EMULATING DIGITAL LOGIC USING TRANSPUTER NETWORKS (VERY HIGH PARALLELISM = SIMPLICITY = PERFORMANCE)

P.H.Welch
Computing Laboratory, University of Kent at Canterbury, CT2 7NF, ENGLAND

ABSTRACT

Modern VLSI technology has changed the economic rules by which the balance between processing power, memory and communications is decided in computing systems. This will have a profound impact on the design rules for the controlling software. In particular, the criteria for judging efficiency of the algorithms will be somewhat different. This paper explores some of these implications through the development of highly parallel and highly distributable algorithms based on *occam* and *transputer* networks. The major results reported are a new simplicity for software designs, a corresponding ability to reason (formally and informally) about their properties, the reusability of their components and some real performance figures which demonstrate their practicality. Some guidelines to assist in these designs are also given. As a vehicle for discussion, an interactive simulator is developed for checking the functional and timing characteristics of digital logic circuits of arbitrary complexity.

INTRODUCTION

The 'real world' consists of large numbers of autonomous structures operating concurrently and communicating (sometimes continually) in order to achieve common goals. It is not surprising that traditional sequential computing techniques — aimed at wringing optimum performance out of the traditional 'von Neumann' computer — eventually cease to be useful for realising systems above a certain level of complexity. The mis-match between target and implementation concepts is too great.

Occam [INMOS 83, May–Shepherd 85, INMOS 87, Bowler 87, Kerridge 87, Jones 87, Pountain–May 87, Burns 88, Welch 88] is a simple programming language which allows us to express and reason about parallel (MIMD) designs with the same ease and fluency to which we are accustomed for sequential systems. Occam enforces some dramatic simplifications (such as not allowing parallel processes directly to share resources like data objects or communication channels) which sidestep many of the standard problems of concurrency. Occam communication is point-to-point, synchronised and unbuffered. However, broadcast, multiplexed, asynchronous or buffered communication may be obtained with minimal overhead by applying (reusable) software components to just those places where they are needed.

The simplicity of occam's concept of parallelism has two important consequences. Firstly, it allows us to be confident about designs with very high levels of concurrency (say > 100,000 parallel processes). Secondly, it permits very fast implementations. The INMOS transputer [INMOS 84, May–Shepherd 85], which was itself designed concurrently with the occam language, imposes an overhead of about one micro-second when switching between parallel processes. Further, depending on the degree of parallelism in the occam design, the software may be distributed on to arbitrary size nets of transputers. Since inter-transputer links (like the occam communication channels that they implement) are point-to-point, the size of the network is never limited by the usual contention problems which arise from more normal methods of combining many processors (e.g. on a common bus). *With careful load-balancing and design*, performance can be improved linearly with the number of processors used. This paper makes a contribution to that care.

This situation compares favourably with the current state of Ada [AJPO 83] technology. Ada tasking is related to occam parallelism (both are derived from CSP [Hoare 85]), but is more complicated. It also violates certain principles of software engineering which makes it very difficult to use for describing highly parallel systems (although with rigorous discipline, this can be overcome to a certain extent — see [Welch 86, 87a]). However, much work needs to be done to reduce the task switching overhead from, typically, one mille-second to something usable. Also, the distribution of a parallel system expressed as a single Ada program — essential for an integrated understanding of the whole design — on to multiple processors is still being researched.

EMULATING DIGITAL LOGIC

There exist many commercial packages [e.g. Cirrus 82, Tamsley-Dow 82] for simulating digital logic circuits so that their functional and timing characteristics can be checked before actual fabrication. Since the design of these packages have been optimised for conventional sequential execution, we do not believe that they are an appropriate starting point from which to try to extract parallelism. At best, some elements of SIMD and/or simple expression evaluation pipelining may be found. We wish to exploit more general forms of parallelism than these.

Further, because digital logic circuits are inherently such parallel systems, sequential simulations will need to manage complex data structures — necessarily in some global, persistent data space — in order to maintain the states of their various components. Also, there will have to be explicit means of scheduling the single execution thread of control around the components. These details are artifacts of the sequential implementation technique and obscure what is trying to be expressed.

In occam, every object in the system may be mapped on to an individual active process with its own local (private and, usually, very simple) state information. The parallel inter-object relationships are expressed directly and scheduling is automatic — the system designer is not concerned with the management of any global data-base to maintain the overall system.

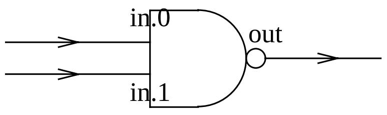
There is one optimisation we lose with this direct approach. In a sequential implementation, advantage may be taken of the fact that the state of the whole system is available globally. For digital logic simulation, this enables us to work only on those parts of the circuit where we know logic levels are changing. In the parallel implementation we have built, we simulate ‘real world’ objects (e.g. gates) very closely and only have local information available (like the ‘real’ gates). So, just as gates are analog devices which continuously transform input signal levels into output signal levels (even when these levels are steady), our gate emulations will continuously operate on (digital samples of) input and output signals. Consequently, it might be argued that our models are inefficient in comparison to the standard ones. However, in the parallel world, the rules for judging efficiency are rather changed.

In [Dowsing 85], an attempt is made to recover this kind of optimisation through a central pool of information to which all gates report. We feel that the extra complexities, resources and restrictions needed to manage this are not worthwhile. We prefer to keep the designs simple and understandable. *In the parallel environment, it will not matter if we ‘burn up’ some MIPS — re-computation will often prove more cost-effective than the management of complex data-structures for the maintainance of previous results.*

There is an analogy with earlier days of computing when memory resources were scarce. It used to be more important to conserve memory than organise clear data structures — great complexities, for instance, would be introduced to manage ‘overlays’. Now that distributed processing power and communications are (becoming) abundant, we should stop being so concerned at optimising their use and seek new simplicity in our designs.

BASIC DIGITAL CYCLES

We concentrate on modelling the behaviour of real hardware components. Consider a *nand* gate with two input pins, *in.0* and *in.1*, and one output pin, *out* :–



In our emulation, we shall be supplying an (infinite) stream of samples of the input signal levels and producing — at the same rate — an (infinite) stream of samples of the output signal. The pseudo-rate of sampling will be defined by the input data for the simulation and may be set to any value (e.g. one sample per nano-second).

In the models we have built, we allow only two sample values, *HIGH* and *LOW*. (The technique could easily be extended to allow a more continuous range of values — see the section on **EXTENSIONS**.) We do not need to develop ‘multi-valued’ logics to represent notions like ‘rising’ or ‘falling’ edges, ‘spikes’ or ‘noise’. For simplicity, we have given ourselves the extravagance of 32-bit quantities to represent this two-valued logic :–

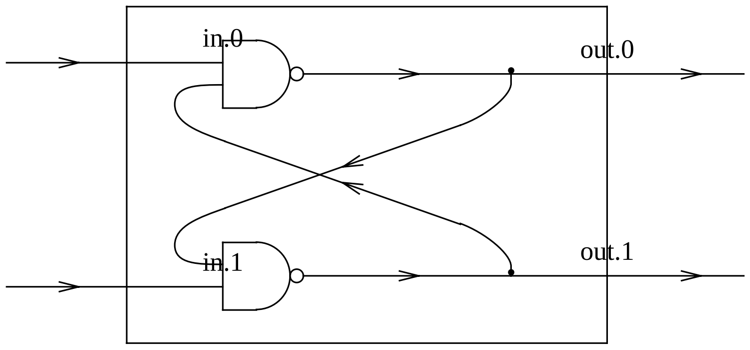
```
VAL INT LOW IS #00000000,   HIGH IS ~ LOW:      -- this would be a DEF in occam
```

[The examples presented in this paper are expressed in *occam 2* [INMOS 87], a strongly typed extension of *occam* [INMOS 83]. We assume the reader has at least encountered the original language, which is sufficiently close to the new standard for the details to be followed. In particular, the constructs supporting parallelism are identical.]

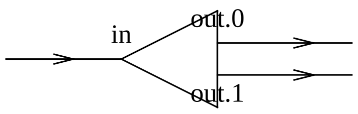
An (incorrect) model of the *nand* gate may now be given :–

```
PROC nand.2 (CHAN OF INT in.0, in.1, out)
  WHILE TRUE
    INT a.0, a.1:
    SEQ
      PAR
        in.0? a.0
        in.1? a.1
      out! ~ (a.0 /\ a.1)
```

The trouble with this model is that it captures no notion of time — it represents a gate with zero ‘propagation delay’. Severe problems arise if we try to use such devices in circuits with feedback. Consider a simple *latch* :–

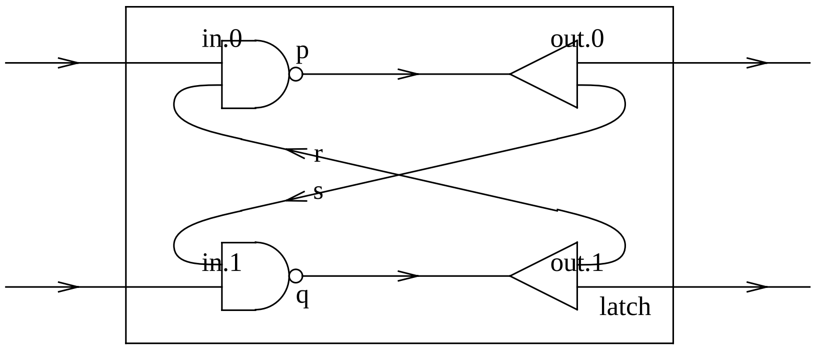


First though, there is another problem. The output from each *nand.2* gate needs to be ‘broadcast’ to two devices — the other *nand.2* gate and whatever is attached to the *latch* output. Rather than modify the *nand* gate to equip it with a pair of outputs, we imitate the hardware and introduce a process to represent the ‘soldering’ :–



```
PROC delta.2 (CHAN OF INT in, out.0, out.1)
  WHILE TRUE
    INT a:
    SEQ
      in? a
      PAR
        out.0! a
        out.1! a
```

The latch circuit may now be drawn :–



and implemented :–

```
PROC latch (CHAN OF INT in.0, in.1, out.0, out.1)
  CHAN OF INT p, q, r, s:
  PAR
    nand.2 (in.0, r, p)
    nand.2 (s, in.1, q)
    delta.2 (p, out.0, s)
    delta.2 (q, r, out.1)
```

Returning to the earlier problem, this circuit deadlocks immediately! There is no notion of delay in the *delta.2* or *nand.2* processes and so, because of the feedback, nothing can proceed.

The problem is solved by adding a ‘propagation delay’ to the model of the gate. A delay of one sample cycle may be introduced by ‘double buffering’ the internal logic of the gate :–

```
PROC nand.2 (CHAN OF INT in.0, in.1, out)
  INT a.0, a.1, b.0, b.1:
  SEQ
    b.0 := UNKNOWN
    b.1 := UNKNOWN
    WHILE TRUE
      SEQ
        PAR
          in.0? a.0
          in.1? a.1
          out! ~ (b.0 /\ b.1)
        PAR
          in.0? b.0
          in.1? b.1
          out! ~ (a.0 /\ a.1)
```

Note that we now input a ‘slice’ of sample signal levels and output a sample level in parallel — a more accurate model of the real gate. This illustrates a general principle of parallel design.

> When logic does not dictate the order in which things need to be done — *especially regarding input and output* — do not arbitrarily specify some sequence. Placing unnecessary sequential constraints on the way the system synchronises is the short-cut to deadlock.

Note that also on its first cycle, *nand.2* outputs some indeterminate value. *UNKNOWN* may be a constant set to *HIGH*, *LOW* or some other value — e.g. :–

```
VAL INT UNKNOWN IS #55555555,   KNOWN IS ~ UNKNOWN:
```

Alternatively, *UNKNOWN* may be set through some extra (VAL or even CHAN) parameter to *nand.2*.

A general method of allocating arbitrary (multiples of sample cycle) delays to any gate is given in the **EXTENSIONS** section. For the moment, all our gates will be modelled as above with a propagation delay of one sample cycle.

BUILDING LARGE CIRCUITS

First, we try to pin down some useful ideas :–

Definition: A device is *I/O-PAR* if it operates (or may be scheduled to operate) cyclically such that, once per cycle, it inputs a complete slice of values from every input channel *in parallel* with outputting a slice of values to every output channel.

Theorem A: any network constructed from *I/O-PAR* components — no matter how large or however much feedback there is in the network — will never deadlock and is also *I/O-PAR*.

'Proof ': the circuit will be self-synchronising around the parallel I/O operation in each device. We may imagine that, at any instant, there is either nothing on all the wires or there is precisely one item of information in flight along every wire. Physically, the circuit may be scheduled very differently with some nodes getting ahead of others — but they will eventually be held up through lack of input data and the rest of the circuit will catch up.

Definition: we call a device *I/O-SEQ* if it is not *I/O-PAR* but operates (or may be scheduled to operate) cyclically such that, once per cycle, it first inputs in parallel a complete slice of values from each input channel *and then* in parallel outputs a slice of values to each output channel.

Theorem B: any acyclic network constructed from *I/O-SEQ* components will never deadlock and is itself *I/O-SEQ*, but a cyclic network will deadlock.

'Proof ': clear.

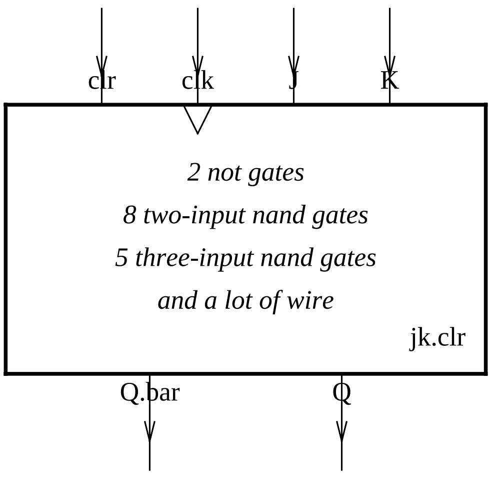
Theorem C: any network constructed from *I/O-PAR* and *I/O-SEQ* components, such that there is (i) no sub-cycle consisting of just *I/O-SEQ* components and (ii) no direct connection from input to output via just *I/O-SEQ* components will never deadlock and is *I/O-PAR*.

'Proof ': because of theorems A and B, this reduces to showing that an *I/O-PAR* device followed by an *I/O-SEQ* device (or vice-versa) is still *I/O-PAR*. This is trivial.

Of course, the above definitions, theorems and proofs need a formal treatment based upon the techniques of CSP and the formal semantics of occam [Roscoe–Hoare 86]. This is outside the scope of this present paper which is to convey practical ideas, keep the reasoning informal and report on results. On the other hand, the insights provided even by such informal analysis are essential to understand how large systems may be built with impunity so that they faithfully reproduce the functional and timing characteristics of the real circuits. (These ideas will be formalised in a later paper.)

Clearly, the second version of *nand.2* above is *I/O-PAR* and *delta.2* is *I/O-SEQ*. By theorem C, the *latch* is *I/O-PAR* and does not deadlock. If the *I/O-SEQ* components merely copy input values to output, we may insert as many of them as we like into existing channels and the functional and timing characteristics will be unaltered. *I/O-SEQ* components introduce no notion of delay.

Any digital logic circuit that can be built from basic gates and wires can now be expressed trivially (although a little tediously) in occam. For instance, an ‘edge-trigered JK flip-flop with clear’ :–

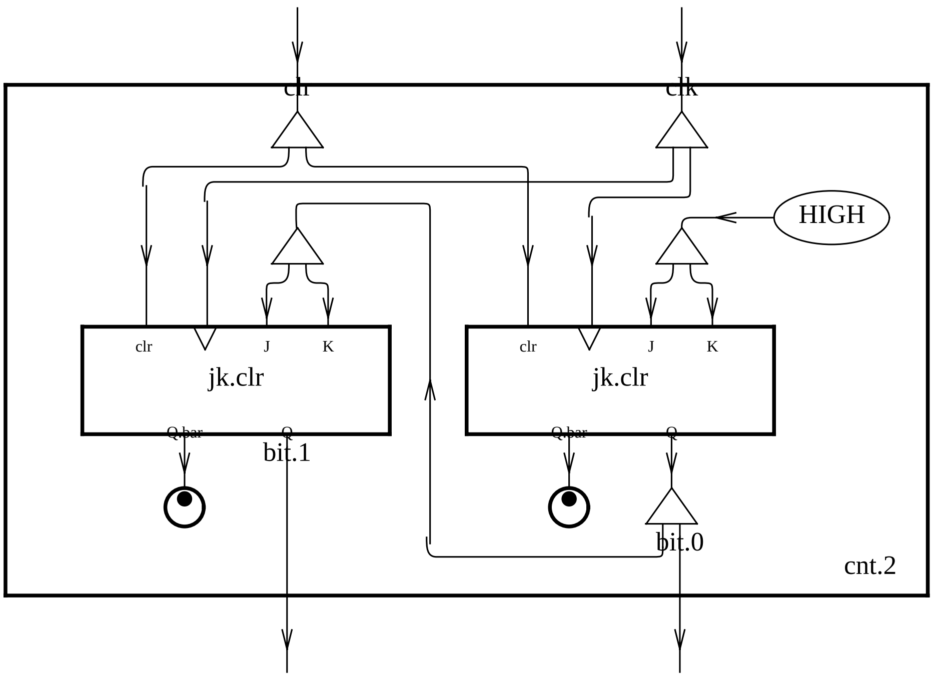


may be written :–

```
PROC jk.clr (CHAN OF INT j, k, clr, clk, q, q.b)
  ...
```

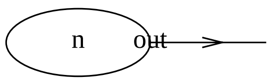
Within the body will be declared a *CHAN* for each length of wire in the circuit, followed by a *PAR* construct of instances of *delta.2* and *delta.3* processes (for the solder points) and *not*, *nand.2* and *nand.3* gates (which are implemented in the *I/O-PAR* style of the earlier *nand.2*).

Notice that, by theorem C, *jk.clr* is itself *I/O-PAR* (no input is directly connected to an output and there are no loops consisting just of bare wire) and it may be therefore be used as a component to construct higher level devices. Such hierarchical designs follow the way the real circuits are built. For instance, a ‘two-bit counter’ may be built from two *jk.clrs* and a constant *HIGH* signal source :–

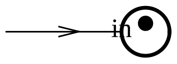


We have used two new basic devices, a constant generator and a ‘black hole’ for masking off unwanted outputs :–

```
PROC generate (VAL INT n,  CHAN OF INT out)
  WHILE TRUE
    out! n
:
```



```
PROC black.hole (CHAN OF INT in)
  WHILE TRUE
    INT any:
    in? any
```



Then :–

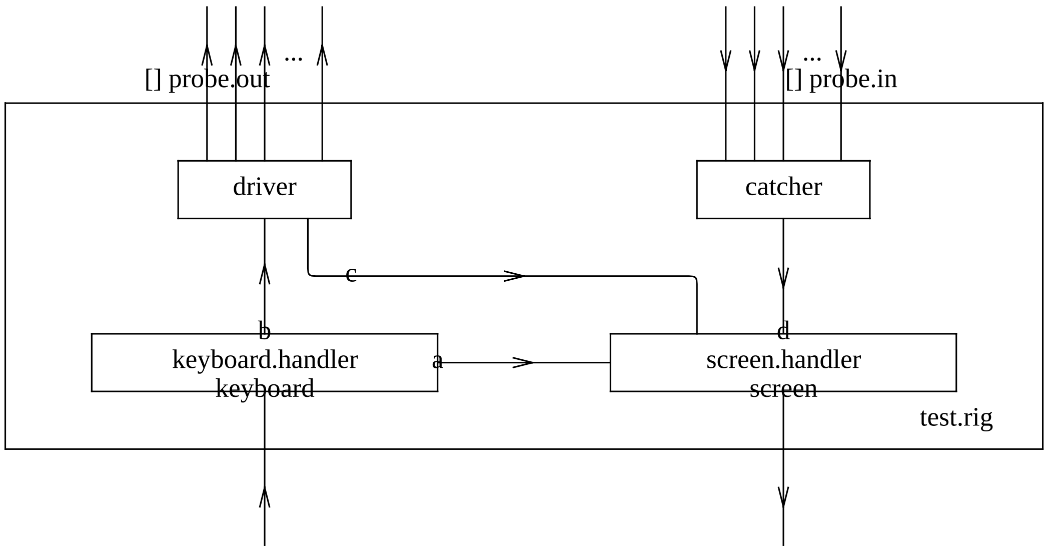
```
PROC cnt.2 (CHAN OF INT clr, clk, bit.0, bit.1)
  CHAN OF INT clk.0, clk.1, clr.0, clr.1, hi, hi.0, hi.1,
              q.b.0, q.b.1, q.0, q.0.0, q.0.0.0, q.0.0.1:
  PAR
    –– { split inputs
    delta.2 (clr, clr.0, clr.1)
    delta.2 (clk, clk.0, clk.1)
    –– }
    –– { jk 0
    generate (HIGH, hi)
    delta.2 (hi, hi.0, hi.1)
    jk.clr (hi.0, hi.1, clr.0, clk.0, q.0, q.b.0)
    delta.2 (q.0, q.0.0, bit.0)
    black.hole (q.b.0)
    –– }
    –– { jk 1
    delta.2 (q.0.0, q.0.0.0, q.0.0.1)
    jk.clr (q.0.0.0, q.0.0.1, clr.1, clk.1, bit.1, q.b.1)
    black.hole (q.b.1)
    –– }
```

When we need to modify the functionality of a device (such as dispensing with the second output from *jk.clr*), we much prefer to reuse the device intact (so that we inherit known characteristics) and add separate components to alter the behaviour. Only later, when the new design has matured, when there is a need to improve its performance and when there are widespread applications likely, might we look to integrate its implementation. Reuse of software components has obvious and urgently needed benefits, *but is seldom achieved*. We can learn many lessons from the hardware components industry. Parallel design techniques, based on the simple disciplines of occam, enable us to apply these lessons directly.

Again, by theorem C, *cnt.2* is *I/O-PAR* and may itself be incorporated in higher level circuits. We have a technique that enables us to express formally the interface (pins) of any digital circuit and its internal logic (down to the gate level) in a way which closely follows hardware design. Further, this expression will execute in a manner which closely emulates the behaviour of the real hardware.

TESTING THE CIRCUITS

An interactive *test.rig* for analysing such circuits may be built as follows :–



```
PROC test.rig (CHAN OF BYTE keyboard, screen,  [] CHAN OF INT probe.out, probe.in)
  CHAN OF BYTE a:
  CHAN OF INT b, c, d:
  PAR
    keyboard.handler (keyboard, b, a)
    driver (b, c, probe.out)
    catcher (probe.in, d)
    screen.handler (a, c, d, screen)
```

The *driver* process generates a sequence of ‘sample slices’ of the test wave-form. Each component of the slice is output in parallel through the *probe.out* channels to the input pins of the circuit under test. The driver maintains a *count* of the number of slices sent — this *count* represents the ‘real-time’ base of the simulation. The particular wave-form it generates is determined by instructions received from the *keyboard.handler*. Constant or periodic signals may be programmed for each output line. The wave-form may be generated continuously or stepped through under user-control. If this wave-form is to be monitored on the screen, a copy is sent to the *screen.handler*. However, to save needless ‘flicker’ on the display, the *driver* only sends a copy of the wave-form slice when it changes, together with a ‘time-stamp’ (i.e. the value of its internal *count*).

Because the circuit under test is *I/O-PAR*, for each input ‘wave-front’ there is, in parallel, an output ‘wave-front’. The *catcher* process inputs these fronts, maintains its own *count* (which will automatically keep in step with the *count* in *driver*) and forwards a copy of the front, with its *count* ‘time-stamp’, to the *screen-handler* when it changes (again to save ‘flicker’).

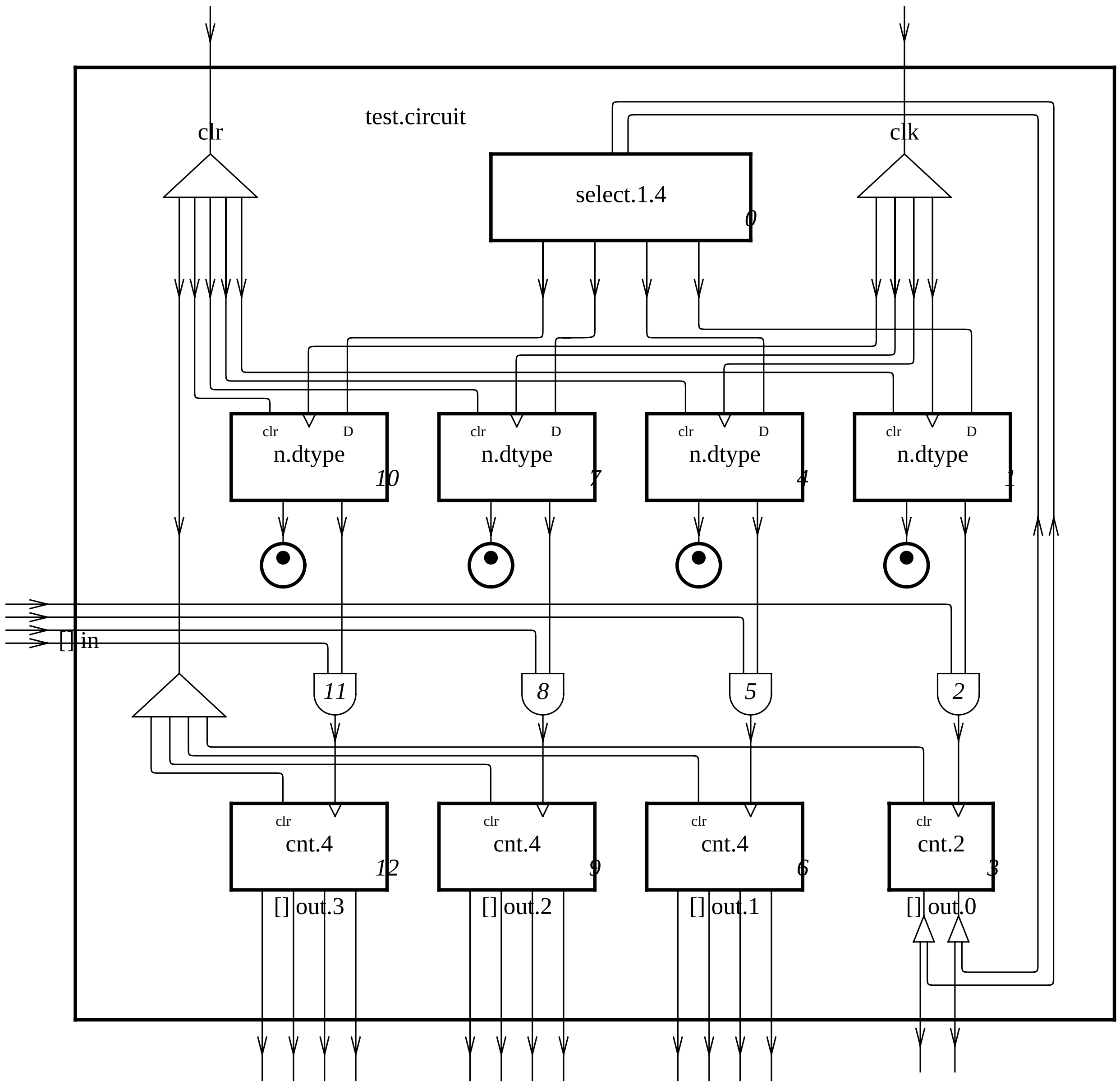
The *keyboard.handler* accepts key-strokes (or ‘mouse’-input) from the user, interprets a human-oriented command language for specifying wave-forms, forwards appropriate ‘echo’ information to the *screen.handler* and transmits coded instructions to the *driver* process.

The *screen.handler* has to accept (*time.stamp, wave.front*) records from the *driver* and *catcher* and maintain a suitable display. [In our implementation, for which we only used a standard VDU, we simply output the *time.stamp* and *wave.form* logic levels numerically. To drive a more friendly (but expensive) graphical display, only this process needs to be modified.] The *screen.handler* also has to accept (at a higher priority) echoing information from the *keyboard.handler* and provide the necessary feedback to the user for his control actions. This behaviour is very easy and elegant to express with the *PRI ALT* construct of occam.

This *test.rig* may be reused without alteration for non-interactive testing by providing two further processes which simulate ‘user-input and user-output’. The ‘user-input’ process simply obtains a (presumably large) pre-defined test-pattern from a file and forwards it to the *test.rig*. The ‘user-output’ process records the output from *test.rig* to some file for later analysis.

PERFORMANCE AND TRANSPUTER NETS

To build confidence in the correctness of these methods, the following *test.circuit* was used :–



Each four-bit counter, *cnt.4,* contains 4 *jk.clr* and 2 *and* gates, making 62 gates overall. The *cnt.2* contains 30 gates. Each 'D-type negative edge with clear' device, *n.dtype*, contains 10 gates and the 'one-from-four selector', *select.1.4,* has 6 gates. In total, the circuit has 266 gates.

Each gate is a continuous parallel process in occam. Internally, these are transient parallel processes to handle *I/O* (and synchronise). In addition, there are the further *delta* processes (also with internal parallelism) to 'solder' channels together, as well as some *generates* and *black.holes*. The total number of parallel processes varies between 700 and 2000 during the simulation. Compared with some other technologies for expressing MIMD parallelism (e.g. semaphores, signals, monitors, Ada tasking, ...), these are high levels.

The simulation was tested with a pre-defined wave-form which exercised all its anticipated properties. The results were compared with those from a commercial HI-LO simulator [Cirrus 82]. The functional behaviour and circuit reaction times were identical from the two simulations.

The performance of the occam simulation on a VAX 11/750 (running single-user on 4.2 BSD UNIX†) was approximately 5 'wave-front' cycles per second. This was just bearable, given that the longest reaction time of this circuit was five cycles. However, for more complex circuits (e.g. > 100,000 gates) with much more lengthy test patterns, this route would become impractical — especially with other users on the machine!

Running under the INMOS Occam Programming System on a Stride 440 (a single-user 10 MHz M68000 workstation), performance increased to 8 cycles per second. Cross-compiled on to a single transputer (a 15 MHz INMOS T414 processor running on a MEiKO MK014 ''Local Host'' board within a MEiKO ''Computing Surface''), performance increased to 71 cycles per second — some 14 times faster than the VAX.

Our simulation techniques pay no regard to trying to keep down the levels of concurrency. This is what makes the algorithms so easy to develop — we just follow whatever is happening in the 'real-world'. The context switch between occam processes on the VAX (Stride) are approximately 60 (35) micro-seconds (respectively). To generate one wave-front about 1000 processes need to be scheduled — i.e. the VAX (Stride) manages one simulation process every 200 (125) micro-seconds, spending about 30% of its time in switching. The (15 MHz) transputer spends only 14 micro-seconds for each process on our simulation. This implies a context switch overhead of about 7%, which is extremely (and comfortably) low considering the fine granularity of the parallel design being executed.

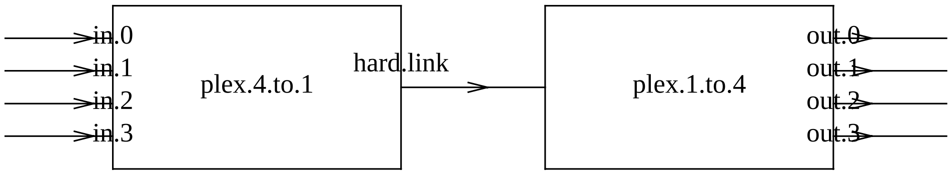
The performance of a single transputer on this kind of simulation degrades linearly with the number of gates being simulated. Allowing for the extra *delta* processes, the method requires about 180 bytes of workspace per gate. Having 3M bytes of external RAM on the MEiKO transputer board, there was sufficient room to run 60 copies of the simulation in parallel. The *test.rig* was also replicated, the keyboard input broadcast (through a *delta* process), the screen outputs gathered, checked for equality (which was, of course, logically unnecessary) and a single output stream sent to the terminal. Everything still worked and the cycle time dropped by a factor of 60 precisely. This was equivalent to a simulation of a circuit of 16,000 gates and required the management of 120,000 parallel processes.

Going in the other direction, we configured the simulation to run on a network of transputers. Because of the parallel design of the simulation, there is great flexibility as to how this can be done — almost any partition

† UNIX is a trademark of AT&T Bell Laboratories in the USA and other countries.

from everything on one transputer down to one gate per transputer is possible (although perhaps not sensible).

A minor technical problem needs to be overcome. The channel connectivity (between the partition elements that are to be placed on individual transputers) exceeds the number of physical links supported by the transputer (currently four — each one implementing a bi-directional pair of channels). For our simulation, where there is continuous synchronised data flow along *all* channels, it is trivial to insert extra processes to multiplex many channels on to the few hard links available — e.g. :–



```
CHAN OF INT hard.link:
PAR
  plex.4.to.1 (in.0, in.1, in.2, in.3, hard.link)
  plex.1.to.4 (hard.link, out.0, out.1, out.2, out.3)
```

where :–

```
PROC plex.4.to.1 (CHAN OF INT in.0, in.1, in.2, in.3, out)
  WHILE TRUE
    INT x.0, x.1, x.2, x.3:
    SEQ
      PAR
        in.0? x.0
        in.1? x.1
        in.2? x.2
        in.3? x.3
      out! x0; x1; x.2; x.3
:

PROC plex.1.to.4 (CHAN OF INT in, out.0, out.1, out.2, out.3)
  WHILE TRUE
    INT x.0, x.1, x.2, x.3:
    SEQ
      in? x.0; x.1; x.2; x.3
      PAR
        out.0! x.0
        out.1! x.1
        out.2! x.2
        out.3! x.3
```

Clearly, the use of these two processes provides transparent *I/O-SEQ* buffers on the *in.i/out.i* channels. Consequently, they may be inserted anywhere into our simulation without affecting the functional or timing data obtained by the simulation.

At present, we only have a small ‘‘Computing Surface’’ with one other board (MEiKO Mk009) which contains four (15 MHz) T414 transputers, each having 0.25M bytes of external RAM. We decided to place the *test.rig* on the original MK014 board and carve up the (computationally intensive) circuit into approximate quarters on the other four transputers (the devices were partitioned *{0, 1, 2, 3}*, *{4, 5, 6}*, *{7, 8, 9}* and *{10, 11, 12}* — see the diagram of *test.circuit* for these device numbers).

For this, we needed 16 multiplexors. Compared with the total concurrency already being managed, this was no serious overhead. Indeed, testing the system modified to include the multiplexors on a single transputer, performance only decreased from 71 to 68 cycles per second. After distribution to the five transputers, performance increased to 317 cycles per second (which corresponds to about 90% of a ‘perfect’ linear speed-up).

The largest number of channels we had to multiplex on a single link was 4. That link, therefore had to carry 1268 cycles per second. Since we were using one word for the signal level, the link was carrying less than 5K bytes per second. Link capacity is currently about 500K bytes per second (and future transputers will support 1.8M bytes per second), so we have an enormous margin on communications. Three of the transputers (each managing 73 gates) were processor bound, a fourth (managing 47 gates) had some idle time and the fifth (just managing the *test.rig*) was more under-used.

If we distributed the simulation over more transputers, performance would continue to increase so long as the capacity of the links was not exceeded (although we would need to introduce some DMA buffering — *a trivial process in occam* — as that limit was approached). About 20 gates could be managed on one transputer using just its internal 2K bytes of fast static RAM. This will operate at about 1500 cycles per second (regardless of the number of logical inputs and outputs). If we had to multiplex 10 channels to one link, this will still only require a data flow of 60K bytes per second — about 10% of capacity. In fact, one gate *could* be managed per transputer, producing about 30K cycles per second. ‘Fan-in/fan-out’ restrictions on the technology being simulated would probably mean that little multiplexing of channels would be needed. We would need to multiplex 4 channels to reach the current communication bandwidth of our links.

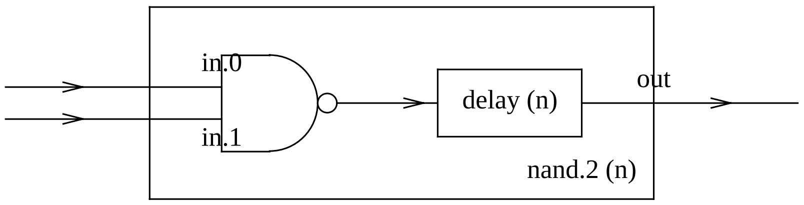
Our design is very easy to balance on a transputer network. We just have to distribute the gates per processor evenly (making use of the *plex* processes described above) and each transputer will utilise its full processing power (currently 10 MIPS) with data always being available (and absorbable) on its links. Thus, a 100,000 gate system may, for instance, be distributed over 10 transputers (giving about 2 cycles per second) or 1000 transputers (giving about 200 cycles per second).

For other parallel designs, where the pattern of communication is not so simple (e.g. where the network does not just consist of pure *I/O-PAR* or *I/O-SEQ* components), load balancing may not be so easy.

In general, we must be careful to ensure that transputers are not kept idle *either* because other transputers have not yet computed their data *or* because they are generating so much data that the link bandwidth is exceeded. In this case, it is quite possible that adding further transputers will actually slow down overall performance.

EXTENSIONS

Arbitrary length propagation delays — in multiples of the sample interval times — may be programmed into the behaviour of gates by maintaining a full ‘cyclic buffer’ of output values. The current ‘double-buffering’ described earlier represents a ‘cyclic buffer’ of length two. For simplicity and clarity, we separate the notions of computing the gate function and effecting the delay into two processes :–



where :–

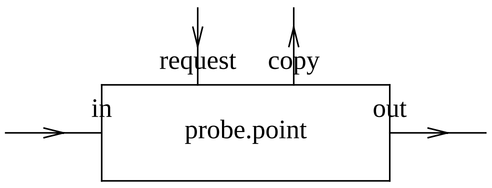
```
PROC delay (INT n, CHAN OF INT in, out)
  -- assume: n <= max.delay (some global constant)
  [max.delay] INT buffer:
  INT pointer:
  SEQ
    -- { initialise
    SEQ i = 0 FOR n
      buffer [i] := UNKNOWN
    pointer := 0
    -- }
    -- { main cycle
    WHILE TRUE
      INT x:
      SEQ
        PAR
          in? x
          out! buffer [pointer]
        buffer [pointer] := x
        pointer := (pointer + 1) \ n
    -- }
```

and where the gate process has the simple *I/O-SEQ* algorithm that was initially proposed (and labelled ‘incorrect’). Notice that *delay* is *I/O-PAR* so that, overall, this new *nand.2(n)* is still *I/O-PAR*. Note also that, from the point of view of its environment, *nand.2(1)* is indistinguishable from the earlier *nand.2*.

We are extravagantly using an *INT* (i.e. 32 bits on a T414 transputer, Stride or VAX) to represent just two logic levels. The boolean operators used in the computation of the gate functions act (in parallel) across every bit in the word. For very long pre-defined test patterns, advantage can be taken of this. Thirty-two independent wave-forms may be multiplexed on to separate bits in a word, passed through the simulation and de-multiplexed into thirty-two independent results. This will execute *at the same speed* as just a single simulation. Conventional sequential simulations, which ‘optimise’ by only computing those parts of the network where signal levels are changing, would find this difficult.

Alternatively, if we want to simulate finer detail, we could use a range of values (say 0..255) to represent sample levels of signal. Individual gates may be programmed to follow published response characteristics for particular devices to varying input signal levels (either through constant ‘look-up’ tables or, more crudely, by calculating minima or maxima). This would impose only a marginal increase in the memory and computational resources needed for each gate.

For interactive testing of the circuit, the user may wish to attach a ‘probe’ to any wire so as to inspect the signals being carried. The simulation needs to be equipped with special ‘probe points’, perhaps on every wire, to allow this :–



This *probe.point* is an *I/O-SEQ* device (with respect to the *in* and *out* channels) so as not to affect the timing characteristics of the circuit. There are two modes of behaviour. Normally, it just copies *in* data to *out*. However, if a *request* signal comes in (i.e. the user ‘touched’ this ‘probe-point’), the *in* data is copied to *copy* as well as to *out*. A second *request* signal (i.e. the user ‘removed’ his ‘probe’) causes *probe.point* to revert to its normal mode. The *probe.point* listens for *request* or *in* data by means of a *PRI ALT* construct (with *request* having priority).

A similar technique lets the user cause a ‘short’ in the circuit. The simplest method is to extend the functionality of *probe.point* to respond to a second type of *request* signal that puts it in a mode where it transmits a constant value on *out* for every *in* data received. Such a capability is very useful in trying to diagnose faults in actual circuits or, at the design stage, for ensuring that dangerous outputs could not be produced should a fault occur.

Finally, we note that while it is quite trivial to express the circuit designs in occam, it is also very tedious. The numerous extra channels that have to be declared, because ‘soldering’ requires an active process, complicate the final code and mistakes can easily be made. For ease of use, it would not be hard to generate the code automatically from a higher level (perhaps graphical) description of the circuits. Similarly, the configuration information for distribution over transputer nets, together with the necessary extra multiplexors/de-multiplexors, could be generated automatically following some interactive discussion with the tester (which might be limited to ‘‘how many gates per transputer?’’). Techniques will be reported in a later paper whereby the circuits may be built at run-time and without introducing a level of interpretation that would slow down the simulation.

CONCLUSIONS

We have demonstrated how the model of parallelism supported by occam allows us to emulate very directly the behaviour of certain ‘real-world’ objects (digital logic gates). We can then construct arbitrarily complex systems of these objects simply by following the way they are constructed in reality. Although we have reported here on one particular type of system, we believe that these techniques lead to simple ‘object-oriented’ design methods that have general application. Further, because the parallelism inherent in the system is retained in the design, the designs are very easy to distribute to multiple processors.

Traditional sequential methods for implementing such designs require extra layers of detail (global data-structures, scheduling, ...) that make both formal and informal reasoning about them very difficult. It is also very hard (probably impossible) to extract the parallelism back from the sequential code in order to take advantage of highly parallel hardware.

In the study reported here, we have shown how to emulate (very large) digital logic circuits. The system may be distributed on to as large a network of transputers (up to the number of gates in the circuit) as the tester deems cost-effective, with linear improvement of performance. Functional behaviour and timing characteristics of the circuits (which, for instance, demonstrate the presence of ‘race hazards’) are faithfully reproduced. We have provided a user-interface which allows the tester many of the facilities of a ‘logic analyser’ working on the real devices. Future tools will allow the user interactively to design circuits and examine their properties at a much higher level (e.g. through graphical displays).

Finally, we note that no effort has been made to optimise on either memory (we use 32 bits for boolean logic), processing (no attempt is made to suspend the simulation of gates whose input signals are steady; no attempt is made to remove parallelism from the source code expression of the algorithm) or communications (no clever ‘run-length encoding’ tricks). We are convinced that for simplicity and confidence in design, the commitment to parallelism — at least in the prototype — should be total. With networks of transputers, where memory, processing and communications resources can be equally balanced, such ‘prototypes’ may make excellent ‘production quality’ systems. (For a rather different kind of application where similar conclusions are reached, see [Welch 87b]).

ACKNOWLEDGEMENTS

These ideas were originally developed while the author was undertaking an Industrial Fellowship sponsored by the Royal Society and the SERC at GEC Avionics Limited (Rochester) [Welch 85]. I wish to thank, in particular, Keith Washington and Ian Roscow of the Automatic Test Equipment division (of GEC Avionics) for many useful discussions. The practical work was done at the University of Kent, which provided the ‘‘Computing Surface’’ and excellent support. Of course, many thanks are owed to the designers of the occam/transputer concept for showing how elegantly and efficiently concurrent systems can be made to work.

REFERENCES

[AJPO 83]	Ada Joint Program Office: Ada Language Reference Manual; 1983.
[Bowler 87]	K.C.Bowler et al.: *An Introduction to Occam 2 Programming*; Chartwell-Bratt (ISBN 0-86-238-137-1); 1987.
[Burns 88]	A.Burns: *Programming in Occam 2*; Addison-Wesley (ISBN 0-201-17371-9); 1988.
[Cirrus 82]	Cirrus Computers Ltd.: *HI-LO 2 Users Manual*; 1982.
[Dowsing 85]	D.Dowsing: *Simulating Hardware Structures in Occam*; Software & Microsystems, Vol. 4, No. 4, pp. 77-84; August, 1985.
[Hoare 83]	C.A.R.Hoare: *Communicating Sequential Processes*; Prentice-Hall, 1985.
[INMOS 83]	INMOS Ltd: *Occam Programming Language Manual*; Prentice-Hall, 1983.
[INMOS 84]	INMOS Ltd: *Transputer Reference Manual*; INMOS Ltd., 1000 Aztec West, Almondsbury, Bristol, BS12 4SQ; 1984.

[INMOS 87] INMOS Ltd.: *Occam 2 Reference Manual*; Prentice-Hall (ISBN 0-13-629312-3); 1987.

[Kerridge 87] J.Kerridge: *Occam Programming — a Practical Approach*; Blackwell Scientific Publications (ISBN 0-632-01659-0); 1987.

[Jones 87] G.Jones: *Programming in Occam*; Prentice-Hall (ISBN 0-13-729773-4); 1987.

[May–Shepherd 85] D.May and R.Shepherd: *Occam and the Transputer*; Concurrent Languages in Distributed Systems; North-Holland, 1985.

[Pountain–May 87] R.Pountain and D.May: *A Tutorial Introduction to Occam Programming*; BSP Professional Books (ISBN 0-632-01847-X); 1987.

[Roscoe–Hoare 86] A.W.Roscoe and C.A.R.Hoare: *The Laws of Occam Programming*: Technical Monograph PRG-53, Oxford University Computing Laboratory, Programming research group, 8-11 Keble Road, Oxford OX1 3QD; 1986.

[Tamsley–Dow 82] J.Tamsley and P.Dow: *A Tutorial Guide to SPICE 2F.1*; Department of Computer Science, Edinburgh University; October, 1982.

[Welch 85] P.H.Welch: *The Effect of New Real-Time Software Engineering Methodologies on Marconi Avionics — Final Report*; Royal Society/SERC Industrial Fellowship, reference number B/IF/43; July, 1985.

[Welch 86] P.H.Welch: *A Structured Technique for Concurrent Systems Design in Ada*; Ada: Managing the Transition; Proceedings of the Ada-Europe International Conference, Edinburgh, May 1986, pp. 261-272; Cambridge University Press, 1986.

[Welch 87a] P.H.Welch: *Parallel Processes as Reusable Components*; Ada: Components, Libraries and Tools; Proceedings of the Ada-Europe International Conference, Stockholm, May 1987; Cambridge University Press, 1987.

[Welch 87b] P.H.Welch: *Managing Hard Real-Time Demands on Transputers*; Proceedings of the 7th Occam User Group Technical Conference and International Workshop on Parallel Programming of Transputer Based Machines, (ed.: T. Muntean), LGI-IMAG, Grenoble, FRANCE; September 1987.

[Welch 88] P.H.Welch: *An Occam Approach to Transputer Engineering*; Proceedings of the 3rd. Conference on Hypercube Concurrent Computers and Applications, Pasadena, California, U.S.A. (January 19-20, 1988), ACM Conference Proceedings, (*in press*).