

# ROMantic: Generation and Optimization of Quasi Delay-Insensitive Read-Only Memories

Mika Nyström, Elaine Ou, Alain J. Martin  
Department of Computer Science  
California Institute of Technology  
Pasadena CA 91125, USA

## Abstract

We describe the capabilities of and algorithms used in ROMantic, a tool for generating asynchronous read-only memory. ROMantic generates ROMs that are optimized for either power or speed. These ROMs are quasi delay-insensitive, and generated from an input file consisting of a value table. Although the algorithms used are based on previously known approaches, we present several enhancements particular to the improvement of asynchronous memory design. We present the measurements of a large set of ROMs of variable sizes in terms of performance, sizing, timing, and energy consumption. This allows future comparisons of asynchronous-layout generation tools and provides a benchmark for memory sizes typical of today's industrial designs.

ROMantic is capable of targeting a broad range of asynchronous applications. It has been used in the Lutonium project, which is an asynchronous version of the 80C51 microcontroller; and it should be useful for many types of asynchronous memory and memory architecture research.

## I. INTRODUCTION

The asynchronous VLSI group at the California Institute of Technology is currently involved in the design of an Intel-8051-architecture microcontroller, the *Lutonium* [12]. This project was begun in 2001 to provide evidence of the speed and energy advantages of asynchronous design. One of the chief obstacles encountered in the hardware design was that the specifications were invented with little or no thought to the implementation. One of the simplest ways of dealing with such uncooperative specifications is to use a uniform implementation technique that is guaranteed to be correct by construction. A well-known example of this is to encode or compile the specification more or less directly as data in a read-only memory (ROM).

ROMs are dense, have dependable, easy-to-predict timing, and, with the development of automatic optimization and generation software-tools, are easy to design and maintain. In this paper, we shall examine the algorithms and circuit techniques used in the design and implementation of the *ROMantic* ROM-generator program.

ROMantic is a software tool that takes an input file in *value table* format (see section II-A) and produces a ROM layout in *.mag* format, the file format used internally by the Magic layout editor [13]. ROMantic generates quasi delay-insensitive (QDI) [5] asynchronous ROMs that are entirely customized. These ROMs perform all the necessary communications to function in an asynchronous environment, and can meet high throughput requirements without sacrificing the low-power advantage of asynchronous circuits.

Generally speaking, the proper performance metric to optimize a CMOS circuit for is energy times delay squared,  $Et^2$  [16], [11]. However,  $Et^2$  does not compose easily unless we are allowed to operate different parts of a system at different supply voltages. Therefore we shall find it useful to be able to generate ROMs with different tradeoffs between speed and energy consumption. ROMantic permits the designer to choose between two distinct speed/energy tradeoffs by varying the amount of concurrency in the ROM. As a result, ROMs can be optimized for speed, or energy, which permits the designer to consider using ROMs in situations where they would formerly have been unacceptable.

The organization of this paper is as follows: Section II describes some of the features of ROMantic and the range of optimization parameters that can be used to customize the generated ROM layouts. Section III deals with architectural issues. Section IV describes the software architecture of ROMantic, and Section V presents the performance results and outlines some future enhancements that are planned for ROMantic.

## II. OVERVIEW OF ROMANTIC

The ROMantic program takes its input from a file in *value table* format, which is similar to the truth-table format used by the Espresso program [17]. While the ultimate goal of ROMantic is the generating of Magic layout, it also generates several types of higher-level descriptions that are useful for simulation. ROMantic generates ROM output in the following four formats:

- m33 - Modula-3 source code that interfaces with Karl Papadantonakis's M3-3 library for simulating communicating hardware processes (CHPs) in Modula-3.
- cast - CAST (Caltech Asynchronous Tools) description of CMOS production rules.
- kast - KAST (Modula-3 version of CAST) description of CMOS production rules.

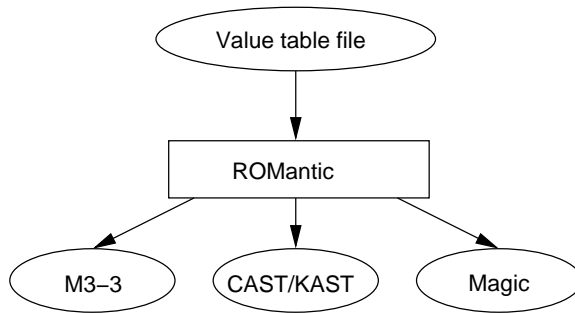


Fig. 1. ROMantic program flow.

- magic - Ready-to-fabricate layout.

Figure 1 outlines the ROMantic CAD flow.

The four formats describe the circuits at different levels of detail. M33 is a library developed at Caltech that uses Modula-3 threads to simulate asynchronous systems of processes; this is used for fast functional simulation, but it is also capable of rudimentary timing simulation, because M33 includes a timing model that takes into account compute, acknowledge, and internal process delays. CAST and KAST are two languages, also developed at Caltech, for describing large hardware systems hierarchically; CAST and KAST are mainly used for describing hierarchies of processes at the production-rule system (PRS) level. Finally, the generated layout can be fabricated.

Each of the levels of description has associated simulation tools. The M33 system is its own simulator; CAST and KAST produce flattened production-rule sets that can be simulated by a PRS simulator such as the Caltech group's `prsim` or `csim`; the layout can be extracted and simulated with a switch-level simulator such as `cosmos` [1] or with a circuit simulator such as SPICE.

#### A. The value table format

The input to ROMantic consists of a two-level description of a function. This is described as a character matrix with keywords embedded in the input to specify the size of the matrix and the logical format of the input function.

Each input and output of the generated ROM is a 1-of- $n$  (one-hot) asynchronous channel, which is implemented as  $n + 1$  wires:  $n$  wires of data, one for each possible value, and one wire for the acknowledge (ROMantic generates ROMs that use the inverted acknowledge, which we call the *enable*, following Lines [3]).

An example value-table file is shown in figure 2. The entries in this file have the following meaning:

- |                     |  |
|---------------------|--|
| • .i 4              | The ROM has four inputs.   |
| • .di a 3           | Input a is a 1-of-3 channel.   |
| • .o 4              | The ROM has four outputs.  |
| • .do x 2           | Output x is a 1-of-2 channel.  |
| • - 0 1 : 1 1 1 0 0 | If the ROM receives a zero on b and a one on c, then it will produce a one on x, a one on y, etc., regardless of what it receives on a. (It will, however, carry out a four-phase handshake on a and ignore the value received.) |
| • 1 1 1 : never     | This input will never occur.   |
| • .e                | This marks the end of the value-table file.  |

A value table is required after the channel declarations. Each position in the input sequence corresponds to an input variable where a number value implies the corresponding input channel encoding, “-” means that the variable is a don’t-care, i.e., that the ROM should perform a handshake on the input but that the value received is then not used, and “X” implies an unused input, i.e., that the ROM should not perform a handshake on that input channel when the other inputs have the indicated values (not currently implemented). When the inputs have arrived, the ROM will select the entry in the value table whose input sequence matches the input values and generate the values on the outputs listed in the output sequence.

It is an error for there to be more than one matching input sequence during operation. Similarly, if no matching input sequence is found, the ROM will deadlock when that input occurs. The designer may specify that certain inputs will never occur by indicating this knowledge with the special string `never`, as shown in figure 2. Also, ROMantic will issue an error message if there are any unspecified input conditions, or if there is more than one matching input sequence for a given input string.

```

# Input decls:
.i 4
.di a 3
.di b 4
.di c 2
# Output decls:
.o 4
.do x 2
.do y 8
.do z 3
# Value table:
- 0 1 : 1 1 1 0 0
- 0 0 : 1 2 2 0 1
0 1 1 : 1 1 1 0 1
1 1 1 : never
2 1 1 : 1 1 1 1 0
- 1 0 : 1 1 1 1 1
- 2 - : 1 1 1 1 0
- 3 - : 1 1 1 1 1
.e

```

Fig. 2. Example value-table file `test1.vt`.

### B. Output options

The following options are recognized:

- `feet` - use a foot transistor (see section III-H) in the ROM portion of the layout; the decode and ROM portions of the layout are treated as two half-buffers
- `nofeet` - sequentialize transitions so that no foot transistor is needed in the ROM portion of the layout

## III. MEMORY ARCHITECTURE AND CIRCUIT TECHNIQUES

All circuits used in the ROMs were designed using the asynchronous synthesis method described by Martin [6].

### A. General Pipeline Structure

A single ROM access consists of four steps: receive the input; decode the input; look it up in the ROM; send the output. Each one of the steps is implemented by a separate piece of datapath hardware, and the pieces are controlled by a “control box” that implements the sequencing. Figure 3 illustrates this decomposition.

The lowest-power ROM design requires that these four steps be executed sequentially; however, different techniques requiring higher power-distribution can be used to implement concurrency in these four steps and reduce the transition counts by half. The hardware used in the datapath is the same in either case (the only thing that changes is the grounding of the ROM array; see section III-H), but the control box varies for the two kinds of ROMs.

### B. Handshaking Protocols

The communication processes used to implement the Lutonium transfer data through channels implemented with a four-phase handshake protocol, as described by Martin [8]. The data is encoded using 1-of- $n$  codes.

A four-phase handshake protocol is characterized by the fact that the data rails alternate between a neutral state that doesn’t represent a valid encoding of a data value (all data rails are low), and a valid state that does represent a valid encoding of a data value (a single data rail is high). During a channel communication, the sending process waits for an *enable* signal from the receiving process, and then sets the data rails to a valid value. The receiver lowers the *enable* signal once the data has been received, and the sender resets the data rails to a neutral value when it receives the lowered *enable* signal. The receiving process once again raises the *enable* signal for the next transmission once the data rails are neutral.

Consider a simple buffer stage that receives one bit of data  $x$ , encoded as a 1-of-2 code, on channel  $L$ , and sends it on channel  $R$ :

```
*[ L?x; R!x ]
```

We implement the buffer stage by expanding the send and receive handshakes:

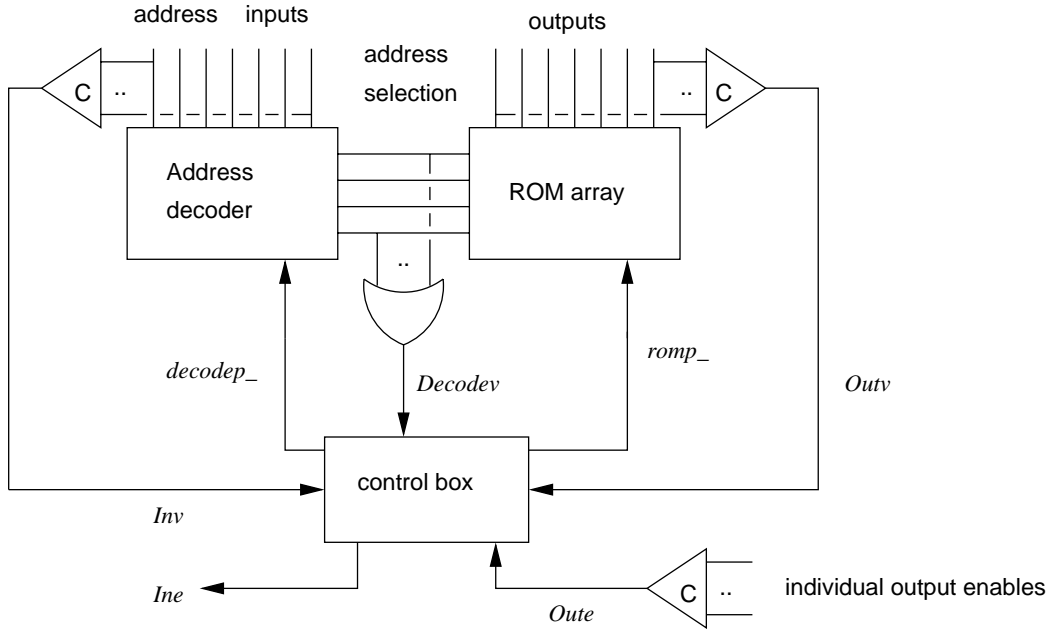


Fig. 3. Overview of ROM structure.

$$* [ [L^0 \rightarrow x^0 \uparrow \square L^1 \rightarrow x^1 \uparrow]; L^e \downarrow; [\neg L^0 \wedge \neg L^1]; L^e \uparrow; \\ [x^0 \rightarrow R^0 \uparrow \square x^1 \rightarrow R^1 \uparrow]; [\neg R^e]; R^0 \downarrow, R^1 \downarrow; [R^e] ]$$

This handshaking expansion can be reshuffled by rearranging the non data-dependent portions of the communication to improve speed and size. The reshuffling used by ROMantic is called the precharge-logic half-buffer [3].

$$PCHB = * [ [R^e]; [L^0 \rightarrow R^0 \uparrow \square L^1 \rightarrow R^1 \uparrow]; L^e \downarrow; \\ [\neg R^e]; R^0 \downarrow, R^1 \downarrow; [\neg L^0 \wedge \neg L^1]; L^e \uparrow ]$$

### C. Address decoding

One way of implementing a ROM is to make it a simple buffer stage that computes a “complicated” function:

$$* [ In?X; Out!Z ] \\ f : X \rightarrow Z \\ X : x_0 \dots x_{n-1} \\ Z : z_0 \dots z_{m-1}$$

This buffer stage receives multiple tokens of data on input channels, then computes the function, and sends the data on output channels:

$$* [ In_0?x_0, In_1?x_1, In_2?x_2 \dots ; \\ Out_0!f_0(x_0, x_1, x_2 \dots), Out_1!f_1(x_0, x_1, x_2 \dots), Out_2!f_2(x_0, x_1, x_2 \dots) \dots ]$$

In implementing the ROM, a memory  $M$  is initialized such that the function  $f$  looks up values for  $X$ :

$$M[x_i] = f(x_i)$$

An intermediate value  $d$  helps implement the functions  $f_0 \dots f_n$ . This intermediate value is computed and sent by the *address decoder*, which transforms the memory address into a 1-of- $n$  code, and received by the *ROM array*, which transforms the 1-of- $n$  decoded address into the desired set of output values (see figure 3).

$$* [ In?X; Out!M[X] ] = \\ * [ In?X; Decode!d(X) ] \parallel * [ Decode?d; R!M[d] ]$$

In summary, the function translates a series of input values to a decode value, and uses the decode value to look up the output values in the memory. With these operations implemented as two separate functions, the single buffer can be broken up into three communications: one to receive the inputs from the environment, one between the decode values and the ROM output values, and one to send the outputs. Then the ROM can be thought of as two separate blocks: a *decode* block for translating the input values,

and a ROM block for looking up the output values and sending the outputs. The communication between the decode block and the ROM block can be represented as a single channel encoded as a 1-of- $n$ , where  $n$  is the number of rows in the ROM table.

#### D. Completion Trees

There are three separate sets of channels in the ROMs. First, the input channels carry the ROM addresses; these  $n$  channels are called  $In_0$  through  $In_{n-1}$ . Secondly, the ROM addresses are decoded into a single 1-of- $n$  code called *Decode*. And finally, the outputs are produced on the  $m$  channels  $Out_0$  through  $Out_{m-1}$ .

We shall use  $In_i^v$  to represent the validity of an input channel  $In_i$ ; thus the wait for the validity of the  $i$ th input  $In_i$ ,  $[In_i^v]$ , becomes the following:

$$[In_i^v] = [In_i^0 \vee In_i^1 \vee \dots \vee In_i^n]$$

Then we let  $[In^v]$  represent the simultaneous validity of all input channels:

$$[In^v] = [In_0^v \wedge In_1^v \wedge \dots \wedge In_{n-1}^v]$$

Similarly,  $Out^v$  represents the validity of all output channels, and  $Decode^v$  represents the validity of the decode channel.

The circuit for the production rules shown above would require  $n$  p-transistors and  $n$  n-transistors in series to implement. A much more efficient implementation to check for simultaneous validity is to use a completion tree. A completion tree groups the inputs together so that a smaller number of transistors are connected in series. This would be equivalent to the following modification:

$$In^v = [(In_0^v \wedge In_1^v \wedge In_2^v) \wedge \dots \wedge (In_{n-2}^v \wedge In_{n-1}^v \wedge In_n^v)]$$

The ROM raises the input enable signals of all the input channels at once, so  $In^e$  controls all input enable signals. We cannot guarantee that all output enable signals will be set or reset by the environment concurrently, so a completion tree generates  $Out^e$  from the enables of the individual output channels; it can be represented as follows:

$$\begin{aligned} Out_0^e \wedge Out_1^e \wedge \dots \wedge Out_{m-1}^e &\mapsto Out^e \uparrow \\ \neg Out_0^e \wedge \neg Out_1^e \wedge \dots \wedge \neg Out_{m-1}^e &\mapsto Out^e \downarrow \end{aligned}$$

#### E. Delay-Insensitivity Issues

The completion trees for  $In^v$  and  $Out^v$  consist of C-elements and OR-gates (the latter being implemented with NAND or NOR gates); these trees are internally entirely delay-insensitive. Similarly, the completion tree for  $Decode^v$  consists entirely of OR-gates. Because of the isochronic forks [5] at the inputs of these trees, the entire ROM is quasi delay-insensitive.

#### F. Implementation of Control Section and Precharge Signals

Both the decode block and the ROM block need a means of lowering all of their output wires and bringing their channel values back down to neutral; the inputs also need to be acknowledged, and all these activities have to be sequenced properly with respect to each other and with respect to the transitions on the output enables. This sequencing is handled by the ‘‘control box’’ in figure 3.

The control box generates the input enable signal and the two precharge signals, *decodep<sub>-</sub>* and *romp<sub>-</sub>*, as shown in the circuit diagram in figure 3.

One way of implementing the ROM is as follows:

$$\begin{aligned} * [ & [In^v \wedge Decode^v \wedge Out^v \wedge \neg Out^e]; In^e \downarrow; \\ & [\neg In^v]; Decodep_{-} \downarrow; [\neg Decode^v]; Romp_{-} \downarrow; \\ & [\neg Out^v \wedge Out^e]; Decodep_{+} \uparrow; Romp_{+} \uparrow; In^e \uparrow ] \end{aligned}$$

This implementation guarantees that the *Decode* channel is neutral when *Romp<sub>-</sub>* is active (the ROM array precharges when *Romp<sub>-</sub>* is low).

#### G. Increasing Slack and Concurrency

The *slack* of a communication channel in an asynchronous system refers to the amount of buffering present in the channel. The slack of a channel indicates the maximum difference possible between the number of communication actions on the channel. Slack can be increased in the sequence of actions in the above handshaking expansion by reshuffling the expansion as described by Martin [9].

By reshuffling the above buffer and implementing the decode and ROM as separate buffers, we can increase the slack on the communication channel by one and thus add pipelining to the circuit:

$$* [L?x; R!d(x)] \quad || \quad * [R?y; Q!g(y)]$$

This can be compiled into the following handshaking expansions:

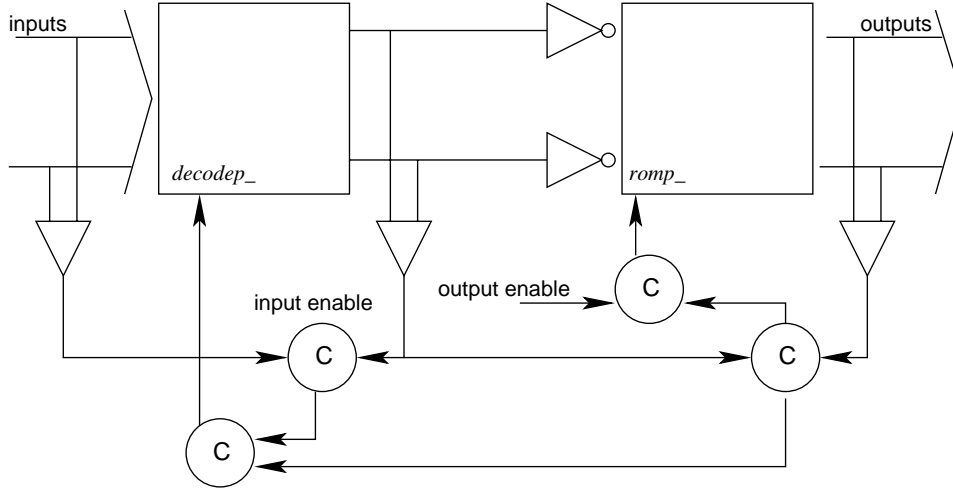
$$\begin{aligned}
\text{DECODE} = & \\
& *[[In^v \wedge Decode^v]; In^e \downarrow; [\neg x^e]; Decode_p \downarrow; \\
& [\neg In^v \wedge \neg Decode^v]; In^e \uparrow; [x^e]; Decode_p \uparrow ] \\
\\
\text{ROM} = & \\
& *[[Decode^v \wedge Out^v]; x^e \downarrow; [\neg Out^e]; Romp\_ \downarrow; \\
& [\neg Decode^v \wedge \neg Out^v]; x^e \uparrow; [Out^e]; Romp\_ \uparrow ]
\end{aligned}$$


Fig. 4. Double half-buffer implementation of ROM control circuitry.

This implementation *does not* guarantee that the *Decode* channel is neutral when *Romp\_* is active (the ROM array precharges when *Romp\_* is low).

#### H. “Foot” Transistors

Because the two-buffer implementation with the separate control processes *DECODE* and *ROM* has the unfortunate property that the *Decode* channel may remain valid while the *Romp\_* bitline precharge signal is active, a single n-transistor cannot be used as a pulldown in the ROM array since it would cause interference. A “foot” transistor must be added to the ROM array, as shown in figure 5.

Adding a separate foot transistor to every bitline pulldown would increase the area of the ROM substantially, which in turn would reduce performance because of added capacitance on the now-longer bitlines. Instead of doing this, the bitline transistors have been converted to pass gates and use a virtual ground (called *bigromp*) in the pulldowns that is set high when the ROM array is being precharged. A *single* virtual ground is used for the entire ROM array; while this would not normally be permissible in a precharge half-buffer, it is possible here because only a single decode signal can be asserted at any given time. Hence, it is impossible for sneak paths to form. By driving *bigromp* with an inverter we also avoid charge-sharing problems because *bigromp* is held high until the decode wire is allowed to trigger the bitline transistors. *bigromp* is also fed into the completion tree for *Out<sup>v</sup>* to ensure that the system remains quasi delay-insensitive; if *bigromp* were not fed into the completion tree, the upward transitions on *bigromp* would not be acknowledged.

## IV. SOFTWARE ARCHITECTURE

ROMantic is written entirely in the Modula-3 programming language [15]. The program is structured as four largely separate pieces: a library for manipulating hand-drawn layout, a library for representing generic completion trees, a library for instantiating the generic completion trees as layout, and finally a set of driver modules that parse the input files, call the libraries, and generate the production rules and M3-3 code.

The M3-3 generator and production-rule generators produce code that simulates with as accurate timing information as is reasonable to achieve in each of the description languages. Because ROMantic includes pass gates in the layout when producing the high-speed ROMs, the production rules do not match the layout exactly, but the timing information is still accurate. The production rules that are produced for the low-energy ROMs match the layout exactly.

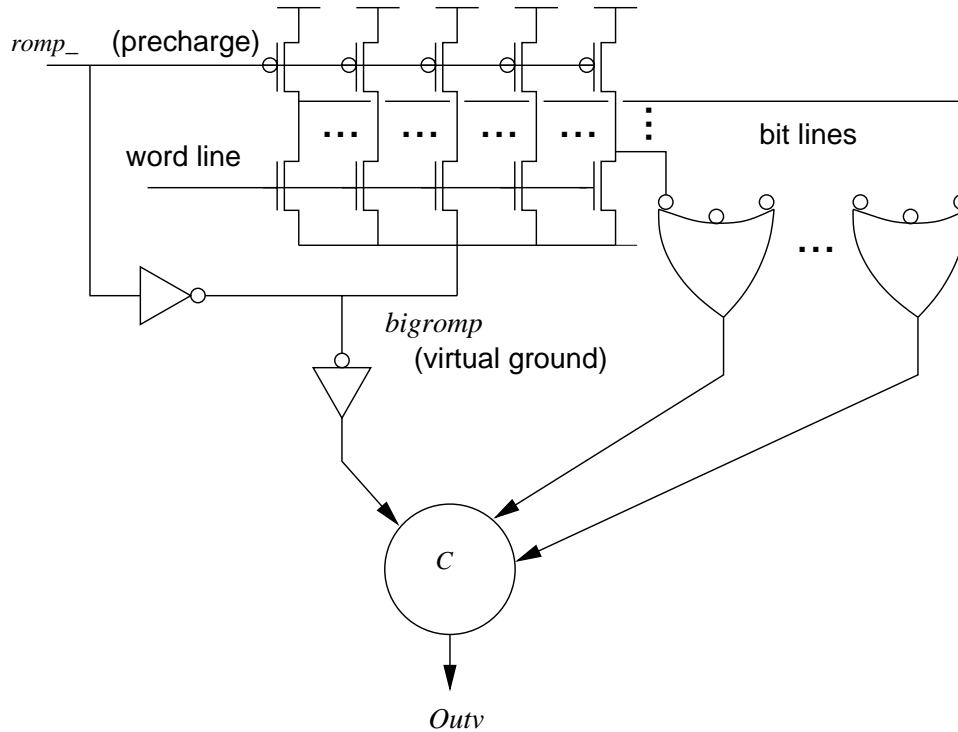


Fig. 5. ROM foot transistor circuit

### A. Layout generation

ROMantic generates layout in the format used by the Magic layout editor. The library that it uses takes hand-drawn layout in a format called *tiles* and uses these tiles to assemble the entire ROM.

The tile format is inspired by Berkeley's *mquilt* program: a tile is described as a layout cell with a single rectangular label *tile* in it. However, the tile library that the authors have developed is more sophisticated than *mquilt*'s: it is possible to have an entirely arbitrary number of different tiles, and the tiles can be transformed and manipulated in a variety of ways—it is even possible to add arbitrary new layout under program control to them. Also, the tiles are assembled recursively: a tile is joined to another tile to make a larger tile; the finished ROM is simply a large tile that is finally written out as a Magic file. Currently, ROMantic's layout library consists of 97 different tiles, ranging from space tiles to the control sequencers described in section III-F.

An example tile is shown in figure 6. This tile holds a single transistor used in the ROM array; the presence of this tile means that when the decode wire (word line) goes high, the output wire (bit line) will go low. The label marked *tile* defines the alignment of the tile; the *tile* label is  $8\lambda \times 8\lambda$ , which corresponds to the packing density of the ROM transistors. The label marked *x@* is a dummy label that ROMantic replaces with a label denoting the name of the output that is actually connected to the bit line; this helps debugging. The ground node marked in the figure is either the virtual ground described in section III-H (for the high-speed ROMs) or simply the global ground node *GND* (for the low-energy ROMs).

The most congested part of the layout is the staticizers (bleeders) on all the state-holding nodes (i.e., the bit lines and selector lines). An example of the staticizer layout is shown in figure 7. The staticizers themselves are laid out on a  $16\lambda$  pitch; by staggering two columns, an overall pitch of  $8\lambda$  is possible, matching the pitch of the ROM array and decoder array. The two staticizer columns and the output inverters are a total of  $215\lambda$  wide. The width of this circuitry means that very small ROMs are relatively inefficient in terms of layout; for larger ROMs, the overhead is less objectionable. As an example, one of the two ROMs used in the decoder of the Lutonium fits 3,611 transistors on roughly  $1,800,000\lambda^2$  of area using MOSIS's design rules for the  $0.18\text{-}\mu\text{m}$  TSMC process.

### B. Performance estimates

The different versions of the control box gives the designer the opportunity to trade speed for energy. A simple and useful first-order approximation to the speed is the number of CMOS transitions in a cycle of operation. In many cases, the overall speed of the ROM is constrained by global system considerations. For instance, in the Lutonium processor, the designers fixed a global cycle-time constraint of 22 CMOS transitions per cycle; hence the decode ROMs have to operate at no more than 22 CMOS

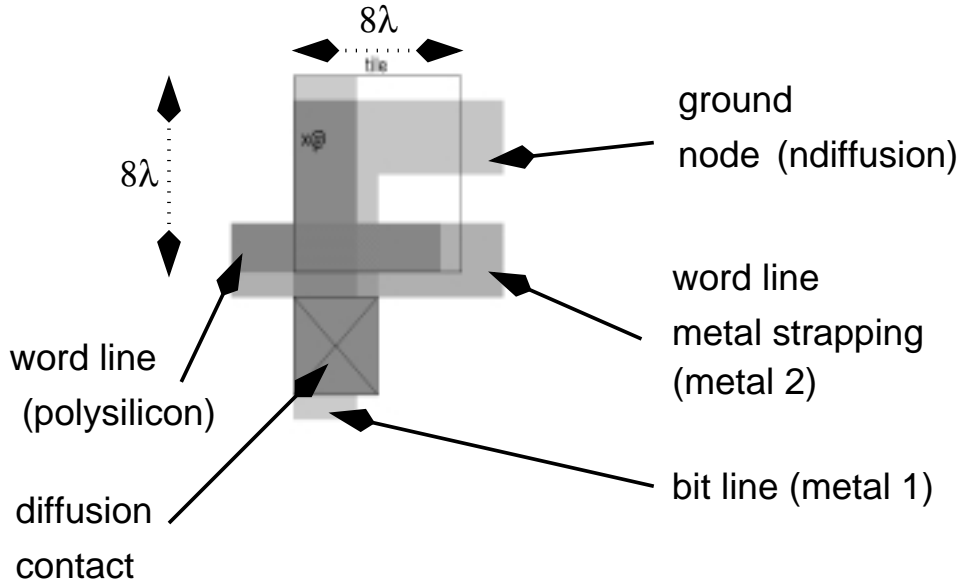


Fig. 6. Example ROM tile.

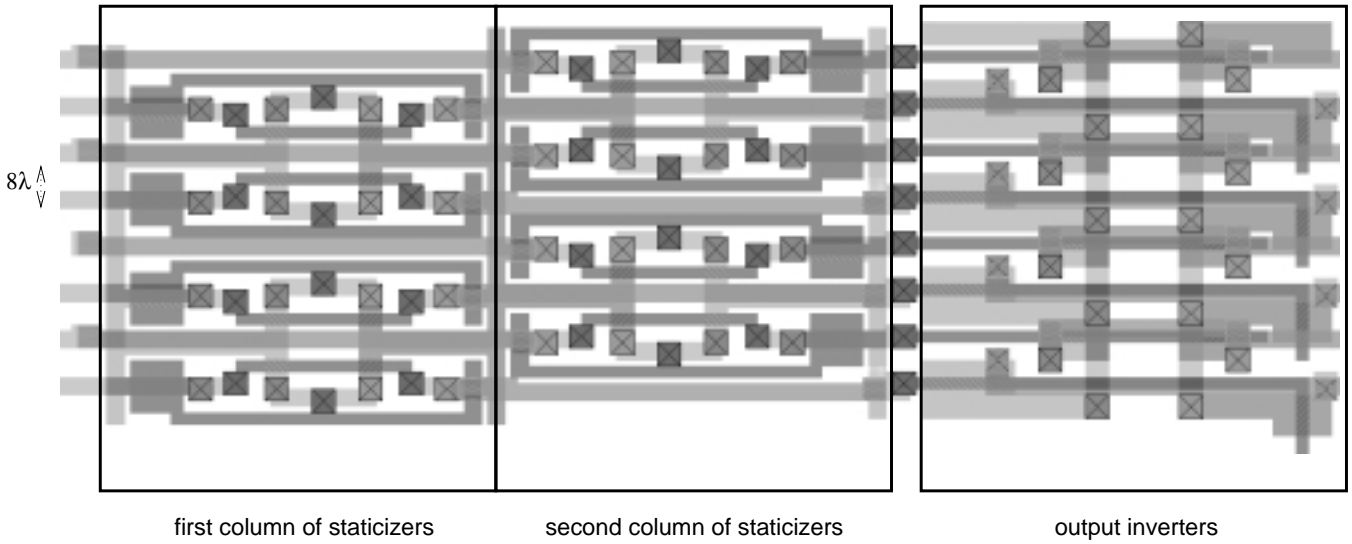


Fig. 7. Packing of ROM staticizers and output inverters on  $8\lambda$  pitch.

transitions per cycle. Most of the transitions in the cycle stem from the completion trees (i.e., the performance depends on the size of the ROM that is being generated). In order to help the designer decide between the different control implementations, ROMantic computes and prints out a performance estimate in terms of the number of transitions per cycle.

We assume that the ROM generated by ROMantic is surrounded by simple half-buffers that acknowledge their inputs (i.e., change their enable lines) in three transitions and respond to a changed enable signal in two transitions. We define the depth of the input, decode, output enable, and output completion trees as  $\Delta_i$ ,  $\Delta_d$ ,  $\Delta_e$ , and  $\Delta_o$ . With these definitions, the cycle time of the low-energy ROM (without foot transistor)  $\Xi_{le}$  is

$$\Xi_{le} = 19 + \max(\Delta_i, \Delta_o + 4, \Delta_d + 2, \Delta_e + 5) + \Delta_i + \Delta_d + \max(\Delta_o, \Delta_e)$$

and the cycle time of the high-speed ROM  $\Xi_{hs}$  is

$$\Xi_{hs} = \max(2\Delta_o + 9, \Delta_e + \Delta_o + 10, \Delta_i + \Delta_d + 5, 2\Delta_d + 9, \Delta_d + \Delta_o + 9, 2\Delta_i + 1) .$$

An even higher-level estimate of the performance can be made by considering the sizes of the input and output channels. The depth of an OR tree is approximately  $\Delta_V(k) = \lceil \log_{\sqrt{m_p m_n}} k \rceil$ , where  $k$  is the number of inputs and  $m_p$  and  $m_n$  are the maximum number of pMOS and nMOS transistors permitted in series. The depth of a C tree is approximately  $\Delta_C(k) = \lceil \log_{\min(m_p, m_n)} k \rceil$ , where  $k$  is the number of inputs. In the *Lutonium*,  $m_p = 3$ , and  $m_n = 6$ . Because ROMantic does not do any optimization across different trees, we can approximate the  $\Delta$ s as

$$\begin{aligned}\Delta_i &\approx \Delta_C(n) + \max(\Delta_V(w_0), \dots, \Delta_V(w_{n-1})) \\ \Delta_d &\approx \Delta_V(\rho) \\ \Delta_o &\approx \Delta_C(m) + \max(\Delta_V(v_0), \dots, \Delta_V(v_{m-1})) \\ \Delta_e &\approx \Delta_C(m),\end{aligned}$$

where we have introduced the following notation:  $w_i$  is the width of the  $i$ th input channel,  $v_j$  is the width of the  $j$ th output channel, and  $\rho$  is the number of entries (rows) in the ROM.

## V. PERFORMANCE RESULTS

In CMOS, designers can trade off energy and delay through voltage adaptation by varying the power supply voltage  $V_{dd}$ . Because ROMantic permits the designer to choose between two distinct speed/energy tradeoffs by varying the amount of concurrency in the ROM, two performance tables illustrate the results of the different implementations. The authors have simulated layout generated by ROMantic using parameters for the 0.18- $\mu\text{m}$  CMOS process from TSMC via MOSIS. The following results were gathered using Caltech's Aspic circuit simulator (an efficient circuit simulator for CMOS circuits that implements Berkeley's BSIM2 and BSIM3 transistor models).

Inputs	Outputs	Rom size	Energy	Speed	Transition Count
4	9	113	70 pJ	215 MHz	22
2	2	16	8 pJ	341 MHz	16
4	4	256	49 pJ	164 MHz	16
6	8	100	33 pJ	197 MHz	20
4	4	1	10 pJ	425 MHz	14
8	8	1	18 pJ	344 MHz	14

TABLE I  
OPTIMIZED FOR SPEED

Inputs	Outputs	Rom size	Energy	Speed	Transition Count
4	9	113	54 pJ	155 MHz	40
2	2	16	6 pJ	214 MHz	36
4	4	256	38 pJ	111 MHz	36
6	8	100	25 pJ	134 MHz	38
4	4	1	7 pJ	386 MHz	34
8	8	1	12 pJ	296 MHz	34

TABLE II  
OPTIMIZED FOR ENERGY

The transition count for a ROM represents the number of sequential transistor switches for a single ROM access.

Figure 8 illustrates the results of speed and energy for a ROMantic-generated ROM with six inputs, eight outputs, and 81 entries. The nominal voltage of the 0.18- $\mu\text{m}$  TSMC process used in the simulation is 1.8 V. Figure 9 is a log-log plot of the delay vs. energy for the same ROM; the fact that the line with slope  $-0.5$  is tangent to the delay-energy curve shows that  $Et^2$  is very nearly constant around the nominal operating voltage.

## VI. CONCLUSION AND FUTURE WORK

We have presented a ROM generation tool that can create asynchronous memory layout with parameterizable optimizations. In addition, we have provided results on a set of benchmark ROMs that are typical of 8-bit microcontroller memory. These preliminary results are encouraging because they show that the generated ROMs already meet the performance and energy requirements of the *Lutonium* for which they were originally designed.

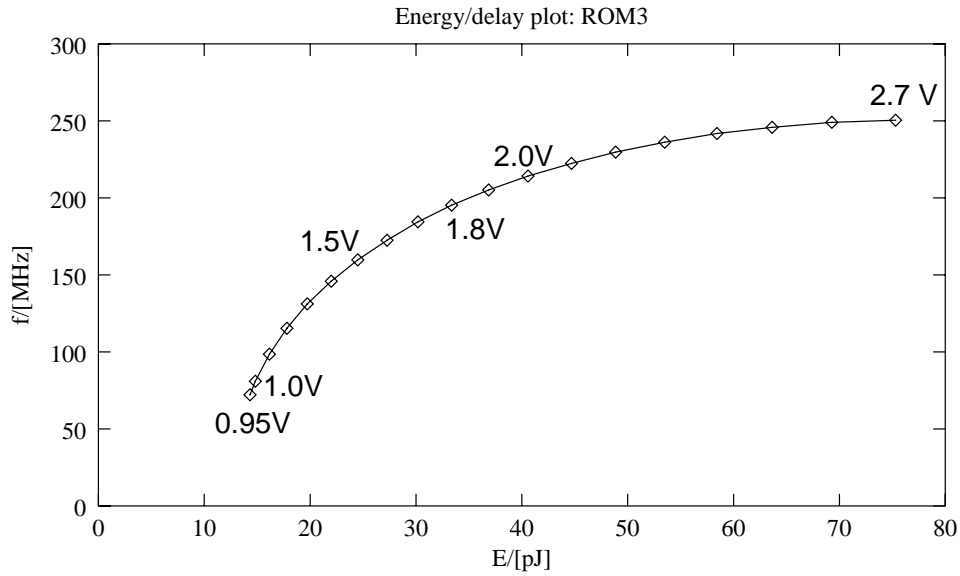


Fig. 8. Speed vs. energy for a ROMantic-generated ROM with feet.

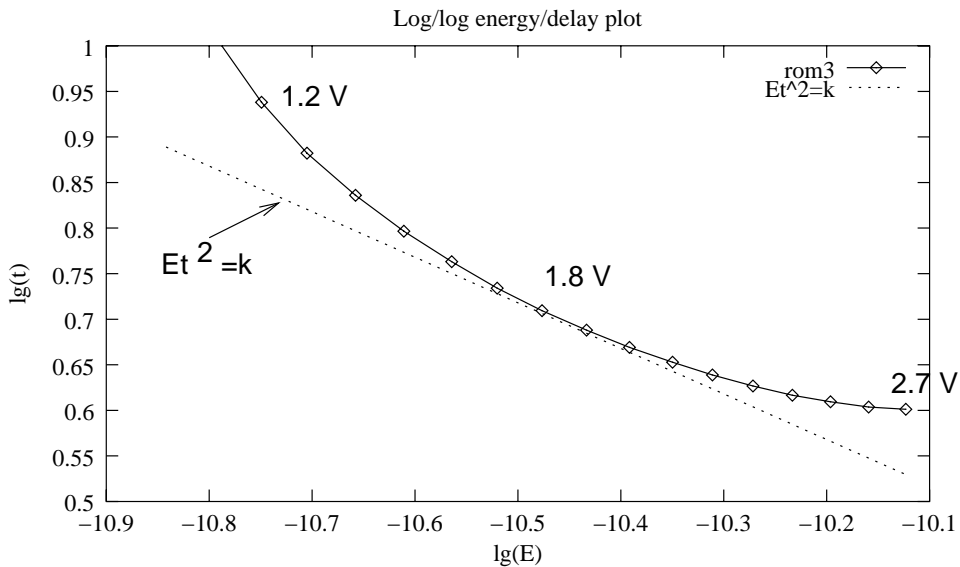


Fig. 9. Delay vs. energy for a ROMantic-generated ROM with feet: log/log scale; constant- $E^2$  shown for reference.

One of the main design goals for ROMantic was to make the tool flexible enough to allow its use in just about any asynchronous application. We are currently working on improvements to ROMantic to further increase its utility in asynchronous VLSI design and research.

It is often the case in practice that some outputs of a function that we desire to implement as a ROM are independent of some of its inputs and other outputs are independent of other of its inputs. For these functions, a different circuit structure, the *programmable logic array* (PLA), is more efficient [14]. PLAs are similar in structure to our asynchronous ROMs in that they contain two planes of logic gates with pulldown transistors to program logic functions.

Unfortunately, in a PLA, the data sent from the decoder (the “AND plane”) to the output array (the “OR” plane) can no longer be expressed as a 1-of- $n$  code; indeed it cannot necessarily be expressed as any delay-insensitive code. The only reasonable way to implement an asynchronous PLA is to replace the circuitry computing  $Decode^v$  with timing assumptions. One of the enhancements we intend for the near future will be for ROMantic to also generate PLAs.

## ACKNOWLEDGMENTS

The research described in this paper was supported by the Defense Advanced Research Projects Agency (DARPA) as part of the program in Power-Aware Computing and Communications (PAC/C) and monitored by the Air Force Office of Scientific Research.

## REFERENCES

- [1] R. Bryant, D. Beatty, K. Brace, K. Cho, and T. Scheffler. COSMOS: A Compiled Simulator for MOS Circuits. In *Proc. 24th Design Automation Conference*, 9-16, 1987.
- [2] C.A.R. Hoare. Communicating Sequential Processes. *Communications of the ACM*, **21**(8):666-677, 1978
- [3] Andrew M. Lines. *Pipelined Asynchronous Circuits*. M.S. thesis, California Institute of Technology CS-TR-95-21, 1995.
- [4] Alain J. Martin. Compiling Communicating Processes into Delay-insensitive VLSI circuits. *Distributed Computing*, **1**(4), 1986.
- [5] A. J. Martin. The limitations to delay-insensitivity in asynchronous circuits. In W. J. Dally, ed., *Sixth MIT Conference on Advanced Research in VLSI*. Cambridge, Mass.: MIT Press, 1990.
- [6] Alain J. Martin. Synthesis of Asynchronous VLSI Circuits. *Formal Methods for VLSI Design*, ed. J. Staunstrup, North Holland, 1990.
- [7] Alain J. Martin, Steven M. Burns, Tak-Kwan Lee, Drazen Borkovic, and Pieter J. Hazewindus. The design of an asynchronous microprocessor. In Charles L. Seitz, editor, *Advanced Research in VLSI: Proceedings of the Decennial Caltech Conference on VLSI*, pp. 351-373, MIT Press, 1991.
- [8] Alain J. Martin. Asynchronous Datapaths and the Design of an Asynchronous Adder. *Formal Methods in System Design*, 1:1, Kluwer, 117-137, 1992.
- [9] Alain J. Martin. *Synthesis of Asynchronous VLSI Circuits*. Caltech Computer Science Technical Report CS-TR-93-28. Pasadena, Calif.: California Institute of Technology Computer Science Department, 1993.
- [10] A.J. Martin, A. Lines, R. Manohar, M. Nyström, P. Penzes, R. Southworth, U. Cummings, and T.K. Lee. The Design of an Asynchronous MIPS R3000 Processor. *Proceedings of the 17th Conference on Advanced Research in VLSI*. Los Alamitos, Calif.: IEEE Computer Society Press, 1997.
- [11] Alain J. Martin, Mika Nyström, and Paul I. Pénez.  $E^2$ : A Metric for Time and Energy Efficiency of Computation. In *Power-Aware Computing*, R. Melhem and R. Graybill, eds. Boston, Mass.: Kluwer Academic Publishers, 2002.
- [12] Alain J. Martin, Mika Nyström, Karl Papadantonakis, Paul I. Pénez, Piyush Prakash, Catherine Wong, Jonathan Chang, Kevin S. Ko, Benjamin Lee, Elaine Ou, Eino-Ville Talvala, James Tong, Ahmet Tura. The Lutonium: A Sub-Nanojoule Asynchronous 8051 Microcontroller. In preparation.
- [13] Robert N. Mayo, Michael H. Arnold, Walter S. Scott, Don Stark, and Gordon T. Hamachi. 1990 DECWRL/Livermore Magic Release. DECWRL Research Report 90/7. Digital Equipment Corporation, 1990.
- [14] Carver Mead and Lynn Conway. *Introduction to VLSI Systems*. Reading, Mass.: Addison-Wesley, 1980.
- [15] Greg Nelson, ed. *Systems Programming with Modula-3*. Englewood Cliffs, N.J.: Prentice Hall, 1991.
- [16] Mika Nyström.  $E^2$  and Multi-voltage Logic, Caltech Technical Report, April 1995.
- [17] R. Rudell and A. Sangiovanni-Vincentelli. Espresso-MV: Algorithms for Multiple-Valued Logic Minimization. *Proc. Cust. Int. Circ. Conf.*, Portland, May 1985.