THE YORKTOWN SIMULATION ENGINE

Monty M. Denneau

IBM T. J. Watson Research Center, P. O. Box 218,
Yorktown Heights, New York 10598, U. S. A.

ABSTRACT

The Yorktown Simulation Engine (YSE) is a high speed special purpose parallel processor designed and built at the I.B.M. Thomas J. Watson Research Center to simulate the logical operation of large digital networks. A full YSE configuration simulates networks of up to 2,000,000 gates at a rate exceeding 3 billion gate computations per second, doing more simulation in just eight hours than an IBM 370/168 does in an entire year.

This paper reviews gate-level logic simulation and describes the architecture and hardware implementation of the YSE. A companion paper by G. Pfister and E. Kronstadt discusses the YSE software.

1. GATE-LEVEL LOGIC SIMULATION

A *logical network* is a set {LV,S,SRC,FS} where LV is a set of *logic values,* S is a directed graph of *signals,* <SRC(s,1),SRC(s,2),...SRC(s,indeg(s))> enumerates the immediate predecessors of each s in S and, when s has non-zero in-degree,

$$FS(s) : LV^{indeg(s)} \dashrightarrow LV$$

is an indeg(s)-ary *logical function.* Here indeg(s) stands for the in-degree of s. If indeg(s) = 0, then s is an *external input.* Every physical configuration of logical gates yields a logical network when buses (e.g. wire-OR, wire-AND, tri-state) are replaced by their equivalent logical functions.

Figure 1 shows a gate configuration equivalent to one half of a 74S74 TTL D-type flip-flop and Figure 2 gives a derived logical network in table form. Note that the particular numbering of the signals is arbitrary.

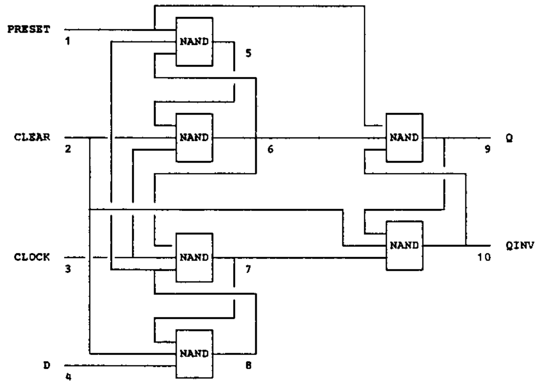


Figure 1. 74S74 TTL D-type flip flop

s	(NAME)	indeg(s)	FS(s)	SRC(s,1)	SRC(s,2)	SRC(s,3)
1	PRESET	0				
2	CLEAR	0				
3	CLOCK	0				
4	D	0				
5		3	NAND_3	1	8	6
6		3	NAND_3	5	2	3
7		3	NAND_3	6	3	8
8		3	NAND_3	7	2	4
9	Q	3	NAND_3	1	6	10
10	QINV	3	NAND_3	9	2	7

```
LF = <0, 1, U, Z>

NAND_3(A,B,C) = 1 if A = 0 or  B = 0 or  C = 0
              = 0 if A = 1 and B = 1 and C = 1
              = U otherwise
```

Figure 2. Table for 74S74

UNIT-DELAY SIMULATION

The *unit-delay logic simulation model* presumes that all physical implementations of logical functions have the same input-to-output delay, called one *unit-delay,* and that external inputs change only on integral unit-delay time boundaries. Let LN be a logical network and, for some positive integer T, let a partial function VAL : S x <0..T> --> LV be specified so that VAL(s,t) exists whenever t = 0 or whenever s is an external input. That is, we are given an initial state of LN and the behavior of its external inputs over a period of T gate-delays. The following *unit delay simulation procedure* extends VAL to a total function that gives the resulting behavior of LN.

Unit Delay Simulation Procedure :

```
for t = 1 to T do
    for all s in S do
        F = FS(s)
        ids = indeg(s)
        VAL(s,t) =
          F(VAL(SRC(s,1),t-1),...,VAL(SRC(s,ids),t-1))
```

The reader should be warned that if LV = {0,1,U} and U *(undefined)* is interpreted as a *set* {0,1} of possible values, then the function VAL computed by the procedure may not correctly model the behavior of an ideal physical realization of LN.

The table in Figure 3 shows the result of applying the unit delay simulation procedure to the circuit of Figure 1 when a negative clear pulse is followed by a negative clock pulse. Note that the computation of VAL for time t+1 requires only those values computed at time t, so that a program can execute the the unit delay simulation procedure for a network with N functions by *A/B-ing* between two N-element sections of storage.

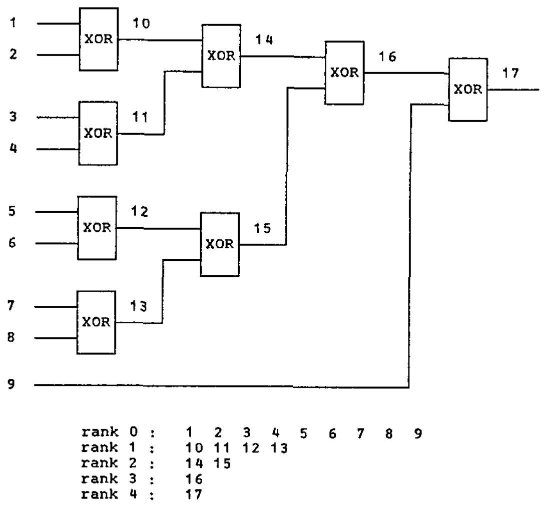
Elapsed Gate Delays ——>

SIGNAL	(NAME)	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
1	PR	U	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
2	CLR	U	1	1	0	0	0	1	1	1	1	1	1	1	1	1	1	1
3	CLK	U	1	1	1	1	1	1	1	0	0	0	1	1	1	1	1	1
4	D	U	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
5		U	U	U	U	U	0	0	0	0	0	0	1	1	1	1	1	1
6		U	U	U	U	1	1	1	1	1	1	1	1	0	0	0	0	0
7		U	U	U	U	U	0	0	0	0	1	1	1	1	1	1	1	1
8		U	U	U	U	1	1	1	1	1	1	0	0	0	0	0	0	0
9	Q	U	U	U	U	U	0	0	0	0	0	0	0	0	1	1	1	1
10	QINV	U	U	U	U	1	1	1	1	1	1	1	1	1	1	0	0	0

Figure 3. Unit delay simulation example

RANK-ORDER SIMULATION

Let LN be a logical network whose underlying directed graph S is acyclic. Define the function *rank* on S recursively by rank(s) = 0 if indeg(s) = 0 and rank(s) = n + 1 if rank(t) < n + 1 for all immediate predecessors t of s while rank(u) = n for some immediate predecessor u of s. The *depth* of an acyclic logical network is the maximum value of rank(s). Figure 4 shows the rank-ordering of a 9-input parity tree with depth 4.



```
rank 0 :   1  2  3  4  5  6  7  8  9
rank 1 :   10 11 12 13
rank 2 :   14 15
rank 3 :   16
rank 4 :   17
```

Figure 4. Rank-ordering of a 9-input parity tree

Let D be the depth of a logical network LN containing N logical functions. If a fixed set of logic values are assigned to the external inputs of LN, then the unit delay simulation procedure will do NxD function evaluations in determining the induced state of LN, since each pass over the network propagates signals through only one functional level. It is possible however to reduce the number of evaluations.

Let VAL : S --> LV be a partial function whose value is specified for every external input of LN. The following *rank-order simulation procedure* extends VAL to a total function using just one evaluation for each logical function in LN.

Rank-Order Simulation Procedure :

```
for r = 1 to depth(LN) do
    for all s in S such that rank(s) = r do
        F = F(s)
        ids = indeg(s)
        VAL(s) =
            F(VAL(SRC(s,1)),...,VAL(SRC(s,ids)))
```

Figure 5 shows the sequence of computations that result when the rank-order simulation procedure is applied to the configuration of Figure 4 using the input values given in the table for rank 0. Note that even the coarse timing assumptions of the unit-delay simulation model are ignored, so that that the function VAL has only one argument.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
RANK 0	0	0	0	1	1	0	1	1	0								
RANK 1										0	1	1	0				
RANK 2														1	1		
RANK 3																0	
RANK 4																	1

Figure 5. Rank-order simulation example

The above two procedures can be combined in a practical simulation program for synchronous digital logic. Each clock cycle is divided into three stages. During the first (rank-order) stage, the system clock is changed and propagated through the clock-distribution logic. During the second (unit-delay) stage, all storage elements update their contents depending on changes at their clock inputs. During the third (rank-order) stage, the outputs of all storage elements are propagated through the combinational logic.

There is also a third simulation technique, *event-triggered simulation,* in which functions are evaluated only when at least one of their arguments has changed value. Thus the number of function evaluations is reduced, but at the expense of queue-management overhead.

2. THE YORKTOWN SIMULATION ENGINE

The Yorktown Simulation Engine (YSE) was designed to execute the unit-delay and rank-order simulation procedures (or combinations of the two) for large digital networks at rates far exceeding those possible on conventional serial computers. From the simplest point of view, the YSE is a collection of identical *logic processors* along with an inter-processor *switch.* Each logic processor contains an *instruction memory* that stores the interconnection and function type information <S,SRC,FS> for a logical network, a *data memory* that holds logic values for signals in the network, and a *function unit* that evaluates logical functions. A large logical network is partitioned into sub-networks, and each sub-network is assigned to a logic processor. During simulation, the logic processors simultaneously step through their instruction memories and use the information fetched to access and update values for signals stored in their data memories. In addition, the logic processors transfer signal values between their data memories and the switch to accommodate the communication introduced by the partitioning. The partition of the network into sub-networks, the sequence of function evaluations within each logic processor, and the sequence of switch routing configurations are all determined and fixed prior to simulation by the YSE programming support running on a conventional computer.

Figure 6 shows an overview of the YSE. We now give a more detailed discussion of each system component.

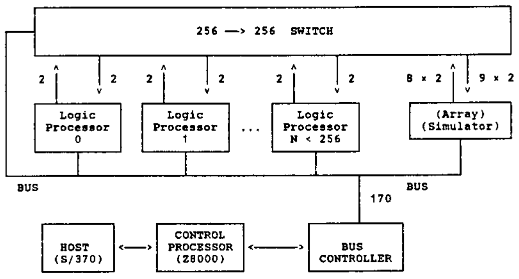


Figure 6. YSE overview

LOGIC PROCESSOR

Figure 7 gives a simplified overview of the YSE logic processor organization.

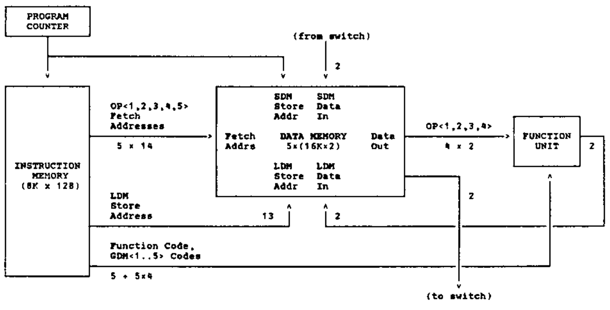


Figure 7. Logic processor overview

Each logic processor accommodates a logical network of up to 8K functions of four 2-bit arguments. Using the 8-stage pipeline shown in Figure 8, a logic processor completes one function evaluation every 80 nanoseconds.

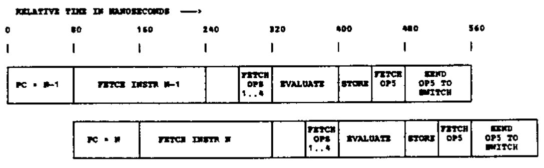


Figure 8. Logic processor pipeline

The ***instruction memory*** contains 8K 128-bit instructions, each of which consists of five *operand fetch addresses,* a *store address,* five *GDM codes,* a *function code,* and an *interrupt mask.* The use of each of these fields will be discussed below. During simulation, the instruction memory is addressed by a *program counter* that repeatedly steps between starting and ending addresses that are the same in all processors. The 80 ns. fetch of one instruction is called an *instruction cycle,* and the execution of all instructions between the starting and ending addresses a *simulation cycle.*

The ***data memory*** implements a 16K x 2-bit random access memory (RAM) with two write ports and five read ports. All seven ports are accessed once for every instruction cycle. Physically, five identical 16K x 2 banks are used, with each bank divided into an 8K x 2 *local data memory* (LDM) and an 8K x 2 *switch data memory* (SDM).

During the first half of each instruction cycle, the output of the function unit (see below) is written into the LDM at the location given by the store address, and data received from the switch (see below) is written into the SDM at the location equal to the value of the (appropriately skewed) program counter. During the second half, five operands are fetched from the banks at the locations given by the five operand fetch addresses. The first four are passed to the function unit while the fifth is sent to the switch. A pipeline bypass mechanism allows an instruction to fetch the value of the function computed by the previous instruction.

For unit delay simulation, values are fetched from one half of the LDM (SDM) and stored into the other half; the roles of the two halves are reversed after each simulation cycle.

The ***function unit*** is shown in Figure 9. It accepts four 2-bit operands from the first four banks of data memory, five 4-bit GDM codes from the instruction memory, and a 5-bit function code from the instruction memory. These are used to compute a 2-bit value which is subsequently stored back into the local data memory.

Each of the 64 x 2 *Generalized DeMorgan Memories* (GDM) shown in Figure 9 is loaded during YSE initialization with the truth tables for sixteen 2-bit-valued functions of a single 2-bit argument. These functions typically include the four constant functions, the identity, and the 2-bit invert function (if 00,01,10,11 are interpreted as logic 0, logic 1, *undefined,* and *floating* then the 2-bit invert function for TTL logic maps 00,01,10,11 to 01,00,10,10). The 4-bit GDM code from the instruction memory is used for the four high-order address bits to select one of the 16 tables, and the 2-bit data memory operand is used as the two low-order address bits to access a particular entry within that table.

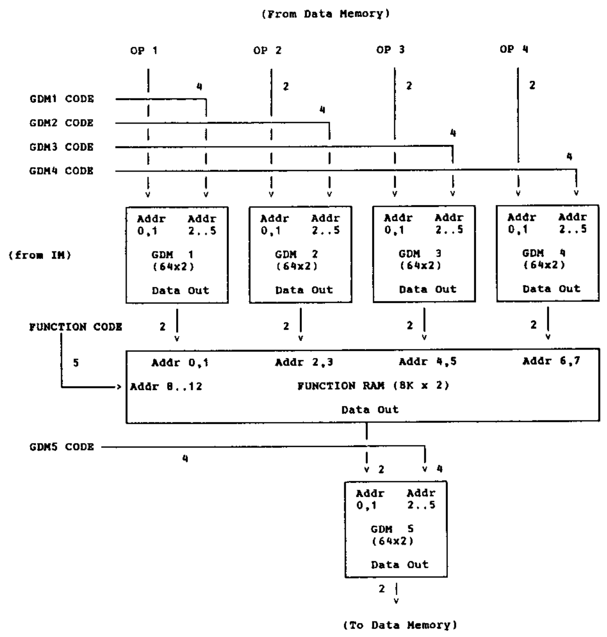


Figure 9. Logic processor function unit

Similarly the 8K x 2 function memory is loaded during initialization with the truth tables for thirty-two 2-bit-valued functions of four 2-bit arguments. The 5-bit function code from the instruction memory selects one of the thirty-two tables, and the four 2-bit outputs from GDM1 through GDM4 select a particular entry within that table.

All common gate functions of up to four arguments can be computed with only the AND, XOR, and IDENTITY functions stored in the function memory. Functions of more than four arguments are computed by iteration. The remaining function memory tables are used for higher level operations such as select, add, majority, and latch-if-clocked.

The execution of each instruction results in the computation of a 2-bit signal that can take one of four possible values. The 4-bit interrupt mask in the instruction memory specifies a (possibly empty) subset of these values. If the computed signal falls within the specified subset, the logic processor ***interrupt logic*** notifies the bus controller (see below) that an interrupt has occurred, and the bus controller optionally halts the machine. The control processor (see below) then locates the source of the interrupt and carries out any appropriate actions. Note that since logic values can be transferred though the switch, any functional combination of simulated signals generated in the logic processors can be used to trigger an interrupt.

Physically, a logic processor consists of about 600 TTL and 130 MOS memory modules mounted on a single Multiwire board measuring roughly two feet on a side. Each board contains about 15,000 holes and 8000 wires. Thirty-two logic processors, with a total simulation capacity of 256,000 functions, occupy a single air-cooled 24"-wide vertical cabinet.

SWITCH

The switch is a 256 --> 256 3-bit (2 data plus 1 parity) crosspoint whose configuration for each of 8K 80-ns time-slots is loaded during YSE initialization. Sixteen boards implement an 8K x (256x8) random access memory that is addressed during simulation in the same manner as the logic processor instruction memories. Forty-eight boards implement an 80-ns 3 x (256 --> 256) pipelined multiplexer. Every 80 ns the multiplexer receives 256 8-bit addresses from the memory boards and 256 3-bit signals from the logic processors, and returns 256 3-bit signals to the logic processors. Four additional boards handle level-shifting and control. The switch is built primarily with Shottky and low power Shottky TTL modules on 14"x18" Multiwire panels and occupies a single 24" vertical rack.

BUS

The YSE bus is used by the bus controller to write, read, and control the operation of the logic processors and switch. It consists physically of 170 twisted pair wires carrying unidirectional signals at differential ECL levels.

Sixteen *x-select* and sixteen *y-select* lines are sent out from the bus controller to address any of 256 logic processors. To facilitate rapid location of interrupts, sixteen *x-response* and sixteen *y-response* wire-OR lines are returned from the logic processors to the bus controller.

Ten parity bits provide error protection. All bus transactions are pipelined, so that 16-bit data can be transferred in either direction at a rate of 25 megabytes/second.

BUS CONTROLLER

The bus controller connects the control processor to the YSE, regulates, packs, and unpacks data during block transfers, and controls the operation of the machine during simulation. It also contains indirect addressing tables that make it possible to access signal values scattered around the machine in a single block transfer operation.

The bus controller is implemented on four wire-wrapped boards with about 500 TTL, ECL, and MOS memory modules.

CONTROL PROCESSOR

The control processor reduces the requirements for frequent communication between the host and the YSE. It sets up block transfer paths among the host, its own memory, and the bus controller, and carries out diagnostics and interrupt handling. The control processor is currently a standard Z8000 microcomputer system with about 100 wire-wrapped TTL modules for the interface logic.

ARRAY SIMULATOR

A large computer contains many arrays (*e.g.*, main memory, cache and register files) which, if simulated at the gate level, would exhaust the capacity of the YSE. Thus array accesses are currently simulated by the relatively slow control processor, with an accompanying degradation in system performance. To illustrate a solution to this problem as well as to the general problem of attaching external devices to the YSE, we give here a simplified version of a proposal by the author for an array simulator that can perform at least a thousand independent simulated array accesses per simulation cycle.

Nine logic processors, MLP0 ... MLP8, are selected for use with the array simulator. Each of MLP0 .. MLP7 is modified by inserting a 2x(2-->1) multiplexer between the output of function memory and the input of GDM5, with the select signal to the multiplexer being provided by a previously unused bit of instruction memory.

A 16-bit storage is provided whose interface consists of 24 address inputs, 16 data inputs, 16 data outputs, and a write-enable input. Since each YSE signal value requires two bits for its representation, the 16-bit words of the storage will hold simulated 8-bit words.

The 24 address inputs, 16 data inputs, and the write-enable input of the storage are furnished by outputs of GDM1, GDM2, and GDM3 in MLP0 through MLP7, the outputs of GDM4 in MLP0 through MLP7, and the output of GDM1 in MLP8. The 16 data outputs of the storage furnish inputs for the 8 multiplexers installed in MLP0 ... MLP7.

During simulation, address, data, and write-enable bits are computed by the logic processors and sent through the switch to MLP0 ... MLP8. When all values necessary for a particular access have been accumulated, the nine processors simultaneously fetch them and carry out a storage access. Depending on the access time of the storage, this operation may be pipelined over several instruction cycles. For read operations, the values returned are selected by the multiplexers and written into data memory just as if it had come from the function unit during normal operation. These values are then available to be sent through the switch to the other logic processors.

Simulated arrays are mapped into different sections of the storage by fetching some of their high order address bits as constants. Arrays wider than 8 bits are implemented with multiple accesses. Additional hardware which we do not describe here can be introduced to efficiently pack simulated arrays of varying widths into the available physical storage. Note that the multiplexers described above permit normal use of the processors when array accesses are not in progress.

As of this writing, no array simulator for the YSE has been constructed.

HOST

An IBM 370/168 is currently the host computer for the YSE. Logic descriptions in several high and low level languages are compiled into into YSE programs which are loaded into the machine over a parallel data link. Other programs in the host allow YSE users to interactively observe and change logic.

DIAGNOSTICS AND ERROR PROTECTION

All sections of memory in the YSE carries parity, as does the YSE Bus and paths between the logic processors and the switch. The following simple scheme makes it possible to check out all 256 processors simultaneously.

Data can be broadcast by the bus controller simultaneously to all processors. Conversely data from the processors can be read simultaneously onto what is effectively a 256-way open-emitter bus. Since this does not normally allow simultaneous verification, the logic processors have the facility to invert their data before placing it on the bus. The correctness of a value written into the 256 processors can thus be determined by reading first in, and then out of phase. Any errors will be detected during at least one of the two operations.

With this mechanism, extensive diagnostics can be run on up to 40 megabytes of logic processor memory in less than ten seconds.

3. HISTORY AND ACKNOWLEDGMENTS

The YSE was proposed in October 1979 by the author who subsequently designed and built the logic processors, switch, and bus controller. The architecture was derived from the Logic Simulation Machine, which was invented by John Cocke, designed by John Shedletsky and Rick Malm (Los Gatos), and built by Rick Malm. Along with improved capacity, speed, and error protection, the YSE is distinguished from the LSM by its rank-order simulation mode, general-purpose function unit, a more powerful switch communication mechanism, and an alternate host attachment.

Logic design for the YSE was done using the ILL system developed by Ravi Nair. Programs written by Mark Laff, Ravi Nair and the author optimized wiring, simulated logical operation and verified electrical performance. Tom Nolan and Mark Laff designed the Control Processor interfaces and Jerry Brody designed the bus distribution system. The overall hardware development was managed by Howard Brauer.

The YSE 370 software was written primarily by Eric Kronstadt, Mark Laff, and Greg Pfister, and is discussed in the companion paper.