MOVE32INT, a Sea of Gates realization of a high performance Transport Triggered Architecture.

Henk Corporaal   Paul van der Arend

Delft University of Technology
Department of Electrical Engineering
Mekelweg 4
2628 CD Delft, The Netherlands
email: heco@duteca.et.tudelft.nl

Abstract

Transport triggered architectures comprise a new class of architectures that are programmed by specifying data transports instead of operations. These architectures are very flexible and scalable, and show several hardware and code generation advantages; they are therefore particularly suited for application specific purposes.

In order to evaluate the specific design and implementation tradeoffs, an instance of a transport triggered architecture, called *MOVE32INT*, has been designed and realized in a 2.0μ CMOS Sea of Gates technology.

MOVE32INT is a 80 MHz, 32-bit general purpose pipelined processor, containing several function units which operate concurrently; in this sense it can be compared to a VLIW processor. To keep the function units busy it is possible to perform four concurrent data transports per cycle. Among the advanced features are the use of so-called hybrid pipelined function units, guarded transports and a flexible locking mechanism. The design of *MOVE32INT* is highly structured, and contains well defined building blocks, which can easily be used to configure application specific processors.

After introducing the architecture of *MOVE32INT*, this paper highlights its pipelining mechanisms, and the time critical pipeline control logic, which (partly) constraints the achievable clock rate. The design of *MOVE32INT* demonstrates that the use of transport triggering allows configurable realizations of very high performance processors, at modest cost and easily controllable complexity.

Keywords: ASP, instruction level parallelism, VLIW, transport triggering, hybrid pipelining.

1 Introduction

Looking into history we see an ever increasing demand for more computing power, both in general and in special purpose applications. Having just experienced the radical change in architecture complexity by going from CISC to RISC processors, most processor manufacturers now upgrade their processor architectures with superscalar capabilities. This allows them to stay binary compatible with what is already on the market, however at the cost of increased design complexity.

As is well known, the exploitation of instruction-level parallelism using superscalar techniques is rather limited. Primarily this is caused by the limited instruction window of the hardware instruction dispatch mechanism. Enlarging this window leads to an enormous increase in complexity of the instruction issue logic.

A potential solution is the use of (binary incompatible) VLIW architectures, which allow concurrent processing of many different operations. Their functionality is adaptable to specific applications. In contrast to superscalars, the (parallel) scheduling of operations is completely done at compile-time. However, as demonstrated in [1, 2], VLIWs have disadvantages with respect to hardware complexity, hardware usage, achievable cycle times and functional flexibility.

The solution appears to be a reapplication of the RISC design philosophy: 1) further (and radical) reduction of the architecture complexity, and 2) giving more control to the compiler. This is achieved by making data transports visible at the architecture level. The result is a so-called *transport triggered architecture*, or TTA. TTAs only have one type of instruction, the data transport or *move*. To enhance hardware concurrency, multiple *moves* are performed each cycle, triggering multiple function units. Single *move* instruction machines have been proposed earlier, e.g. in [3, 4]; however, they did not exploit the inherent VLIW parallelism which is naturally available in the architecture, by scaling the data transport capacity.

In order to explore the specific design and implementation tradeoffs of TTAs, we implemented *MOVE32INT*, a 32-bit integer processor, which has been realized in a 2μ (minimal gate length is 1.6μ) CMOS Sea of Gates (SoG) technology. This implementation served several goals:

- Demonstrating the feasibility of TTAs.
- Exploration of new design features. Especially we were interested how guarded transports, hybrid function units and locking (see section 2) could be efficiently implemented and realized.
- Getting experience with single phase clocking and effective usage of dynamic logic.
- Verifying timing figures obtained from timing analysis with SPICE. We liked to get a feeling for critical hardware paths, and demonstrate that short cycle times are easily achievable.
- Testing the suitability of SoG implementations for processor design. In particular we were concerned about VLSI area and power consumption.
- Use *MOVE32INT* as a prototype for following implementations.

The latter point is particular important to us. TTAs are very flexible and scalable, and therefore well suited for the design of application specific processors (ASPs). Therefore our next step will be the design and implementation of a processor framework for the semi-automatic realization of ASPs and corresponding system software.

MOVE32INT includes advanced features like hybrid pipelined function units, transports guarded by boolean expressions, and a flexible locking mechanism. Interesting is also the fact that TTAs hardly contain any global control logic. This paper describes *MOVE32INT* as an example of a whole range of TTAs, and focuses in particular on the pipelining mechanisms used for transport and operations, and their time critical control.

The remainder of this paper is organized as follows. Section 2 shortly describes TTAs and their advantages. In section 3 we give an architecture overview of the *MOVE32INT* processor. Section 4 concentrates on the pipelining of transports and operations. Section 5 describes the building blocks of TTAs, and concentrates on the blocks for decoding and pipeline control, as implemented in *MOVE32INT*. Section 6 gives results of the realization of this processor in Sea of Gates. Finaly section 7 concludes the paper and points at future developments.

2 Transport Triggered Architectures

TTAs can be viewed as a number of function units (FUs) performing operations and register units (RUs) containing general purpose registers (GPRs), connected by some kind of transport network. By changing the type and number of FUs and the connectivity and capacity of the transport network a whole range of architectures can be implemented; TTAs are therefore ideal for tailoring to any kind of application. Figure 1 shows the operational (or functional) view of a TTA.

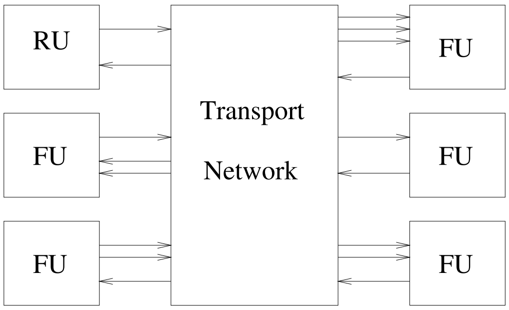

Figure 1: TTA, operational view.

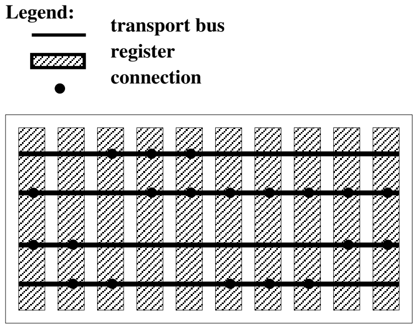

Figure 2: TTA, transport view.

In the next subsections we first describe into some detail the concept of transport triggering, and explain the programming paradigm of architectures based on this concept. Second, we indicate possible advantages of *MOVE*-like architectures (for a complete qualitative analysis see [1]).

2.1 The concept of Transport Triggering

Traditional CISC, RISC, VLIW and Superscalar architectures are programmed by specifying the operations. As a side effect of these operations several data transports are triggered during execution. Therefore we call these architectures *operation triggered*. In *transport triggered* architectures this programming paradigm is mirrored: instead of operations which trigger transports, we specify transports which trigger operations as possible side effect. TTAs separate transport and operations; data transports are made visible at the architecture level. The transport (or programming) view of TTAs shows a set of registers connected in some way by a data transport network. Figure 2 shows a simple transport network, where all registers are connected in some way by a multiple bus structure[1]. High performance TTAs handle multiple moves in parallel; in this sense they resemble VLIW architectures.

TTAs are programmed by specifying the necessary transport of data values in the transport network. There is only one type of operation: the *move* from FU to FU. In general, FUs are connected to the transport network by way of three types of registers, operand, trigger and result registers. All *moves* are than from register to register ($r_x \rightarrow r_y$). The function of the three register types is:

Operand registers: These act as normal general purpose registers, except when an operation is triggered (see below) on the accompanying FU its contents is used as an operand for the operation.

Trigger registers: A *move* to a trigger registers causes the start of an operation on the accompanying FU. The moved value is used as an operand of the operation. If more operands are needed, they are taken from operand registers of the FU. To handle different operations on a single FU, multiple logic addresses may be mapped onto one physical trigger location.

Result registers: Most FUs are fully pipelined, so an operation can be started in every cycle. When a result of an operation leaves the pipeline it is placed in the result register of the FU, from where it can be moved to other FUs.

[1] Full connectivity makes code generation easier, but is VLSI area inefficient and increases the cycle time.

$$\begin{array}{lll} r_a \rightarrow Oint & r_b \rightarrow Tadd & \text{; trigger an addition} \\ ... & & \text{; space for other moves} \\ r_c \rightarrow Osa & Rint \rightarrow Tsd & \text{; trigger a store operation} \end{array}$$

Figure 3: The *MOVE* code for an addition followed by a store.

In order to make the concept of transport triggering more clear, a code example is given in figure 3. The example shows the addition of GPRs r_a and r_b and a store of the result at memory address r_c. The code starts with moving r_a and r_b to the operand and trigger registers of the integer unit (registers $Oint$ and $Tadd$). The trigger register is accessed at the location that causes an addition on the integer unit. The operand move(s) must always be done before or in the same cycle as the trigger move. When the result of the addition leaves the pipeline it is available in the result register $Rint$ of the integer unit. In our example the latency of the integer unit is two, so the result *move* can take place two cycles after the trigger move. The result *move* passes the result directly to the trigger register of the memory unit. At the same time the address of the store is placed in the operand register of the memory unit. Thus a store is triggered with the data to be stored, and the address of the store is supplied via the operand register. Since a store has no result, no result *move* is needed. On average about 2 moves are necessary per operation.

2.2 Advantages of using TTAs

TTAs have several advantages, both for implementation in VLSI and for generating high quality code. The most important advantages are:

- Cycle time minimization. Cycle time is mainly constrained by register-to-register data transport. Pipelining degree of FUs can be adapted to the achievable transport cycle time.
- Functional flexibility. TTAs do not constrain FU functionality in any way. E.g., FUs can have many source and destination operands.
- Implementation flexibility. TTAs do not unnecessarily constrain the implementation of transport network or FU. E.g. combinations of pipelined and nonpipelined FUs and/or transport networks are possible.
- Performance scalability. Performance scales by adding multiple FUs to an existing architecture, pipelining existing FUs, or by increasing the data transport capacity. In contrast to current VLIW processors, the number of FUs is not coupled to the data transport capacity (e.g. the number of register ports), e.g., FUs can be added without having to change the transport capacity, and therefore the instruction format.
- Simplicity. Despite (or maybe because of) the range of flexibility and scalability, TTAs have a very simple and regular organization. RISC design principles are even more apparent than in well known RISC processors. E.g., one of the RISC design goals was to simplify decoding logic. To this respect, TTAs take the ultimate approach; they support only type of (*move*) operation.
- Hardware efficiency. Looking carefully at TTAs, it appears that they use their registers and data transport capacity more efficiently. E.g. the data transport capacity is designed for average behavior, instead of worst case, as has to be done for traditional VLIWs.

- Programming transports instead of operations gives opportunities for new code scheduling optimizations ([5, 6]).

For a more extensive evaluation of TTAs we refer to [1].

3 Overview of the *MOVE32INT* processor

This section gives a quick insight in the architecture and capabilities of *MOVE32INT*. First, we describe its operational view. Second, its transport view, and finally the main architecture limitations. Remark, that we have to skip many details, because a complete architecture description requires a whole manual [7].

3.1 Operational view

The architecture mainly consists of a transport network, controlled by the network controller, and several function units (FUs). The network contains 4 busses; each bus contains a data bus which is capable of transporting one data value of 32 bits, an Id-bus, which transports one move operation specifier of 16 bits (as shown in figure 5), and a control bus which contains a few control signals. This means that four data transports (*moves*) can be handled in parallel.

Figure 4 shows the operational view of *MOVE32INT*, including operand (O), trigger (T) and result registers (R). As indicated, the processor uses a harvard architecture with separate address and data paths to instruction and data memory. Because of pin limitations not all internal address lines are externally available. *MOVE32INT* contains the following FUs:

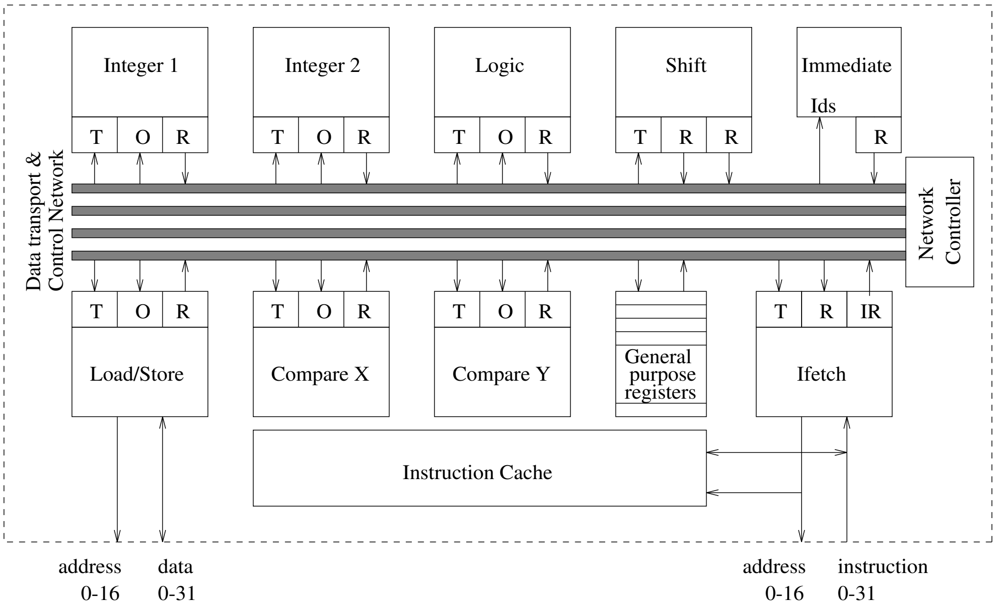

Figure 4: Operational view of *MOVE32INT*.

1. Two integer units, based on the SNAP adder [8][2].

2. Logic unit, performing logic *and*, *or*, and *exor* operations.

[2]The current adder realization uses static logic, and shows a latency of 18 ns, which does not fit into one transport cycle; in the near future we will replace it by a dynamic version, with a latency less than 12.5 ns.

3. Shift unit. This unit has two result registers, containing 1-bit and 2-bit shifted results.
4. Immediate units (one per bus). Reading from this unit has the effect of putting the 6 bit source identifier of the move operation as an unsigned integer on one of the data transport busses.
5. Two compare units. They produce boolean result values which can be read from the result registers. Their output is not connected to the data bus, but to the control bus, and is used to guard (conditionalize) the data transport.
6. Load & Store unit. This unit contains a trigger and result register for the load part, and a trigger and operand register for the store. This unit is connected to the external data memory, using a 32 bit data path, and a 17 bit address path; this allows for a maximum data memory of 128 kwords, or 512 kbyte.
7. Ifetch unit. This is basicly a load unit, where the address (the PC) is auto incremented. However writing to the trigger register forces a jump. Reading from the result register gives the address of the current instruction plus 2. The Ifetch unit is connected to a small internal instruction cache (using a 64 bit data path) and the external instruction memory (using a 32 bit data path). The maximum size of the external instruction memory is 128 kbyte. The size of the instruction cache (16 lines of 128 bits) is limited by the available SoGs area.
8. Register unit. This function unit implements 12 general purpose registers.

For clarity, many details are not shown in figure 4, like status registers, and the precise connectivity (most registers are only connected to 2 of the 4 busses).

3.2 Transport view

As mentioned in section 2 the transport view of a TTA is extremely simple; it contains a set registers acting as source and/or destination for *move* operations. Table 1 gives an overview of the register set, separated by function unit, and by destination (T and O) and result (R). Most register name abbreviations will be

function unit	trigger	operand	result
integer-1/2	Tadd, Tsub	Oint	Rint
logic	Tand, Tor, Texor	Olog	Rlog
shift	Tasr, Tlsl, Tlsr	-	Rs1, Rs2
immediate		-	Rimm
compare-x/y	Teq, Tle, Tleq, Teqz	Ocmp	Rx/y
load	Tla	-	Rld
store	Tsd	Osa	-
Ifetch	Tpc, Tpo, Texc, Texo, Tpch	-	Rpc, Rpch
Register file	r1-r12	-	r1-r12

Table 1: Function units and their registers on *MOVE32INT*.

clear, however, a little explanation may be helpful:

la, sa	load address and store address.
ld, sd	load data and store data.
s1, s2	shift result 1 or 2 bits.
pc	program counter, used for absolute jumps.
po	page offset, used for 6 bit, unsigned, page relative jumps.
pch	pc-chain, used for saving and restoring active pc values.
exc,exo	similar to pc and po, however, writing to these registers causes an exception.

Each instruction for *MOVE32INT* contains 64 bits and specifies 4 transports, called *moves*. The format for an instruction and a *move* are shown in figure 5. It contains a 3-bit guard specifier, a 1-bit immediate

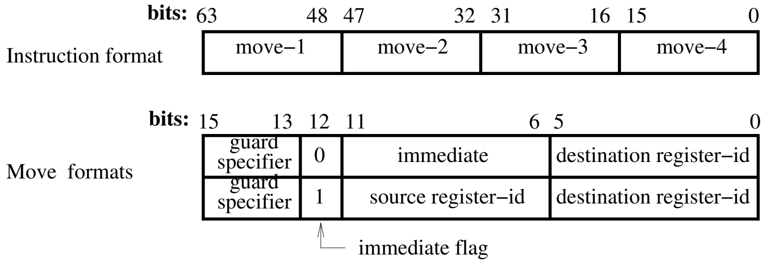

Figure 5: *MOVE32INT* instruction and *move* formats.

flag, and 6-bit source and destination fields. The immediate flag determines the interpretation of the source field. This source field contains either a short 6 bit immediate, or a specification of a source register[3].

MOVE32INT supports a very general way of conditional execution; each *move* is conditionally executed; this is in contrast to many traditional architectures where only the control operations have a conditional execution mode. The condition is specified by a 3-bits guard specifier in each *move*. During each cycle 4 guards are produced, one for each transport bus. The guard determines if a *move* has to be performed or has to be canceled. The guard specifiers indicate how to evaluate the guards; many boolean expressions of the booleans Rx and Ry can be specified[4]. Expressions are useful in guarding nested if-then-else code.

3.3 Architecture limitations

In accordance with the design goals of *MOVE32INT*, as stated in the introduction, the architecture is rather limited. Many restrictions result from area limitations. Remember that *MOVE32INT* has been implemented in 2μ CMOS SoG technology. Major restrictions, in comparison with state of the art processors, are:

1. Limited number of registers. This is partly circumvented by the possibility of using FU pipeline stages as temporary storage, and by the fact that *MOVE* architectures require less registers.

2. Limited cache support. *MOVE32INT* only provides a small Instruction cache, and does not have a data cache.

[3]Actually, there exists only one *move* format, containing a 7-bit source-id. Half of this id-space maps on the result register of the immediate FU. Checking the immediate flag is done locally in this FU. *MOVE32INT* does not contain any global instruction decoding logic.

[4]An interesting fact is that the guard evaluation mechanism can be implemented like a normal 1-bit data transport bus, which can be written by combinations of Rx and Ry simultaneously, using the wired-or function. The guard specifier is essentially a source-id for a multicast write on this bus!

3. Limited exception support.
4. No virtual addressing and MMU support.
5. No protection support. No kernel mode.
6. No multiprocessor support.
7. Restricted operation repertoire; e.g. no floating point units.

On the other hand the architecture is quite powerful; it can initiate four operations per cycle, it supports multiway branching, and allows easy filling of branch delay slots, using the flexible guarding mechanism.

4 Pipelining and Control mechanisms

Traditional RISC like architectures have streamlined their instruction set in such a way that they can be easily pipelined; e.g. they omitted complex addressing modes[5]. Many RISC processors use a 4 or 5 stage pipelining mechanism. Figure 6 shows a well known 5 stage pipeline execution mechanism. In the Ifetch stage an instruction cache lookup is performed. Tag comparison can be overlapped with the second pipeline stage. Transport of data values from and to registers is done during the RF and WB stages (register fetch and write back). The actual operations are performed during the Execute stage (and Memory stage for load and store operations). At the end of this stage, the result of the operation has to go through the bypass network.

Instruction	**time** →						
i	Ifetch	Decode/RF	Execute	Memory	WB		
i+1		Ifetch	Decode/RF	Execute	Memory	WB	
i+2			Ifetch	Decode/RF	Execute	Memory	WB

Figure 6: RISC pipelining execution mechanism.

The actual data transport in *MOVE32INT* takes 1 clock cycle only. However, that does not mean that *MOVE* architectures are not pipelined. FUs and the data transport network are completely separated. Pipelining decisions for transport network and FUs can be taken independently; they are described next. Pipeline control mechanisms are introduced thereafter. Their implementation and realization is discussed in sections 5 and 6.

4.1 Transport Network pipelining

In *MOVE32INT* Icache lookup, bus control and data transport each take one cycle. The resulting pipelining picture for *MOVE32INT* is shown in figure 7.

Comparing this to the traditional RISC pipeline we observe the following important differences:

- Although decoding *move* operations is trivial, a separate stage is needed for the transport of the source and destination register identifiers. Including this transport into the actual *move* stage would unnecessarily stretch the clock period.
- RF, WB, and bypass of data values between FUs are all done during the transport stage; they all have to be programmed explicitly.

[5] In fact they only support register-base and register-base plus offset addressing modes for a very restricted class of instructions, namely the traditional register to register moves.

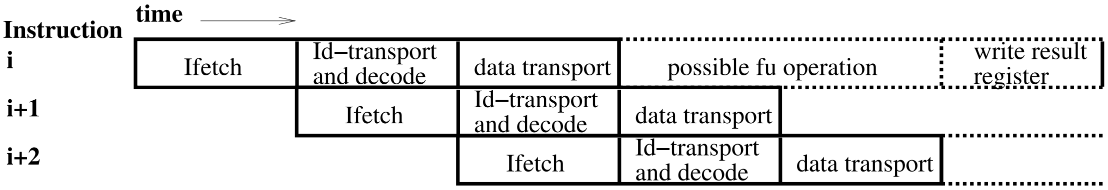

Figure 7: *MOVE* pipelining execution mechanism.

- The *move* operation ends with the data transport stage. Everything else happens as possible side effect of the data transport. These side effects range from nothing (e.g. when writing to a GPR or operand register), performing an operation (writing to a trigger register) to flushing a value from an FU (reading from a result register).

4.2 FU pipelining

Use of the transport triggering concept enables many compiler optimizations at the data transport level. In order to enhance the scheduling freedom for the compiler even further, and to make the design of *MOVE32INT* easier, FUs are implemented using a so-called *hybrid* pipelining mechanism. Figure 8 shows a typical hybrid FU. Its pipeline has a latency of 3 clock cycles; this means that reading of the result register (R) can be done 3 cycles after writing to the trigger register (T).

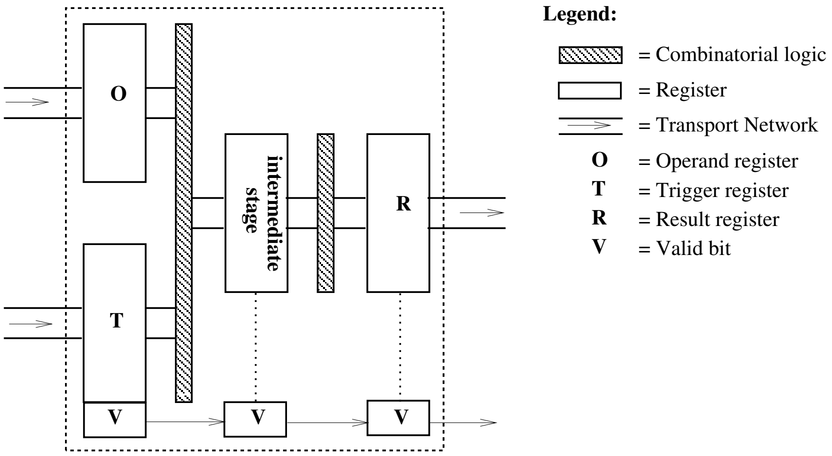

Figure 8: The pipeline of a typical function unit.

Current high-performance architectures generally implement pipelines where either intermediate pipeline stages continue always on each clock cycle to the next stage (e.g. [9, 10]) or intermediate stages continue only in case a new operation is issued (e.g. [11]). In hybrid pipelines intermediate values continue to the next stage on each clock cycle, as far as they do not overwrite results from previous operations. Valid bits (V) are used to control this mechanism. This offers extra scheduling freedom; results may be moved from the FUs at any time after triggering the operation[6]. Hybrid pipelines also reduce register usage by using intermediate stages as temporary storage.

[6]Even before the result has been calculated, as will be explained in section 4.3.

4.3 Pipeline control

In order to handle non-deterministic events (like cache misses and exceptions), and to perform conditional execution we need mechanisms for *locking* (delaying) and *squashing* (canceling) data transports. A lock can be requested by a FU when it is not able to perform the action required by the executing instruction but will be able within some cycles. If postponing the data transport does not solve the problem, the FU could use the exception system. Squashing of transports may occur only in case of guarded execution or exceptions.

Guarded execution The guard system contains 2 types of lines:

- Local squash request line (local to a specific bus).
- Local squash line

If a FU, typically the Ifetch or guarding FU, wants to cancel a transport it can pull the local squash request line. Local squashes can not be masked (overruled) in *MOVE32INT* like local locks, so there is no need for an extra unmaskable global squash request line. If a squash for all busses is needed, all squash request lines should be activated. If a squash is granted (always in the *MOVE32INT*), this is signaled to the FUs through the local squash line of the squashed transport busses.

Locking The lock system contains 3 types of lines:

- Maskable local lock request line.
- Nonmaskable global lock request line
- Global lock line

There is a local lock request line for each bus. When a FU detects a temporary problem, this line can be pulled to request an extra cycle for the current transport on the bus. A lock request can be masked (overruled) by a squash request. When the problem does not only affect the transport on the bus related to the lock but to all busses, and should occur without considering the guard status, the global lock request line can be used. This will e.g. be used by the Ifetch FU, in case of a Icache miss[7].

In *MOVE32INT* it is not possible to lock only one bus. If a requested lock is granted, all transports have to be synchronized. So only one global lock line to signal all FUs to prepare for a locked cycle is enough.

Exception handling The exception system contains only one line. It is used for both requesting and signaling an exception. The exception line is monitored by FUs which should change their mode when starting an exception service. In most cases this will be the Ifetch FU or a special exception unit. In *MOVE32INT* the only FU which will change mode or can request an exception is the Ifetch FU[8]. Therefore, a global exception line is not needed in the implementation of *MOVE32INT*.

5 Processor building blocks

TTAs are highly regular and constructed using a restricted number of building blocks. Figure 9 illustrates the hierarchical structure of building blocks as used in *MOVE32INT*. Different TTAs can easily be configured by assembling different combinations of these blocks.

[7]The compare FUs also use this line when they can not deliver the boolean result in time; this lock may not be masked by a squash request from an yet to be calculated guard.

[8]Upon exception, the Ifetch unit activates all the squash lines for 3 cycles, effectively squashing the whole transport pipeline.

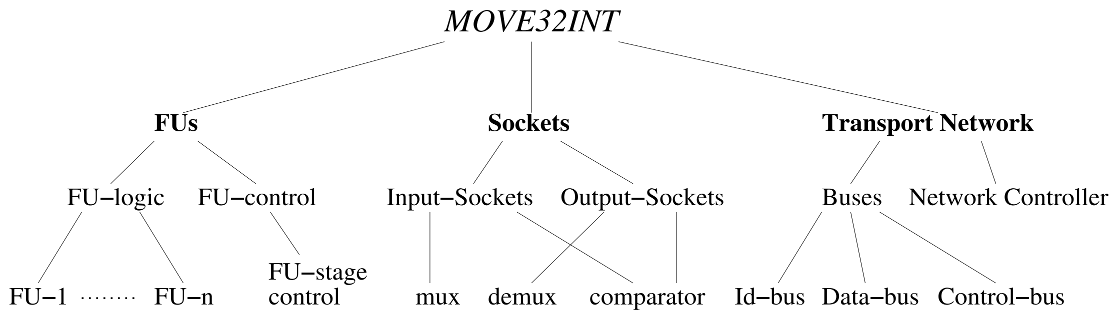

Figure 9: Building block hierarchy for *MOVE32INT*.

In this section we will pay attention to three important blocks, sockets, network control unit, and the FU-stage control. They are distinctive for *MOVE32INT*, and establish the core of any TTA. They are also in the critical path of our processor, as will be shown in section 6.

5.1 Sockets

Generally, transport triggered architectures do not have to be fully connected; that means that (some) registers are not connected to all busses. In *MOVE32INT* the connection between the network and FUs is established by so-called input and output *sockets*. Figure 10 shows a possible connection.

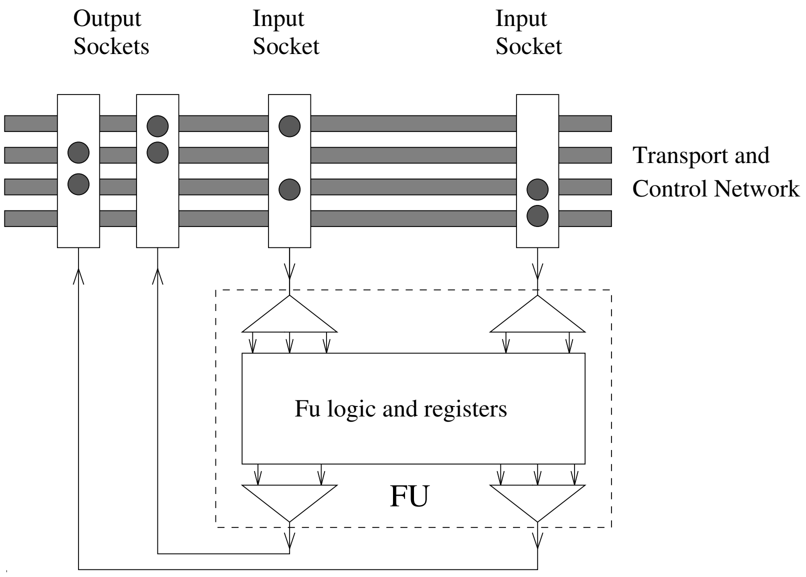

Figure 10: Connection between FU and the network using sockets

The figure shows a FU which is connected to 2 input and 2 output sockets. Most sockets in *MOVE32INT* are connected to 2 (of the 4) move busses. Internally the FU may connect multiple inputs to one input socket; the same holds for outputs. A socket can be viewed as a gateway which is able to pass one data item per cycle. Figure 11 presents some more details of an input socket, which is connected to 2 move busses.

The instruction decoding is performed locally, inside a socket. During the decode stage, a check is done for each bus. If the socket is selected for a bus (by matching the value on the identifier bus) a corresponding select bit is set. Furthermore, an optional opcode is extracted from the identifier bus and, together with the select bits, latched for later use.

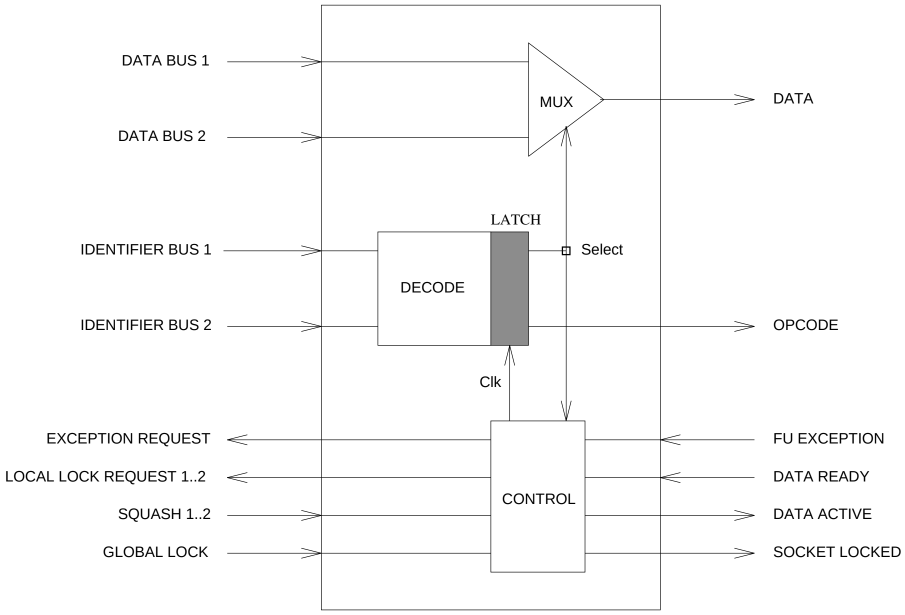

Figure 11: Details of an input socket

During the transport stage, the opcode is just forwarded to the FU, but the select bits are used for driving the data (de-)multiplexer and the control logic. The control logic has to deal with the synchronization of the move transport network and the FU. If an input socket is selected for instance, it will give a data active signal to the FU. But if the FU does not signal data ready, the input socket should alarm and give a local lock request signal for the selecting bus. The next event is that the socket can receive a squash signal from the selecting bus. This urges the socket to revoke its data active signal because no transport will be done. The latest event before the data on the selected bus will be loaded by the FU is the occurrence of a global lock. The socket will immediately forward it by signaling a socket locked signal to the FU. Because this signal is late in comparison with the squash signal, it is kept separately as it can be fed in parallel to the stage controllers. Of course, any FU exception report will be forwarded directly by the socket to the network.

The equations of the input socket controller are:
$DataActive = (\overline{Squash_1} \wedge Select_1) \vee (\overline{Squash_2} \wedge Select_2)$
$SocketLocked = GlobalLock$
$Lock_i = Select_i \wedge \overline{DataReady}$
$ExceptionRequest = FUException$

5.2 Network control

All timing aspects in the *MOVE32INT* are measured against two clocks. The control clock is not maskable and is a delayed version of the off-chip clock. It is only used to generate a data clock. The data clock is a maskable clock, latching all (control) data in the processor. The phase difference between both clocks is fixed using a standard buffer which can be enabled. The clock enabling signal should arrive at the buffer before the rising edge of the control clock. The timing aspects of the network control are shown in figure 12.

From the local lock and squash requests, the global lock signal is calculated by the only global unit in the *MOVE32INT*, the network controller. This unit contains critical logic because the global lock should arrive at the sockets with enough margin left from the control clock to spread the signal over the FU stages.

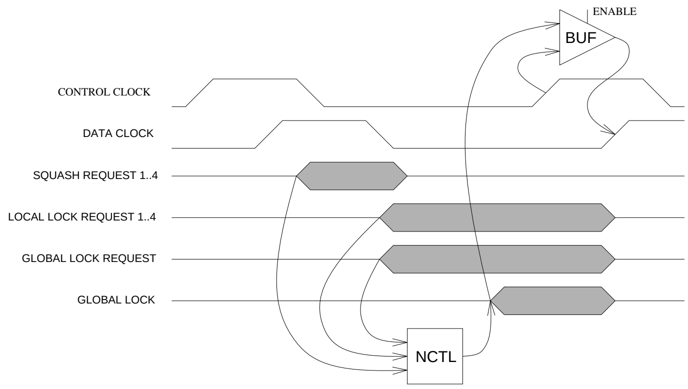

Figure 12: Network control unit implementation

The main task of the network control unit is to monitor the control lines for requests and to decide if a request should be granted and at which scope. In *MOVE32INT* the task is limited to checking each local lock request against the local squash request of the same bus. A local lock request on a squashed bus is always ignored. It also generates a global lock when requested through the (unmaskable) global lock request.

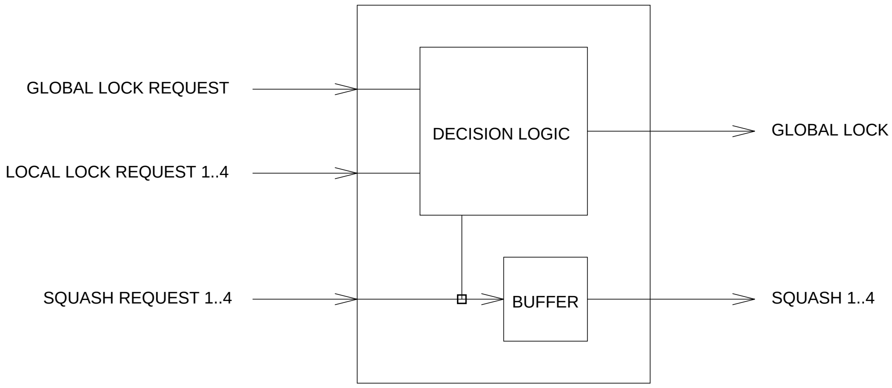

Figure 13: Network control unit implementation

The equation of the network controller is:
$GlobalLock = GlobalLockRequest \vee (\sum_{i \in allbusses} LocalLockRequest_i \wedge \overline{SquashRequest_i})$

5.3 FU Stage Control

FUs in *MOVE32INT* are hybrid pipelined, which means that (intermediate) stages may only be loaded if their old value is accepted by the following stage. For *MOVE32INT* we developed a universal stage controller which is suitable for both fully and non or sub-pipelined FUs. The latter have a stage delay larger than one.

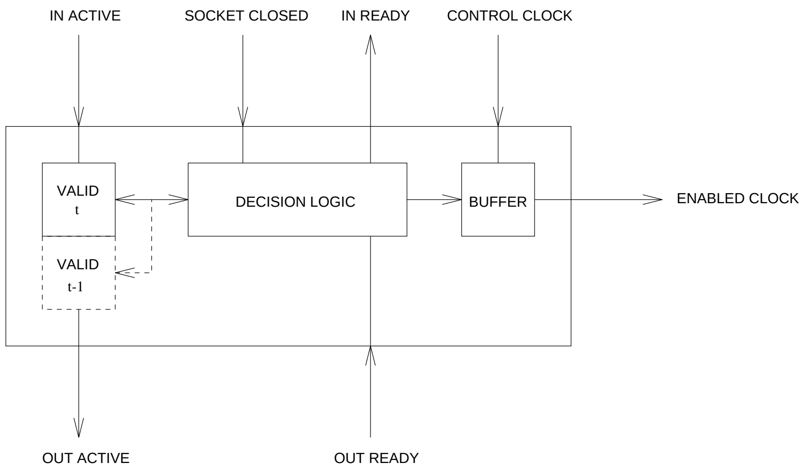

Figure 14: Function Unit stage control implementation

The stage controller should memorize if the data in the stage registers is valid and how many cycles it already has been stored. The valid bit storage is implemented as a single register. The delay information is measured by an always shifting register which shifts the valid bit in and is reset when new data is loaded in the stage registers. So the last bit in the queue will tell if the data is available for the next stage. As an extra feature the decision logic is designed in such a way that the stage registers will never load data which is marked invalid (garbage data). This is to make sure that the the controller will not waste any power.

The equations of the stage controller are:

$EnabledClock = ControlClock \wedge InActive \wedge (InReady \vee \overline{SocketClosed})$

$InReady = \overline{Valid_t} \vee (OutReady \wedge OutActive)$

$OutActive = Valid_{t-delay+1}$

$Valid_t = InActive \vee (\overline{InReady} \wedge SocketClosed)$

6 Realization and Critical path

MOVE32INT is realized in a 2μ Sea of Gates (SoG) image with a channel length of 1.6μ. The image is build from 88 rows N and P transistor pairs of 1088 pairs per row. The horizontal wiring is mainly done over the gates in metal 1 and the vertical wiring in metal 2. The transistor dimensions are pretty large to allow enough horizontal wiring. This causes high power dissipation and makes the image not very useful for compact layouts like RAMs. On the other hand, the large capacity ensures a reliable dynamic signal behavior. The use of SoGs restricts the design space, but reduces design time. As an example of this easy design method, the realization steps will be shown for the 2 cycle delayed stage controller from figure 14.

Figure 15(A) shows the schematic of the stage controller implementation in which all dynamic parts are clocked by the data clock. The logic type used for each gate is indicated (see legend for an explanation). The two and-or combinations in the figure decide if the data clock should be enabled and if the valid bit should be set. Together with the surrounding transmission gates, the latter and-or forms the valid register. The boxed delay stage contains a resettable register; it can be chained to match the number of stage delay cycles.

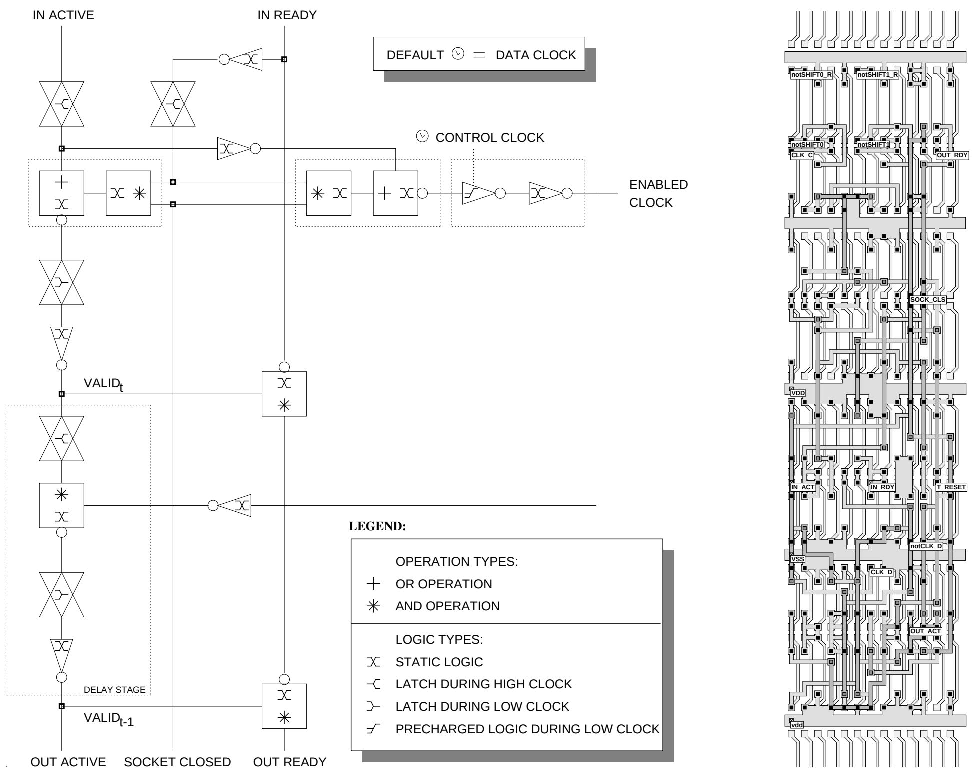

Figure 15: A 2 cycle delayed stage controller: A) Schematic B) SoG Pattern

All gates are selected from a library (or newly designed) and a placer and router take care of the rest. Figure 15(B) shows the stage controller circuitry which is used in a *MOVE32INT* processor design. It can be fitted in the controller area beneath any FU which needs the controller.

Floorplanning The floorplan of the *MOVE32INT* chip is very simple. The datapath containing the FUs, registers (RG), and move busses are placed into one strip on the top side of the chip. In the middle a vertical channel is saved which leads the instruction address (PC) downwards to the cache and the data address upwards to the pads. The load/store unit is spread over the datapath in four parts. An address generator, load/store control, the low data interface part and the high data interface part. The instruction fetch unit is also divided in several parts for pc generation, ifetch control, cache control and cache storage.

In the floorplan which is shown in figure 16, strips for instruction decode and datapath control are shown. The decode strip contains the identifier busses and selection comparators (part of the sockets). The control strip is used to place stage, register, socket controllers and buffers.

Critical path analysis The target speed for the sea of gates realization is 80 Mhz. The high phase of the clock is limited to a minimum of 4 ns which is constrained by the instruction cache address decoding logic which is beyond the scope of this report. The low phase, however, is caused by the described control system. The critical path is formed by the lock system from local lock request through the network controller to the global lock signal, the sockets and finally the stage controller which sets a data clock enable signal at 6 ns after the causing local lock request. The buffer which combines the enable with the control clock to generate the data clock uses another 2.5 ns setting the low part of the data clock to 8.5ns.

The spice output of a simulation of the critical path at 125 degrees Celsius is shown in figure 17. The simulated critical path network is extracted from the SoG layout including nodes loaded with the proper

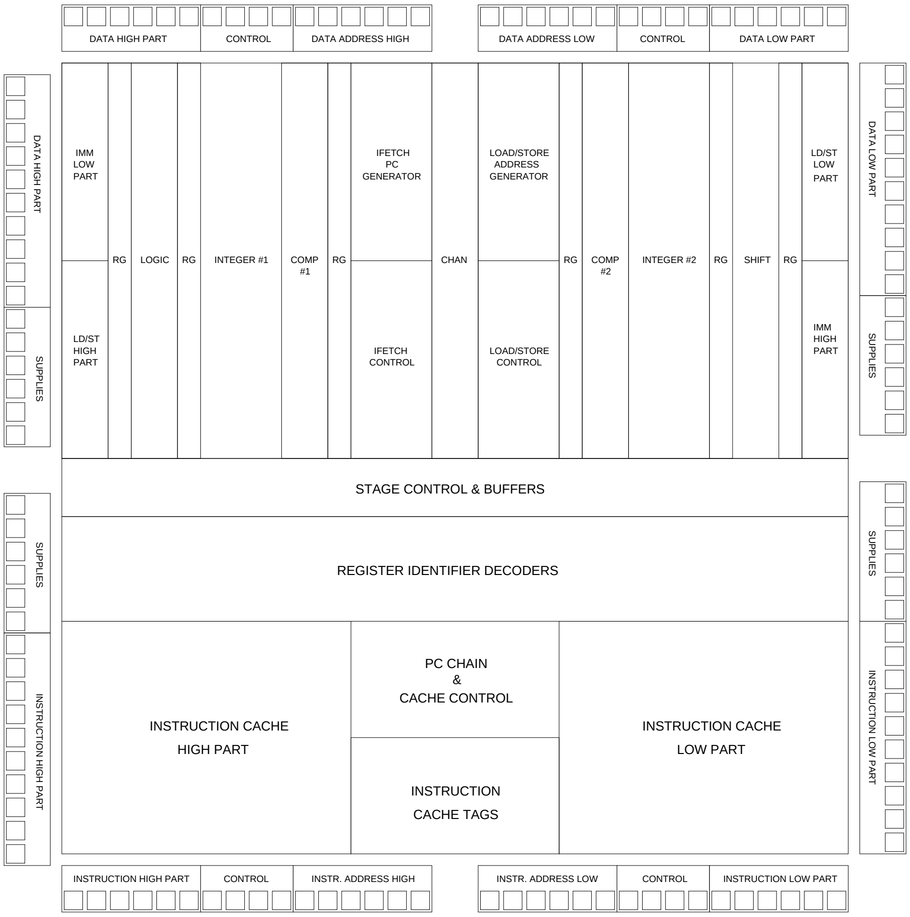

Figure 16: Floorplan of *MOVE32INT* on the available image

capacities. Signals DO_LOCK and DO_SQUASH represent internal requests within the sockets. We see that at $t = 34ns$ a GLOBAL_LOCK is generated in time to disable the clock buffer (and corresponding stage latching); ENABLED_CLOCK remains inactive.

7 Conclusions and future research

In this paper we presented *MOVE32INT* an instance of a TTA, which exploits instruction level parallelism by issuing four data transports per cycle and letting many FUs operate concurrently. *MOVE32INT* has several special features like general guarding and locking mechanisms, absence of global decoding logic, a minimum amount of global control logic, and a new pipelining mechanism for FUs. The latter results in more scheduling freedom, and makes FUs more autonomous.

MOVE32INT is constructed using a restricted and well defined set of building blocks. These blocks can easily be reassembled to generate different, application specific processors (ASPs). In this sense *MOVE32INT* represents a large class of TTAs.

The use Sea of Gates was very helpful for quickly evaluating different implementations and layouts of critical parts like bus design and control logic. Despite the modest technology available (2μ CMOS), and the use of fixed size (and relative large) transistors, we were able to achieve a very high clock rate

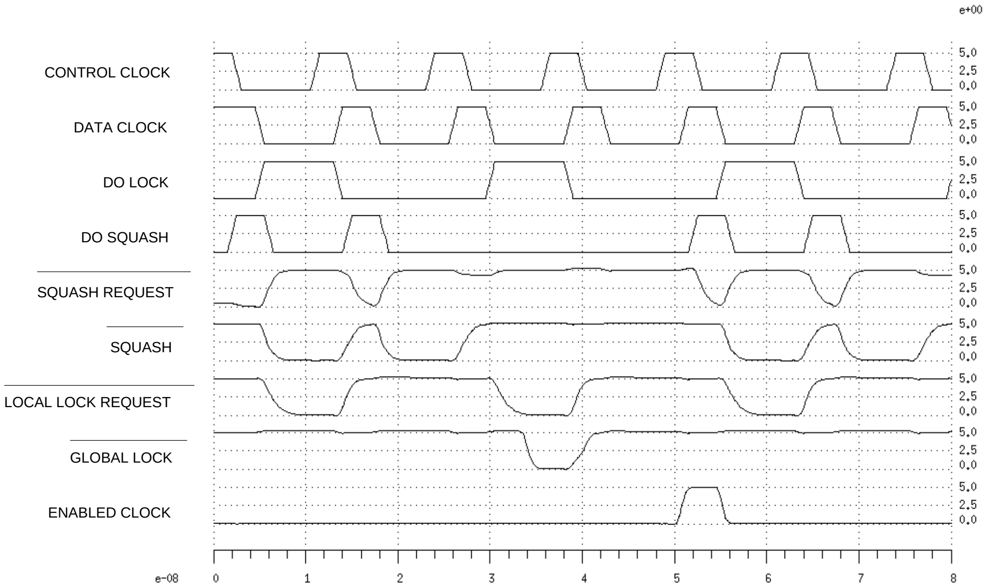

Figure 17: Timing of critical path in network control unit

of $80MHz$, resulting in a theoretical performance of $320Mops/second$[9]. We can compare our design to a full custom RISC processor we have made (see [12]), which used the same CMOS technology (but no SoGs). This processor achieved a clock rate of $20MHz$, resulting in a maximum performance of $20Mops/second$. The very high performance of *MOVE32INT* is a direct consequence of the simplicity of TTAs. Performance could easily be increased when a) denser technologies are used, and/or b) a larger image is available, which allows more FUs and transport capacity to be implemented. Of course, extra concurrency must be exploitable by the application.

Status and research Currently (January 1993), all blocks of *MOVE32INT* are realized and tested. Final assembly will be finished within 2 months, after which the SoG image will be metalized (2 layer metal).

Based on the design experience of *MOVE32INT* we will continue to design a flexible framework for generation of ASPs, which will be largely technology independent. All building blocks will be generated by technology independent generators.

An important aspect, not touched in this paper, is the code generation for TTAs. Currently, we have implemented a compiler generator for a large class of TTAs. However, the generator has to be extended in order to deal efficiently with restricted connectivity of the transport network, as e.g. in *MOVE32INT*. This is one of the main research topics in the compiler area. A compiler generator will also be indispensable for evaluating different processor alternatives during the design trajectory of an ASP; it will therefore be an integral part of our future ASP framework.

References

[1] Henk Corporaal and Hans (J.M.) Mulder. MOVE: A framework for high-performance processor design. In *Supercomputing-91*, pages 692–701, Albuquerque, November 1991.

[2] Paul Stravers and Eric Aardoom. A procesor framework customized for navigation computations. *INTEGRATION, the VLSI journal*, 14(2):197–214, December 1992.

[9] Assuming that each transport triggers an operation, which is a very ideal situation.

[3] G. J. Lipovski. Architecture of a simple, effective control processor. In *Second Symposium on Micro Architecture*, pages 187–194, 1976.

[4] Zdenek Blazek. Preliminary design of a parallel SIC architecture. *Microprocessing and Microprogramming*, 28:103–108, 1989.

[5] Jan Hoogerbrugge and Henk Corporaal. Comparing software pipelining for an operation-triggered and a transport-triggered architecture. In *Lecture Notes in Computer Science 641, Compiler Construction*, pages 219–228. Springer-Verlag, 1992.

[6] Henk Corporaal. Evaluation of Transport Triggered Architectures for General Purpose Applications. Technical Report submitted for publication, Delft University of Technology, Electr. Eng. Department, The Netherlands, August 1992.

[7] Henk Corporaal. Move32int, architecture and programmer's reference manual. Technical Report 1-68340-44-(1993)01, Delft University of Technology, Electr. Eng. Department, The Netherlands, 1993.

[8] Nhon T. Quach and Michael J. Flynn. High speed addition in CMOS. Technical report, Stanford University, Department of Electrical Engineering and Computer Science, February 1990.

[9] R. Ramakrishna Rau, David W.L. Yen, Wei Yen, and Ross A. Towle. The Cydra 5 Departmental Supercomputer. Design Philosophies, Decisions and Trade-offs. *IEEE Computer*, pages 12–35, January 1989.

[10] Trace: Technical summary. Multiflow Computer Inc., June 1987.

[11] Intel. *i860 64-bit Microprocessor Programmer's Reference Manual*, 1989.

[12] J.M. Mulder, R. Portier, and A. Srivastava. A framework for high-speed controller design. In *Proceedings of the 23th workshop on microprogramming and microarchitectures*, November 1990.