Design of Transport Triggered Architectures

Henk Corporaal
Delft University of Technology
P.O. Box 5031, Delft, The Netherlands
email: heco@duteca.et.tudelft.nl

Abstract

Transport triggered architectures (TTAs) form a superclass of traditional very large instruction word (VLIW) architectures, in the sense that they not only exploit operation style parallelism, but also the parallelism available at data transport level. This is possible by making all transports visible to the compiler.

The main advantages of transport triggered architectures are its simplicity and flexibility, allowing short processor cycle times and a quick (application specific) processor design. Transport triggered architectures also have certain advantages with respect to scheduling freedom and transport utilization.

The paper will discuss the concept of transport triggering and its corresponding advantages. It further concentrates on a prototype VLSI implementation in a 1.6 μ Sea of Gates technology, called MOVE32INT, which demonstrates the feasibility of transport triggering. Finally it explores the automatic generation of arbitrary TTAs.

Keywords: *VLIW, pipelining, operation triggering, transport triggering, Sea of Gates.*

1 Introduction

In order to increase the performance of single processing nodes (and of MIMD systems built out of these nodes) one may exploit the available instruction level parallelism inside applications. This type of parallelism can be supported by hardware using either *superpipelining* of function units (FUs), or *functional parallelism*, resulting in multiple FUs. Ideally we would like to combine these two techniques. However the resulting architectures (superpipelined VLIWs or superscalar architectures) exhibit disadvantages which result in complex organizations, poor hardware utilization, difficult to change functionality, and poor performance scalability [1].

Figure 1 shows the internal data path of such a superpipelined VLIW architecture. A set of FUs is connected to a bypass unit containing operand and bypass latches. The bypass unit is connected to a register file.

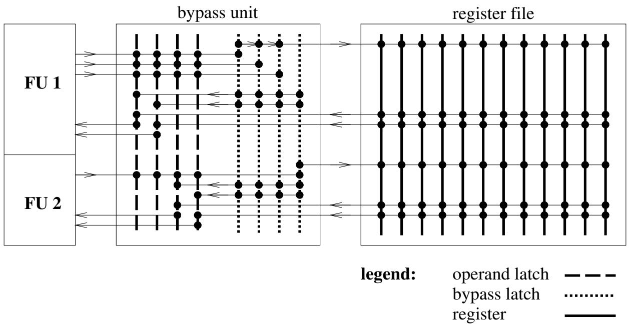

Figure 1: Communication between FUs and Register file in a VLIW architecture.

Looking at figure 1 we make the following observations:

1. Values written from the bypass into the register file may be dead. Measurements indicate that many generated results are consumed immediately. Deeper pipelining enlarges this number. The bandwidth of the write port into the register file is therefore largely redundant.

2. Operations requiring one or two operands under utilize the register-bypass bandwidth (e.g., monadic operations and branches).

3. The bypass becomes very complex when we increase the number of FUs N, and/or the superpipelining degree S; the complexity is of order($N^2 \times S$).

4. Adding FUs requires a redesign of the architecture.

We conclude that the organization is complex, the communication requirements are high, and that the communication network is under-utilized. However, the compiler is able to detect all cases of under-utilization, and knows which temporary results it needs from the bypass. This motivates us to change the programming paradigm from *operation triggered* to *transport triggered*; instead of specifying operations which trigger transports, Transport Triggered Architectures (TTAs) are programmed by specifying transports which may trigger operations as side effect. TTAs make all transports *visible* in the architecture.

The remainder of this paper discusses the concept of transport triggering, describes a prototype TTA implementation, called MOVE32INT, and discusses how to automatically generate configurable TTAs.

2 Transport triggering

As shown in figure 2, TTAs can be viewed as a superset of traditional VLIW architectures. RISC architectures are

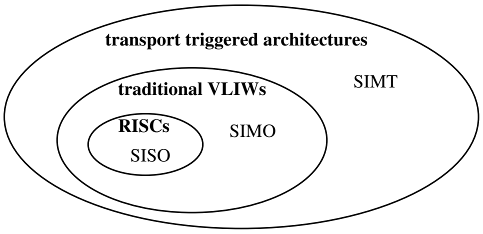

Figure 2: Architecture space of long instruction word machines.

of type **SISO** (single instruction, single operation), VLIW architectures are of the **SIMO** type (single instruction, multiple operation). TTAs are of type **SIMT** (single instruction, multiple transports); one instruction may specify multiple concurrent transports. Transports may trigger operations as side effect.

Central to the idea of transport triggering is to have more control about what is going on within a central processing unit. In this sense the change from operation to transport triggering is a natural extension of the change from CISC to RISC architectures. While the application of RISC principles simplified the design and reduced the instruction set, transport triggering extends these principles: it reduces the number of instructions to one (a data transport) and simplifies the design even further.

More compiler control means more code optimization opportunities. In order to get more control, the data transport has to be separated from the operations.

TTAs are more general than VLIWs, because they put fewer constraints on the scheduling of the data transports. TTAs can even emulate VLIWs by instructing the compiler to not use the extra available transport-level optimizations.

Separating transport from operations

The structure of a TTA is very simple; figure 3 shows an example architecture, containing 9 FUs and 6 transport buses. Each FU is connected to the transport network with one or more input and output sockets. In general, the network does not have to be fully connected, although full connectivity would relax the code generation process.

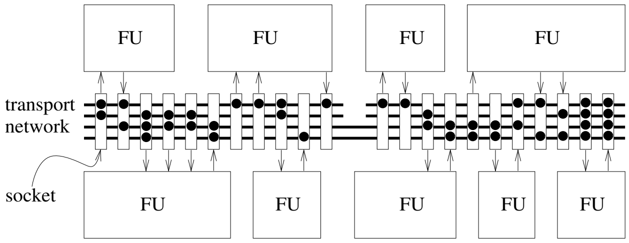

Figure 3: General structure of a MOVE architecture.

Each function unit (FU) includes one or more operand and result registers which are part of the total register address space (as seen by the programmer). We distinguish the registers according to their purpose: FU-operand registers (O), 2) FU-trigger registers (T), 3) FU-result registers (R), and 4) general-purpose registers (r)[1]. In principle one can use O, T, and R registers for general purposes; practically this happens mainly to O registers.

Triggering operations occurs by moving data into the appropriate FU-trigger (T) registers. All operations, including load/store and branch/jump/link occur solely through move instructions. FUs operate concurrently and may incorporate pipelined execution.

As an example we show how a three address register add instruction translates into move operations:

$$\texttt{ADD r3,r2,r1} \quad \Rightarrow \quad \texttt{r1}\rightarrow\texttt{O}_{ADD}\texttt{; r2}\rightarrow\texttt{T}_{ADD} \quad \texttt{R}_{ADD}\rightarrow\texttt{r3}$$

First the values of r1 and r2 are moved to the input registers of the adder. After some time (depending on the latency of the adder) the result can be moved to r3. In practise most FUs implement multiple operations (like add and subtract). Operations are distinguished by mapping multiple trigger identifiers on the same physical trigger register.

An interesting feature of a TTA is the fact that bypassing (or forwarding) of data is programmed, instead of relying on expensive associative bypassing hardware. We call this *software bypassing*. The next example illustrates this:

$$\texttt{ADD r3,r1,r2} \Rightarrow \texttt{r1}\rightarrow\texttt{O}_{ADD}\texttt{; r2}\rightarrow\texttt{T}_{ADD}$$
$$\texttt{ADD r5,r1,r3} \Rightarrow \texttt{R}_{ADD}\rightarrow\texttt{T}_{ADD}\texttt{; r1}\rightarrow\texttt{O}_{ADD}\texttt{; R}_{ADD}\rightarrow\texttt{r3} \quad \texttt{R}_{ADD}\rightarrow\texttt{r5}$$

An FU-result required immediately as operand or trigger only does not have to be moved via a register ($R \rightarrow T$).

When subsequent uses of the same FU share the data of one of the operands, an additional saving in moves is

[1]General-purpose registers can also be expressed as monadic-identity FUs.

possible. This is called *common operand elimination* (e.g. r1→O_{ADD}). It also often occurs that the register $r3$ is not live any more; this allows for the elimination of the move to $r3$ (which we call dead *write back elimination*). If all optimizations apply, the minimal number of moves for an add operation becomes one.

Advantages of TTAs

The main design advantages of TTAs are 1) extreme simplicity and therefore quick design time, 2) great flexibility and performance scalability (transport network and FUs are orthogonal, and can be optimized to the application independently), and 3) short processor cycle times (optimal superpipelining; cycle time is determined mainly by data transport).

Apart from design advantages, TTAs also have advantages with respect to the following issues: 1) better scheduling because the operations are more fine grain, 2) efficient usage of transport capacity, 3) extra fine grain parallelism as a result of separating ALU functionality into its individual components (*FU-splitting*), and 4) less register usage as a result of the mentioned *bypassing*.

3 MOVE32INT

MOVE32INT is a 80 MHz, 32-bit general purpose pipelined processor, designed as a prototype TTA implementation. It contains 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.

MOVE32INT includes advanced features like hybrid pipelined function units (which never overwrite previous results), transports guarded by boolean expressions, and a flexible locking mechanism. Interesting is also the fact that TTAs hardly contain any global control logic. Full details of *MOVE32INT* can be found in [2].

Overview of *MOVE32INT*

The architecture mainly consists of a transport network, controlled by the network controller, and several function units (FUs). The network contains 4 buses; 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.

Figure 4 shows a block diagram of *MOVE32INT*. 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.

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-

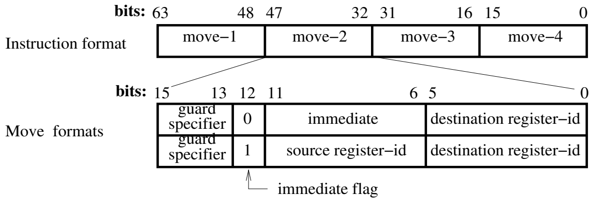

Figure 5: *MOVE32INT* instruction and *move* formats.

bit guard specifier, a 1-bit immediate 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[2].

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[3]. Expressions are useful in guarding nested if-then-else code.

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.

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 6.

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

- Although decoding *move* operations is trivial, a separate stage is needed for the transport of the source and

[2] 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.

[3] 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!

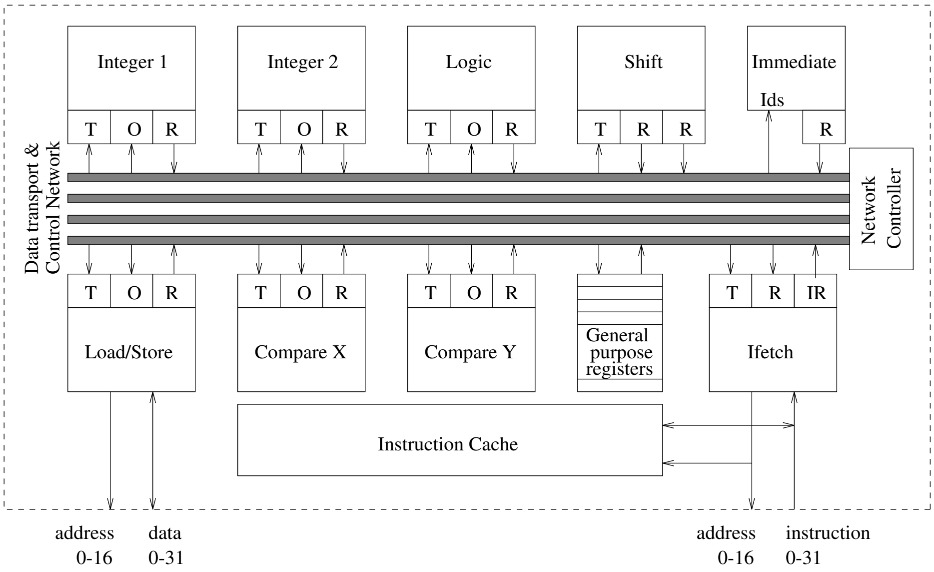

Figure 4: *MOVE32INT*; block diagram.

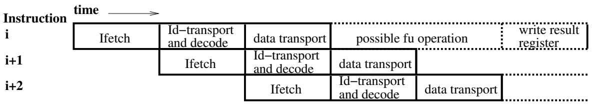

Figure 6: *MOVE* pipelining execution mechanism.

destination register identifiers. Including this transport into the actual *move* stage would unnecessarily stretch the clock period.

- Register fetch, write back, and bypass of data values between FUs are all done during the transport stage; they all have to be programmed explicitly.

- 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 an operand register), performing an operation (writing to a trigger register) to flushing a value from an FU (reading from a result register).

FU pipelining

FU pipelining decisions can be taken completely independent from the transport network. 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 (especially exception support), FUs are implemented using a so-called *hybrid* pipelining mechanism.

Current high-performance architectures generally implement pipelines where either intermediate pipeline stages continue always on each clock cycle to the next stage (e.g. [3, 4]) or intermediate stages continue only in case a new operation is issued (e.g. in the intel-860). 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 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[4]. Hybrid pipelines also reduce register usage by using intermediate stages as temporary storage.

Realization and Floorplanning

MOVE32INT is realized on a 2μ Sea of Gates (SoG) image with an area of $1cm^2$. It contains 200k transistors (of which about 50% is used) with a channel length of 1.6μ. Wiring is done using 2 layers of metal.

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.

The floorplan of the *MOVE32INT* chip is very simple and reflects the structure of a TTA. The datapath containing the FUs, registers, and move buses are placed into one strip on the top side of the chip. Strips for instruction decode and stage controller are also shown. In the middle a vertical channel is saved which leads the instruction address (PC)

[4] Even before the result has been calculated.

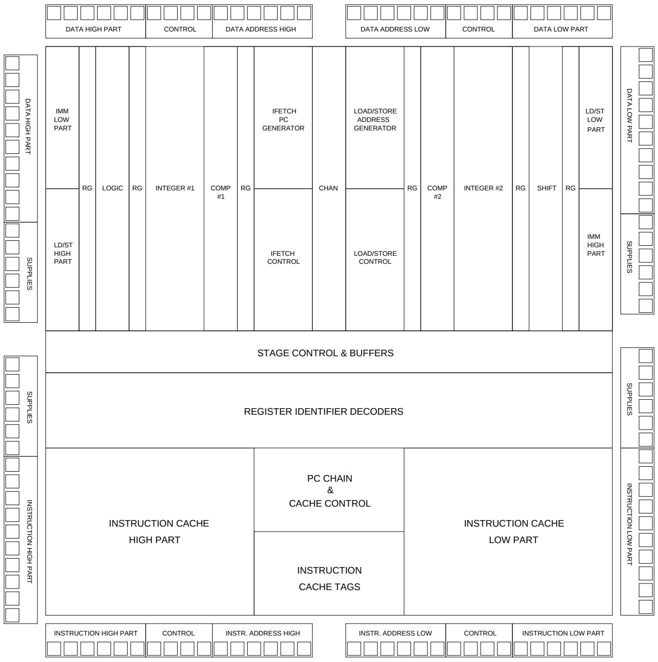

Figure 7: Floorplan of *MOVE32INT* on the available image

downwards to the cache and the data address upwards to the pads.

Critical path analysis
The target speed for the sea of gates realization is 80 Mhz, corresponding to a cycle time of 12.5 ns. The high phase of the clock is limited to a minimum of 4.0 ns which is constrained by the instruction cache address decoding logic. The low phase has a minimum of 8.5 ns; it is constrained by the control of the data transport (especially the locking mechanism). Timings are verified by spice simulations of extracted circuits at 125 degrees Celsius. A redesign of the (global) network control circuitry could enhance the cycle time a little bit. Further enhancements require a major change in the locking mechanism; e.g. by using delayed locking (locking in the next cycle).

4 Automated design

In the former section we showed the design of a Sea of Gates version of a TTA. To make circuits, the designer only had to add the metal-1 and metal-2 layers, and corresponding connections to it. However it still takes a reasonable amount of time to design a chip with the complexity of *MOVE32INT*. What we aim at in our project is the automated design of processing systems based on TTAs. To this purpose we are developing a so called *MOVE framework* as shown in figure 8. The purpose of this framework is that a user can, starting from an application description in a high level language (currently C or C++) together with cost and performance constraints, quickly iterate to a satisfactory solution for the mapping of his application on a combination of hardware and software. The framework should be able to a) guide the user through the architectural design space (this is the function of the optimizer), and given the required architectural design parameters deliver b) the chip layout (performed by the hardware framework), and c) the compiler for this dedicated TTA (this is the responsibility of the software framework, see [5]). For this paper we will elaborate on the hardware framework.

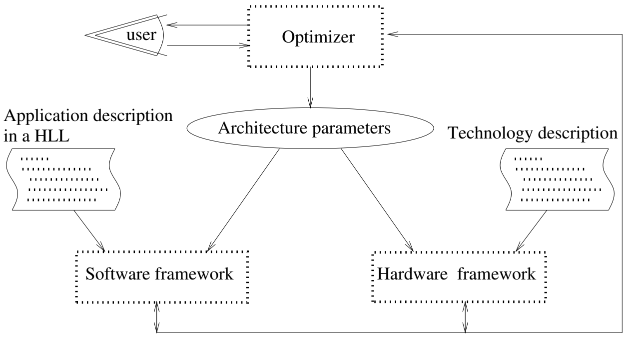

Figure 8: MOVE framework.

The regularity of TTAs allows the automation of the hardware design process. TTAs are constructed using a restricted number of building blocks. Figure 9 illustrates the hierarchical structure of building blocks as used in *MOVE32INT*. Essentially TTAs are built up by a proper connection of function units (FUs) and bus-connections (input and output sockets). FUs are completely independent of each other and of the data transport network; they only have to apply to the socket interface specifications. FUs can therefore be designed separately, and pipelined independently. Each FU pipeline stage is controlled by a FU-stage controller; a chain of them implements the hybrid pipelining mechanism.

Sockets connect in- and outputs of FUs to one or more data transport buses. They primarily consists of decoding logic (comparators), input multiplexers, and output demultiplexers (bus drivers). The transport network contains besides the wiring (for id's, data and control lines) some global control logic. This logic is responsible for generating locks (e.g. when a function can not yet deliver a result), squashes (e.g. in case of guarded transports) and exception control.

Different TTAs can easily be configured by assembling different combinations of these blocks. We are currently finishing a hardware generator based on these building

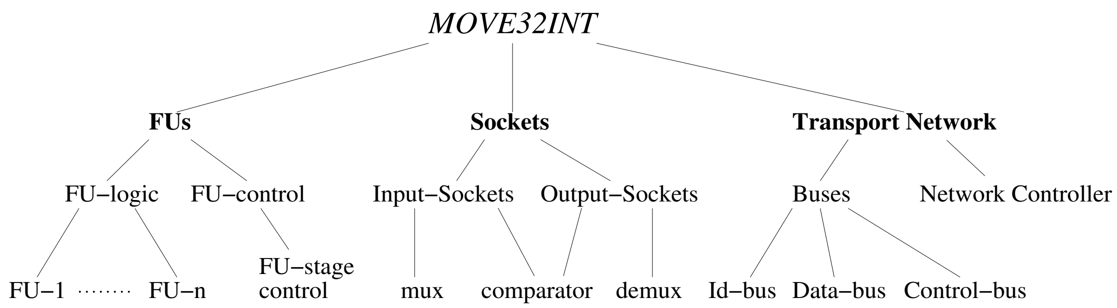

Figure 9: Building block hierarchy for *MOVE32INT*.

blocks using silicon compiler tools. Future papers will report about this work.

5 Conclusions

In this paper we discussed the design of transport triggered architectures, and presented a prototype TTA implementation, *MOVE32INT* which exploits instruction level parallelism by issuing four data transports per cycle and letting many FUs operate concurrently.

TTAs have many advantages as compared to traditional architectures; these advantages are both at the design level and the hardware exploitation level. The price to be paid is a more complex code compilation process, because the compiler is not only responsible to schedule operations, and bind operations and variables to function units and registers, but it now also has to schedule transports and bind them to appropriate data transport buses (see e.g. [6, 7, 5]).

In the design of *MOVE32INT*, we demonstrated the feasibility of TTAs, the potential for very high performances, and its reduced complexity. 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, and the use of fixed size (and relative large) transistors, we were able to achieve a very high clock rate of $80MHz$, resulting in a theoretical performance of $320Mops/second$.

We can compare our design to a full custom RISC processor we have made (see [8]), 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.

TTAs are highly regular; they are 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. We will explore this in a future processor generator which is part of our *MOVE framework*. Using this framework users can quickly design ASPs including the necessary system software.

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] 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.

[3] 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.

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

[5] Jan Hoogerbrugge and Henk Corporaal. Transport-triggering vs. operation-triggering. In *Compiler Construction conference CC-94*, 1994.

[6] Henk Corporaal. Evaluating Transport Triggered Architectures for scalar applications. *Microprocessing and Microprogramming*, 38:45–52, September 1993.

[7] Jan Hoogerbrugge, Henk Corporaal, and Hans Mulder. Software pipelining for transport-triggered architectures. In *MICRO-24*, Albuquerque, November 1991.

[8] 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.