Mixed–mode, analogue–digital simulation using SPICE–like circuit analysis programs

P.E. Allen* and W.M. Zuberek†

*School of Electrical Engineering, Georgia Institute of Technology, Atlanta, GA 30332, USA
†Department of Computer Science, Memorial University of Newfoundland, St.John’s, Canada A1B 3X5

An approach to integrated mixed-mode simulation is described in which the popular SPICE circuit simulator is extended to analyze digital circuits at the gate, functional or behavioral levels. The analogue-to-digital conversion is provided by (multilevel) parameterized “thresholders” which assign digital signals to subranges of voltages determined by a set of voltage thresholds. The digital-to-analogue conversion is implemented by voltage sources with “controlled” characteristics that convert discrete digital signals into continuous piecewise linear functions. The paper outlines internal organization of mixed, analogue–digital simulation, discusses modification of analyses and extensions to the input language needed for analogue–digital interactions, and highlights the specification of digital circuits. Simple examples of mixed, analogue–digital circuits and mixed–mode simulation illustrate this approach.

1. Introduction

The phrase “mixed analogue–digital” or “mixed–mode” simulation has been used to refer to the simulation of electrical networks consisting of both analogue and digital parts, regardless of the level of design abstraction. Design abstraction, in this context, means the level at which the network is specified for simulation; it can be device, circuit, functional or behavioral level.

The popularity that mixed–mode simulation has been gaining [3,4,6,9,10] is due to analogue circuitry that exists in virtually all digital systems. Some of that circuitry is now showing up in application–specific ICs. It is expected that within a year, half of all semicustom cirquitry, and one third of standard–cell designs will be more than 10 percent analogue [4]. Roughly 80 percent of all printed circuit boards contain some analogue components today.

It appears, however, that even sophisticated users of design automation tools prefer to separate the analogue and digital portions of their systems; the analogue circuitry is generally verified using some derivation of the popular SPICE circuit simulator [2,11], and there are many commercially available logic simulators that can be used for simulation of digital circuits [12]. Unfortunately, it is rather difficult to correlate these two distinct simulations, and to analyze how analogue and digital parts affect each other. Clearly, an “integrated” approach is needed which can handle both analogue and digital simulation within one simulation environment.

Three fundamental approaches have been taken toward true mixed–mode simulation. The “coupled” (or “glued”) approach uses separate analogue and digital simulators and establishes some communication mechanism between them; for example, PSPICE–VIEWSIM.
[3] uses UNIX “pipes” for communication between two simulators running as two different processes. The “augmented” approach extends either a digital simulator to handle behavioral analogue models, or an analogue simulator to handle digital models; for example, DIANA [6] extends analogue simulation by Boolean equations describing the digital component; SAMSON [9] applies event–driven control, typical for digital simulation, to both analogue and digital components. Finally, the “integrated” approach presumes that a tightly coupled pair of algorithms handles both analogue and digital simulation.

Any implementation of integrated mixed-mode simulation must solve two basic questions [10], (i) conversion of analogue to digital and digital to analogue information on interfaces of analogue and digital components, and (ii) synchronization of the (usually variable) timesteps of the analogue simulation [8] with the event list that drives the (usually event–driven) digital simulation [7]. The analogue–to–digital conversion can be handled by establishing voltage thresholds and corresponding digital signals (the conversion is performed by elements called “thresholders” [6]). The digital–to–analogue conversion is much more difficult because it must generate a continuous analogue signal on the basis of discrete digital values. Two popular simple solutions assume that the converted waveforms are either piecewise linear (with averaged levels) or piecewise exponential.

This paper describes an integrated mixed-mode simulation in which the analogue simulation is provided by a modified SPICE simulator (the SPICE–PAC simulation package) while the digital simulation is either user–supplied (at any level of abstraction) by appropriate simulation routines, or is performed by a typical digital simulator with a “flexible” time control mechanism to provide synchronization with the analogue simulation algorithm, as shown in Fig.1.

The paper is organized in four main sections. Section 2 describes analogue simulation with modified time–domain analysis, extended by an interface to digital simulation. Section 3 presents extensions to the input language required for specification of analogue–digital and digital–analogue interfaces; it also describes the interfacing routines in some detail. User–defined simulation of the digital part of a circuit is discussed in section 4, and section 5 describes a new digital simulation software that is being developed specifically for integrated mixed-mode simulation.

2. Analogue simulation

Analog simulation is provided by SPICE–PAC [13,14], a simulation package that is upward compatible with the popular SPICE simulation program [2,11]. It means that SPICE–PAC accepts the same circuit description and performs all the analyses which are available in the SPICE programs, but also provides mechanisms that do not exist in SPICE, for example accessing internal values of circuit elements, dynamic definitions of parameters and outputs, hierarchical naming scheme for subcircuits, parameterized subcircuit expansion, and also “enhanced” circuit simulation in which users can extend, modify or replace some standard simulation capabilities by their own algorithms in order to increase efficiency, accuracy or applicability of simulators. Externally controlled time–domain analysis [14] is an extension which is used in mixed–mode simulation to synchronize the analogue and digital simulation algorithms at the internal level of timestep/time–event control [2,8].

SPICE–PAC, similarly to the SPICE program, implements the time–domain (transient) analysis in two consecutive steps, the so called “Initial Transient” analysis and the proper “Transient” analysis [2]. Initial transient analysis finds the “initial” solution (i.e., the solu-
tion of circuit equations for the zero time instant) with user–specified or default initial conditions. The proper transient analysis integrates the differential equations describing the circuit from one timepoint to another, using either the trapezoidal rule (default) or backward differentiation (or Gear’s) method.

The proper time–domain analysis is controlled by two mechanisms, (i) the variable timestep which is determined by user specified parameters as well as truncation errors estimated during integration steps, and (ii) the so called “breakpoint table”, BPT, which contains all “characteristic” time instances of time–dependent source functions as well as contributions introduced by transmission lines.

The variable timestep is controlled by iteration count and an estimated truncation error. The iteration count uses the number of Newton–Raphson iteration steps required to converge at a given timepoint; if this number is less than the parameter ITL3, the timestep is doubled provided it does not exceed the maximum timestep determined by the value of $\text{DELMAX}$. $\text{DELMAX}$ is either specified as one of parameters of time–domain analysis, or (by default) $\text{DELMAX} = (\text{Time\_stop} - \text{Time\_start})/50$, where Time\_stop and Time\_start indicate the beginning and the end of the simulation interval.

Furthermore, if the number of iterations is greater than the limit ITL4, the Newton–Raphson iteration is terminated as nonconvergent, the timestep is divided by 8, and the iterative solution begins for a new timepoint, provided the reduced timestep is greater than the minimum timestep, $\text{DELMIN} = 10^{-9}\times \text{DELMAX}$ (otherwise the notorious message “INTERNAL Timestep TOO SMALL” is reported and the analysis terminates).

The estimation of (local) truncation errors determines the minimum timestep for which the truncation errors are “acceptable”. When the truncation errors (for the assumed timestep) are “unacceptable”, the integration backtracks in time, and the new, “acceptable” timestep is used provided it is greater than the minimum timestep $\text{DELMIN}$.

In SPICE–PAC, the original implementation of the time–domain analysis has been extended by invocations of an auxiliary routine digital for simulation of digital parts of circuits. These additional invocations are within the initial section of analysis and within the iterative integration scheme. digital is an (internal) interfacing routine, which returns the “time” of the next event $\text{TNEXT}$ if the digital simulation is event–driven (otherwise $\text{TNEXT}$ is negative).

digital is discussed in greater detail in the following section.

initialize;
TIME:=0;
update_time_dependent_sources(TIME);
solve_the_initial_system_of_circuit_equations;
create_breakpoint_table(BPT);
DELTAT:=Time\_step;
DELBKP:=DELTAT;
DELMIN:=1D-9*DELTAT;
if mixed-simulation then
call digital(initial_parameters);
IBRTAB:=1;
BREAKP:=true;
error:=false;
while not error do
if mixed_simulation then
{ call digital(current_parameters,TNEXT);
if terminate then Time\_stop:=TIME-DELMIN;
if $\text{TNEXT} > 0$ then
if $\text{TNEXT} < \text{TIME} + \text{DELTAT}$ then
DELTAT:=TNEXT-\text{TIME} 

store_the_solution;
if TIME > Time\_stop then
interpolate_output_results_and_return;
if BREAKP then
{ IBRTAB:=IBRTAB+1;
DELTAT:=\text{min}(DELTAT, 0.1*\text{min}(DELBKP,BPT[IBRTAB]-\text{TIME})]);
if IBRTAB=2 then DELTA:=0.1*DELTAT;
BREAKP:=false }
else if TIME+\text{DELTA} > BPT[IBRTAB] then
{ DELBKPD:=DELTAT;
DELTAT:=BPT[IBRTAB]-\text{TIME};
BREAKP:=true }
call solve_timepoint;

while not error do
if mixed_simulation then
{ call digital(current_parameters,TNEXT);
if terminate then Time\_stop:=TIME-DELMIN;
if $\text{TNEXT} > 0$ then
if $\text{TNEXT} < \text{TIME} + \text{DELTAT}$ then
DELTAT:=TNEXT-\text{TIME} 

store_the_solution;
if TIME > Time\_stop then
interpolate_output_results_and_return;
if BREAKP then
{ IBRTAB:=IBRTAB+1;
DELTAT:=\text{min}(DELTAT, 0.1*\text{min}(DELBKP,BPT[IBRTAB]-\text{TIME})]);
if IBRTAB=2 then DELTA:=0.1*DELTAT;
BREAKP:=false }
else if TIME+\text{DELTA} > BPT[IBRTAB] then
{ DELBKPD:=DELTAT;
DELTAT:=BPT[IBRTAB]-\text{TIME};
BREAKP:=true }
call solve_timepoint;

while not error do
if mixed_simulation then
{ call digital(current_parameters,TNEXT);
if terminate then Time\_stop:=TIME-DELMIN;
if $\text{TNEXT} > 0$ then
if $\text{TNEXT} < \text{TIME} + \text{DELTAT}$ then
DELTAT:=TNEXT-\text{TIME} 

store_the_solution;
if TIME > Time\_stop then
interpolate_output_results_and_return;
if BREAKP then
{ IBRTAB:=IBRTAB+1;
DELTAT:=\text{min}(DELTAT, 0.1*\text{min}(DELBKP,BPT[IBRTAB]-\text{TIME})]);
if IBRTAB=2 then DELTA:=0.1*DELTAT;
BREAKP:=false }
else if TIME+\text{DELTA} > BPT[IBRTAB] then
{ DELBKPD:=DELTAT;
DELTAT:=BPT[IBRTAB]-\text{TIME};
BREAKP:=true }
call solve_timepoint;

B BREAKP is a logical flag that is used in coordination of the variable timestep DELTA with the BPT; its “true” value indicates that a breakpoint from BPT has been used, and then the timestep is reduced at least 10 times (and it is reduced once more 10 times for the initial step) in anticipation of “special” changes of voltages and/or currents at the breakpoint.
Because digital simulation may change the timepoint (iterating a threshold value of an analogue–to–digital signal, as described in the next section), it is invoked before the store_the_solution operation.

The procedure solve_timepoint is as follows:

```plaintext
timepoint:
  TIME:=TIME+DELTA;
  if total_no_of_iterations>ITL5 and ITL5>0 then
    set_error("limit of iteration steps reached")
  else if execution_time > time_limit then
    set_error("execution time limit reached")
  else
    { update_time_dependent_sources(TIME);
      solve_the_system_of_circuit_equations;
      if converged then
        { DELOLD:=DELTA;
          estimate_integration_errors_and_adjust(DELTA);
          if integration_error_is_acceptable then
            return;
          TIME:=TIME-DELOLD }
      else
        { TIME:=TIME-DELTA;
          DELTA:=DELTA/8 ;
          BREAKP:=false;
          if DELTA >= DELMIN then go to timepoint ;
        }
      set_error ("timestep too small");
    return;

The solve_timepoint procedure is also used in analogue–to–digital conversion during the iteration of “threshold timepoints”.

3. Interfacing digital simulation

A basic analogue–to–digital interface implemented in SPICE–PAC provides a table–driven conversion of analogue to (multivalued) digital signals and vice versa. Piecewise linear characteristics of independent voltage and/or current sources [2,11] are used for interactions between digital and analogue subnetworks; the “smoothing” of discrete digital signals is thus implemented by piecewise linear functions. The interface is composed of two sections, one for analogue–to–digital communication, and the second for communication in the opposite direction. In the input (circuit specification) language, these two sections are described by two new directives, PUTLIST and GETLIST, respectively:

```plaintext
.PUTLIST:Tname1 Voutput1,Voutput2,...
.GETLIST:Tname1:Tname2 Vsource1,Vsource2,...
```

where Tname1 indicates a TABLE pseudoelement (similar to a MODEL of semiconductor devices) [14] that defines the conversion table for analogue–to–digital (and digital–to–analogue) interface; Tname2 indicates another TABLE pseudoelement that defines the delay table for digital–to–analogue conversion; each Voutput is a voltage output in the SPICE sense, i.e., it is either “V(node1,node2)” or “V(node1)” if the second node is zero; each Vsource is the name of an independent voltage source with a piecewise linear time–dependent function.

The conversion table is defined as an ordered sequence of increasing (threshold) voltages interposed with (internal) values of corresponding digital signals:

```plaintext
.TABLE Tname Volt0 Num1 Volt1 Num2 ... Numk Voltk
```

The digital equivalent of voltages in the range Volt0 to Volt1 is a signal represented by the value Num1, etc. For digital–to–analogue conversions, the extreme values Num1 and Numk are translated into Volt0 and Voltk, respectively, while all intermediate values are converted into “median” voltages, i.e., Num2 corresponds to 

```
VV 1 0 PULSE(-5.0,+5.0,0.5US,10NS,10NS,2US,5US)
R1 1 2 1K
C1 2 0 200PF
R2 1 3 1K
C2 3 0 1NF
VX 5 0 PWL(0 -5.0,15U -5.0)
RX 5 0 1K
```
Mixed-mode, analogue–digital simulation using SPICE–like circuit ...

The digital results of time-domain simulation for this circuit are shown in Fig.2, and the conversion to analogue “V(5)” is shown in Fig.3.

The output begins to rise at approximately 0.8µs, i.e., 0.2µs (the delay in the TABLE TDEL) after the “higher” input (“V(2)” in this case) reaches the threshold of 0V. Similarly, “V(5)” starts to fall at approximately 1.4µs, i.e., the same 0.2µs after the “lower” input (in this case “V(3)”) reaches the threshold of 0V, etc.

The condition mixed_simulation, used in the modified time-domain analysis (previous section), is satisfied only if both PUTLIST and GETLIST are nonempty, and then the interfacing routine digital is invoked for each successfully solved timepoint. The interfacing routine performs analogue–to–digital conversion of
all PUTLIST voltages, and then checks if any digital value created during this conversion differs from the “previous” value; if all digital values remain the same, a check is made if the present timepoint has been explicitly requested by the digital simulation routines (for example, because of the “internal” timing mechanisms). If both checks fail, the interfacing routine terminates, and the time-domain analysis resumes otherwise the digital simulation is performed. If any one of the PUTLIST signals changes its (digital) value, before invocation of the digital simulator the timepoint is (iteratively) adjusted to a value corresponding to the closest analogue–digital conversion threshold.

Digital simulation is performed either by a “standard” digital (or logic) simulator, or by a user–defined routine which analyzes the digital part at the gate, functional or behavioral level. After completion of digital simulation of a single event, the digital–to–analogue conversions are performed for all those (digital) signals which are indicated in the GETLIST specification and which changed their (digital) values during the simulation.

The outline of the interfacing routine is as follows:

```plaintext
digital_simulation:=false;
min_threshold:=max_value;
for i:=1 to length_of_PUTLISTs do
  extract_the_voltage(i,voltage);
  convert_analogue(voltage,signal);
  if signal <> old_input_signal[i] then
    { digital_simulation:=true;
      find_the_closest_threshold(voltage,x);
      if x < min_threshold then
        { min_threshold:=x;
          n:=i;
        }
      old_input_signal[i]:=signal }
endfor;
if digital_simulation then
  iterate(n,min_threshold)
else if TIME=next_event_time then
  digital_simulation:=true;
perform_one_step_of_digital_simulation;
for j:=1 to length_of_GETLIST do
  extract_the_signal(j,signal);
  if signal <> old_output_signal[j] then
    { update_the_voltage_source(j,signal);
      old_output_signal:=signal }
endfor;
check_pending_events(TNEXT);
```

The `iterate(n,threshold)` procedure performs a simple but robust K–step iteration (K is a parameter) in which the `old_input_signal` is now the “new” value of the corresponding signal:

```plaintext
Dmin:=0.0;
Dmax:=DELTA;
tol:=epsilon*abs(threshold+epsilon)
i:=0;
while i < K do
  TIME:=TIME-DELTA;
  DELTA:=(Dmin+Dmax)/2;
  call solve_timepoint;
  extract_the_voltage(n,voltage);
  if abs(threshold-voltage) < tol then
      return
  convert_analogue(voltage,signal);
  if signal=old_input_signal[n] then
    Dmax:=DELTA
  else Dmin:=DELTA;
  i:=i+1
endwhile;
```

`epsilon` is another parameter that determines relative accuracy of “threshold iteration”; its default value is 0.05 .

### 4. User–defined digital simulation

Digital simulation is controlled by a routine called SPUSIM, which is either a simple interfacing routine to a “standard” logic or digital simulation program [7,12], or a user–defined routine which performs the required digital simulation (at any level). Its definition must be consistent with the following (FORTRAN) header:

```fortran
SUBROUTINE SPUSIM (TIME,LINP,NINP,LOUT,NOUT,MARK)
  DOUBLE PRECISION TIME
  INTEGER LINP(NINP),LOUT(NOUT),MARK
```

where the parameters are:

- `TIME` – the value of the actual (simulated) time; its value cannot be changed as it is determined by the simulation algorithm,
- `LINP` – an array of length `NINP` which contains the converted (i.e., digital) values of PUTLIST data,
- `NINP` – the length of LINP, i.e., the number of analogue–to–digital signals,
LOUT – an array of length NOUT which returns the new (digital) values of GETLIST variables; on entry, LOUT contains “previous” values of GETLIST variables, so only changes need to be stored in LOUT.

NOUT – the length of LOUT, i.e., the number of digital-to-analogue signals.

MARK – an entry/return flag; on entry: MARK=-1 indicates the initial invocation, MARK=0 indicates an accepted timepoint (i.e., a “regular” invocation), while nonconvergence (and termination of analogue simulation) is indicated by MARK=+1; on exit: MARK=0 indicates continuation of analysis, and MARK=+1 a request to terminate the analysis at the current timepoint (the condition ”terminate” in the modified time-domain analysis of section 2).

The following (Fortran) SPUSIM routine simulates a two-input XOR (exclusive–OR) gate and prints a trace of all invocations:

```
SUBROUTINE SPUSIM (TIME,LINP,NINP,LOUT,NOUT,*MARK)
  DOUBLE PRECISION TIME
  DIMENSION LINP(NINP),LOUT(NOUT)
  IF (LINP(1).EQ.LINP(2)) THEN
    LOUT(1)=-1
  ELSE
    LOUT(1)=+1
  ENDIF
  WRITE(*,500) TIME,LINP(1),LINP(2),LOUT(1)
500 FORMAT('*spusim* :- time :',1PD12.4,' inp',2I3,' out',I3)
RETURN
END
```

When it is used with the analogue circuit shown previously (section 3), it generates – at the digital side of the interface – the trace which is shown below:

```
*spusim* :- time : 0.0000d+00 inp -1 -1 out -1
*spusim* :- time : 6.4360d-07 inp -1 1 out 1
*spusim* :- time : 1.1986d-06 inp 1 1 out -1
*spusim* :- time : 2.6536d-06 inp 1 -1 out 1
*spusim* :- time : 3.0649d-06 inp -1 -1 out 1
*spusim* :- time : 5.6436d-06 inp -1 1 out -1
*spusim* :- time : 6.1541d-06 inp 1 1 out -1
*spusim* :- time : 7.6536d-06 inp -1 -1 out -1
*spusim* :- time : 8.0716d-06 inp -1 -1 out -1
```

For simulation of digital circuits with internal delays, the TNEXT parameter of the procedure digital (section 3) has to be set properly. This is done by an internal procedure that is controlled (from SPUSIM) through a COMMON area

```
COMMON /SPP0WE/ LENEVQ,TMNXEV
```

in which LENEVQ indicates the length of the event queue, and for nonzero values of LENEVQ, TMNXEV is the “next event time” (which is immaterial when LENEVQ=0).

Simulation of the digital part with internal delays is illustrated by another simple example in which a two-input one-output digital block is composed of a NOR gate connected to a two-input OR gate; the second argument of the OR gate is the same as the second argument of the NOR gate. The propagation delay of the NOR gate is 100 ns, and the propagation delay of the OR gate is included in the digital-to-analogue conversion. The SPUSIM routine that performs digital simulation for this example is as follows:

```
SUBROUTINE SPUSIM (TIME,LINP,NINP,LOUT,NOUT,*MARK)
  DOUBLE PRECISION TIME,TMNXEV
  INTEGER LINP(NINP),LOUT(NOUT)
  COMMON /SPP0WE/ LENEVQ,TMNXEV
  SAVE INT1,INT2
  IF (MARK.LT.0) THEN
    LENEVQ=0
    INT1=-MAX0(LINP(1),LINP(2))
    INT2=INT1
  ELSE
    IF (LENEVQ.GT.0 .AND. DABS(TMNXEV-TIME).LT.1D-20) THEN
      INT2=INT1
      LENEVQ=0
    ENDIF
    NOR=-MAX0(LINP(1),LINP(2))
    IF (NOR.NE.INT1) THEN
      IF (LENEVQ.GT.0 .AND. DABS(TMNXEV-TIME).LT.1D-20) THEN
        INT2=INT1
        LENEVQ=0
      ENDIF
    ELSE
      ENDIF
      IF (NOR.EQ.INT1) THEN
        IF (LENEVQ.EQ.0) THEN
          TMNXEV=TIME+1D-7
          WRITE(*,200) TIME,TMNXEV,INT1,NOR
200 FORMAT('*spusim* :- delayed event :',1P2D9.2,2I3)
          INT1=INT1
          LENEVQ=0
          ENDIF
          IF (NOR.EQ.INT1) THEN
            IF (LENEVQ.EQ.0) THEN
              TMNXEV=TIME+1D-7
              WRITE(*,200) TIME,TMNXEV,INT1,NOR
```

For simulation of digital circuits with internal delays, the TNEXT parameter of the procedure digital (section 3) has to be set properly. This is done by an internal procedure that is controlled (from SPUSIM) through a COMMON area

```
COMMON /SPP0WE/ LENEVQ,TMNXEV
```

in which LENEVQ indicates the length of the event queue, and for nonzero values of LENEVQ, TMNXEV is the “next event time” (which is immaterial when LENEVQ=0).

Simulation of the digital part with internal delays is illustrated by another simple example in which a two-input one-output digital block is composed of a NOR gate connected to a two-input OR gate; the second argument of the OR gate is the same as the second argument of the NOR gate. The propagation delay of the NOR gate is 100 ns, and the propagation delay of the OR gate is included in the digital-to-analogue conversion. The SPUSIM routine that performs digital simulation for this example is as follows:

```
SUBROUTINE SPUSIM (TIME,LINP,NINP,LOUT,NOUT,*MARK)
  DOUBLE PRECISION TIME,TMNXEV
  INTEGER LINP(NINP),LOUT(NOUT)
  COMMON /SPP0WE/ LENEVQ,TMNXEV
  SAVE INT1,INT2
  IF (MARK.LT.0) THEN
    LENEVQ=0
    INT1=-MAX0(LINP(1),LINP(2))
    INT2=INT1
  ELSE
    IF (LENEVQ.GT.0 .AND. DABS(TMNXEV-TIME).LT.1D-20) THEN
      INT2=INT1
      LENEVQ=0
    ENDIF
    NOR=-MAX0(LINP(1),LINP(2))
    IF (NOR.NE.INT1) THEN
      IF (LENEVQ.GT.0 .AND. DABS(TMNXEV-TIME).LT.1D-20) THEN
        INT2=INT1
        LENEVQ=0
      ENDIF
    ELSE
      ENDIF
      IF (NOR.EQ.INT1) THEN
        IF (LENEVQ.EQ.0) THEN
          TMNXEV=TIME+1D-7
          WRITE(*,200) TIME,TMNXEV,INT1,NOR
200 FORMAT('*spusim* :- delayed event :',1P2D9.2,2I3)
          INT1=INT1
          LENEVQ=0
          ENDIF
          IF (NOR.EQ.INT1) THEN
            IF (LENEVQ.EQ.0) THEN
              TMNXEV=TIME+1D-7
              WRITE(*,200) TIME,TMNXEV,INT1,NOR
Mixed-mode, analogue–digital simulation using SPICE–like circuit ...

When this procedure is used with the analogue part shown earlier (section 3), it generates the following execution trace:

*spusim* :- time : 0.0000D+00 inp -1 -1 out 1
*spusim* :- delayed event : 6.45D-07 7.45D-07 1 -1
*spusim* :- time : 6.4475D-07 inp 1 -1 out 1
*spusim* :- time : 7.4475D-07 inp 1 -1 out -1
*spusim* :- time : 1.1983D-06 inp 1 1 out 1
*spusim* :- time : 2.6536D-06 inp -1 1 out 1
*spusim* :- delayed event : 3.06D-06 3.16D-06 -1 1
*spusim* :- time : 3.0643D-06 inp -1 -1 out -1
*spusim* :- time : 3.1643D-06 inp -1 -1 out 1
Mixed–mode, analogue–digital simulation using SPICE–like circuit ...

It can be observed that the “delayed” events are the only events that are analyzed without changes of (digital) input signals. The corresponding waveforms are shown in Fig.4 and Fig.5.

The delay of 100 ns is clearly visible between the rising section of the “V(2)” waveform and the switch of the digital signal from “high” to “low” (and also between the falling section of “V(3)” and the switch from “low” to “high”). Fig.5 shows characteristic “spikes” which can be generated because of internal delays and which depend upon the values of these delays.

5. Integrated digital simulation

User–defined simulation routines (the SPUSIM procedure) are quite flexible and very efficient since they can use different levels of representation without any restrictions, however, they also are rather unreliable (a new version of SPUSIM is needed for each new application) and they require a detailed knowledge of simulator’s interfaces. In many cases, more “user friendly” approach is needed, in which the digital simulation is derived directly from design specifications. Such a capability is provided by a new digital simulator that is being developed specifically for integration with SPICE–PAC [5]. It is an event–driven, gate–level simulator with functional and – possibly – behavioral levels of simulation to be implemented as subsequent extensions.

5.1. Event–driven simulation

In event–driven simulation, analysis of the circuit is controlled by “events” which indicate changes of node values (“node events”) or internal states (“state events”). Formally, an elementary event can be represented as a quadruple (Type,Id,Value,Time), where “Type” is either NODE or STATE, “Id” is the corresponding node or block identifier, “Value” is either the new node signal or the new block state, and “Time” is the time instant at which the event (i.e., the change) should occur. An event is defined as the collection of all node and state elementary events with the same “Time” component.

To execute events in proper order (i.e., in increasing time sequence), an ordered list of events (called the “event queue”) is maintained by the simulator. Generally, the event queue must support two basic operations: (1) insertion of an elementary event in increasing time order (denoted by insert(event,Event_queue) in the following description), and (2) extraction of the first (in time order) event that becomes the “current” event (denoted by extract(event,Event_queue)).

perform_one_step_of_digital_simulation is a procedure (used in section 3) which extracts and executes the first Event from the event queue Event_queue using an iterative forward propagation of signals through zero–delay components of the digital circuit [5]. It is composed of three consecutive phases. The first phase analyzes Elements (or elementary events) of the extracted Event and subdivides them into immediate events (the set Immed) and “delayed” events (the set Later); the immediate events correspond to changes of node values, and these changes can propagate further through “zero–delay” elements – processing of the immediate events is done in the second phase. Handling of block events, i.e., changes of (internal) block states, depends on the type of the new state. For SELFTIMED states (i.e., states with delays associated with them, as in clock generators), a new elementary event is added to the Later set with the “Time” component including the required state delay. For all new states, the block outputs are set to (new) state–dependent values which may also include output delays (and then the corresponding elementary events are added to the Later set):

extract(Event,Event_queue);
TIME:=Time(Event);
Immed:=empty;
Later:=empty;
for each Element in Event do
    if Type(Element)=NODE then
        include(Element,Immed)
    else
        { Block:=Id(Element);
          State:=change_state(Block);
          if Type(State)=SELF_TIMED then
            { New_element:=(BLOCK,Block,
                Next_state(Block),TIME+Delay(State));
              include(New_element,Later) ;
            } for each Output in Outputs(Block) do
            New_element:=(NODE,Node(Outout),
                Value(Output),TIME);
            if Delay(Output)=0 then
                include(New_element,Immed)
            else
                { Time(New_element):=TIME+Delay(Output);
                  include(New_element,Later) }
            endfor }
        endfor;
    endfor;

    The second phase iteratively propagates changes of node values through blocks connected the nodes; fanout(Node) is used to denote the set of all blocks that use Node as one of their inputs. The iteration continues until the set Immed becomes empty, or until the number of block evaluations Count(Block) reaches the limit Limit which is assumed to indicate oscillations in the circuit. Used holds all blocks used in the iterative process, and each of these blocks is “marked” by a binary flag flag(Block) set to ON. Each step of the iteration has to stages; the first stage creates the set Active of all blocks that need to be evaluated because at least one of their inputs have changed (as a result of the previous iteration step); the second stage evaluates the blocks in Active and creates the (new) set Immed of all those nodes which are connected to “zero–delay” outputs of evaluated blocks and which change their values:

    Used:=empty;
    while nonempty(Immed) do
        Active:=empty;
        for each Element in Immed do
            Node:=Id(Element);
            if Value(Element)<>Value(Node) then
                for each Block in fanout(Node) do
                    if flag(Block)=OFF then
                        { flag(Block):=ON;
                          Count(Block):=0;
                          append(Block,Used) };
                    Count(Block):=Count(Block)+1;
                    if Count(Block) > Limit then
                        stop("Oscillations");
                        include(Block,Active)
                    endfor
                endfor;
            endfor;
            Immed:=empty;
            for each Block in Active do
                evaluate(Block);
                for each Output in Outputs(Block) do
                    New_element:=(NODE,Node(Output),
                        Value(Output),TIME);
                    if Delay(Output)=0 then
                        include(New_element,Immed)
                    else
                        { Time(New_element):=TIME+Delay(Output);
                          include(New_element,Later) }
                    endfor
                endfor
            endwhile;

        The last phase resets all used blocks and then inserts all elements of Later into the Event_queue (it also performs some other functions [5] which do not seem important here):

        for each Block in Used do
            flag(Block):=OFF
        endfor;
        for each Element in Later do
            insert(Element,Event_queue)
        endfor;

        Finally, LENEVQ and TMNXEV parameters (section 3) are set before return from the procedure perform_one_step_of_digital_simulation:

        LENEVQ:=length(Event_queue);
        if LENEVQ>0 then TMNXEV:=Time(first(Event_queue));

        where first(Event_queue) denotes the first event of the event queue.

5.2. Circuit description

The input description (as a section of the input data) for this integrated simulator is composed of three sections, “input”, “output” and “circuit”. The input section specifies a list of primary (digital) inputs that must match (in number and in order) the PUTLIST elements indicated at the analogue side of the analogue–to–digital interface. The output section indicates a list of primary (digital) outputs that must match the GETLIST elements. The circuit–end section is a gate–level description of the circuit. The description of the first example from section 4 is:
Mixed–mode, analogue–digital simulation using SPICE–like circuit ...  43

input : x1,x2;
output : y;
circuit
  y:=xor(x1,x2)
end

Since the simulator is intended to deal primarily with gate–level circuit descriptions, its input language (for the “circuit” section) is oriented toward multiple–input, single–output blocks. For this reason, the basic language primitives are gate functions which applied to a set of arguments yield a single result, as in:

output := operator(input1,input2,...)

where each input can be another function specification:

output := operator1(operator2(input21,...),...)

The list of available operators includes the collections of two– and three–input AND, OR, NAND, NOR, XOR gates, inverters and buffers, denoted AND or AND2, AND3, OR or OR2, OR3, and so on.

In order to extend the notation to multiple–output blocks, the language allows the construct:

(output1,output2,...) := function(input1,...)

The list of outputs must be consistent with function implemented at each of function’s outputs. Several “standard” two–output blocks are available:

<table>
<thead>
<tr>
<th>Block name</th>
<th>List of inputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLK-SR-FF</td>
<td>S,R,Clock,Preset,Clear</td>
</tr>
<tr>
<td>CLK-D-FF</td>
<td>D,Clock,Preset,Clear</td>
</tr>
<tr>
<td>CLK-JK-FF</td>
<td>J,K,Clock,Preset,Clear</td>
</tr>
<tr>
<td>CLK-T-FF</td>
<td>T,Clock,Preset,Clear</td>
</tr>
</tbody>
</table>

All these devices are clocked, have direct (not inverted) inputs, and supply both direct and inverted outputs. Other blocks can be added easily.

There is an additional, optional define section of the circuit description that provides several new capabilities to the input language; it allows for assigning delay values to circuit elements, and it introduces a pseudo–“macro” mechanism to set up constant block inputs and redistribute block input lines. If used, it must be the first section of the circuit description. For example:

define nor = nor2[rise=2E-8,fall=3E-8];

defines (or redefines, more precisely) nor as a nor2 gate with the rise delay of 20ns and the fall delay of 30ns. All subsequent occurrences of nor in the circuit description will refer to this refined nor2 gate with indicated rise and fall delay times.

The second example from section 4 (with internal delay) can thus be specified as:

define nor=nor2[rise=1E-7,fall=1E-7];
inout : x1,x2;
output : y;
circuit
  y:=or2(nor(x1,x2),x2)
end

Sometimes it may be useful to “redistribute” the inputs to all instances of a particular block in a circuit; the define pseudo-macro mechanism is helpful in such situations. For example, if inverters are not available as primitive blocks in some technology, they can be created from nand2 gates by the following definition:

define inv = nand2(_1,_1);

which indicates that the (single) input “_1” of the inv block is assigned to both inputs of the nand2 gate. Usually this will be coupled with delay specifications, for example:

define nand = nand2[rise=5E-8,fall=5E-8];
inv = nand(_1,_1);

or, if the delay times of the nand2 gates and the inv are different, as:

define nand = nand2[rise=5E-8,fall=5E-8];
inv = nand2(_1,_1)[rise=4E-8,fall=4E-8];

Similarly, certain block inputs can be set to constant values using the pseudo–macro mechanism. For example, to set the “Clock”, “Preset” and “Clear” inputs of a CLK-SR-FF block to “high”, “low” and “low” permanently (these values are denoted by .0., .1. and .X., respectively), the following definition can be used:
define new-sr = clk-sr-ff(.,1,_,2,_,1,.,0,.,0,.,0,.,0);

which is then used in the circuit description as:

circuit ... (q,notq) := new-sr(s,r); ... end

More details and complete examples of digital descriptions are given in [5].

6. Concluding remarks

A SPICE-compatible circuit simulator has been modified and extended to allow simulation of circuits composed of analogue and digital blocks. The description of analogue parts has been enhanced by two new directives that specify parameterized analogue-to-digital and digital-to-analogue interfaces. Analog-to-digital conversion of signals is performed by multilevel ideal thresholders while “controlled” voltage sources with piecewise linear characteristics convert discrete digital signals into continuous analogue ones. Two methods of digital simulation are supported, a flexible and efficient but difficult to use direct program representation of simulated circuits (the user-defined simulation) and an integrated gate-level event-driven simulation capability with a block-oriented input description language.

The extended simulator has been used to analyze a number of mixed analogue-digital circuits, and in particular several A/D converters (Allen et al., 1990). In all cases a significant reduction of simulation times with respect to “all-analogue simulation” has been observed (in the range of one to two orders of magnitude depending on the level of digital simulation).

Because of complexity of integrated circuits, further research is needed in – at least – the following directions:

- Implementation of higher-level simulation capabilities within both digital and analogue simulators; for the digital parts, functional and – possibly – behavioral extensions should be included in the specification, representation and evaluation of circuits; for the analogue parts, not only the circuit level, but also higher levels of specification should be available.
- High-level analogue-digital interfaces must be designed and implemented to avoid the complexities and inconveniences of signals converted at a very low level of representation.
- Availability of high-level specifications could be used for automatic verification of circuit specifications; in this case, a very high level specification of requirements would be compared for consistency with a (high level) specification of the design, and all inconsistencies reported with detailed diagnoses and predictions of consequences.

7. Acknowledgements

This research was partially supported by Texas Instruments, Dallas, TX, by the Natural Sciences and Engineering Research Council of Canada through Operating Grant A8222, and by Northern Telecom Canada through University Interaction Program.

8. References


Mixed-mode, analogue–digital simulation using SPICE–like circuit ...


Biographies

Dr. Phillip E. Allen is currently the Schlumberger Professor of Electrical Engineering at Georgia Tech. His present research interest are in the area of analogue IC design methodology and CAD techniques with emphasis on modeling and simulation, the application of silicon and GaAs technology to wideband, precision, analogue signal processing applications, and the testability of analogue ICs.

Dr. Allen has taught at the University of Kansas, University of California (Santa Barbara), and Texas A & M University, and was employed at the Lawrence Livermore Laboratory from 1962 to 1965. He is the coauthor of Introduction to the Theory and Design of Active Filters (with L. Huelmsman, 1980), Switched Capacitor Circuits (with E. Sanchez-Sinencio, 1984), CMOS Analog Circuits (with D.R. Holberg, 1987), and VLSI Design Techniques for Analog and Digital Circuits (with R.L. Geiger and N.R. Strader, 1990). See also the Journal of Semiconductors, vol.4, no.2, December 1986, pp.22–32.

Dr. W. M. Zuberek is presently an Associate Professor of Computer Science at Memorial University of Newfoundland in St. John’s, Canada. His research interests include modeling and evaluation of systems, computer-aided design software, software specification and programming languages.

Dr. Zuberek has taught at Warsaw Technical University, he spent one year at the International Institute for Applied Systems Analysis in Laxemburg, Austria, worked as a postdoctoral fellow at McMaster University, Hamilton, Canada, and had a visiting appointment at Texas A & M University, College Station, TX. Since 1984 he has been with with Memmorial University of Newfoundland. He has published numerous papers in two principal areas, modeling and analysis of systems using timed Petri nets, and simulation and optimization software with particular emphasis on circuit simulation.