

8. Hybrid simulation in Alecsis

If a circuit has both analogue and digital components, the simulation is hybrid. Analogue components are time-continuous, and analogue links (`node`, `flow`, etc.) have real values. That means, for analogue components system of equations is built and solved in many time-instants. Digital components are discrete-event, i.e. they are active only in discrete time instants. That means that an event-driven simulation algorithm is used, where propagation of events through the system is traced (no system of equations is necessary). Links in discrete-event components (`signal`) usually have discrete values (usually some enumeration type). However, in Alecsis signals can have real values (type `double`), too.

In hybrid simulation two kinds of coupling has to be performed:

- time-synchronization of analogue and discrete-event simulator;
- conversion of signals for all links with hybrid aspects.

Time-synchronization is performed by Alecsis, and user has no responsibility for it. However, second aspect of coupling demands user attention. Converters of signals are automatically inserted by the simulator for all links with hybrid aspect. However, Alecsis has not built-in system of states for digital (discrete-event) simulation. Therefore, user can define his own system of states. If he does that, he have to define his own D/A and A/D converters for that system of states, and these converters are later automatically inserted by Alecsis. For that reason, this Chapter concentrates on this second aspect of analogue/digital coupling.

Alecsis knows in advance only the nature of built-in components -- they are analogue. Other models, that are described in AleC++ or are already compiled in Alecsis object-code, can be analogue, digital or hybrid -- the syntax is the same. The nature of these models, and aspect of links (`analog`, `digital` or `hybrid`) can be determined only when the whole circuit description is read (the hierarchical tree, describing the circuit hierarchy is built).

The link has `hybrid` aspect if it is used both as analogue and as digital link. It is used as analogue if it appears as unknown in the system of equations. It is used as digital if some processes are sensitive to it, or it is driven by some digital driver. Between an analogue link and a component where appropriate formal signal has

direction `in` or `inout`, Alecsis inserts A/D converter. On the other hand, Alecsis inserts D/A converter between an analogue link and a component, if the component has a driver for the appropriate signal.

8.1. Implicit converters of link aspects

Alecsis automatically inserts a converter whenever it detects a hybrid link, but it cannot determine its structure. Converter structure depends on the system of states used for digital simulation, and of desired conversion accuracy. Therefore, converters have to be defined by the user. (Of course, this does not mean that you have to determine new converters for every new hybrid problem -- if you use some standard system of logic states from the library, and appropriate standard logic gates, you normally have standard A/D and D/A converters available in the library, too.)

Converters are just a special type of modules that user defines according to the manufacturing technology and the desired accuracy of the conversion. These modules must have two formal signals, one with the analogue, the other with the digital aspect. Converter cannot accept `action` parameters, but can accept parameters through the model card. However, converter has not its own model card, it accepts model card from the inserted digital component. To enable that, appropriate model class for converter has to be the same as the model class of the inserted component, or its base class.

Converters are modules of hybrid nature, since they have:

- ◆ formal links of both aspects;
- ◆ processes with digital synchronization (sensitive to signals) and analogue, synchronized by the internal simulator signals;
- ◆ analogue components declared in the structural region.

These analogue components are representing structural, or combined structural-functional model. It is the model of the input of the digital component (in case of A/D conversion), or output of the digital component (in case of D/A conversion), as seen from the analogue part of the circuit. For instance, this can be only a capacitor marking input capacitance (for digital component in CMOS technology), or the input to a TTL circuit with a `bjt` and additional components. The number and the type of analogue elements depend on the technology and desired accuracy of conversion.

8.1.1. A/D conversion

A/D converters are inserted for every hybrid link that is connecting analogue components and the component with the appropriate formal signal having direction `in` or `inout`. A/D converter has two formal links - analogue link and digital signal. It is usually a combination of several elements modelling the input of a digital circuit; and a `process` where analogue link is compared with a series of fixed threshold, to determine state of appropriate digital signal.

This process is always synchronized using `post_moment` synchronization signal. It activates only when the analogue part of the circuit has an *accepted solution* for the present time-instant (analogue simulator reject solution in a time-instant if local truncation error is greater than the appropriate tolerance, and repeats the simulation with the shorter time-step). Using `post_moment` synchronization guarantees the validity of the analogue solution for the present time-instant.

It is already stated that the conversion inside that `process` comes down to comparison of an analogue quantity with a series of thresholds, which effectively converted a continuous quantity into discrete domain. The number of thresholds depends upon the system of states the converter is defined for. The values of thresholds can be

fixed, or depending upon parameters in case the converter has a model class. Usage of model class enables modelling of inputs in different technologies.

Conversion can be *direct* or *delayed*. In the first case, the values are converted as they come (e.g., if we have transition of a node from 0 to 5v, there is a passage through an zone in between thresholds 0 and 1 where converter will give output state 'x'). The second method assigns the new signal only after it determines whether the transition to 'x' state is a true undetermined state, or just transient phase $0 \rightarrow 1$ or $0 < -1$.

Here is an example with *direct* conversion:

```
implicit { capacitor c, C; }

typedef enum { 'x', '0', '1' } three_t;

module cmos_a2d (node analog; three_t out digital) {
    capin (analog, 0) 0.1p;
    action post_moment {
        three_t last_state='x', new_state;
        if (analog > 3.5v) new_state = '1';
        else if (analog < 1.5v) new_state = '0';
        else new_state = 'x';
        if (last_state != new_state) {
            lasr_state = new_state;
            digital <- new_state;
        }
    }
}
```

This example presents a simple A/D converter intended for the connecting a hybrid link with the input of CMOS logic gate. The new event is generated in every time-instant of analogue simulation, but only after a threshold is reached. Assignment of signals in processes `post_moment` is processed in the same time-instant as analogue simulation (keyword `now` returns the same time for processes `per_moment` and processes `post_moment`, but the `post_moment` executes after the solution is accepted, i.e. local truncation error is small enough). No delay is generated. The digital circuit will not notice the difference between signals from the converters and other signals in the digital part (although you can check the predefined signal attribute `hybrid` to see whether the signal originates from digital component, or is generated by the converter).

The previous example uses direct conversion. The signals from the output of a digital circuit will have short intervals of 'x' states in the transition period. This is a 'false alarm', although some logic simulators (e.g. HILO) have the capability for the logic circuits to generate those kinds of signals. Here is an example with *delayed* conversion:

```
module a2d (node analog; fift_t out digital) {
    capin (analog,0) 0.3p;
    action {
        process post_moment {
            fift_t last_event='x', new_event;
            double dval, last_val;

            dval = analog;
            if (dval >3.5) {
                /* rising !! */
                new_event = '1';
            }
            else if ( dval < 1.5 ) {
                /* falling */
                new_event = '0';
            }
        }
    }
}
```

```

else {
    /* is it real 'x' or just a transition? */
    if (last_event == '0' && dval < last_val
        || last_event == '1' && dval > last_val)
        new_event = 'x';
    }
    if (last_event != new_event) {
        last_event = new_event;
        digital <- new_event;
    }
    last_val = dval;
}
}
}
}

```

This converter uses not only the value of the analogue variable, but also the sign of the slope (the sign of the first derivative) of the analogue curve. If the previous state was '0', the state 'x' will not be assigned immediately when the threshold for '0' is passed. The new state is not assigned while the slope is positive, i.e. during the transition. The same is valid if the previous state was '1'. State 'x' is not assigned when the threshold for '1' is passed, while the slope is negative. When the first derivative changes the sign, state 'x' is assigned. In this way, the normal transitions from one to another state do not generate 'x' state, but only fluctuations, that are really undefined ('x') states.

Note: Note that delayed conversion does not mean that any artificial delay is generated in the process of conversion. For instance, when the analogue signal is rising, state '1' is assigned immediately after threshold for '1' is passed. Name *delayed* means that the simulator waits to study the real behaviour of the signal during the transition only.

Converters for systems with more logic states can have more voltage thresholds. Also, converters for circuits with bipolar inputs need additional analogue components for correct modelling. However, the conversion procedure does not differ.

8.1.2. D/A conversion

For every digital driver of a hybrid link, D/A conversion is necessary. As was the case with A/D converters, compiler inserts D/A converters for appropriate digital component. That means, if we have an analogue component connected to the bus, driven by more digital components, mechanism of digital resolution will not be applied. D/A converters are generated for every driver, and the analogue simulator would resolve conflict on the bus.

The component with more than one driver for hybrid signal (more than one `process` can assign to given signal) would be then resolved in different way: digital resolution is performed inside the component, and the solution is converted into the analogue domain. This is usually not what is wanted, so the simulator issues warnings in those situations. Generally, in one digital module you should avoid creation of more than one signal driver (more than one `process` that assigns to one signal).

D/A converter is actually a controlled source. The analogue part of the converter should create the illusion for the analogue subcircuit that the whole circuit consists of analogue components. Model of output of digital component usually comprises a controlled source, output resistance, and output capacitance. These components can be linear or nonlinear, depending on the desired conversion accuracy.

The transition of state on the digital driver is abrupt. However, it should not be modelled as abrupt change of controlled source in a D/A converter. An abrupt change of the parameters of analogue component can create convergence problems, and, what is even more important, is not an accurate model of real behaviour. Abrupt changes of states in digital circuits are modelling of real circuit behaviour on the higher level of abstraction. In the

analogue part, we should use more accurate modelling, on the lower level of abstraction, where all changes are continuous. For that reason, good D/A converter ought to have two processes - one that sensitive to a digital signal, and the other to model continuous transitions.

This transition of the analogue controlled source in a D/A converter contributes to a total delay of a digital signal. Therefore, if the signal has attribute `hybrid` with value 1, the delay assigned in modelling of digital circuit should be shortened for the transition time of the analogue source (so the total delay is correct). Beside that, delay due to load (see Chapter 6 on digital simulation, section on user-defined attributes of signals) should not be used, since the load is here analogue and the delay will be determined by analogue simulator. Therefore, **the delay, defined in digital processes for drivers of hybrid links, must not take into account capacitive load on the output, and is reduced for the time needed by the controlled source of the D/A converter to reach the half of the transition.**

Here is an example of D/A converter:

```
enum status { Rising, Falling, Steady };
#define NewR(state) (state=='x' ? 100k : state=='z' ? 1e9 : 1k)

#define RiseTime 10ns
#define FallTime 10ns
#define NewLevel(state) (state=='1' ? 5v : state=='x' ? 2.5v : 0v)
module cmos_d2a (four_t in digital; node analog) {
    resistor Rout;
    capacitor Cout;
    cgen Iout;
    Rout (analog, 0);
    Cout (analog, 0);
    Iout (0, analog);

    action {
        four_t old_state='0', new_state='0';
        status stat = Steady;
        double start_value=0, end_value=0, start_time=0, end_time=0;

        process (digital) {
            old_state = new_state; new_state = digital;
            if (new_state == 'z' && old_state == '0' ||
                new_state == '0' && old_state == 'z' )
                stat = Steady;
            else if (old_state == '1' ||
                    old_state=='x' && new_state != '0')
                stat = Falling;
            else stat = Rising;
            if (state != Steady) {
                start_value = NewLevel(old_state);
                end_value = NewLevel(new_state);
                start_time = now;
                end_time=start_time+(stat==Falling?FallTime:RiseTime);
            }
        }

        process per_moment {
            if (stat == Steady) {
                *Rout = NewR(new_state);
                *Iout = end_value/(*Rout);
            }
            else {
                if (now < end_time) {
```

```

        *Rout=new_value=='z'?1e9:100k;
        slope=(end_value-start_value)/end_time-start_time);
        *Iout = (start_value+now*slope)/(*Rout);
    }
    else {
        *Rout = NewR(new_state);
        *Iout = end_value / (*Rout);
        stat = Level;
    }
}
}
}
}
}
}
}

```

The concept of this D/A converter is the in the recognition of the three states, which are enumeration constants `Rising`, `Falling`, and `Steady`. The process sensitive to change of the formal signal `digital` controls the state of the converter, while the process `per_moment` controls the values of analogue components according to the converter state. Analogue components are capacitor `Cout`, resistor `Rout`, and current source `Iout`. In the `Steady` state, these analogue components have fixed values, according to the last logic state. In the other two states current sources `Iout` changes its current according to the predefined equation. Converter can be transferred into the `Rising` or `Falling` state only by the first process, that is sensitive to signal `digital`. On the other hand, converter moves into the `Steady` state when the current source `Iout` reaches the final value.

The analogue simulator can reject the solution and backtrack, in order to shorten time step. However, this model is valid even if this happens (it will not backtrack beyond `start_time`).

We use current source `Iout` in parallel with resistance `Rout`, rather than voltage source in series with resistance. The latter would demand two more equations (one for the internal node for the series connection with the output resistance, and another for the current through the voltage source).

The drawback of this D/A converter model is that the model parameters are fixed. Such parameters should be grouped into model cards, as they depend on production technology.

8.2. Converter declaration

The simulator does not know in advance which converter models are to be used for given problem. You need to define this explicitly, whenever you expect a hybrid simulation. **Converter models are declared for digital circuits.** There are three ways of declaring converters:

- for the whole circuit (if technology is unified), in the `root` module;
- at the individual `module` level;
- for particular formal signal.

8.2.1. Converter declaration for the whole circuit

Declaration of the converter for the entire circuit is in the `root` module, after the structural region.

```

root module X () {
    ...      // declarative part
    ...      // structural part
    conversion { a2d = cmos_a2d; d2a = cmos_d2a; }
    ...      // printout control, timing and options control
    ...      // functional part
}

```

You create block for declaration of modules for A/D (parameter `a2d`) and D/A (parameter `d2a`) conversion using the key word **conversion** and parentheses. The names of modules can be also given together with the names of libraries they are stored in (e.g., `lib1.cmos_a2d`), otherwise linker will search all libraries.

When forming a hierarchical tree, the information on converters will be recursively passed to all children modules. The conflict with model cards will not arise if:

- children modules do not accept model cards, or
- the model class of the converter is the same as the model class of children module, or one of its base classes.

Converter declaration for the `root module` can be masked by declaration for children modules, or with declaration for individual formal signals.

8.2.2. Converter declaration for module

You can mask the converter declaration from the `root module` with an identical declaration in the current module. The new declaration applies to all submodules of that module (i.e to its children), if not separately masked in some of them. If only one converter is declared for the module (`a2d` or `d2a`), another one is obtained from the parent module.

Converter declaration for module can be masked by declaration for children modules, or with declaration for individual formal signals.

8.2.3. Converter declaration for formal signals

The declaration of converters is allowed for individual formal signals. This declaration has the priority. The of names of conversion modules are given using following syntax:

```

module rsff(three_t in reset:cmos_a2d, set;
            three_t inout q, nq: (cmos_a2d, cmos_d2a)) {
    ...
}

```

Converters are declared after the declarator (and after an optional initializer), and apply to only one signal (here only `nq`).

You can declare only one converter for formal signals with `in` direction (this must be A/D converter) and only one converter for formal signals with `out` direction (D/A). Here, signal `reset` with direction `in` has A/D converter declared. For `inout` signals, you can list both converters (in parentheses, separated by a coma, as for signal `nq`).

These converters are created as the children of the current module, and will be able to inherit its model card (under the conditions that model class of converter is the same as model class of the module, or its base class). Formal signals that do not have this declaration will inherit converters from the current module, or its parents (signals set and q). **Local signals cannot have converters in their declaration.**

8.2.4. Organization of class hierarchy for digital model classes

Since converters do not take action parameters or model cards directly, the only way they make them dependent on parameters is using the model cards of the parents. To use the same converter for digital circuits with different functions and parameters, you should do the following:

- You need to create the base class, which is accepted by the converters. It will be a common class for a large group of digital circuits, which makes converters common, too. You can place here all parameters that have no connections with the conversion, but are also common. To make them accessible from derived classes, you need to declare these parameters as `protected` and not `private`.
- For each type of digital circuits, that demands new, different parameters, you need to develop a new model class by **deriving** it from the **base** class. To avoid problems with offset, it is important for all derived classes to have a single class as base class.

The address of the derived model card will be the address of the *base part*, and so converters will be able to find their parameters. Initialization of the *base part* of the card can be achieved by defining a separate constructor. As Chapter 4 on object-oriented programming emphasizes, during a construction of a card (being a static object) the constructors of the base class will be invoked first, and then the ones from derived classes (vice versa for destructors). However, **base preprocessors are not automatic, and should be called from the derived processor**. All parameters of the base class should have usable initial values, which makes their redefinition necessary only in the case we want to change them.

```
class common {
    protected:
        int tech; // basic technology
        double Cin, Rin; // impedance for A2D
        double Cout, Rout; // impedance for D2A
        double zero_level, unit_level, x_level; // analog levels
        double z_impedance; // turn-off impedance
        double rising_time, falling_time; // transition times
    public:
        common();
        >common();
        friend module common_a2d, common_d2a;
};

class gate : public common {
    double tplh, tphl;
    double skew;
    public:
        gate();
        >gate();
        double f_delay(four_t new_state);
};

// define base constructor
common::common() {
    tech = CMOS; Cin=0.1p; Rin=1e15; Cout=0.1p; Rout=1k;
```



```

    zero_level=0v; unit_level=5v; x_level=2.5v; z_impedance=1e9ohm;
    rising_time = falling_time = 10ns;
}

// define base preprocessor
common::>common() { ... /*anything*/ }

// define derived constructor
gate :: gate () { tplh = tphl = 10ns; }
// define derived preprocessor
gate :: >gate () { common::>common(); .../* anything ELSE */ }

// define one model card
gate :: mod1 {
    tplh = 3.4ns; tphl = 5.6ns;
    common::rising_time = 2ns; falling_time=3ns; tech = CMOS;
}

```

The previous example defines class `common`, which stores common, mainly conversion data for a series of digital circuits. By declaring converters `common_a2d` and `common_d2a` as friends of the class `common` you enable them to access the private parameters. Setting those parameters in the constructor enables correct conversion, even if they are not set in the model card.

You can use the derived class `gate` for a larger number of standard logic gates, which use parameters `tplh`, `tphl`, and `skew` to model delay. Base class constructor is called before the derived class constructor automatically, while you have to call the base preprocessor explicitly from the derived class preprocessor.

Model card `mod1` of class `gate` can set parameters of both the base and the derived classes. You can access base parameters with or without the operator of access resolution (`::`).

This way of modelling allows modelling of new groups of digital circuits 'in layers'. Notice that every change in the base class automatically means changes in derived classes, according to the rules of object-oriented programming.

8.3. An example of hybrid circuit simulation

We left an example of simulation of parallel AD converter for the end of this Manual. Note that we are not speaking here about module for conversion of link aspects, but about the circuit for conversion of analogue signal into digital one. The ADC consists of comparators, resistance network and combinational logic (Figure 8.1.).

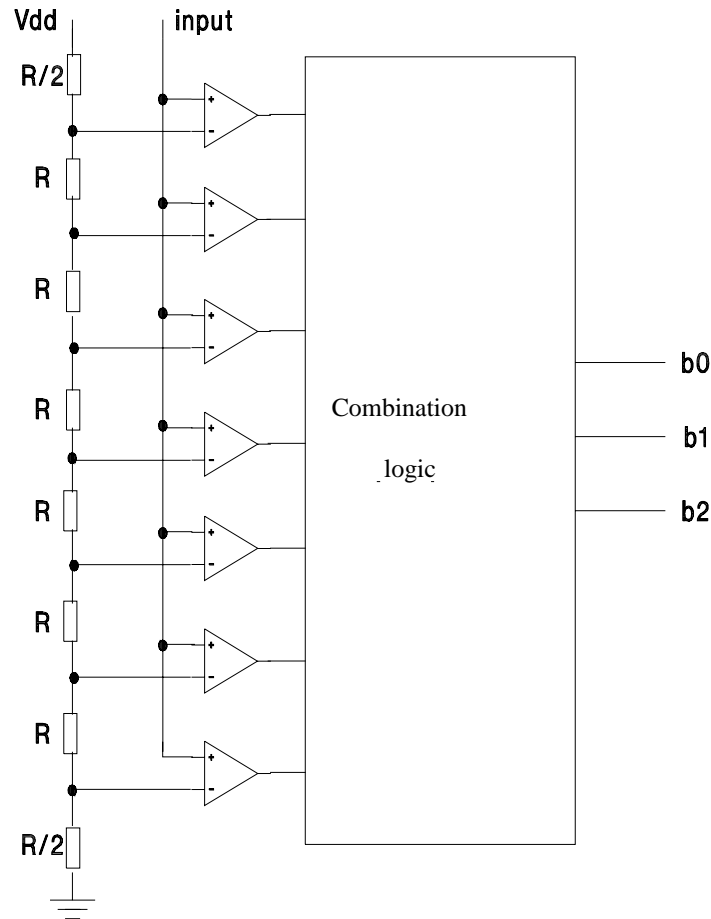


Figure 8.1. Parallel A/D converter as an example of hybrid circuit simulation

The circuit consists of seven CMOS opamps, which are used as comparators. Comparison etalons come from the serial resistance network. With the increase of input voltage, the outputs of comparators, one by one, change from the lower voltage supply level v_{ss} to the upper voltage supply level v_{dd} , that is from 0 to 5V. The outputs of the comparators are directly fed to combinational network. The output of this digital network is a three-digit number. Input voltage of 0V corresponds to output 000, and 5V to 111. Every opamp has 15 MOS transistors, and is in cascade OTA configuration. Combinational network is modelled functionally, using overloaded logic operators. We have used libraries `opamp` (with amplifiers), `op15` (program support for system with 15 logic states) and `alec` (standard Alecsis library).

The nodes where analogue circuit and digital circuit meet are opamp outputs. These nodes are `hybrid`, and implicit insertion of converter of link aspects is performed there.

The code in full follows:

```
#include <alec.h>
spice {
#include "omos.mod"
}
library opamp, op15, alec;
#include "ss15.h"
#include "opamp.h"
/*****
module comp_line ( node input; node line[], udd, uss) {
    module Ota opamp;
    resistor r, rup, rdown;
    node joint[auto];
```

```

action structural (int resolution) {
    int i;
    node joint(resolution);

    if (resolution < 2)
        warning("illegal A2D converter resolution", 1);

    /* clone edge resistors */
    clone rup (udd, joint [resolution-1]) Rvalue/2;
    clone rdown (uss, joint [0]) Rvalue/2;

    for (i=0; i<resolution; i++) {
        clone opamp [i] (input, joint [i], line [i], udd, uss);
        if (i<resolution-1)
            clone r[i] (joint[i], joint[i+1]) Rvalue;
    }
}

/*****/
module comb_logic4 (fift_t in line[7]; fift_t out b[3]) {
    action (double delay = 10ns) {
        process (line) {
            three_t l0, l1, l2, l3, l4, l5, l6;
            three_t l23, all, b0, b1;

            l0 = Con15to3[line[0]];
            l1 = Con15to3[line[1]];
            l2 = Con15to3[line[2]];
            l3 = Con15to3[line[3]];
            l4 = Con15to3[line[4]];
            l5 = Con15to3[line[5]];
            l6 = Con15to3[line[6]];
            all = l0 & l1 & l2 & l3 & l4 & l5 & l6;
            l23 = l2 ^ l3;
            b1 = (l5 ^ l6) | l23 | (l1 ^ l2) | all;
            b0 = (l4 ^ l5) | l23 | (l0 ^ l1) | all;
            b[0] <- tech_tab[CMOS][Con3to15[b0]] after delay;
            b[1] <- tech_tab[CMOS][Con3to15[b1]] after delay;
            b[2] <- tech_tab[CMOS][Con3to15[l3]] after delay;
        }
    }
}

/*****/
#define Period 100us
#define Resolution 7
#define VDD 5v
root parallel () {
    vpwl vin;
    comp_line comparator;
    comb_logic4 combinatorial_logic;
    vgen vdd;
    fift_t line[Resolution];
    fift_t b[3];

    comparator (input, line, udd, 0) resolution = Resolution;
}

```

```

combinatorial_logic (line, b);
vin (input, 0) { 0,0v; Period, VDD; };
vdd (udd, 0) VDD;

conversion { a2d = "a2d"; }
plot      { node input; node line; signal fift_t b; }
timing    { tstop = Period; a_step = Period/100; }
options  { dcon=1; maxiter = 20; maxdump = 5; }
}

```

The results of hybrid simulation of this circuit are given in Figure 8.2. We have obtained correct conversion. You can notice the shift in the average signal value at the input of the opamp (increase in voltage on the '-' input), which changes its characteristics (reduces the slope), and that introduces distortion into the converter characteristics. The whole circuit contains 105 MOS transistors. To solve it, Alecsis forms a system of 103 equations, seven post_moment processes, and one digital process (for combinational logic).

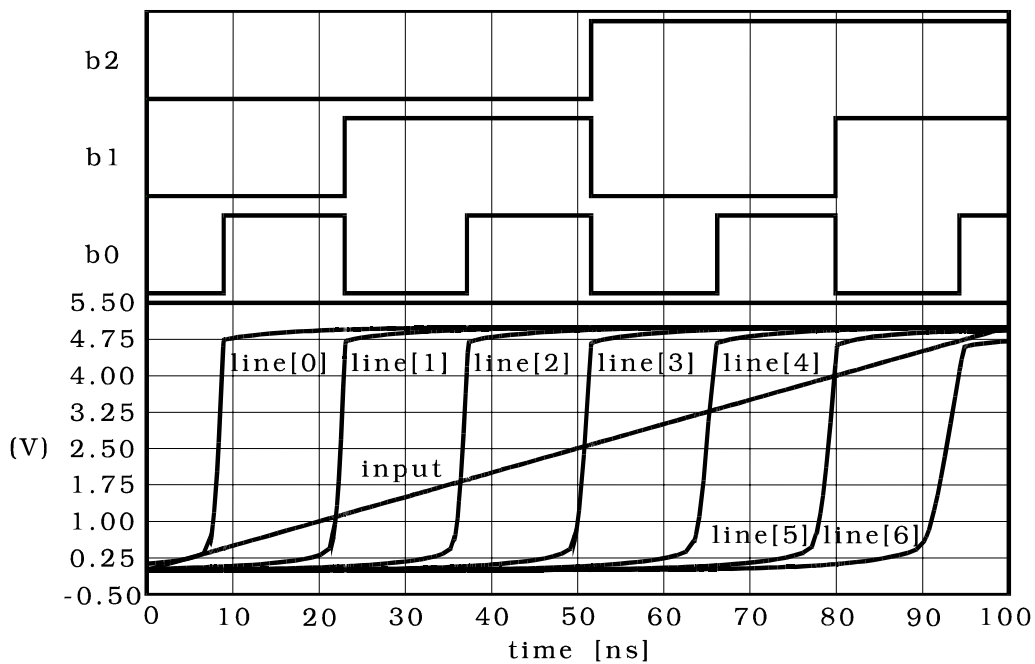


Figure 8.2. Simulation results of parallel A/D converter in CMOS technology.