Design and modelling of a high performance differential bipolar self-timed microprocessor

R. Kelly
L.E.M. Brackenbury

Indexing terms: Self-timed systems, Micropipeline, Bipolar technology, Processor performance, Modelling language Verilog

Abstract: Current interest in self-timed systems is motivated by the area, power and design effort required for the global clock of VLSI synchronous designs. A self-timed datapath, based on the ARM processor, using 'micropipeline' control techniques has been developed for a newly updated high performance differential bipolar technology. The paper describes the architectural model produced to verify the correctness of the prototype design, and the use of the model in evaluating and enhancing the processor performance. Self-timed design comprises independent blocks whose operation depends solely on input data and unit availability. The modelling of the dynamic behaviour of blocks and the control structures required are presented. These illustrate how easily and well the self-timed operation is mapped onto the Verilog modelling language. Benchmark results on the processor indicate a factor-of-two performance improvement over a CMOS version. The system state at a particular instant is difficult to determine and the effects of interactions between modules are difficult to quantify. The use of the model to explore design changes, particularly to the buffering structures, is presented. This allows the design to be 'tuned' to the technology. It also enables a better understanding of total system behaviour.

1 Introduction

Renewed interest in self-timed systems has arisen as the problems associated with the design of the global clock used in synchronous systems are becoming severe. Currently, significant silicon area and design effort are required for clock generation and distribution to maintain skew within acceptable limits. These factors become progressively more difficult as feature sizes shrink.

IEE Proceedings online no. 19971600

Paper first received 24th March and in revised form 18th August 1997

R. Kelly is with ICL (UK) Ltd., Wenlock Way, West Gorton, Manchester M12 5DR, UK

L.E.M. Brackenbury is with the Department of Computer Science, University of Manchester, Oxford Road, Manchester M13 9PL, UK

Self-timed systems comprise independent modules which communicate when data is ready; there is no global clock. As a result, a self-timed design has the potential for reduced area and power consumption relative to its synchronous counterpart. Furthermore, higher performance can also result since synchronous systems need to be designed for worst-case performance whereas self-timed designs exhibit average-case performance.

The framework used for self-timed design is that of Sutherland's 'micropipelines' [1]. This is an elastic, bounded delay, event driven pipeline where communication between stages consists of a bundle of data accompanied by locally produced handshake control signals which control the flow of data. This approach, rather than a delay insensitive model, is adopted for its relative simplicity which minimises power and area. The feasibility of using this approach for designing a fully operational self-timed CMOS microprocessor based on the ARM architecture has already been demonstrated [2]. A follow-on project is aimed at transforming the self-timed CMOS microprocessor, AMULET1, into the recently updated high performance differential bipolar technology manufactured by GEC Plessey Semiconductors (GPS).

The technology transfer would demonstrate the applicability of a 'micropipelines' framework to technologies other than CMOS and was also expected to demonstrate a performance improvement due to the inherently faster speed of the underlying technology.

Due to the prototype nature of the architecture, design methodology and fabrication process, an essential stage in the technology transformation was the development of a model for the self-timed microprocessor in the target technology. This would verify that the proposed design was functionally correct, enable the design to be 'tuned' to the target technology and act as a tool with which to investigate architectural alternatives.

2 Multilevel differential current mode logic

The target technology is multilevel differential current mode logic (MDCML) using the newly updated process of GPS. It has complimentary inputs and outputs, allowing common mode noise to be rejected. High speed results from not saturating transistors and using signal swings of only 160mV.

The logic is arranged in a current switch tree of up to three levels and operates from a 3V supply. This number of levels represents the best compromise between functionality, area and power [3].

Fig. 1 shows a 3-input AND gate. The inputs at the top level, level 3, are compatible with output levels making level shifters necessary to drive inputs at levels 1 or 2. One current path always exists between either the true or the inverse output and ground causing this output to be pulled low and the other to remain high; the output pulled low corresponds to the path where all inputs are at a high level. This means that every input combination needs to be explicitly defined by the circuit.

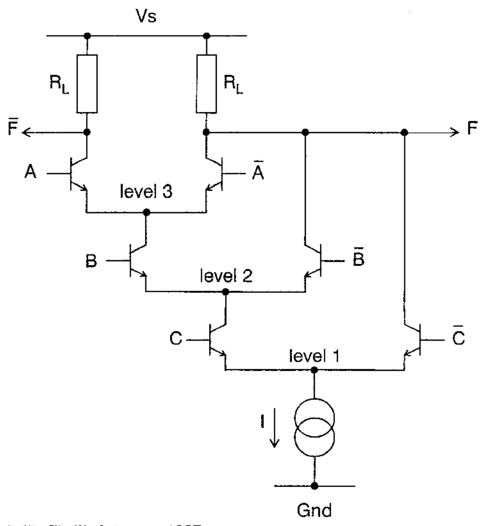


Fig. 1 *MDCML 3-input AND gate*

Multiple switch levels enable a complex function to be accommodated within a gate with a single current source. This includes a 4-to-1 multiplexer, a transparent latch with reset (this copies the output to input when its enable is active), or *any* function of three variables. The use of differential signals removes the requirement for inverters but extra silicon area is required for the differential routing of all signals. Extra power and area is also required for the level shifters. Since designing in terms of the highest gate functionality minimises the area, propagation delay and power, this approach characterises MDCML design at the gate level.

3 The Verilog modelling language

Verilog [4] is the modelling language adopted by GPS for its MDCML designs and this section gives an overview of the features used to model the self-timed processor.

The processor is defined as a set of hierarchically instantiated modules. These modules are defined at a variety of levels, ranging from behavioural descriptions at the algorithmic and register transfer levels to structural specifications at the gate level. Twenty-six available Verilog primitives (such as **and**, **or** etc.) are used to define the structural models and an example is given in the following Section.

The syntax of Verilog for behavioural modelling is similar to that for the 'C' programming language. However, statements can be grouped together in a sequential or concurrent block. The former is indicated by the **begin** and **end** keywords and statements within these keywords execute sequentially; control passes out of the block when the last statement is executed. The beginning and end of concurrent blocks is indicated by the **fork** and **join** keywords. Statements within these keywords execute in parallel; control passes out of the block when all statements have executed.

In a self-timed system, there is no clock and actions are initiated by the occurrence of event(s). The **initial** statement, which is executed only once at the start of simulation time, is used to initialise signals and internal module variables. Thereafter, the basic Verilog construct used to define the behavioural model of a self-timed module is one or more

always @ (event(s))

<block statement|statement>

The **always** @ construct executes the block statement whenever the positive or negative edge of the specified event(s) occur. This is of particular use in the MDCML processor where two-phase signalling is used so that every transition on a control line signals an event. Positive and negative edges can be selected for initiating actions using **always** @(posedge)/**always** @(negedge). In the MDCML processor modelling, they are used within (the few) modules where the two-phase timing needs to be converted to four-phase.

always @ indicates an independent flow of activity enabling the system to be modelled as a set of independent, intercommunicating processes. In the self-timed processor model, all behavioural functionality is specified within **always** @ statements, as illustrated in the code for the signalling protocol in Section 6.

Apart from the **always** @ timing control, the #<time units> <statement> is used in the processor model for local timing within a module. It specifies the time duration between the activity flow reaching the statement and the time at which it is executed. A useful feature of Verilog behavioural modelling is the ability to include monitoring or error messages for the user. This greatly adds to the observability of the model and assists with debugging of the design.

4 Gate modelling

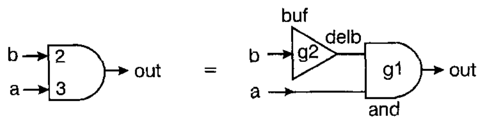
The propagation time of MDCML gates varies according to the level at which an input is applied. When typically loaded, the difference between the top and bottom level delay is significant. This gives the designer an additional design parameter in formulating designs. For example, information can be transmitted on the lower levels in noncritical paths, or where extra delay is required to ensure that control signals arrive after valid data. In a similar way, signals on critical paths or frequently used paths are placed at the highest level to optimise performance. However, where more than one input would benefit from level-3 placement, then it is necessary to consider the expected sequence of signals to optimise performance.

In implementing a Verilog model for each gate used in the model, HSPICE circuit simulations were first run. These used component models based on measurements made of test samples from the prototype line. The HSPICE simulations were used to establish the propagation delay at each level, to investigate the effect of output loading and input drive variations, and to observe the skew between true and inverse outputs.

The results showed that under all driving conditions, the observed skew between true and inverse outputs was negligible. This enabled the Verilog modelling to be reduced to just defining the operation of the true phase of a signal.

The investigation of loading and drive effects showed that output loading dominated. The HSPICE equivalent load which reflects increased propagation delay due to gate loading was found empirically to be typically equal to four level-3 BUFFER loads. This has been incorporated into each Verilog gate model. When loading is taken into account, the propagation difference between levels is significant, with the typical delay from level 2 and level 1 being, respectively, 1.5 and 2 times that of the level-3 propagation time.

Twenty structural models have been used for the gates in the processor and Fig. 2 shows the code for a two-input AND gate. The MDCML AND gate is modelled using the **and** (AND) and **buf** (BUFFER) primitives with user specified delays. Input a is applied at the top level and input b at level 2.



```
'timescale 1ps/1ps
module and2 (out, a, b);
'define and2A_delay 263
'define and2B_delay 424

input a, b;
output out;
wire delb;

and #('and2A_delay) g1 (out, a, delb);
buf#('and2B_delay - 'and2A_delay) g2 (delb, b);

endmodule
```

Fig. 2 *Modelling a two-input AND gate*

The model uses an AND gate having a propagation delay of 263ps and a BUFFER gate which represents the additional propagation delay experienced by the lower level input.

In practice the **assign** statement is used to replace the instantiations of (the **and** and **buf**) primitives in the structural models. This increases the simulator performance by directly assigning values to outputs of combinatorial primitives based on the current values on the inputs.

5 Control

The protocol uses a two-phase handshake signal convention and its operation between two blocks is summarised in Fig. 3.

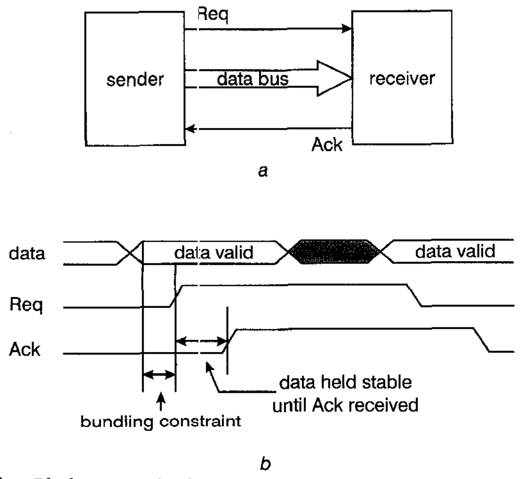


Fig. 3 *Block communication*
a Bundled data model
b Timing constraints

Data is accompanied by two control wires and *transitions* on these indicate events between blocks. Provided the receiver has acknowledged the receipt of the last bundle of data, the sender is free to send further data to the receiver. When valid data is assembled, Req changes state to inform the receiver that new data is present. The sender now holds the data lines stable until the receiver accepts the data and acknowledges this with an Ack transition. An essential requirement for the correct operation of the protocol is that, regardless of the path taken, valid data arrives prior to the Request transition at the receiving module; this is known as the 'bundling constraint'.

In practice the coordination of activities between and within many modules of the processor is more complex than indicated in Fig. 3. The most typical examples in the design are when senders compete for the use of a bus, or when there is a choice of control path which is dependent upon internal conditions, or when a control transition needs to be converted to a level.

At the gate implementation level, the transition protocol and the coordination of activities within the processor require a set of control elements. These are summarised in Fig. 4 and apart from the XOR gate need to be user defined.

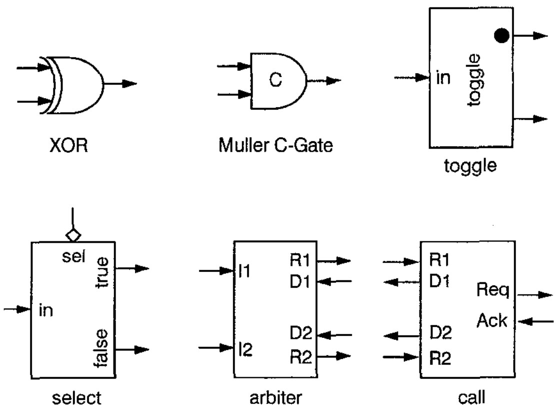


Fig. 4 *Transition control elements*

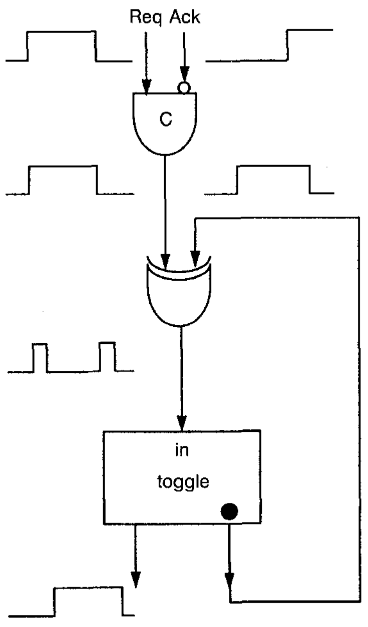


Fig. 5 *Example of use of the control elements*

The transition protocol described in Fig. 3 can be implemented with the 2-input Muller C-gate. This performs an AND function of two events (transitions) and is extensively used in the control. The element contains storage which is placed at the bottom level so that the inputs can be placed at levels 2 and 3 to minimise path delays. The XOR gate is used to merge events as its output changes every time an input transition occurs. It can be used when inputs are guaranteed not to be simultaneous. The TOGGLE element steers an input transition alternatively to the two outputs; a positive input transition causes a transition on the 'dot' output while negative input transitions drive the 'blank' output. A TOGGLE is frequently used in conjunction with a Muller C-gate and an XOR gate to convert each input transition to a positive pulse which can be used to enable a transparent latch, as shown in Fig. 5. The transition on Req is converted to a positive pulse on the XOR output whose width is equal to the propagation delay through the XOR and TOGGLE elements.

The SELECT element directs the input transition to the output selected by the control input *sel*. It is used where there is a choice of actions dependent on internal conditions. The remaining two elements are used where it is necessary to synchronise activity within the processor. The ARBITER selects between one of two asynchronous (transition) inputs. If it is unable to choose, because both arrived simultaneously, then neither is selected until the resulting metastability is resolved. The CALL element is used where two mutually exclusive sources request access to the same module. Since the ARBITER outputs are mutually exclusive, the ARBITER is normally used in conjunction with the CALL element. The input selected by the ARBITER is passed to the CALL element and is used to control any multiplexer selection while the CALL element issues a Req to the receiving module.

The more complex control elements, namely the TOGGLE, SELECT, ARBITER and CALL blocks, can be constructed from a combination of simpler structures comprising transparent latches, XOR gates, Muller C-gates and in the case of the ARBITER a comparator. Fig. 6 of the CALL element illustrates the typical complexity of these control structures

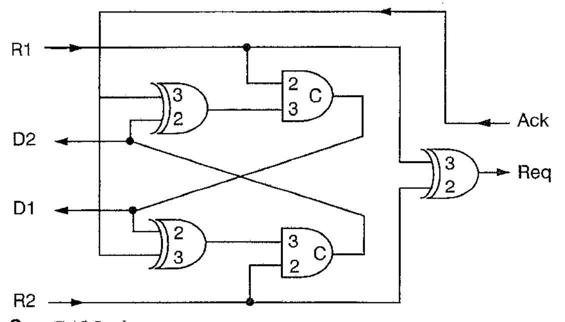


Fig.6 *CALL element*

The allocation of inputs to the different gate levels is performed on the basis of the expected sequence of signals. In Fig. 6, the incoming transition request R1 or R2 will always precede the acknowledgement Ack from the called module. Thus the later event, Ack, is placed to propagate through the top level of the XOR and Muller C-gates to minimise the time to generate the done transition (D1 or D2). At a higher level, it is advantageous to connect the most frequently occurring calling source to R1 since the propagation delay to the Request transition (Req) to the called module is less than for R2. Similar considerations determine input allocations in the other elements.

Apart from the XOR gate, the control elements are modelled behaviourally based upon their constituent transistor composition and the allocation of input signals to levels.

Self-timing encourages a modular design style where the design is partitioned into independent, concurrent blocks. This approach has also been adopted for the processor timing and control. In the MDCML processor, control comprises approximately one-third of the overall design and is partitioned into several modules which reflect the organisation of the major data blocks within the processor. Control modules are defined at the gate level using the control elements and conventional gates previously described. Programmable logic arrays are also used in many places to convert a set of input signals to the form required by a data block.

6 Modelling data blocks

The modelling of the dynamic behaviour of the data blocks of Fig. 3 is given in Fig. 7. All control signals are assumed to be initially at a low level. A time delay of one unit is used to ensure that the sender's Request line occurs after valid data is placed on the bus. The handshaking is initiated by the Req of the first data sent. Thereafter Req and Ack alternate.

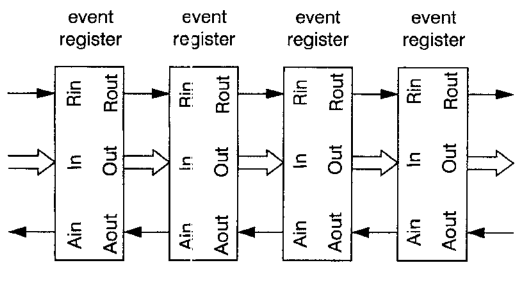
```
   sender...                       receiver...
initial
 begin
  prepare_initial_data;
  #1 Req = 1;
 end
always @ (Ack)                  always @ (Req)
 begin                           begin
  prepare_new_data;               consume_data_value;
  #1 Req = ~Req;                  Ack = ~Ack;
 end;                            end;
```

Fig.7 *Dynamic block behaviour*

Consumption of data in the receiver normally involves capturing the data in its input register when this register becomes free. Newly prepared data in the sender is usually valid once the sender loads its output register. Its output register is released once the Ack signal arrives from the receiver. It should be noted that both blocks operate at their natural rate when data is available. This is another advantage of a self-timed system over a clocked design.

A self-timed approach encourages modularity in design and concurrent operation. The block structure outlined in Figs. 3 and 7 lends itself to a 'micropipeline' implementation where the design is partitioned into pipeline register stages, termed event registers, without or with intervening combinatorial logic, Fig. 8. In the former case, which shows a 4-stage FIFO buffer, the data is accompanied by the Request and Acknowledge signals while in the latter case, the Request signal needs to be delayed until the data output from the combinatorial logic is valid. In practice this delay is usually best implemented by an additional data bit which experiences more propagation and line delay through the combinatorial logic than any other bit.

The event register comprises transition latches. A transition on Rin is a request to capture the input data while a transition on Aout signifies that the held data is no longer required and the Register is free to enter the transparent or *pass* state.



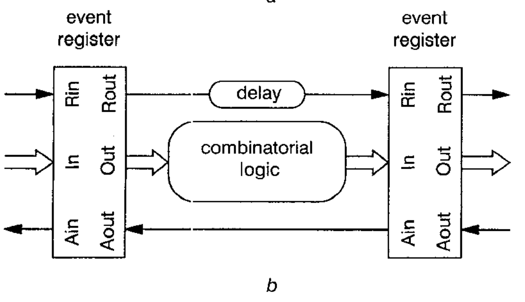


Fig. 8 *Micropipeline structure*
a Without processing
b With intervening processing

32-bit event registers are used extensively throughout the self-timed MDCML design. They provide the basic mechanism by which blocks are connected and also (as FIFOs) provide buffering between blocks to even out the flow of data. In the CMOS AMULET1 design, event registers are implemented using conventional transparent latches with control logic added to convert the input control edges to levels which clock the latch in the normal way. This was chosen in preference to the Capture–Pass transition latch on the basis of both speed and area [5].

However, in bipolar technology, the structure of the MDCML gate leads to an efficient and direct implementation of a transition-controlled storage element in a single 4-to-1 multiplexer gate, Fig. 9*a*.

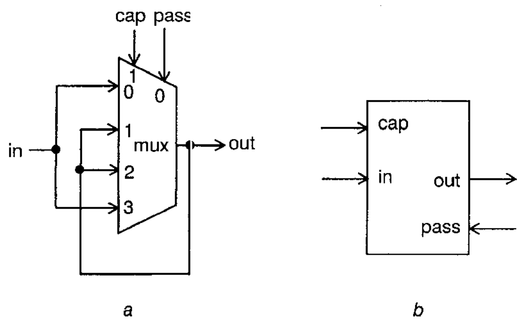


Fig. 9 *Capture–pass storage element in MDCML*
a Logic
b Symbol

The data inputs to the multiplexer are placed at the top level with the control signals, Capture and Pass, placed on the lower levels of the gate. This assignment of signals, with control placed at lower levels than the data, is used throughout the MDCML design to provide an inbuilt timing margin in order to meet the bundling constraint. Capture and Pass signals alternate. When the Capture and Pass states are the same the element is transparent, while it stores data when they are not equal. To provide the event register of Fig. 8, a Muller C-gate needs to be included with the storage elements as shown in Fig. 10.

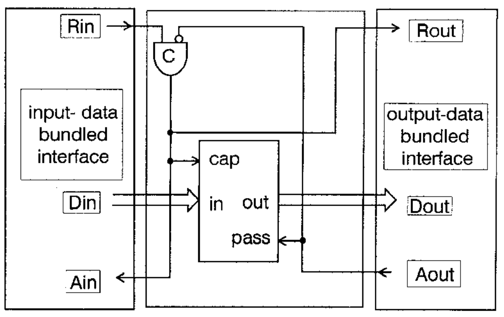


Fig. 10 *Event register*

The Muller C-gate only generates an output transition when a new request is present and the succeeding stage has accepted the currently held data. The timing to meet the bundling constraint is inherent in the structure, being provided by the propagation delay of the Muller C-gate. If Rin and Din arrive simultaneously and the event register is free to capture this data, the data immediately passes through to Dout on the fastest propagation path since the register is in the transparent state. The capture transition is then delayed by the Muller C-gate guaranteeing the capture of correct data in all circumstances and that Dout precedes Rout.

When Aout is generated, it returns the register to the pass state allowing Din to pass to Dout. Even if a new request is waiting on the input at this time, Capture is again delayed by the Muller C-gate so that the output bundling constraint is met. Since the bundle timing can be smaller when Aout is returned, Aout is placed on level 2 of the Muller C-gate.

In practice, further timing margin at the bundled interfaces is included as the Capture and Pass signals for a 32-bit register require buffering.

```
          :
pass=1;

always @ (in)
  if (pass) #(`in_out) out=in;

always @ (Rin)
  if (pass)
    fork
      #(`Din_Dout) out=in;
      #(`Rin_Rout) Rout=~Rout;
      #(`Rin_Ain) Ain=~Ain;
      pass=0;
    join

always @ (Aout)
  begin
    pass=1;
      if (Rin!=Aout)
        fork
          #(`Aout_Dout) out=in;
          #(`Aout_Ain) Ain=~Ain;
          #(`Aout_Rout) Rout=~Rout;
          pass=0;
        join
      else #(`Aout_Dout) out=in;
  end
          :
```

Fig. 11 *Functional code for the event register*

Fig. 11 shows the operational part of the behavioural model for an event register based on the underlying MDCML transistor structure. Because the relative arrival of Rin and Aout is unknown, an internal variable *pass* is required to indicate the pass/capture state of the register. It is assumed to be initialised to '1' (transparent). The time delays (#) used are specified before initialisation and define the delays between inputs and outputs. These have been derived from HSPICE circuit simulation of the structure and show an in-built bundling margin of 1.2ns at the output interface.

The code illustrates the use of the **always @**, **fork** and **join** constructs to support independent and parallel activity within the module. These are used in a similar way in the behavioural modelling of the combinatorial logic blocks between event registers. These logic blocks are usually substantial, e.g. a multiplier, an address incrementer, an ALU, etc., and tend to be written at the algorithmic level. Although no detail is implied in this model about the underlying gate structure, this is usually known and used to define the block operation time.

7 Self-timed ARM architecture

The Advanced RISC Machine (ARM) microprocessor has a 32-bit load/store RISC architecture with a three-address register-oriented instruction set [6]. The highest level of design of the MDCML self-timed ARM closely follows that of the AMULET1 and the processor organisation is shown in Fig. 12.

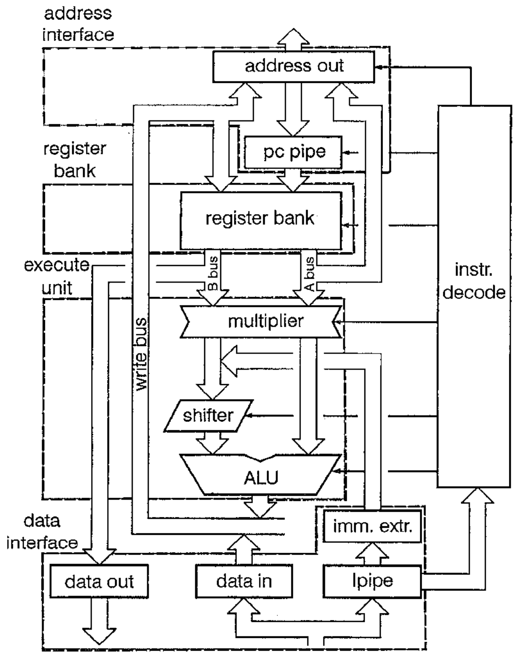


Fig. 12 *Asynchronous processor organisation*

Four main units of the processor can be identified in Fig. 12: the register bank, the execution unit, the address interface and the data interface.

The execute pipeline comprises the register bank followed by the execution unit. The register bank contains the processor's general purpose and status registers. In addition, the register bank incorporates a lock FIFO [7]; this detects data dependencies and prevents a register from being read if it awaits an update by a previous instruction. It also enables read and write operations to proceed asynchronously and concurrently without the need to arbitrate.

The execute unit contains the processor's computational logic. It comprises a multiplier, shifter and arithmetic and logic unit (ALU). Normally, instructions read one or two operands from the register bank. These pass to the multiplier which performs a bypass operation or an autonomous 2-bits-at-a-time multiplication forming partial sum and partial carry outputs for the ALU. The shifter is connected to the B bus allowing a register bank operand to be shifted. The ALU performs all other logical and arithmetic operations; its adder is a ripple carry adder with data-dependent completion signalled when the carry propagation has ceased. The ALU result is normally written back to the register bank via the write bus.

The interface between the external memory system and the processor consists of the address and data interfaces. The former handles all addresses sent to memory. It also generates addresses for the autonomous instruction prefetching and the multiple load and store orders. The values of the program counter are kept in a FIFO buffer (PC pipe) within the address interface and its current value is available to the rest of the system as a general purpose register.

Data to and from the memory passes through the data interface. Incoming instructions are stored in the Ipipe with any immediate value extracted by the imm. extr. block. Incoming data is manipulated as required by the data in block before being passed to the register bank. Data to be written to memory passes to the data out block which comprises a FIFO buffer and byte replication logic. Data written to memory needs to be synchronised with the address and control information in the address interface before the request is sent to the memory as a single bundle.

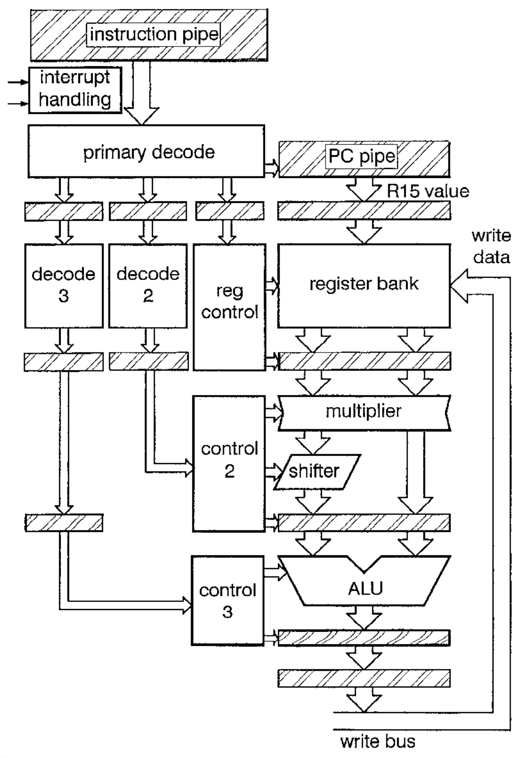


Fig. 13 *Block organisation of execute pipeline*

8 Architectural modelling

As an aid to developing a full model of the processor, the four units were individually designed down to the RTL level. These were then behaviourally modelled in terms of their constituent blocks and tested. As an example, Fig. 13 shows the main control and data blocks of the execute pipeline.

Shaded blocks represent event registers. Each data block has an associated control block. The primary decode block decodes the instruction and this is then appropriately latched together with the value of PC. The accessing of the register bank now proceeds in parallel with the decoding required for the later parts of the execution. Although event registers are placed in parallel in Fig. 13, they operate independently with synchronisation only performed between the control block and the data to which it is applied. Thus control 2 coordinates with the multiplier and shifter data and control 3 with the ALU data.

The individual unit is defined as a single module with an input/output signal interface which connects it to the other units of the processor and to the processor input/output pads. The unit module invokes a hierarchy of module calls to the blocks of the unit, typically to a depth of three. Each hierarchical level contains instantiations of the primitive gates and control elements for local timing and control. The unit module calls its associated control blocks and for large units calls a module which in turn contains calls to all the datapath modules including any event registers. As previously stated, the bottom level description for control modules is normally in terms of primitive gates and control elements while the bottom level for data modules is usually behavioural. Modules are linked through the hierarchical levels by their input/output interface.

Although traditionally associated with synchronous systems, Verilog proved to be a supportive environment for the modelling and testing of modules and units of the self-timed MDCML design. The modular hierarchical structure of Verilog is well matched to the modelling of a self-timed system comprising many inter-communicating independent modules, and the self-timed operation maps well onto its control constructs. In particular, a high degree of concurrency is supported by the **fork** and **join** compound statements which allows nondeterministic ordering of the notionally parallel execution of individual statements. In addition, the **always** @ construct mapped well onto the transition signal protocol enabling many threads of execution to be simultaneously active throughout the model.

Verilog also enables code to assist with fault finding to be included in the modules. In particular, a bundle checker has been used on data buses to check that there is a safe margin (typically 1ns with this technology) between the Request In transition and the data arriving; this has enabled modules with insufficient tolerance to be identified and modified.

In general the units yielded a higher individual performance than their CMOS counterparts due to their higher inherent speed. However, in the places where advantage is taken of the CMOS technology to implement functions such as a barrel shifter, or a wide wired-OR, both of which can be implemented in principle in a single stage using pass transistors, then MDCML technology is at a disadvantage because conventional gates have to be used. This is apparent in the MDCML ALU where although the datapath operation is faster, the formation of the zero condition flag from the result slows the overall operation time down as it requires a multistage gate network to implement the 32-bit wide NOR function.

The complete processor model consists of a single module which instantiates the major functional units. The processor is itself contained within a module which also consists of a simple memory management unit (MMU) and an external memory which supports the transition signalling protocol. The connection to the external environment is shown in Fig. 14.

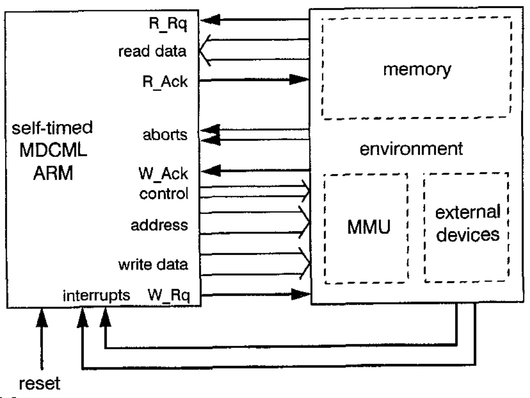


Fig. 14 *Processor connection to its environment*

On initialisation, the memory model is loaded with the Verilog equivalent of the ARM instructions to be run. Following Reset, the processor self-starts by fetching and executing the instruction at line 0 in memory.

The self-timed MDCML processor model comprising a source file of over 100 Kbytes was tested by running the validation suite of test programs produced and used by Advanced RISC Machines (ARM) Limited for their synchronous processors. Confidence in the correctness of the model was achieved when all internal tests and the test suite passed.

The performance of the model was assessed by running the Dhrystone benchmark [8]. This does not compute anything meaningful but is syntactically and semantically correct. Furthermore it has a representative mix of instruction types which include a typical mixture of operator types, operand types and operand locations. Its results are dependent on factors such as compiler influence, the timing measurement method and cache interaction which makes comparisons between different processors difficult. However, it provides a useful metric for the relative evaluation of design alternatives on the MDCML processor and also enables a comparison with the CMOS AMULET1 design.

The benchmark yielded a figure of 43500 Dhrystones per second for the processor based on a 5ns external memory and typical figures for the underlying trench-isolated 1.2μm bipolar technology. Since the comparable figure for the 1μm AMULET1 design is 20500 Dhrystones per second [9], this leads to the expectation that an MDCML processor will exhibit at least a factor of two improvement in performance over a comparable CMOS design.

9 Performance investigations

The effect that the interaction of the many intercommunicating self-timed modules of the system has on the overall performance is not well understood. This makes it difficult to predict the effectiveness of possible design changes. The processor model is a valuable tool in assisting the exploration of the dynamic behaviour of the system and enabling the assessment of design changes.

In order to gain a better understanding of the factors influencing the performance and their contribution to the efficiency, various aspects of the MDCML processor were examined.

9.1 Effect of nonsymmetrical propagation delay

Changing the assignment of input levels of control elements alters the system timing. To assess this, inputs were swapped on the XOR and Muller C-gates; these dominate the control elements used. The XOR gate generates an output event for every input event and in the original benchmark run, the input that changes most is assigned to the top (fastest) gate level. In the Muller C-gate, an output event is only generated when both input events have arrived leading to the placement of the input event which arrives later the most often being placed on the fastest gate level.

Swapping the inputs on the XOR and rerunning the benchmark had only a small effect on the overall performance but did indicate that it was slightly more beneficial to connect the fastest gate level to the most active input (as might be expected). Swapping the inputs on the Muller C-gates, however, caused the system performance to drop off by 3% illustrating the contribution that the differing propagation delay of the gate makes when it causes the initiation of an action to be delayed.

9.2 Block processing performance

To investigate the effect that different, substantial blocks operating on each instruction have on the overall performance, the times through the register bank and the ALU, which are both on the datapath and the primary decode PLA, which is in the control section, were individually varied. The results are shown in Fig. 15; the solid markers indicate the original block operation time.

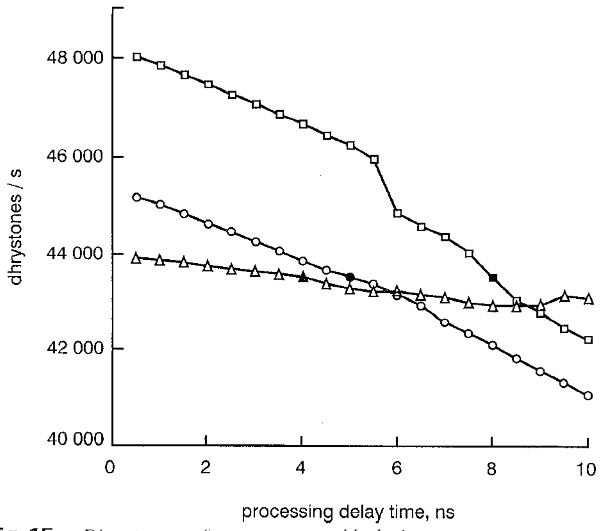


Fig. 15 *Dhrystone performance versus block time*
□ ALU
○ register read access
△ decode PLA

The results indicate that both modules on the datapath can be considered to be on the critical path since an increase in block time from their nominal time degrades the overall performance. By comparison, the performance of the primary decode PLA is approximately constant over the block delay range and it can therefore be assumed that its operation is overlapped with the activity of the slower datapath elements. The graphs show that for the blocks considered, design effort to reduce the ALU time will make the greatest contribution to the overall performance.

9.3 Pipeline occupancy efficiency

Following the AMULET1 design, it was felt that some of the pipelines were longer than necessary. This led to inefficiency in the use of silicon area and in the time to progress data down the pipe since the pipe was often empty or only partially full. The lengths of some of the internal processor pipelines are fixed, since they perform a particular function or are used to prevent potential deadlock situations. For example, the PC pipe in the address interface must be two stages long. The five-stage instruction FIFO pipeline in the data interface must be three stages longer than the PC pipe to prevent a complex deadlock state [5]. Also, the memory control pipe in the data interface must be the same length as the instruction FIFO pipeline.

The operation of four remaining pipelines have been examined. These are the ALU and memory lock FIFOS in the register bank, used for storing the write (destination) register addresses of ALU result values or loaded memory values (respectively), the immediate field extraction pipe in the data interface, used for holding immediate operand values obtained from the instruction, and the write data buffer in the data interface which holds data values for transfer to memory; the only constraint on length is that the immediate field extraction pipe must contain at least one stage for correct system operation.

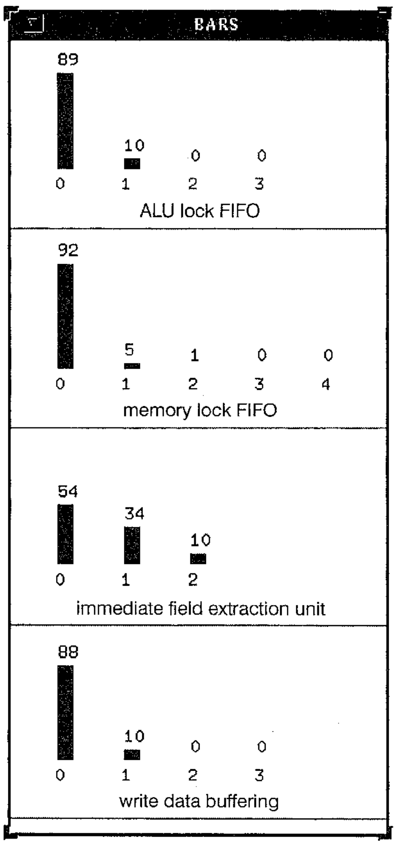


Fig. 16 *Pipeline occupancy*

The information regarding the dynamic operation of each of the FIFO buffering pipelines used throughout the design has been monitored by writing a Verilog system instrumentation function. This pipeline occupancy monitor tool was connected to the external request and acknowledge signals of the pipelines under investigation while the benchmark program was executed. The monitor provides an indication of when valid data is held in each stage of the pipeline to which it is attached. The results for the original benchmark tests, which used the same pipeline lengths as those in AMULET1, are displayed in Fig. 16.

For each of the pipelines, the fraction of the total simulation time that the pipeline occupancy is a particular value is shown. For example, the ALU lock FIFO comprises three stages; for 89% of the total time the FIFO is empty and it contains one item for 10% of the time (rounding errors account for the remaining 1%). The results suggest that the ALU lock FIFO, memory lock FIFO and write data buffer pipelines are too long and could be reduced to contain only 1 stage (or possibly removed altogether). The immediate field extraction pipe appears to be the correct length.

The investigation was extended by modifying the length of each of the pipelines, in isolation, and noting the effect of rerunning the benchmark. These results are shown in Figs. 17 and 18 the '*' in each graph shows the original pipe length.

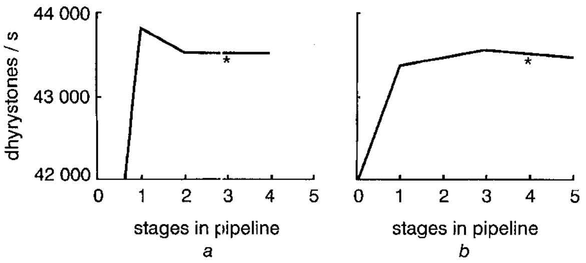


Fig. 17 *Performance versus pipeline length*
a ALU lock FIFO
b Memory lock FIFO

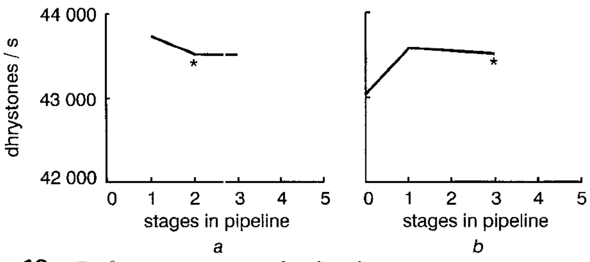


Fig. 18 *Performance versus pipeline length*
a Immediate field extraction pipe
b Write data buffering

These results indicate that on the MDCML processor, the ALU lock FIFO should be shortened to one stage, the memory lock FIFO should be shortened to three stages, the immediate field extraction pipe should be shortened to one stage and the write data buffer should contain only one stage. When simultaneously incorporated into the processor and the benchmark again executed, the resulting performance is measured at 44045 Dhrystones per second. The increase in performance (1.25%) is approximately equal to the sum of the performance increases when the best case of each individual pipeline graph is considered separately.

Since the area occupied by an MDCML design is larger than its CMOS counterpart, the saving in area resulting from reducing pipe lengths is of greater significance in this design than the modest performance gain.

9.4 Single-port register bank operation

Considerations concerning the area of an MDCML design mean that an area–speed compromise may be required if an entire processor is to be integrated in a single chip using this technology and a minimum feature size of 1.2μm.

Significant area can be saved by adopting a single port for the read operand in the register bank rather than the dual-access bank which is standard on ARM microprocessors; two operand instructions would then require sequential accesses to the bank.

Rerunning the benchmark with a single port read facility reveals a 7% loss of performance. Most of this loss could be recouped by directly forwarding the ALU result to the output of the register bank. Alternatively, Fig. 15 shows that improving the performance of the ALU would be sufficient to maintain the performance at the dual-port level. This latter improvement would be relatively easy as the current design uses a simple ripple technique in the adder.

9.5 Summary

Of the features examined, no single feature appears to dominate the performance. Furthermore, relatively large changes to the hardware seem to have only a relatively small effect on performance.These results illustrate that it is difficult to relate the features of an architecture comprising many self-timed blocks to the performance measured or to predict the effect of design changes on the performance. The model is a valuable tool in allowing the design to be optimised to the technology and to measure the effect of changes to the design.

An understanding of total system operation in a large, complex self-timed design is at an early stage and clearly requires significant further research. Models such as the self- timed MDCML processor and the tools that can be constructed in Verilog will assist in such an investigation.

10 Conclusions

A model of a self-timed bipolar ARM processor has been developed and implemented in Verilog as a set of hierarchical intercommunicating self-timed control and data modules. The model incorporates an additional design parameter due to the nonsymmetrical propagation delay of the gates; this affects the modelling produced at all levels. The self-timed behaviour has been found to map well onto the structures provided by the Verilog language. This has also enabled checking and monitoring tools to be incorporated to assist with the debugging of the model.

The correct functioning of such a model gives confidence in the prototype design and provides a valuable tool for investigating performance. The model indicates that the performance should be higher than that for a comparable CMOS design with a factor of two expected. The design changes proposed, particularly to optimise pipeline lengths, enable the design to be 'tuned' to the technology to enhance performance as well as reducing silicon area.

11 Acknowledgements

The work described in this paper was carried out as part of the Transforming Architectural Models (TAM-ARM) project funded within the Design Automation section of the DTI/SERC Advanced Technology Programme and the authors are grateful for this support.

The authors are also grateful for material support in various forms from their TAM-ARM partners, Advanced RISC Machines Limited and GEC Plessey Semiconductors. They would also like to thank past and present members of the AMULET research group in the Computer Science Department of the University of Manchester for many stimulating discussions.

12 References

1 SUTHERLAND, I.E.: 'Micropipelines', *Commun. ACM,* 1989, **32**, (6), pp. 720–738
2 FURBER, S.B., DAY, P., GARSIDE, J.D., PAVER, N.C., and WOODS, J.V.: 'AMULET1: a micropipelined ARM'. Proceedings of IEEE Computer conference (CompCon'94), San Francisco, USA, March 1994, pp. 476–485
3 GEC–Plessey Semiconductors: 'Differential logic design manual' (FAB 4)'. 1.0 Edition, July 1988
4 Cadence Design Systems Inc.: 'Verilog–XL reference manual'. **1** & **2**, 1992
5 PAVER, N.C.: 'The design and implementation of an asychronous microprocessor'. PhD thesis, University of Manchester, 1994
6 Advanced RISC Machines (ARM) Ltd.: 'ARM6 macrocell datasheet' (Cambridge England, May 1992)
7 PAVER, N.C., DAY, P., FURBER, S.B., GARSIDE, J.D., and WOODS, J.V.: 'Register locking in an asynchronous microprocessor'. Proceedings of ICCD'92, October 1992, pp. 351–355
8 WEICKER, R.P.: 'Dhrystone: a synthetic systems programming benchmark', *Commun. ACM,* 1984, **27**, (10), pp. 1013–1030
9 FURBER, S.B., DAY, P., GARSIDE, J.D., PAVER, N.C., TEMPLE, S., and WOODS, J.V.: 'The design and evaluation of an asynchronous microprocessor'. Proceedings of ICCD'94, October 1994, pp. 217–220