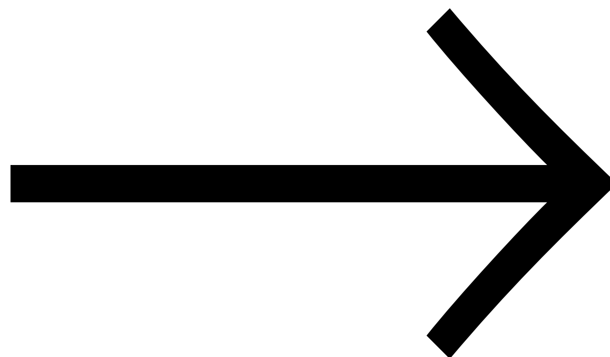
A MOVE Processor Generator

Reinoud Lamberts



Delft, November 1993

Delft University of Technology
Faculty of Electrical Engineering
Computer Architecture Department

Preface

Computers are changing. This should not be surprising, as the technology available to realize computers is changing rapidly. The performance of electronic circuits is still increasing at an exponential rate. Compiler optimization technology has taken off with the insights provided, and challenges posed, by the RISC concept.

Applications of computers are changing too. Larger problems can be tackled with each new computer generation. Prices are dropping, making a wider range of applications possible. More sophisticated software allows for faster implementation of applications, and makes more complex systems feasible.

But what is the most efficient computer design? How can we best use available technology? How well can a compiler generate code for target hardware? What are the requirements of applications? These questions are faced by computer architects, who try to find the best solutions [10]. Designing a computer architecture, however, obviously means designing for a moving target!

At the Computer Architecture Department of the Faculty of Electrical Engineering, Delft University of Technology, a processor architecture has been developed called MOVE. The MOVE architecture can be expanded to make use of larger, faster electronic circuits, without running into bottlenecks. It can adapt to the requirements of different applications, with performance or cost as possible constraints. The MOVE architecture allows for very efficient use of hardware resources, and is a good target for advanced compiler optimizations. For these reasons, the MOVE architecture is an excellent foundation for a framework that can quickly produce a processor which is adapted to specific requirements. Such a MOVE framework is currently under construction at the Computer Architecture Department.

A MOVE processor generator has been constructed, which takes parameters defining a specific MOVE architecture instance as input. The output of the generator can be actual chip layout, and characteristics of the processor such as timing and die area can be determined. Functionality of the processor can be verified by simulation. The basic tool used for the MOVE processor generator is Sagantec's ASA silicon compiler.

This report describes the MOVE processor generator. It describes the MOVEASAINT (MOVE–ASA–INTeger) parametrized architecture, and the implementation of the processor generator. Resulting MOVE processors are evaluated and compared to another realization (under construction), the Sea of Gates MOVE32INT.

A processor similar to the SoG MOVE32INT processor (32 bit, 4 parallel move operations) has been generated with ASA for a 1.0 μ CMOS process; it requires 1 cm^2 die area and runs at 43 MHz.

The MOVE processor generator is the Master thesis project of the author. Supervisor to the project is professor A. J. van de Goor; Henk Corporaal is mentor. I want to thank them, not in the last place for their forgiveness towards a pig-headed student. The contribution of Henk to the work described in this report is extensive. Special credit goes to Rogier Wolff, who has provided much-needed support during difficult times with ASA. Paul van der Arend answered a lot of questions about the MOVE32INT for me. Jean Paul van der Jagt has shown admirable patience with me, and helped me coping with 'his' workstations. Hans Mulder has been most inspiring; alas, he collaborates

with the enemy now. Other people (who are or were at the Computer Architecture Section) I would like to thank are Sietze Schukking, Robert Portier, Andy Verberne, and Jan Hoogerbrugge.

Paul Stravers taught me a lot about VLSI design, and has been an invaluable source of information on too many subjects to mention. Thomas Okken has been both a friend and an ever surprising source of insight; I wish him well on his new academic venture. Jan Carel has always been more a friend than a fellow student. Bas van Dam, Robert van Muyen, and Eric Noordanus took off in other directions, but always came back to share their experiences. Martijn Zwartjes reassured me that being crazy is okay, which was a relief. Merik Voswinkel displayed his anal retentive personality in such a disgusting way that I hurried back to Delft to do this project, which I enjoyed very much. Fred van Kempen taught me about UNIX and lent his wonderful modem through which countless megabytes went during this project. Jeffrey Kingston deserves respect for his elegant Lout document formatter, and his personal support for it (right, this report is *not* formatted with TeX). And this is the right place to thank Vincent Vermeer who was my first, and admirably patient, teacher in electronics.

Very special thanks go to my parents. Without their unconditional support, I would never have made it this far. They are mentioned last in the list of thanks, but they rank first without any doubt.

Contents

Abstract 1

Chapter 1. Introduction 2

1.1. MOVE architecture 2

1.2. MOVE framework and processor generation 3

1.3. Problem and solution overview 3

1.4. Report overview 4

Chapter 2. MOVE architecture concept 5

2.1. Transport triggered architectures 5

2.2. Functional units with hybrid pipelining 7

2.3. Connections between FUs and transport network 8

2.4. Distinguishing MOVE architecture features 9

Chapter 3. MOVEASAINT architecture 11

3.1. Architecture requirements 11

3.2. Architecture overview 12

3.2.1. Summary 12

3.2.2. Register assignment 16

3.2.3. Instruction format 18

3.3. Changes versus MOVE32INT 21

Chapter 4. Functional units 23

4.1. Instruction fetch unit 24

4.2. Guard unit 29

4.3. Immediate unit 31

4.4. Load-store units 32

4.5. Compare units 34

4.6. Integer units 36

4.7. Logic units 38

4.8. Shift units 40

4.9. Valid unit 42

4.10. Register units 43

4.11. Scan register units 44

Chapter 5. Pipelining and control 45

5.1. Data transport network 45

5.2. FU hybrid pipelining 45

Chapter 6. Implementation 52

6.1. ASA introduction 52
6.2. MOVEASAINT system hierarchy in SID 53
6.3. Limitations of SID 55
6.4. Parametrizing SID 56
6.5. Details of MOVEASAINT implementation 58
6.6. Limitations of MOVEASAINT implementation 60

Chapter 7. Realization 61
7.1. Design details 61
7.2. Design verification 63
7.3. External interface 64

Chapter 8. Evaluation of realization 68
8.1. Area evaluation 68
8.2. Timing evaluation 70
8.3. Scalability evaluation 73

Chapter 9. Evaluation of architecture 74
9.1. MOVE architectures in general 74
9.2. MOVEASAINT architecture 75

Chapter 10. Conclusion 77

References 78

Appendix A. Parameters 80

Appendix B. Timing details 89

Appendix C. Layout results 94

Appendix D. Sources 99

Abstract

At the Computer Architecture Department of the Faculty of Electrical Engineering, Delft University of Technology, a processor architecture has been developed called MOVE. It is a transport triggered architecture: instructions primarily specify data transports (*moves),* and can imply an operation to be performed. The MOVE architecture can easily be extended with new functionality, adapts to a wide range of performance and cost requirements, and can scale up to large high performance designs without running into bottlenecks. The architecture allows for very efficient use of hardware resources. For these reasons, the MOVE architecture is an excellent foundation for a framework that can quickly produce a processor which is adapted to specific requirements. Such a MOVE framework is currently under construction at the Computer Architecture Department.

A MOVE processor generator has been constructed, which takes parameters defining a specific MOVE architecture instance as input. The output of the generator can be actual chip layout, and characteristics of the processor such as timing and die area can be determined. Functionality of the processor can be verified by simulation. The basic tool used for the MOVE processor generator is Sagantec's ASA silicon compiler.

This report describes the MOVE processor generator. It describes the MOVEASAINT parametrized architecture, and the implementation of the processor generator. Resulting MOVE processors are evaluated and compared to another MOVE processor realization (under construction), the Sea of Gates (SoG) MOVE32INT.

A processor similar to the SoG MOVE32INT processor (32 bit, 4 parallel move operations) has been generated with ASA for a 1.0 μ CMOS process; it requires 1 cm^2 die area and runs at 43 MHz.

Chapter 1. Introduction

In this chapter, the first two sections give a general introduction to the MOVE architecture and framework, and indicate how the MOVE processor generator fits in. Section 1.3 discusses what may be learned by building a MOVE processor generator, and how the problem was approached. An overview of this report can be found in the last section.

1.1. MOVE architecture

Historically, computer architectures have been designed with specific realization technology and application area in mind. This is not surprising: what architecture is best depends strongly on both technology (hardware and software) and application. However, traditional architectures are limited with respect to scalability and are not easily customized to specific application requirements. As a result of this, architectures tend to age quickly and become outdated when they cannot efficiently utilize the benefits of new technology any more, and little use is made of customized processors. The SCARCE processor [1] is a good example of the scaling and customization possible with a RISC architecture.

What makes traditional architectures, roughly classified as CISC, RISC and super-scalar, inflexible? These architectures are designed using a model of sequential execution of operations, and have the operations to be performed specified explicitly in the instruction, while data transport is often only implied. As a result of this, when more functional units are added, or when implementing a RISC as super-scalar, and even when designing a super-scalar or VLIW architecture from scratch but with the traditional architecture model in mind, these architectures all run into the same bottleneck: *data transport.* The traditional model, where operations to be performed can be freely specified but data transport is implied, results in an explosion of data communication requirements when the implementation grows. The cause of this is the lack of the traditional model to reflect the true properties of the realization. It is no longer true that performing an operation on data is more expensive than moving that data. Actually, at the current or soon to be achieved level of parallel execution of operations by microprocessors, if each operation can freely specify any source and destination, the cost of data transport quickly becomes dominant. The number of ports on register files and the number of execution stage bypasses increase rapidly. See [13] and [15] for an impression of how designers try to cope with these problems on (large) high-performance designs.

The MOVE processor architecture developed at the Computer Architecture Department avoids these data transport problems by taking a fundamentally different approach [5]. Instructions primarily specify data transports in stead of operations. By making data transport visible in the architecture, it can be rightly scheduled as the critical resource it is. The data transport network can be configured to specific requirements and does not restrict scalability of the architecture; more instruction level parallelism and more functionality can be added without problem.

1.2. MOVE framework and processor generation

The MOVE architecture can easily adapt to different requirements and evolving realization technology. This property is exploited in the MOVE framework [4], where application performance and cost requirements steer the design of a MOVE processor. Parameters of the architecture are determined automatically as much as possible.

Determination of suitable architecture parameters for a given set of requirements is not trivial; it is not likely that a first estimate of parameters will result in a processor design that is close to (locally) optimal. To achieve results that are as close as possible to the constraints, several design iterations will usually be necessary. At least during the last iterations, detailed analysis of the result is required; the resulting performance of a system at this level of complexity can depend with high sensitivity on small changes in parameters. To be able to analyze the effect of parameter changes, and to be able to provide a design ready for fabrication quickly, efficient (automated) processor generation and characterization is desirable for the MOVE framework.

It is not evident beforehand what is likely to be the best approach to processor generation. Here, too, design iterations may be necessary to achieve satisfactory results. The first route chosen for implementation and realization of a MOVE processor has been Sea of Gates (SoG) design: the MOVE32INT processor, which is still being worked on [3]. Use of SoG in stead of full custom design reduces the design effort significantly. Still, depending on the tools used, the manual design and layout effort can be significant. For the MOVE32INT processor, the NELSIS tools are used, see [2].

1.3. Problem and solution overview

The primary problem addressed in this report is whether a MOVE processor generator based on the ASA silicon compiler would be a viable realization path for the MOVE framework. Additionally, the ASA results are to be evaluated against the SoG MOVE32INT, and the MOVE architecture evaluated for effectiveness.

The processor generator should accept parameters such as bus width and connectivity, number of parallel operations, and number and type of functional units. Characteristics (timing, area) of a generated processor with appropriate parameters are to be compared to those of the SoG MOVE32INT; Also, design ease and flexibility of the ASA MOVE processor generator are to be evaluated.

The problem of designing an effective MOVE processor generator was approached by first defining the functional requirements. It should be possible to generate a processor based on the MOVE32INT architecture [7] using the right parameters; also, a list of required (and some optional) parameters for the architecture was assembled.

The tools (mainly the ASA silicon compiler) were explored by going through the entire design path for a relatively simple problem: an intelligent RAM device implementing Conway's 'Game of Life' cellular automaton. The number of cells and word size were used as parameters, and the SILIFE chip design was the result.

From the requirements, and the experience with the SILIFE chip, a hierarchy of systems was designed for implementation of a parametrized MOVE processor in SID, ASA's hardware description language. The systems were gradually filled in, verifying results at each level. At the highest level, generated processors were simulated using various (if small) test programs. After critical path optimization, layout was generated and characteristics of the generated processors were determined and compared to those of the SoG MOVE32INT processor.

During the work on the SID description, several (undocumented) SID limitations surfaced, which posed a problem for implementing several features in a parametrized way. To cope with these limitations, preprocessing tools have been used (and created).

1.4. Report overview

The first chapter is the one you are reading now, the introduction. Chapter 2 introduces MOVE architectures in general. The MOVEASAINT parametrized architecture used for the ASA MOVE processor generator is described in chapter 3. Functional unit details are provided in chapter 4; details of the pipelining and control mechanisms, important to understand the architecture, are given in chapter 5. Information on ASA's SID language and how it is used, and interesting implementation details of the MOVEASAINT can be found in chapter 6. Details on the realization and its verification are in chapter 7. Chapter 8 discusses the characteristics of the realization and compares it to the MOVE32INT. How the experience gained from the MOVEASAINT realization may influence decisions in architecture is discussed in chapter 9. The conclusion (chapter 10) summarizes the viability of the MOVEASAINT processor generator.

Appendix A contains the parameter declaration file for the processor generator, with the settings for the MOVE32INT configuration. Details of the major critical paths are presented in appendix B. Layout plots can be found in appendix C. Finally, appendix D lists the sources of the processor generator in small print, and provides an index to these sources.

Chapter 2. MOVE architecture concept

This chapter gives a general introduction to MOVE architectures, which represent a special case of Transport Triggered Architectures (TTAs). The contents of this chapter are based on chapter 2 of the MOVE32INT architecture manual [7]. For a more general view on TTAs see [6].

TTAs can be viewed as a number of Functional Units (FUs) performing operations, which are connected by a transport network (see figure 2.1). General purpose registers are considered functional units performing the identity operation. The number and type of FUs can be changed; so can the connectivity and capacity of the data transport network. This way, a range of architectures can be implemented, and TTAs are therefore well suited for being tailored to specific requirements.

MOVE architectures are TTAs which use hybrid pipelining for the functional units. Hybrid pipelining allows for maximum scheduling freedom, decoupling FU pipeline stages as much as possible from transport network data movements.

More details on transport triggering can be found in the next section, and details on hybrid pipelining in section 2.2. The section thereafter discusses FU – transport network connections. The final section of this chapter indicates in what respect MOVE architectures distinguish themselves from other architectures.

2.1. Transport triggered architectures

Traditional architectures (CISC, RISC, super-scalar, VLIW) have the operations to be performed specified explicitly in the instruction, while data transport happens only as a side effect. Data transports are coupled to the operation specified in the instruction, and are necessarily scheduled in time slots imposed by the time of execution of the operation. For example, the instruction *Add* r_1, r_2, r_3 forces data transports scheduled at specific times from and to those registers, and through bypasses if the instruction sequence requires it. Architectures designed this way are called *operation triggered architectures.*

In contrast, *transport triggered architectures* or TTAs invert the traditional programming paradigm by specifying data transports in the instruction, with operations happening only as a side effect. TTAs separate data transport and operations; the data transports are made visible at the architecture level. A transport triggered architecture can be viewed as a set of registers connected in some way by a data transport network.

TTAs are programmed by specifying transport of data values through the transport network. Instructions contain only one (and therefore implied) type of operation: *move.* Generally, functional units are connected to the transport network by three possible types of registers: operand, trigger and result registers. All *moves* are between registers of those types ($r_x \rightarrow r_y$). The function of the register types is as follows:

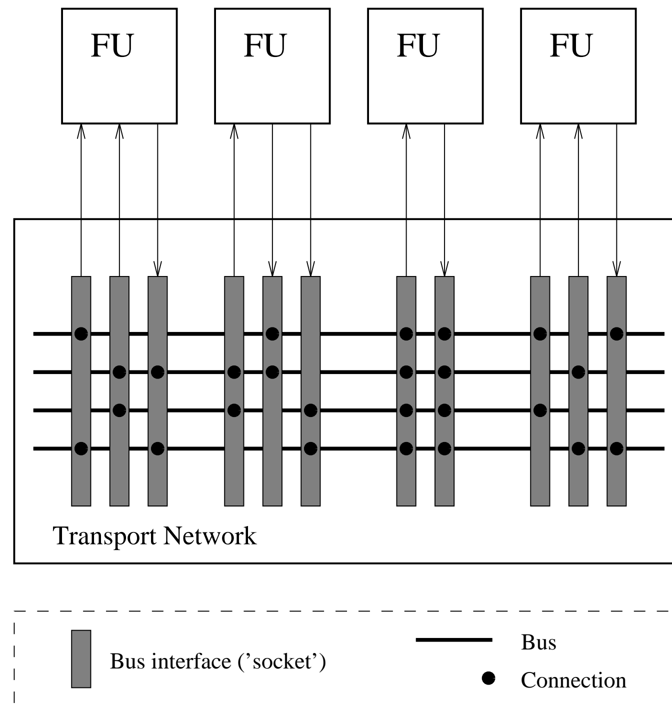


Figure 2.1. TTA with bus transport network

Operand Operand registers are like general purpose registers, accessing an operand register has no other effect than reading or writing its contents. Unlike general purpose registers, though, the contents of an operand register can be used by a FU when an operation is triggered there.

Trigger A *move* to a trigger register starts an operation on the FU the register belongs to. The value moved is used as an operand. If more than one operand is required, these are taken from operand registers. Different operations to be performed on the FU can be selected by accessing the trigger register on different logical addresses.

Result A result register contains the result of a FU operation after the FU pipeline has finished the operation, which has been triggered through a trigger register.

To get more work done for each instruction executed, multiple *move* operations can be executed in parallel in a TTA. TTAs then resemble VLIW architectures in this respect.

A code example is given in figure 2.2 to clarify the concept of TTAs. The contents of general purpose registers r_a and r_b are added, and the result is stored to the memory location with the address contained in r_c. The first instruction contains moves of r_a and r_b to the operand and trigger registers of the integer unit (*Oint* and *Tadd*). The trigger register is accessed at the register address that signifies an addition. Moves to operand registers must occur before or at the same time as moves to trigger registers. The result of the addition becomes available in the result register when the integer unit pipeline has finished. In the example a latency of two cycles for the integer unit is used. The third instruction moves the result directly to the store data register of the memory unit, and moves the address to the store address register. In the example, the data register of the

$r_a \rightarrow Oint$ $r_b \rightarrow Tadd$; trigger add
. . . ; other moves
$r_c \rightarrow Osa$ $Rint \rightarrow Tsd$; trigger store

Figure 2.2. Code for a TTA, adding two registers and storing the result

memory unit is the trigger register, and the address is the operand; this may be reversed though, depending on the design of the memory unit. A store operation on a memory unit usually has no result that has to be read again. Typically, an operation requires about 2 moves to be specified.

To support full programmability, flow control operations, conditional execution and immediates must be supported. Flow control operations on a TTA can be implemented with moves that change the program counter. For example, moving a new address to the program counter implements a jump. Of course, a jump will usually have delay slots associated with it, before execution proceeds from the new address.

Conditional execution of moves can be implemented using guards. For example $g : r_a \rightarrow r_b$, with g defining a guard field in a move instruction, indicates that the move is to be executed only if g evaluates true. The condition can depend on results available in FUs.

Small immediate values can be provided by making (part of) the source register address of move operations available for reading. This can be done with a FU that has result registers carrying source parts of the instruction word. Larger immediates can be used if move operations can be suppressed to allow their part of the instruction word to be made available for reading as data. Effectively, chunks of the move source address space, or chunks of the entire instruction word are used for immediates this way.

2.2. Functional units with hybrid pipelining

Transport triggering makes several compiler optimizations possible, by allowing for explicit scheduling of data transport ([14], [9]). To increase scheduling freedom for the compiler, most FUs in MOVE architectures are implemented using *hybrid* pipelines. Usual ways to implement pipelines are:

Continue Always Pipeline stages advance in each clock cycle, or, more useful, in each cycle in which the instruction pipeline does not stall.

Push–Pull Pipeline stages advance only when a new item is inserted in or removed from the pipeline.

Continue always pipelines do not allow for much scheduling of data transport: results emerge, and must be used, at a fixed number of cycles from the start of the operation. Push or pull pipelines allow for more scheduling freedom but require large overhead when use of the pipeline is not needed each cycle (a likely situation in a TTA with several FUs for different types of operations). Therefore hybrid pipelines are used in MOVE architectures:

Hybrid Pipeline stages advance if they can do so without overwriting other (intermediate) results.

Figure 2.3 shows an example of a FU with hybrid pipelining. The pipeline in the figure has 3 stages

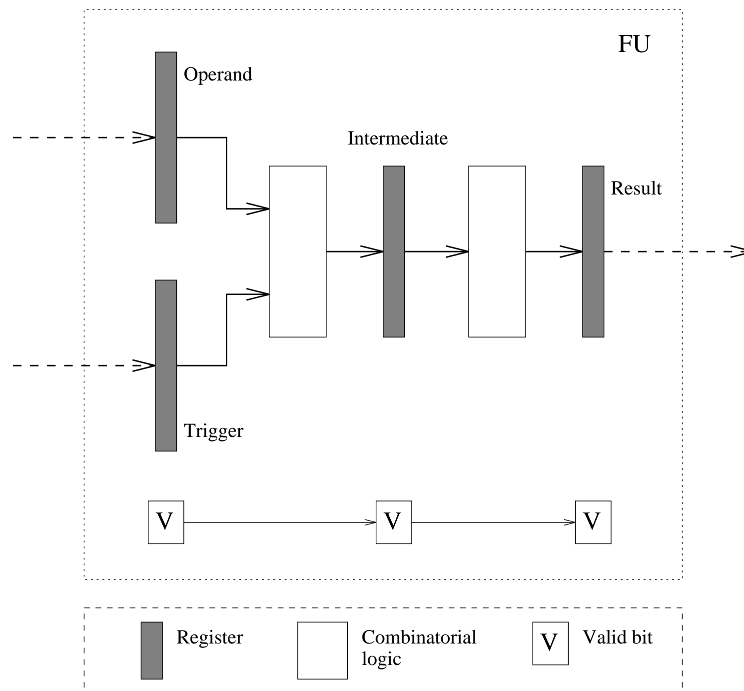


Figure 2.3. Hybrid functional unit pipeline

and therefore a delay of 3 cycles: the result can be read in the third cycle after the write to the trigger register (reading earlier will generate a lock). The control of the stages and the generation of lock signals is determined by valid bits, which signal whether a stage actually contains valid data.

Hybrid pipelines allow for maximum scheduling freedom; the only limitations are the fixed depth of the pipeline, and the fixed speed at which stages advance. Data may be entered and read at any time, provided that causality is taken into account (e.g. it is not possible to read a result before operands are entered).

For a detailed analysis of scheduling constraints and optimizations see [7].

2.3. Connections between FUs and transport network

The data transport network of a TTA is not necessarily fully connected; registers do not have to connect to all of the busses. The connections between FUs and the data transport network go through bus interfaces or *sockets*. There are input and output sockets. A FU can connect several inputs or outputs to one socket, but a socket can transfer only one data item each cycle. An output socket can place the same item on several busses at the same time. For an example of an incompletely connected bus network for sockets see figure 2.1.

The structure of an input socket is presented in figure 2.4. The *move* busses are subdivided into data, ID, and control busses. The IDs are decoded locally on each socket; a range may be decoded to be able to extract an opcode. When an ID addresses the socket, the corresponding data bus is selected. The control bus carries signals for locking, guarding and exceptions.

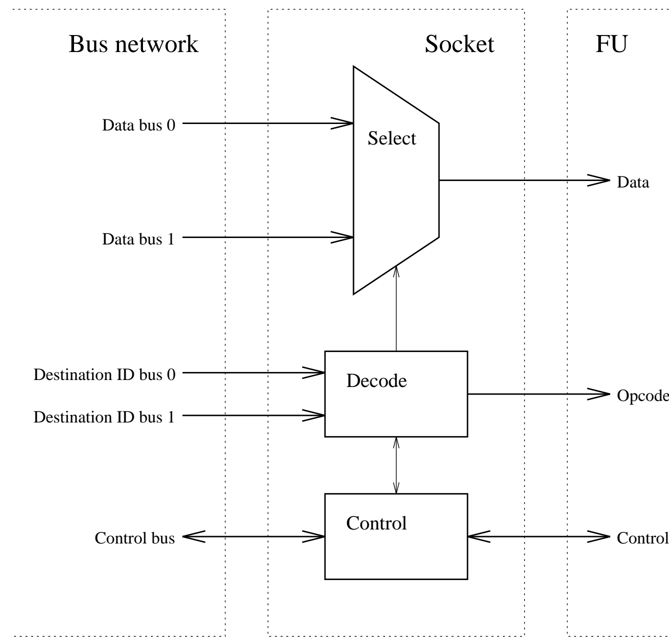


Figure 2.4. Input socket connected to two busses and a FU

2.4. Distinguishing MOVE architecture features

MOVE architectures offer several advantageous features when compared to other architectures. The differences are in efficiency and adaptability; major ones are presented in the following list:

- Functional flexibility. There is no constraint in the MOVE concept that limits the number of, connections to, or functionality of FUs.
- Hardware efficiency. A MOVE architecture can be configured freely; there is no assumption in the architecture that fixes the transport network structure or FU organization. Therefore hardware can be allocated for those functions where it is most needed.
- Simplicity. MOVE architectures result in a very simple and regular organization, which allows for scalability.
- Performance scalability. Performance can be scaled by increasing the number of FUs, by increasing pipelining of either FUs or the transport network, by increasing transport network connectivity, and by increasing transport network capacity. MOVE architectures do not imply coupling of the number of FUs to the transport network capacity or instruction format, as do VLIW architectures.
- Implementation flexibility. The MOVE concept does not impose constraints on implementation of either FUs or transport network. For example, pipelined and unpipelined implementations of FUs and transport networks may be combined.
- Trivial decoding. Only one instruction format (*move)* is used.
- Compiler optimizations. Programmed transport allows for new degrees of scheduling freedom.

Potential disadvantages of MOVE architectures are the sophisticated optimizations required from compilers to be able to use a MOVE processor to its full potential, and the increase in instruction size (when not all the *move* slots in instruction words can be filled).

Chapter 3. MOVEASAINT architecture

The MOVEASAINT processor generator realizes instances of the MOVEASAINT parametrized architecture. The architecture is based on the MOVE32INT architecture as described in [3] and [7]. Basically, the MOVEASAINT architecture consists of an integer MOVE architecture with parameters to allow for performance scaling and adaptation to application requirements.

The requirements for the architecture of the MOVE processor generator can be found in the first of the following sections. An overview of the MOVEASAINT architecture actually implemented by the MOVE processor generator is given in section 3.2. Although designed to be a superset of the MOVE32INT, the architecture contains deviations from the MOVE32INT architecture (improved functionality for exceptions, some bug fixes, etc.); these differences are summarized in section 3.3.

Details on functional units, pipelining, and control can be found in chapters 4 and 5.

3.1. Architecture requirements

The purpose of the MOVE processor generator is to provide for a realization path in the MOVE framework [4], and to determine the viability of the ASA silicon compiler for that purpose. Therefore, the major architecture requirements for the processor generator are:

1. MOVE architecture
2. Parameters for scaling
3. Compatibility with MOVE32INT

A specific goal is the evaluation of the ASA-based processor generator in comparison to the Sea of Gates MOVE32INT. Hence the requirement that a MOVE32INT architecture based processor can be generated; the same FU types must be supported and the system has to be capable of generating a MOVE32INT compatible bus network.

To support performance scaling and customization of the processor in the MOVE framework, the following architecture feature parameters are required:

- Bus widths
- Number of *move* busses
- Connectivity of the *move* busses
- Number of FUs
- Number of GP registers
- Guard for each *move* or for entire instruction

The basic list of parameters presented here implies a higher number of actual parameters (e.g.

bus widths $\Rightarrow$ *move* data bus width and address sizes). Additionally, the following parametrized features are desirable:

- Pipelining degree
- Immediate size

Of these, variable pipelining degree is hard to implement.

At least a functional processor core has to be implemented by the MOVE processor generator; strict compatibility with MOVE32INT for the load-store FU is not required.

In the current version of the MOVE processor generator all required parameters (and more) are implemented; a mostly MOVE32INT compatible processor can be generated. Remaining differences with MOVE32INT are: improved functionality in several places (e.g. exception support) and a less sophisticated load-store unit. See section 3.3 for a summary of the differences. The desired features of the second list are not currently implemented. Details on the currently implemented parameters can be found in the following sections.

3.2. Architecture overview

This section provides an overview of MOVEASAINT architecture. Section 3.2.1 summarizes the main features of the architecture without providing specific details. The sections thereafter provide specifics on parameters, assignment of addresses to registers, and instruction formats. Details on FUs, pipelining and control are deferred to later chapters.

3.2.1. Summary

The MOVEASAINT architecture is a MOVE architecture with a number of FUs and a transport network that consists of busses. The number of busses, the connectivity of the busses, and bus width are parametrized; also, how many FUs of a certain type are to be used is determined through parameters. Each data transport bus corresponds to a move operation that can be specified in an instruction; one instruction word contains as many move operations as there are busses.

The following FU types are supported:

Ifetch unit The instruction fetch unit contains the program counter, instruction cache, and a PC chain to support exception recovery. This unit reads instructions from external memory and sets up the *move* busses as specified by the instructions.

Guard unit Determines which moves are actually executed, depending on the state of the compare units and the guard fields in instructions.

Immediate unit Makes part of move operation source fields in the instruction word available for reading as immediate values.

Load-store units For accessing external (data) memory.

Compare units Compare operands and provide the results to the guard unit.

Integer units Implement integer addition and subtraction.

Logic units Provide for bitwise logic *and, or,* and *exor* operations.

Shift units Provide for arithmetic and logic word-size shift operations over 1 bit or 2 bits.

Valid unit Contains the result valid bits of FUs for exception processing support (the exception handler needs to know which FU stages are actually used).

Register units General purpose registers (no operation, trigger and result registers are the same).

Scan reg units General purpose registers with external serial load inputs, for testing.

Of these, always exactly one instruction fetch unit, one guard unit, and one valid unit are used. For the other units, how many of them should be used may be specified freely (zero or more instances are allowed). Not all configurations are useful, such as a processor without load–store unit. On the other hand, a processor with multiple load–store units may be specified which can be quite useful.

How the functional units fit into the system is depicted if figure 3.1. Note that separate external data and instruction busses are used (Harvard architecture). The FUs interface to the *move* busses by means of sockets; the sockets provide access to the trigger, operand, and result registers of the FUs (see previous chapter). Parameters determine to which busses each socket connects.

Both the FUs and the transport network in the MOVEASAINT architecture are pipelined. Most FUs use hybrid pipelining; exceptions are the register units, with obviously little opportunity for pipelining, and the instruction fetch unit. The instruction fetch unit differs from other units in that the triggering is implied: it will enter a new program counter value into the pipeline during each cycle that the processor is not locked. Accesses to the instruction fetch unit therefore do not trigger operations or remove results, only the *value* of a pipeline stage can be altered or read; the program counter registers are always 'valid'. Note that the PC chain for exception support is dealt with differently, for details see section 4.1.

The integer and load–store FUs can be sub-pipelined. Both units implement full hybrid pipelining, but both external memory and wide adders can be relatively slow. Therefore the number of cycles that the corresponding pipeline stages may stall can be set through parameters.

The data transport network has two pipeline stages: *decode* and *move* (figure 3.2). Decoding is part of the transport pipeline, because decoding is performed locally on the sockets of FUs. So, in the *decode* stage, source and destination IDs for the move operations are distributed by the instruction fetch unit to all units (including itself), where each socket determines if it is being addressed. In the *move* stage, sockets addressed in the decode stage drive and select the appropriate busses, and the actual data transfer occurs.

The pipelining of the transport network is not visible in the architecture when accessing most units. For all units except the instruction fetch unit, all visible action occurs during the move stage of the transport network. The instruction fetch unit, however, appears to be one pipeline stage deeper than it actually is, because the transport network inserts a decode stage, see figure 3.3. When doing a jump, the instruction path through the pipelines requires two extra cycles before the first instruction is executed from the new address, so there are two delay slots. (Also, when a cache miss occurs, the processor is locked during the extra cycles for external instruction fetch.)

The FU pipelines operate in parallel with the transport pipeline. There is no fixed number of FU pipeline stages, nor does an operation always happen immediately following a *move;* FU pipelines

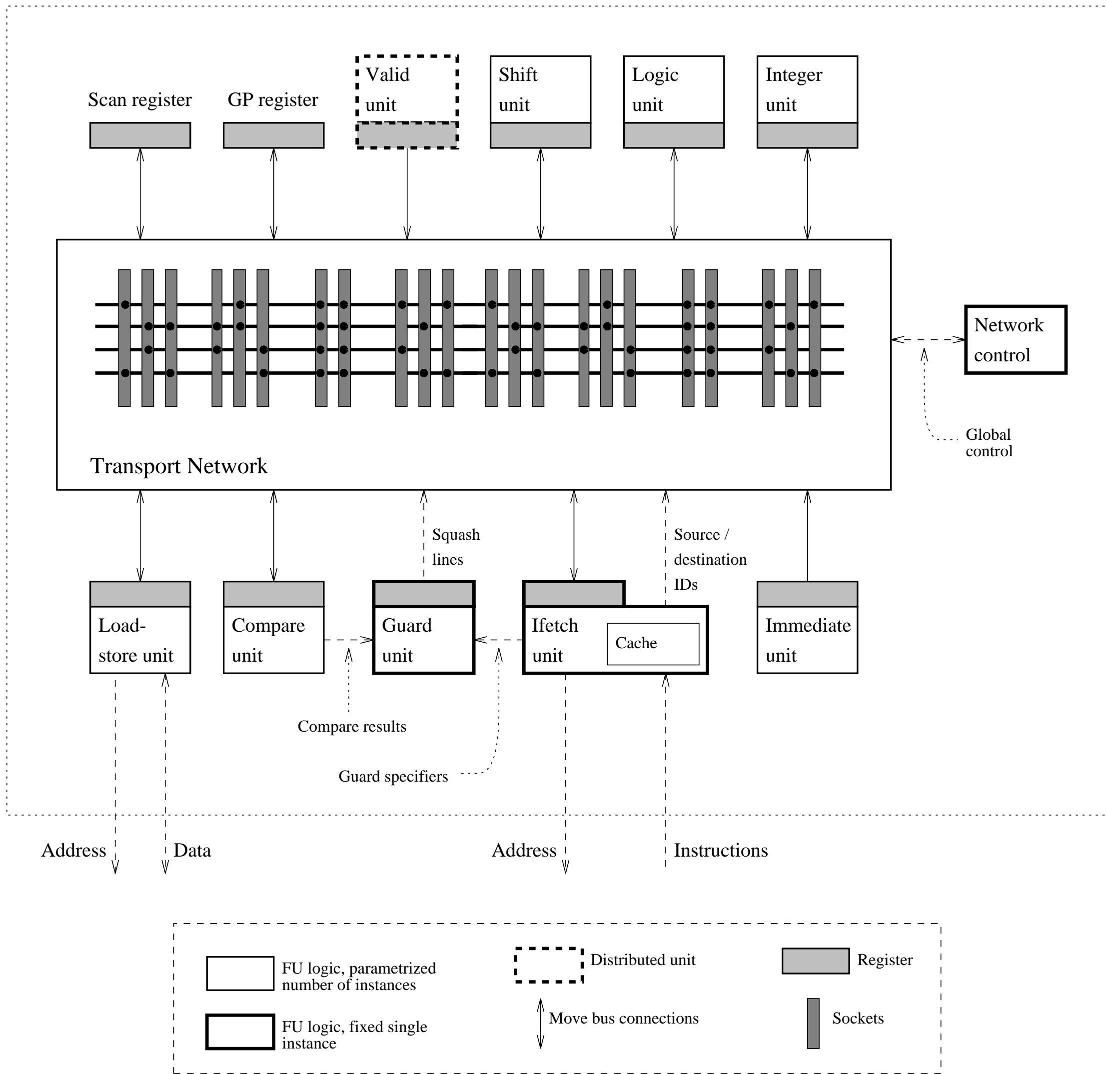


Figure 3.1. MOVEASAINT structure

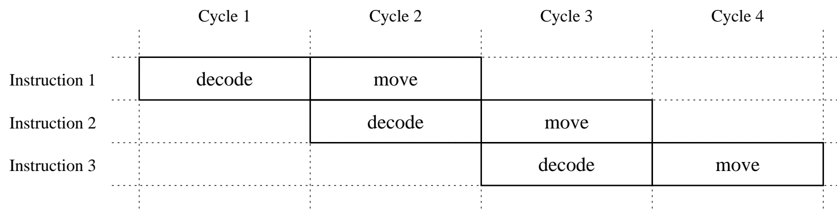


Figure 3.2. MOVEASAINT transport network pipeline

operate independently from the transport network pipelines (besides synchronization points). Therefore FU pipeline stages are not shown in the diagrams.

Whether data is actually transferred in the move stage of the transport network depends on guards and locks. In the MOVEASAINT architecture, each move operation in an instruction word may be guarded separately, or all move operations in an instruction may be guarded with one guard.

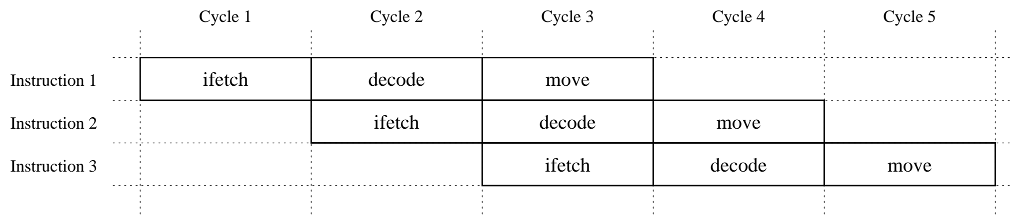


Figure 3.3. MOVEASAINT instruction/transport pipeline

Which option is implemented is determined by the setting of a parameter. The guard specifier fields in instructions are used to specify conditions for execution of the associated move operations. The condition is a function of the guard field values and the result values of the compare units. The size of the guard specifier fields and the number of compare units are determined through parameters, although only one guard evaluation function is implemented in the current version of MOVEASAINT. For this reason the guard field size must be set to 3 bits and the number of compare units must be two. See 4.2 for specification of the guard evaluation function.

If the guard condition evaluates true for a move operation, then the data is actually transferred as specified by the move. If the guard evaluates false, then the data transfer is not performed; other move operations may proceed during that cycle though, and transport and FU pipelines continue where possible.

The data transport network can be *locked* ('stalled'), to provide for synchronization with FUs. Locks can be global; for example, the instruction fetch unit can unconditionally lock the processor while performing an external instruction fetch on a cache miss. The lock stops the transport pipeline, and thus move operation execution, completely. However, internal *hybrid* FU pipeline stages may continue where possible!

Local locks originate from FUs to signal one of these conditions:

Write lock When writing a trigger or operand register of a FU that does sub-pipelining, a lock is raised when the register can not yet be written, but will be available for writing later. If the register can not become available during a lock, an exception is raised instead.

Read lock When attempting to read a result from a FU that contains data, but has not finished the operation yet, a lock is raised until the result is available.

These local locks are like global locks, except for guards: a move operation with a guard evaluating false cannot cause a lock. Global locks are independent of guards.

The MOVEASAINT architecture supports an exception mode. In exception mode exception requests are ignored and on entering exception mode program counter values are saved in the PC chain of the instruction fetch unit, to support return from exception. Two instructions in the pipeline are squashed ('guarded') when an exception occurs: there is no such thing as a delay slot on an exception. Exception mode is entered upon the following conditions, ordered according to priority level:

Reset When the external reset input is asserted, the next instruction to be executed comes from the reset vector (parameter). This is the only 'exception' signal that is not masked when in exception mode, and is independent from locking state. A reset clears all FU valid bits, so all FUs are available after reset.

Exception Internal exceptions are synchronous exceptions generated by FUs to signal pipeline errors. The instruction causing the exception will be completed, the next instruction will come from the exception vector (parameter). A pipeline error is considered fatal (e.g. emulation of deeper pipelines is not supported), and can often indeed not be recovered because of the completion of the offending *moves* (so the exceptions are imprecise).

Trap Traps are synchronous exceptions generated in the instruction stream by a *move* to a specific instruction fetch unit address. The result is a jump to the specified trap address.

Interrupt When the external Interrupt input is asserted, the next instruction to be executed comes from the interrupt vector (parameter). Unlike reset, an interrupt signal is ignored until lock conditions are finished and the processor is out of exception mode.

There are two types of internal pipeline exceptions (which generate the same exception vector though):

Pipeline full Pipeline full exceptions are generated when a FU implementing the exception is being written to (trigger or operand), has valid data in all pipeline stages, and the result register is not being read in the same cycle.

Pipeline empty Pipeline empty exceptions are generated when the result register of a FU implementing the exception is being read, and there is no valid data in any of its stages.

Currently, there is no full support for precise FU-generated exceptions. How this feature may be added is discussed in 9.2.

FU specific details on lock and exception support can be found in chapter 4. Details on lock specification and implementation can be found in chapter 5.

3.2.2. Register assignment

Address (ID) assignment for registers is done automatically by the MOVE processor generator. This section describes how addresses are assigned and how to determine the actual address assignment for a MOVEASAINT instance. Assignment results are presented for the MOVE32INT configuration.

Trigger registers may be accessed at several address locations to select different operations when triggering a FU. Such multiple–address mapped trigger registers are assigned a contiguous range of addresses. Which address offset within a range corresponds to which operation is determined by the FU implementation and is fixed; the offsets used for current FU implementations can be found in the chapter describing the functional units.

Operand and result registers can also be accessed at multiple locations. In the current MOVEASAINT implementation only the immediate unit has multiple results, and no multi-address operand registers are used.

The number of FUs used is variable, so socket addresses change with the configuration. The preprocessing part of the MOVE processor generator takes care of the address assignment for FU sockets. The assigned addresses can be found in the *parmeval.sih* file after preprocessing. This

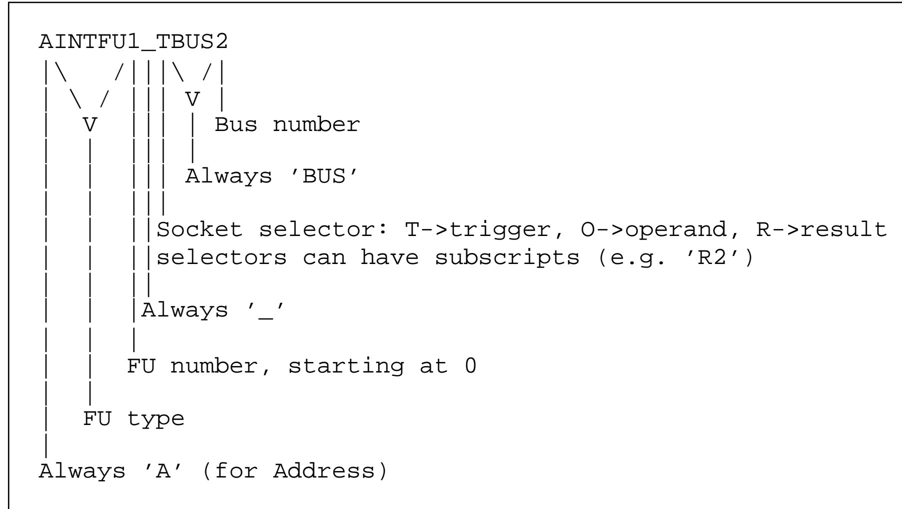


Figure 3.4. Socket address identifiers in parmeval.sih

file also contains other evaluated parameters; how the socket address identifiers can be recognized and interpreted is shown in figure 3.4. The value associated with the identifier in the file is the address.

To understand the address assignment algorithm, observe that there are two address spaces associated with each bus: a source and a destination address space. Trigger and operand sockets occupy parts of destination address spaces, result sockets occupy parts of source address spaces. So, with the *PARMOVES* parameter defining the number of move busses, the number of socket address spaces is $PARMOVES \cdot 2$. Each of these address spaces can be of different size, depending on the socket connections. Address assignment is done independently for each address space, so one register may have different addresses on different busses.

To simplify decoding, socket address ranges with size equal to a power of 2 are used; so, such n exists that:

$$R_{S_i} = 2^n,\ n \in \{0, 1, 2, \ldots\} \tag{3.1}$$

where R denotes address range, and S_i is any socket. The ranges are aligned such that a minimal number of address bits need to be compared by each socket, and the total range of addresses for each bus is kept as small as possible. To achieve that, the lowest addresses are assigned to sockets with the largest address ranges. Specifically, if the following constraint is met:

$$R_{S_x} > R_{S_y} \Rightarrow A_{S_x} < A_{S_y} \tag{3.2}$$

with A the base address of a range, and if addresses of sockets S_k connected to one bus are assigned contiguously as follows:

$$A_{S_0} = 0 \tag{3.3}$$

and:

$$A_{S_n} = \sum_{k=0}^{n-1} R_{S_k},\ n > 0 \tag{3.4}$$

then each range R will be aligned such that only the lowest $\log_2 R$ address bits vary within the range, because (3.1)–(3.4) imply n exists such that:

$$A_{S_i} = nR_{S_i},\ n \in \{0, 1, 2, \ldots\} \tag{3.5}$$

for any socket S_i.

Any ordering of socket base addresses A is correct, as long as the condition in (3.2) is met. Part of the ordering is therefore arbitrarily chosen; the actual ordering is specified in the socket ID section of the *parmderv.sih* file. An example assignment is shown in tables 3.1 (source addresses) and 3.2 (destination addresses), which also show the ordering actually used. Shown in the example is the socket address assignment for the MOVE32INT compatible configuration. The identifiers used in these tables are as described in figure 3.4; the left and top parts of the tables contain left and right parts of the identifiers. The following small excerpt from the file *parmeval.sih* shows how the identifiers are applied there (cpp syntax):

```
#define AIMMFU0_RBUS2 0
#define AIFEFU0_TBUS2 0
#define ACMPFU0_TBUS2 8
#define ACMPFU1_TBUS2 8
#define ALOGFU0_TBUS2 12
#define ASHIFU0_TBUS2 12
#define AINTFU0_TBUS2 12
```

Note the redundant definest: only addresses for which connections have actually been defined in the parameter declaration file are valid. To determine the actual address assignment for a processor configuration, it is best to refer to this file. Details of the sockets the identifiers refer to can be found in chapter 4 on FUs.

Another assignment that is done automatically is the assignment of bit positions for valid bits in the valid unit. The valid unit makes the result valid bits readable of all FUs that have one. There are no constraints on the order of the valid bits. A valid bit occupying the most significant bit in a word can be easily tested though, with a compare unit. This facilitates exception processing.

The assignment performed by preprocessing can be found by inspecting the file *parmeval.sih,* just as for the socket address assignment. The interpretation of identifiers for valid bit assignment can be found in figure 3.5. Bit positions and register numbers are associated with these identifiers. Note that register numbers are ‘socket local’, and not physical bus numbers: for details, see the file *parmderv.sih.* The valid bit assignment for the MOVE32INT configuration can be found in table 3.3. All valid bits in the MOVE32INT configuration are in one register accessible on bus 3 only (see table 3.1).

3.2.3. Instruction format

The instruction format of a MOVEASAINT architecture depends on the parameter settings. Parameters which influence the format directly are:

PARMOVES Sets the number of move operations in an instruction.

SINGLEGUARD Determines if each move is guarded individually or one guard is used for the entire instruction.

GUARDSPECSIZE Sets the size of the guard specifier fields.

FU	identifier (split)	...BUS0	...BUS1	...BUS2	...BUS3
immediate unit	AIMMFU0_R...	0	0	0	0
ifetch unit	AIFEFU0_R1...	64	64	–	–
	AIFEFU0_R2...	65	65	–	–
compare unit 0	ACMPFU0_R...	66	–	–	64
compare unit 1	ACMPFU1_R...	–	66	64	–
logic unit	ALOGFU0_R...	–	–	65	65
load–store unit	ALDSFU0_R...	67	–	–	66
integer unit 0	AINTFU0_R...	68	–	66	–
integer unit 1	AINTFU1_R...	–	67	–	67
shift unit	ASHIFU0_R1...	69	–	67	–
	ASHIFU0_R2...	–	68	–	68
register units	AREGFU0_R...	70	69	–	–
	AREGFU1_R...	71	70	–	–
	AREGFU2_R...	–	71	–	69
	AREGFU3_R...	–	72	–	70
	AREGFU4_R...	72	–	68	–
	AREGFU5_R...	73	–	69	–
	AREGFU6_R...	–	73	70	–
	AREGFU7_R...	–	74	71	–
	AREGFU8_R...	74	–	–	71
	AREGFU9_R...	75	–	–	72
scan reg unit	ASCRFU0_R...	–	–	72	73
valid unit	AVALFU_R...	–	–	–	74

Table 3.1. MOVE32INT *source* address assignments (shown with actual preprocessor identifiers)

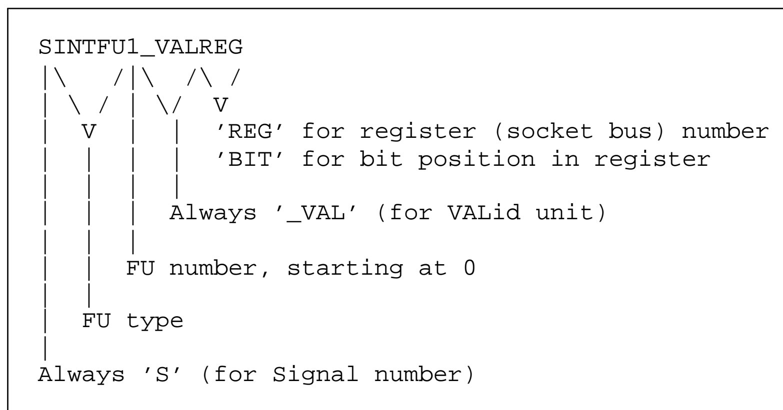


Figure 3.5. Valid bit assignment identifiers in parmeval.sih

The format of instructions without SINGLEGUARD defined is shown in figure 3.6, with SINGLEGUARD defined in figure 3.7.

The source and destination field sizes depend on the address space demands of the sockets

FU	identifier (split)	…BUS0	…BUS1	…BUS2	…BUS3
ifetch unit	AIFEFU0_T…	0	–	0	–
compare unit 0	ACMPFU0_T…	8	0	–	–
compare unit 1	ACMPFU1_T…	–	–	8	0
logic unit	ALOGFU0_T…	–	4	–	4
shift unit	ASHIFU0_T…	–	8	–	8
integer unit 0	AINTFU0_T…	12	12	–	–
integer unit 1	AINTFU1_T…	–	–	12	12
load–store unit	ALDSFU0_T…	14	–	–	14
compare unit 0	ACMPFU0_O…	–	14	14	–
compare unit 1	ACMPFU1_O…	16	–	–	16
logic unit	ALOGFU0_O…	17	–	15	–
load–store unit	ALDSFU0_O…	–	15	16	–
integer unit 0	AINTFU0_O…	18	–	–	17
integer unit 1	AINTFU1_O…	–	16	17	–
register units	AREGFU0_T…	–	–	18	18
	AREGFU1_T…	–	–	19	19
	AREGFU2_T…	19	–	20	–
	AREGFU3_T…	20	–	21	–
	AREGFU4_T…	–	17	–	20
	AREGFU5_T…	–	18	–	21
	AREGFU6_T…	21	–	–	22
	AREGFU7_T…	22	–	–	23
	AREGFU8_T…	–	19	22	–
	AREGFU9_T…	–	20	23	–
scan reg unit	ASCRFU0_T…	23	21	–	–

Table 3.2. MOVE32INT *destination* address assignments (shown with actual preprocessor identifiers)

FU	valid identifier	bit position
shift unit	SSHIFU0_VALBIT	31
logic unit	SLOGFU0_VALBIT	30
load–store unit	SLDSFU0_VALBIT	29
compare unit 0	SCMPFU0_VALBIT	28
compare unit 1	SCMPFU1_VALBIT	27
integer unit 0	SINTFU0_VALBIT	26
integer unit 1	SINTFU1_VALBIT	25

Table 3.3. MOVE32INT valid bit position assignments (shown with actual preprocessor identifiers)

connected to the respective busses, see section 3.2.2. The field sizes are computed by the processor generator as follows:

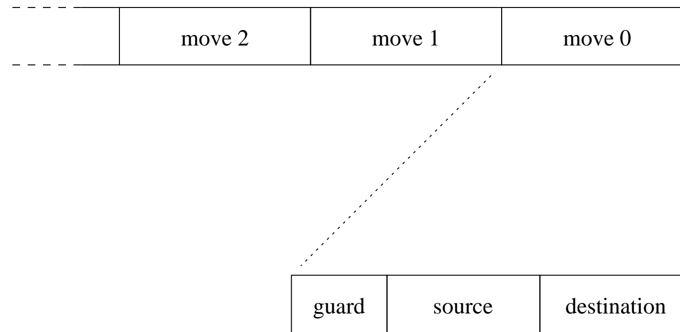


Figure 3.6. MOVEASAINT instruction formats with SINGLEGUARD *undefined.*

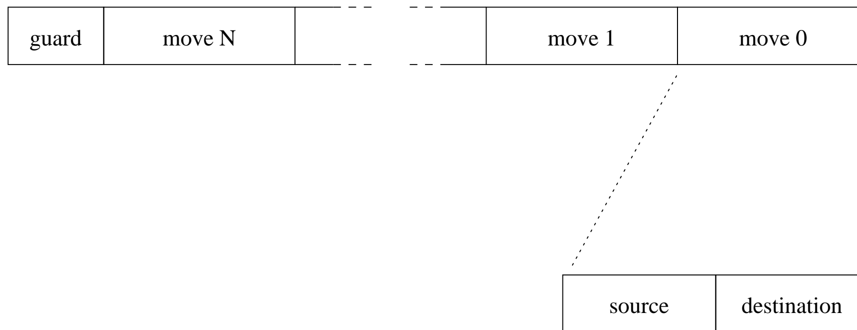


Figure 3.7. MOVEASAINT instruction formats with SINGLEGUARD *defined.*

$$W_f = \lceil \log_2 N_{A_f} \rceil$$

where W_f is the width of a source or destination field and N_{A_f} is the number of assigned address locations for that field.

3.3. Changes versus MOVE32INT

One goal of the ASA-based processor generator is comparison of the results with the Sea of Gates MOVE32INT. Therefore the generator is designed to allow the MOVE32INT architecture as a possible instance. Strict compatibility was not required; the aim was a fair comparison of the performance of the design and evaluation of the design methods. Changes versus the MOVE32INT have been incorporated into the MOVEASAINT: some to improve the architecture, some influenced by differences in technology or design method, some to be able to finish the design in limited time.

The MOVASAINT architecture features the following improvements on the MOVE32INT:

- **Internal exceptions.** The MOVEASAINT supports FU-generated exceptions. The current version raises exceptions on pipeline errors, as opposed by the MOVE32INT which locks in those situations. This change implies changes to the conditions for raising in a lock.
- **Compare unit exception support.** The compare units in the MOVEASAINT allow read access to the operand registers. This way, full state can be saved on exception.

- **Valid unit.** In the MOVEASAINT architecture all FU result valid bits are accessible in one unit. By gathering the bits together in words, register address space is saved.

- **Guarded narrowcasts.** Multiple guarded writes to a single destination is allowed for all destinations. The current realization does not even require the guard conditions of multiple writes to one location to be mutually exclusive: a write collision does a bitwise logical *or* on the data values.

- **Independent decoding.** Register address space is saved by making decoding in sockets independent for each socket, and within a socket independent for each bus. This results in changes in register mapping and reduces destination field size in instructions for the MOVE32INT configuration.

- **Instruction address alignment.** The MOVEASAINT provides options for address (pointer) register alignment in the instruction fetch unit and the load–store unit.

Feature changes driven by technology differences:

- **Instruction cache control.** The cache cannot be invalidated, but it can be disabled and loaded. There is no provision in ASA for implementing resettable cache line valid bits with reasonable area efficiency.

- **Single-register FUs.** General purpose register FUs contain one register each, as opposed to two for the MOVE32INT. The bus load and area disadvantages in SoG technology, when not combining 2 register outputs, are not present in ASA's standard-cell layout (see 7.1). The advantage of one-register FUs is that the access limitations posed by sockets (only one data item can pass through a socket in one cycle) is removed.

The following change versus the MOVE32INT is because of limited design time:

- **Simpler load–store unit.** The load–store unit is less aggressively pipelined than the MOVE32INT version, and has a simpler socket configuration (made possible by the limited pipelining). Store–through for exception handling is supported, though.

More details on how technology influenced the architecturally visible changes can be found in chapter 7.

Chapter 4. Functional units

This chapter provides architectural details on the MOVEASAINT functional units. In the MOVEASAINT architecture FU pipelining is made visible, so the FU descriptions include pipeline organization. Each of the following sections contains a description of one FU type. The sections are organized in a number of standard topics, outlined below.

General

Short description of the FU type.

FU type **parameters**

A list of the, for the unit type that is being described, most important parameters. Details on the parameters can be found in appendix A.

FU type **structure**

Overview of the FU pipeline organization.

FU type **operations**

A list of on the FU supported operations. The **Operation** part shows move operations like $g(r) : A_{src} \rightarrow A_{dst}$, where $g(r)$ is a guard specifier (which depends on r), and $A_{src} \rightarrow A_{dst}$ is a *move* with the source address that belongs to *src* and with the destination address that belongs to *dst*. Addresses of sockets are specified with the socket names used in the pipeline diagrams (e.g. A_T). Operations indicated with *–implicit–* are not coupled directly to move operations, but are performed at times dictated for example by (hybrid) pipeline control.

The **Semantics** part describes the effect of operations. The notation used is as in $v : r \leftarrow f_{int}(op, t, o)$, where $\leftarrow$ indicates assignment, and most of the identifiers used are names of registers, control state, and sockets which can be found in the pipeline diagram. In the example, an (integer unit) function of the opcode, the trigger register, and the operand register is assigned to the result register. The assignment is ‘guarded’ by the valid bits v. Opcodes are described symbolically in square brackets, e.g. $[t + o]$ for the ‘add trigger and operand register’ opcode. Often used function subscripts include unit-specific ones (such as *int* for the integer unit) and *hybrid* to indicate the hybrid pipeline control function. Hybrid pipelining details can be found in 5.

FU type **control support**

An overview of supported locks and exceptions.

Notes

Information that does not fit under one of the previously mentioned topics.

4.1. Instruction fetch unit

General

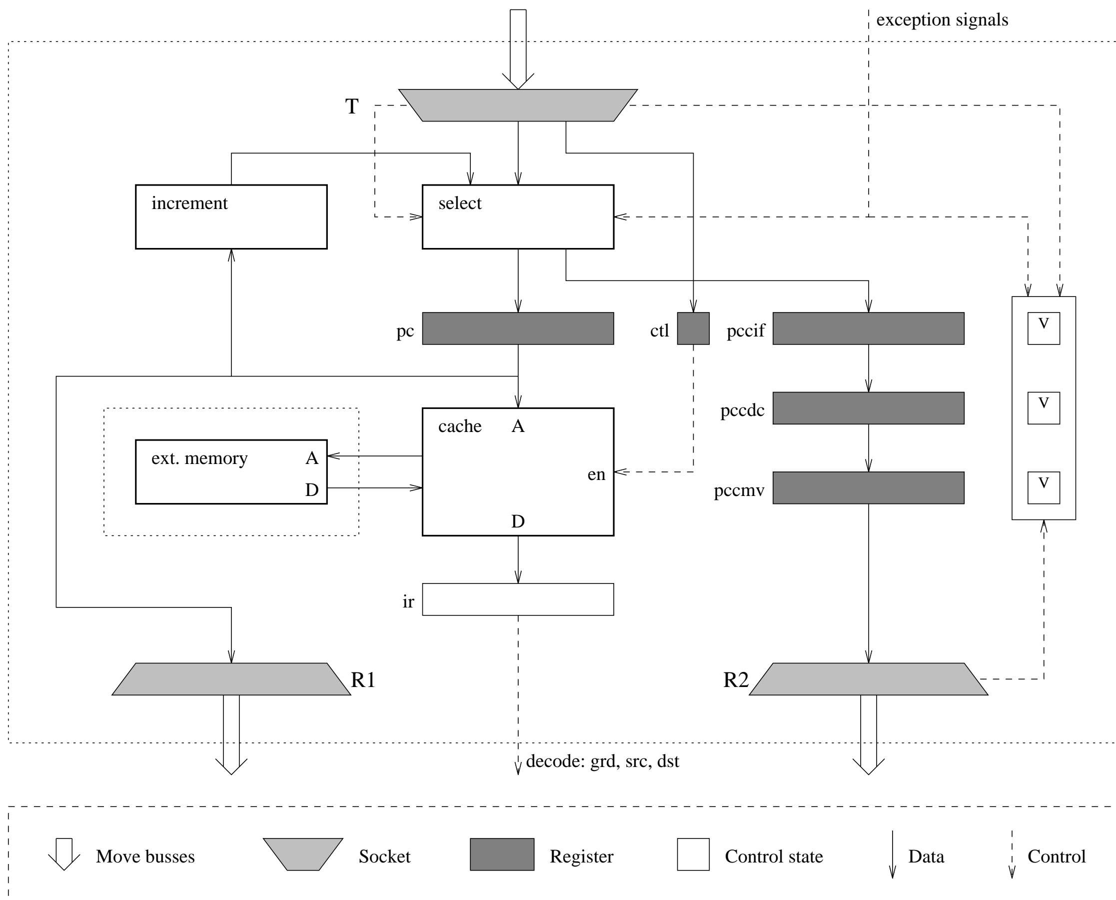
The instruction fetch unit reads instructions from external memory and sets up the *move* busses as specified by the instructions. There is always exactly one instruction fetch unit.

The instruction fetch unit contains a program counter, instruction cache, and a PC chain to support exception recovery. This is the only unit with implied triggering; pipelining therefore differs from the hybrid type used in other units.

Instruction fetch unit parameters

BUSWIDTH	Data word size
PARMOVES	Number of move busses
IFESHIFTADDR	Address register alignment
IFEADDRSIZE	Instruction address size
IFEAUTOADDRSIZE	For automatic IFEADDRSIZE determination
IFEOFFSETSIZE	Number of offset bits
IFEFUADDTYPE	Incrementer ASA adder type
IFEFUDELAY	Delay cycles for external memory
IFERSTVEC	Reset vector
IFEEXCVEC	Exception vector
IFEINTVEC	Interrupt vector
IFECACHESIZE	Number of cache lines

Instruction fetch unit structure



Instruction fetch unit operations

Operation	Semantics*
–implicit–	$v \leftarrow f_{control}(v, exc, T, R2)$
	$v : pccmv \leftarrow pccdc,\ v : pccdc \leftarrow pccif$
	$v : pccif \leftarrow f_{select,pcc}(pc, T, exc)$
	$v : pc \leftarrow f_{select,pc}(pc, T, exc)$
	$ir : f_{cache}(pc, ctl)$
$A_{src} \rightarrow A_T$	$pc \leftarrow src,\ pccif \leftarrow src$
$A_{src} \rightarrow A_T + 1$	$pc_{off} \leftarrow src_{off},\ pccif_{off} \leftarrow src_{off}$
$A_{src} \rightarrow A_T + 2$	$pc \leftarrow src,\ pccif \leftarrow src,\ v \leftarrow exc.mode$
$A_{src} \rightarrow A_T + 3$	$pc_{off} \leftarrow src_{off},\ pccif_{off} \leftarrow src_{off},\ v \leftarrow exc.mode$
$A_{src} \rightarrow A_T + 4$	$ctl \leftarrow src_0$
$A_{src} \rightarrow A_T + 5$	$ctl \leftarrow src_0$
$A_{src} \rightarrow A_T + 6$	$— \leftarrow src$
$A_{src} \rightarrow A_T + 7$	$— \leftarrow src$
$A_{R1} \rightarrow A_{dst}$	$dst \leftarrow pc$
$A_{R2} \rightarrow A_{dst}$	$dst \leftarrow pccmv,\ v \leftarrow f_{control,read}(v)$

* pc_{off} *is short for pc bits* $0 \ldots$ *IFEOFFSETSIZE*–1

A_T operation interpretation:

$A_{src} \rightarrow A_T$	jump
$A_{src} \rightarrow A_T + 1$	page relative jump
$A_{src} \rightarrow A_T + 2$	trap
$A_{src} \rightarrow A_T + 3$	page relative trap
$A_{src} \rightarrow A_T + \{4, 5\}$	write cache control register
$A_{src} \rightarrow A_T + \{6, 7\}$	not implemented

Instruction fetch unit control support

The instruction fetch unit sockets do not support locks or exceptions:

Socket	Lock support	Exception support
T	—	—
R1	—	—
R2	—	—

A global lock is raised when a cache miss occurs.

The instruction fetch unit responds to the following exception types, in order of priority:

Reset External reset signal

Exception Internal pipeline exceptions

Trap Software interrupt ($A_{src} \rightarrow A_T + \{2, 3\}$)

Interrupt External interrupt signal

A reset is always executed immediately; the other exception types are ignored until lock conditions are finished and the processor is out of exception mode. Exceptions are dealt with in the following way:

- The instruction cycle during which the exception is signalled is completed.
- The instruction fetch unit enters exception mode (*v* bits state).
- The remaining two instructions in the pipeline are discarded.
- The PC chain (*pcc* registers) freezes when the last user mode instruction is completed.
- The first instruction to be executed in exception mode is fetched from the appropriate exception vector (reset, exception, interrupt), or from the trap address.

Note that on reset, exception, and interrupt, a write to the *pc* and *pccif* registers reaches *pccif* only, while a vector is loaded into *pc*. A trap write reaches both registers.

Reading *pccmv* through R2 once enables the *pccif* register to be updated synchronously with the *pc* again; reading *pccmv* again enables the next register; reading *pccmv* a third time has the PC chain completely enabled again and disables exception mode.

Notes

- The instruction fetch unit does not implement hybrid pipelining. Triggering is implied: a new program counter value is entered automatically into the pipeline during each cycle that the processor is not locked. There is no lock or exception support on the sockets, the program counter registers are always ‘valid’.
- The instruction cache is a one-cycle, direct-mapped cache without subblocking.
- The instruction cache is always operating: instructions fetched from external memory are always stored in the cache.
- The *ctl* register contains the cache enable bit on bit position 0. Setting this bit enables the cache; resetting it disables the cache. On reset, the cache is disabled. If the cache is enabled, instructions are fetched from the cache when a hit is detected. If the cache is disabled, an external fetch and cache update is always performed, independently from cache contents.
- Changes of the *ctl* register take effect starting with the third instruction after the *move* to ctl (so there are two delay slots).

- There is no cache flush operation. The cache can be 'wiped', for example, by executing an instruction sequence of size IFECACHESIZE in contiguous memory locations. The wipe can be done with cache enabled or disabled.
- The operations for $A_T + \{6, 7\}$ are not defined; they are used only to get a trigger socket address range equal to a power of two (see 3.2.2).

4.2. Guard unit

General

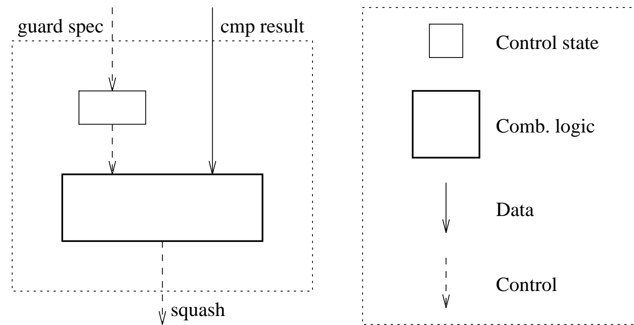
The guard unit determines which *moves* are actually executed, depending on the state of the compare units and the guard specifier fields in instructions. There is always exactly one guard unit.

This unit is not accessible with with the source or destination part of move operations, but with the guard specifier part. The guard unit can raise squash signals to disable *moves*.

Guard unit parameters

SINGLEGUARD	One guard in instruction, or one for each *move*
GUARDSPECSIZE	Number of guard specifier bits

Guard unit structure



Guard unit operations

Operation	Semantics*
$0 : A_{src} \rightarrow A_{dst}$	*if* r_1 *then* $dst \leftarrow src$
$1 : A_{src} \rightarrow A_{dst}$	*if* $\overline{r_1}$ *then* $dst \leftarrow src$
$2 : A_{src} \rightarrow A_{dst}$	*if* r_2 *then* $dst \leftarrow src$
$3 : A_{src} \rightarrow A_{dst}$	*if* $\overline{r_2}$ *then* $dst \leftarrow src$
$4 : A_{src} \rightarrow A_{dst}$	*if* $r_1 \wedge r_2$ *then* $dst \leftarrow src$
$5 : A_{src} \rightarrow A_{dst}$	*if* $\overline{r_1} \wedge r_2$ *then* $dst \leftarrow src$
$6 : A_{src} \rightarrow A_{dst}$	*if* $r_1 \wedge \overline{r_2}$ *then* $dst \leftarrow src$
$7 : A_{src} \rightarrow A_{dst}$	$dst \leftarrow src$

* r_1 *and* r_2 *denote results from the first and second compare unit*

Guard unit control support

— See compare unit —

Notes

- The current guard unit implementation requires two compare units, and a GUARDSPECSIZE equal to 3.

4.3. Immediate unit

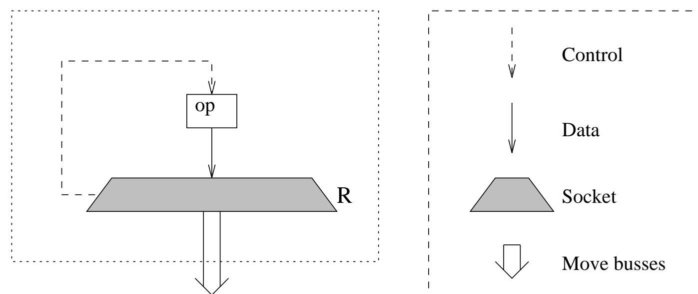
General

Immediate units make part of move operation source fields (in the instruction word) available for reading as immediate values.

Immediate unit parameters

BUSWIDTH	Word size
IMMFUS	Number of instantiations
IMMEDIATES	Number of immediate values

Immediate unit structure



Immediate unit operations

Operation	Semantics
$A_R + n \rightarrow A_{dst}$	$dst \leftarrow n$

Immediate unit control support

Socket	Lock support	Exception support
R	—	—

Notes

- The immediate unit provides for immediate values in the range $0 \ldots IMMEDIATES - 1$. Note that the output socket is not an ordinary one: on different busses, different immediates can be presented.
- The immediate unit can always be read, like a general purpose register. There is no pipeline control.

4.4. Load-store units

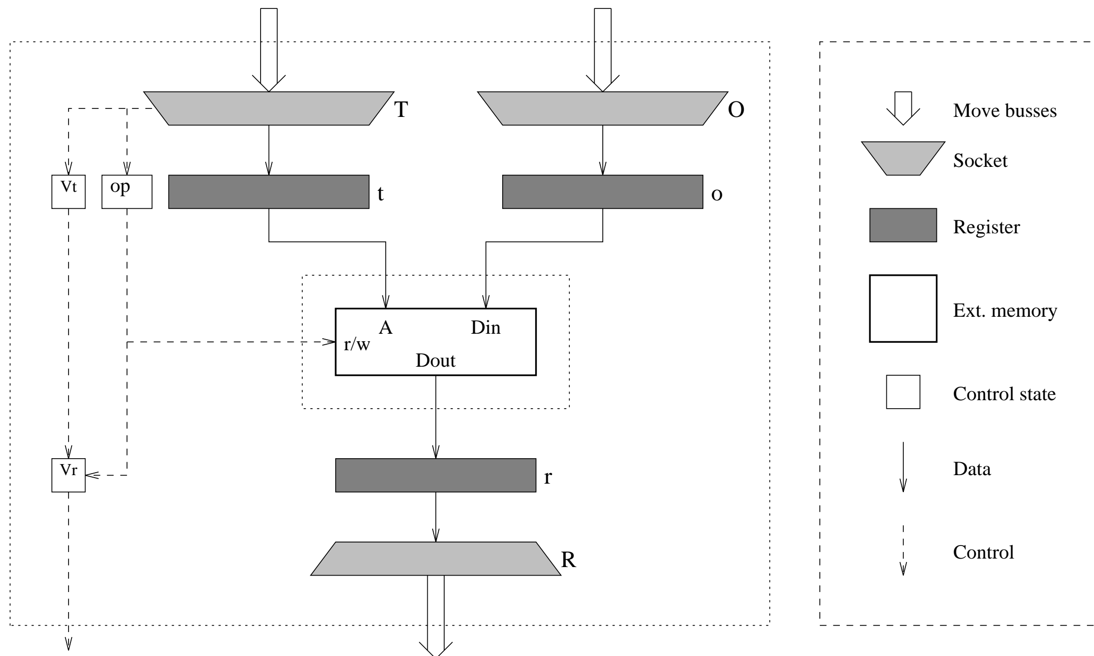
General

Load-store units provide access to external memory.

Load-store unit parameters

BUSWIDTH	Data word size
LDSFUS	Number of instantiations
LDSFUDELAY	Delay cycles for memory
LDSADDRSIZE	Address size
LDSOLOCKFULL	Choose O lock conditions

Load-store unit structure



Load-store unit operations

Operation	**Semantics**
–implicit–	$v \leftarrow f_{hybrid^*}(v, T, R)$
	$\{v, op = load\} : r \leftarrow f_{mem}(t),\ \{v, op = store\} : mem_t \leftarrow o$
$A_{src} \rightarrow A_T$	$t \leftarrow src,\ op \leftarrow [load]$
$A_{src} \rightarrow A_T + 1$	$t \leftarrow src,\ op \leftarrow [store]$
$A_{src} \rightarrow A_O$	$o \leftarrow src$
$A_R \rightarrow A_{dst}$	$dst \leftarrow r$

* *hybrid pipeline operation is guaranteed only if there is at most one outstanding load*

Load-store unit control support

Socket	**Lock support**	**Exception support**
T	write	—
O	write	—
R	read	pipeline empty

Notes

- Only one outstanding load at a time can be guaranteed to complete. If a new load is triggered *before* the result of the previous load is read, results are undefined. Note that there is no exception support to signal violation of this constraint.
- If LDSOLOCKFULL is defined, writes to the O (store data) socket will raise a lock only if a store is in progress. Without LDSOLOCKFULL defined, a lock will also (unnecessarily) be raised if a read is in progress. Writes to the T (address) socket will always cause locks if a memory operation is in progress.
- A store is always completed correctly, unless interrupted by a reset. Store completion is independent from an outstanding load.
- Triggering a store does *not* give a result, only triggering a load does.
- Multiple load–store unit instantiations will have multiple external data memory interfaces. These interfaces operate independently.
- Loads and stores are always full word size. There is no support for accessing subwords (e.g. bytes).
- Address (trigger) register size depends on LDSADDRSIZE.
- On exception, the store data (o) register can be saved by simply triggering a store to the desired address. After that, the unit is free for performing stores, independently of a possible unfinished load.
- The result valid bit v_r is available for reading in the valid unit.

4.5. Compare units

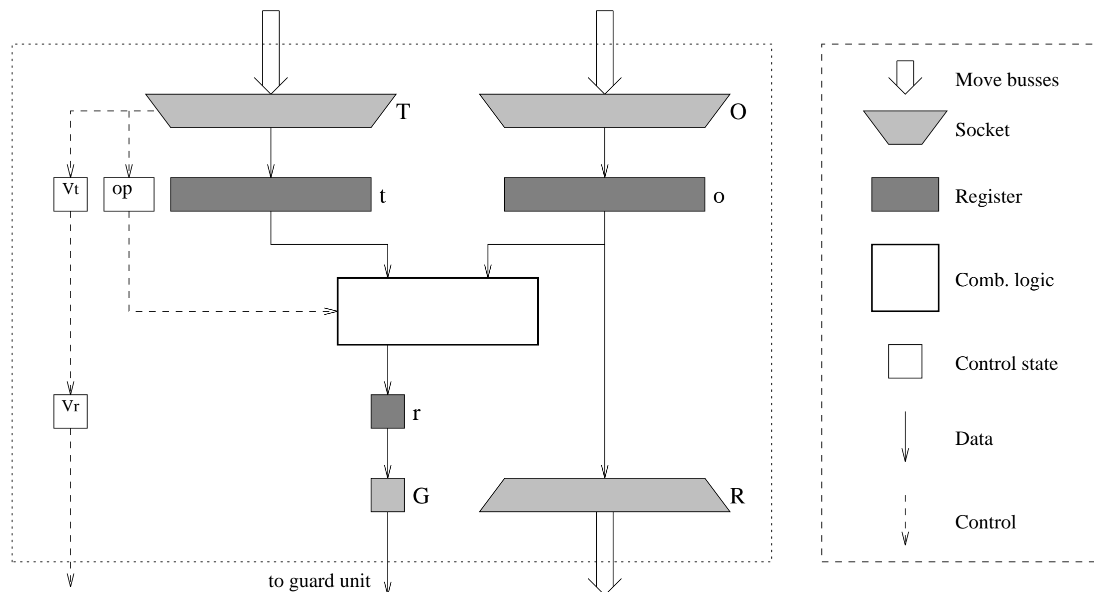
General

Compare units perform comparisons and provide a boolean result to the guard unit. The guard conditions for move operations depend on the compare unit results.

Compare unit parameters

BUSWIDTH	Operand word size
CMPFUS	Number of instantiations

Compare unit structure



Compare unit operations

Operation	Semantics
–implicit–	$v \leftarrow f_{hybrid}(v, T, G),\ v : r \leftarrow f_{cmp}(op, t, o)$
$A_{src} \rightarrow A_T$	$t \leftarrow src,\ op \leftarrow [t = 0]$
$A_{src} \rightarrow A_T + 1$	$t \leftarrow src,\ op \leftarrow [t = o]$
$A_{src} \rightarrow A_T + 2$	$t \leftarrow src,\ op \leftarrow [t < 0]$
$A_{src} \rightarrow A_T + 3$	$t \leftarrow src,\ op \leftarrow [t > 0]$
$A_{src} \rightarrow A_O$	$o \leftarrow src$
$A_R \rightarrow A_{dst}$	$dst \leftarrow o$
$g(r) : A_{src} \rightarrow A_{dst}$	$\text{if } g(r) \text{ then } \{dst \leftarrow src\},\ g \leftarrow r$

Compare unit control support

Socket	Lock support	Exception support
T	—	pipeline full
O	—	pipeline full
G	read	—
R	—	—

Notes

- The operation $g(r) : A_{src} \rightarrow A_{dst}$ is a move with a guard condition that is a function of r, and performs the equivalent of reading r. Whether a guard condition depends on the result of a compare unit is determined by the guard specifier for the move operation and the guard unit evaluation function.
- The read lock on the G socket is global: it is independent from the bus-local guard results (squashes). This means that in a move operation, the data move is guarded, but the guard specification is *not* guarded.
- The result valid bit v_r is available for reading in the valid unit.

4.6. Integer units

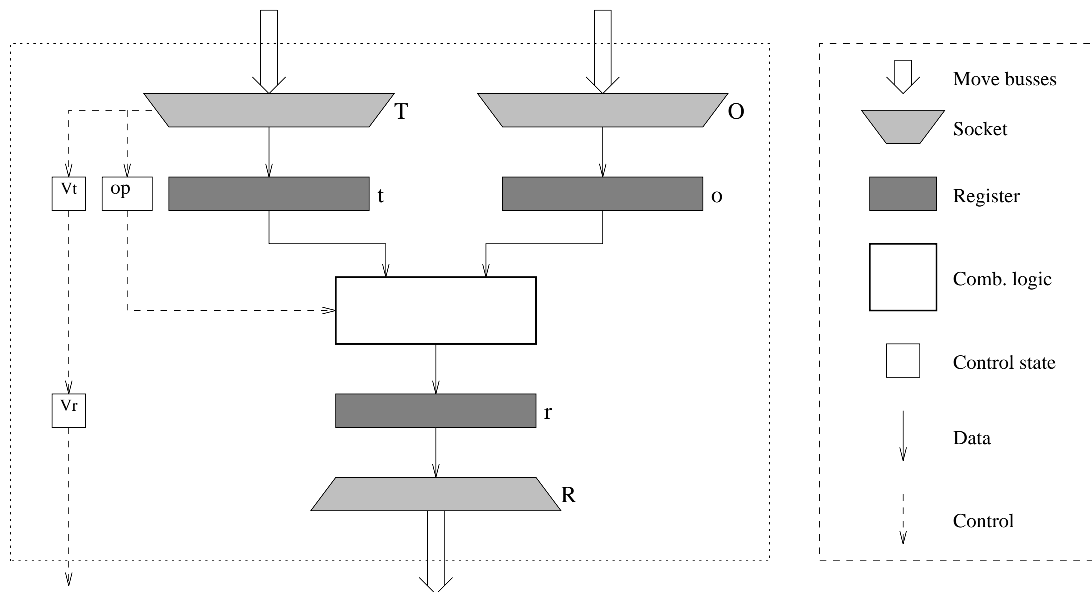
General

Integer units perform two's complement add and subtract operations.

Integer unit parameters

BUSWIDTH	Word size
INTFUS	Number of instantiations
INTFUDELAY	Delay cycles for adder logic
INTFUADDTYPE	ASA adder type

Integer unit structure



Integer unit operations

Operation	Semantics
–implicit–	$v \leftarrow f_{hybrid}(v, T, R),\ v : r \leftarrow f_{int}(op, t, o)$
$A_{src} \rightarrow A_T$	$t \leftarrow src,\ op \leftarrow [t + o]$
$A_{src} \rightarrow A_T + 1$	$t \leftarrow src,\ op \leftarrow [t - o]$
$A_{src} \rightarrow A_O$	$o \leftarrow src$
$A_R \rightarrow A_{dst}$	$dst \leftarrow r$

Integer unit control support

Socket	Lock support	Exception support
T	write	pipeline full
O	write	pipeline full
R	read	pipeline empty

Notes

- Write locks for T and O sockets will be enabled only if INTFUDELAY > 1.
- The result valid bit v_r is available for reading in the valid unit.

4.7. Logic units

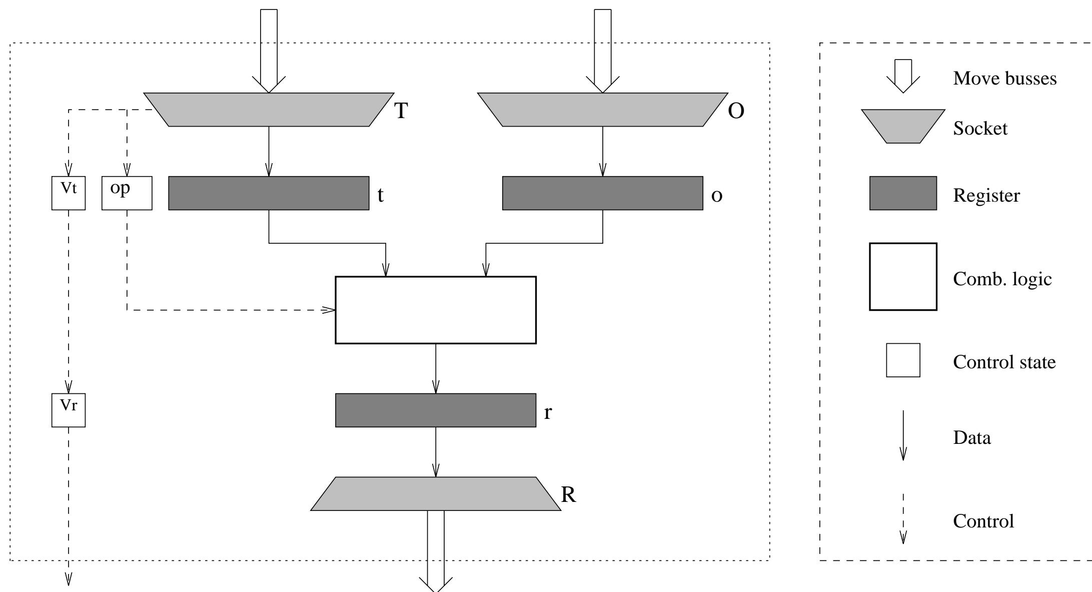
General

Logic units perform bitwise logic operations.

Logic unit parameters

BUSWIDTH	Word size
LOGFUS	Number of instantiations

Logic unit structure



Logic unit operations

Operation	Semantics
–implicit–	$v \leftarrow f_{hybrid}(v, T, R),\ v : r \leftarrow f_{log}(op, t, o)$
$A_{src} \rightarrow A_T$	$t \leftarrow src,\ op \leftarrow [t \wedge o]$
$A_{src} \rightarrow A_T + 1$	$t \leftarrow src,\ op \leftarrow [t \vee o]$
$A_{src} \rightarrow A_T + 2$	$t \leftarrow src,\ op \leftarrow [t \oplus o]$
$A_{src} \rightarrow A_T + 3$	$t \leftarrow src,\ op \leftarrow [undefined]$
$A_{src} \rightarrow A_O$	$o \leftarrow src$
$A_R \rightarrow A_{dst}$	$dst \leftarrow r$

Logic unit control support

Socket	Lock support	Exception support
T	—	pipeline full
O	—	pipeline full
R	read	pipeline empty

Notes

- The operation for $A_T + 3$ is not defined; it is used only to get a trigger socket address range equal to a power of two (see 3.2.2).
- The result valid bit v_r is available for reading in the valid unit.

4.8. Shift units

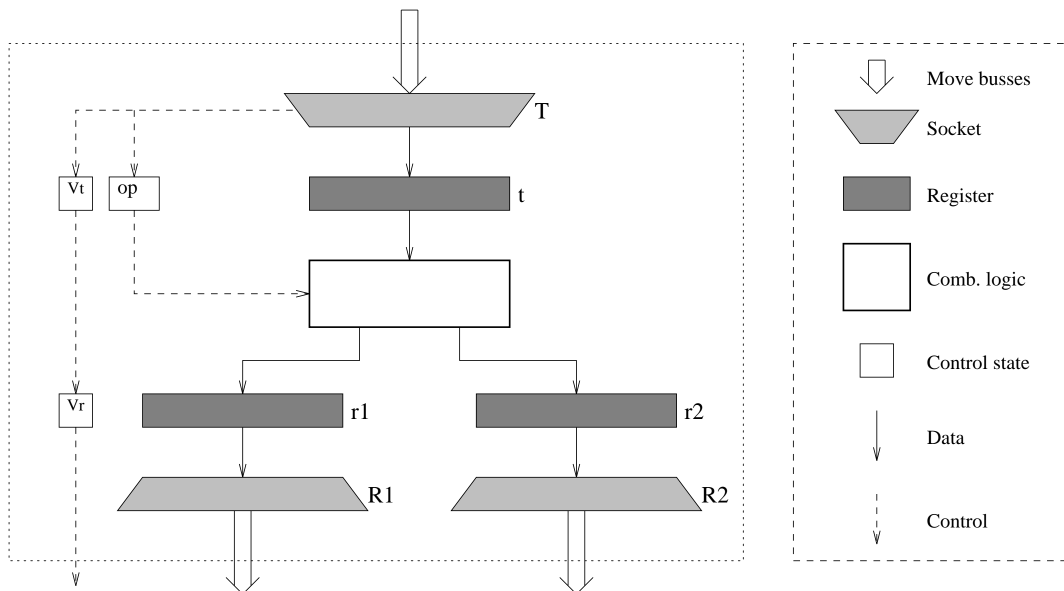
General

Shift units perform arithmetic and logic word-size shift operations over 1 bit or 2 bits.

Shift unit parameters

BUSWIDTH	Word size
SHIFUS	Number of instantiations

Shift unit structure



Shift unit operations

Operation	Semantics
–implicit–	$v \leftarrow f_{hybrid}(v, T, R1, R2),\ v : \{r1, r2\} \leftarrow f_{shi}(op, t)$
$A_{src} \rightarrow A_T$	$t \leftarrow src,\ op \leftarrow [ASR\ t]$
$A_{src} \rightarrow A_T + 1$	$t \leftarrow src,\ op \leftarrow [LSR\ t]$
$A_{src} \rightarrow A_T + 2$	$t \leftarrow src,\ op \leftarrow [SL\ t]$
$A_{src} \rightarrow A_T + 3$	$t \leftarrow src,\ op \leftarrow [undefined]$
$A_{R1} \rightarrow A_{dst}$	$dst \leftarrow r1\ [1\text{–}bit\ shift]$
$A_{R2} \rightarrow A_{dst}$	$dst \leftarrow r2\ [2\text{–}bit\ shift]$

Shift unit control support

Socket	Lock support	Exception support
T	—	pipeline full
R1	read	pipeline empty
R2	read	pipeline empty

Notes

- Reading either R1 or R2 invalidates both results and allows the pipeline to advance.
- The operation for $A_T + 3$ is not defined; it is used only to get a trigger socket address range equal to a power of two (see 3.2.2).
- The result valid bit v_r is available for reading in the valid unit.

4.9. Valid unit

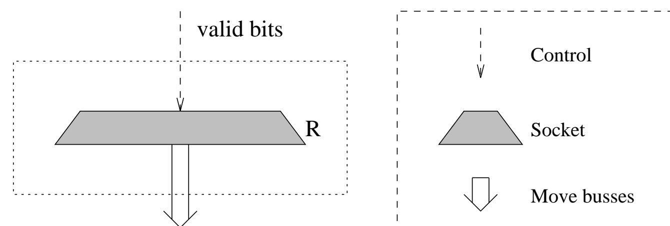
General

The valid unit provides access to the result valid bits of FUs that have one. There is always exactly one valid unit.

Valid unit parameters

BUSWIDTH Word size

Valid unit structure



Valid unit operations

Operation	Semantics
$A_R \rightarrow A_{dst}$	*dst* $\leftarrow$ *valid bits*

Valid unit control support

Socket	Lock support	Exception support
R	—	—

Notes

- The valid unit contains a string of valid bits. For the ordering of bits in a word see 3.2.2. Note that the output socket is not an ordinary one: on different busses, different valid bits are presented.
- The valid unit can always be read, like a general purpose register. There is no pipeline control.
- This unit is intended for exception state saving support.

4.10. Register units

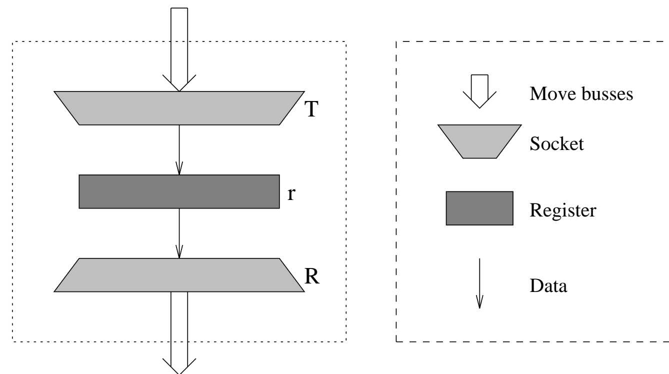
General

Register units contain general purpose registers. No operation is performed on data.

Register unit parameters

BUSWIDTH	Word size
REGFUS	Number of instantiations

Register unit structure



Register unit operations

Operation	Semantics
$A_{src} \rightarrow A_T$	$r \leftarrow src$
$A_R \rightarrow A_{dst}$	$dst \leftarrow r$

Register unit control support

Socket	Lock support	Exception support
T	—	—
R	—	—

Notes

- There is no need for pipeline control on register units. The registers are always available for read and write access.

4.11. Scan register units

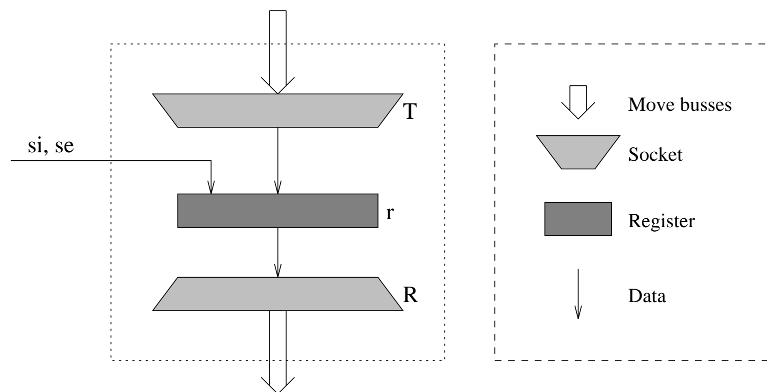
General

Scan register units contain general purpose registers. No operation is performed on data. External scan input and scan enable connections are provided for testing.

Scan register unit parameters

BUSWIDTH	Word size
SCRFUS	Number of instantiations

Scan register unit structure



Scan register unit operations

Operation	Semantics
$A_{src} \rightarrow A_T$	$r \leftarrow src$
$A_R \rightarrow A_{dst}$	$dst \leftarrow r$

Scan register unit control support

Socket	Lock support	Exception support
T	—	—
R	—	—

Notes

- There is no need for pipeline control on scan register units. The registers are always available for read and write access.
- See chapter 7 for details on the external scan inputs.

Chapter 5. Pipelining and control

Pipelining and control of transport network and functional units has already been discussed to some extent in 3.2.1. This chapter contains a more detailed description of pipelines and control signals. Generic control equations are presented together in section 5.2.

5.1. Data transport network

Pipelining of the data transport network has been discussed in 3.2.1. In this section, the relation between instruction and transport pipelines will be considered in more detail, and transport pipeline control signals will be explained.

As has been discussed in section 3.2.1, the pipeline of the instruction fetch unit seems one stage deeper than is obvious from the pipeline diagram (see 4.1). The instruction fetch unit is the only one where transport network pipelining becomes visible this way. Actually, the transport network pipeline may be considered as just a part of the instruction pipeline, because the transport pipeline is strictly synchronized with the instruction pipeline.

The instruction fetch unit pipeline is self-triggered, meaning that a new item (*pc)* is entered each cycle when the processor is not locked. The transport pipeline operates similarly: the pipeline advances each cycle when the processor is not locked. This instruction and transport pipeline is shown schematically in figure 5.1.

In the figure, three stages are shown: IF (Instruction Fetch), DC (DeCode), and MV (MoVe). 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 *moves* and the actual data transfer occurs. The MV stage state is kept in select and opcode registers in the sockets.

Figure 5.2 shows details of an input socket. Decode logic and MV stage registers, control, and data multiplexor are shown. Details of the socket signals are presented in the next section; here, note the glock input, which can disable updating of the MV stage registers. The instruction fetch unit responds to the same glock signal, holding its IF and DC stage registers when it is raised.

Squash signals are also part of the instruction pipeline, although not shown in figure 5.1. The squash signals originate from the guard unit, to implement guarded *move* execution, or from the instruction fetch unit, to discard instructions remaining in the pipeline on exception. Squash signals are generated during the MV stage, and determine which data bus and opcode, if any, are ultimately selected in a socket.

For clarity, the socket control equations are presented together with the FU stage control equations in the next section.

5.2. FU hybrid pipelining

Hybrid pipelining is used in functional units to maximize 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

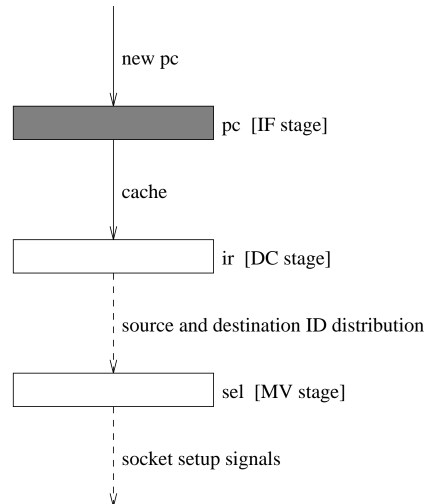


Figure 5.1. Schematic instruction/transport pipeline

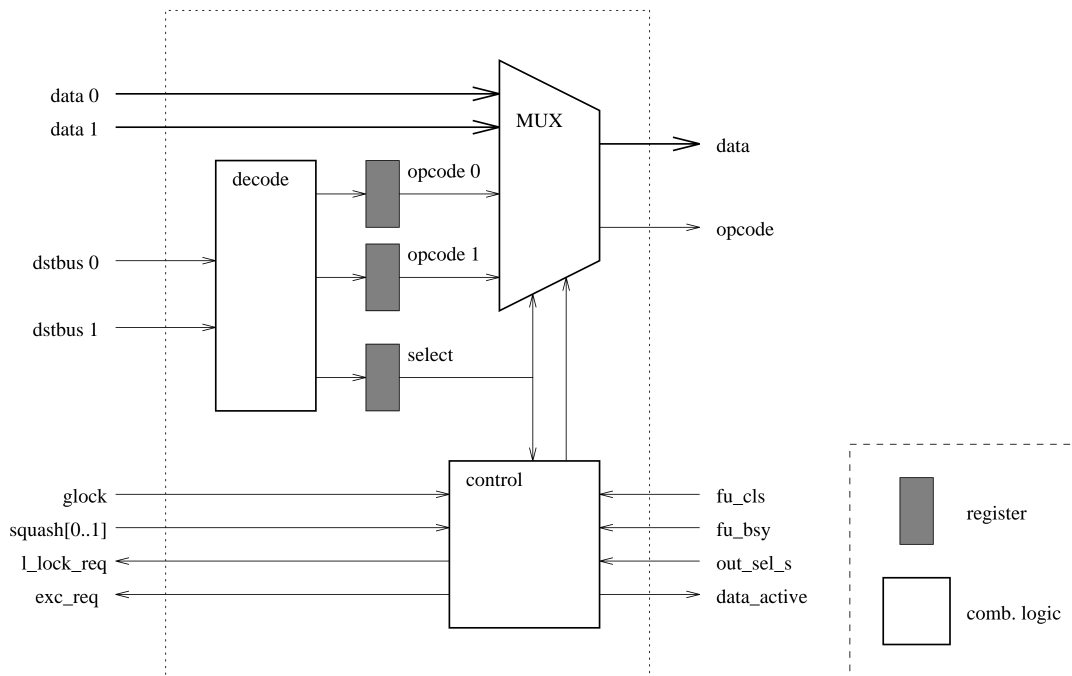


Figure 5.2. Input socket connecting to two *move* busses

advances to the next stage as soon as that stage becomes free.

FU hybrid 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 stall 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.

Below, a detailed control signal description is presented of the FU hybrid pipelines (stage control), the connection with the transport network pipeline (socket control), and the transport network (network control). The hybrid pipeline control is the most complex part; description of the other signals is presented here too because of the strong relationship between the systems. The relation between the systems is shown in figure 5.3.

Besides clock and reset, there are three kinds of global control signals:

lock The global lock signal halts the instruction and transport pipeline when raised.

squash Squash signals inhibit the execution of (individual) move operations but do not affect the instruction and transport pipeline.

exception The internal exception signals are generated by FU pipelines to indicate a pipeline error.

The control logic takes care of generation and distribution of these signals, and derives appropriate local setup signals from them.

Input socket

An input socket with control signals is shown in figure 5.2. The *in_sel_s* signal (used within the socket only) indicates whether the socket is selected (taking squash masking into account). The decode result for bus i is latched in $select_i$.

$$in_sel_s = \bigvee_{i=0}^{N_b-1} \overline{squash_i} \wedge select_i \tag{5.1}$$

Where N_b is the number of busses connected to the socket. The *in_sel_s* signal is combined with the global lock signal to determine if a data item will actually pass through the input socket in the current cycle:

$$data_active = \overline{glock} \wedge in_sel_s \tag{5.2}$$

Data and opcode from bus i will be selected if:

$$bus_select_i = \overline{squash_i} \wedge select_i \tag{5.3}$$

Local lock requests from input sockets are bus-local signals, prompted by move operations to a FU pipeline that is not ready yet:

$$l_lock_req_i = select_i \wedge fu_bsy \tag{5.4}$$

for bus i. The *fu_bsy* input of the input socket connects to the *in_bsy* signal from the first stage controller.

An exception is generated if the FU pipeline is full, the input socket is selected for writing, and the output socket is *not* selected for reading. The latter is because if the pipeline result is read in

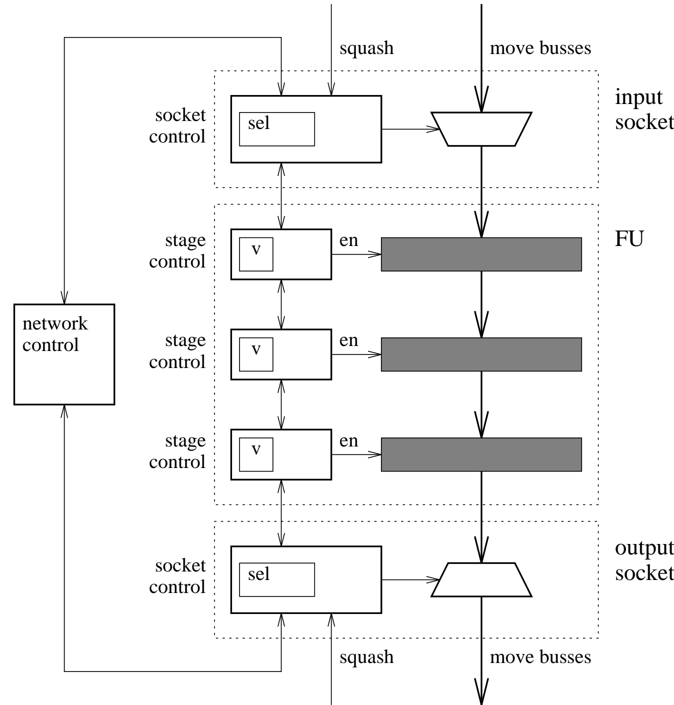


Figure 5.3. Pipeline control structure

the cycle when a new item is entered, all pipeline stages can advance.

$$exc_req = fu_cls \wedge in_sel_s \wedge \overline{out_sel_s} \tag{5.5}$$

The *fu_cls* input connects to the *in_cls* signal from the first stage controller, and the *out_sel_s* signal comes from the output socket at the other side of the pipeline.

The select and opcode registers (MV stage) enable depends on global locking:

$$mv_reg_enable = \overline{glock} \tag{5.6}$$

Stage control

Figure 5.4 shows a generic stage control section (for a 2-cycle stage) with its signals.

The data register of a pipeline stage is loaded with a new value when the previous stage has data available (*in_active*), the pipeline is not busy at this stage or further on (*in_bsy*), and the pipeline is not full till the and and not being read (*in_cls*, *socket_cls*):

$$enable = in_active \wedge \overline{in_bsy} \wedge (\overline{in_cls} \vee \overline{socket_cls}) \tag{5.7}$$

The *in_active* input connects to the *out_active* of the previous stage or the *data_active* of an input socket (in which case the use of *in_bsy* is redundant). The *socket_cls* signal coming from the output socket indicates that no result is being read. The *in_bsy* and *in_cls* signals generated by the stage controller itself are defined below.

The valid bit V_0 is assigned the following value:

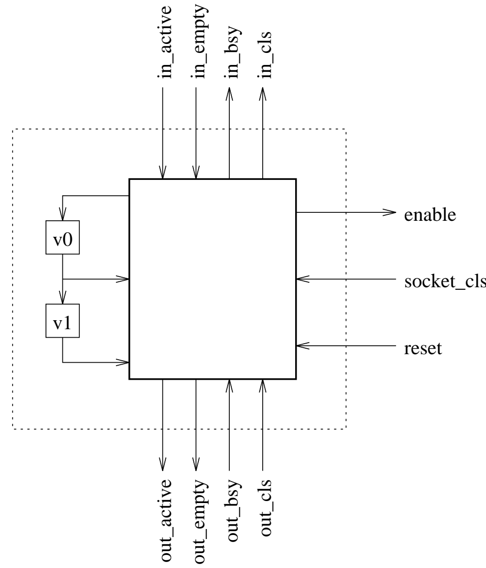


Figure 5.4. Stage control with two-cycle delay

$$V_0 = in_active \vee in_bsy \vee in_cls \wedge socket_cls \tag{5.8}$$

so it is set when *in_active*, and remains set in case the data in this stage cannot advance to the next. Valid bits for delay cycles shift data from V_0 unless cleared:

$$V_n = V_{n-1} \wedge not_clear,\ \ n > 0 \tag{5.9}$$

and the delay bits are cleared when the pipeline can advance:

$$not_clear = in_bsy \vee out_cls \wedge socket_cls \tag{5.10}$$

where *out_cls* is an input, connecting to the *in_cls* signal from the next stage, or fixed ‘1’ for the last stage.

The *in_bsy* signal indicates whether the pipeline is busy from here on, due to multi-cycle stages that are not yet finished. This is true when the stage is occupied (V_0) and not yet finished with delay cycles (*out_active*, see below) or the pipeline is busy further on:

$$in_bsy = V_0 \wedge (\overline{out_active} \vee out_bsy) \tag{5.11}$$

with *out_bsy* an input connecting to *in_bsy* from the next stage, or fixed ‘0’ for the last stage. The *in_cls* signal, asserted when the pipeline is full from here on, is simply:

$$in_cls = V_0 \wedge out_cls \tag{5.12}$$

Whether the stage is finished, and data can be loaded into the next stage, is indicated by:

$$out_active = V_N \tag{5.13}$$

With V_N the last delay valid bit, or equal V_0 for a single-cycle stage. This signal indicates that the pipeline is empty from the beginning:

$$out_empty = \overline{V_0} \wedge in_empty \tag{5.14}$$

where *in_empty* is an input connecting to *out_empty* from the previous stage, or fixed ‘1’ for the first stage.

Output socket

The *out_sel_s* output is like the input socket *in_sel_s* signal:

$$out_sel_s = \bigvee_{i=0}^{N_b-1} \overline{squash_i} \wedge select_i \tag{5.15}$$

Data will be put on bus i if:

$$bus_select_i = select_i \tag{5.16}$$

Depending on squash and lock signals, the receiving input sockets may reject the data. The following signal indicates to the stage controllers that the output socket is not used for reading:

$$socket_cls = glock \vee \overline{out_sel_s} \tag{5.17}$$

Output socket local lock requests are asserted when the socket is selected, there is no result ready, and the pipeline is not empty:

$$l_lock_req_i = select_i \wedge \overline{fu_active} \wedge \overline{fu_empty} \tag{5.18}$$

for bus i, where *fu_active* connects to the final stage *out_active* signal, and *fu_empty* to the final stage *out_empty* signal.

An exception is generated if an attempt is made to read the pipeline result while the pipeline is empty:

$$exc_req = fu_empty \wedge out_sel_s \tag{5.19}$$

And the select (MV stage) register enable is:

$$mv_reg_enable = \overline{glock} \tag{5.20}$$

Network control

The equations below are somewhat simplified, only the part relevant for generic FUs is shown.

The local lock requests are gathered:

$$l_lock_i = \bigvee_{j=0}^{L_i-1} l_lock_req_{i,j} \tag{5.21}$$

for bus i, with L_i the number of local lock request signals from the units for bus i. The global lock signal that is distributed to all pipelines is masked with squash signals, because local lock requests from a squashed move are not valid:

$$glock = \bigvee_{i=0}^{N_b-1} \overline{squash_i} \wedge l_lock_i \qquad (5.22)$$

Where N_b equals PARMOVES. The exception signal to the instruction fetch unit is simply:

$$exc = \bigvee_{k=0}^{N_{exc}-1} exc_req_k \qquad (5.23)$$

With N_{exc} the number of *exc_req* signals from functional units.

Note: the equations presented above are the generic versions, without reset taken into account. In many cases, simplifications are possible, e.g. no *bsy* chain is necessary when a FU has only stages with single cycle delay. Also, slight but obvious changes may be required in special cases (e.g. combining socket control signals for a FU with two output sockets). The instruction fetch unit has special stage control which is relatively complicated due to cache control and exception mode requirements. More information on the instruction fetch unit can be found in chapter 6.

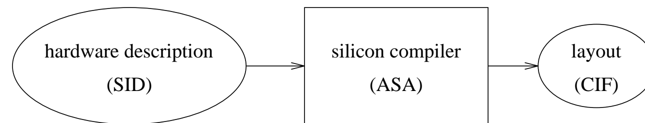
Chapter 6. Implementation

This chapter provides information on the implementation level of the MOVE processor generator design. The first section summarizes ASA. Sections 6.2 to 6.4 give an overview of how ASA's hardware description language SID has been applied, and how it has been augmented to cope with parameters of the MOVEASAINT architecture. The last two sections provide for MOVEASAINT implementation details.

6.1. ASA introduction

The ASA silicon compiler is the primary tool used for implementing the MOVE processor generator. ASA is an integrated software package that takes high-level hardware description as input and includes all the functions required to generate and evaluate chip layout from there [16]. This section gives a quick overview of major features of ASA; for details consult the ASA manuals.

In a nutshell, this is what ASA does:



Of course this view of ASA is oversimplified, many functions are available for transformation and verification of designs, as outlined below.

ASA generates full custom CMOS layout for digital circuits, and can be characterized for different processes. Layout generation is fully automatic, but some manual changes are possible.

The SID hardware description language used by ASA is syntactically similar to the Pascal programming language, and allows for a mix of high and low level statements much like the C programming language. Necessarily, SID (as a hardware language) differs substantially from programming languages though. Especially the parallelism required to describe hardware adequately has profound effects on the language design; also, only a limited set of high level constructs can be transformed into a low level description suitable for generating layout.

Major levels of specification possible in SID are:

Behavioral High level description similar to high level programming language. Behavioral SID can be simulated, but may not be suitable for synthesis (transformation to gate level).

Synthesizable Like behavioral, but without constructs that cannot be transformed to gate level. Synthesizable SID does not allow the use of variables to describe state in a circuit, nor can arithmetic operators like '+' be synthesized. For these, library cells must be explicitly connected, or a gate level description must be created (or a mix of the two).

Structural Structural SID is a netlist description using predefined cells; it is also the synthesis target of synthesizable SID.

In the MOVE processor generator, behavioral SID is used for a functional description of the system, and a combination of synthesizable and structural SID is used for the implementation.

ASA can perform a variety of operations on a SID source. Major ASA functions are:

- SID compilation. ASA can compile SID sources into an internal format that is suitable for simulation, analysis, or synthesis. Various checks are performed during compilation. Note that different purposes impose different constraints on the SID source (e.g. some statements are not synthesizable).
- Synthesis. High level SID constructs can be transformed into structural SID. The FSM command takes a finite state machine description that consists of states, output expressions, and transitions; it does state assignment and determines new state functions, suitable for synthesis with the TOGATES command. The TOGATES command can transform (a limited set of) expressions into a netlist of gates. Timing optimization on specific paths through the circuit is possible when TOGATES is used.
- Buffering. An existing circuit can be improved by adding or modifying buffers according to loading estimates. Dedicated facilities for clock buffering are available.
- Layout generation. Fully automatic layout generation is possible with ASA. ASA layout consists of a set of interconnected ‘standard cells’, each of which consists of interconnected ‘basic cells’. Different placement and routing algorithms are used at the two levels. For reasonable results, layout hints in the SID source are often required. It is possible to manually modify layout (placement of standard cells).
- Database. Entire designs, at any stage of synthesis or layout, can be saved and restored using the database. Also, ASA-generated cells or cells from other sources can be stored and retrieved from there.
- Simulation. A compiled representation of SID of any type can be simulated to test the circuit. Test vectors can be generated using most SID features; verification of simulation results can be done interactively, or predefined checks can be performed automatically.
- Timing analysis. Performance of a circuit can be estimated, with or without taking layout information into account. Verification of timing correctness versus predefined constraints is also possible. Automatic feedback of timing results to the synthesis phase is possible for an improved synthesis iteration, although this works only with small systems.

ASA lacks provisions for editing layout at a low level, but can import layout. All in all, ASA is a complete design tool for quick IC realization of a digital circuit, and does well on random logic; it is less suitable for designs which require high performance or low area use.

6.2. MOVEASAINT system hierarchy in SID

A MOVEASAINT architecture implementation in SID is unavoidably quite large. Like any design with complexity beyond a basic level, to keep the implementation understandable and manageable, the system must be partitioned into sufficiently small parts. These parts are best organized in a

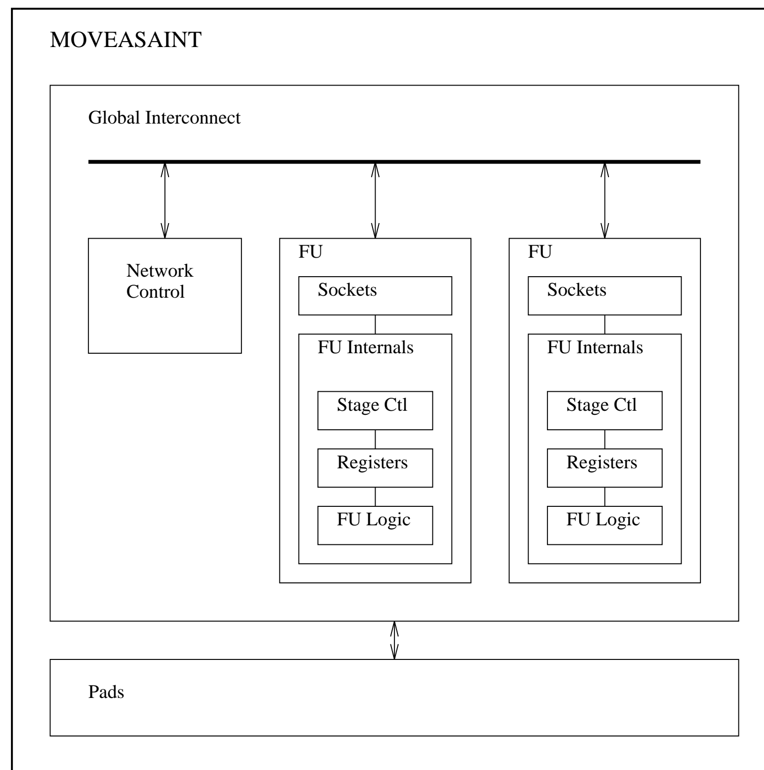


Figure 6.1. MOVEASAINT SID hierarchy in the MOVE processor generator

hierarchy, where each level of the hierarchy corresponds to a meaningful level of abstraction.

A hierarchical, partitioned design not only makes the design better manageable. A well-chosen partitioning also improves final layout and timing by keeping communication local within one part as much as possible.

In the MOVE processor generator, there is an another factor in the partitioning of the design. There are many parameters determining the configuration of functional units and interconnect. This parametrization is responsible for significant part of the complexity of the design. Therefore, the implementation of the MOVE processor generator is partitioned not only to keep communication local and hide *functional* complexity at hierarchical levels, but specifically to hide *parametrization* complexity as much as possible. So the aim is not only to minimize functional dependency, but also parameter dependency between parts of the system.

With this in mind the SID hierarchy chosen for the MOVEASAINT, shown in figure 6.1, will be clear. At the highest level, just the pads and the processor core interface are visible. At the *global interconnect* level, only move busses, global control, and FU interfaces are visible. So, mainly the bus connect and 'number of FUs' parameters are used here. At the *FU* level, only the sockets and the FU internals interface are visible. This level uses the bus connect parameters and knowledge of FU interface properties. Beyond this level, at the *FU internals* level and below, no dependency on connection parameters exist, only FU specific parameters. Within the FU internals, the implementation is split in *stage control, registers,* and *FU logic,* the latter usually containing only combinatorial logic. Each of these has an obviously distinct function and depends on different FU-specific parameters.

For layout, the design can be grouped at different levels for generation as a standard cell: either

at the leaf system level, at the FU level, or at the global interconnect level.

6.3. Limitations of SID

Unfortunately, ASA has several limitations that pose a problem for implementation of the MOVE processor generator. Especially implementation of parametrized features suffers from unexpected limitations; the current version of ASA is unable to compile many SID constructs that are legal according to the documentation. There are also several problematic limitations that are documented, most of which have to do with variable hardware structures. Apparently the SID compiler was implemented with fixed hardware structures in mind.

Note that in the remainder of this section some knowledge of ASA and SID is assumed.

SID limitations that pose problems for parametrization include limitations of the use of *for* and *if* statements. The *for* macro statement is specified as (and implemented as) a macro expander, but this is not supported by the rest of the compiler (beginning with the lexical analyzer). It is not possible to expand incomplete SID statements. This results in the following problems:

- Expressions with variable number of input signals are not possible. A workaround exists using arrays of signal variables, but this is not only obscure, it also tends to reduce performance because of the logic synthesis algorithms used by TOGATES. If the *for* statement would have acted as a true macro expander, it would have been possible to generate the required expressions.

- It is not possible to use *for* index variables (or variables derived from it) in ranged array indices. Therefore it is impossible to partition a bus, for example.

- Variable nesting of statements (like *if–then–else)* is not possible. Recursive system instantiation is also not supported to implement this.

- It is not possible to generate a series of identifiers, for example to implement a variable number of busses. Together with the impossibility of using a *for* variable in ranged array indices, and the limitations on the use of multidimensional indices, this makes the use of a variable (parametrized) number of busses particularly troublesome.

The *if* statement poses problems similar to those with *for.* It is specified to be a macro statement, like *for,* but not applicable as such. So even if test expressions can be evaluated at compile time, the following problems arise:

- The branches of *if* statements for which the test expression evaluates false are not properly ignored. This results in name space clashes and compiler (declaration) errors for the *use* system parts.

- Conditional *contact* part entries are not allowed.

- Linked with the previous problems, the compiler generates errors when it encounters identifiers in function parts that it considers not appropriately declared, even if the identifier is used only in the unused part of a conditional statement.

Other SID limitations that hinder parametrization:

- Arrays of variables are not allowed.

- Memory variables can be at most 16 bits wide.
- Stimuli (test vector) files do not handle SID subsystem parameters appropriately, so parameters have to be hard-wired in the stimuli file.

Presented here for completeness, other general ASA limitations that can be problematic for a large and parametrized design:

- The TOGATES synthesis command can handle only 255 signals in one system.
- The BUFFER CLOCK command cannot generate a clock net if there is a real cell in a leaf system.
- The simulator does not handle bidirectional contacts correctly.
- The timing analyzer can not handle large designs.

The inevitable conclusion that must be drawn from all these problems is that the SID language is not well suited for implementation of the MOVE processor generator. Also particularly questionable is the fact that most of these limitations can not be found in the ASA documentation. An overview of how ASA has been extended to support implementation of the MOVE processor generator is presented in the next section.

6.4. Parametrizing SID

The problems of SID with parametrization, described in the previous section, have been alleviated by means of preprocessing. The sources of the MOVE processor generator are preprocessed to implement those necessary features which are lacking from SID. The result of the preprocessing is SID suitable for compilation by ASA.

The basic purpose of the preprocessing in its current state is to support true textual loop expansion and conditional code inclusion. This way, the limitations of the SID *for* and *if* statements are circumvented. The basic preprocessing framework is provided by *cpp*, the C preprocessor; *for* macro expansion is performed by *cp4*, a tool implemented for this purpose (see below); parameter evaluation that is beyond the scope of *cpp* is performed by the *m4* macro preprocessor found on UNIX systems.

In hindsight, it would probably have been better to implement the entire preprocessing using the powerful *m4* preprocessor exclusively. However, as has been noted already in the previous section, the limitations of SID are not well documented. Because of this, the necessary extent of preprocessing became only gradually clear, and the preprocessing implementation has grown gradually to cope with emerging problems. Initially, only *cpp* was used for preprocessing. As the the preprocessing requirements grew, *cp4* and *m4* were added.

The *cp4* filter is an implementation of a *for* macro loop expander in C. Its syntax is as follows:

```
$for[1] <identifier> <first index> <last index>
<stuff>
$endfor
```

It repeats text bracketed in *$for* and *$endfor* delimiters as many times as the loop bounds indicate; all occurrences of the identifier are substituted with the current loop index value. A *$for* statement example:

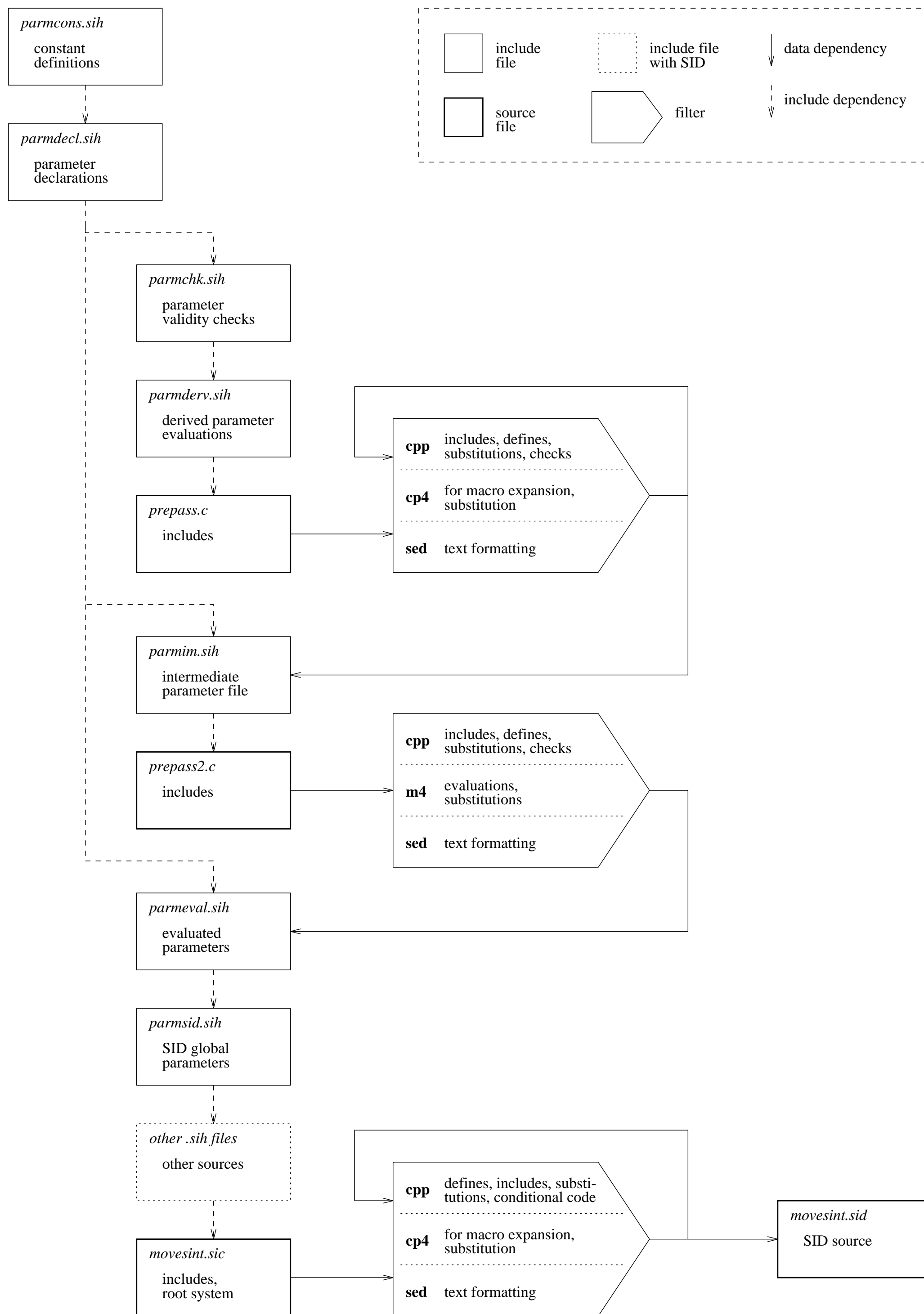


Figure 6.2. SID source preprocessing in the MOVE processor generator

```
$for FORVAR 0 3
  busFORVAR[0..31]: input;
$endfor
```

This is expanded by *cp4* into:

```
bus0[0..31]: input;
bus1[0..31]: input;
bus2[0..31]: input;
bus3[0..31]: input;
```

Figure 6.2 shows how the preprocessing is organized. Parameters are evaluated first (using *m4* primarily), and placed in a file (*parmeval.sih);* these evaluated parameters are used in *cpp* and *cp4* statements later on to produce SID source text. In the figure, the dependencies of the *movesint.sic* file are shown; other *.sic* files are processed similarly. Note that the elements of ‘filter blocks’ in the figure are applied on several passes, as indicated by the feedback data dependencies. Details of the actual order of processing can be found in the *pc* file.

6.5. Details of MOVEASAINT implementation

Much of the MOVEASAINT implementation is visible in the architecture, see chapters 3 to 5. Implementation of most functional units and components is straightforward from the architectural definition. The instruction fetch unit is an exception, though, and implementation details of this unit are presented below.

The design of the instruction cache for the MOVEASAINT is determined by efficiency trade-offs (see [12]), some of which depend on ASA constraints. To begin with, because the cache of the MOVE32INT configuration is small (32 instructions), and cache configurations are expected to be small for typical configurations, a direct mapped cache configuration is a good choice. The simple structure and control of a direct mapped cache not only reduce area relatively much for small caches, but also reduce design time.

For practical processor configurations, cache subblocking provides no advantage. MOVE processor instructions tend to be wide (e.g. 64 bits in the MOVE32INT), but ASA RAM cells have limited word size, depending on the number of words in the cell. If cache line size exceeds a threshold determined by maximum RAM cell word size, more RAM cells must be used. Use of subblocking increases line size, and may therefore increase the required number of RAM cells (even though the number of RAM bits decreases). Increasing the number RAM cells is very expensive in ASA, due to increased routing area (more cells must be addressed) and address decode logic overhead in RAM cells (two cells of half size are considerably larger than one cell). The specification of the ASA RAM cell shows that even just the address decode overhead cancels the storage are reduction of subblocking for typical cases.

Note that the current implementation does make use of multiple RAM cells when necessary to allow for wide instruction words, even though the number of cells is minimized for area.

The RAM cells in ASA are not resettable. Implementing cache entry valid bits with regular resettable flip-flops is extremely expensive in ASA; therefore valid bits have not been implemented, and a cache flush operation is not supported. The cache can be enabled or disabled to provide for the necessary control.

Figure 6.3 shows an overview of the instruction cache datapath. Note that the ASA RAM cell is synchronous: all inputs of the cell are clocked. Expecially the RAM address latch has an impact on the cache design, and is shown explicitly in the figure. Also note that the tag compare results are latched in *ir_valid* and influence control one stage late. This creates a pipeline control hazard that is repaired with the use of the *pcdc* register.

Consider the case when the cache is enabled, cache hits occur, and there are no locks. Then the RAM address latch is loaded each cycle with a new PC taken from *pc select* directly, and the

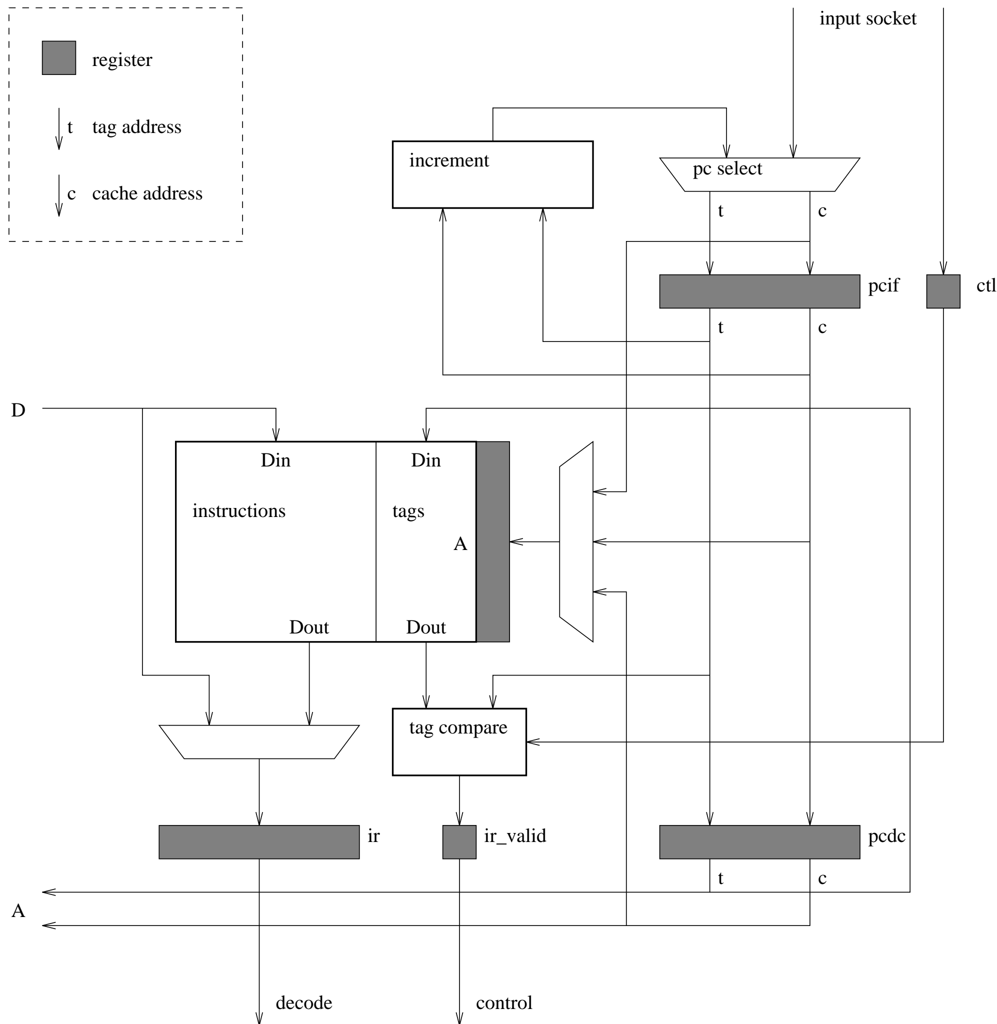


Figure 6.3. Simplified instruction cache datapath

instruction register *ir* is loaded with the result of a cache lookup. The *ir_valid* latch indicates that *ir* is valid and can be used. Both *pc* registers remain unused (for cache operation); the cache latch represents the IF stage register in this case.

Now when a global lock is raised, the pipeline must stall. However, the RAM cell latch cannot be controlled, and is loaded unconditionally each cycle. Therefore the cache address inputs are switched to *pcif* register outputs when a lock occurs, to implement a hold function.

When a cache miss occurs, the *ir* register is loaded with invalid data and *ir_valid* with the ('invalid') tag compare signal. After that, a global lock request is made to stall the pipeline. The cache control state machine continues operation and initiates a fetch from external memory, independently of the lock. The address is available from *pcdc*; the *pcdc* register is updated synchronously with the *ir* register and contains the address associated with the invalid instruction in *ir* after a miss. The fetch is completed with a cache write (with the cache address and tag taken from *pcdc)* and a direct *ir* write with the fetched instruction. To recover from the pipeline hazard, the RAM address latch is loaded again with the address that is still in *pcif* for another lookup. This second lookup for the address in *pcif* (the first lookup occurred during the first cycle the pipeline

was stalled to handle the cache miss) represents an avoidable ‘lost cycle’. With appropriate but expensive hardware the pipeline could continue one cycle earlier, so that the pipeline stalls only during external memory access, see below.

If the cache is disabled (by resetting the *ctl* register), the tag compare result is masked. The cache operates (loads) normally, but cache hits are not detected (and so not used). This way, the cache can be validated programmatically when necessary.

The cycle lost on recovery of the hazard can be avoided by using a second instruction register. This register should be loaded while the pipeline locks, with the result of the cache lookup that is otherwise lost during the cache write. Now, the pipeline can be restarted immediately after the cache write, without loading the cache address latch from *pcif* for another lookup for the address there.

The dual-ported RAM provided by ASA may seem a viable alternative to an extra instruction register (because the register is very large), but it is not. A second instruction register would indeed not be necessary if a cache lookup could be performed during the cache write. However, timing of the ASA dual-ported RAM cell is undefined if read and write access are to the same location (the timing analyzer is not aware of this!). This situation can arise in a cache implemented this way: after a jump of one instruction back. Detecting the collision with a comparator defeats the purpose of reducing area. Therefore the dual-ported RAM is not suitable for this cache.

Note that if cache miss latency is not considered critical, the selector that allows direct writing of *ir* may be omitted, introducing an extra recovery cycle to transfer the fetched instruction from the cache RAM to *ir*.

A description of the cache control implementation is beyond the scope of this report. Implementation of other parts of the instruction fetch unit is relatively straightforward. For details such as the exception priority encoding in the PC selector, or the PC chain stage control equations, see the sources in appendix D.

6.6. Limitations of MOVEASAINT implementation

A summary of limitations of the MOVE processor generator caused by the implementation:

- The *BUSWIDTH* parameter must be ≥ 3, because of the shift unit implementation.
- The number of result valid bits is limited in the valid unit to *BUSWIDTH* · *PARMOVES*.
- The valid unit allows access from only one *move* bus to a register of valid bits; the valid unit sockets do not support multiplexing.
- The functional version (suitable for simulation only) limits *BUSWIDTH* to at most 16, because of SID memory variables that cannot be larger.

The TOGATES synthesis command limitation that it can handle a maximum of 255 signals has been avoided in the MOVE processor generator by implementing all potentially wide paths in structural SID.

Chapter 7. Realization

Various MOVEASAINT realization level design decisions are discussed in 7.1. The testing strategy for the MOVE processor generator is presented in 7.2. The final section contains the external interface specification.

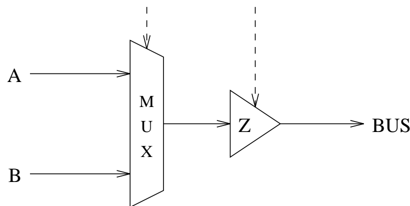
7.1. Design details

Below, realization level design details are presented for a few selected topics. The topics selected are those where interesting design decisions have been taken; included are sockets, valid unit, and clocking.

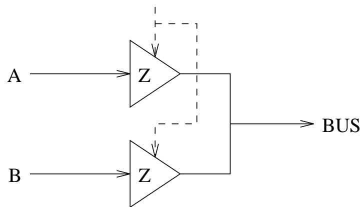
Output sockets

The *move* data busses are implemented as tri-state busses. Other bus types (e.g. precharged dynamic) are not well supported in the fully static ASA environment. The tri-state buffer output enable signals are driven directly from latches in the output sockets, so that transitions to and from tri-state on the bus are well synchronized. This way bus conflicts and bus delays are minimized.

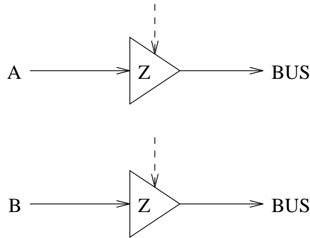
There are no MOVEASAINT output sockets which provide data multiplexing. This is unlike the SoG MOVE32INT, where all output sockets provide 2-way multiplexing (and general purpose registers share output sockets in pairs) to reduce bus load. Actually, when using ASA there is no advantage to combining register outputs in one socket. Consider the obvious way to combine two outputs:



which is actually the largest and, for busses that are not overloaded, the slowest. Currently, the data busses are not in the critical path for reasonable configurations anyway. A multiplexer in ASA is larger than a tri-state buffer, so this is smaller:



but, except for a very small control overhead, this is the same as using two separate output sockets:



which provides one generic solution for all cases. With full decoding implemented to avoid the scheduling limitations present in the SoG MOVE32INT output sockets, the previous solution would be practically indistinguishable from two separate sockets. Therefore the last solution is preferred, and the only one used in the MOVE processor generator.

Input sockets

Input sockets do not detect a collision – which is when the socket is accessed at more than one bus at a time. The current realization performs a logical *or* on colliding data items. This is actually a faster way to perform a bitwise *or* than using the logic FU (if connectivity and scheduling constraints allow).

Valid unit

The layout of the valid unit is special: the output socket is distributed over the units with result valid bits. If a normal output socket had been used, the outputs of all result valid bits would have to be connected to the valid unit output socket, which would require a lot of global interconnect. To avoid that, the bus buffers of the valid unit socket are placed locally on each FU, while the socket control (decode) remains at one place. All valid bits which fit in one word share the same select signal, so the global routing can be reduced significantly.

Clocking

ASA imposes a true single phase clocking scheme on circuits. This is not because of any fundamental limitation of ASA, but simply by convention throughout ASA's tools. The major reasons for using single phase clocking in ASA are: all supplied basic cells are static, and all storage elements are edge triggered; the clock buffering command can be used on only one clock signal; the timing analyzer assumes true single phase clocking.

ASA provides for a BUFFER CLOCK command, which analyzes clock load and generates appropriate buffering for the clock signal. The clock buffer is realized as a buffer tree of one or two levels deep (depending on the load), with all outputs on the deepest level interconnected. The wire used to connect the outputs is not wide, so a small clock skew is not guaranteed if there is high and uneven loading. Therefore the local buffer capacity is scaled with the local loading. Remaining skew is computed with special clock buffer analysis within the timing analyzer.

Note: in ASA version 5.4 the BUFFER CLOCK command contains a bug that is hard to track down. Apparently it is unable to generate a clock net if there is a real cell (e.g. s_ram) in the design *in a leaf system.* Therefore, always place cells on a system level closer to the root system.

True single phase clocking provides for fewer timing optimization opportunities than multi phase clocking. The MOVE processor generator has been designed from the ground up with true single phase clocking in mind. At a few places the implementation did not map directly to a satisfactory realization with respect to timing, though. At these places, the point of latching in the circuit has been moved to obtain a better balanced timing. This has been done in the guard unit (selection signal computation for the compare units moved to before the latches) and the instruction cache

(full tag compare is done in the same stage as the lookup).

7.2. Design verification

The verification strategy for the MOVE processor generator has been incremental testing, closely following implementation of levels in the design hierarchy (see 6.2). System components at all levels have been tested by means of manually created test vectors, before being assembled into larger components at the next hierarchy level (ad-hoc implementational checkout, see [11]). At each level, a functional version of a component has been implemented and tested before the synthesizable version. All testing has been performed using the ASA simulator.

No attempt has been made at full testing coverage of the design, but rather a reasonable coverage with bias towards potential problem areas. Full testing coverage would be extremely difficult to achieve, not only because of the complexity of the circuit, but especially because of the level of parametrization. That is, not just one processor has to be tested, but rather a processor *generator* with all its possible instances. With ASA, only one instance can be compiled and simulated at a time, and these operations consume considerable time. This makes it hard to verify all processor generator paths [8].

Independent testing has been performed at the following levels, each completed before construction of the next level:

1. FU pipeline control
2. Sockets
3. Complete FU control including sockets
4. FU internal (combinatorial) sections
5. FU internals (as a whole)
6. Complete FUs (including sockets)
7. Move busses (with FUs attached but without instruction fetch unit)
8. Complete processor (in various configurations)

Testing for levels 1 to 6 has been performed by isolating the system under test and providing hand-crafted test vectors to the system. Test vector generation and verification has been by means of SID stimuli files with *check* statements for automated verification. Automatic verification has been employed whenever possible, rather than interactive inspection of simulation results, for fast and reliable testing after each design modification.

Testing at the *move* bus level (7) has been performed through a dummy instruction fetch unit which connected to, and controlled, the busses.

The complete processor (level 8) has been tested by emulating external instruction memory containing programs. An elaborate testing facility has been created with the following features:

- Programs can be entered with instructions containing symbolic source and destination IDs.
- Configuration of the test system, and translation of symbolic IDs, adapt automatically to the parameter configuration.

- Automatic verification is possible, and several levels of diagnostic output are supported.

This test system can be found in the file *movesi.sic*, see appendix D. Using this system, processors have been tested with several small programs concentrating on the following features:

- FU functionality
- FU pipelining
- Move bus properties
- Guarding, especially narrowcast
- Single guard configuration (SINGLEGUARD)
- Instruction cache operation and global locking
- Exception mode

The design has passed all tests, defined beforehand, at all levels. **Note:** in ASA version 5.4, the simulator does not handle contacts correctly which have the attribute *bidirectional.* Bidirectional contacts simulate correctly if (tri-state) outputs are connected at both sides directly, without intermediate connect statements. In all other cases, simulation may fail while the circuit is correct. So, use *bidirectional* contacts only when necessary, and use signal variables to 'short circuit' what would otherwise require multiple connects.

7.3. External interface

The external interface of a MOVEASAINT instance is determined by parameters and process technology. The properties of I/O pads vary with the process used; for that reason, the signals are described below according to the *internal* specification. Externally, the interpretation of signals may be inverted. In the ES2/CMS1M0 and ES2/CMS1M5 processes used for the evaluation as described in the next chapter, all pads invert signals.

There are external signals for the instruction fetch unit, load–store units, scan register units, control, clock and power. In the lists below, ranges are indicated with square brackets; ASA removes the brackets for pad names, as shown in layout pictures (appendix C).

Instruction fetch unit external interface signals:

i [0..Instrsize-1]	instruction data inputs, with *Instrsize* as the evaluated instruction size
ia [0..Ifeaddrsize-1]	instruction address outputs, with *Ifeaddrsize* as the evaluated instruction address size
ias	instruction address strobe output, indicates address valid
il	instruction latch output, indicates termination of operation started with *ias*
int	external interrupt input

Load–store units external interface signals:

dN [0..BUSWIDTH-1] — load–store unit *N* bidirectional data bus, with the BUSWIDTH parameter

wnrN — load–store unit *N* *read/$\overline{write}$* output

daN [0..LDSADDRSIZE-1] — load–store unit *N* address outputs, with the LDSADDRSIZE parameter

dasN — load–store unit *N* address strobe output, indicates address valid

dlN — load–store unit *N* data latch output, indicates termination of operation started with *dasN*

Scan register units external interface signals:

siN — scan register unit *N* scan data input

seN — scan register unit *N* scan enable input

Control, clock, and power pads:

rst — $\overline{reset}$ input

ck — clock input

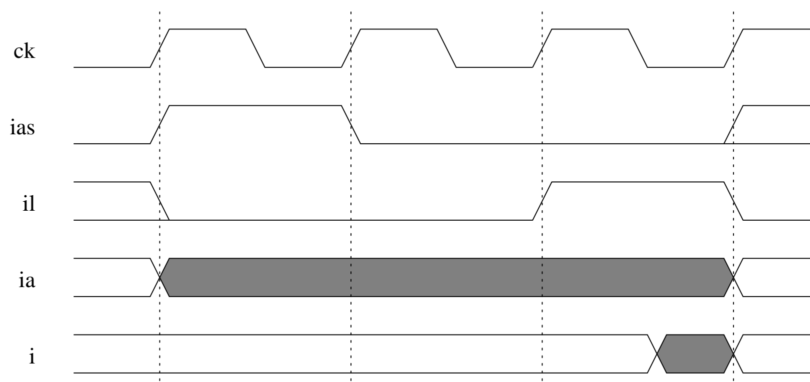
power — V_{dd} supply input

xpower [1..XPOWERPADS] — extra V_{dd} supply inputs, with the XPOWERPADS parameter

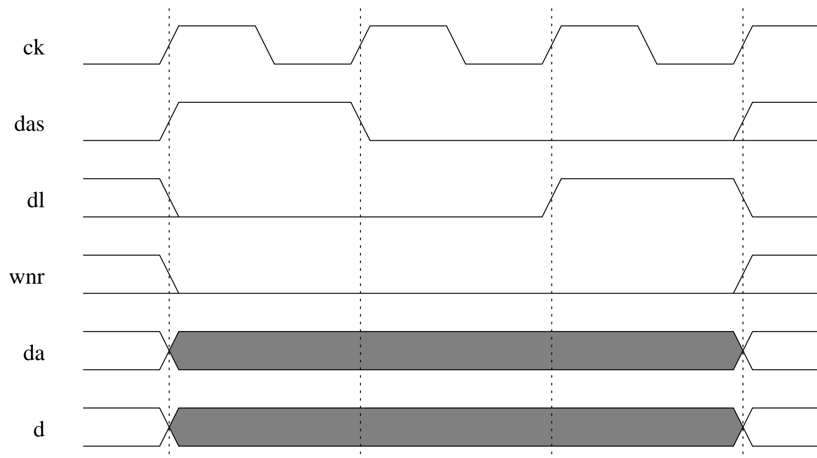
ground — V_{ss} ground input

xground [1..XPOWERPADS] — extra V_{ss} ground inputs, with the XPOWERPADS parameter

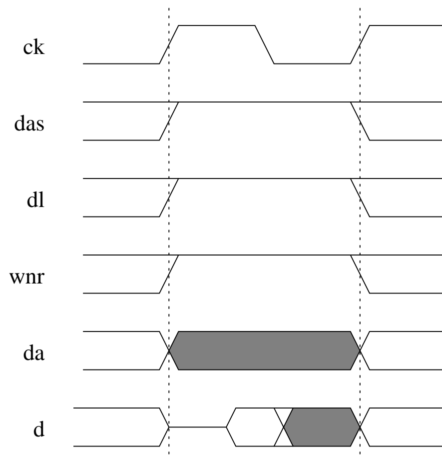
External instruction fetch operation with IFEFUDELAY set to three:



Store operation with LDSFUDELAY set to three:



Load operation with LDSFUDELAY set to one:



Notes

- The processor is guaranteed to be not driving the *d* lines only during a load operation (from *das* to *dl*).
- For scan register operation see the ASA *s_d_reg* composed cell documentation.
- All inputs are synchronous, latched on the rising clock edge.
- The *rst* input must be asserted for at least 2 cycles to guarantee successful reset with all FU pipelines invalidated.
- Assertion of the *int* signal must be removed programmatically before return from exception, see 4.1

- Timing analysis is performed by ASA under the assumption of a symmetric clock input. For optimal cache RAM timing in cases cache operation determines the critical path, an asymmetric clock signal may be required. Currently, cache timing is not critical. Actual timing details depend on parameter settings and process parameters. For more information on timing see 8.2.

Chapter 8. Evaluation of realization

For evaluation, layout has been generated and timing analysis has been performed for a MOVEASAINT instance. The parameter configuration used is the MOVE32INT compatible configuration, see appendix A for complete parameter details. Note especially that the layout has been generated with standard cells on the FU level; using standard cells on the leaf system level results in considerably higher area use. Currently, ASA cannot generate a standard cell for the entire (MOVE32INT configuration) processor core, because it is too large. Also, note that the ASA version of the MOVE32INT features single-cycle integer units, in stead of the sub-pipelined two cycle integer units in the SoG version.

The results are compared in the following sections with the SoG MOVE32INT specifications. These results depend not only on the design method (Sea of Gates versus ASA's full custom / standard cell) but also on process characteristics. Both the technology used for the SoG MOVE32INT and the technologies used for the MOVEASAINT are CMOS, two metal layer processes (see table 8.1). The SoG and ES2/CMS1M5 processes are actually nearly identical, with 2.0 μ minimal feature size.

In the following sections, the results of the MOVE processor generator for the MOVE32INT compatible processor will be analyzed and compared with the SoG MOVE32INT. Topics of the sections are area, timing, and scalability. Power evaluation is lacking, because the tools currently do not calculate power consumption accurately.

8.1. Area evaluation

A summary of physical dimensions for SoG and ASA MOVE32INT realizations can be found in table 8.2. Note that no manual changes have been made to the ASA-generated layout; layout generation has been completely automatic[1]. Plots of layout can be found in appendix C.

The most important architecture and implementation changes of MOVEASAINT versus the SoG MOVE32INT that influence area are:

- The MOVEASAINT implements FU-generated exceptions. This requires additional FU pipeline and global control.
- The instruction fetch unit can use any vector for reset, interrupt and exception.
- The MOVEASAINT implements multiple guarded writes correctly to all destinations. This requires significant overhead in input sockets, especially for opcode state.
- Decoding on different sockets is fully independent, even within the same FU.
- The compare units have an output socket that enables reading of the operand register, which is necessary for full state saving on exceptions. The MOVE32INT lacks this socket.

[1]Placement of I/O pads has been specified explicitly by means of a pad file.

Process	Type	Gate length	Metal layers
Fishbone SoG	CMOS	1.6 μ	2
ES2/CMS1M5	CMOS	1.5 μ	2
ES2/CMS1M0	CMOS	1.0 μ	2

Table 8.1. Primary process characteristics for SoG and ASA MOVE processor realizations

Realization	Area	Size
SoG	$1\,cm^2$	1×1 cm
ASA 1.5 μ	$2.61\,cm^2$	1.42×1.84 cm
ASA 1.0 μ	$1.07\,cm^2$	0.90×1.19 cm

Table 8.2. Area comparison of MOVE32INT realizations

– The load–store unit in the current MOVEASAINT is less pipelined than the SoG MOVE32INT version.

When comparing the ASA MOVE32INT realization with the SoG one, consider that basic technology differences between ASA and Sea of Gates strongly influence area. ASA uses full custom layout, and can generate this layout for any characterized technology. However, placement and routing are significantly less efficient than manual layout. Also, ASA does not support dynamic circuitry; especially the static registers used in ASA are very large. Sea of Gates technology fixes transistor size (and even place), but the use of manual layout and dynamic circuits allows for many improvements.

For example, ASA lacks support for (dynamic) wired-or. This means that comparisons for decoding on sockets and in the compare units must be performed with (large, static) gates. Also, lock and exception requests must be gathered individually from the FUs and combined using gates. Using wired-or, only a single *or* line would be required.

Realization features in the current MOVE processor generator that require more area than necessary, and can be improved, are:

– Sockets for address registers in the instruction fetch unit and load–store units have data paths that are wider than necessary for many parameter configurations; they are of full word size now.

– If the configuration is such that the range of immediate values is much larger than the number of other source IDs, then decoding on sockets of FUs other than the immediate unit is inefficient. The comparisons for decoding are performed on all high bits of the ID, in stead of only the required part in this case.

– Even when the single guard option is selected, the full squash control busses and logic remain in place, though most of it is redundant in this case.

– The guard unit uses more latches than necessary to store compare unit selection information.

- There are many relatively small units rendered each into separate standard-cell layout. Routing area can be reduced significantly by combining several of these in one standard-cell layout. (Note that a full MOVE32INT processor does not fit into a single standard-cell.) Units that are usually small and would benefit from being placed together in standard-cells are register units, the valid unit, and the global network control.
- All buffering suggested by the AUTOBUFFER command, except that for tri-state outputs, has been added to the design. However, only the buffering necessary to minimize critical path delay needs to be added.
- It may be possible to use a smaller adder type for the PC incrementer.

The best opportunity at this point for improving area use is probably by gathering small units into single standard-cell layout parts. Another promising approach is the design of a register file FU.

8.2. Timing evaluation

Table 8.3 contains the projected SoG MOVE32INT clock frequency and the ASA 1.0 μ version clock frequency as determined with the timing analyzer. The clock frequency for the ASA 1.5 μ version could be determined reliably; a rough estimate can be found in the analysis of the results below. Note that the ASA figure is with single-cycle integer unit operation, compared to the two cycle operation used in the SoG MOVE32INT. Also, the SoG version does not support FU exceptions, unlike the ASA version. The single-cycle integer unit and the exception path do determine the first two critical paths in the ASA version, though.

Determination of the clock frequency has been problematic, due to the ASA timing analyzer limitation on the size of the circuit. The timing analyzer has been unable to complete a clock period analysis on a full MOVE32INT configuration with layout information. Analysis of the circuit without layout information, taking only gate delays and loads into account, is possible though[1].

Table 8.4 contains an overview of the first five major critical paths for the 1.0 μ version, obtained with gate level analysis. Table 8.5 contains similar data for the 1.5 μ version; note that there are minor differences in the global lock (secondary) critical path. An overview of the distribution of global control path delays is shown in figure 8.1. The delay figures there are rough (rounded to 0.5 ns) indications based on the 1.0 μ timing analysis. Path timing analysis details can be found in appendix B. The pipeline control of the shift unit shows up in these paths, because it has longer delays than other hybrid FUs due to the double result sockets. The control signals from these sockets have to be combined on the shift FU. The path descriptions in the tables do not contain detailed steps within hybrid FU control; hybrid FU pipeline control has already been presented in detail in 5.2.

The path through the integer unit adder dominates by 1.5 ns for the 1.0 μ version. The difference with the next potential critical path, the exception path, is small enough to choose for single-cycle delay integer units. Fortunately, the integer unit path is local to the unit, and its delay can be determined accurately without taking other parts of the processor into consideration. This means that the timing analyzer circuit size limitation can be circumvented by analyzing only part of the processor with full layout information. The result of this partial analysis has already been presented in table 8.3. The increase in critical path delay for layout versus gate level is about 20%

[1] Clock skew is included in the gate level analysis.

SoG	**ASA** 1.0 μ
80 MHz 12.5 ns	43 MHz 23.3 ns

Table 8.3. MOVE32INT clock frequency comparison, projected SoG versus analyzed ASA 1.0 μ figures

Path	**Delay**	$\frac{1}{delay}$
integer FU: opcode latch → AU → result register	19.5 ns	51.2 MHz
exception: compare FU result → guard unit → squash lines → shift FU pipeline exception logic → network control → ifetch FU PC select → cache RAM address select → cache RAM address inputs	18.0 ns	55.7 MHz
global lock: guard unit guard specifier latches → guard evaluation logic → squash lines → network control → global lock → shift FU *socket_cls* pipeline lock path	17.9 ns	55.9 MHz
squash: compare FU result → guard unit → squash lines → ifetch FU trigger socket → PC select → cache RAM address select → cache RAM address inputs	16.8 ns	59.4 MHz
ifetch FU PC increment: PC register → PC increment → PC select → PC register	13.9 ns	71.7 MHz

Table 8.4. ASA 1.0 μ MOVE32INT gate level critical path analysis

Path	**Delay**	$\frac{1}{delay}$
exception: compare FU result → guard unit → squash lines → shift FU pipeline exception logic → network control → ifetch FU PC select → cache RAM address select → cache RAM address inputs	27.7 ns	36.1 MHz
integer FU: opcode latch → AU → result register	27.3 ns	36.6 MHz
global lock: compare FU result → guard unit → squash lines → network control → global lock → shift FU *socket_cls* pipeline lock path	26.5 ns	37.7 MHz
squash: compare FU result → guard unit → squash lines → ifetch FU trigger socket → PC select → cache RAM address select → cache RAM address inputs	25.6 ns	39.0 MHz
ifetch FU PC increment: PC register → PC increment → PC select → PC register	19.5 ns	51.2 MHz

Table 8.5. ASA 1.5 μ MOVE32INT gate level critical path analysis

in this case.

It is to be expected that the secondary critical paths, which contain global control, suffer more from additional layout capacitance than the primary one, which is local to a functional unit. This expectation is supported by analysis results for a reduced processor; here, increase in delay for the

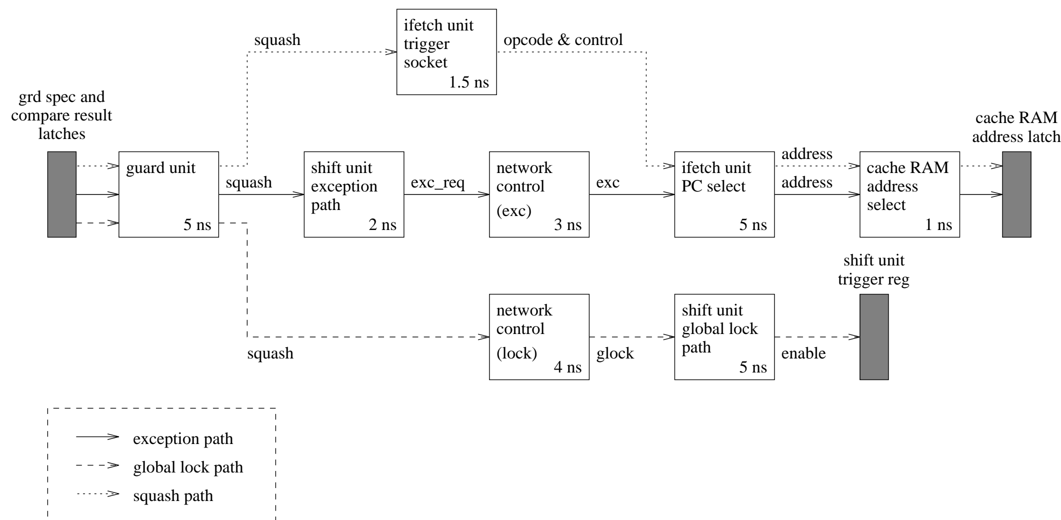


Figure 8.1. Approximate distribution of global control critical path delays (ASA 1.0 μ MOVE32INT)

exception path of about 35% was found. The actual delays of the global paths depend on buffer driving capacity for the global lines though, and accurate buffer capacity determination cannot be performed without detailed analysis of the complete system. As the latter is not possible with the ASA version 5.4 timing analyzer, and the actual exception path delay with appropriate buffering is expected to be about as long as the integer unit path delay, the integer unit path delay is assumed to represent the correct critical path.

The gate level analysis for the 1.5 μ version (Table 8.5) shows that the dominant critical path is a global one here. Using the estimate of 35% for additional global routing delay, an actual clock frequency of about 27 MHz may be expected. The basis for this estimate is too thin to present it as a result, though.

Suggestions for timing improvement are:

- Buffering is important for timing; to arrive at optimal buffer driving capacity, additional analysis of the loading on global paths is necessary, with layout information taken into account. Although version 5.4 of ASA cannot analyze the global paths accurately, it may be possible with other tools, by inspection of the layout, or by estimating it using extrapolation.

- Additional improvement of the global critical paths can possibly be obtained by improving the guard evaluation logic. It may be possible to do part of the calculation before the current latching point, and the current circuitry may not be optimal.

Notes

- The positions of latches in the guard unit (for compare unit select signals) and in the instruction fetch unit cache tag compare circuitry have been chosen such as to improve critical path delays. Also, the lock path has been implemented carefully to avoid unnecessary delays, both in hybrid FUs and the instruction fetch unit.

- When generating layout with standard cells at the leaf cell level, the clock input pad becomes overloaded. This because the BUFFER CLOCK command will place clock buffers only inside existing standard cells, and there are too many standard cells to drive them directly from a single clock pad output. The load on the clock pad can be relieved by applying a BUFFER SIGNAL command on the clock pad output *after* clock buffering. This way, an extra level of clock buffering is included.

- Improving area may improve timing also, by shortening the global control paths.

8.3. Scalability evaluation

The scalability of the current MOVE processor generator is limited primarily by the timing analyzer limitation on circuit size (see 8.2). The timing analyzer limitation is not a fundamental one, as processors of size beyond the timing analyzer limit can be generated. For the MOVE framework, which requires automatic generation and analysis iterations, it does present a serious problem though. This (arbitrary) limit of the timing analyzer should be removed by Sagantec. Alternatively, other timing analysis tools may be tried.

Although ASA's SID language has several limitations that hinder parametrization (see 6.3), these can be circumvented by means of preprocessing (6.4).

Other ASA limitations which should be considered, although of less importance, are the limits on standard cell layout unit size and the TOGATES command 255 signal limitation. Both these limitations can be avoided, if they occur, by means of reorganizations in the SID description. Note that Sagantec has announced before that the TOGATES 255 signal limitation would be removed, but that it has not yet happened.

All in all, the freedom of scalability with the current MOVE processor generator is quite good. A wide range of processors can be generated by simply setting parameters appropriately, and further modifications (e.g. pipelining method or new FUs) are possible by means of modifying a high-level SID description.

Chapter 9. Evaluation of architecture

This chapter contains suggestions for architectural improvements based on the results of the current MOVE processor generator. The first section contains comments that apply to MOVE architectures in general; the second section presents suggestions for improvement of the current MOVEASAINT architecture.

9.1. MOVE architectures in general

Below, suggestions are made for changes to hybrid pipeline control, FU and transport pipelining.

Hybrid pipeline control

Currently, the SoG MOVE32INT locks when an attempt is made to read an empty hybrid FU pipeline. In the MOVEASAINT an exception is generated in that situation. However, it may be an advantage to allow reading from a pipeline that is now considered 'empty' (all stages invalid). This allows results to be used more than once. Disadvantages of this approach are less strict checking, and exception state save and restore taking more time. Checking is less strict, because there is no strict coupling any more of writes and reads on a pipeline, and an uninitialized pipeline may be read (unless extra state is added to detect this condition). On exception the state of 'empty' pipelines has to be saved, and on return from exception these values have to be entered into the pipelines and read from it again to invalidate the result.

FU pipelining

The control required for hybrid FU pipelines is substantial (see 5.2). Because of the dependencies between stages, control chains are required that span the entire length of the pipeline. These FU pipeline control chains are part of the global lock and exception signal paths, and therefore limit the clock frequency that can be achieved.

To improve timing, a less sophisticated pipelining scheme may be used, at the penalty of losing scheduling freedom. Analysis suggests a performance loss of about 10% when going from hybrid to continue always pipelines. For a MOVEASAINT instance with MOVE32INT configuration in 1.0 μ technology and with gate level analysis (see 8.2 and appendix B), the hybrid pipelining overhead on the exception path (the longest global path) is about 2 ns. With layout information taken into account, the overhead is about 3 ns. The hybrid pipelining overhead on the global lock path is about 2.5 ns (gate level). These figures indicate an increase in global control path delay of about 13% (both for gate and layout level analysis). The gate level critical path analysis with the integer unit adder as the critical path suggests that the lock path would become the critical one by a small margin, with an increase of only 5% in critical delay. However, this gate level figure is probably not realistic, as the global paths suffer more from (long) wiring delays.

The current figures are only estimates, and have not been verified accurately. However, they suggest that the 10% performance advantage of hybrid pipelines almost balances the global control critical path overhead of 13%. Note that the integer unit critical path may dominate in many cases, though, which would tip the balance in favor of hybrid pipelining.

When moving to pipelining schemes that are simpler and faster than hybrid, it is not necessary to go all the way to continue always pipelines. The best solution may be between the extremes, such as FU pipelines that do synchronize with the transport network, but have simpler rules for stage control in the FUs.

Transport pipelining

Compared to the two stage transport pipeline used in the MOVEASAINT (decode in the cycle before the data transport, see figure 9.1), other pipelining schemes may provide performance advantages, depending on application requirements. One way to reduce latencies, at the expense of throughput, is by combining decode and FU operation (execute) in the first part of a clock cycle with data transport in the second part, see figure 9.2. This way FU pipeline latencies can be reduced, including instruction fetch unit branch delay. Note that when ASA is used, it may be possible to use the area-efficient RAM cell for a register unit with this kind of pipelining. Because of the synchronous nature of the ASA RAM cell, this would require early transport of the *move* source IDs to the register unit.

In the true single phase clocking environment imposed by ASA (see 7.1), there is a problem associated with the single cycle transport pipeline described above. To avoid bus conflicts during the decode part of the cycle, the tri-state bus buffers should remain disabled during decode. However, there is no timing reference available between latching points in a cycle. The buffer enable can of course be gated with the clock signal, but the assumption of the ASA timing analyzer that clock signals are symmetrical will cause problems for timing analysis with tuned clock ratios. Also, timing of the RAM cell in ASA depends on the clock ratio, which complicates timing analysis.

9.2. MOVEASAINT architecture

The current MOVEASAINT architecture has some obvious deficiencies which can be removed. Below, the problem areas are listed together with suggestions for improvements:

- Pipeline exceptions are not fully recoverable. This is because when a move operation initiates an exception, the operation is completed before exception processing is started. Pipeline (*move*) exceptions can be made precise by locking the processor when an exception occurs[1]. This increases the lock path delay though (by approximately the exception (*cls*) path delay through the hybrid FU with the deepest pipeline[2]), making it the critical path. Another way to make the exceptions recoverable is by keeping more state on all units, but this is expensive.

- Currently, the only supported FU exceptions are for pipeline errors. The FU implementations can be extended so that they signal data-dependent exceptions also (such as overflows). Support for precise exceptions may require complex extensions to the current FU versions, though.

- The current load–store unit provides for less pipelining than is possible, and does not cache data items. A completely new load–store unit will have to be designed to improve performance.

---

[1]The PC pipeline in the instruction fetch unit should *not* be locked on an exception; the processor would simply lock in that case, and not start exception processing.

[2]Currently about 2 ns through the shift unit for the 1.0 μ realization, obtained with gate level analysis.

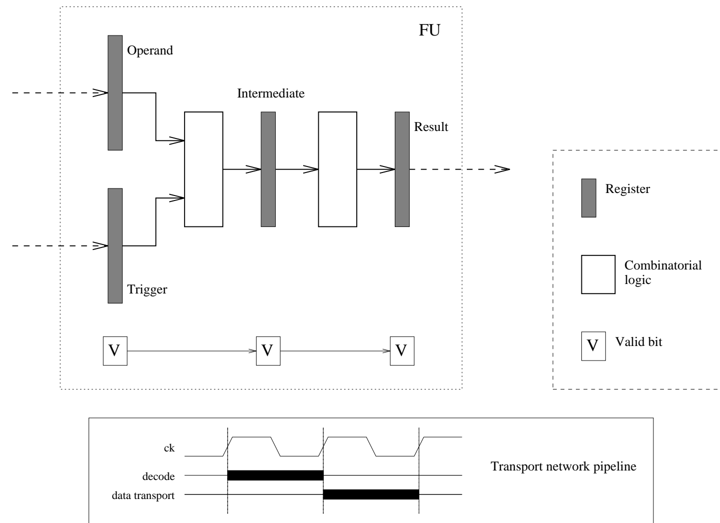


Figure 9.1. Two stage transport network with example FU configuration

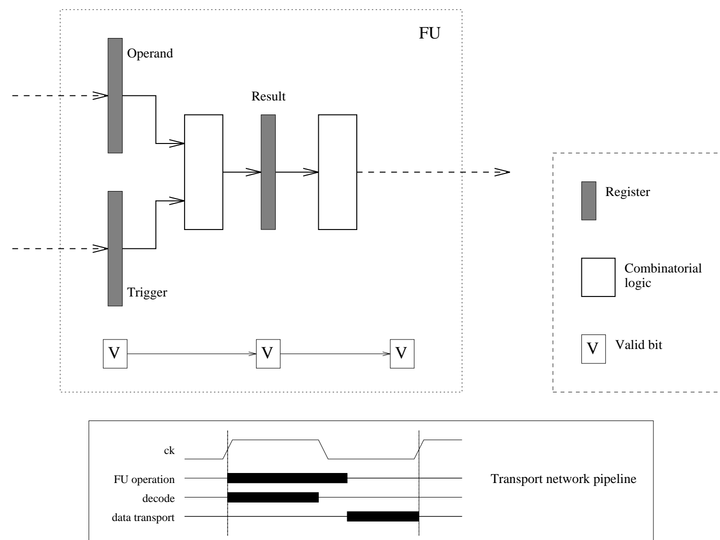


Figure 9.2. Single stage transport network with example FU configuration

– General purpose registers are currently placed each in separate FUs, and are implemented with large ASA standard cell registers. The area use for these registers is large enough to consider using the much more efficient RAM real cells for registers. However, reading a RAM cell is slow, which will result in either a pipelined register FU or the use of a register fetch pipeline stage in the instruction / data transport pipeline. Alternatively, a slower transport pipeline may be used where the register values are fetched in the first part of the combined execute–*move* stage.

Chapter 10. Conclusion

A MOVE processor generator based on the ASA silicon compiler has been successfully constructed. A system has been built, based on ASA's SID hardware description language augmented by preprocessing, that can generate a wide range of MOVE processors. The requirements with respect to architecture (MOVE architecture, compatible with MOVE32INT for comparison) and parametrization (general scalability) have been met.

The MOVE processor generator consists of 15,000 lines of commented source. Only the parameter declaration file needs to be edited to change the processor configuration. Directions for adding new functional unit types to the processor generator are included; because of the high level of description the MOVE processor generator is quite easily modified.

Major problems encountered during implementation the processor generator have been limitations of, and bugs in, the SID language compiler. These problems account for significant part of the implementation time of the processor generator. As part of the implementation, a preprocessing system has been created to cover the blind spots of SID.

The major limitation at this point to scalability is the ASA timing analyzer limit on circuit size. The timing analyzer limit prevents quick and accurate analysis of processors of reasonable size at this moment.

For evaluation of the system, a processor similar to the 32 bits, 4 parallel *move,* Sea of Gates (SoG) MOVE32INT processor has been generated and characterized. The SoG MOVE32INT, currently under construction at the Computer Architecture Department, is a 1.6 µ CMOS gate array realization with 1 cm^2 die area and projected 80 MHz clock frequency. The processor generated with ASA for a 1.0 µ CMOS process requires 1 cm^2 die area and runs at 43 MHz. One significant difference between the two processors is that the ASA version is configured with single cycle delay integer units at the specified frequency, while the SoG version requires two cycles delay for its integer units.

The best opportunity at this point for reducing die area use is probably by gathering small units into single so-called 'standard cell' layout parts. Timing may be improved by determining optimal buffer driving capacity on critical paths. The ASA version 5.4 timing analyzer is unable to perform the required analysis accurately.

The MOVE architecture, as defined before work on the processor generator started, appears to be well designed when considering the ASA realization. The increase in critical path delay caused by the complex hybrid pipeline control is in balance with the scheduling performance gains associated with this pipelining scheme. The two stage transport network pipeline maps well on the single phase clocking scheme imposed by ASA. A single stage transport network may be more appropriate for certain scalar applications though.

References

1. P. C. M. van der Arend, *Processor generation with the ASA silicon compiler*. Master Thesis, Delft University of Technology, EE Faculty, the Netherlands, 1991.

2. Paul van der Arend, *Implementation and Realization of MOVE32INT*. Technical Report in preparation, Delft University of Technology, EE Faculty, the Netherlands, 1993.

3. Henk Corporaal and Paul van der Arend, *MOVE32INT, a Sea of Gates realization of a high performance Transport Triggered Architecture*. Technical Report, Delft University of Technology, EE Faculty, the Netherlands, 1993.

4. Henk Corporaal, Jan Hoogerbrugge, Paul van der Arend and Paul Stravers, *MOVE framework, status: description and discussion document*. Internal Report, Delft University of Technology, EE Faculty, Computer Architecture Section, the Netherlands, January 1993.

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

6. Henk Corporaal, *Explicit Pipelined Architectures, What exists beyond MOVE?*. Internal Report, Delft University of Technology, EE Faculty, the Netherlands, 1992.

7. Henk Corporaal, *MOVE32INT Architecture and Programmer's Reference Manual*. Delft University of Technology, EE Faculty, the Netherlands, 1993.

8. Richard Fairley, *Software Engineering Concepts*. McGraw-Hill, 1985.

9. Joseph A. Fisher, Trace Scheduling: A Technique for Global Microcode Compaction. *IEEE Transactions on Computers, Vol C-30, No 7*, pages 478–490, July 1981.

10. A. J. van de Goor, *Computer Architecture and Design*. Addison–Wesley, 1989.

11. A. J. van de Goor, *DIGITAL SYSTEM TESTING, Hardware, Software and System requirements*. Delft University of Technology, 1989.

12. John L. Hennessy and David A. Patterson, *Computer Architecture A Quantitative Approach*. Morgan Kaufmann Publishers, 1990.

13. Junien Labrousse and Gerrit A. Slavenburg, CREATE-LIFE: A Modular Design Approach for High Performance ASIC's. *COMPCON '90 San Fransisco*, 1990.

14. Monica Lam, Software Pipelining: An Effective Scheduling Technique for VLIW Machines. *SIGPLAN '88 Atlanta, Georgia*, 1988.

15. B. 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.

16. Sagantec, *ASA version 5.4 manuals*, Eindhoven, the Netherlands.

Appendix A. Parameters

An example parameter declaration file for the MOVE processor generator (file *parmdecl.s32)* can be found on the following pages. The parameters are set to MOVE32INT compatible values, which provide for good example settings. These settings are the ones used for evaluation and comparison with the Sea of Gates MOVE32INT. A few of the parameters affect realization rather than architecture. The syntax used in the file is that of cpp (C preprocessor); it consists for a large part of commentary text though.

More information relating to some of these parameters can be found in the chapter describing the functional units (chapter 4).

Note that the integer units are set to single-cycle operation, in stead of the sub-pipelined two cycle operation chosen in the Sea of Gates MOVE32INT.

```
/*
------------------------------------------------------------------------------
  Parameter Declaration
  standard 32 bit version

  The parameters below define the processor characteristics. Unless you want
  to add functionality, only this file needs to be changed to customize
  the processor. If you want to generate layout, you may want to use a pad
  file (see movepaf.*).
------------------------------------------------------------------------------
*/

/*
   Connection defines

   Most units have sockets connecting to move busses. For each socket it
   can be specified to which of the busses it will connect. This is done
   by defines such as the following:

     #define CINTFU1_TBUS0 0
     #define CINTFU1_TBUS1 0
     #define CINTFU1_TBUS2 1
     #define CINTFU1_TBUS3 1

   In this example, the trigger socket connections for the second
   integer FU are defined, for a processor with PARMOVES == 4. The
   socket will connect to busses 2 and 3, but not to 0 and 1.

   For other FUs and sockets, the parameter naming is similar, see
   further below in this file. The fields in the identifiers are as
   follows:
       CINTFU1_TBUS2
       |\   /|||\ /|
       | \ / ||| V |
       |  V  ||| | Bus number
       |  |  ||| |
       |  |  ||| Always 'BUS'
       |  |  |||
       |  |  ||Socket selector: T->trigger, O->operand, R->result
       |  |  ||selectors can have subscripts (e.g. 'R2')
       |  |  ||
       |  |  |Always '_'
       |  |  |
       |  |  FU number, starting at 0
       |  |
       |  FU type
       |
       Always 'C' (for Connect)
*/

/*----------------------------------------------------------------------------
  Realization Level Switches
----------------------------------------------------------------------------*/

/*  FUNC: functional SID if defined, synthesizable SID if undefined.
*/
#undef FUNC

/*  SIMUCK: include explicit connections for clock and reset if defined.
    Must be defined currently.
*/
#define SIMUCK

/*  DEBUG: use debugging code when defined. For use during development
    and testing only.
*/
#undef DEBUG

/*  TIMEHINTS: include hints at potential critical paths for timing
    driven technology mapping, if defined.
```

```
*/
#define TIMEHINTS
/*  STDC_FU: generate standardcell layout at the FU level if defined.
    STDC_GLOB: generate standardcell layout at the global interconnect
    level if defined.
    If neither of the above are defined, then each leaf system will be
    generated as a separate standardcell. When STDC_GLOB is defined it
    makes no difference if STDC_FU is defined or not.
*/
#define STDC_FU
#undef STDC_GLOB
#define XPOWERPADS 11  /* number of extra power pads */
/*----------------------------------------------------------------------------
  Primary Processor Characteristics
----------------------------------------------------------------------------*/
/*  BUSWIDTH: defines the data (move-) bus width, and implies maximum address
    pointer size. It does not define the number of external address bits.
    Must be >= 3, because of the shift unit implementation.
*/
#define BUSWIDTH 32
/*  PARMOVES: number of move busses. Equals the number of move operations
    in an instruction in this implementation.
*/
#define PARMOVES 4
/*----------------------------------------------------------------------------
  Ifetch FU
----------------------------------------------------------------------------*/
/*  IFESHIFTADDR: if defined, instruction address registers in the
    ifetch unit are connected to the busses such that they map directly
    to data addresses; the lower bits are ignored on writes and return
    zero on read. If undefined, the lowest instruction address bit
    connects to bit 0 of the bus.
    Instructions are assumed to be aligned to data word boundaries in
    external memory, with no more than 1 instruction stored per data
    word.
    Note that using shifted address registers reduces the number of
    immediate bits than can be used to specify an address (offset)
    directly. This may also influence the exception handler.
*/
#undef IFESHIFTADDR
/*  IFEADDRSIZE: instruction address size. Instruction space is
    limited by BUSWIDTH for address register size if IFESHIFTADDR
    is undefined. If IFESHIFTADDR is defined, maximum instruction
    address size is reduced by the number of bits shifted.
    You can leave IFEADDRSIZE undefined if IFEAUTOADDRSIZE is defined.
*/
#undef IFEADDRSIZE
/*  IFEAUTOADDRSIZE: if defined, automatically choose IFEADDRSIZE to
    get instruction and data spaces of the same size; you can set
    IFEADDRSIZE yourself if IFEAUTOADDRSIZE is undefined. Automatic
    IFEADDRSIZE determination assumes no more than 1 instruction is
    stored per data word (BUSWIDTH); an instruction may take several
    data words in memory though. Alignment of instructions to word
    boundaries is taken into account.
    Works independently from IFESHIFTADDR.
*/
```

```
#define IFEAUTOADDRSIZE
/*  IFEOFFSETSIZE: number of 'pco' offset bits. Note that this parameter
    determines the number of offset bits actually used in an instruction
    address, so if the lower address bits are unused (see IFESHIFTADDR),
    the offset bits are shifted to the left. Therefore you may want to
    set this value to less than the number of immediate bits.

    Requirement: IFEOFFSETSIZE < IFEADDRSIZE.
*/
#define IFEOFFSETSIZE 6
#define IFEFUADDTYPE 15             /* ASA s_addern type for PC incr. */
/*  IFEFUDELAY: number of cycles for instruction fetch from external
    memory (strobe and latch cycles inclusive). Must be >= 2.
*/
#define IFEFUDELAY 2
#define IFERSTVEC "040"h            /* reset vector */
#define IFEEXCVEC "080"h            /* exception vector */
#define IFEINTVEC "0C0"h            /* interrupt vector */
/*  IFECACHESIZE: number of cache lines, one instruction per line. Must
    be a power of two, 4 <= IFECACHESIZE <= 2048. The maximum allowed
    cache size is more than 64 kbyte (including tag). The number of
    cache lines may be at most half the number of instruction addresses.

    For the current cache test code in movesi.sic, the cache size must
    be 4 lines.
*/
#define IFECACHESIZE 32
/*  Connection defines
*/
#define CIFEFU0_TBUS0 1
#define CIFEFU0_TBUS1 0
#define CIFEFU0_TBUS2 1
#define CIFEFU0_TBUS3 0
#define CIFEFU0_R1BUS0 1
#define CIFEFU0_R1BUS1 1
#define CIFEFU0_R1BUS2 0
#define CIFEFU0_R1BUS3 0
#define CIFEFU0_R2BUS0 1
#define CIFEFU0_R2BUS1 1
#define CIFEFU0_R2BUS2 0
#define CIFEFU0_R2BUS3 0
/*-----------------------------------------------------------------------------
  Guard Unit
-----------------------------------------------------------------------------*/
/*  SINGLEGUARD: only one guard signal is used if defined, capable of
    squashing the entire instruction. If undefined, each move bus is
    guarded individually.
*/
#undef SINGLEGUARD
/*  GUARDSPECSIZE: defines the number of bits per guard specifier in
    an instruction. Must be 3 for now.
*/
#define GUARDSPECSIZE 3
/*-----------------------------------------------------------------------------
  Immediate Unit
-----------------------------------------------------------------------------*/
/*  IMMFUS: number of immediate FUs. Using more than one is useless,
    but possible.
```

```
*/
#define IMMFUS 1
/*  IMMEDIATES: number of immediate values. Defines the range of
    immediates as 0 .. IMMEDIATES - 1 (assuming IMMSIGNEXTEND
    undefined). IMMEDIATES must be a power of two.
*/
#define IMMEDIATES 64
#undef IMMSIGNEXTEND                /* not implemented */
/*  Connection defines
    Unlike most other units, here each bus connection defines connecting
    a different result register to the bus (immediates are independent
    for each bus).
*/
#define CIMMFU0_RBUS0 1
#define CIMMFU0_RBUS1 1
#define CIMMFU0_RBUS2 1
#define CIMMFU0_RBUS3 1

/*-----------------------------------------------------------------------------
  Load-Store Units
----------------------------------------------------------------------------*/
#define LDSFUS 1                    /* number of load-store FUs */
/*  LDSFUDELAY: number of cycles for data load or store operation
    (strobe and latch cycles inclusive). Must be >= 1; address
    strobe and data latch may overlap.
*/
#define LDSFUDELAY 2
/*  LDSADDRSIZE: data address size. Data space is limited by BUSWIDTH
    which determines maximum address register size.
*/
#define LDSADDRSIZE 15
/*  LDSOLOCKFULL: full operand socket (store data) lock implementation
    if defined: lock only if store in progress. If undefined, lock
    always when memory operation in progress. In the latter case the
    lock path may be shorter, but a write to the O register while a
    load is in progress will lock the processor unnecessarily.
*/
#define LDSOLOCKFULL
/*  Connection defines
*/
#define CLDSFU0_TBUS0 1
#define CLDSFU0_TBUS1 0
#define CLDSFU0_TBUS2 0
#define CLDSFU0_TBUS3 1

#define CLDSFU0_OBUS0 0
#define CLDSFU0_OBUS1 1
#define CLDSFU0_OBUS2 1
#define CLDSFU0_OBUS3 0

#define CLDSFU0_RBUS0 1
#define CLDSFU0_RBUS1 0
#define CLDSFU0_RBUS2 0
#define CLDSFU0_RBUS3 1

/*-----------------------------------------------------------------------------
  Compare Units
----------------------------------------------------------------------------*/
#define CMPFUS 2                    /* number of compare FUs, must be 2 now */
/*  Connection defines
```

```
*/
#define CCMPFU0_TBUS0 1
#define CCMPFU0_TBUS1 1
#define CCMPFU0_TBUS2 0
#define CCMPFU0_TBUS3 0

#define CCMPFU0_OBUS0 0
#define CCMPFU0_OBUS1 1
#define CCMPFU0_OBUS2 1
#define CCMPFU0_OBUS3 0

#define CCMPFU0_RBUS0 1  /* not in MOVE32INT spec */
#define CCMPFU0_RBUS1 0
#define CCMPFU0_RBUS2 0
#define CCMPFU0_RBUS3 1

#define CCMPFU1_TBUS0 0
#define CCMPFU1_TBUS1 0
#define CCMPFU1_TBUS2 1
#define CCMPFU1_TBUS3 1

#define CCMPFU1_OBUS0 1
#define CCMPFU1_OBUS1 0
#define CCMPFU1_OBUS2 0
#define CCMPFU1_OBUS3 1

#define CCMPFU1_RBUS0 0  /* not in MOVE32INT spec */
#define CCMPFU1_RBUS1 1
#define CCMPFU1_RBUS2 1
#define CCMPFU1_RBUS3 0

/*------------------------------------------------------------------------------
  Integer Units
------------------------------------------------------------------------------*/
#define INTFUS 2                      /* number of integer FUs */

/*  INTFUDELAY: determines the number of cycles the stage control
    allows for execution of the operation.
*/
#define INTFUDELAY 1
#define INTFUADDTYPE 15               /* ASA s_addern type to use */

/*  Connection defines
*/
#define CINTFU0_TBUS0 1
#define CINTFU0_TBUS1 1
#define CINTFU0_TBUS2 0
#define CINTFU0_TBUS3 0

#define CINTFU0_OBUS0 1
#define CINTFU0_OBUS1 0
#define CINTFU0_OBUS2 0
#define CINTFU0_OBUS3 1

#define CINTFU0_RBUS0 1
#define CINTFU0_RBUS1 0
#define CINTFU0_RBUS2 1
#define CINTFU0_RBUS3 0

#define CINTFU1_TBUS0 0
#define CINTFU1_TBUS1 0
#define CINTFU1_TBUS2 1
#define CINTFU1_TBUS3 1

#define CINTFU1_OBUS0 0
#define CINTFU1_OBUS1 1
```

```
#define CINTFU1_OBUS2 1
#define CINTFU1_OBUS3 0
#define CINTFU1_RBUS0 0
#define CINTFU1_RBUS1 1
#define CINTFU1_RBUS2 0
#define CINTFU1_RBUS3 1
/*---------------------------------------------------------------------------
  Logic Units
---------------------------------------------------------------------------*/
#define LOGFUS 1                    /* number of logic units */
/*  Connection defines
*/
#define CLOGFU0_TBUS0 0
#define CLOGFU0_TBUS1 1
#define CLOGFU0_TBUS2 0
#define CLOGFU0_TBUS3 1
#define CLOGFU0_OBUS0 1
#define CLOGFU0_OBUS1 0
#define CLOGFU0_OBUS2 1
#define CLOGFU0_OBUS3 0
#define CLOGFU0_RBUS0 0
#define CLOGFU0_RBUS1 0
#define CLOGFU0_RBUS2 1
#define CLOGFU0_RBUS3 1
/*---------------------------------------------------------------------------
  Shift Units
---------------------------------------------------------------------------*/
#define SHIFUS 1                    /* number of shift units */
/*  Connection defines
*/
#define CSHIFU0_TBUS0 0
#define CSHIFU0_TBUS1 1
#define CSHIFU0_TBUS2 0
#define CSHIFU0_TBUS3 1
#define CSHIFU0_R1BUS0 1
#define CSHIFU0_R1BUS1 0
#define CSHIFU0_R1BUS2 1
#define CSHIFU0_R1BUS3 0
#define CSHIFU0_R2BUS0 0
#define CSHIFU0_R2BUS1 1
#define CSHIFU0_R2BUS2 0
#define CSHIFU0_R2BUS3 1
/*---------------------------------------------------------------------------
  Valid Unit
---------------------------------------------------------------------------*/
/*  Connection defines
    Each valid bit can appear on only one bus. Only as many connects
    need be defined as the number of valid bits vs. BUSWIDTH requires.

    The valid bits are assigned from the highest bus number and the
    most significant bit downwards.
*/
#define CVALFU_RBUS0 0
#define CVALFU_RBUS1 0
#define CVALFU_RBUS2 0
```

```
#define CVALFU_RBUS3 1

/*---------------------------------------------------------------------------
  Register Units
---------------------------------------------------------------------------*/
#define REGFUS 10                    /* number of register units */

/*  Connection defines
*/
#define CREGFU0_TBUS0 0
#define CREGFU0_TBUS1 0
#define CREGFU0_TBUS2 1
#define CREGFU0_TBUS3 1

#define CREGFU0_RBUS0 1
#define CREGFU0_RBUS1 1
#define CREGFU0_RBUS2 0
#define CREGFU0_RBUS3 0

#define CREGFU1_TBUS0 0
#define CREGFU1_TBUS1 0
#define CREGFU1_TBUS2 1
#define CREGFU1_TBUS3 1

#define CREGFU1_RBUS0 1
#define CREGFU1_RBUS1 1
#define CREGFU1_RBUS2 0
#define CREGFU1_RBUS3 0

#define CREGFU2_TBUS0 1
#define CREGFU2_TBUS1 0
#define CREGFU2_TBUS2 1
#define CREGFU2_TBUS3 0

#define CREGFU2_RBUS0 0
#define CREGFU2_RBUS1 1
#define CREGFU2_RBUS2 0
#define CREGFU2_RBUS3 1

#define CREGFU3_TBUS0 1
#define CREGFU3_TBUS1 0
#define CREGFU3_TBUS2 1
#define CREGFU3_TBUS3 0

#define CREGFU3_RBUS0 0
#define CREGFU3_RBUS1 1
#define CREGFU3_RBUS2 0
#define CREGFU3_RBUS3 1

#define CREGFU4_TBUS0 0
#define CREGFU4_TBUS1 1
#define CREGFU4_TBUS2 0
#define CREGFU4_TBUS3 1

#define CREGFU4_RBUS0 1
#define CREGFU4_RBUS1 0
#define CREGFU4_RBUS2 1
#define CREGFU4_RBUS3 0

#define CREGFU5_TBUS0 0
#define CREGFU5_TBUS1 1
#define CREGFU5_TBUS2 0
#define CREGFU5_TBUS3 1

#define CREGFU5_RBUS0 1
#define CREGFU5_RBUS1 0
#define CREGFU5_RBUS2 1
```

```
#define CREGFU5_RBUS3 0
#define CREGFU6_TBUS0 1
#define CREGFU6_TBUS1 0
#define CREGFU6_TBUS2 0
#define CREGFU6_TBUS3 1
#define CREGFU6_RBUS0 0
#define CREGFU6_RBUS1 1
#define CREGFU6_RBUS2 1
#define CREGFU6_RBUS3 0
#define CREGFU7_TBUS0 1
#define CREGFU7_TBUS1 0
#define CREGFU7_TBUS2 0
#define CREGFU7_TBUS3 1
#define CREGFU7_RBUS0 0
#define CREGFU7_RBUS1 1
#define CREGFU7_RBUS2 1
#define CREGFU7_RBUS3 0
#define CREGFU8_TBUS0 0
#define CREGFU8_TBUS1 1
#define CREGFU8_TBUS2 1
#define CREGFU8_TBUS3 0
#define CREGFU8_RBUS0 1
#define CREGFU8_RBUS1 0
#define CREGFU8_RBUS2 0
#define CREGFU8_RBUS3 1
#define CREGFU9_TBUS0 0
#define CREGFU9_TBUS1 1
#define CREGFU9_TBUS2 1
#define CREGFU9_TBUS3 0
#define CREGFU9_RBUS0 1
#define CREGFU9_RBUS1 0
#define CREGFU9_RBUS2 0
#define CREGFU9_RBUS3 1
/*-----------------------------------------------------------------------------
  Scan Register Units
-----------------------------------------------------------------------------*/
#define SCRFUS 1                    /* number of scan register units */
/*  Connection defines
*/
#define CSCRFU0_TBUS0 1
#define CSCRFU0_TBUS1 1
#define CSCRFU0_TBUS2 0
#define CSCRFU0_TBUS3 0
#define CSCRFU0_RBUS0 0
#define CSCRFU0_RBUS1 0
#define CSCRFU0_RBUS2 1
#define CSCRFU0_RBUS3 1
```

Appendix B. Timing details

This appendix contains timing details of paths used in the critical path analysis of chapter 8. The paths presented below are for the MOVE32INT compatible configuration of the MOVEASAINT architecture (see appendix A). They represent the first five major critical paths for the 1.0 μ ES2/CMS1M0 technology realization. The results are obtained using the ASA timing analyzer, with analysis on gate level (without wiring delays). For interpretation of the paths, refer to section 8.2.

Integer FU path

```
- clock period results -
 nr    period       path   ck_skew     setup    stable
  1     19534      20826      3561      2268         0
  setup node : MOVEASAINT.GLOBIC.INTFU1.INTFUI.RREG.REG.FF2[28].D[1]
entry 1 :
1...2_________3.......4 5.....6 7.....8___9...............10__11..12 13
......14____15..............16__17
......................................................................
...............................................18 19
. = 100 ps., cell delay
- = 100 ps., interconnect delay
  1   (      0,  RISING )  MOVEASAINT.CK.BOND
  2   (    390,  RISING )  MOVEASAINT.CK.Y
  3   (   1370,  RISING )  MOVEASAINT.GLOBIC.VALFU.S_CLOCKBUF2.I
  4   (   2116,  RISING )  MOVEASAINT.GLOBIC.VALFU.S_CLOCKBUF2.Y
  5   (   2140,  RISING )  MOVEASAINT.GLOBIC.VALFU.S_CLOCKBUF1.I
  6   (   2650,  RISING )  MOVEASAINT.GLOBIC.VALFU.S_CLOCKBUF1.Y
  7   (   2717,  RISING )  MOVEASAINT.GLOBIC.SCRFU0.S_CLOCKBUF.I
  8   (   3238,  RISING )  MOVEASAINT.GLOBIC.SCRFU0.S_CLOCKBUF.Y
  9   (   3561,  RISING )  MOVEASAINT.GLOBIC.INTFU1.INTFUI.OPCREG.CK
 10   (   5069, FALLING )  MOVEASAINT.GLOBIC.INTFU1.INTFUI.OPCREG.Q[0]
 11   (   5301, FALLING )  MOVEASAINT.GLOBIC.INTFU1.INTFUI.INTFUC.S_INV2[0].I
 12   (   5540,  RISING )  MOVEASAINT.GLOBIC.INTFU1.INTFUI.INTFUC.S_INV2[0].Y
 13   (   5577,  RISING )  MOVEASAINT.GLOBIC.INTFU1.INTFUI.INTFUC.S_BUFN.I
 14   (   6237,  RISING )  MOVEASAINT.GLOBIC.INTFU1.INTFUI.INTFUC.S_BUFN.Y
 15   (   6704,  RISING )  MOVEASAINT.GLOBIC.INTFU1.INTFUI.INTFUC.S_XNOR[21].I[1]
 16   (   8196, FALLING )  MOVEASAINT.GLOBIC.INTFU1.INTFUI.INTFUC.S_XNOR[21].Y
 17   (   8467, FALLING )  MOVEASAINT.GLOBIC.INTFU1.INTFUI.INTFUC.ADDERN.B[10]
 18   (  20766,  RISING )  MOVEASAINT.GLOBIC.INTFU1.INTFUI.INTFUC.ADDERN.S[28]
 19   (  20826,  RISING )  MOVEASAINT.GLOBIC.INTFU1.INTFUI.RREG.REG.FF2[28].D[1]
```

Exception path

```
________
- clock period results -
________
 nr    period      path   ck_skew     setup    stable
_________________-
  1     17967     19665      3561      1862         0
  setup node : MOVEASAINT.GLOBIC.ICCHREMRAM.RBL.A[0]
entry 1 :
1...2_________3.......4 5.....6 7.....8___9..............10_11......12
___13....14_____15......16_17.....18 19....20____21........22__23....24__25
.......26 27..28 29........30__31......32________33......34_35....36____37.....38
__39..40__41.....42_43......44_45...46 47
. = 100 ps., cell delay
- = 100 ps., interconnect delay
  1   (      0,  RISING )  MOVEASAINT.CK.BOND
  2   (    390,  RISING )  MOVEASAINT.CK.Y
  3   (   1370,  RISING )  MOVEASAINT.GLOBIC.VALFU.S_CLOCKBUF2.I
  4   (   2116,  RISING )  MOVEASAINT.GLOBIC.VALFU.S_CLOCKBUF2.Y
  5   (   2140,  RISING )  MOVEASAINT.GLOBIC.VALFU.S_CLOCKBUF1.I
  6   (   2650,  RISING )  MOVEASAINT.GLOBIC.VALFU.S_CLOCKBUF1.Y
  7   (   2717,  RISING )  MOVEASAINT.GLOBIC.SCRFU0.S_CLOCKBUF.I
  8   (   3238,  RISING )  MOVEASAINT.GLOBIC.SCRFU0.S_CLOCKBUF.Y
  9   (   3561,  RISING )  MOVEASAINT.GLOBIC.CMPFU1.CMPFUI.RREG.CK
 10   (   5069, FALLING )  MOVEASAINT.GLOBIC.CMPFU1.CMPFUI.RREG.Q[0]
 11   (   5198, FALLING )  MOVEASAINT.GLOBIC.CMPFU1.S_BUFN1.I
 12   (   5832, FALLING )  MOVEASAINT.GLOBIC.CMPFU1.S_BUFN1.Y
 13   (   6135, FALLING )  MOVEASAINT.GLOBIC.GRDFU.S_NOR2[4].I[0]
 14   (   6536,  RISING )  MOVEASAINT.GLOBIC.GRDFU.S_NOR2[4].Y
 15   (   7050,  RISING )  MOVEASAINT.GLOBIC.GRDFU.S_ANDNOR22[7].I[2]
 16   (   7727, FALLING )  MOVEASAINT.GLOBIC.GRDFU.S_ANDNOR22[7].Y
 17   (   7847, FALLING )  MOVEASAINT.GLOBIC.GRDFU.S_ANDNOR21[4].I[0]
 18   (   8351,  RISING )  MOVEASAINT.GLOBIC.GRDFU.S_ANDNOR21[4].Y
 19   (   8448,  RISING )  MOVEASAINT.GLOBIC.GRDFU.S_ORNAND21[0].I[1]
 20   (   8856, FALLING )  MOVEASAINT.GLOBIC.GRDFU.S_ORNAND21[0].Y
 21   (   9331, FALLING )  MOVEASAINT.GLOBIC.GRDFU.S_BUFN3.I
 22   (  10150, FALLING )  MOVEASAINT.GLOBIC.GRDFU.S_BUFN3.Y
 23   (  10389, FALLING )  MOVEASAINT.GLOBIC.SHIFU0.R2SOCK.S_ORNAND22[0].I[3]
 24   (  10860,  RISING )  MOVEASAINT.GLOBIC.SHIFU0.R2SOCK.S_ORNAND22[0].Y
 25   (  11107,  RISING )  MOVEASAINT.GLOBIC.SHIFU0.SHIFUI.SHIFUC.S_OR2[0].I[1]
 26   (  11857,  RISING )  MOVEASAINT.GLOBIC.SHIFU0.SHIFUI.SHIFUC.S_OR2[0].Y
 27   (  11923,  RISING )  MOVEASAINT.GLOBIC.SHIFU0.TSOCK.S_NOR3[0].I[2]
 28   (  12222, FALLING )  MOVEASAINT.GLOBIC.SHIFU0.TSOCK.S_NOR3[0].Y
 29   (  12306, FALLING )  MOVEASAINT.GLOBIC.NETCONTROL.S_NOR4[1].I[1]
 30   (  13137,  RISING )  MOVEASAINT.GLOBIC.NETCONTROL.S_NOR4[1].Y
 31   (  13384,  RISING )  MOVEASAINT.GLOBIC.NETCONTROL.S_NAND4[0].I[3]
 32   (  14027, FALLING )  MOVEASAINT.GLOBIC.NETCONTROL.S_NAND4[0].Y
 33   (  14836, FALLING )  MOVEASAINT.GLOBIC.IFEFU0.IFEFUI.PCSEL.S_NOR2[6].I[1]
 34   (  15513,  RISING )  MOVEASAINT.GLOBIC.IFEFU0.IFEFUI.PCSEL.S_NOR2[6].Y
 35   (  15613,  RISING )  MOVEASAINT.GLOBIC.IFEFU0.IFEFUI.PCSEL.S_ORNAND21[0].I[0]
 36   (  16016, FALLING )  MOVEASAINT.GLOBIC.IFEFU0.IFEFUI.PCSEL.S_ORNAND21[0].Y
 37   (  16488, FALLING )  MOVEASAINT.GLOBIC.IFEFU0.IFEFUI.PCSEL.S_NOR2[4].I[1]
 38   (  17007,  RISING )  MOVEASAINT.GLOBIC.IFEFU0.IFEFUI.PCSEL.S_NOR2[4].Y
 39   (  17285,  RISING )  MOVEASAINT.GLOBIC.IFEFU0.IFEFUI.PCSEL.S_INV2[26].I
 40   (  17516, FALLING )  MOVEASAINT.GLOBIC.IFEFU0.IFEFUI.PCSEL.S_INV2[26].Y
 41   (  17773, FALLING )  MOVEASAINT.GLOBIC.IFEFU0.IFEFUI.PCSEL.S_ORNAND22[11].I[2]
 42   (  18320,  RISING )  MOVEASAINT.GLOBIC.IFEFU0.IFEFUI.PCSEL.S_ORNAND22[11].Y
 43   (  18504,  RISING )  MOVEASAINT.GLOBIC.IFEFU0.IFEFUI.CCHPTH.S_ANDNOR22[4].I[3]
 44   (  19146, FALLING )  MOVEASAINT.GLOBIC.IFEFU0.IFEFUI.CCHPTH.S_ANDNOR22[4].Y
 45   (  19268, FALLING )  MOVEASAINT.GLOBIC.IFEFU0.IFEFUI.CCHPTH.S_NAND2[9].I[0]
 46   (  19631,  RISING )  MOVEASAINT.GLOBIC.IFEFU0.IFEFUI.CCHPTH.S_NAND2[9].Y
 47   (  19665,  RISING )  MOVEASAINT.GLOBIC.ICCHREMRAM.RBL.A[0]
```

Global lock path

```
________
- clock period results -
________
 nr    period       path   ck_skew     setup    stable
__________________
  1     17897     19225      3561      2232         0
  setup node : MOVEASAINT.GLOBIC.SHIFU0.SHIFUI.TREG.REG.FF2[0].S[0]
entry 1 :
1...2_________3.......4 5.....6 7.....8___9.....10_11..............12
___13..14_15...16_17....18 19....20 21....22_____23.......24__25....26_27
........28__29...30__________31.........32____33..34_35...36 37......38_39..40
_41........42 43......44____45........46
. = 100 ps., cell delay
- = 100 ps., interconnect delay
  1   (      0,  RISING )  MOVEASAINT.CK.BOND
  2   (    390,  RISING )  MOVEASAINT.CK.Y
  3   (   1370,  RISING )  MOVEASAINT.GLOBIC.VALFU.S_CLOCKBUF2.I
  4   (   2116,  RISING )  MOVEASAINT.GLOBIC.VALFU.S_CLOCKBUF2.Y
  5   (   2140,  RISING )  MOVEASAINT.GLOBIC.VALFU.S_CLOCKBUF1.I
  6   (   2650,  RISING )  MOVEASAINT.GLOBIC.VALFU.S_CLOCKBUF1.Y
  7   (   2717,  RISING )  MOVEASAINT.GLOBIC.SCRFU0.S_CLOCKBUF.I
  8   (   3238,  RISING )  MOVEASAINT.GLOBIC.SCRFU0.S_CLOCKBUF.Y
  9   (   3561,  RISING )  MOVEASAINT.GLOBIC.GRDFU.S_BUF2[0].I
 10   (   4144,  RISING )  MOVEASAINT.GLOBIC.GRDFU.S_BUF2[0].Y
 11   (   4337,  RISING )  MOVEASAINT.GLOBIC.GRDFU.GUARDSPECL3.CK
 12   (   5865, FALLING )  MOVEASAINT.GLOBIC.GRDFU.GUARDSPECL3.Q[2]
 13   (   6186, FALLING )  MOVEASAINT.GLOBIC.GRDFU.S_INV2[2].I
 14   (   6474,  RISING )  MOVEASAINT.GLOBIC.GRDFU.S_INV2[2].Y
 15   (   6583,  RISING )  MOVEASAINT.GLOBIC.GRDFU.S_NAND2[0].I[1]
 16   (   6897, FALLING )  MOVEASAINT.GLOBIC.GRDFU.S_NAND2[0].Y
 17   (   7027, FALLING )  MOVEASAINT.GLOBIC.GRDFU.S_ANDNOR22[7].I[0]
 18   (   7431,  RISING )  MOVEASAINT.GLOBIC.GRDFU.S_ANDNOR22[7].Y
 19   (   7522,  RISING )  MOVEASAINT.GLOBIC.GRDFU.S_ANDNOR21[4].I[0]
 20   (   7997, FALLING )  MOVEASAINT.GLOBIC.GRDFU.S_ANDNOR21[4].Y
 21   (   8093, FALLING )  MOVEASAINT.GLOBIC.GRDFU.S_ORNAND21[0].I[1]
 22   (   8536,  RISING )  MOVEASAINT.GLOBIC.GRDFU.S_ORNAND21[0].Y
 23   (   9042,  RISING )  MOVEASAINT.GLOBIC.GRDFU.S_BUFN3.I
 24   (   9756,  RISING )  MOVEASAINT.GLOBIC.GRDFU.S_BUFN3.Y
 25   (   9968,  RISING )  MOVEASAINT.GLOBIC.NETCONTROL.S_ANDNOR21[0].I[2]
 26   (  10420, FALLING )  MOVEASAINT.GLOBIC.NETCONTROL.S_ANDNOR21[0].Y
 27   (  10536, FALLING )  MOVEASAINT.GLOBIC.NETCONTROL.S_NOR4[5].I[3]
 28   (  11361,  RISING )  MOVEASAINT.GLOBIC.NETCONTROL.S_NOR4[5].Y
 29   (  11608,  RISING )  MOVEASAINT.GLOBIC.NETCONTROL.S_NAND2[0].I[0]
 30   (  11971, FALLING )  MOVEASAINT.GLOBIC.NETCONTROL.S_NAND2[0].Y
 31   (  13003, FALLING )  MOVEASAINT.GLOBIC.NETCONTROL.S_BUFN.I
 32   (  13994, FALLING )  MOVEASAINT.GLOBIC.NETCONTROL.S_BUFN.Y
 33   (  14428, FALLING )  MOVEASAINT.GLOBIC.SHIFU0.R1SOCK.S_INV2[0].I
 34   (  14695,  RISING )  MOVEASAINT.GLOBIC.SHIFU0.R1SOCK.S_INV2[0].Y
 35   (  14846,  RISING )  MOVEASAINT.GLOBIC.SHIFU0.R1SOCK.S_NAND2[0].I[0]
 36   (  15167, FALLING )  MOVEASAINT.GLOBIC.SHIFU0.R1SOCK.S_NAND2[0].Y
 37   (  15230, FALLING )  MOVEASAINT.GLOBIC.SHIFU0.SHIFUI.SHIFUC.S_AND2[0].I[1]
 38   (  15879, FALLING )  MOVEASAINT.GLOBIC.SHIFU0.SHIFUI.SHIFUC.S_AND2[0].Y
 39   (  16024, FALLING )  MOVEASAINT.GLOBIC.SHIFU0.SHIFUI.STAGECTL[0].S_NAND3[0].I[0]
 40   (  16233,  RISING )  MOVEASAINT.GLOBIC.SHIFU0.SHIFUI.STAGECTL[0].S_NAND3[0].Y
 41   (  16333,  RISING )  MOVEASAINT.GLOBIC.SHIFU0.SHIFUI.STAGECTL[0].S_AND2[0].I[0]
 42   (  17224,  RISING )  MOVEASAINT.GLOBIC.SHIFU0.SHIFUI.STAGECTL[0].S_AND2[0].Y
 43   (  17290,  RISING )  MOVEASAINT.GLOBIC.SHIFU0.SHIFUI.S_BUFN1.I
 44   (  17910,  RISING )  MOVEASAINT.GLOBIC.SHIFU0.SHIFUI.S_BUFN1.Y
 45   (  18328,  RISING )  MOVEASAINT.GLOBIC.SHIFU0.SHIFUI.TREG.LOAD
 46   (  19225, FALLING )  MOVEASAINT.GLOBIC.SHIFU0.SHIFUI.TREG.REG.FF2[0].S[0]
```

Squash path

```
________
- clock period results -
________
 nr    period      path   ck_skew     setup    stable
__________________
  1     16833     18531      3561      1862         0
  setup node : MOVEASAINT.GLOBIC.ICCHREMRAM.RBL.A[0]
entry 1 :
1...2_________3.......4 5.....6 7.....8___9..............10_11......12
___13....14_____15......16_17.....18 19....20__21.......22____23....24_____25
...26 27..28___29..30_31...32_33....34__35....36_37....38____39.....40__41..42
__43.....44_45......46_47...48 49
. = 100 ps., cell delay
- = 100 ps., interconnect delay
  1   (      0,  RISING )  MOVEASAINT.CK.BOND
  2   (    390,  RISING )  MOVEASAINT.CK.Y
  3   (   1370,  RISING )  MOVEASAINT.GLOBIC.VALFU.S_CLOCKBUF2.I
  4   (   2116,  RISING )  MOVEASAINT.GLOBIC.VALFU.S_CLOCKBUF2.Y
  5   (   2140,  RISING )  MOVEASAINT.GLOBIC.VALFU.S_CLOCKBUF1.I
  6   (   2650,  RISING )  MOVEASAINT.GLOBIC.VALFU.S_CLOCKBUF1.Y
  7   (   2717,  RISING )  MOVEASAINT.GLOBIC.SCRFU0.S_CLOCKBUF.I
  8   (   3238,  RISING )  MOVEASAINT.GLOBIC.SCRFU0.S_CLOCKBUF.Y
  9   (   3561,  RISING )  MOVEASAINT.GLOBIC.CMPFU1.CMPFUI.RREG.CK
 10   (   5069, FALLING )  MOVEASAINT.GLOBIC.CMPFU1.CMPFUI.RREG.Q[0]
 11   (   5198, FALLING )  MOVEASAINT.GLOBIC.CMPFU1.S_BUFN1.I
 12   (   5832, FALLING )  MOVEASAINT.GLOBIC.CMPFU1.S_BUFN1.Y
 13   (   6135, FALLING )  MOVEASAINT.GLOBIC.GRDFU.S_NOR2[4].I[0]
 14   (   6536,  RISING )  MOVEASAINT.GLOBIC.GRDFU.S_NOR2[4].Y
 15   (   7050,  RISING )  MOVEASAINT.GLOBIC.GRDFU.S_ANDNOR22[6].I[2]
 16   (   7727, FALLING )  MOVEASAINT.GLOBIC.GRDFU.S_ANDNOR22[6].Y
 17   (   7847, FALLING )  MOVEASAINT.GLOBIC.GRDFU.S_ANDNOR21[5].I[0]
 18   (   8351,  RISING )  MOVEASAINT.GLOBIC.GRDFU.S_ANDNOR21[5].Y
 19   (   8448,  RISING )  MOVEASAINT.GLOBIC.GRDFU.S_ORNAND21[1].I[1]
 20   (   8856, FALLING )  MOVEASAINT.GLOBIC.GRDFU.S_ORNAND21[1].Y
 21   (   9094, FALLING )  MOVEASAINT.GLOBIC.GRDFU.S_BUFN2.I
 22   (   9816, FALLING )  MOVEASAINT.GLOBIC.GRDFU.S_BUFN2.Y
 23   (  10248, FALLING )  MOVEASAINT.GLOBIC.IFEFU0.TSOCK.S_NOR2[0].I[1]
 24   (  10701,  RISING )  MOVEASAINT.GLOBIC.IFEFU0.TSOCK.S_NOR2[0].Y
 25   (  11222,  RISING )  MOVEASAINT.GLOBIC.IFEFU0.TSOCK.S_INV[3].I
 26   (  11550, FALLING )  MOVEASAINT.GLOBIC.IFEFU0.TSOCK.S_INV[3].Y
 27   (  11625, FALLING )  MOVEASAINT.GLOBIC.IFEFU0.TSOCK.S_NAND2[0].I[1]
 28   (  11857,  RISING )  MOVEASAINT.GLOBIC.IFEFU0.TSOCK.S_NAND2[0].Y
 29   (  12205,  RISING )  MOVEASAINT.GLOBIC.IFEFU0.IFEFUI.PCSEL.S_INV2[1].I
 30   (  12491, FALLING )  MOVEASAINT.GLOBIC.IFEFU0.IFEFUI.PCSEL.S_INV2[1].Y
 31   (  12618, FALLING )  MOVEASAINT.GLOBIC.IFEFU0.IFEFUI.PCSEL.S_NOR2[1].I[1]
 32   (  13013,  RISING )  MOVEASAINT.GLOBIC.IFEFU0.IFEFUI.PCSEL.S_NOR2[1].Y
 33   (  13200,  RISING )  MOVEASAINT.GLOBIC.IFEFU0.IFEFUI.PCSEL.S_ANDNOR21[0].I[0]
 34   (  13675, FALLING )  MOVEASAINT.GLOBIC.IFEFU0.IFEFUI.PCSEL.S_ANDNOR21[0].Y
 35   (  13919, FALLING )  MOVEASAINT.GLOBIC.IFEFU0.IFEFUI.PCSEL.S_NOR2[6].I[0]
 36   (  14379,  RISING )  MOVEASAINT.GLOBIC.IFEFU0.IFEFUI.PCSEL.S_NOR2[6].Y
 37   (  14479,  RISING )  MOVEASAINT.GLOBIC.IFEFU0.IFEFUI.PCSEL.S_ORNAND21[0].I[0]
 38   (  14882, FALLING )  MOVEASAINT.GLOBIC.IFEFU0.IFEFUI.PCSEL.S_ORNAND21[0].Y
 39   (  15354, FALLING )  MOVEASAINT.GLOBIC.IFEFU0.IFEFUI.PCSEL.S_NOR2[4].I[1]
 40   (  15873,  RISING )  MOVEASAINT.GLOBIC.IFEFU0.IFEFUI.PCSEL.S_NOR2[4].Y
 41   (  16151,  RISING )  MOVEASAINT.GLOBIC.IFEFU0.IFEFUI.PCSEL.S_INV2[26].I
 42   (  16382, FALLING )  MOVEASAINT.GLOBIC.IFEFU0.IFEFUI.PCSEL.S_INV2[26].Y
 43   (  16639, FALLING )  MOVEASAINT.GLOBIC.IFEFU0.IFEFUI.PCSEL.S_ORNAND22[11].I[2]
 44   (  17186,  RISING )  MOVEASAINT.GLOBIC.IFEFU0.IFEFUI.PCSEL.S_ORNAND22[11].Y
 45   (  17370,  RISING )  MOVEASAINT.GLOBIC.IFEFU0.IFEFUI.CCHPTH.S_ANDNOR22[4].I[3]
 46   (  18012, FALLING )  MOVEASAINT.GLOBIC.IFEFU0.IFEFUI.CCHPTH.S_ANDNOR22[4].Y
 47   (  18134, FALLING )  MOVEASAINT.GLOBIC.IFEFU0.IFEFUI.CCHPTH.S_NAND2[9].I[0]
 48   (  18497,  RISING )  MOVEASAINT.GLOBIC.IFEFU0.IFEFUI.CCHPTH.S_NAND2[9].Y
 49   (  18531,  RISING )  MOVEASAINT.GLOBIC.ICCHREMRAM.RBL.A[0]
```

PC increment path

```
- clock period results -
 nr    period      path   ck_skew     setup    stable
  1     13940     15232      3561      2268         0
  setup node : MOVEASAINT.GLOBIC.IFEFU0.IFEFUI.PCCIFREG.REG.FF2[10].D[1]
entry 1 :
1...2_________3.......4 5.....6 7.....8___9...............10_____11
...........................................................................
........12__13..14 15....16 17
. = 100 ps., cell delay
- = 100 ps., interconnect delay
  1   (      0,  RISING )  MOVEASAINT.CK.BOND
  2   (    390,  RISING )  MOVEASAINT.CK.Y
  3   (   1370,  RISING )  MOVEASAINT.GLOBIC.VALFU.S_CLOCKBUF2.I
  4   (   2116,  RISING )  MOVEASAINT.GLOBIC.VALFU.S_CLOCKBUF2.Y
  5   (   2140,  RISING )  MOVEASAINT.GLOBIC.VALFU.S_CLOCKBUF1.I
  6   (   2650,  RISING )  MOVEASAINT.GLOBIC.VALFU.S_CLOCKBUF1.Y
  7   (   2717,  RISING )  MOVEASAINT.GLOBIC.SCRFU0.S_CLOCKBUF.I
  8   (   3238,  RISING )  MOVEASAINT.GLOBIC.SCRFU0.S_CLOCKBUF.Y
  9   (   3561,  RISING )  MOVEASAINT.GLOBIC.IFEFU0.IFEFUI.PCIFREG.CK
 10   (   5069, FALLING )  MOVEASAINT.GLOBIC.IFEFU0.IFEFUI.PCIFREG.Q[4]
 11   (   5664, FALLING )  MOVEASAINT.GLOBIC.IFEFU0.IFEFUI.PCINCR.ADDERN.A[4]
 12   (  14048,  RISING )  MOVEASAINT.GLOBIC.IFEFU0.IFEFUI.PCINCR.ADDERN.S[10]
 13   (  14331,  RISING )  MOVEASAINT.GLOBIC.IFEFU0.IFEFUI.PCSEL.S_INV2[4].I
 14   (  14596, FALLING )  MOVEASAINT.GLOBIC.IFEFU0.IFEFUI.PCSEL.S_INV2[4].Y
 15   (  14682, FALLING )  MOVEASAINT.GLOBIC.IFEFU0.IFEFUI.PCSEL.S_ORNAND22[15].I[3]
 16   (  15174,  RISING )  MOVEASAINT.GLOBIC.IFEFU0.IFEFUI.PCSEL.S_ORNAND22[15].Y
 17   (  15232,  RISING )  MOVEASAINT.GLOBIC.IFEFU0.IFEFUI.PCCIFREG.REG.FF2[10].D[1]
```

Appendix C. Layout results

The following pages show MOVE processor generator layout results for the MOVE32INT configuration in the 1 μ ES2/CMS1M0 process. Standard cells are generated on the FU level. The layout results were obtained without manual intervention.

Figure C.1 shows the initial cell placement performed by ASA. The cell sizes are estimates in this figure. Placement of cells is done in the order in which they appear in the *use* part of the *globic* system (global interconnect) in the file *globic.sih*. The order of placement is critical for the ASA placement algorithm to perform well (e.g. the wide RAM cell with its fixed cell ratio is placed first, so that the ratio of standard cells that follow can be adjusted properly).

Final cell placement and routing between the cells is shown in figure C.2. Note that the initial placement is modified to account for actual cell sizes and interconnect. Also note that the RAM cell (*icchremram)* is relatively small at 256 bytes. The actual die size is 0.90×1.19 cm.

Figures C.3 and C.4 show example pieces of standard cell and RAM (real, regular) cell, respectively. Note the routing channels visible in the standard cell layout.

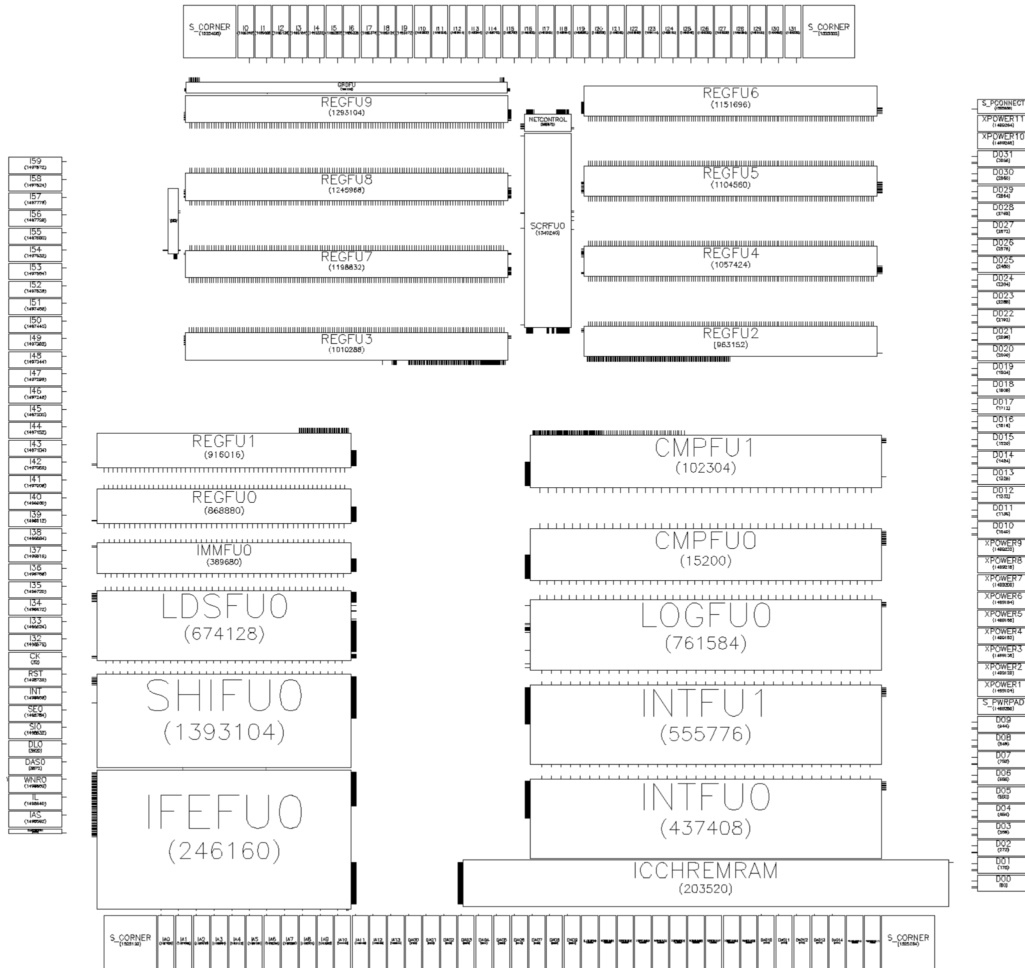


Figure C.1. Initial cell placement

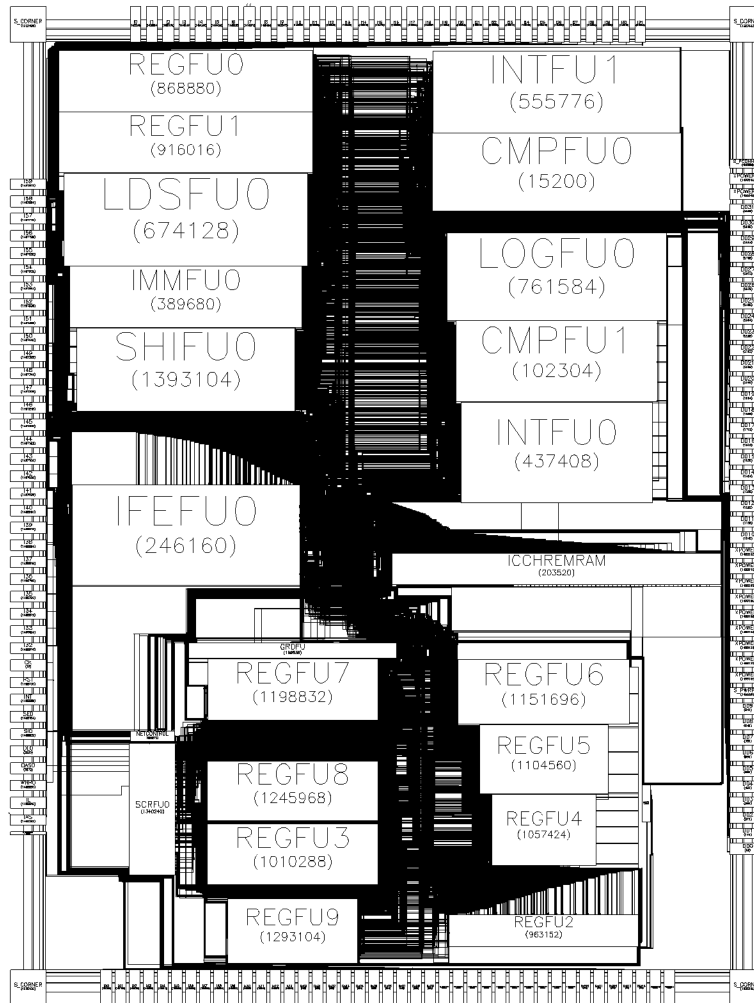


Figure C.2. Final layout with routing, no cell details

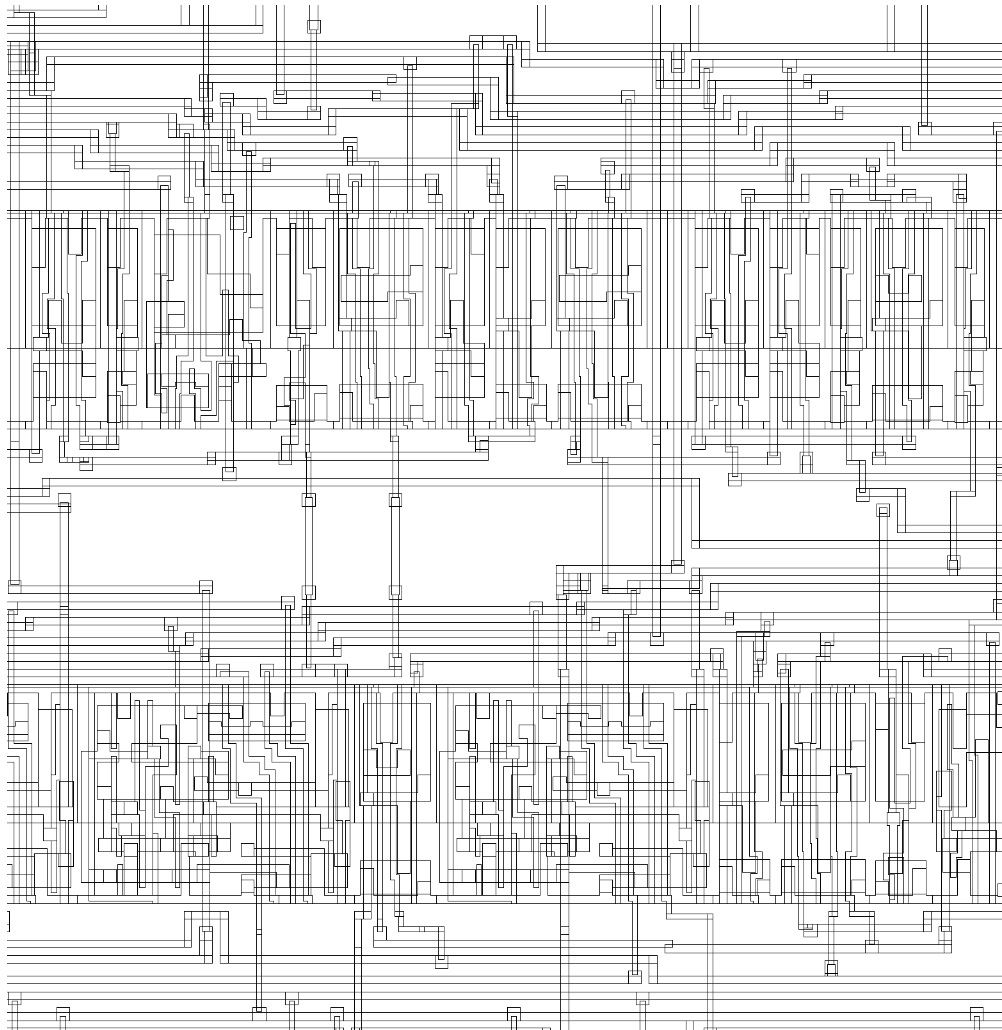


Figure C.3. An example piece of standard cell layout

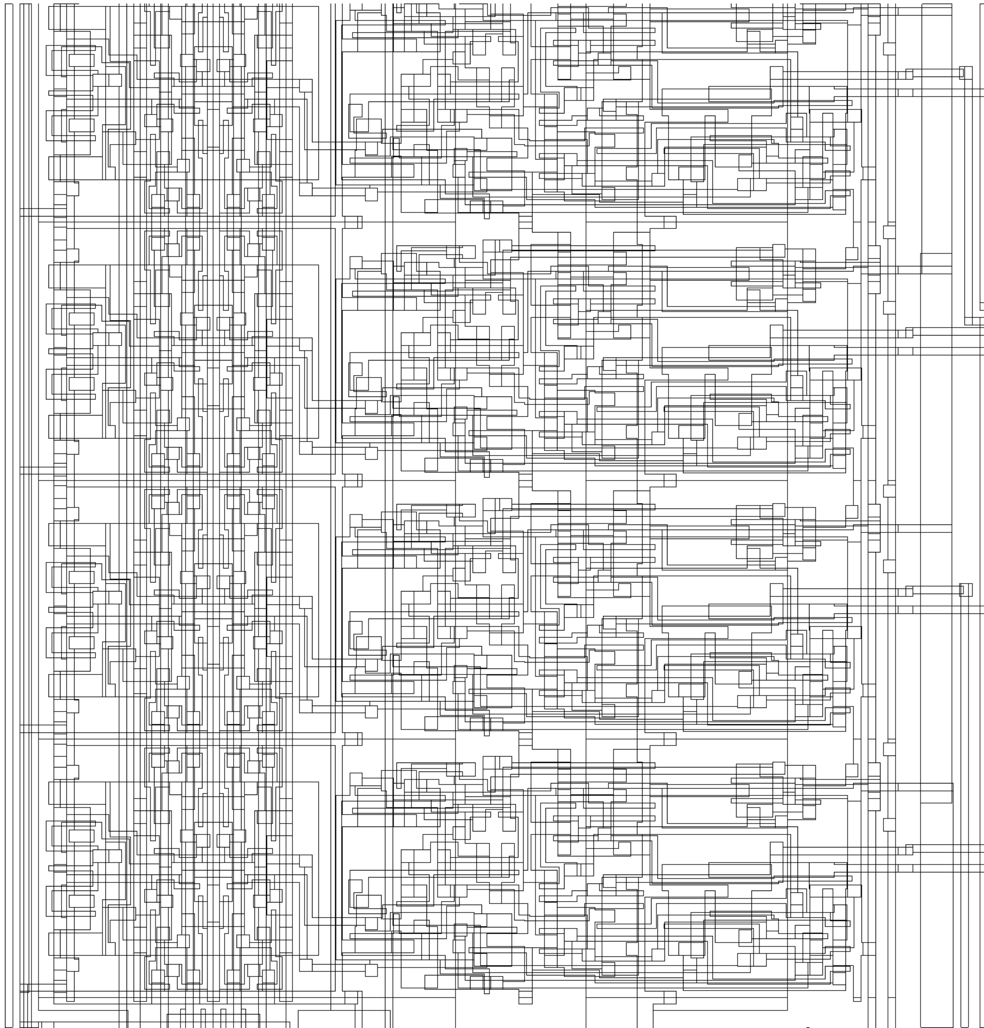


Figure C.4. An example piece of RAM (regular) layout

Appendix D. Sources

The remainder of this report consists of the most important sources of the MOVEASAINT processor generator. The tools used to implement this system are Sagantec's ASA (silicon compiler) version 5.4, m4 (macro preprocessor), and ANSI C. The sources consist mainly of SID (ASA's hardware description language) intermixed with preprocessing statements.

Sources not included in this version.