TTA Processor Synthesis

Henk Corporaal and Reinoud Lamberts

Abstract

Due to the decreasing feature size of VLSI technology, the amount of hardware which can be integrated into a single chip increases. As a result, future processor chips can execute tens of operations concurrently. Many applications can profit from these huge amounts of hardware parallelism by designing application specific processors. Two problems emerge however: 1) the design space gets large; it is difficult to chose a satisfactory solution, and 2) the design complexity increases and therefore design cycle gets too long.

Solving these problems makes tools for automatic processor design necessary. In this paper we present such a tool. It generates processor layout for so called *transport triggered processors*. The processor configuration is largely parameterized. A working processor has already been produced using this generator.

An important question concerns the quality of automatically generated designs: what do we pay in area and cycle time, when compared to a manual design? In order to answer this question, these two design styles are compared for an almost equal processor configuration.

Keywords: synthesis, transport triggering, silicon compilation, sea of gates.

1 Introduction

To combine the advantages of general purpose processors, DSPs and ASICs we propose to use ASIPs, Application Specific Instruction set Processors. ASIPs are 1) fully programmable, i.e. high level language applications can be compiled efficiently for these processors, and 2) can be synthesized automatically from an architecture template which covers a large processor design space. The proposed ASIPs have a *transport triggered architecture* (TTA) [1, 2, 3]. TTAs can be easily configured for vastly different application requirements and can be scaled over a wide performance range.

This paper presents a *processor synthesis tool* which is part of the MOVE framework, a design environment for automatic generation of hardware and software for application specific systems. Given a set of architecture parameters describing a TTA this tool automatically synthesizes a transport triggered processor. However, the area of the generated layout and the achievable cycle time can not (yet) compete with those of a manual design. To investigate these issues we make a comparison based on the MOVE32INT processor. A manual design of this processor was done earlier at our institute [4]. A similar processor can also be generated with our processor synthesis tool. It will be shown that there is still a large gap between these two design methods, however, use of adequate techniques certainly reduces this gap.

The paper is structured as follows. In section 2 the processor synthesis process is described. Section 3 describes details of the processor pipelining. Section 4 makes a comparison between automatic synthesis and manual design. It will outline several problems related to automatic synthesis. Section 5 describes methods to (partly) overcome the observed problems. Finally, section 6 concludes this paper.

2 Processor synthesizer

The processor synthesizer generates application specific, single chip TTA processors. The synthesizer accepts architecture parameter values, VLSI technology information and a cell library as input, and produces layout of the generated processor as output together with information about area and timing; see figure 1.

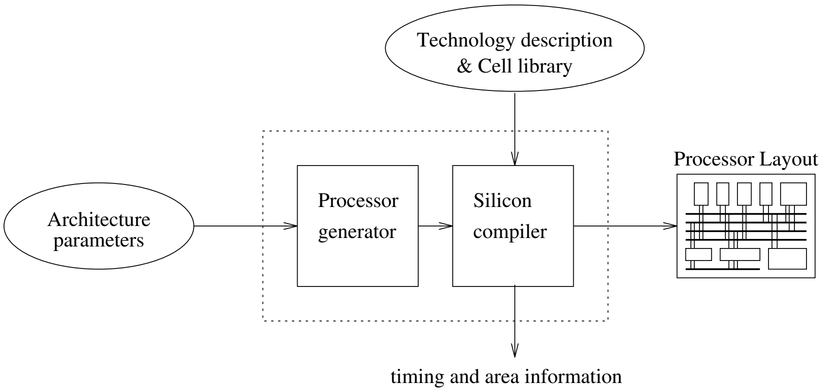

Figure 1: The processor synthesizer

The processor generator translates the architecture parameter set into a description in SID (a HDL) that can be synthesized by the silicon compiler (Sagantec's ASA[1]). The silicon compiler produces layout based on this processor description.

The synthesizer uses a VLSI library of pre-designed cells to generate a processor. These cells can either be standard cells used by standard cell routing tools, or user designed basic cells used for generating regular structures (like memories). Currently only cells provided by Sagantec are used.

Design space. The main architecture parameters supported by the processor synthesizer are:

- Bus widths (e.g. databus, $\geq$ 3 bits).
- Number of *move* buses ($\geq$ 1).
- Connectivity of the *move* buses; each connection between a socket and a specific data bus can be specified individually.
- Number and type of FUs; figure 2 shows most of the units which are currently supported.
- FU specific options, like cache size (for the instruction fetch unit), use of bypassing within the general purpose register file, use of scan registers for testing purposes, etc.
- Number of register files, and number of registers for each register file.
- Guarding (conditional execution) for each *move* separately, or for entire instruction.

The processor structure of the synthesized processors is shown in figure 2. This figure includes most of the supported FUs.

[1]ASA is a trademark of Sagantec Europe b.v., Eindhoven.

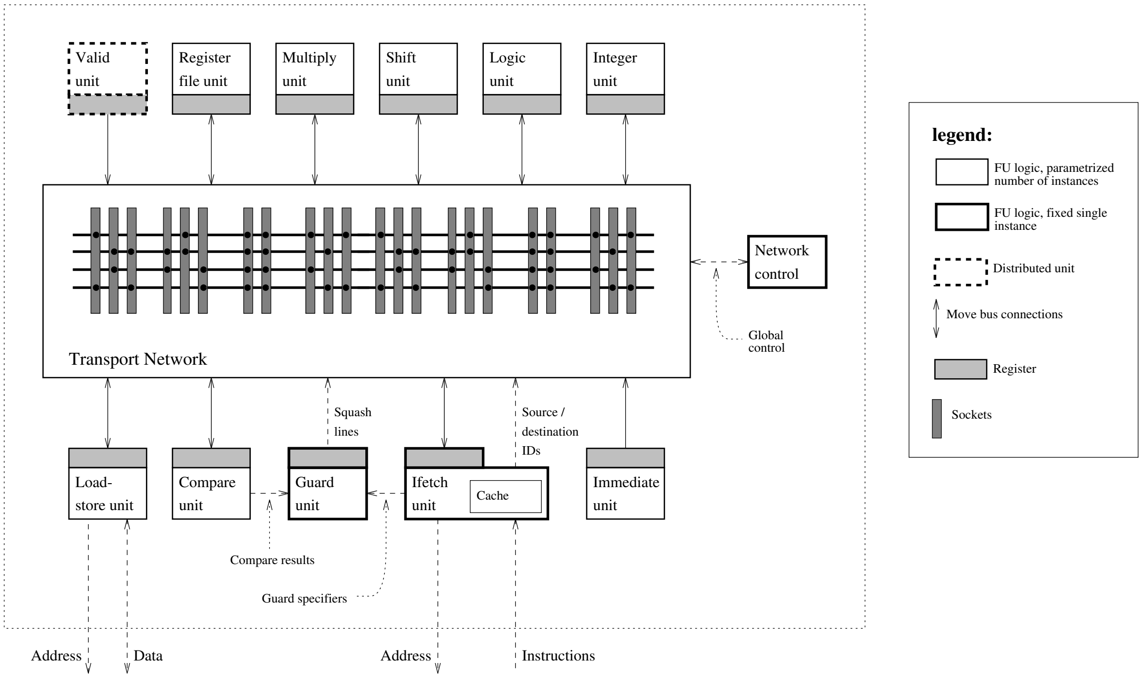

Figure 2: Processor structure and FU overview

3 Processor pipelining

In TTAs FUs, which perform the operations, and the interconnection network, which handles the data transports, are separated from each other. To enhance performance, both can be pipelined separately. We shortly describe these pipelining mechanisms, and their related control.

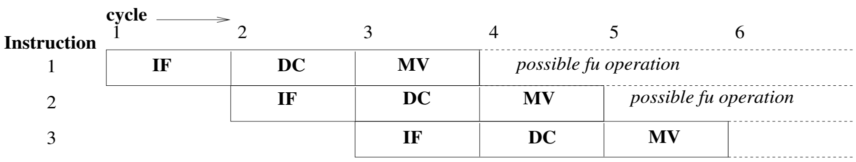

Figure 3: Pipelined execution of instructions.

Transport pipelining Figure 3 shows the pipelined execution of successive instructions, as implemented by the processor synthesizer. Three stages are shown: IF (Instruction Fetch), DC (DeCode), and MV (MoVe). The actions performed during stages are visualized in figure 4. During the IF stage, instruction cache lookup is done; during the DC stage, source and destination IDs are distributed to and decoded on the sockets; during the MV stage, the decoded signals are used to set up the *moves* and the actual data transfer occurs. If the *move* happens to be a trigger-move, i.e. a move of an operand value to the trigger input of a FU, the FU starts operating.

FU pipelining Several alternatives exist for pipelining FUs [2]. E.g. pipelines may continue always (latch each cycle), or continue only in case the transport network reads from or writes to the pipeline. We have chosen a hybrid solution, called *hybrid pipelining*. It is chosen to maximize

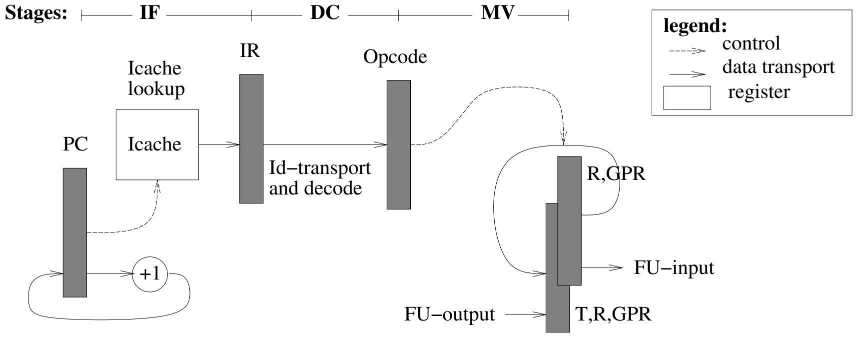

Figure 4: Pipeline stages in a 3-stage pipelined TTA.

scheduling freedom. Hybrid pipelines operate according to the rule: pipeline stages advance if they can do so without overwriting other (intermediate) results. So a data item in a stage will be held there as long as necessary, and advances to the next stage as soon as that stage becomes free.

Pipeline control FU pipelines and the transport pipeline have to synchronize. FU pipeline operation depends on whether or not a move operation specifies a transfer to or from a FU (e.g. final FU stages halt until a result is read). This also works the other way: the transport network pipeline may have to *lock* to synchronize with a FU (e.g. when attempting to read a result that has not reached the final FU stage). The transport pipeline is coupled to FU hybrid pipelines when specified so by move operations; it may be argued that the transport pipeline temporarily becomes part of the hybrid pipelines. It is possible, though, to specify move operations that imply coupling of the transport pipeline to a FU pipeline at a point where it makes no sense (e.g. when attempting to read from an empty hybrid pipeline). In this situation, synchronization is not possible and an *exception* is raised.

The relation between the transport network control and FU hybrid pipeline control is given in figure 5. As shown each socket and FU-stage has a corresponding controller attached to it (these controllers are standardized building blocks). The FU-stage controller generates the stage enable signals (*en*). The socket controllers generate the lock and exception requests signals which are communicated with the network controller; the latter might generate global lock and exception signals. Locks and exception requests are not generated when the move operation activating these requests is squashed. As shown, squashing of moves, caused by guards which evaluate to false, is handled within the sockets.

4 Synthesis vs. manual design

A comparison for area and timing is made between a manually designed and a synthesized MOVE processor. The characteristics of the two designs, which both implement a 32 bit processor with 4 *move* buses (the so-called MOVE32INT configuration), are:

1. Sea of Gates version (SoG-version). This version is realized using a SoG-image in 1.6μm CMOS, containing about 190k transistors. Most circuits are designed and realized manually.
2. The version generated by the MOVE Processor Generator (MPG-version). This version makes use of the ASA silicon compiler from Sagantec. The MPG uses the predefined ASA

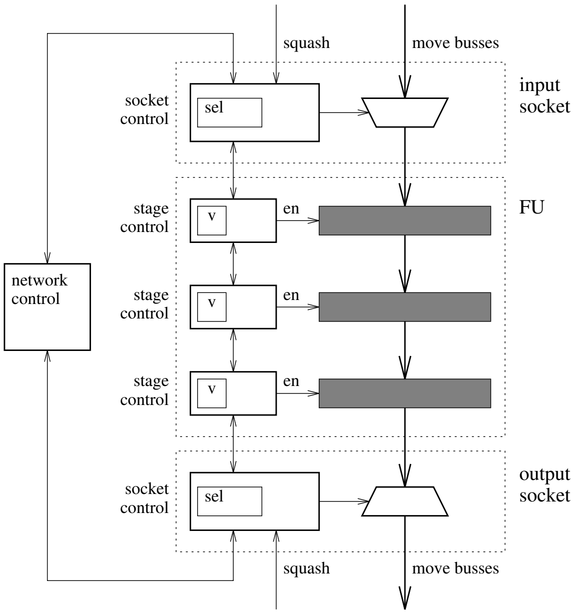

Figure 5: Pipeline control structure

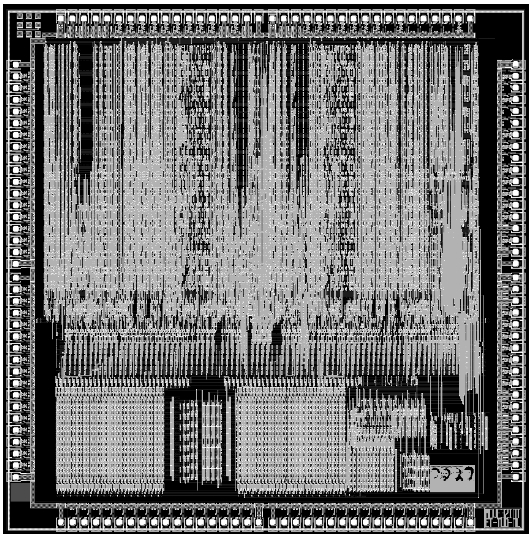

Figure 6: SoG MOVE32INT layout

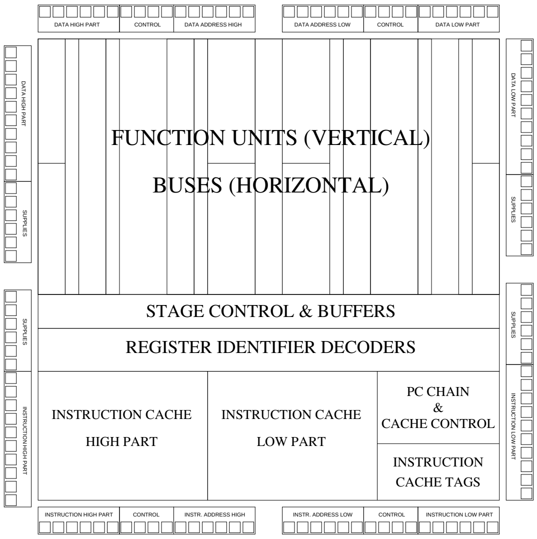

Figure 7: SoG MOVE32INT floorplan

library cells. The processor can be realized in different technologies (e.g. CMOS 1.5, 1.2 and 1.0μm).

Figure 6 shows a layout plot of the SoG processor, with accompanying floorplan in figure 7. Layout for the MPG version is shown in figure 8; the standard cell blocks are not expanded here. Figure 9 shows a small piece of standard cell layout, as it is found inside the blocks in figure 8.

Resulting area requirements and timing characteristics for these two processor realizations are summarized in table 1. More details on these results are presented in section 4.1 (area) and section 4.2 (timing).

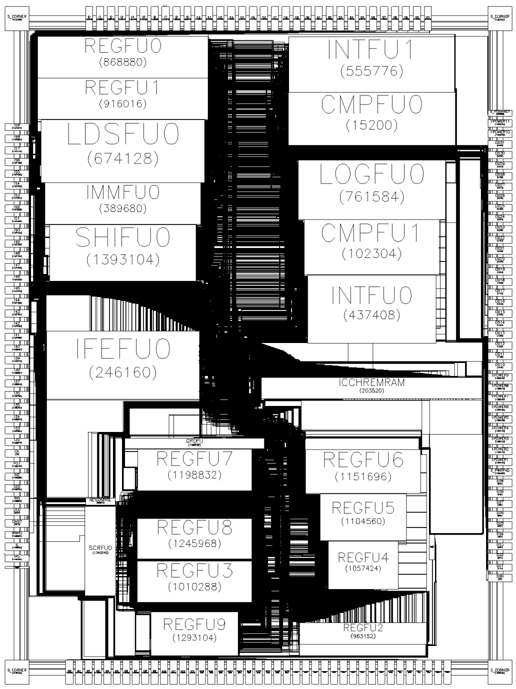

Figure 8: MPG synthesized MOVE32INT layout

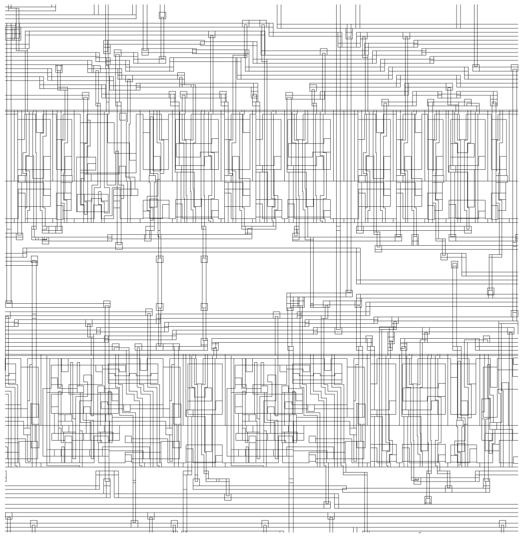

Figure 9: ASA standard cell layout

Realization	Area	Clock period
SoG 1.6μm	100 mm^2	12.5 ns
ASA 1.5μm	261 mm^2	37 ns

Table 1: Area and timing of the synthesized (ASA) and the manually designed (SoG) processors

4.1 Area comparison of MPG versus SoG

This section compares the area requirements of the synthesized and SoG MOVE processors. The configuration of the synthesized processor is similar to, but not exactly the same, as the SoG version; the additional functionality of the synthesized processor (such as additional sockets on the compare units, used for exception state saving) adds to the area requirements. The area increase for this additional functionality is around 10%.

When comparing both designs, consider that basic differences between the layout generated by the ASA silicon compiler, and manually designed Sea of Gates layout, strongly influence area. ASA uses full custom layout, and can generate this layout for any characterized technology. However, the possibilities of full custom design are used only to a limited extend. Limitations of MPG design with ASA, when compared with manual design, are:

- Inefficient placement. Even with standard cell layout, much better area use must be possible. About 25% of the total area, used for global routing, seems wasted because of sub-optimal global placement of standard cell blocks, see figure 8. Another 25% is wasted on routing inside the standard cells, partly due to inefficient global placement, partly due to the low number of gates for some standard cells.

- Inefficient routing approach (e.g. no routing of buses over units).
- No support for dynamic circuits (so large static registers are required, no wired-or circuits).
- Fixed gate fanout (so many transistors are larger than necessary).

As an example, lock and exception requests must be gathered individually from the FUs and combined using gates, lacking a means to realize wired-or. Using wired-or circuitry, only a single *or* line would be required.

Sea of Gates technology, on the other hand, fixes transistor size (and even place), but the use of manual design (dynamic circuits, routing of buses over FUs etc.) allows for many improvements on area use.

The difference in area for the two designs is quite large, see table 1. Much of the area required for the synthesized processor is spent on routing (and wasted on placement), as is obvious from figure 8; inside the blocks (*standard cells*, not expanded in this figure) similar inefficient routing is often found. About 50% of area improvement can be obtained by simply improving placement and better distribution of logic over the standard cells.

4.2 Timing comparison of MPG versus SoG

This section compares the timing of the synthesized and SoG MOVE processors. The SoG version is designed to run at 80 MHz; the MPG version in a similar CMOS process runs at 37 MHz. In both versions it appears that the control determines the achievable cycle time by a large margin over other paths.

Comparing the two realizations of MOVE32INT, we see that their critical paths are fairly similar, however, the SOG-version is much faster (about a factor of 2.5). This is mainly caused by the following:

1. Use of dynamic logic in the SOG-version. The MPG-version uses the static CMOS gates supported by the ASA silicon compiler.
2. Manually fine-tuning of the control path for the SOG-version. Doing this for the MPG-version will likely reduce the critical path with several nanoseconds. However, in the MPG one is practically constrained to using only one clock phase for timing reference (a requirement of the ASA timing analyzer).
3. In the SoG-version the FUs do not generate exceptions when the reading from an empty pipeline or writing to a full one. If we remove the critical exception path from the MPG-version the next critical path is the *lock-path* (which is slightly shorter, about 1 ns).
4. The MPG-version uses ASA RAMs for the instruction cache. Usage of these synchronous RAMs required an extra address selector in the critical path (adding about $2ns$).

Both versions have the problem of fixed transistor sizes (this is obvious for SoG; ASA uses a fixed fanout for its standard cells). Also, for both versions holds that the actual data transport takes far less time than required by the critical control paths. Section 5 will present new approaches for improving MOVE processor timing by reducing control path delays.

5 New developments

In chapter 4, a synthesized MOVE processor appeared to be less efficient (in area and timing) than a manually designed processor. This result is not surprising; synthesis obviously reduces the design effort, but it also reduces opportunities for optimization. Several causes for the observed inefficiency were exposed, however, providing directions for improvement. This chapter discusses the possibilities for improvement and presents some results that have already been obtained. Section 5.1 discusses area, and section 5.2 discusses timing issues.

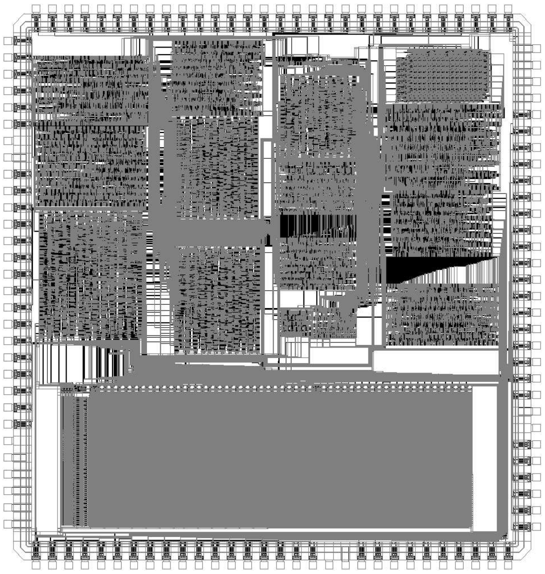

Figure 10: Phoenix synthesized processor layout

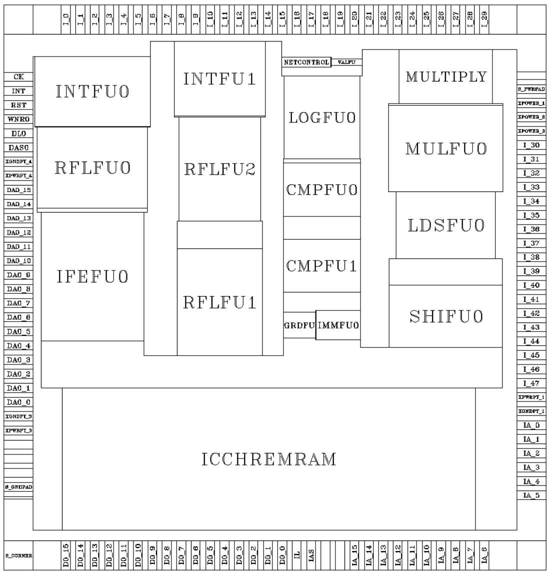

Figure 11: Phoenix processor floorplan

Some improvements with respect to the design presented in chapter 4 have already been applied to a design (called Phoenix). The Phoenix processor is a 16 bit, 3 *move* bus configuration; it has actually been processed through Eurochip (ES2 1.0μm). The fabricated processor has been functionally tested and works correctly; it runs at 44 MHz. An impression of the layout is shown in figure 10, where the two metal layers are plotted; figure 11 shows the floorplan. The block at the bottom (ICCHREMRAM) is the (512 line, 3 kbyte) cache RAM; the die area is 64 mm^2.

5.1 Area improvements

In section 4.1, a number of causes for area inefficiency of the MPG were mentioned. Below, these causes are listed again, with ways to improve on area use.

Inefficient placement. For the Phoenix design, fewer standard cells have been used, with more logic per standard cell. This way, the density of the standard cells increases (less local routing overhead) and less (inefficient) global routing is needed. Also, better global placement for the standard cells has been used; the standard ASA placement algorithms appear to be not well suited for the MPG. Experiments show an area improvement of 50% to 60% when doing better placement.

Inefficient routing approach. A promising approach for routing is the use of datapath layout, where buses run through or over function units, with all units designed (synthesized) to fit to the bus routing channels. It would require significant effort to implement this in the MPG though.

No support for dynamic circuits. It is possible to introduce dynamic circuits in the MPG by (manually) designing the required cells and adding these to the libraries. However, characterization of the cells, simulation and timing analysis then become the responsibility of the designer. This adds significantly to the design effort (and risk), making the design trajectory less attractive in comparison with synthesis using static logic.

Fixed gate fanout. There is little opportunity to circumvent the technology mapping done in ASA, which assumes all gates have the same fanout. It is possible though to choose another (global) fanout value than the default one, which may help for some designs (this might also improve timing).

5.2 Timing improvements

There are two basic approaches for improving the timing of the synthesized MOVE processors: by changing the implementation and realization, or by changing the architecture. Changing the implementation (e.g. by moving logic to the other side of a latching point) or realization (e.g. use of dynamic logic) provides an obvious path to improvement by optimization. By inspection of critical paths, the potential for improvement by changing the implementation is estimated at about 10%. If realization is changed (dynamic logic, more active clock phases), the potential for improvement is higher: the comparison with a SoG design suggests a factor of two to three, see section 4.1. However, changing the realization this way requires a lot of effort (comparable to manual design effort), and is partly outside the scope of our current tools (ASA).

Significant improvements in timing are probably easier to achieve through architectural changes. Below the major opportunities for reducing critical paths are listed:

No FU pipeline exceptions. The FU pipeline generated exception path creates a dominant critical path, just for signaling errors that could have been detected at compile time with little effort.

Squash in decode stage. Currently, squashing of *move* operations (e.g. for guarded moves) is done in the move stage, which is quite late. All FU actions depend on the squash signals (because all *move* operations depend on them), so a lot of signals have to be evaluated *after* the squash values have become available. If squash signals are distributed during decode, the operations to be performed during the move stage are completely determined beforehand (except for global locking or exceptions, which simply disable execution of the entire processor).

Simpler FU pipelining scheme. The hybrid FU pipelining currently used in the MPG requires rather complex control. By implementing a pipelining scheme with simpler control (e.g. virtual time latching: continue every non-locking cycle), less time is required to lock the pipelines.

The items mentioned above may be sufficient to reduce the control path to less than the data transport path in the (MPG) processor.

6 Conclusions

Application Specific Processors (ASPs) can be more cost-effective than standard components. The primary obstacles for using ASPs, however, are the associated design effort and time. Because of these, there is a long time to market, engineering cost is high, and only limited architecture optimization is possible.

The synthesis tool presented here is an attempt to overcome these obstacles. The MOVE Processor Generator (MPG) synthesizes processors from an architecture description. The synthesized processors are based on the highly flexible Transport Triggered Architectures (TTAs).

The MPG is capable of correctly synthesizing a wide range of processors. There is a problem when doing synthesis, though: quality of the resulting design is usually not as good as what can be achieved with a manual design. This is confirmed by the comparison of a synthesized processor with a, similarly configured, manually designed processor. The synthesized processor required 2.6 times the area and 3 times the cycle time of the manually designed processor, which was realized in sea-of-gates technology.

This result is not as discouraging as it may seem. The MPG, used to obtain this result, was only a first attempt at synthesizing TTAs. By improving placement, 50% improvement on area can easily be obtained. Also, hardly any critical path improvement has been done on the MPG. Some 10% improvement can be readily obtained here. Much more improvement on timing (about 50%) is possible by changing the architecture; this is relatively easy to implement in the MPG. Several other improvements are possible, such as datapath routing, but these require significant effort to implement. Still, it is more attractive to spend the effort on a synthesizer than on a single processor design.

Even if some design quality advantage remains for a manually designed processor, such a processor must compete with a processor that can be generated (and brought to production) almost immediately, with almost no application specific design effort. The advantages of a synthesized processor on time to market, process technology, and the very low engineering cost are likely to overcome the performance advantage of a manual design for some specific technology.

Actually, the automatic generation of processors allows for extensive searching of the architecture design space for better solutions. The possibility of doing such a search may compensate for a performance difference which is found on the *same* architecture when comparing synthesized with manually designed processors.

References

[1] Jan Hoogerbrugge and Henk Corporaal. Automatic Synthesis of Transport Triggered Processors. In *First Annual Conf. of ASCI*, May 1995.

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

[3] Henk Corporaal. Design of Transport Triggered Architectures. In *Fourth Great Lakes Symposium on VLSI*, pages 130–135, Univ. of Notre Dame, Indiana, March 1994.

[4] Henk Corporaal and Paul van der Arend. MOVE32INT, a Sea of Gates realization of a high performance Transport Triggered Architecture. *Microprocessing and Microprogramming*, 38:53–60, September 1993.