepts are directly applicable to the construction of "self–checking timers", a topic which is being separately published by this author.

**(b) Instruction–Level Fault Model:**

In an alternate mode of fault model definition, we may model defects in terms of illegal pipeline paths adopted by the instruction flow within the pipeline timer model, and defective dependence behavior between pairs of instructions. One or more of the following defects may occur, causing an error in the overall execution timing:

1. A FIX–type instruction gets dispatched erroneously to the FLT unit.
2. A FLT–type instruction gets dispatched erroneously to the FIX unit.
3. A floating point load/store instruction ignores the interlock on a fixed point register on which it depends for address generation.
4. A fixed point load/store instruction ignores the interlock on another fixed point register on which it depends for address generation.
5. A floating point arithmetic operation ignores the interlock on a source or target operand register.
6. A fixed point arithmetic operation ignores the interlock on a source or target operand register.

**Lemma:**

In terms of detectable timing errors in pipeline flow, the unit–level fault model and the instruction–level fault model, as stated above, are *equivalent*. That is, designing test sequences to cover defects under one fault model, automatically covers faults in the other model and vice–versa.

The proof is omitted here for brevity. In this paper, defects in branch processing are not discussed. Note that even for straight–line (branchless) execution, the functional fault models can be made much more complex. The above simple model, however, has been found to be more than adequate for deriving a comprehensive test case suite, which is able to cover a large class of other defects as well. (This is analogous to the case of multiple fault coverage or unmodeled fault coverage, of test sets derived on the basis of single stuck–at faults, for digital logic).

*Assumption*: It is assumed that fault masking is not present; i.e., presence of multiple defects at the same time is not ruled out, but the effect of a given defect is still observable as if that defect existed alone.

This assumption turns out to be a very reasonable one, from the point of view of experienced timer writers. It also simplifies the task of test case generation in the same way as a single stuck–at fault model reduces the complexity of test generation for digital logic.

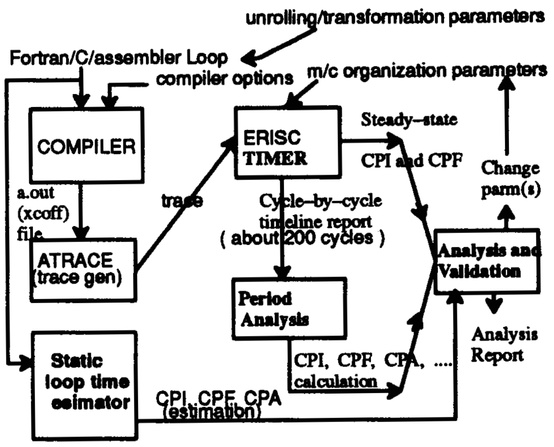
unrolling/transformation parameters
Fortran/C/assembler Loop
compiler options
m/c organization parameters
COMPILER
ERISC TIMER
Steady–state CPI and CPF
Change parm(s)
a.out (xcoff) file
trace
Cycle–by–cycle timeline report ( about 200 cycles )
ATRACE (trace gen)
Analysis and Validation
Period Analysis
Analysis Report
Static loop time esimator
CPI, CPF, CPA, .... calculation
CPI, CPF, CPA (estimation)

Figure 3. Architectural timing validation methodology.

## IV. Loop trace–driven timing validation:

The overall architectural timing validation methodology, using source/assembler test loops, is illustrated in Figure 3. The test cases are (usually) in the form of high–level Fortran (or C) kernels (loops) specifically designed to test for modeled defects. The loops are compiled using an existing compiler and the object code is traced using an existing tracer (see also Fig. 2). The dynamic trace can then be used to drive the ERISC timer; or, the static code can be used as input to a static timer (estimator) program. The cycle by cycle behavior and the cycles–per–instruction (CPI) produced by the ERISC timer are matched against analytically predicted behavior. (For the deterministic, infinite cache model under consideration, a formula–based method for static evaluation of CPI is available from prior research [6,7,9] and is used in the static estimator code).

The loop kernel is traced over a large number of iterations, to be able to drive the timer to a steady–state pipeline pattern. The "Period Analysis" procedure is used to analyze the steady–state periodicity of the various pipelined units (including queues) in order to relate them to the timer–printed statistics and the static estimator–predicted numbers.

## Statistics for timing analysis and verification:

The following metrics, evaluated via alternate paths, are used in the architectural timing verification and test methodology:

### Definitions:

**L(n):** The length, in number of instructions, of a loop trace, obtained by executing the source loop over n iterations.

**T(n):** The execution time, in cycles, of a given loop test kernel, traced for n iterations.

**CPI:** The average number of executed cycles per instruction; it is obtained by dividing the total number of cycles, T, taken to execute a given program (trace) on the timer model by the total number of instructions, L, in the trace.

**$P_{ss}$:** The steady–state period, in cycles, of the timer pipeline state–transition pattern.

**$N_{ss}$:** The cycle count which marks the onset of steady–state cycle–by–cycle timer output pattern, for a given trace.

**$CPI_{ss}$:** The steady–state CPI for a given loop trace:

$$CPI_{ss} = \lim_{n \to \infty} [T(n)/L(n)]$$

**CPF:** The average number of executed cycles per floating point operation. CPF is related to MFLOPS performance of the processor.

**$CPF_{ss}$:** The steady–state CPF for a given loop trace (containing floating ops):

$$CPF_{ss} = \lim_{n \to \infty} [T(n)/(n*F)]$$

where F is the number of flops per iteration.

**CPX:** The average number of executed cycles per fixed point operation; it is obtained by dividing the total number of executed cycles by the total number of fixed point operations in the trace. (Floating point loads and stores count as fixed point operations in such super scalar machines).

**CPB:** The average number of executed cycles per encountered branch operation.

**CPA:** The average number of processor cycles per memory access (load/store). This metric measures the processor–memory (cache) traffic for a given workload.

### Example test case and timer output:

We now present a specific loop test case, with timer output, to illustrate the concepts and definitions introduced above.

#### The daxpy test case:

"Daxpy" is the key loop within the well known floating point benchmark of Linpack. The Fortran specification of daxpy is:

```
do i = 1,n
  x(i) = x(i) + s*y(i)
enddo
```

where the 1–dimensional arrays x, y and the scalar s are declared to be double precision floating point variables. The corresponding compiled code, in mnemonic notation, per iteration, is:

```
A:    lfd    fr1, r6, 0x8
B:    fma    fr1, fr0, fr2, fr1
C:    lfdu   fr2, r5, 0x8
D:    stfdu  fr1, r6, 0x8
E:    bc
```

The alphabetical labels A, B, C, ... are assigned to successive instructions in the execution trace in alpahabetical order, with Z being succeeded by a, b, c, ...., and z wrapping around again to A. Thus, in the actual trace, F stands again for the first load instruction (lfd), G for the fma, etc. The ERISC cycle–by–cycle timer output for the first 40 cycles, is shown in Figure 4. The labeled functional units or queues, with dashes (–) representing individual stages, are explained below:

PIB: primary instruction buffer: nominally set to a size of 12 for this run.
IST: the instruction sequencing table or completion buffer (queue), nominally set to 16 for this run.
LSTQ: the load–store instruction queue, for holding load and store instructions for the fixed point unit FIX.
FLT_Q: the floating point instruction queue for the floating point unit, FLT.
CA: the on–chip, level–1 cache access pipe, nominally set to 1 stage for this run.
STQ: the pending store queue, which holds stores waiting for data, prior to writing in cache.

With reference to Figure 4, in cycle 1, the first three instructions in the buffer, (A, B, C, representing the lfd, fma and lfdu) are dispatched, with the corresponding instruction id's allotted to the completion buffer slots. A goes to the first stage ("decode") of FIX, while C gets queued in LSTQ, and B goes to the first stage ("decode") of FLT. Instruction B (fma) has to wait in "decode" for 4 cycles because it has to wait for one of its operands to be produced by A (lfd). It then advances along the FLT execution pipe (set to 4 stages for this run). The lfd A moves from "decode" to "address gen" to "request to cache" to actual cache access stage (CA). Following this, A gets "finished", followed by "completion", which is manifested by its disappearance from the completion buffer (IST) in cycle 5. As mentioned earlier, all instructions get dispatched in order, as evidenced by the appearance of instruction id's in the completion buffer; also, the instructions get completed in–order, as evidenced by the deletion of id's from the head of the completion buffer. By monitoring

```
Cycle           PIB    IST              LSTQ  FIX   FPU_IQ    FLT  CA    STQ

  0    -------EDCBA ---------------- ---- ----- -------- ----- - --------
  1    ----------ED ABC------------- C--- A---- -------- B---- - --------
  2    -------JIHGF ABCDE----------- D--- CA--- -------- B---- - --------
  3    ----------JI ABCDEFGH-------- FH-- DC--A G------- B---- - --------
  4    -------ONMLK ABCDEFGHIJ------ HI-- FD--C G------- B---- A --------
  5    ----------ON -BCDEFGHIJKLM--- IKM- HF--D L------- GB--- C --------
  6    -------TSRQP -BCDEFGHIJKLMNO- KMN- IH--F L------- G-B-- D --------
  7    ---------TSR QBCDEFGHIJKLMNOP MNP- KI--H LQ------ G--B- F D-------
  8    ---------TSR QBCDEFGHIJKLMNOP NP-- MK--I Q------- LG--B H D-------
  9    -----------T QRSDEFGHIJKLMNOP PRS- NM--K Q------- L-G-- I D-------
 10    -----------T QRSDEFGHIJKLMNOP RS-- PN--M Q------- L--G- D I-------
 11    -------YXWVU QRST--GHIJKLMNOP S--- RP--N Q------- L---G K I-------
 12    ----------YX QRSTUVW-IJKLMNOP UW-- SR--P V------- QL--- M IN------
 13    -----------Y QRSTUVWXIJKLMNOP WX-- US--R V------- Q-L-- I N-------
 14    -------dcbaZ QRSTUVWXY--LMNOP X--- WU--S V------- Q--L- P N-------
 15    ---------dcb QRSTUVWXYZaLMNOP Z--- XW--U a------- VQ--L R NS------
 16    -----------d QRSTUVWXYZabcNOP bc-- ZX--W a------- V-Q-- U NS------
 17    -----------d QRSTUVWXYZabcNOP c--- bZ--X -------- aV-Q- N S-------
 18    -------ihgfe QRSTUVWXYZabcd-- ---- cb--Z -------- a-V-Q W SX------
 19    ----------ih g-STUVWXYZabcdef g--- ec--b f------- a--V- Z SX------
 20    -----------i ghSTUVWXYZabcdef h--- ge--c -------- fa--V S X-------
 21    -------nmlkj ghi--VWXYZabcdef ---- hg--e -------- f-a-- b Xc------
 22    ----------nm ghijkl-XYZabcdef l--- jh--g k------- f--a- e Xc------
 23    -----------n ghijklmXYZabcdef m--- lj--h -------- kf--a X c-------
 24    -------srqpo ghijklmn--abcdef ---- ml--j -------- k-f-- g ch------
 25    ----------sr ghijklmnopq-cdef q--- om--l p------- k--f- j ch------
 26    -----------s ghijklmnopqrcdef r--- qo--m -------- pk--f c h-------
 27    -------xwvut ghijklmnopqrs--f ---- rq--o -------- p-k-- l hm------
 28    ----------xw -hijklmnopqrstuv v--- tr--q u------- p--k- o hm------
 29    -----------x whijklmnopqrstuv w--- vt--r -------- up--k h m-------
 30    -------CBAzy wx--klmnopqrstuv ---- wv--t -------- u-p-- q mr------
 31    ----------CB wxyzA-mnopqrstuv A--- yw--v z------- u--p- t mr------
 32    -----------C wxyzABmnopqrstuv B--- Ay--w -------- zu--p m r-------
 33    -------HGFED wxyzABC--pqrstuv ---- BA--y -------- z-u-- v rw------
 34    ----------HG wxyzABCDEF-rstuv F--- DB--A E------- z--u- y rw------
 35    -----------H wxyzABCDEFGrstuv G--- FD--B -------- Ez--u r w-------
 36    -------MLKJI wxyzABCDEFGH--uv ---- GF--D -------- E-z-- A wB------
 37    ----------ML wxyzABCDEFGHIJK- K--- IG--F J------- E--z- D wB------
 38    -----------M wxyzABCDEFGHIJKL L--- KI--G J------- E---z w B-------
 39    -------RQPON M--zABCDEFGHIJKL ---- LK--I -------- JE--- F BG------
 40    ----------RQ MNOP--CDEFGHIJKL P--- NL--K -------- OJE-- I BG------
```

Figure 4. ERISC timer cycle–by–cycle output for daxpy test case

the last stage of the FLT execution pipe, we can see that the pipe reaches steady state in about 20 cycles, beyond which one fma is produced every three cycles, implying, $CPF_{ss} = 3/2 = 1.5$, since an fma counts as two flops.

It is to be noted that each unit or queue individually attains a steady–state pattern, with the same fundamental period of 3 cycles. Thus, beyond cycle $N_{ss} = 20$, the PIB exhibits a recurring pattern of 5–2–1, i.e. 5 instructions, followed by 2, followed by 1. This phenomenon of attainment of a uniform steady–state pattern, across the modeled pipelined units, when driven by an iterative loop trace, is the characteristic signature of a *level–0* validated timer (see definition, in the following sub–section).

**Levels of timer model validation:**

We define three distinct levels of timer model validation, as follows:

Definitions:

**Level–0:** In this level of validation, all test loop traces applied result in a uniform, steady–state timer output pattern of *finite periodicity*, $P_{ss}$, attained within a *finite number of cycles*, $N_{ss}$.

If the observed steady–state period is infinite, i.e. if the timer output pattern stabilizes to the exact same overall pipeline state for any cycle count greater than $N_{ss}$, then clearly the timer is in an illegal, deadlock state, and in such a case, the model is said to *fail* level–0 validation for the particular test loop trace.

The timer model is said to be weakly level–0 validated, if it passes level–0 validation in the above defined sense, across the test loop trace suite, for the specified design point characterized by an exact setting of organizational parameters. The model is said to be strongly level–0 validated if it passes level–0 validation for all legal (allowed or defined) combinations of organizational parameters.

**Level–1**: In this level of validation, for each test loop trace, the performance metrics (CPI, etc.) are observed to vary *monotonically* as a function of any given organizational parameter size, (within its defined limits). (Only one parameter is varied at a time).

**Level–2**: In this level of validation, each test loop trace is verified to produce a steady–state performance (typically $CPI_{ss}$ or $CPF_{ss}$) which is in agreement with deterministic (infinite cache) static prediction formulae (see next sub–section).

We state the following lemma without proof. The proof is based on the theory behind the actual test case generation methodology. Due to lack of space, we only provide a few examples of generated test cases, (see Section V), without discussing the actual generation procedures; these are being made available in a more detailed report.

**Lemma:** Passing Level–1 *and* Level–2 validation for the generated loop trace test case suite, guarantees *strong* Level–0 validation (but not vice–versa), under the infinite cache execution semantics assumed for ERISC.

In addition to the above, in our verification methodology, we define a level–3 validation step, where the cycle–by–cycle timer outputs are validated against the full–scale logic simulation model of the processor–under–design. This level of validation is beyond the scope of this paper, and is therefore not discussed further.

**Examples of level–1 validation:**

An example of level–1 validation is the variation of CPI performance with variation of the cache access latency parameter, c. In the absence of speculative fetches and branch prediction, infinite cache $CPI_{ss}$ for a given test loop trace is expected to increase *linearly* as a function of c. Deviation from linearity indicates presence of one or more modeled timing faults.

Another example is the variation of $CPI_{ss}$ with a specific hardware resource size, e.g. the size of the completion buffer, IST or the number of rename buffers. Such variation must show monotonic non–increase (i.e. decreasing or constant), as the resource size is increased. *(Note that increase in CPI implies decrease in performance).*

**Static loop execution time prediction:**

Based on prior work on static execution time estimation for loop structures [6,7,9,15–17], we have formulated an exact algorithm for predicting the steady–state period, and hence $CPI_{ss}$, as well as the cycle of onset, $N_{ss}$ of the steady–state pattern, in terms of the organizational parameters stated earlier (Section II). We omit the discussion of that methodology in this paper. We give a simplified example of the kind of end result one can obtain using the theory. Consider the daxpy test case example, discussed earlier. The following "rules of thumb" may be used to roughly summarize the theoretical estimation methods for daxpy–like floating point loops:

1. The loop–ending branch is fully overlapped with computation and takes "zero cycles" in the steady–state sense.
2. Let $N_L$ be the number of loads needed per iteration. This is the number of elements newly accessed on a given loop iteration.
3. Let $N_S$ be the number of stores needed per iteration. This is the number of target elements (to the left of an assignment) newly referenced on this iteration of the loop.
4. Let $N_F$ be the number of functional arithmetic instructions (other than divides) needed for the computation.
5. Let $N_D$ be the number of divides.

Remembering [16] that for this RS/6000 like ERISC machine,

1. A store takes 1 cycle (pipelined) and cannot be overlapped with loads or FMAs;
2. A load takes 1 cycle (pipelined) but can be overlapped with FMAs.
3. An FMA (floating multiply–add instruction) costs 1 cycle if it is independent of the previous FMA and 2 cycles if is dependent.
4. A divide is assumed to take D cycles (non–pipelined); D = 16 to 19 cycles for the RS/6000.

The minimum number of execution cycles per iteration, assuming independent FMAs, perfect instruction overlap, no divides, and a terminating zero–cost branch, is then:

$$T_{min} = N_S + \max(N_L + N_F).$$

If there are divides in the loop, $N_F$ needs to be replaced by $N_F + D*N_D$ in the above equation.

Substituting $N_S = 1$, $N_L = 2$, and $N_F = 1$ for the daxpy loop case, we get an analytically predicted steady state cycles per iteration of (1 + 2) = 3, which gives us a CPF of 1.5, identical to the $CPF_{ss}$ obtained from the timer run (Figure 4).

## V. Loop test case generation:

The test case generation methodology takes as input the basic organizational parameters of interest (Section II) and optional "bias" parameters, to focus attention on a limited aspect of the defined performance (timing) fault model (Section III). Once a complete suite of test loops is generated to cover all modeled faults across instruction classes, the level–x validation methodology (Section IV) is applied (x = 0, 1, 2). We provide three specific examples to illustrate our application loop–based methodology.

Examples of test case generation:

*Example 1:* The load–store test case. The high–level code segment used for generating compiled test sequences is clearly one or more instance of the assignment operation.

```
do i = 1, n
  a(i) = b(i)
enddo
```

The loop index n and the number of rename buffers are made sufficiently large. The functional timing faults detected by this test case are: (a) dispatch decode, execution or sequence fault: decode fault causes observed $CPI_{ss}$ to be less than expected value; execution fault causes observed $CPI_{ss}$ to be larger than expected; sequence fault is detected via disagreement between total number of instructions dispatched ($\Sigma$ num_disp) [see Procedure IIL_Gold, in Section III] and the actual trace length L. (b) FIX execution or sequence fault: an execution fault in the finish signal dispatch path will cause $CPI_{ss}$ to increase (possibly to $\infty$, causing level–0 failure of validation); sequence fault will also cause similar effect.

The fault–free $CPI_{ss}$ for this loop trace is clearly seen to be: 2/3 = 0.667 (see discussion on static loop execution prediction, Section IV).

*Example 2:* The reduction test–case: peak performance. We could construct this case by using a sequence of: lfd, fma and bc, and iterating over the loop.

```
do i = 1, n
  t = t + b*a(i)
enddo
```

The scalars t and b are loaded once outside the loop, causing the iteration loop to have the lfd, fma, bc sequence. Under adequate settings for rename buffers and queue sizes, this loop trace is expected to generate a $CPI_{ss}$ of 0.333 and a $CPF_{ss}$ of 0.5. Deviation upwards would generally point to execution or decode faults; downward deviation is not possible under modeled failures for this particular test case.

*Example 3:* The daxpy test case: store–bound behavior.

```
do i = 1, n
  x(i) = x(i) + s*y(i)
enddo
```

As mentioned previously, this test case is expected to generate a $CPF_{ss}$ of 1.5. Deviation upward to 2.0 will suggest a fault in the instruction completion logic (ICL), where the number of instructions completed per cycle may be erroneous, either due to local (BRN) logic fault or due to execution fault (finish signal dispatch error) in FIX or FLT. CPF increases may also be caused by faults in the load/store priority logic in accessing the single cache port. (This latter logic has not been specified in our earlier discussion).

**Experimental results with spec92 workload**

Systematic loop test case generation coupled with level–x validation/verification enables the designer to detect modeled timing faults, which are otherwise hard to detect and diagnose from large benchmark run results alone. The effect of making the model progressively robust through level–x validation is seen in practice through changes in experimentally observed SPEC integer92 benchmark suite cpi numbers for ERISC, as we progressed from level–0 to level–2. Table 1 shows the observed changes in cpi, over the course of verifying the machine timings, as embodied in the ERISC timer. In most cases, level–1 validation alone did not detect all the problems in the level–0 validated model. The overall difference in figures between the level–0 numbers and the level–2 numbers are quite significant, and it underlines the importance of performing systematic performance model verification/validation tests.

Table 1

| Benchmark | CPI (level–0 model) | CPI (level–1 model) | CPI (level–2 model) |
|---|---|---|---|
| compress | 0.992 | 0.892 | 0.891 |
| eqntott | 0.801 | 0.763 | 0.773 |
| espresso | 1.035 | 1.031 | 0.998 |
| gcc | 1.243 | 1.241 | 1.141 |
| li | 0.989 | 0.989 | 1.119 |
| sc | 1.133 | 1.133 | 1.134 |

## VI. Conclusion:

A systematic methodology for architectural timing verification and testing of super scalar CPU chips has been presented. This methodology is currently being used for testing, debugging and validation of advanced super scalar timer models within IBM's RS/6000 division. In principle, the test loop trace driven period analysis and related techniques may be used to test out the timing behavior of actual implemented hardware, or the full-scale logic simulation model, prior to chip tape-out. In practice, we have found the described methodology to be very useful for augmenting the traditional architectural verification programs (AVPs) with performance verification programs (PVPs) to form a complete, robust verification suite. Our robust timer validation methodology has helped us project very accurate processor performance numbers, and make accurate trade-off analyses, well before hardware implementation.

The goal of quantifying fault coverage in a manner analogous to hardware testing, has not been directly addressed. In other words, test case generation followed by the standard validation experiments, does not necessarily give us a measure of the percentage of faults covered. In ongoing research, we hope to address this issue.

Some of the ideas on testing and validation of timer models, described in this paper, have extended themselves into another application: design of self-checking architectural timers. Initial description of this approach is being published elsewhere.

**Acknowledgement:**

The author is indebted to Danny Shieh and Lawrence Hannon for many fruitful discussions on floating point loop timing assessment and measurement. The author is also grateful to his manager, Robert Amos, for his support and encouragement for applying the research ideas to the real product world.

## REFERENCES

1. T. Brodnax, M. Schiffli and F. Watson, "The PowerPC 601 design methodology," *Proc. IEEE Int'l. Conf. on Computer Design (ICCD)*, pp. 248-252, Oct. 1993.

2. S. M. Thatte and J. A. Abraham, "Test generation for microprocessors," *IEEE Trans. on Computers*, Vol. C-29, No. 6, pp. 429-441, June 1980.

3. D. Brahme and J. A. Abraham, "Functional testing of microprocessors," *IEEE Trans. on Computers*, Vol. C-33, pp. 475-485, June 1984.

4. S. Tahar and R. Kumar, "Towards a methodology for the formal hierarchical verification of RISC processors," *Proc. IEEE Int'l. Conf. on Computer Design (ICCD)*, pp. 58-62, Oct. 1993.

5. M. Srivas and M. Brickford, "Verification of a pipelined microprocessor using Clio," in: Hardware Specification, Verification and Synthesis: Mathematical Aspects, Eds., M. Leeser and G. Brown, Springer, 1990.

6. P. Bose, "Early performance estimation of super scalar machine models," *Proc. IEEE Int'l. Conf. on Computer Design, (ICCD)*, pp. 388-392, Oct. 1991.

7. P. Bose and J-D Wellman, "MIPS-driven design and estimation of VLSI CPUs," *Proc. IEEE VLSI Design '93*, Bombay, Jan. 1993.

8. P. Bose, "The TRISC machine architecture and timer," IBM Internal Document, T. J. Watson Research Center, Yorktown Heights, NY, Nov. 1991.

9. P. Bose, "Time attributed dependence graph scheme for prediction of execution time for a block of assignment statements with looping," IBM Tech. Discl. Bulletin, Vol. 36, No. 09A, pp. 621-622, September 1993.

10. Papers in session on: The Power PC 603 Microprocessor, Charles Moore, session chair, *Proc. IEEE Compcon*, pp. 300-323, 1994.

11. IBM, AIX Version 3.2 for RISC System/6000$^{TM}$, Assembler Language Reference, IBM Publication Number SC23-2197-01, 2nd Edition, Jan. 1992.

12. R. R. Oehler and R. D. Groves, "IBM RISC System/6000 processor architecture," *IBM Journ. of Res. and Develop.*, Vol. 34, No. 1, pp. 23-36, Jan. 1990.

13. IBM, AIX Operating System, C Language Reference, Publ. No. SC23-2058-02, 3/91.
IBM, AIX XL Fortran Compiler/6000, Language Reference, ver. 2.3, Publ.No. SC09-1353-02, Sept. 1992.

14. "Atrace": IBM Internal Software; 1991; author: R. Nair, IBM T. J. Watson Research Center, Yorktown Heights, NY.

15. IBM, International Technical Support Centers, "Predicting execution time on the IBM RISC System/6000," Document No. GG24-3711, July 1991

16. IBM, International Technical Support Centers, "IBM RISC System/6000 NIC Tuning Guide for Fortran and C," Publication Number GG24-3611-01, July 1991.

17. W. Mangione-Smith, S. G. Abraham and E. S. Davidson, "A performance comparison of the IBM RS/6000 and the Astronautics ZS-1," *Proc. IEEE Computer*, pp. 39-46, Jan. 1991.