MOVE32INT

Architecture and Programmer's Reference Manual

Henk Corporaal

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

April 20, 1994

Abstract

The MOVE project at the Faculty of Electrical Engineering of the TU Delft aims at designing and building high-performance processors by means of a new class of architectures called *transport triggered architectures*. Transport triggered architectures are programmed by specifying data transports instead of operations; their flexibility makes them particular suited for application specific purposes.

In order to evaluate the specific design and implementation tradeoffs, an instance of a transport triggered architecture, called *MOVE32INT*, has been designed and implemented in a 1.6 μ (minimum gate length) CMOS Sea of Gates (SoG) technology at the Computer Architecture Department. The SoG pattern used for this implementation, and the tools for generating layout on this SoG, are developed at the Electronics Department.

MOVE32INT is a 80 MHz, 32 bit general purpose integer pipelined processor, containing several function units which operate concurrently. To keep these function units busy it is possible to perform four concurrent data transports per cycle, giving a peak performance of 320 million operations per second.

Although the functionality of *MOVE32INT* is designed towards general purpose applications, the concept of transport triggering is very flexible with regard to type, number and implementation of function units, and to required transport capacity. Current developments therefore aim at the implementation and realization of a *MOVE* framework which allows for automatic generation of application specific processors.

This manual describes the architecture of *MOVE32INT*, and is meant for software developers, like system programmers and compiler writers. A separate manual [vdA93] describes the implementation and realization of this processor.

Keywords: General purpose processor, transport triggering, instruction level parallelism, guards, locking mechanism, hybrid pipelining.

Acknowledgements

The design, implementation and realization of *MOVE32INT* has been an important part of the MOVE project at the Delft University of Technology. The MOVE project started in the fall of 1990. Many employees and students contributed to different parts of this project. I would like to mention:

1. Hans Mulder, who gave the project a big boost during the first year of its existence.
2. Implementation and realization of *MOVE32INT*: Paul van der Arend, Paul Stravers, Rene Bodenstaff, Erik Aardoom, Harry Snier, Guillermo Cuppers.
3. Design of PC test board for *MOVE32INT*: Sietze Schukking, Eric Demeijer.
4. Architecture: Reinoud Lamberts, Eddy Olk.
5. Compiler support: Jan Hoogerbrugge, Robert Portier, Andy Verberne, Ivan Sadinsky, Arno Roelofs, Theo Baan.
6. Computer system management and Monitor software: Jean Paul van der Jagt and Rogier Wolff.

Listed are the major fields to which people contributed. However, most persons influenced on other areas as well; there was much crosstalk. Besides the mentioned persons, many other students were involved in research which (partially) contributed to the MOVE project.

Contents

1 Introduction

Within the Computer Architecture Department of the Electrical Engineering Faculty at the Delft University of Technology (TUD) research is done into the field of High Performance Computing. For this purpose a processor framework, which is called *MOVE*, has been developed. This framework combines great functional flexibility with a high scalability in performance. Therefore the *MOVE* framework is particularly suited for the design of application specific processors with an outstanding cost-performance ratio. Even for general purpose applications or massive parallel MIMD applications a *MOVE* processor offers a combination of advantages, like extra scheduling opportunities and very short cycle times, which makes its performance superior to other current available solutions.

Currently, we are in the process of implementing and realizing this *MOVE* framework. This includes architecture design, VLSI design, circuit simulation, the implementation of retargetable compilers and processor simulators, and the implementation of tools for application analysis.

As part of this process, a 32-bit integer *MOVE* processor, called *MOVE32INT*, has been implemented and realized in a 1.6 μ CMOS Sea of Gates (SoG) technology, having a minimal feature size of 2.0 μ, and a minimal gate length of 1.6 μ. The used SoG pattern, and the tools for generating layout on this SoG, are developed at the Electronics department of the TUD.

Following sections describe the design goals for *MOVE32INT*, and clarify the structure of this architecture and programmer's reference manual.

1.1 Design goals

MOVE32INT is a fully programmable single chip processor realization of a Transport Triggered Architecture (TTA) supporting 32 bit integer operations. In implementing and realizing *MOVE32INT* we had several goals in mind:

1. Demonstrating that high cycle times could be achieved using the *MOVE* concept.
2. Demonstrating the feasibility of the *MOVE* concept. Especially we were interested how new features like guarding of all transports, hybrid function units and locking (see chapter 2) could be efficiently implemented and realized.
3. Verifying timing figures obtained from timing analysis with SPICE.
4. Testing the suitability of SoG implementations for processor design. In particular we were worried about VLSI area and power consumption.
5. Getting experience with high frequency board and system design.
6. Use *MOVE32INT* as a prototype for following implementations.

In order to reach these goals as quickly as possible, many beautiful, but unnecessary features are omitted in. For example, *MOVE32INT* does not contain fancy exception handling using exception propagation, complex function units like multiply and divide, a data cache, byte support instructions, etc. Many of these features will be reconsidered for implementation in the currently designed *MOVE* framework.

1.2 Manual structure

This manual describes and defines the architecture of *MOVE32INT*. Because the *MOVE* framework is based on a rather unconventional architecture, chapter 2 introduces the underlying concept of this architecture, the *MOVE* concept, its programming model, and its advantages. *MOVE* architectures contain a couple of function units (FUs) which perform the necessary operations. In general the FUs of *MOVE* architectures are fully pipelined. The pipelining method deviates from traditional approaches. These differences are also explained in this chapter. The following chapters (3, 4, and 5) describe the architecture, the supported instructions and operation set, and the locking, guarding and exception model of *MOVE32INT*; they are intended for *MOVE* programmers. Chapter 6 describes the external appearance of *MOVE32INT*. It contains necessary information for board level designers who want to integrate a *MOVE32INT* processor into their design.

Several appendices are added which contain more architecture and implementation details. Appendix A describes the register assignments of all the available registers within *MOVE32INT*. The next appendix B pictures detailed timing information of the *MOVE* instruction cycle. This information is not needed for writing programs or designing *MOVE* systems, however, it adds in understanding the *MOVE* programming model. Appendix C examines the correct usage of each function unit and its corresponding operations in detail. It can be used as a quick reference. Appendix D describes how the state of the machine can be saved and restored during exceptions. Finally, appendix E gives several characteristics about the Sea of Gates VLSI chip realization and pin layout of *MOVE32INT*.

The *MOVE* class of architectures

2

This chapter gives a general introduction to Transport Triggered Architectures (TTAs) in general and *MOVE* architectures in particular. TTAs can be viewed as a number of function units (FUs) performing operations and register units (RUs) containing general purpose registers (GPRs)[1], connected by some kind of transport network (see figure 2.1). By changing the type and number of FUs and the connectivity and capacity of the transport network a whole range of architectures can be implemented; TTAs are therefore ideal for being tailored to any kind of application.

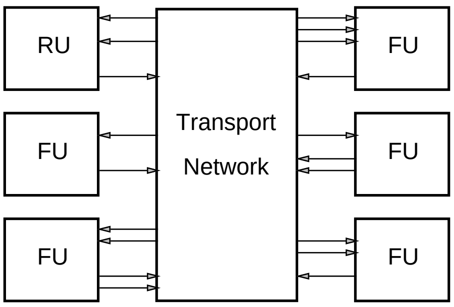


Figure 2.1: TTA, operational view.

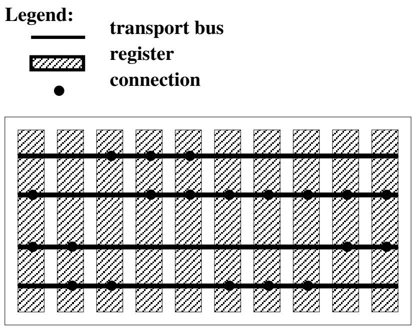


Figure 2.2: TTA, transport view.

MOVE architectures form a specific subclass of TTAs. Besides being based on the concept of transport triggering, they are enhanced with so called hybrid FUs, which allow for more scheduling freedom and an easier design.

In the next sections we first describe into some detail the concept of transport triggering, and explain the programming paradigm of architectures based on this concept. Second, the particular implementation

[1]Register units will be viewed as normal function units performing the identity operation.

of FUs using a so called hybrid pipelining mechanism is discussed. The scheduling constraints of these FUs are described in the third section. Third, a section is spend to connectivity aspects between FUs and the transport network. Finally we indicate possible advantages of MOVE-like architectures (for a complete qualitative analysis see [CM91]).

2.1 The concept of Transport Triggering

Traditional CISC, RISC, VLIW and Superscalar architectures are programmed by specifying the operations. As a side effect of these operations trigger several data transports during execution. For example, when we specify the operation $Addr_1, r_2, r_3$, the transports of the values from registers r_1, r_2, and r_3 are scheduled during specific time slots, and on specific busses[2]. Therefore we call these architectures *operation triggered*. In *transport triggered* architectures (TTAs) this programming paradigm is mirrored: instead of operations which trigger transports, we specify transports which trigger operations as possible side effect. Transport triggered architectures separate transport and operations; data transports are made visible at the architecture level. The transport view of TTAs shows a set of registers connected in some way by a data transport network. Figure 2.2 shows a simple transport network, where all registers are connected to each other by a multiple bus structure.

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

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

Trigger registers: A *move* to a trigger register causes the start of an operation on the accompanying FU. The moved value is used as an operand for the operation. If more operands are needed, they are taken from operand registers of the FU. To handle different operations on a single FU, the trigger register is mapped into different logical address locations.

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

All FUs contain at least one trigger type of register. Practical TTAs handle multiple moves in parallel; in this sense they resemble VLIW architectures.

In order to make the concept of transport triggering more clear, a code example is given in figure 2.3. It shows the addition of GPRs r_a and r_b and a store of the result at memory address r_c. The code starts with moving r_a and r_b to the operand and trigger registers of the integer unit (registers io and add). The trigger register is accessed at the location that causes an addition on the integer unit. The operand move(s) must always be done before or in the same cycle as the trigger move. When the result of the addition leaves the pipeline it is available in the result register of the integer unit. In our example the latency of the integer unit is two, so the result *move* can take place two cycles after the trigger move. The result *move* passes the result directly to the trigger register of the memory unit. At the same time the address of the store is placed in the operand register of the memory unit. Thus a store is triggered

[2] In general many other data transports happen as additional side effects in the bypass unit.

$r_a \rightarrow Oi$	$r_b \rightarrow Tadd$	; trigger addition
...		; space for other moves
$r_c \rightarrow Osa$	$ir \rightarrow Tsd$	; trigger store

Figure 2.3: The *MOVE* code for an addition followed by a store.

with the data to be stored, and the address of the store is supplied via the operand register. Since a store has no result, no result *move* is needed. On average about 2 moves are necessary per operation.

Fully programmable architectures need at least support for flow control operations, conditional execution and immediates. Flow control operations are also done by *move* operations. An absolute jump is simply writing the target address into the program counter, which is visible in the register address space. Page relative jumps are supported by writing a small immediate to the program counter displacement register. Both jumps generally have delay slots before instructions at the target address are executed.

Conditional execution is supported by guards. Each *move* can be guarded. A guarded *move* $g : r_a \rightarrow r_b$ only takes place when the guard g evaluates to true. In *MOVE32INT* g can be a boolean expression of two boolean values produced by the compare function units. The translation of several flow control operations into *moves* is given in figure 2.4.

jump r_a	$\Rightarrow$	$r_a \rightarrow Tpc$	
cjump $r_a = r_b, r_c$	$\Rightarrow$	$r_a \rightarrow Ocmp$ $g : r_c \rightarrow Tpc$	$r_b \rightarrow Teq$
call r_a	$\Rightarrow$	$Rpc \rightarrow ra$	$r_a \rightarrow Tpc$
return	$\Rightarrow$	$ra \rightarrow Tpc$	

Figure 2.4: Several flow control operations.

The final point that should be told about programming TTAs concerns immediates. Small sized immediates can be provided by treating the source register address of a *move* operation as an immediate. Larger immediates can be supported by making the instruction register visible in the architecture. As an example, Figure 2.5 shows the *move* operation format as supported by *MOVE32INT*. The interpretation of the source field is determined by the immediate flag.

2.2 Hybrid pipelined function units

Use of the transport triggering concept allows for many compiler optimizations at the data transport level. In order to enhance the scheduling freedom for the compiler even further, and to make the design of *MOVE* based processors easier, most FUs within *MOVE* architectures are implemented using a so called

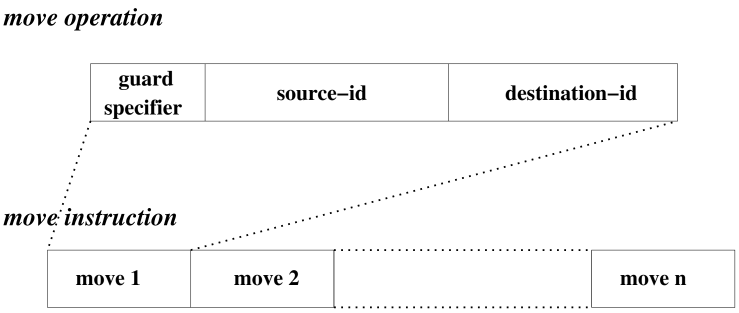


Figure 2.5: *MOVE32INT* operation format.

hybrid pipelining mechanism. Figure 2.6 shows a typical hybrid function unit. Its pipeline has a latency of 3 clock cycles; this means that reading of the result register (R) can be done 3 cycles after writing to the trigger register (T).

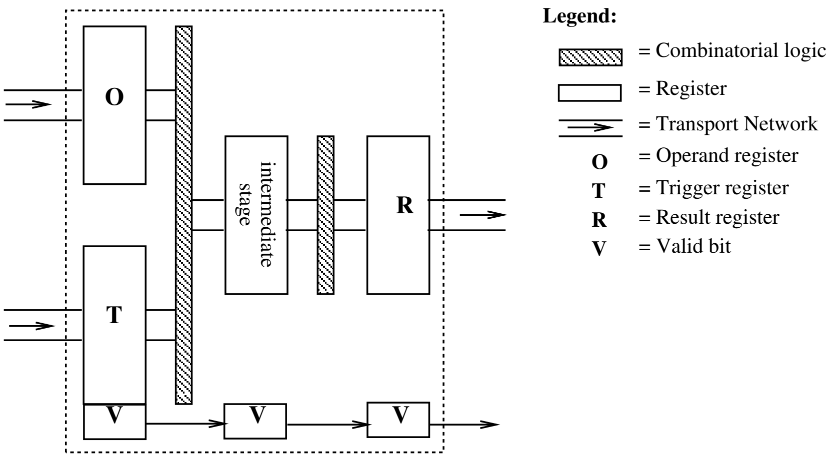


Figure 2.6: The pipeline of a typical function unit.

Current high-performance architectures generally implement pipelines where either intermediate pipeline stages continue always on each clock cycle to the next stage (e.g. [RYYT89, Tra87]) or intermediate stages continue only in case a new operation is issued (e.g. [i8689]). In hybrid pipelines intermediate values continue to the next stage on each clock cycle, as far as they do not overwrite results from previous operations. Valid bits (V) are used to control this mechanism. This offers extra scheduling freedom; results may be moved from the FUs at any time after triggering the operation[3]. Hybrid pipelines also reduce register usage by using intermediate stages as temporary storage.

Pipelines are described by the following parameters:

- l : latency (in clock cycles)
- d : distance between pipeline stages (in clock cycles)
- t : throughput (in operations/cycle); $t = 1/d$

[3]Even before the result has been calculated; *MOVE32INT* implements a locking mechanism which avoids the insertion of no-op instructions.

s : number of pipeline stages

These parameters are coupled by the equation[4]

$$l = (s - 1)d + 1 \tag{2.1}$$

Latency l is here interpreted as the number of cycles one has to wait after triggering the function unit, in order to read the result from that function unit[5].

2.3 Scheduling constraints for hybrid pipelined function units

This section discusses the restrictions on scheduling data transports (*moves*) for hybrid pipelined function units, like the one shown in figure 2.6. Let us first introduce some terminology.

Def. 2.1 *T_i, O_i and R_i are defined as the trigger, operand and result move corresponding to an operation OP_i on a function unit $FU(OP_i)$.*

Def. 2.2 *For any move M_i, $C(M_i)$ defines the clock cycle in which M_i is executed.*

Def. 2.3 *For any moves M_i and M_j, $D(M_i, M_j)$ denotes the difference in clock cycli between the execution of M_i and M_j, i.e.:*

$$D(M_i, M_j) = C(M_j) - C(M_i)$$

Normally $D(M_i, M_j)$ is equal to the number of cycles between M_i and M_j in the generated code, as produced by the compiler or by hand. Due to e.g. cache misses $D(M_i, M_j)$ may increase however.

Def. 2.4 *$NT(C, FU)$, $NO(C, FU)$, and $NR(C, FU)$ are defined as the number of trigger, operand and result moves which are executed for function unit FU until, and including, cycle C.*

In order to define the scheduling restrictions we consider independent operations OP_i, where index i ranges from $1 - n$, n being the total number of operations which map on the same function unit. This means that $\forall i, j \in [1, n]$, $FU(OP_i) = FU(OP_j)$. Each operation consists of a trigger, operand and result move. Figure 2.7 shows two independent operations; the indicated numbers give the minimum dependence distance between dependent moves. We further assume that operations OP_i are mapped in order, i.e.

$$i > j \leftrightarrow C(T_i) > C(T_j)$$

Using above definitions we are able to define the scheduling restrictions.

[4]Assuming constant distance d between all stages.

[5]Some MOVE architectures omit the (physical) result register; they combine the final FU logic and the data transport into one cycle. This changes equation 2.1 to $l = s \times d$.

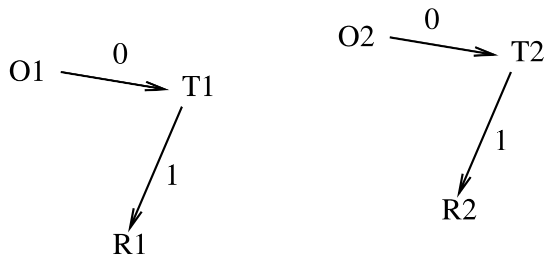


Figure 2.7: Dependencies between moves of two independent operations.

2.3.1 General constraints

Operand moves have to be scheduled earlier, or at least in the same cycle, as trigger moves. Due to the implemented locking discipline, result moves may be scheduled in any instruction slot after the corresponding trigger move has been scheduled. Therefore:

$$D(O_i, T_i) \geq 0$$

$$D(T_i, R_i) \geq 1$$

Fifo behavior of hybrid pipelines results in the following restriction:

$$C(T_i) > C(T_j) \leftrightarrow C(R_i) > C(R_j)$$

The operand value may not be overwritten if it's not yet consumed; this means:

$$C(T_i) > C(T_j) \leftrightarrow C(O_i) > C(T_j)$$

Two moves with the same physical destination may not be scheduled in the same cycle, therefore:

$$\forall i \neq j, D(T_i, T_j) \neq 0, D(O_i, O_j) \neq 0, and\ D(R_i, R_j) \neq 0$$

2.3.2 Exceptions

Above constraints are not sufficient in order to generate semantically correct code. We have also to make sure that function units are not overloaded. This happens when a trigger, or operand move is scheduled on a full FU. Therefore, for a FU with s stages one has to make sure that:

$$C(T_{i+s}) \geq C(R_i)$$

and

$$C(O_{i+s}) \geq C(R_i)$$

In a similar fashion, it is not allowed to schedule a result move on an empty FU, therefore:

$$C(R_i) > C(T_i)$$

This latter condition is however taken care off by the general constraints, as a result of data dependencies. A FU may generate full or empty exceptions in case trigger or result moves are scheduled which invalidate the former two constraints[6]. The condition for an empty exception caused by result move R_i is given by:

$$empty\!-\!exception(R_i) \equiv C(R_i) \leq C(T_i) \tag{2.2}$$

The conditions for a full exception are:

$$full\!-\!exception(T_i) \equiv C(T_i) < C(R_{i-s}) \tag{2.3}$$

$$full\!-\!exception(O_i) \equiv C(O_i) < C(R_{i-s}) \tag{2.4}$$

2.3.3 Locking

MOVE architectures support two different locking mechanisms:

- *FU locking*. Inside hybrid pipelined function units, as explained in section 2.2. This allows result moves to be scheduled later, as compared to continue always pipelined function units.
- *Transport locking*. The transport network also supports locking. This is used in case function units are not yet ready to accept new operands or to deliver a result.

Both locking mechanisms enhance the scheduling freedom, and let the function units operate more autonomously from the transport network. In general a compiler will try to schedule operations in such a way as to avoid locking of the data transport network, unless it can not schedule any useful transport in a particular cycle. In the latter case the resulting empty instruction may be removed, which results in transport locking during execution.

Internal FU locks are not always avoided by the compiler; e.g., it may be advantageous to schedule (non critical) result moves not directly as soon as the result is available (according to the FU latency), but in a later cycle, resulting in less general purpose register usage and/or usage of the data transport network.

Two types of transport locks are distinguished:

- *Read locks*; they are generated when a FU can not deliver the result in time. So read locks are only generated for result moves.
- *Write locks*; they occur when writing to a FU which can temporarily not accept the operand. Write locks may be generated both for operand and trigger moves.

The occurrence condition for a read lock caused by a result move R_i is given by[7]:

$$read\!-\!lock(R_i) \equiv C(R_i) < C(T_i) + l \tag{2.5}$$

Write lock conditions for T_i and O_i are:

$$write\!-\!lock(T_i) \equiv \overline{pipeline\!-\!full(C(T_i))} \wedge (D(T_{i-1}, T_i) < d) \tag{2.6}$$

[6] These type of exceptions have not been implemented in *MOVE32INT*. A simple code checker can verify (at compile time) the fulfillment of given conditions.

[7] It is not necessary to generate a read lock if the pipeline is empty. However in *MOVE32INT*, where full and empty exceptions are not supported, most (except for the compare units) still generate a read lock in that case.

$$write - lock(O_i) \equiv \overline{pipeline-full(C(O_i))} \wedge (D(T_{i-1}, O_i) < d) \tag{2.7}$$

where the condition for the pipeline of function unit FU being full at cycle C is given by:

$$pipeline-full(C, FU) \equiv NT(C, FU) - NR(C, FU) = s$$

To summarize all the different situations, table 2.1 illustrates the behavior of a 2 stage FU with distance $d = 2$ between the pipeline stages (maximal throughput is one operation per 2 cycles). According to equation 2.1 the latency $l = 3$ for this FU. We assume that the FU is empty before cycle 0. Remark that

scheduling cycle	execution cycle	T, O and R moves	full? y/n	function unit behavior
0	0	R0	n	empty exception
1	1	T1, O1	n	
2	3	T2	y	1 cycle write lock
3	4	R1	n	
4	6	T3, O3, R2	n	1 cycle read and write lock
5	8	O4	n	1 cycle write lock
6	9	T4	y	
7	10	O5	y	full exception

Table 2.1: Behavior of a function unit with $d = 2$, $s = 2$

different triggers may use the same operand, e.g. T2 uses operand scheduled by O1. This is a typical case of so called *common operand elimination*, which is further explained in the next section.

2.4 Scheduling optimizations

In general a three address operation is converted into three move operations. For example, the translation of a three address register add operation is given by:

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

Note that many RISC-like operations can be modeled with less than 3 moves, e.g. monadic operations, jumps, and register copies.

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

ADD r3,r1,r2 $\Rightarrow$ r1$\rightarrow$O$_{ADD}$; r2$\rightarrow$T$_{ADD}$
ADD r5,r3,r4 $\Rightarrow$ R$_{ADD}$$\rightarrow$O$_{ADD}$; r4$\rightarrow$T$_{ADD}$; R$_{ADD}$$\rightarrow$r3
R$_{ADD}$$\rightarrow$r5

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

When subsequent uses of the same FU share the data of one of the operands, an additional saving in moves is possible. This is called *common operand elimination*. Instead of needing 3 moves, three-address operations can now be translated into 1.5 to 2 moves.

It also may happen that the register $r3$ is not live any more; this allows for the elimination of the move to $r3$ (which we call dead *write back elimination*). As shown in the following code:

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

The minimal number of moves for an addition now becomes one.

2.5 Connection between FUs and the network

In general transport triggered architectures do not have to be fully connected; that means that some or all registers are not connected to all the available busses. The connection between the network and FUs is established by so called input and output *sockets*. Figure 2.8 shows a possible connection. The figure

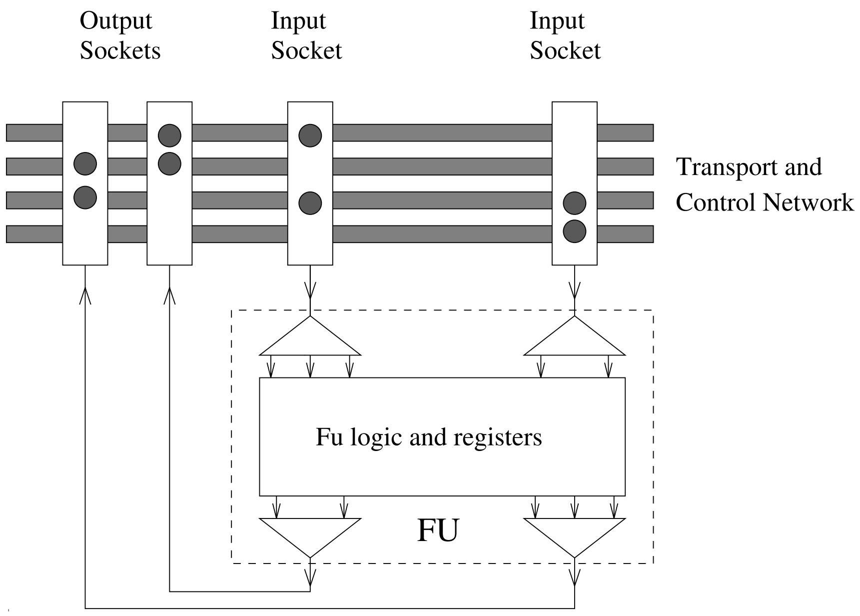


Figure 2.8: Connection between FU and the network using sockets

shows a FU which is connected to 2 input and 2 output sockets. All sockets in the figure are connected to 2 (of the 4) move busses. Internally the FU may connect multiple inputs to one input socket; the same holds for outputs. A socket can be viewed as a gateway which is able to pass one data item per cycle. Figure 2.9 presents some more details of an input socket, which is connected to 2 move busses. A move bus consists of a data, identifier and control bus, which are all connected to the socket. The instruction decoding is performed locally, inside the sockets; it checks if the socket is selected, and if so, it extracts an opcode from the source identifier on the identifier bus and signals the function unit with this opcode. Control logic is used for locking and squashing (guarding transports).

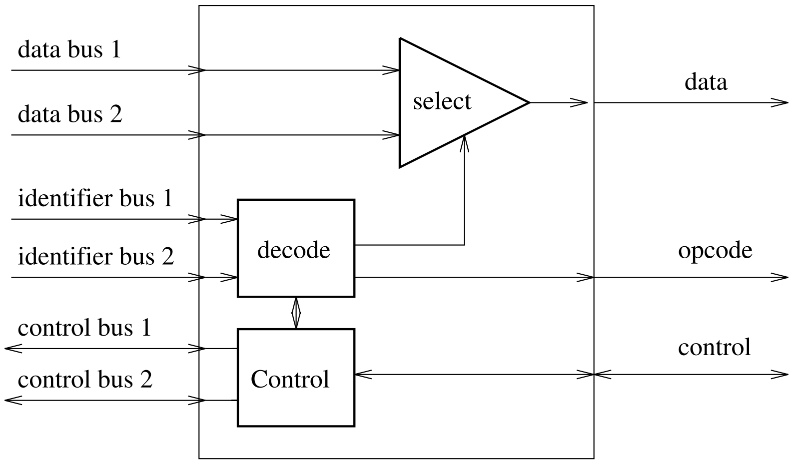


Figure 2.9: Details of an input socket

2.6 Advantages of using *MOVE* architectures

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

1. Cycle-time minimization. Cycle-time is mainly constrained by data transport.

2. Functional flexibility. The *MOVE* concept does not constrain function-unit functionality in any way. E.g., FUs can have many source and destination operands.

3. Implementation flexibility. The *MOVE* concept does not constrain the implementation of transport network or FU unnecessary. E.g. combinations of pipelined and nonpipelined FUs and/or transport networks are possible.

4. Performance scalability. Performance scales by adding multiple FUs to an existing architecture, pipelining existing FUs, or by increasing the data transport and storage capacity. In contrast to current VLIW processors, the number of function units is not coupled to the data transport capacity (e.g. the number of register ports), e.g., function units can be added without having to change the transport capacity, and therefore the instruction format.

5. Trivial decoding. One of the RISC design goals was to reduce decoding logic. To this respect, *MOVE* architectures take the ultimate approach; they support only one operation format.

6. Simplicity. Despite (or maybe because of) the range of flexibility and scalability, the *MOVE* architecture has a very simple and regular organization. RISC design principles are even more apparent than in well known RISC processors.

7. Hardware efficiency. Looking carefully at the *MOVE* architecture it appears that it uses its registers and its data transport capacity more efficiently. E.g. the data transport capacity is designed for average behavior, i.s.o. worst case. This becomes especially important when integrating intra-node parallelism onto a single chip.

8. The *MOVE* programming paradigm (programming transport) gives rise to undeveloped code scheduling degrees of freedom.

9. Last but not least, the systolic MIMD programming paradigm, which combines large throughput with very low latency inter node communication, naturally fits into the transport-triggered architecture model (see e.g. [CO91].

For a more extensive description and evaluation of the *MOVE* concept, see e.g. [CM91, HCM91, HC92, Cor92].

Architecture overview 3

This chapter presents an architecture overview of *MOVE32INT*. It gives a quick insight in the function and capabilities of this architecture. Following sections pay attention to the operational model of *MOVE32INT*, the connection between function unists (FUs) and the transport network by way of sockets, the pipelining model, supported data types, allignment restrictions, and list the major limitations of this architecture in comparison to current RISC processors. Chapters 4 and 5 treat the instruction formats, available operations and the support mechanisms (like exceptions) in more detail.

3.1 Operational model of *MOVE32INT*

The architecture consists mainly of a data transport and control network controlled by the bus controller, and a couple of function units (FUs). The network contains 4 busses; each bus contains a data bus which is capable of transporting one data value of 32 bits, a identifier bus which transports one move operation specifier of 16 bits (as shown in figure 2.5), and a control bus which contains a few control signals. This means that four data transports (*moves*) can be handled in parallel. Therefore, each instruction has to specify four move operations. Figure 3.1 shows a operational view of the *MOVE32INT* architecture. As indicated, *MOVE32INT* uses a harvard architecture with separate address and data paths to instruction and data memory. *MOVE32INT* contains many FUs. Most units contain Trigger, Operand, Result, and Status registers; some units contain also Control registers (the first letter of a register indicates its type). E.g., for the integer units these registers are TI_i, OI_i, RI_i, and SI_i respectively. The status registers are used to access the valid bit V of the result register (see figure 2.6). Access to this valid bit is necessary for exception purposes, as described in chapter 5.

Looking into figure 3.1 the following FUs are distinguished:

1. Register units. *MOVE32INT* contains 11 general purpose registers, R0-R9, and SR. The latter

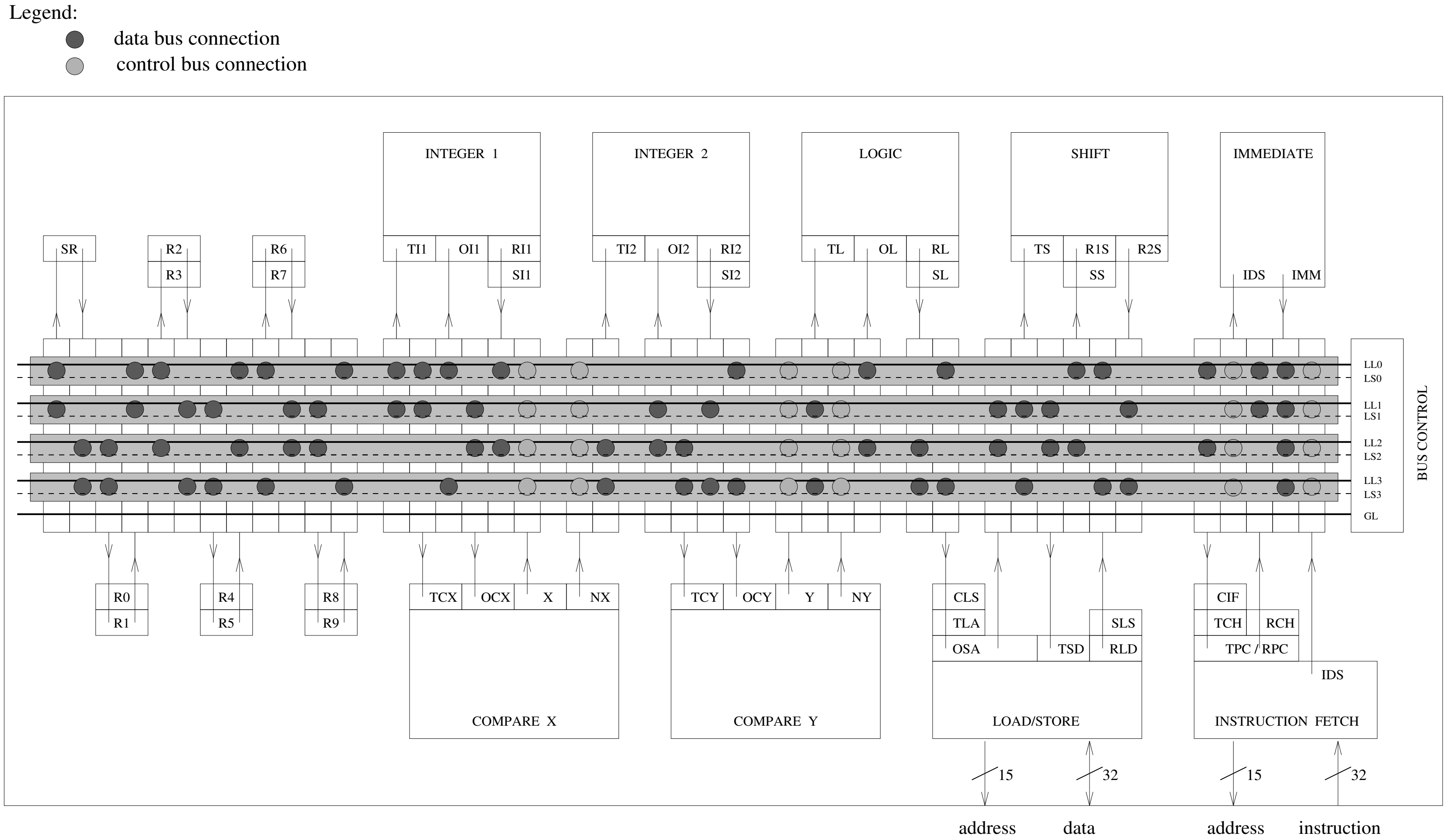


Figure 3.1: Operational view of *MOVE32INT* architecture.

register is also scannable, using the builtin scan path.

2. Two integer units for add and subtract operations.
3. Logic unit, performing bitwise and, or and exor operations.
4. Shift unit for arithmetic and logical shift-1 and shift-2 operations. Both shifted results are simultaneously available in two separate result registers (R1S and R2S).
5. Immediate unit. Reading from this unit has the effect of putting the 6 bit source identifier of the move operation as a unsigned integer on one of the data transport busses. check unsigned!
6. Two compare units. They produce boolean result values (X and Y and their complements) which can be read from the result registers. Their output is not connected to the data bus, but to the control bus, and is used to guard (conditionalize) the data transports.
7. Load & Store unit. This unit contains a trigger and result register for the load part, and a trigger and operand register for the store. This unit is connected to the external data memory, using a 32 bit data path, and a 15 bit address path; this allows for a maximum data memory of 32k words, or 128 kbyte. The timing of the memory interface is configurable through the control register CLS.
8. Ifetch unit. This is basicly a load unit, where the address (the PC) is auto incremented. However writing to the PC forces a jump. Reading from the PC gives the current instruction address plus 2. The ifetch unit is connected to a internal instruction cache (using a 64 bit data path) and the check plus 2 external instruction memory (using a 32 bit data path). The maximum accessible size of the external instruction memory is 32k words, or 128 kbyte. The timing of the memory interface is configurable through the control register CIF.

   The instruction register is scannable. To support exceptions, the PC-chain is accessible through registers TCH and RCH.

MOVE32INT contains a very small (256 byte) direct mapped instruction cache (inside the instruction fetch unit), containing 16 lines of 128 bits (excluding tags). Each line contains 2 64-bit instructions.

On one side of the transport network we see the Bus Control unit. This unit is responsible for generating proper lock, guard and exception signals.

3.2 Connections between FUs and the network

A move bus consists of a data, identifier and control bus, which are all connected to the socket. The instruction decoding is performed locally, inside the sockets; it checks if the socket is selected, and if so, it extracts an opcode from the source identifier on the identifier bus and signals the function unit with this opcode.

As explained in section 2.5 TTAs do not have to be fully connected (although that's easier for the compiler). That means that some or all registers are not connected to all the available busses. As shown in figure 3.1 most registers (both general purpose and FU registers) are only connected to two of the four available move busses. Further, multiple registers may share one socket; in that case simultaneous usage of these registers is not possible[1]. Connections are choossen such that

[1] Although it looks like all short immediates share one input and output socket, in reality there are separate sockets for each immediate, allowing concurrent usage of all 4 short immediates.

- the loading of the busses is well balanced, and
- supported connectivity is sufficient for most code fragments.

The implemented connectivity is tabulated in appendix A.

3.3 Pipelined execution of *moves*

Traditional RISC like architectures have streamlined their instruction set in such a way that they can be easily pipelined; e.g. they omitted complex addressing modes[2]. Many RISC processors use a 4 or 5 stage pipelining mechanism. Figure 3.2 shows a well known 5 stage pipeline execution mechanism. In the Ifetch stage an instruction cache lookup is performed. Tag comparison can be overlapped with the second pipeline stage. Transport of data values from and to registers is done during the RF and WB stages (register fetch and write back). The actual operations are performed during the Execute stage (and Memory stage for load and store operations).

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

Figure 3.2: RISC pipelining execution mechanism.

The actual data transport in *MOVE32INT* only takes 1 clock cycle. However, that does not mean that *MOVE* architectures are not pipelined. Function units are completely separated from the data transport network. Pipelining decisions for FUs, and transport network can be taken independently.

In *MOVE32INT* Icache lookup, decode and the actual data transport each take one cycle. The execution of move instructions is therefore done in three stages:

1. Ifetch stage. During this stage the instruction register (IR) is fed with the next instruction from the instruction cache:

   $$IR := icache[PC]$$

2. Decode stage. Source and destinations identifiers are transported to the sockets and decoded locally (inside the sockets).
   A possible instruction cache miss is reported during this stage.

3. Move stage. In this stage the actual data is transported from the source to the destination registers. FUs may request a lock as a consequence of these data transports. Guards are evaluated during this stage; this may result in squashing of one or more data transports.

The resulting pipelining picture is shown in figure 3.3.

Comparison with the RISC pipeline gives the following observations:

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

Instruction					
i	ifetch	decode	move	possible fu operations	
i+1		ifetch	decode	move	possible fu operations
i+2			ifetch	decode	move

Figure 3.3: *MOVE* pipelining execution mechanism.

1. For the instruction fetch the same amount of work has to be done. Because *MOVE* architectures aim at much higher clock frequencies, the design of a fast Icache becomes much more important. To avoid many pipeline stages in the Ifetch unit fast sensing techniques have to be applied[3].

2. RF and WB are explicitly programmed.

3. Although decoding *move* operations is trivial, a second stage is needed for the transport of the source and destination register identifiers. Including this transport into the actual *move* stage would unnecessarily stretch the clock period.

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

3.4 Supported Data Types

MOVE32INT only supports 32 bit data and small 6 bit unsigned immediates; immediates are not sign extended. Bytes and halfwords are not supported in hardware. Because of this, there is no difference between the Big and Little Endian representation of data at the architecture level (there is at the software level !). The arithmetic units (integer and shift units) do not support overflow exceptions. As a consequence, there is no difference between unsigned en 2'complement operations.

3.5 Memory Allignment Restrictions

MOVE32INT does not support byte or halfword addressing. Therefore no data allignment problem exists for 32 bit data words. When loading or storing values, the address resolution is at word boundaries; that is, the lower 2 address bits are not ignored.

Instructions have a length of 64 bits and have to be alligned at even word boundaries. They have to be stored into memory in a Little Endian way; that means, the least significant word at the lower (even) address (see figure 3.4)[4].

In writing to the pc, the lower 3 bits are not ignored, that is, the pc address resolution is at double word boundaries.

[3]The Icache lookup problem is strongly reduced in *MOVE32INT*, because the Icache only implements a few lines.

[4]The order of the 16-bit moves inside an instruction is choosen little Endian as well; this is important to the compiler and also in case other processors can access the memory at half word boundaries.

		mshw	lshw
address	0	move 1	move 0
	1	move 3	move 2
	2	move 1	move 0
	3	move 3	move 2

Figure 3.4: Instruction endianness.

3.6 Architecture limitations

In accordance with the design goals of *MOVE32INT*, as stated in the introduction, the architecture is rather limited. Many restrictions result from area limitations. Remember that *MOVE32INT* has been implemented in conventional 1.6 μ CMOS SoG technology; the number of transistors was fixed to about
check 180k 180k. Major restrictions, in comparison with state of the art RISC processors, are:

1. Limited number of registers. This is partly circumvented by the possibility of using FU pipeline stages as temporary storage, and by the fact that MOVE architectures require less registers.
2. Limited cache support. *MOVE32INT* only provides a small Instruction cache, and does not have a data cache.
3. Limited exception support (see chapter 5).
4. No virtual addressing and MMU support.
5. No protection support and kernel mode.
6. No multiprocessor support.
7. Restricted operation repertoire (see chapter 4).

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

Instruction formats and operation set

4

Each instruction for *MOVE32INT* contains 64 bits and specifies 4 operations, called *moves*. The next sections explain instruction and operation formats, summarize all available operations, and compare the reduced operation repertoire of *MOVE32INT* with those of more full blown RISC architectures.

4.1 Formats

The format for an instruction and a *move* operation are shown in figure 4.1. Move operations are formatted into an instruction in little endian order. It contains a 3 bit guard-id or guard specifier, a 7 bit

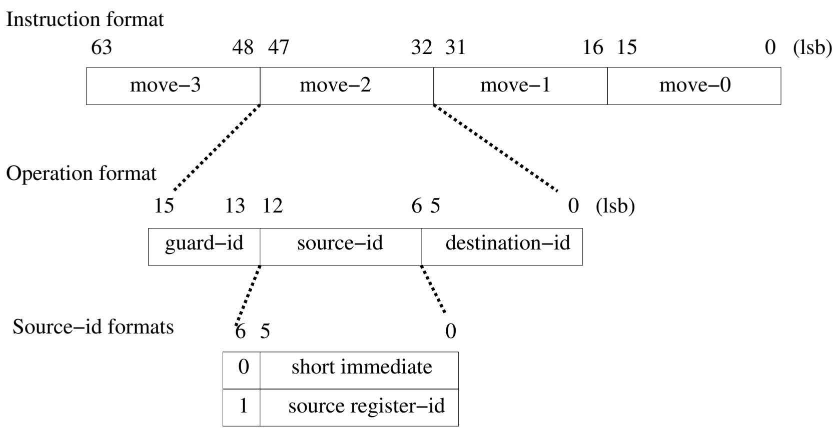


Figure 4.1: *MOVE32INT* instruction and operation formats.

source field and a 6 bit destination field. As shown in the figure, the upper half of the source-id space is consumed by short immediates; so the source field can be interpreted as a 1 bit immediate flag, and a 6 bit register-id or 6 bit immediate.

MOVE32INT supports a very general way of conditional execution; each move operation is conditional executed or guarded; this is in contrast to traditional architectures where only the control operations have a conditional execution mode. During each cycle 4 guards are produced, one for each transport bus. The guard determines if an operation has to be performed or has to be cancelated. The guard specifiers indicate how to evaluate the guards. Guarded execution is treated further in chapter 5.

The immediate flag determines the interpretation of the source field. This source field contains either a short 6 bit immediate, or a specification of a source register. The immediate is not sign extended when moved to a destination register.

4.2 Available operations

MOVE architectures essentially contain one type of transport operation, the *move*. However, to be suitable for general applications, a complete set of arithmetic, logical, load, store and other operations have to be implemented. Those data operations are triggered (specified) by moving a value to a particular trigger register. Multiple logical registers may map on the same physical register. The programmer only specifies logical registers, however he has to be aware of their physical mapping. E.g. the triggers for the add and subtract operations share one physical register; therefore it is not allowed to issue an add and subtract operation simultaneously to the same FU. Details of this mapping, and register identifiers are given in appendix A.

Tables 4.1 and 4.2 give an overview of all the available logical registers and shortly describes their function.

Check subtract T-O / O-T

check SAX

check 6-bit offset

Six types of registers are distinguished: triggers (T), operands (O), results (R), status (S), control (C) and boolean (B) registers. Valid bits are read through the status registers; they are mapped onto bit 31. Boolean registers x, nx, y and ny are connected to the guard (control) bus. The details of the control registers (used to control the timing behavior of the ifetch and load-store units) are treated in appendix C.

The tables 4.1 and 4.2 also determines the naming convention used in assembly language. E.g. to specify an arithmetic shift right operation on the result of the first integer unit requires the *move:*

$$ir1 \rightarrow asr$$

In table 4.3 the pipeline values for the parameters l (latency), d (stage distance), s (number of stages), and t (throughput) are listed with some remarks (see section 2.2 for a discussion of these parameters. The listed number of stages s includes a possible result register.

As indicated, the ifetch unit implements 2 delay slots (a direct consequence of the 3-stage transport pipeline). This means that the 2 successor instructions of a jump are always executed.

Careful readers will notify that the throughput and distance specified for the load-store unit is not in accordance with $t = 1/d$. Actually the throughput is larger as expected. This is a consequence of partially using *wave pipelining* in the load-store memory interface.

check numbers for store

Function unit	Register	Type	Operation
General purpose registers	R0 - R10	T	identity
	R0 - R10	R	
Integer unit 1	add1	T	addition
	sub1	T	subtract: R := T - O
	io1	O	
	ir1	R	
	ir1v	S	reading valid bit
Integer unit 2	add2	T	addition
	sub2	T	subtract
	io2	O	
	ir2	R	
	ir2v	S	reading valid bit
Logic unit	and	T	bitwise and
	or	T	bitwise or
	xor	T	bitwise exclusive or
	lo	O	
	lr	R	
	lrv	S	
Shift unit	asr	T	arithmetic shift right
	lsr	T	logic shift right
	sl	T	shift left[1]
	sr1	R	shift-1 result
	sr2	R	shift-2 result
	srv	S	reading valid bit
Immediate unit	#0-#63	R	reading unsigned short immediate

Table 4.1: Function units, (logical) Registers and their operations.

4.3 Comparison with full blown RISC architectures

It will be clear from the former section that *MOVE32INT* only supports a limited set of operations as compared to current RISC processors. This is in accordance with the goals formulated in the introduction. What are the operations we did not support? The next list shows the major operations we omitted in *MOVE32INT*:

1. Unsigned addition and subtraction. The difference between signed and unsigned integer operations lies in the generation of overflow exceptions. Our exception model (see chapter 5) is very limited, and does not support overflow exceptions.
   Overflow detection has to be done in software using for

   addition $c := a + b: \quad overflow = (sign(a) = sign(b)) \wedge (sign(a) \neq sign(c))$
   subtraction $c := a - b: \quad overflow = (sign(a) \neq sign(b)) \wedge (sign(a) \neq sign(c))$

Function unit	Register	Type	Operation
Compare unit X	xeo	T	test if trigger equal zero; x := (T=0)
	xeq	T	test if trigger equal operand
	xls	T	test if trigger less than zero
	xgt	T	test if trigger greater than zero
	xo	O	
	x	B	result of compare
	nx	B	negated result of compare
Compare unit Y	yeo	T	test if trigger equal zero; y := (T=0)
	yeq	T	test if trigger equal operand
	yls	T	test if trigger less than zero
	ygt	T	test if trigger greater than zero
	yo	O	
	y	B	result of compare
	ny	B	negated result of compare
Load-Store unit	la	T	load address; Result := MEM(la)
	sd	T	store data; MEM(sa) := sd
	sa	O	store address
	sa	R	address can be read for state saving
	ld	R	
	ldv	R	reading valid bit
	ctl	C	control of external timing behavior
Ifetch unit	pc	T	jump; PC := T
	pc	R	save PC
	pco	T	page relative jump, offset 6-bit; PC := PC(31-6)‖T(5-0)
	ex	T	jump with exception
	exo	T	page relative jump with exception, offset 6-bit
	pcc	T	write PC-chain
	pcc	R	read PC-chain
	cti	C	control of external timing behavior

Table 4.2: Function units, (logical) Registers and their operations (continued).

See appendix C of [MIP88] for assembly examples. Compare this to hardware, where a simple exor of the 2 most significant carry bits suffices.

2. Add and subtract with carry. Like on the MIPS we have to work arround this omission using a software approach. See appendix C of [MIP88] for assembly examples. A problem is that we also do not support SLTU (set less than unsigned). However an easy work arround exists (see item full compare).

3. Restricted shifts. Only 1 and 2 bit static shifts are supported. Larger shift amounts and dynamic shifts (like SLLV on MIPS) have to be programmed.

4. Full compares (e.g. $A \leq B$) have to be done with the subtract operation. Make a distinction between signed and unsigned compares.

Function unit		l	d	s	t	**Remarks**
General Purpose Registers		1	-	1	1	
Integer units		3	2	2	1/2	
Logic unit		2	1	2	1	
Shift unit		2	1	2	1	
Immediate unit		0	1	0	1	numbers not really applicably for this unit
Compare units		2	1	2	1	
Load-Store unit[a]	load	4	3	2	2	assuming loads after loads [b]
	store	-	-	1	2	assuming store after stores
Ifetch unit[c]		3	1	3	1	all jumps have 2 delay slots 3 stages in the PC-chain

[a] Assuming fast static memory.
[b] Loads after Store and visa versa take longer.
[c] Assuming instruction cache hit.

Table 4.3: Delays of function units.

SLT a,b: *if sign(a) = sign(b) then result = sign(a-b)*
else result = sign(a)
SLTU a,b: *if sign(a) = sign(b) then result = sign(a-b)*
else result = sign(b)

5. Halfword and byte support. Most architectures support these data formats with their load and store operations. We intended to limit this kind of support to byte/halfword insert/extract and sign extend function units. However, because of time limitations, even these function units are not implemented. Shift and logic units have to be used to support these data types.
6. Not operation (1's complement). This one can be implemented using exor with all ones.
7. No complex operations like multiply, divide and floating point.
8. Jump-and-Link (or Call) instruction. Use separate *moves* to read and write the PC instead; e.g.:

#instr.	move instruction
i	#label $\rightarrow$pc; pc $\rightarrow$R1
i+1	delay slot
i+2	delay slot
Label:	first instruction of the called function

Remark that R1 contains return address i+2, which has to be incremented before returning from the called function[2].

9. Cache flush instructions. However, we do support an external cache reset signal, which invalidates all the cache entries.
10. Kernel support. There are no priviledged instructions.

[2] Address i+2 does not have to be the successor of i+1 if e.g. instruction i-1 contained a branch.

Support mechanisms 5

A real processor needs additional mechanisms to support high-level languages and operating systems. We distinguish three support mechanisms:

1. Guarding, which allows conditional execution.
2. Locking, for handling nondeterministic events.
3. Exceptions, making the processing interruptible.

These three support mechanisms are described in the following sections.

5.1 Guarded execution

Guards support a very general way of conditional execution. In *MOVE32INT* not only operations which change the program counter can be conditionalized, but every *move* can independently be specified to be executed conditional. To this purpose, each data transport bus is accompanied with a so called *guard* bus (single bit), which controls the occurrence of the actual data transport. If a guard signals a 'false' value, the corresponding data transport is squashed.

As already shown in figure 4.1, each move operation specification includes 3 bits for the guard specifier, or guard-id. This guard-id is essential interpreted as a source-id for the guard signal; it specifies which booleans put their value on the guard bus. In *MOVE32INT* two booleans, the outputs X, NX, Y and Y of the compare units, are connected to the guard busses. It is also possible to specify boolean expressions of these booleans as guard conditions[1]; table 5.1 lists all possible implemented guard expressions, and defines how to use them in assembler code.

[1]Using dynamic guard busses, expressions of booleans could be easily implemented as multicast reads!

Guard specifier	Boolean expression	Assembler notation
0	x	x: src →dst
1	$\overline{x}$	!x: src →dst
2	y	y: src →dst
3	$\overline{y}$	!y: src →dst
4	$x \wedge y$	x.y: src →dst
5	$\overline{x} \wedge y$	!x.y: src →dst
6	$x \wedge \overline{y}$	x.!y: src →dst
7	TRUE	src →dst

Table 5.1: Guard specifier

Guard expressions give the compiler extra scheduling freedom in moving instructions from branches (and even nested branches) up to a position where the branch has not yet been taken, but the condition is already known. This enhances the effective utilization of the available hardware resources in *MOVE32INT*.

An example move instruction could look like:

$$x.!y : ir1 \rightarrow asl; !y : r2 \rightarrow pc; x.y : r5 \rightarrow add1; \#12 \rightarrow io1$$

Remark that the fourth *move* is executed unconditionally. Guarded execution allows also the specification of multiway branching, as e.g. in:

$$x : \#33 \rightarrow pc; y : \#12 \rightarrow pc; ....$$

This instruction has 3 possible successors[2]. Remark however that the compiler has to assure that conditions x and y are mutually exclusive!

5.2 Lock mechanism

Compile-time synchronization of all operations is not possible when FU latencies cannot be determined during compile time (e.g. cache hit versus cache miss) and is not efficient when insufficient operations can be found to bridge the latency in case of RaW hazard. Both problems can be solved by synchronizing the dependent operations in hardware. In general, synchronization implies that the supplier of data locks the consumer until the data is ready.

As explained in subsection 2.3.3, we have to distinguish *FU locking* and *transport locking*.

5.2.1 Transport locking

Within *MOVE32INT* locking of the transport network reduces to two types of locks:

1. *Read lock*; a result is still pending. This allows the scheduling of a result read independent of the FU latency.

[2]Connectivity restrictions in the data transport network do not allow for more than 3 pc destinations; see appendix A.

Most FUs, except general purpose registers, compare units and immediate unit, may request a read lock.

2. *Write lock*; a FU is not yet available, but will be available in the near future[3].
This lock may only be requested by the load/store FU and integer FUs.

Locks are requested to, and possibly granted by, the *Bus control unit* (see figure 3.1). Most lock requests have *local scope*; this means that they correspond to a particular transport bus. A local lock on bus_i can be masked by a squash on bus_i. The ifetch and compare units can request *global locks* which can not be masked by squashing; e.g. when a cache miss appears.

Locking stretches the move cycle until the locking condition disappears. Figure 5.1 shows the insertion of a *stall* cycle in *MOVE32INT* transport pipeline when one of the *moves* of instruction i encounters a lock. This lock could also be generated by instruction $i + 1$ because of an instruction cache miss. The latter lock is detected during the decode stage (instead of the move stage).

instruction:						
i	ifetch	decode	*stall*	move		
i+1		ifetch	*stall*	decode	move	
i+2			*stall*	ifetch	decode	move

Figure 5.1: The insertion of a *stall* cycle caused by locking.

All possible lock requests are summarized in table 5.2.

FU	**Type**	**Scope**	**Remarks**
Registers	-	-	
Integer units	write	local	both on trigger and operand register
	read	local	on result register
Logic unit	read	local	on result register
Shift unit	read	local	on result register
Immediate unit	-	-	
Compare units	read	global	
Load/Store unit	write	local	on load and store address, and on store data
	read	local	on load from result register
Ifetch unit	read	global	reading PC which is not yet incremented
	read	global	cache miss; lock during ifetch stage

Table 5.2: Lock requests.

The following remarks apply:

1. As probably expected, register and immediate units never request locking.

2. FUs which have a stage distance $d = 1$ do not request write locks.

[3] A FU, whose stages are all occupied, does not generate a write lock upon writing to trigger and/or operand. This is considered an exceptional case.

3. Most FUs, except for the compare units, request read locks, even when the pipeline is empty.

4. Write locks are not requested when the pipeline is full. In that case trigger and/or operand registers are simply overwritten.

is this correct for load/store unit as well?

5. The compare unit does not request read locks in case the pipeline stages are all empty. This means that the boolean result values can be read multiple times; the same boolean can be used to guard multiple operations. The following program fragment illustrates its behavior:

Instr#	Instruction		Comments
1	gr1 → xeo	......	
2	gr2 → xeo	......	
3	x: gr3 → add1	......	uses x-result from instruction 1
4	x: gr4 → and	......	uses x-result from instruction 2
5	x: gr4 → or	gr4 → xeo	uses x-result from instruction 2 again
6	x: gr3 → sub1	......	lock request (1 cycle); uses x-result from instruction 5

The lock request generated by instruction 6 can (of course) not be masked by a squash, and has therefore global scope.

6. The number of cycles during which the load/store and ifetch unit request locking depends on the speed of the external memory (see appendix C for details).

7. The ifetch unit requests global locks in 2 situations.

   (a) If the PC is not incremented in time; this happens when an carry occurs from bit 6 to bit 7 during PC-increment. The lock is requested if PC+1 is needed; either for instruction fetch, or if the programmer reads the next PC.

   (b) An instruction cache miss leads to several lock request cycles.

5.2.2 FU locking

All multi stage FUs implement the hybrid pipelining concept as explained in section 2.2. This means that intermediate results do not continue to the next pipeline stage if that next stage is already occupied with (intermediate) result from operations started earlier. The programmer can make use of this fact, by temporarily leaving results in the FU pipeline which can not yet be consumed by other operations, effectively using FU stages as temporary storage.

5.3 Exception support

General purpose CPUs change into *exception mode* upon acceptance of an exception request. In exception mode the processor can respond to abnormal or asynchronous events. After handling this event the processor state is restored, and processing continues at the interrupted point.

We distinguish three basic exception categories:

1. Precise exceptions. These exceptions require that the source operands of the excepting operation are available upon recovery from exception. Examples are loads, stores and floating point operations (according to the IEEE-754 floating point standard).

2. Imprecise exceptions. These do not require source operands to be available upon recovery. Examples are integer arithmetic exceptions [4].

3. Fatal exceptions. In contrast to the former two, these exceptions are not recoverable.

Exceptions from the first two categories are called restartable; the processor state can be recovered before entering the exception routine and restored afterwards.

Three kinds of exceptions can occur in the *MOVE32INT* processor:

Reset Asynchronous, external exception. The program counter starts at location 0x40 after a reset[5]. check reset address

Interrupts Asynchronous exceptions, caused by (external) hardware (interrupts).

Traps Synchronous exceptions, caused by a trap instruction. Traps take precedence over interrupts, when they occur in the same cycle.

MOVE32INT does not support full and empty exceptions as discussed in section 2.3[6],nor does it support arithmetic or other operation related exceptions.

When encountering an interrupt or trap, the instructions which are currently in the ifetch and decode stages are squashed, the PC chain is stopped after finishing the instruction which is in the move stage, and the processor enters *exception mode*.

The program counters in the PC chain have to be saved when nested exceptions are required. Further details about the handling of traps and interrupts can be found in appendix D.

[4]However, some Lisp systems could use precise integer arithmetic. Upon arithmetic overflow of fixnum operations the representation of integers could be changed from fixnum to bignum.

[5]Locations 0 - 0x3F are used as exception frame. These are the only locations which can be accessed directly using short immediates.

[6]Potential occurrence of these exceptions can easily be verified by a software code parser.

External interface 6

This chapter describes how *MOVE32INT* has to be interfaced to the external world. First the external connections are described, next the timing behavior, and finally the use of the builtin scan path. Pin layout and other chip details can be found in appendix E.

6.1 External connections

The external appearance of *MOVE32INT* is shown in figure 6.1. The following signals are distinguished:

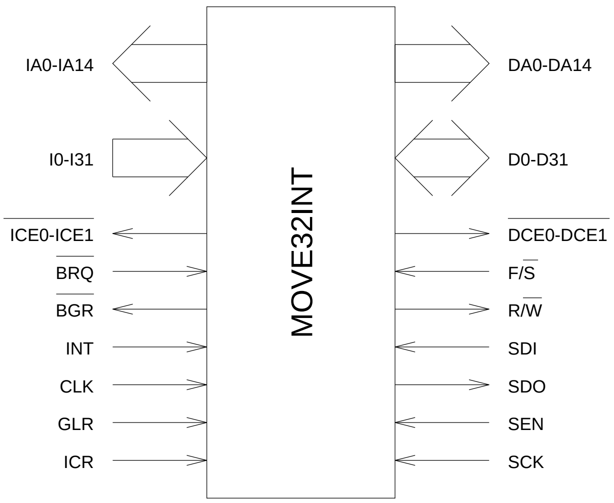


Figure 6.1: External view of *MOVE32INT*.

1. Separate address lines for instruction and data ($IA0-14$ and $DA0-14$).
2. Separate data lines for instruction and data ($ID0-31$ and $DD0-31$).
3. Control lines:
   (a) $\overline{BRQ}$ and $\overline{BGR}$: bus-request and bus-grant. When bus-grant is active, all signals (except bus-grant) are in tristate mode.
   (b) INT: external interrupt request.
   (c) GLR: global reset.
   (d) ICR: instruction cache reset. Makes all entries invalid.
   (e) $F/\overline{S}$: fast slow mode. Influences the timing of the data memory interface.
4. Strobe lines:
   (a) $R/\overline{W}$: read write control for data memory.
   (b) $\overline{ICE0-ICE1}$ and $\overline{DCE0-DCE1}$: two separate write enable signals are necessary in order to accommodate the interface with a 16 bit host processor bus (like in an IBM-AT). The host can enable the lower and upper halfwords of a memory instruction or data word independently. *MOVE32INT* enables both lines simultaneously.
5. Scan path:
   (a) SDI: scan data in.
   (b) SDO: scan data out.
   (c) SEN: scan enable.
   (d) SCK: scan clock.
6. CLK: general (asymmetric) clock.
7. Power and Ground lines. These are not shown in the figure; see appendix E instead.

6.2 Timing behavior

The maximal clock frequency is 80 MHz, assuming a duty cycle as shown in figure 6.2. The delay t_{cc}

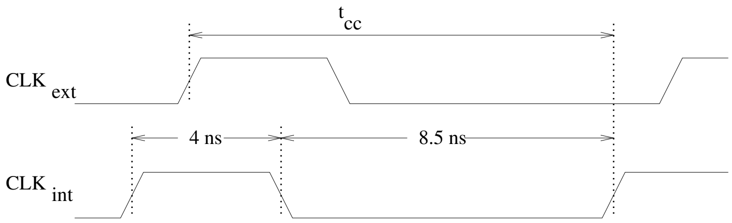


Figure 6.2: Duty cycle of clock signal at maximal frequency.

between external and internal clock is caused by pad delay and clock buffering. Further timing diagrams will only refer to the internal clock.

In the next subsections the following timing issues will be described (possibly by timing diagrams):

1. Global and icache reset timing.
2. Bus-request and bus-grant.
3. Interrupt handling.
4. Load-store interface timing: Load after Load, Load after Store, etc.
5. Ifetch unit interface timing.

6.2.1 Global and Icache reset

Global reset GLR has to be low for at least 3 cycles for resetting the processor (bringing the processor to a known state). All stage registers of the FU pipelines are valid after a global reset, and have to be emptied first. The program counter starts at address 0x40.

In order to reset all valid bits of the icache, the line ICR has to be kept low for at least 2 cycles.

6.2.2 Bus-request

Figure 6.3 shows what happens when $\overline{BRQ}$ gets active (low). The number of clock cycli between a bus-

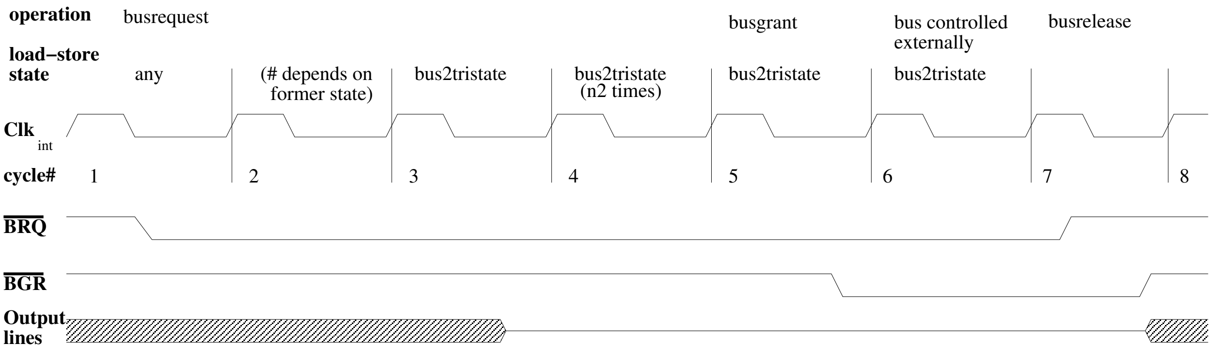


Figure 6.3: Bus-request timing diagram.

request and a bus-grant depends on current state of the load-store unit (and ifetch unit). The maximum number of cycli is $3 + N2 + N3 + N4$, where N_i is determined by the value of the load-store control register (maximum $N_i = 8$).

Is bus grant signal controlled by both ifetch and ls unit?

6.2.3 External interrupt

An external interrupt can be issued at any time. Details about how an external interrupt is handled by *MOVE32INT* are described in appendixapp:exception.

Remark that no interrupt acknowledge is given. This has to be controlled by software.

6.2.4 Load-store interface timing

This section describes the timing of load and store operations for certain important sequences: load after load, a single store, and store after store. The timing of the load-store unit is controlled by the 4 values,

$N_1..N_4$. These values can be set by writing to the load-store control register (see appendix C for details).

Load after Load

In fast mode the maximal load throughput is 1 load per 2 cycles. The load latency is 4. E.g. in the following program the load unit is kept optimal busy:

Instr#	Instruction	Comments
0	scr $\rightarrow$la	start first load
1	delay slot	
2	scr $\rightarrow$la	start second load
3	delay slot	data of first load is latched in load data register
4	scr $\rightarrow$la ld $\rightarrow$dst	start third load; read first load
5	etcetera	

The corresponding timing diagram is shown in figure 6.4.

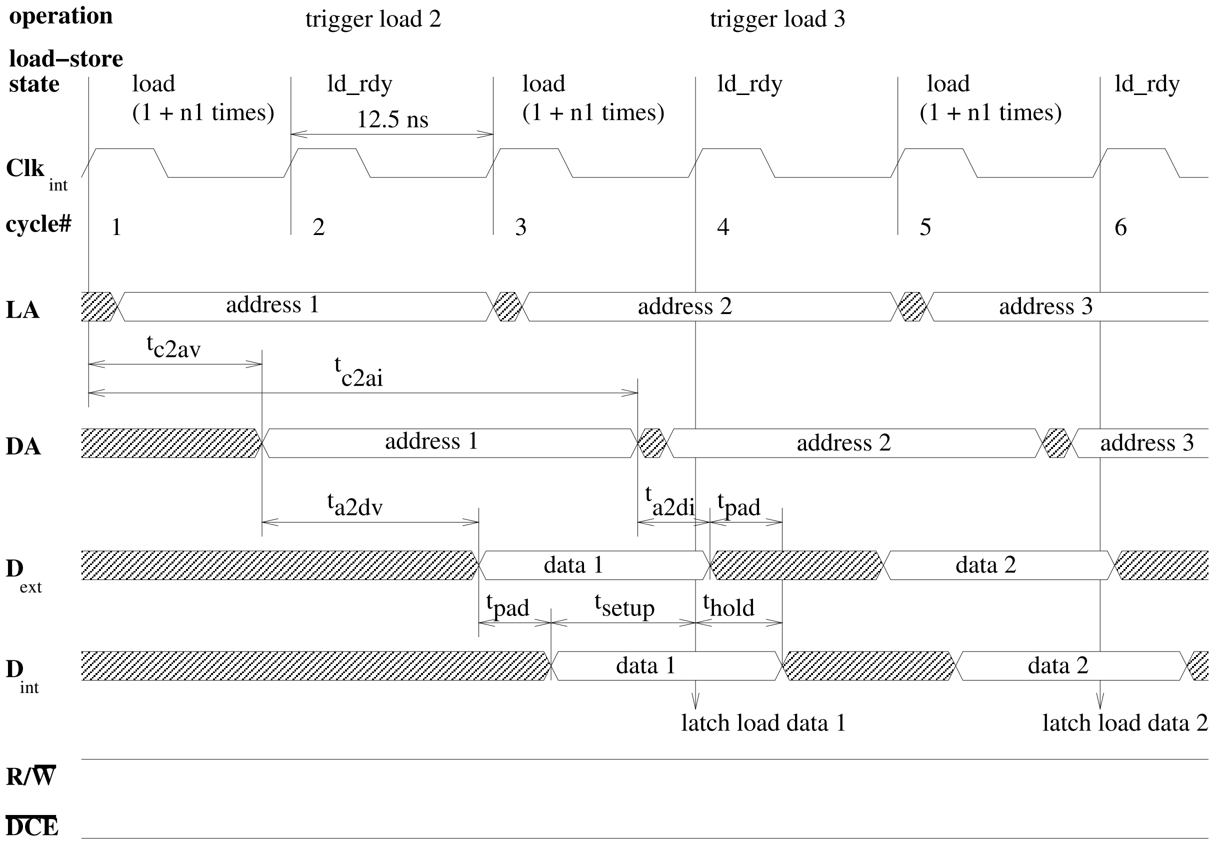


Figure 6.4: Timing of loads after loads.

The figure shows: 1) external clock, 2) internal data clock, 3) internal load address LA, 4) external data load address DA, 5) read/write signal, 6) data chip select, 7) external data lines D, and 8) the internal data lines. We assume a cycle time of 12.5 ns, and fast 12 ns access time static RAM data memory (the processor functions in fast mode). Note that counter n1 controls the load access time; in this figure its value is taken 0.

Further the following timing parameters are defined:

Parameter	**Description**	**Range (in ns)**
t_{c2av}	clock to address valid	+/- 8 - 11
t_{c2ai}	clock to address invalid	+/- 32 - 35
t_{a2dv}	address to data valid	memory dependent, access time
t_{a2di}	address to data invalid	memory dependent, data hold time
t_{pad}	pad delay	+/- 4 - 5
t_{setup}	latch setup time	> +/- 1
t_{hold}	latch hold time	> +/- 1

Times have to be verified

Note that the memory device is default in read mode, and selected continuously ($\overline{DCE} = \overline{CS} = low$, $\overline{OE} = low(fixed)$, and $R/\overline{W} = \overline{WE} = high$). The data memory designer has to make sure that setup and hold times specifications are met. Therefore:

$$t_{setup} = 3 \times t_{clock} - t_{c2av} - t_{pad} - t_{a2dv} \geq 1ns$$

$$t_{hold} = t_{c2ai} + t_{pad} + t_{a2di} - 3 \times t_{clock} \geq 1ns$$

rewritten as:

$$t_{a2dv} \leq 3 \times t_{clock} - t_{c2av} - t_{pad} - t_{setup}$$

$$t_{a2di} \geq 3 \times t_{clock} + t_{hold} - t_{c2ai} - t_{pad}$$

The indicated values result the following constraints for the memory used[1]:

$$t_{a2dv} \leq 20.5ns$$

$$t_{a2di} \geq 2.5ns$$

Single store

Figure 6.5 shows the timing of a single store, starting from the default *bus-to-input* state (memory in read mode). Counters n4 controls the store write time, and n3 the number of cycles needed to change the bus from read to write mode.

The following parameters are defined:

[1]A test board for *MOVE32INT* uses static memory chips, CY7B199-12, which have $t_{a2dv} \leq 12ns$ and $t_{a2di} \geq 3$.

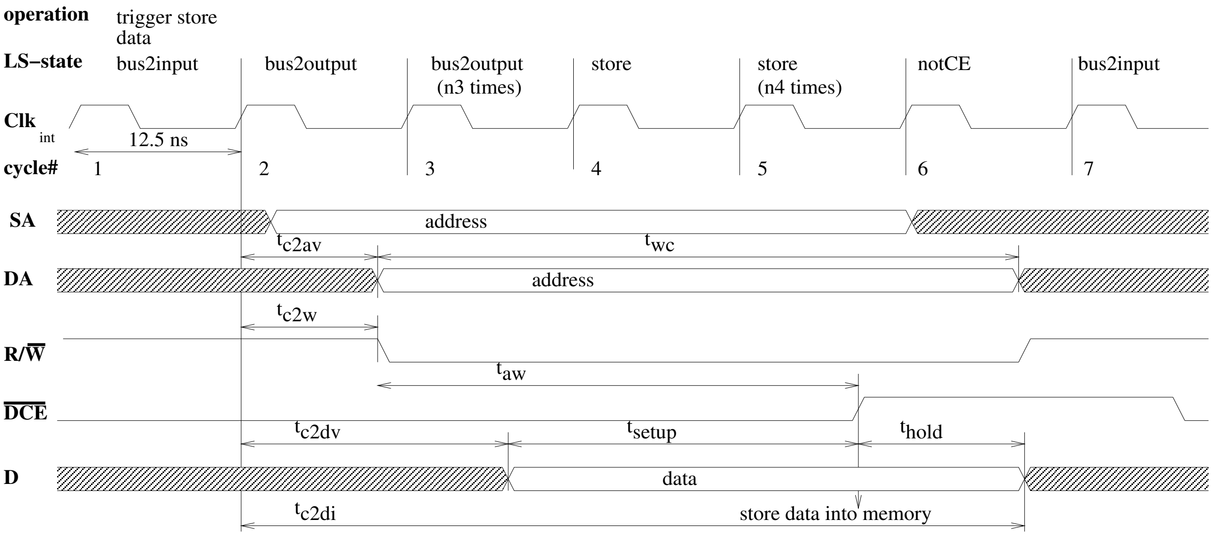


Figure 6.5: Timing of a single store operation.

Parameter	Description	Range (in ns)
t_{c2av}	clock to address valid	+/- 8 - 11
t_{c2w}	clock to write	+/- 8 - 11
t_{aw}	address setup to write end	-
t_{wc}	write cycle time; address valid	-
t_{setup}	data setup time	
t_{hold}	data hold time	
t_{c2dv}	clock to data valid	8 - 11 + $n3 \times t_{clock}$
t_{c2di}	clock to data invalid	8 - 11 + $(n3 + n4 + 1) \times t_{clock}$

The memory specifications may require that $t_{c2w} \geq t_{c2av}$, and further constrain t_{aw}, t_{wc}, t_{setup} and t_{hold} times! Designers have to verify these specifications.

Store after Store

The maximal store throughput is one store per 3 cycles. E.g. in the following program the store unit is kept optimal busy:

Instr#	Instruction	Comments
0	$\text{scr}_1 \rightarrow$sa $\quad$ $\text{scr}_2 \rightarrow$sd	start first store
1	delay slot	
2	delay slot	
3	$\text{scr}_1 \rightarrow$sa $\quad$ $\text{scr}_2 \rightarrow$sd	start second store
4	etcetera	

As shown, the data and address have to be written in the same cycle in order to keep the unit busy at full speed. The corresponding timing diagram is shown in figure 6.6. This figure assumes that in cycle 0 a store has just been finished (state = notCE), and delay counter n4 = 0.

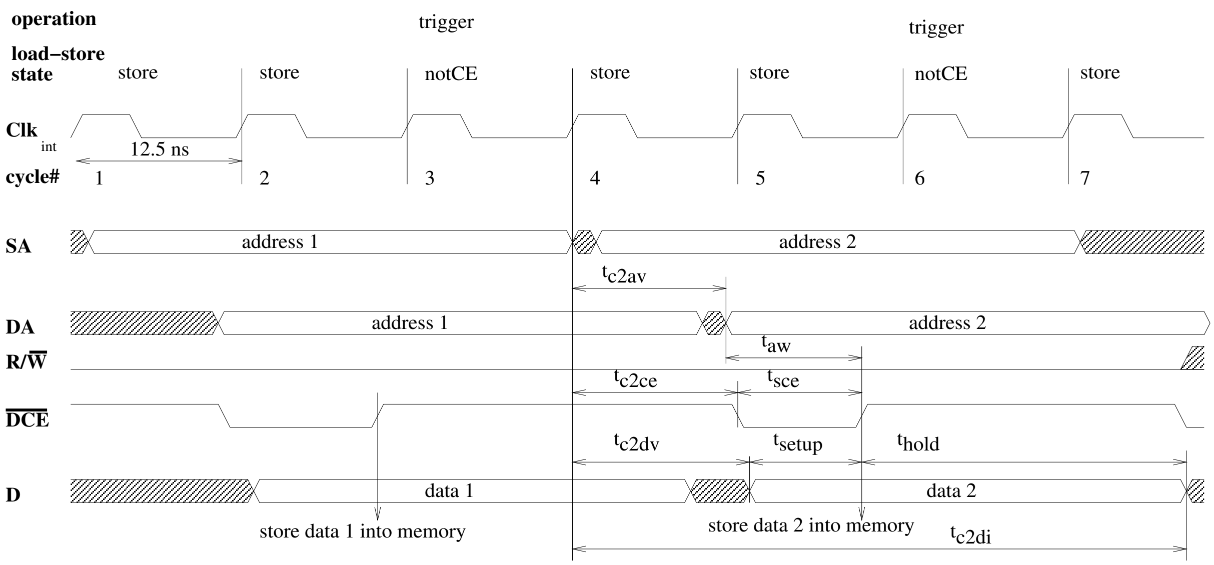


Figure 6.6: Timing of store after stores.

The following parameters are defined[2]:

Parameter	Description	Range (in ns)
t_{c2av}	clock to address valid	+/- 8 - 11
t_{c2ce}	clock to chip enable	+/- 8 - 11
t_{aw}	address setup to write end	-
t_{sce}	chip enable low to write end	-
t_{setup}	data setup time	
t_{hold}	data hold time	
t_{c2dv}	clock to data valid	8 - 11
t_{c2di}	clock to data invalid	45 - 49

Memory specifications may require that $t_{c2ce} \geq t_{c2av}$, and further constrain t_{aw}, t_{sce}, t_{setup} and t_{hold} times! Designers have to verify these specifications.

6.2.5 Instruction fetch interface

Upon an instruction cache miss, the instruction unit has to fetch instructions (in 32-bit chunks) from the external instruction memory. It uses an interface similar to the load part of the load-store unit, except that it does not rely on waving of loads (as was illustrated in figure 6.4). Therefore a load takes 3 instead of 2 cycles. What happens upon an instruction miss is shown in figure 6.7 The number of cycles needed to fetch half an instruction is controlled by a counter N; the minimum number of cycles is 3.

[2]Why is t_{c2di} so long!!. A possible change: 1) allow a single store cycle state and 2) make $\overline{DCE}$ high only in notCE state.

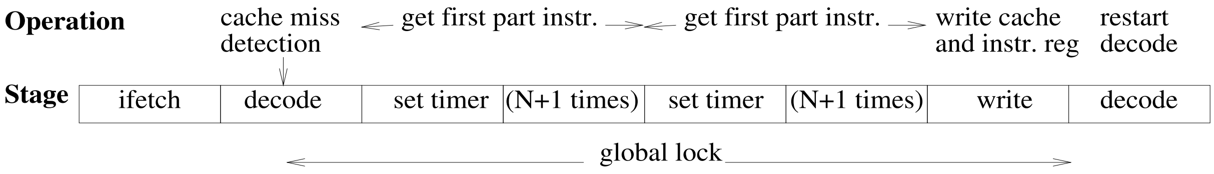


Figure 6.7: Loading an instruction after an Icache miss.

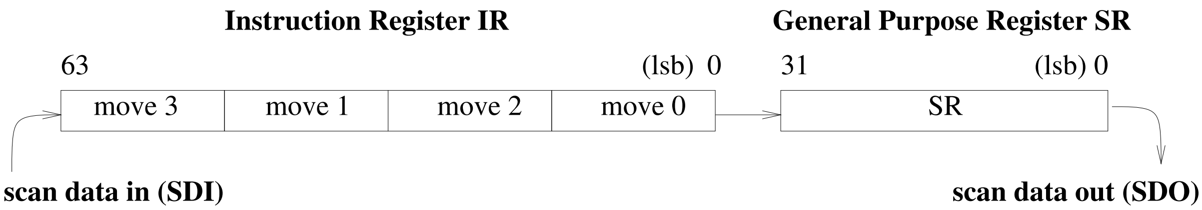


Figure 6.8: Scan path.

6.3 Scan path

For testing purposes *MOVE32INT* contains a scan path through two registers, a general purpose register SR, and the instruction register IR. The length of the scan path is therefore 96 bits. The scan path is shown in figure 6.8. Its timing behavior is illustrated in figure 6.9.

The following timing parameters are defined (cc denotes a processor clock cycle):

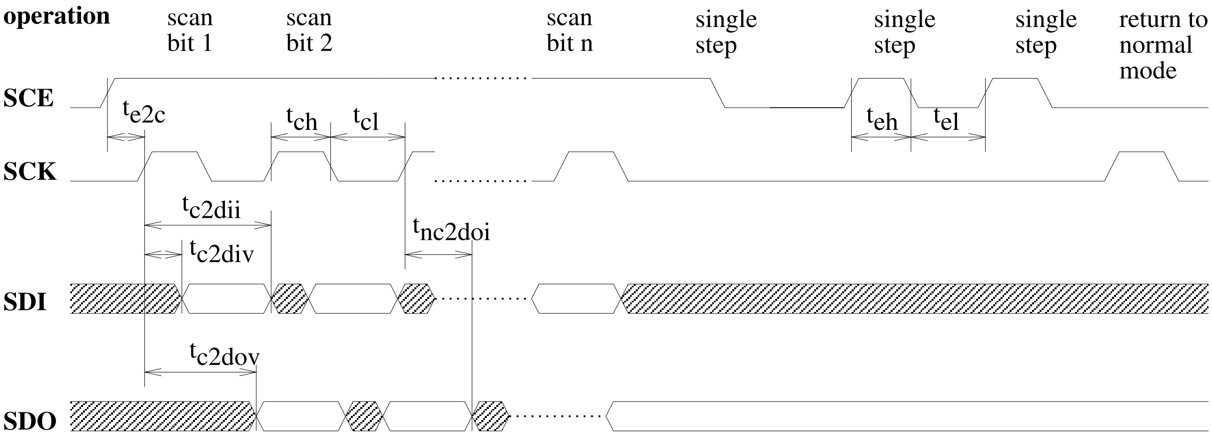


Figure 6.9: Timing behavior of the scan path.

Parameter	**Description**	**Range**
t_{e2c}	enable high to scan clock	> 1 cc
t_{ch}	scan clock high	> 1 cc
t_{cl}	scan clock low	> 1 cc
t_{eh}	scan enable high	> 1 cc
t_{el}	scan enable low	> 1 cc
t_{c2div}	scan clock to input data valid	< 2 cc
t_{c2dii}	scan clock to input data invalid	> 3 cc
t_{c2dov}	scan clock to output data valid	> 2 cc + 9 ns
t_{nc2doi}	next scan clock to output data invalid	> 2 cc + 8 ns

In scan mode, the processor is globally locked, using the existing locking logic. As illustrated in figure 6.9 the possible scan path commands are:

Scan enable	**Scan clock**	**Description**
$\uparrow$	0	scan path enabled; lock the processor
1	$\uparrow$	bit scanned in and out
$\downarrow$	0	single step
$\downarrow$	1	not defined
0	$\uparrow$	return to normal execution mode; unlock the processor

Bibliography

[CM91] Henk Corporaal and Hans (J.M.) Mulder. MOVE: A framework for high-performance processor design. In *Supercomputing-91*, pages 692–701, Albuquerque, November 1991.

[CO91] H. Corporaal and J.G.E. Olk. Design and Evaluation of Communication Processors supporting Message Passing in Distributed Memory Systems. In *DMCC-6 conference proceedings*, Portland, April 1991.

[Cor92] Henk Corporaal. Evaluation of Transport Triggered Architectures for General Purpose Applications. Technical Report submitted for publication, Delft University of Technology, Electr. Eng. Department, The Netherlands, August 1992.

[HC92] Jan Hoogerbrugge and Henk Corporaal. Comparing software pipelining for an operation-triggered and a transport-triggered architecture. In *Lecture Notes in Computer Science 641, Compiler Construction*, pages 219–228. Springer-Verlag, 1992.

[HCM91] Jan Hoogerbrugge, Henk Corporaal, and Hans Mulder. Software pipelining for transport-triggered architectures. In *MICRO-24*, Albuquerque, November 1991.

[i8689] Intel. *i860 64-bit Microprocessor Programmer's Reference Manual*, 1989.

[MIP88] Mips Computer Systems, Inc. *MIPS RISC Architecture, R2000/R3000*, 1988.

[RYYT89] R. Ramakrishna Rau, David W.L. Yen, Wei Yen, and Ross A. Towle. The Cydra 5 Departmental Supercomputer. Design Philosophies, Decisions and Trade-offs. *IEEE Computer*, pages 12–35, January 1989.

[Tra87] Trace: Technical summary. Multiflow Computer Inc., June 1987.

[vdA93] Paul van der Arend. Implementation and Realization of MOVE32INT. Technical Report in preparation, Delft University of Technology, Electr. Eng. Department, The Netherlands, 1993.

Connectivity and Register assignment

A

This appendix lists the implemented connectivity between registers and transport network and the actual register identifiers and their corresponding function.

A.1 Connectivity

Table A.1 lists all the available connections within *MOVE32INT*. It does so by listing for each input and output socket which registers and which busses it is connected to. check totals

Some remarks apply to this table:

- As shown in figure 3.1 TPC and RPC are physically the same register.
- Bus connections I, O and C indicate Input and Output connections to data bus and Control connection to identifier and control busses.
- Logical register are grouped; each group corresponds to one physical register on which these logical registers are mapped. Groups are listed in the same order as the corresponding physical registers[1].
- Essentially there are 4 separate immediate units, each connected to move-id and data bus. The immediate units do not contain (extra) physical registers; instead the decode registers in the output sockets are used to store (temporarily) the short immediates.

[1]An exception is the ifetch unit. Writing to the PC means writing to the PC-chain as well (at least in user mode). In reality two sockets are used which are accessed simultanously, using a multicast write.

Restrictions
A socket acts as a gateway which limits the throughput to one data value per clock cycle. This means however that instructions like

$$x : gr1 \rightarrow add1;\ \overline{x} : gr2 \rightarrow add1;\ ....$$

can still be specified, because the two conditions are mutually exclusive. The realization of *MOVE32INT* restricts such a usage to input sockets only. Therefore instructions like

$$x : ir1 \rightarrow gr1;\ ....;\ \overline{x} : ir1v \rightarrow gr2; ....$$

will give unreliable results. Of course,

$$x : ir1 \rightarrow gr1;\ ....;\ \overline{x} : ir1 \rightarrow gr2; ....$$

is still allowed.

A.2 Register mapping

Table A.2 lists the register mapping of all the logical registers. An X indicates a don't care. Source register 'ones' maps on all unused locations. Because precharged data busses are used all read data bits will be one (-1 in 2's complement notation).

The boolean registers X and Y are accessed by specifying appropriate guard specifiers. Their mapping is specified in table A.3

Socket	Physical Registers	Logic Registers	Bus 0	Bus 1	Bus 2	Bus 3
General Purpose Registers						
rgf0_in	R0, R1	(R0), (R1)			I	I
rgf0_out	R0, R1	(R0), (R1)	O	O		
rgf1_in	R2, R3	(R2), (R3)	I		I	
rgf1_out	R2, R3	(R2), (R3)		O		O
rgf2_in	R4, R5	(R4), (R5)		I		I
rgf2_out	R4, R5	(R4), (R5)	O		O	
rgf3_in	R6, R7	(R6), (R7)	I			I
rgf3_out	R6, R7	(R6), (R7)		O	O	
rgf4_in	R8, R9	(R8), (R9)		I	I	
rgf4_out	R8, R9	(R8), (R9)	O			O
rgf5_in	SR	(R10)	I	I		
rgf5_out	SR	(R10)			O	O
Integer unit 1						
int1_tin	TI1	(add1, sub1)	I	I		
int1_oin	OI1	(io1)	I			I
int1_out	RI1, SI1	(ir1), (ir1v)	O		O	
Integer unit 2						
int2_tin	TI2	(add2, sub2)			I	I
int2_oin	OI2	(io2)		I	I	
int2_out	RI2, SI2	(ir2), (ir2v)		O		O
Logical unit						
logc_tin	TL	(and, or, xor)		I		I
logc_oin	OL	(lo)	I		I	
logc_out	RL, SL	(lr), (lrv)			O	O
Shift unit						
shft_tin	TS	(asr, lsr, sl)		I		I
shft_out1	R1S, SS	(sr1), (srv)	O		O	
shft_out2	R2S	(sr2)		O		O
Immediate unit						
imm_out	—	#0-#63	O	O	O	O
Compare unit X						
cmpx_tin	TCX	(xeo, xeq, xls, xgt)	I	I		
cmpx_oin	OCX	(xo)		I	I	
cmpx_x	X	(x)	C	C	C	C
cmpx_nx	NX	(nx)	C	C	C	C
Compare unit Y						
cmpy_tin	TCY	(yeo, yeq, yls, ygt)			I	I
cmpy_oin	OCY	(yo)	I			I
cmpx_y	Y	(y)	C	C	C	C
cmpx_ny	NY	(ny)	C	C	C	C
Load & Store unit						
ldst_ain	TLA, OSA, CLS	(sa, sax), (la), (ctl)	I			I
ldst_din	TSD	(sd)		I	I	
ldst_aout	OSA	(sa)		O	O	
ldst_dout	RLD, SLS	(ld), (ldv)	O			O
Instruction Fetch unit						
ifet_in	TPC, TCH, CIF	(pc, pco, ex, exo), (pcc), (cif)	I		I	
ifet_out	RPC, RCH	(pc), (pcc)	O	O		
ifet_ctl	IR	—	C	C	C	C
Total Data Inputs			10	10	10	10
Total Data Outputs			8	8	8	8

Table A.1: Implemented connectivity in *MOVE32INT*.

Unit	Name	Source-id	Name	Destination-id
Registers	gr0	0110000	gr0	110000
		|		|
	gr10	0111010	gr10	111010
Integer1	ir1	000100X	add1	001000
			sub1	001001
	ir1v	000101X	io1	00101X
Integer2	ir2	000110X	add2	001100
			sub2	001101
	ir2v	000111X	io2	00111X
Logic	lr	010000X	and	100000
			or	100001
	lrv	010001X	xor	100010
			lor	100011
Shift	sr1	0100100	asr	100100
	srv	0100101	lsr	100101
	sr2	010011X	sl	10011X
Immediate	#i	1XXXXXX		
CompareX			xeo	010000
			xeq	010001
			xls	010010
			xgt	010011
			xo	0101XX
CompareY			yeo	011000
			yeq	011001
			yls	011010
			ygt	011011
			yo	0111XX
Load-Store	sa	01010XX	sa	101000
			la	101001
			lsc	10101X
	ld	010110X	sd	10110X
	ldv	010111X	sdx	10111X
Ifetch	pc	00000XX	pc	000000
			pco	000001
	pch	00001XX	ex	000010
			exo	000011
			pch	00010X
			ctl	00011X
Unused	ones	001XXXX		
	ones	11d-15d		

Table A.2: Register mapping.

Number	Guardbits	Function
0	000	x
1	001	$\overline{x}$
2	010	y
3	011	$\overline{y}$
4	100	x $\wedge$ y
5	101	$\overline{x}\wedge$ y
6	110	x $\wedge$ $\overline{y}$
7	111	TRUE

Table A.3: Mapping of the guard specifier conditions.

Timing of a MOVE instruction B

This appendix contains details about how a move operation is timed. This information is not needed for a compiler writer, but maybe handsome to understand (and appreciate) the operation of *MOVE32INT*.

Figure B.1 shows the main actions during the execution of a move operation. As shown, at certain points lock and squash requests may occur:

1. PC increment failure: global lock request. The program incrementer can only add a limited number of bits within one cycle.
2. Instruction cache miss: global lock request.
3. Function unit: local lock request. A FU can not deliver or accept a data item in time.
4. Instruction fetch unit: squash request. This happens when interrupts or traps occur.
5. Compare unit: squash request. The guard has been evaluated to a false value.

These requests go to the bus controller, which determines from these signals if a global lock should be distributed to all the units. The timing of this global lock signal is shown in figure B.2; derivation of this signal is in the critical path of the clock-high phase. The critical path of the clock-low phase is determined by the Instruction cache decode.

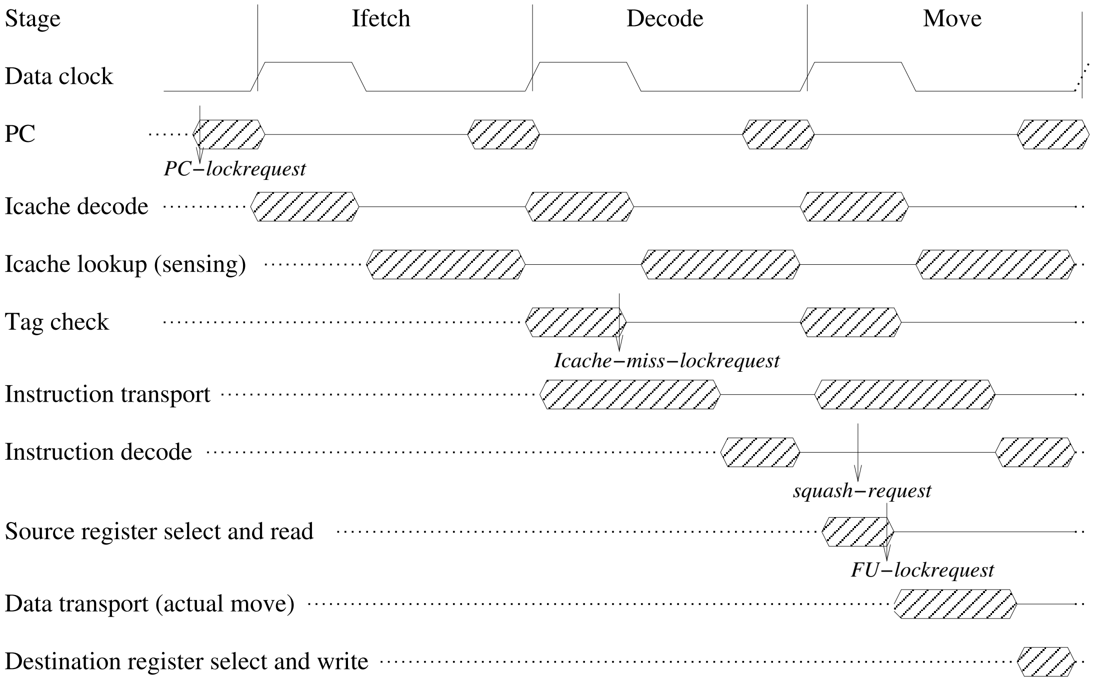


Figure B.1: Actions during the execution of a 3-stage pipelined move operation.

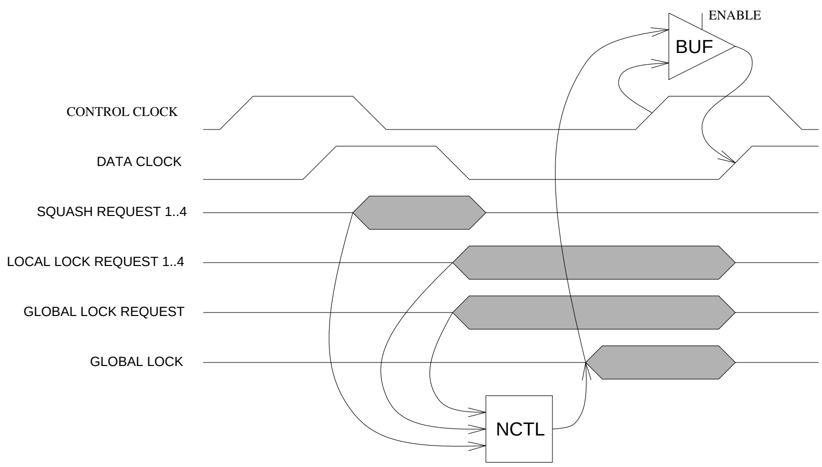


Figure B.2: Timing diagram of control signals during the move stage.

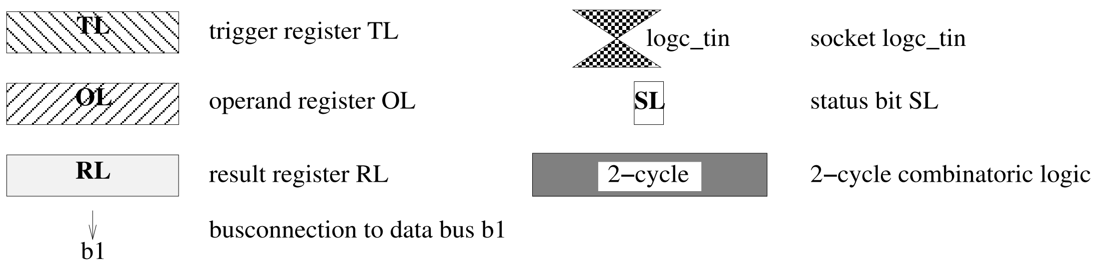
Details of function units

C

Although most information needed for programming *MOVE32INT* has been described elsewhere in this manual, it is convenient to group all the programming details per function unit in separate sections, as is done in this appendix. Besides a few details of the load-store and ifetch units, which have not yet been discussed, will be treated here as well.

In each section we describe the following items:

1. Block diagram. This shows the main characteristics of the described function unit and the sockets to which it is connected. Symbols used in these pictures obey the following convention:



2. Connectivity table, describing the connectivity between registers and transport network.
3. Supported operations. The type of operations indicates if it concerns a move to a trigger, operand or control register, or if it is a move from a result or status register. Identifiers *src* and *dst* indicate arbitrary source and destination register-ids.
4. Exceptions (optional).
5. Lock requests. This paragraph gives examples of all possible lock request types.

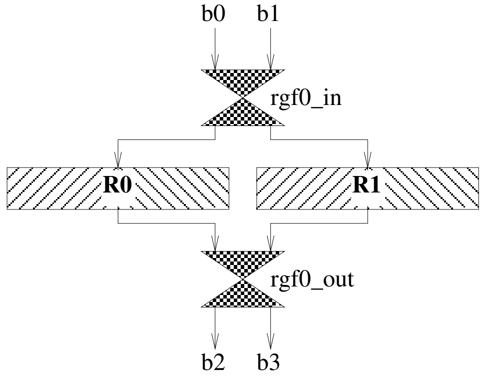
6. Squash requests (optional).
7. Control (optional).
8. Remarks (optional).

What is reset status of each FU?

C.1 General purpose registers

There are 11 general purpose registers: R0 - R10. For testing purposes R10 has been made scannable.

Block diagram



The block diagram shows how R0 and R1 are connected. Other registers are connected similarly, except R10, which does not share sockets with another register.

Connectivity

Socket	Physical Registers	Logic Registers	Bus 0	Bus 1	Bus 2	Bus 3
rgf0_in	R0, R1	(R0), (R1)			I	I
rgf0_out	R0, R1	(R0), (R1)	O	O		
rgf1_in	R2, R3	(R2), (R3)	I		I	
rgf1_out	R2, R3	(R2), (R3)		O		O
rgf2_in	R4, R5	(R4), (R5)		I		I
rgf2_out	R4, R5	(R4), (R5)	O		O	
rgf3_in	R6, R7	(R6), (R7)	I			I
rgf3_out	R6, R7	(R6), (R7)		O	O	
rgf4_in	R8, R9	(R8), (R9)		I	I	
rgf4_out	R8, R9	(R8), (R9)	O			O
rgf5_in	SR	(R10)	I	I		
rgf5_out	SR	(R10)			O	O

Supported operations

Operation	Type	Description
src $\rightarrow R_i$	trigger	R_i := src
$R_i \rightarrow$dst	result	dst := R_i

Lock requests

None.

Remarks

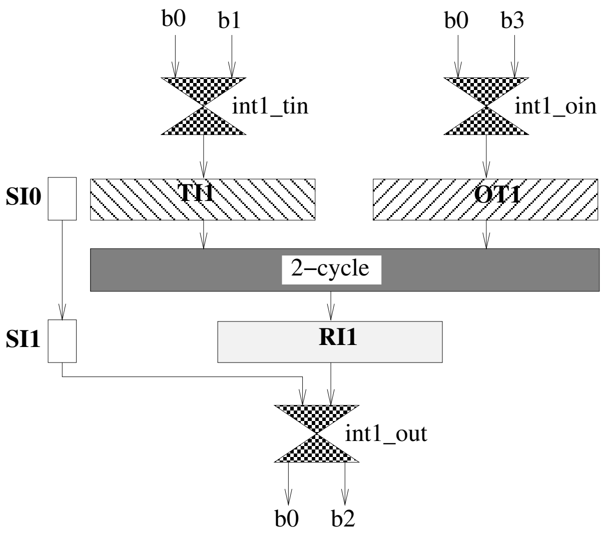
- Of both registers only one can be written and one (not necessarily the same) can be read each cycle.
- Registers act both as trigger and result.
- No valid status bits are needed.
- The following instruction is useful to swap R1 and R2:
  R1 →R2   R2 →R1

  Note that for even i, $i \leq 8$, R_i and R_{i+1} can not be swapped in one instruction!

C.2 Integer units

MOVE32INT contains two integer units, INT1 and INT2, performing add and subtract operations.

Block diagram of INT1



Connectivity

Socket	Physical Registers	Logic Registers	Bus 0	Bus 1	Bus 2	Bus 3
Integer unit 1						
int1_tin	TI1	(add1, sub1)	I	I		
int1_oin	OI1	(io1)	I			I
int1_out	RI1, SI1	(ir1), (ir1v)	O		O	
Integer unit 2						
int2_tin	TI2	(add2, sub2)			I	I
int2_oin	OI2	(io2)		I	I	
int2_out	RI2, SI2	(ir2), (ir2v)		O		O

Supported operations

Operation	Type	Description
Integer unit 1		
src $\rightarrow$add1	trigger	TI1 := src; RI1 := TI1 + OI1
src $\rightarrow$sub1	trigger	TI1 := src; RI1 := TI1 - OI1
src $\rightarrow$io1	operand	OI1 := src
ir1 $\rightarrow$dst	result	dst := RI1; flush result
ir1v $\rightarrow$dst	status	dst_{31} := SI1
Integer unit 2		
src $\rightarrow$add2	trigger	TI2 := src; RI2 := TI2 + OI2
src $\rightarrow$sub2	trigger	TI2 := src; RI2 := TI2 - OI2
src $\rightarrow$io2	operand	OI2 := src
ir2 $\rightarrow$dst	result	dst := RI2; flush result
ir2v $\rightarrow$dst	status	dst_{31} := SI2

Lock requests

Operation	Type	Scope	Description
ir1 $\rightarrow$dst	read	local	lock if valid bit not set (result not valid)
src $\rightarrow$add1	write	local	lock if $(\overline{pipe-full} \wedge \overline{SI0})$; i.e., not ready to start next operation
src $\rightarrow$io1	write	local	lock if $(\overline{pipe-full} \wedge \overline{SI0})$; i.e., not ready to accept the next operand

Remarks

- Reading the status register does not flush the result (this holds for all other units as well).
- To keep an integer FU optimal busy, the compiler should generate code like:

Instr#	Instruction	Comments
1	src_1 $\rightarrow$add1 src_2 $\rightarrow$io1	start addition 1
2	- delay cycle -	
3	src_3 $\rightarrow$add1 src_4 $\rightarrow$io1	start addition 2
4	ir1 $\rightarrow$dst	read result of addition 1
5	src_5 $\rightarrow$add1 src_6 $\rightarrow$io1	start addition 3
6	ir1 $\rightarrow$dst	read result of addition 2
7	- etcetera -	

- Write locks are not generated when the pipeline is full. An exception should be generated in that case, but full exceptions are not supported. Be aware that the following code results in unwanted results:

Instr#	Instruction	Comments
1	src_1 $\rightarrow$add1 src_2 $\rightarrow$io1	start addition 1
2	- delay cycle -	
3	src_3 $\rightarrow$add1 src_4 $\rightarrow$io1	start addition 2; pipeline is now full
4	src_5 $\rightarrow$io1	overwrite operand!

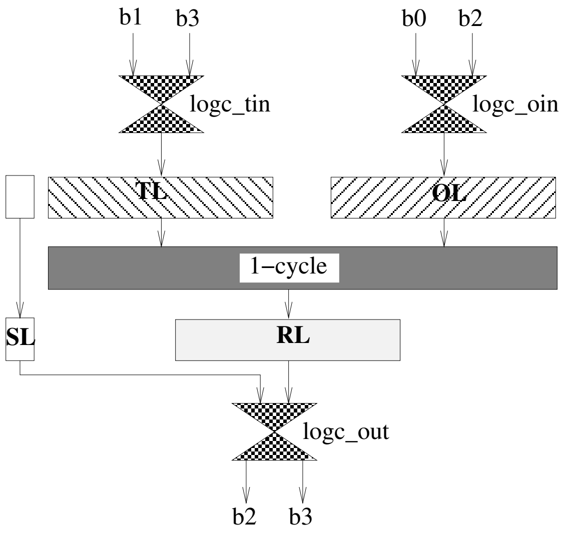
Addition 2 will be done with operand = src_5 instead of src_4.

- Status registers SI1 and SI2 are also used for storing valid bits of the compare units (see section C.6).

C.3 Logic unit

MOVE32INT contains 1 unit for 32-bit bitwise logic and, or and exor operations.

Block diagram



Connectivity

Socket	Physical Registers	Logic Registers	Bus 0	Bus 1	Bus 2	Bus 3
logc_tin	TL	(and, or, xor)		I		I
logc_oin	OL	(lo)	I		I	
logc_out	RL, SL	(lr), (lrv)			O	O

Supported operations

Operation	Type	Description
src →and	trigger	TL := src; RL := TL $\wedge$ OL
src →or	trigger	TL := src; RL := TL $\vee$ OL
src →xor	trigger	TL := src; RL := TL $\oplus$ OL
src →lo	operand	OL := src
lr →dst	result	dst := RL ; flush result
lrv →dst	status	dst_{31} := SL

Lock requests

Operation	Type	Scope	Description
lr →dst	read	local	lock if valid bit not set (result not ready)

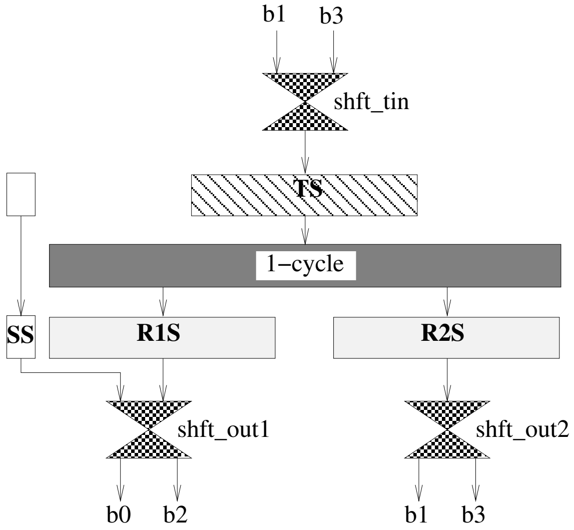
Remarks

- The not operation can easily be implemented using an exor with all ones.

C.4 Shift unit

MOVE32INT contains 1 unit for 1-bit and 2-bit arithmetic and logic shift operations.

Block diagram



Connectivity

Socket	Physical Registers	Logic Registers	Bus 0	Bus 1	Bus 2	Bus 3
shft_tin	TS	(asr, lsr, sl)[1]		I		I
shft_out1	R1S, SS	(sr1), (srv)	O		O	
shft_out2	R2S	(sr2)		O		O

Supported operations

Operation	Type	Description
src →asr	trigger	TS := src; $R1S_{30..0}$:= $TS_{31..1}$; $R1S_{31}$:= TS_{31} $R2S_{29..0}$:= $TS_{31..2}$; $R2S_{31..30}$:= TS_{31}
src →lsr	trigger	TS := src; $R1S_{30..0}$:= $TS_{31..1}$; $R1S_{31}$:= 0 $R2S_{29..0}$:= $TS_{31..2}$; $R2S_{31..30}$:= 0
src →sl	trigger	TS := src; $R1S_{31..1}$:= $TS_{30..0}$; $R1S_0$:= 0; $R2S_{31..2}$:= $TS_{29..0}$; $R1S_{1..0}$:= 0;
sr1 →dst	result	dst := R1S ; flush both shift results
sr2 →dst	result	dst := R2S ; flush both shift results
srv →dst	status	dst_{31} := SS

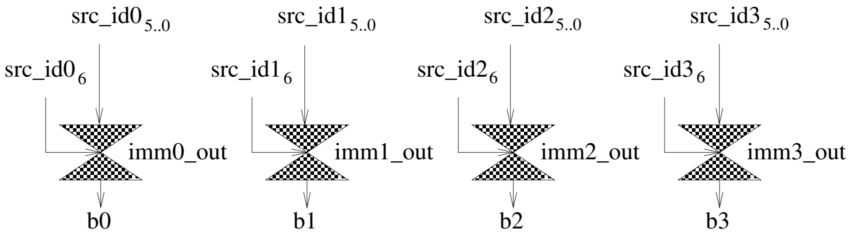
Lock requests

Operation	Type	Scope	Description
sr1 →dst	read	local	lock if valid bit not set (result not ready)
sr2 →dst	read	local	lock if valid bit not set (result not ready)

C.5 Immediate unit

MOVE32INT supports 6-bit short immediates on each data transport bus. The source field is interpreted as an unsigned immediate when the immediate bit of the move operation (= bit 12) is set.

Block diagram



Connectivity

Socket	Physical Registers	Logic Registers	Bus 0	Bus 1	Bus 2	Bus 3
$imm_{0..3}$_out	—	#0-#63	O	O	O	O

Supported operations

Operation	Type	Description
#immediate $\rightarrow$dst	-	dst := 0^{26} $\|$#immediate

Lock requests
None.

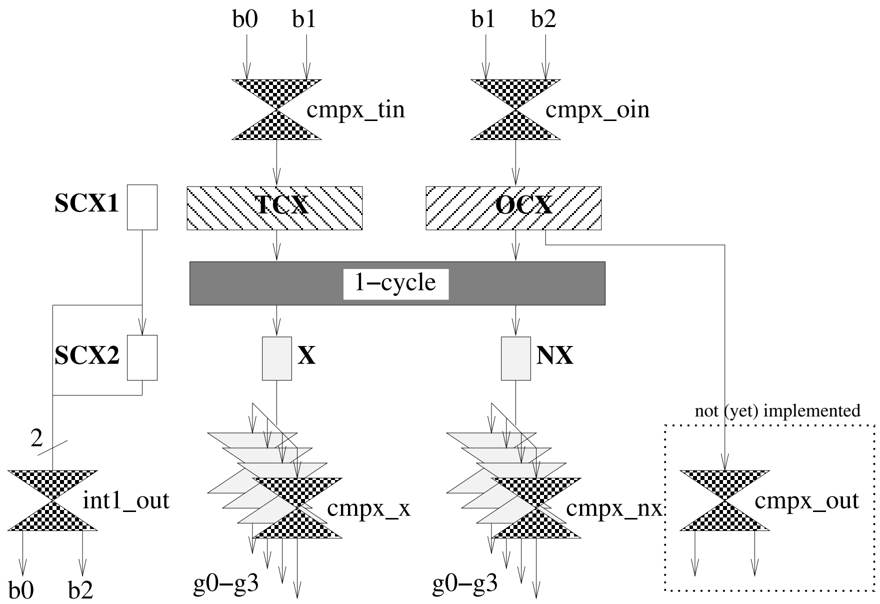
Remarks

- Short immediates are 6 bit unsigned numbers.
- The data inputs of the output sockets of the immediate unit are directly connected to the source-id bus. Bits 0..6 of the source-id are latched inside these sockets when short immediates are specified.

C.6 Compare units

MOVE32INT contains two compare units, CMPX and CMPY, producing boolean results X and Y (and their complements) respectively. These boolean results can be read and put on the guard busses (part of the control busses).

Block diagram of CMPX



check connectivity of valid bits; do we include x as well?

Connectivity

Socket	Physical Registers	Logic Registers	Bus 0	Bus 1	Bus 2	Bus 3
Compare unit X						
cmpx_tin	TCX	(xeo, xeq, xls, xgt)	I	I		
cmpx_oin	OCX	(xo)		I	I	
cmpx$_{0..3}$_x	X	(x)	C	C	C	C
cmpx$_{0..3}$_nx	NX	(nx)	C	C	C	C
int1_out	SCX1, SCX2	(ir1v)	O		O	
Compare unit Y						
cmpy_tin	TCY	(yeo, yeq, yls, ygt)			I	I
cmpy_oin	OCY	(yo)	I			I
cmpx$_{0..3}$_y	Y	(y)	C	C	C	C
cmpx$_{0..3}$_ny	NY	(ny)	C	C	C	C
int2_out	SCY1, SCY2	(ir2v)		O		O

** not yet complete **

Supported operations

Operation	Type	Description
Compare unit X		
src →xeq	trigger	TCX := src; X := TCX = OCX; NX := TCX $\neq$ OCX
src →xeo	trigger	TCX := src; X := TCX = 0; NX := TCX $\neq$ 0
src →xls	trigger	TCX := src; X := TCX < 0; NX := TCX $\geq$ 0
src →xgt	trigger	TCX := src; X := TCX > 0; NX := TCX $\leq$ 0
src →xo	operand	OCX := src
x: src →dst	result	if x then dst := src; flush both results: X and NX
!x: src →dst	result	if !x then dst := src; flush both results: X and NX
ir1v →dst	status	$dst_{30..29}$:= SCX1 ||SCX2; no flush
Compare unit Y		
src →yeq	trigger	TCY := src; Y := TCY = OCY; NY := TCY $\neq$ OCY
src →yeo	trigger	TCY := src; Y := TCY = 0; NY := TCY $\neq$ 0
src →yls	trigger	TCY := src; Y := TCY < 0; NY := TCY $\geq$ 0
src →ygt	trigger	TCY := src; Y := TCY > 0; NY := TCY $\leq$ 0
src →yo	operand	OCY := src
y: src →dst	result	if y then dst := src; flush both results: Y and NY
!y: src →dst	result	if !y then dst := src; flush both results: Y and NY
ir2v →dst	status	$dst_{30..29}$:= SCY1 ||SCY2; no flush
Combined (read) result operations		
x.y: src →dst	result	if $(x \wedge y)$ then dst := src; flush all 4 results: X, NX, Y and NY
!x.y: src →dst	result	if $(\overline{x} \wedge y)$ then dst := src; flush all 4 results: X, NX, Y and NY
x.!y: src →dst	result	if $(x \wedge \overline{y})$ then dst := src; flush all 4 results: X, NX, Y and NY

Lock requests

Operation	Type	Scope	Description
x: src →dst	read	global	lock if $(\overline{pipe-empty} \wedge \overline{SCX2})$; result not valid
y: src →dst	read	global	lock if $(\overline{pipe-empty} \wedge \overline{SCY2})$; result not valid
x.y: src →dst	read	global	lock if $(\overline{Xpipe-empty} \wedge \overline{SCX2}) \vee (\overline{Ypipe-empty} \wedge \overline{SCY2})$

Similar read locks occur for other guard usages.

Squash requests

Reading X, NX, Y and/or NY, and putting their values on the control busses (guard lines) effectively means that one our more local squash requests are done to the bus-controller. In *MOVE32INT* these requests are always granted.

Remarks

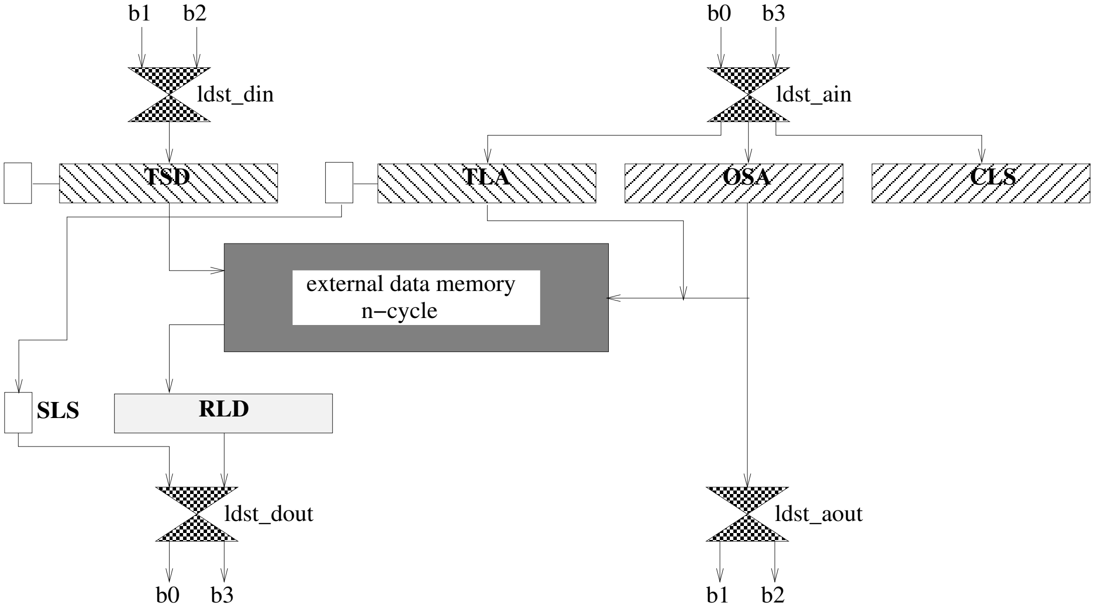
- The compare units are the only FUs which do not request a lock on an empty pipeline. This allows to use the same boolean result for guarding multiple move operations.
- The status bits of the compare units can be read (and saved) using bits 29..30 of the output sockets of the integer units.

- The operand registers OCX and OCY can not be read. Therefore the compiler writer has to make sure that these registers do not contain state (e.g. by writing operand and trigger always in the same cycle).
- In contrast to most other FUs who implement *local* locks, the compare units implement *global* locks. This is of course necessary, otherwise the guard would not be evaluated correctly. check read operands

C.7 Load-Store unit

MOVE32INT contains 1 load-store unit which communicates with the external data memory.

Block diagram



Connectivity

Socket	Physical Registers	Logic Registers	Bus 0	Bus 1	Bus 2	Bus 3
ldst_ain	TLA, OSA, CLS	(sa, sax), (la), (ctl)	I			I
ldst_din	TSD	(sd)		I	I	
ldst_aout	OSA	(sa)		O	O	
ldst_dout	RLD, SLS	(ld), (ldv)	O			O

Supported operations

Operation	Type	Description
src $\rightarrow$la	trigger	TLA := src; RLD := DMEM[TLA]
src $\rightarrow$sd	trigger	TSD := src; DMEM[OSA] := TSD
src $\rightarrow$sa	operand	OSA := src
src $\rightarrow$ctl	control	CLS := src; change timing behavior
ld $\rightarrow$dst	result	dst := RLD; flush result
ldv $\rightarrow$dst	status	dst_{31} := SLS
sa $\rightarrow$dst	status	dst := OSA; used for state saving

Lock requests

Operation	Type	Scope	Description
src →la	write	local	lock if unit not ready to start next load operation
src →sd	write	local	lock if current store not yet finished
src →sa	write	local	lock if current store not yet finished
ld →dst	read	local	lock if valid bit not set (result not valid)

Control

The load-store timing behavior depends on the value of the control register CLS and on the (hardwired) mode to which *MOVE32INT* is configured. This mode is either Fast (F) or Slow (S). CLS is a 32 bit register containing four 8-bit numbers, t4..t1; t1 (= $\text{CLS}_{7..0}$) is the least significant byte. The value n_i of t_i is given by the number of consecutive one bits (counting from the least significant side). So its maximum value is eight. These values govern the latency and throughput (= distance^{-1}) of the load-store unit, as shown in the following table (fast mode):

Operation	Latency (cycles)	Distance (cycles)
RaR	4 + n1	2 + n1
WaW	4 + n4	3 + n4
RaW	7 + n1 + n2	5 + n1 + n2
WaR	7 + n3 + n4	6 + n3 + n4

RaR means Read after Read (or Load after Load), RaW Read after Write, etc. When *MOVE32INT* operates with a clock cycle larger than about 14 ns the slow mode should be used. In slow mode, the RaR latency is 3 + n1.

Check RaW and WaR latencies in slow mode

CLS has to be set according to the speed of the connected data memory. After reset CLS contains all ones.

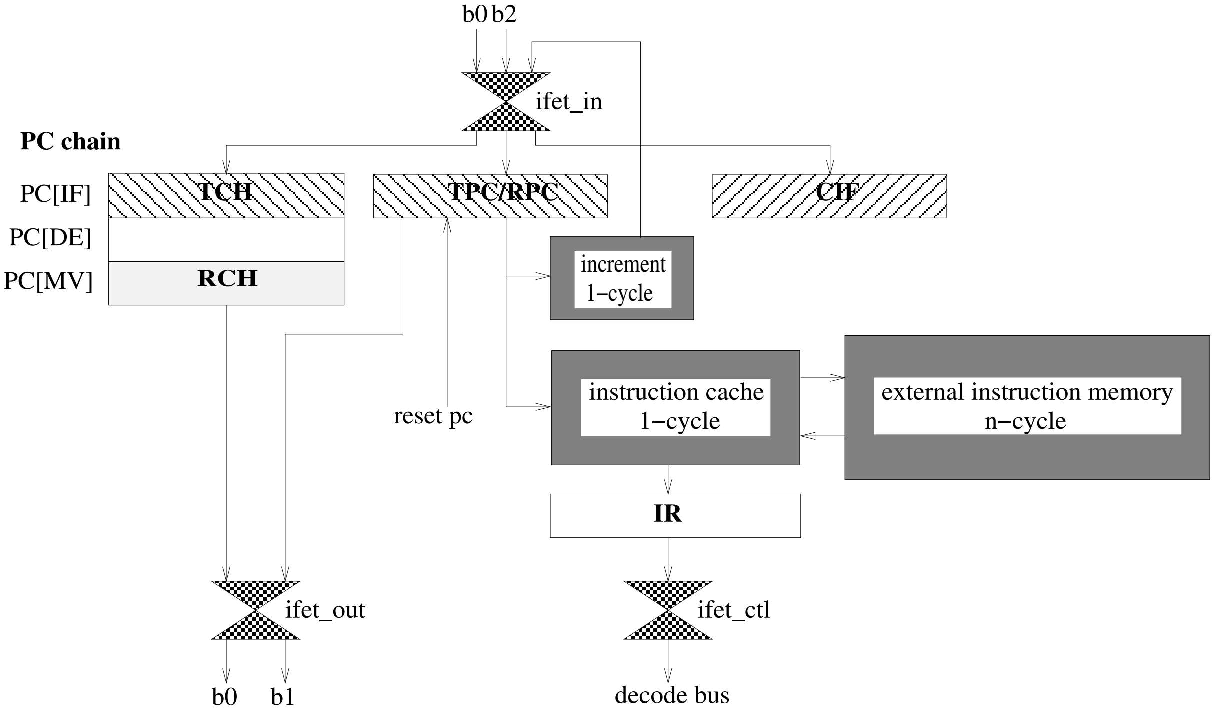
Remarks

- There can be 3 outstanding operations, 2 loads and 1 store operation.
- If a store is triggered when 2 loads are outstanding, the store gets highest priority.
- Value n_2 of the control register also determines how long it takes before a bus-request is acknowledged.

C.8 Instruction fetch unit

The instruction fetch unit (ifetch unit) is the only unit which is triggered each cycle autonomously with the incremented PC value (if not trigged by the program itself). In order to support exceptions the ifetch unit contains a three stage PC chain, which, in user mode, runs synchronously with the transport pipeline, but locks in exception mode.

Block diagram



Connectivity

Socket	Physical Registers	Logic Registers	Bus 0	Bus 1	Bus 2	Bus 3
ifet_in	TPC, TCH, CIF	(pc, pco, ex, exo), (pcc), (cif)	I		I	
ifet_out	RPC, RCH	(pc), (pcc)	O	O		
ifet_ctl	IR	—	C	C	C	C

Supported operations

Operation	Type	Description
src →pc	trigger	TPC := src
src →pco	trigger	$TPC_{5..0}$:= $src_{5..0}$
src →ex	trigger	TPC := src; start exception (TRAP)
src →exo	trigger	$TPC_{5..0}$:= $src_{5..0}$; start exception (TRAP)
src →cif	control	CIF := src
src →pcc	trigger	TCH := src; shift PC chain
pc →dst	result	dst := RPC
pcc →dst	status	dst := RCH; shift PC chain; enabling the next chain register

Exceptions
The ifetch unit supports two modes: *user* and *exception* mode. To support exceptions the ifetch unit contains a 3-stage PC chain which runs synchronously with the PC. The ifetch unit enters exception either through a software TRAP or by an external interrupt. When entering exception mode:

1. The current instruction in the move stage is finished.
2. Instructions which are in the ifetch and decode stage are squashed.
3. The PC chain locks after the current cycle. This means that a current move to the PC is still written in the PC chain.
4. The next PC is either the TRAP address, or the interrupt address 0x40.
5. Exceptions will be disabled.

User mode is entered by reading three times from the PC chain. Each read enables the next chain register; e.g. after one read, TCH containing PC[IF] runs again synchronously with the PC, the other two chain registers are still blocked.

Lock requests

Operation	Type	Scope	Description
	internal	global	lock if PC is not yet incremented
	internal	global	cache miss; lock during ifetch stage

As indicated, lock requests from the ifetch unit are not related to external operations; they are of type *internal*.

Control
The ifetch timing behavior can be controlled by writing to the control register CIF. Bits 24..31 of this register contains an 8-bit number t1. The value of t1 is given by the number of consecutive ones, counting from bit 31 (the msb). This value governs the number of cycles between 2 instruction word loads. ** finish **

Remarks

- TRAPs take precedence over external interrupts.
- The ifetch unit incorporates a small, 256 byte instruction cache. This allows for single cycle instruction accesses. The cache is direct mapped and has a 16 byte line size (2 instructions per line). Upon a cache miss only one instruction is fetched from memory; so the cache implements subblocking.

D Exception handling

Exceptions were discussed in section 5.3. This appendix gives full details of how exceptions are treated, and how to implement exception handlers. An exception has the following characteristics:

- An exception is only granted by the processor when the processor is in *user mode*, when exceptions are enabled.
- An exception causes an instruction sequence change to a predefined or specified target instruction address (exception handler entrance).
- The code in the exception handler is started in *exception mode*; exceptions are disabled.
- Complete or partial state saving and restoring is possible.
- For state saving and restoring, a *base exception state frame* in low memory (BESF) (accessible with short immediate addresses) is needed. Large addresses can only be generated when e.g. an integer, shift, or load/store unit is free.
- Nested exceptions are possible when a stack of *dynamical exception state frames* (DESFs) is created somewhere in memory. The BESF has to be copied to a DESF before enabling exceptions.

The configuration of an exception handler routine is free programmable by the programmer. Each handler will however match one of the following templates:

- **XXX handler**
  This handler starts as each handler in exception mode (X) and disables new exceptions during the whole routine. This is needed for handlers which must finish before any other task can be started or which should be fast. An XXX handler does not need to save the PC chain or create a DESF.

- **XUX handler**
  This handler contains a period in which a new exception may be granted. As long as the handler uses the BESF to store data, the routine should stay in exception mode. To restore the data from a DESF, the handler should use the BESF and should therefore return to exception mode during return.

D.1 Processor state

MOVE32INT contains the following state:

- 11 General purpose registers.
- FU stages
- PC chain

A possible layout of the BESF and DESF which contains the whole processor state is shown in figure D.1. The exception stack pointer is not part of the frames, but if nested interrupts are desired, something like an exception stack pointer is needed. The addresses shown in the figure correspond with the addresses used in sections D.5 and D.6.

0	Store Address	15	X–Boolean #1	30	Integer1 Result Count
	Y–Flag / Count		X–Boolean #2		Integer1 Operand
	Y–Boolean #1		GPR Contents #0		Logic Result #1
	Y–Boolean #2		GPR Contents #1		Logic Result #2
5	Integer2 Result #1		GPR Contents #2	35	Logic Result Count
	Integer2 Result #2	20	GPR Contents #3		Logic Operand
	Integer2 Result Count		GPR Contents #4		Shift Result1 #1
	Integer2 Operand		GPR Contents #5		Shift Result2 #1
	Load Result #1		GPR Contents #6		Shift Result1 #2
10	Load Result #2		GPR Contents #7		Shift Result2 #2
	Load Result Count	25	GPR Contents #8	40	Shift Result Count
	Saved PC #1		GPR Contents #9		
	Saved PC #2		GPR Contents #10	41	Exception Stack Pointer
	Saved PC #3		Integer1 Result #1		
	X–Flag / Count		Integer1 Result #2		

Figure D.1: Exception state frame.

D.2 Trap

A software trap happens when writing to the trigger registers ex or exo. This causes a jump to the specified address and changes the processor mode from *user* to *exception mode*. Figure D.2 illustrates what happens in the pipeline and the PC chain when such a trap is initiated. The figure shows the contents of the PC chain as saved PC#0 to saved PC#2, which stand for the address of the current instruction fetch, current instruction decode and current instruction execution (the move) respectively. After the move of instruction U2, the mode changes from user to exception. When the processor is in exception mode, the PC chain does not shift along with the program counter anymore.

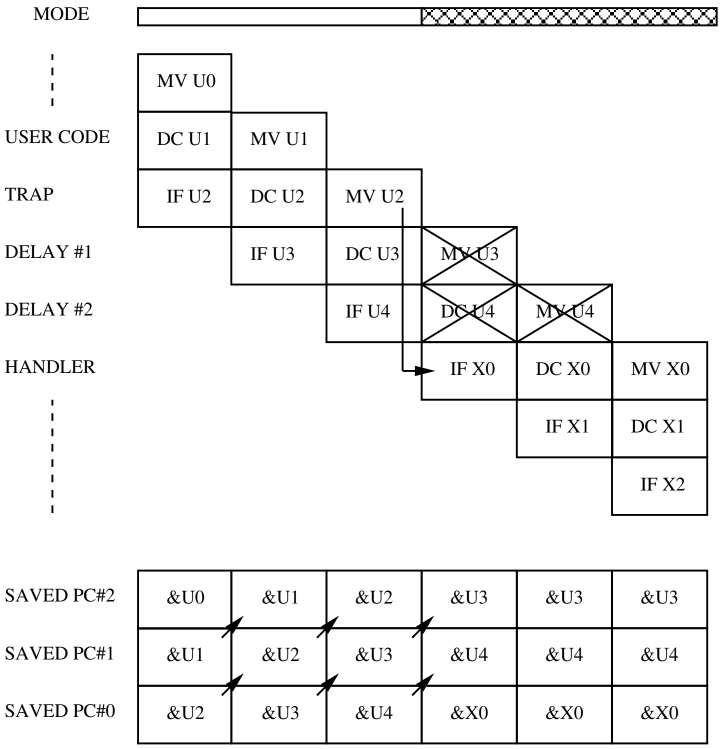


Figure D.2: Software trap.

Note that a software trap requires only the saving of 2 PC values from the PC chain. The third PC (X0 in the figure) is the address of the first trap handler instruction.

D.3 Interrupt

Figure D.3 illustrates what happens to the PC chain and pipeline when an interrupt occurs. After executing instruction U2, the processor changes to exception mode. The first handler instruction is fetched and two instructions are squashed (U3 and U4). The PC chain contains the addresses of the two squashed instructions and of the instruction that would be fetched if the interrupt didn't occur.

Remark that the figure also demonstrates what happens when a lock request and an external interrupt happen in the same cycle. The interrupt is accepted after the lock has ended. The main difference in the PC chain with a trap is that the PC chain doesn't contain the starting address of the handler.

D.4 Return from exception

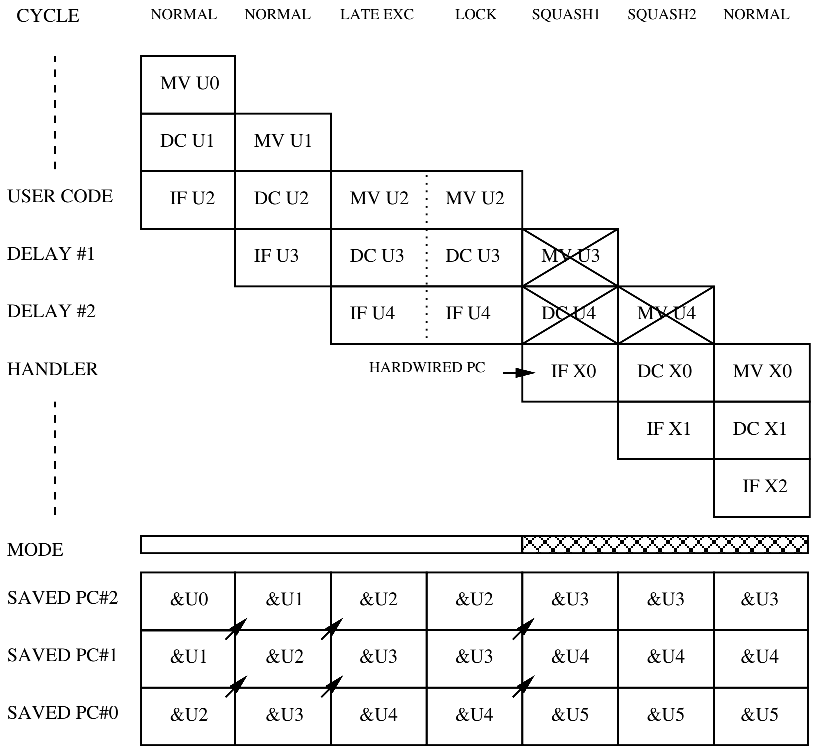


Figure D.3: Hardware interrupt.

Returning from an exception is done by restoring the state including the PC chain. The last three statements of the handler contain jumps (writes to the program counter). The actual restoring from exception to user mode is done by reading the PC chain for the third time.

Figure D.4 illustrates this process for a XXX interrupt handler. The PC chain contains three valid addresses which are being written to the program counter. Reading the PC chain makes room for another program counter, which is moved there in the same cycle by writing to pc. The result is that the PC chain is rotated once, but mainly that the PC chain is read three times, returning to user mode.

D.5 State saving code

State saving code for the various function units will now be discussed. The order in which the function units are discussed coincides with an order in which the states can (and sometimes should) be saved. For some units pseudocode is given to illustrate the state saving process. At the end the whole assembler program is given. The code given for each unit can be used directly for an exception handler, but it has not been optimized in order to be fast. The object of this section (and the next) is not to give fast exception handling code, but to present a detailed and clear example of such code.

Load-Store Unit - part 1 The state of this unit must be written to memory first because this unit will be used to save the state. The store address (sa) will be overwritten the first time, so it must be saved in the first instruction. The load part can not yet be saved, because then the valid flag must be tested. This

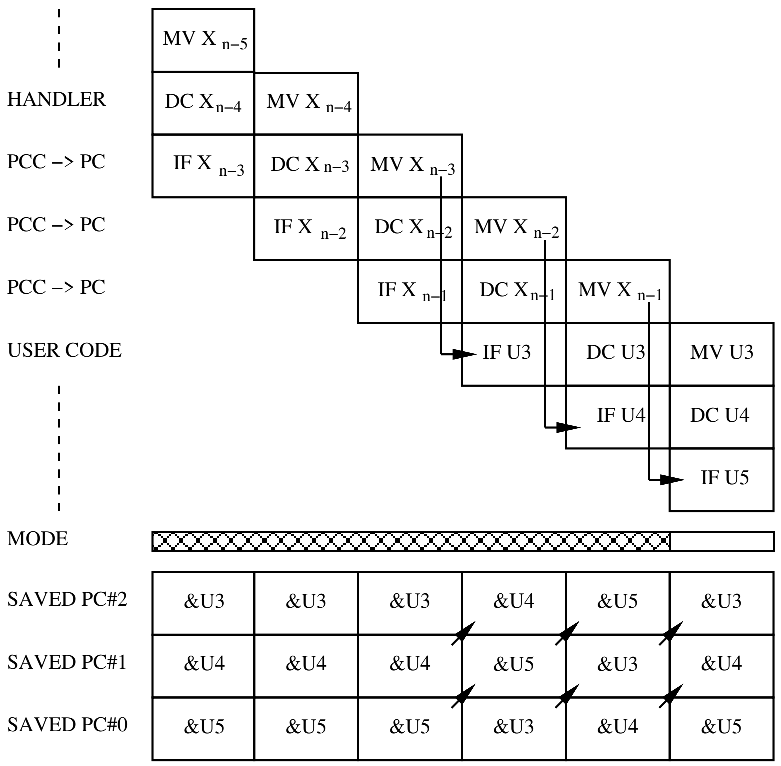


Figure D.4: Return from exception.

implies the free use of guarded execution, but the state of the compare units has not yet been saved.

```
BASEFRAME[0] := StoreAddress
```

Compare Unit The compare unit has a special property with regard to the valid flags. If all stages are empty and no trigger is given, reading the result register gives the last result, no lock. This means that even if the result is not valid, it has to be saved.

Unlike other units, the valid flag cannot be tested to see whether results have to be saved because this would imply the free use of a compare unit. Note that the valid flags of the compare units X and Y can be read from the status register of integer unit one and two respectively and that the operand cannot be read. The next pseudocode describes how the state of compare unit X can be saved. The code for storing the state of compare unit Y is based on the same principle.

```
BASEFRAME[14] := Integer1Status
IF X THEN BASEFRAME[15] := 1
     ELSE BASEFRAME[15] := 0
IF X THEN BASEFRAME[16] := 1
     ELSE BASEFRAME[16] := 0
```

Register File All the registers must be saved. This can be done by simply writing them to memory, as shown in the next pseudocode.

```
BASEFRAME[17] := Register0
...
BASEFRAME[27] := Register10
```

Logical Unit The state of the logical unit consists of an operand and two possible valid stages. Valid stages can be saved by saving the result register. The operand can be saved by triggering e.g. an or operation with argument zero.

```
IF LogicalResultValid THEN
BEGIN
  BASEFRAME[32] := LogicalResult
  Count := 1
  IF LogicalResultValid THEN
  BEGIN
    BASEFRAME[33] := LogicalResult
    Count := 2
  END
END
ELSE
  Count := 0
BASEFRAME[34] := Count
BASEFRAME[35] := LogicalOperand
```

Integer Units Saving the state of the integer units is done the same way as saving the state of the logical unit. The operand can be obtained by adding zero to the operand.

Load-Store Unit - part 2 When guarded execution is available, the load part of the load-store unit can be saved. The used method to save the load part is much like the one used for the logical unit, however the operand of this unit (sa) has already been saved.

Shift Unit In contrast to some other units, this unit has no operands. It does however have two result registers which must be read in the same cycle. The following pseudocode describes the state saving for this unit.

```
IF ValidShiftResult THEN
BEGIN
  BASEFRAME[36] := Shift1Result; Temp := Shift2Result
  BASEFRAME[37] := Temp
  Count := 1
  IF ValidShiftResult THEN
  BEGIN
    BASEFRAME[38] := Shift1Result; Temp := Shift2Result
    BASEFRAME[39] := Temp
    Count := 2
  END
```

```
END
ELSE
  Count := 0
BASEFRAME[40] := Count
```

XUX move BESF to DESF Only for XUX exceptions, the BESF has to be copied to a DESF before exceptions are enabled again.

```
FramePointer := BASEFRAME[41]
FOR Count := 0 TO 40 DO
BEGIN
  MEMORY[FramePointer] := BASEFRAME[Count]
  FramePointer := FramePointer + 1
END
BASEFRAME[41] := FramePointer
```

The code might be optimized by saving the state of some units directly to DESF instead of saving it to a BESF and then copying it.[1]

Instruction Fetch Unit - Interrupt handler If the handler is for a XXX interrupt, the PC chain does not need to be saved. For a XUX interrupt handler however, the PC chain has to be saved because the chain will be overwritten during the user mode part of the handler. Reading the PC chain the third time enables exceptions again, so this must be done as the last of the XUX state saving. At least the last PC chain value must be written directly to a DESF because the BESF might be used by another exception.

```
Framepointer := BASEFRAME[41] - 30
MEMORY[FramePointer] := PCChain
FramePointer := FramePointer + 1
MEMORY[FramePointer] := PCChain
FramePointer := FramePointer + 1
MEMORY[FramePointer] := PCChain
```

Instruction Fetch Unit - Trap handler For a trap handler, the same code as for the interrupt handler can be used for saving the PC chain. Saving the third PC chain value to memory is redundant (see section D.2), however it must be read in order to restore to user mode.

Assembler state saving code The next assembler code can be used to save the state of the machine. Most labels are not used for jumping, but mark the beginning of the code for a specific unit.

```
LoadStorePart1:
    #0->sa          sa->sd
```

[1]Only the first eleven locations in the BESF have to be used directly.

```
CompareX:
    #14->sa                           ir1v->sd
    #15->sa         X:#1->sd          !X:0->sd
    #16->sa         X:#1->sd          !X:0->sd


CompareY:
    #1->sa          ir2v->sd
    #2->sa          Y:#1->sd          !Y:0->sd
    #3->sa          Y:#1->sd          !Y:0->sd


RegisterFile:
    #17->sa         gpr0->sd
    #18->sa         gpr1->sd
    #19->sa         gpr2->sd
    #20->sa         gpr3->sd
    #21->sa                          gpr4->sd
    #22->sa                          gpr5->sd
    #23->sa                          gpr6->sd         gpr8->gpr0
    #24->sa                          gpr7->sd         gpr9->gpr1
    #25->sa         gpr0->sd
    #26->sa         gpr1->sd
    #27->sa                          gpr10->sd


Logical:
                                     lrv->yls
    Y:#32->sa       !Y:#0->gpr4      Y:lr->sd         Y:#1->gpr4
                                     lrv->yls
    Y:#33->sa                        Y:lr->sd         Y:#2->gpr4
    #34->sa                          gpr4->sd         #0->lor
    #35->sa                          lr->sd


Integer1:
    ir1v->xls
    X:#28->sa       X:#1->gpr4       X:ir1->sd        !X:#0->gpr4
    ir1v->xls
    #29->sa         X:#2->gpr4       X:ir1->sd
    #30->sa         #0->add1         gpr4->sd
    #31->sa                          ir1->sd


Integer2:
                    ir2v->xls
    #4->sa          X:ir2->sd        X:#1->gpr0       !X:#0->gpr0
                    ir2v->xls
```

```
    #5->sa          X:ir2->sd       X:#2->gpr0
    #6->sa          gpr0->sd                        #0->add2
    #7->sa          ir2->sd


LoadStorePart2:
    ldv->xls
    X:ld->gpr6      X:#1->gpr4                      !X:#0->gpr4
    #8->sa          X:gpr6->sd
    ldv->xls
    X:ld->gpr6      X:#2->gpr4
    #9->sa          X:gpr6->sd
    #10->sa                         gpr4->sd


Shift:
    srv->xls
    #36->sa         X:#1->gpr4      X:sr1->sd       X:sr2->gpr6
    #37->sa         !X:#0->gpr4     X:gpr6->sd
    srv->xls
    #38->sa         X:#2->gpr4      X:sr1->sd       X:sr2->gpr6
    #39->sa                         X:gpr6->sd
    #40->sa                         gpr4->sd


XUXMoveBESFtoDESF:
; This section is only needed for XUX handlers.
; At the start of 'Loop' the integer units contain:
; integer unit 1: fp...fp+40
; integer unit 2: 0...40

    #41->la         #0->io2                         #0->add2
    ld->add1                                        #0->io1
    #1->io1         #1->io2
Loop:
                    ir2->gpr4                       ir2->la
    gpr4->xeo       #40->xo         gpr4->add2
    !X:Loop->pco
    ir1->sa                         ir1->gpr2       ld->gpr4
                    gpr2->add1      gpr4->sd

                    ir2->ones       ir1->sd         #41->sa


XUXIFetch:
; This section is only needed for XUX handlers.

    #41->la
```

```
    ld->sub1                                          #30->io1
    ir1->sa        pcc->sd        ir1->gpr2
    #1->io1        gpr2->add1
    ir1->sa        pcc->sd        ir1->gpr2
                   gpr2->add1
    ir1->sa        pcc->sd

; At this point exceptions are enabled again for XUX handlers. State
; saving is finished, so code to handle the exception can start here.
```

D.6 State restoring code

This section describes state restoring code for the various function units. The CALCULATE in the pseudocode means calculating the result without reading it from the result register.

XUX Return from user mode The state restoring starts by returning to exception mode because the BESF will be used. This can be done in several ways. One possibility is that before the user mode part of the handler is being called, the return address is calculated and saved. The return address can e.g. be saved as part of the DESF or passed to the user mode part in a register. At the end of the user mode part, a jump to the return address is performed by writing to the exception register. Effectively this is a call - return mechanism in which the return also changes to exception mode.

Especially for an interrupt handler, the constant return address can be stored at a fixed address in low memory (accessible with short immediates). At the end of the user mode part, an exception jump to the return address is performed. This is possible as long as only a few interrupt entry points exist, because of the lack of low memory. The *MOVE32INT* currently supports one interrupt entry point, making this a feasible option.

A third possibility arises when the whole handler is stored in consecutive memory locations. In that case writing an immediate to the exception offset register (exo) can be sufficient. This case is shown in the assembler code at the end of this section.

XUX move DESF to BESF For XUX exceptions, at least part of the DESF has to be copied to a BESF before restoring the last states. In order to keep the code as clear and simple as possible, the whole state frame is moved.

```
Framepointer := BASEFRAME[41]
FOR Count := 40 DOWNTO 0 DO
BEGIN
  FramePointer := FramePointer - 1
  BASEFRAME[Count] := MEMORY[FramePointer]
END
BASEFRAME[41] := FramePointer
```

The code might be optimized by restoring the state of some units directly from the DESF instead of copying it to a BESF and then restoring it.

Calculate address of ifetch & load-store handler At the end of the state restoring code, guarded execution is not possible because the compare units have already been restored. Because of this, for each different possibility a handler is written. By calculating the address of the right handler in advance, the jump to that routine can easily be taken.

For the calculation of the address of the instruction fetch & load-store handler, choose one of the four routines, depending on whether a XXX or XUX handler and a trap or interrupt handler is desired. For each of these routines, three pieces of code exist. This is for restoring the load-store unit with two valid load results, one and none.

Compare Units - Part 1 The valid bits of the compare unit are stored as two bits in an integer. Because extracting these bits requires the shift or logical unit, this is done in advance since the compare units are one of the last to be restored. The pseudocode for compare unit Y is given, but the same principle is used for compare unit X.

```
IF (BASEFRAME[1] sl 2) < 0 THEN
  IF (BASEFRAME[1] sl 1) < 0 THEN
    BASEFRAME[1]:=2
  ELSE
    BASEFRAME[1]:=1
ELSE
  BASEFRAME[1]:=0
```

Shift unit The shift unit has two outputs: sr1 and sr2 (shifted one or two positions). The shifted results are saved. Depending on these values, the following actions are taken:

1. sr2 has two zero's at the lsb position. This means the result can be obtained by shifting sr1 once to the right, and then to the left again.
2. If the first point is not applicable, the result was originally shifted to the right. First shift the result back to the left. If the msb of sr1 was zero, use a logic shift right, otherwise an arithmetic shift right.

No operands exist for this unit, so they don't have to be dealt with. Notice that this unit uses the logical unit, and must therefore be restored first. The next pseudocode describes the restoring of the shift unit.

```
IF BASEFRAME[40]>0 THEN
BEGIN
  ShiftLeft := (BASEFRAME[38] sl 1)
  ShiftRight := (BASEFRAME[38] asr 1)

  IF (BASEFRAME[37] AND 3) = 0 THEN
```

```
    CALCULATE ((BASEFRAME[36] lsr 1) sl 1)
  ELSE
    IF BASEFRAME[36] < 0 THEN
      CALCULATE ((BASEFRAME[36] sl 1) asr 1)
    ELSE
      CALCULATE ((BASEFRAME[36] sl 1) lsr 1)

  IF BASEFRAME[40] = 2 THEN
    IF (BASEFRAME[39] AND 3) = 0 THEN
      CALCULATE (ShiftRight sl 1)
    ELSE
      IF BASEFRAME[38] < 0 THEN
        CALCULATE (ShiftLeft asr 1)
      ELSE
        CALCULATE (ShiftLeft lsr 1)
END
```

Logical unit If two valid stages must be restored for this unit, the operand must be used in restoring the second stage. This restoring is done by using the exclusive or (and and or functions are not always possible). Using the exclusive or twice gives the same result.

```
IF BASEFRAME[34] = 2 THEN
  Temp := BASEFRAME[33] XOR BASEFRAME[35]

IF BASEFRAME[34] > 0 THEN
BEGIN
  CALCULATE (BASEFRAME[32] or 0)
  LogicalOperand := BASEFRAME[35]
  IF BASEFRAME[34] = 2 THEN
    CALCULATE (Temp xor LogicalOperand)
END
  ELSE LogicalOperand := BASEFRAME[35]
```

Integer units The integer units are a lot like the logical unit. Restoring a second valid stage can be done by subtracting the operand from the result and adding it again.

Compare units Even if a compare unit has no valid stages, the last result must be restored. The operand has not been saved and therefore cannot be restored. The following pseudocode indicates how the state can be restored. The code does not take into account that the compare unit uses itself; the assembler code does. The given pseudocode has been written for compare unit y.

```
  CALCULATE (BASEFRAME[2] > 0)
  IF BASEFRAME[1] > 0 THEN
  BEGIN
    IF BASEFRAME[1] = 2 THEN
      CALCULATE (BASEFRAME[3] > 0)
```

```
  END
  ELSE
    IF Y THEN
      Dummy_if_to_remove_Y_valid_flag
```

Register File The register file can be restored directly from memory. This should be done at a late stage because other state restoring code uses some registers.

Instruction Fetch & Load-Store Unit In restoring the PC chain, a distinction must be made between an interrupt handler and a trap handler. This is because a trap handler has only two valid addresses in the PC chain instead of three. The third address is the starting address of the trap handler itself (see section D.2).

For each of the four handlers (XXX or XUX and trap or interrupt), three routines exist for restoring the instruction fetch and the load-store unit. For a XUX interrupt handler all three are given (XUXint0load to XUXint2load) in assembler code. For the other three handlers, only the routines with two loads are given (XXXint2load, XUXtrap2load and XXXtrap2load). The other routines (with zero or one load) can be obtained by removing the '#8->la' for one load and '#7->la' and '#8->la' for zero loads.

Assembler state restoring code The next code can be used to restore the state of the machine. If pseudocode is available, the assembler code is written in accordance with it.

```
XUXReturnFromUserMode:
; This section is only needed for XUX handlers.
    XUXMoveDESFtoBESF->exo


XUXMoveDESFtoBESF:
; This section is only needed for XUX handlers.
; At the start of 'Loop' the integer units contain:
; integer unit 1: 40..0
; integer unit 2: fp-1...fp-41

    #41->la         #39->add1       #1->ygt         #1->io1
                    #1->io2                         ld->sub2
Loop:
    Y:Loop->pco     ir2->io2        ones->add2      ir2->la
    ir1->sa                         ir1->gpr2       ld->gpr4
    #1->yo          gpr2->sub1      gpr4->sd        gpr2->yeo

    #41->sa         ir2->sd         !Y:ir1->ones


; For the next routine, the Q in QCalcIF_LdSt_Handler and Q0load to Q2load
; must be changed into XXXint, XUXint, XXXtrap or XUXtrap depending on the
; desired handler type.
QCalcIF_LdSt_Handler:
```

```
    #10->la                                         #42->sa
    ld->xeo         #2->xo                          ld->yeq
                    X:Q2load->sd    Y:Q0load->sd
                    !X&!Y:Q1load->sd


CompareP art1:
    #1->la
    #14->la                                         ld->sl
                    sr2->xls        sr1->yls        ld->sl
    #1->sa          !X:#0->sd
                    X&Y:#2->sd      X&!Y:#1->sd
                    sr2->xls        sr1->yls
    #14->sa         !X:#0->sd
                    X&Y:#2->sd      X&!Y:#1->sd


Shift:
; gpr3 = result #2 sr1
; gpr0 = result #2 sr1 asr 1
; gpr1 = result #2 sr1 sl 1

    #40->la
    ld->xeo         #2->xo                          ld->yeq
                                    Y:EndShift->pco X:#38->la
    X:ld->gpr3                                      X:ld->asr
    X:sr1->gpr0                                     X:gpr3->sl
    X:sr1->gpr1

; Restore first result.
    #37->la
    ld->lo          #3->and                         #36->la
    ld->gpr2                        lr->yeq         ld->gpr6
                    Y:gpr2->lsr                     !Y:gpr2->sl
    sr1->gpr2
                    Y:gpr2->sl      X:EndShift->pco
                    gpr6->xls
                    X&!Y:gpr2->asr                  !X&!Y:gpr2->lsr

; Restore second result.
    #39->la
    ld->lo          #3->and
                    gpr3->xls       lr->yeq
                    Y:gpr0->sl
                    X&!Y:gpr2->asr                  !X&!Y:gpr2->lsr
EndShift:
```

```
Logical:
; gpr0 = count
; gpr1 = operand

    #35->la
    ld->lo                                                  ld->gpr1
    #34->la
    ld->xeo         #2->xo                                  ld->gpr0
    X:#33->la
                                                            X:ld->xor
    gpr0->xgt
    X:#32->la
                                      #0->lo                X:ld->or
    gpr0->xeo       #2->xo
    gpr1->lo                                                X:lr->xor


Integer1:
; gpr0 = count
; gpr1 = operand

    #31->la
    ld->io1                                                 ld->gpr1
    #30->la
    ld->xeo         #2->xo                                  ld->gpr0
    X:#29->la
    X:ld->sub1
    gpr0->xgt
    X:#28->la
    X:ld->add1                                              X:#0->io1
    gpr0->xeo       #2->xo
    gpr1->io1
    X:ir1->add1


Integer2:
; gpr0 = operand
; gpr1 = count
    #7->la
    #6->la                                                  ld->gpr0
    ld->xeo         gpr0->io2         #2->xo                ld->gpr1
    X:#5->la
                                                            X:ld->sub2
    gpr1->xgt
    X:#4->la
                                      #0->io2               X:ld->add2
    gpr1->xeo       gpr0->io2         #2->xo
                                                            X:ir2->add2
```

```
CompareX:
    #14->la
    ld->xeo         #2->xo                          ld->yeq
    #15->la         Y:CompareXEnd->pco
    ld->xgt                                         #16->la
    X:ld->xgt                                       !X:ld->ones

    X:ones->ones
CompareXEnd:


CompareY:
    #1->la
    ld->gpr2                                        ld->yeq
    #2->la
    Y:CompareY0->pco
    ones->ones                                      Y:ld->ygt
    ones->ones

    #2->yo                                          gpr2->yeo
    ld->ygt         CompareYEnd->pco
                                                    #3->la
    !Y:ld->ones                                     Y:ld->ygt

CompareY0:
    Y:ones->ones

CompareYEnd:


RegisterFile:
    #17->la
    #18->la
    #19->la                                         ld->gpr0
    #20->la                                         ld->gpr1
    #25->la                                         ld->gpr4
    #26->la                         gpr4->gpr2      ld->gpr5
    #21->la                         gpr5->gpr3      ld->gpr6
    #22->la                         gpr6->gpr8      ld->gpr7
    #23->la                         gpr7->gpr9      ld->gpr4
    #24->la                                         ld->gpr5
    #27->la                                         ld->gpr6
                                                    ld->gpr7
    ld->gpr10


IFetch_LoadStore:
```

```
; Address 42 contains the address of the last handler.
    #0->la
    ld->sa
    #42->la
    ld->pco
    ones->ones
    ones->ones

; Make three routines depending on interrupt/trap and XXX/XUX for loading
; 0,1 or 2 data's into the load-store unit.
XUXint2load:
    #10->la
    #11->la
    #12->la         pcc->ones                          ld->pc
    #7->la          pcc->ones                          ld->pc
    #8->la          pcc->ones                          ld->pc

XUXint1load:
    #10->la
    #11->la         pcc->ones                          ld->pc
    #12->la         pcc->ones                          ld->pc
    #7->la          pcc->ones                          ld->pc

XUXint0load:
    #10->la
    #11->la         pcc->ones                          ld->pc
    #12->la         pcc->ones                          ld->pc
                    pcc->ones                          ld->pc


XXXint2load:
    pcc->pc                                            #7->la
    pcc->pc                                            #8->la
    pcc->pc


XUXtrap2load
    #10->la
    #11->la         pcc->ones       XUXtrap2loadLast->pco
    #7->la          pcc->ones                          ld->pc
    #8->la          pcc->ones                          ld->pc
XUXtrap2loadLast:
    ones->ones


XXXtrap2load:
    #21->la
                    pcc->gpr4       XXXtrap2loadLast->pco
```

```
    gpr4->pc        pcc->gpr4
    gpr4->pc        pcc->ones                           #7->la
XXXtrap2loadLast:
    #8->la                                              ld->gpr4
```

D.7 Writing an exception handler

When writing an exception handler, several things should be thought of:

- Know what kind of handler you are writing. A XXX or XUX handler and an interrupt or a trap handler. Each of these four possibilities results in different code.
- When writing a XUX handler, make sure the exception stack pointer is correctly initialized and know whether the exception stack grows up or down.
- In user mode, one cannot rely on the contents of the BESF.
- Inside a XXX handler, don't overwrite the BESF, unless the value will be restored or the trap handler uses a register to return a value.
- When using a register as parameter to a trap routine, the contents of that specific register should be restored from the BESF because the register contents can be overwritten during the state saving.
- For short handlers, not the whole state has to be saved. This is especially true for short XXX handlers.

E

Chip and Packaging characteristics

MOVE32INT is implemented in a Sea of Gates (SoG) technology. The SoG is designed by Paul Stravers at the Electronics Department from the Electrical Engineering Faculty of the Delft University of Technology. The SoG chips are fabricated by Philips (Nijmegen), and metalized by DIMES (Delft). The characteristics of the used SoG chip are listed in table E.1.

characteristic	value
size	1 x 1 cm^2
number of transistors	180 k
rows (N+P transistors)	88
columns	1024
metal layers	2
number of pads	144
transistor gate length	1.6 μ
minimal feature size	2.0 μ
N transistor width	23.2 μm
P transistor width	29.6 μm
$V_{threshold}$(N)	0.7 V
$V_{threshold}$(P)	-1.1 V
β(N)	1081 $\frac{\mu A}{V^2}$
β(P)	542 $\frac{\mu A}{V^2}$

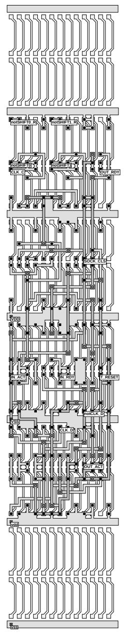


Figure E.1: Sea of Gates characteristics and implementation example of the stage controller

Figure E.1 also displays part of the SoGs image; the realization of the stage controllers used in the hybrid pipelined function units. Clearly visible is the SoG transistor pattern, consisting of rows of p- and n-transistors (p-transistors are the larger ones).

The floorplanning of *MOVE32INT* is shown in figure E.2. The top half contains the 4 move data busses

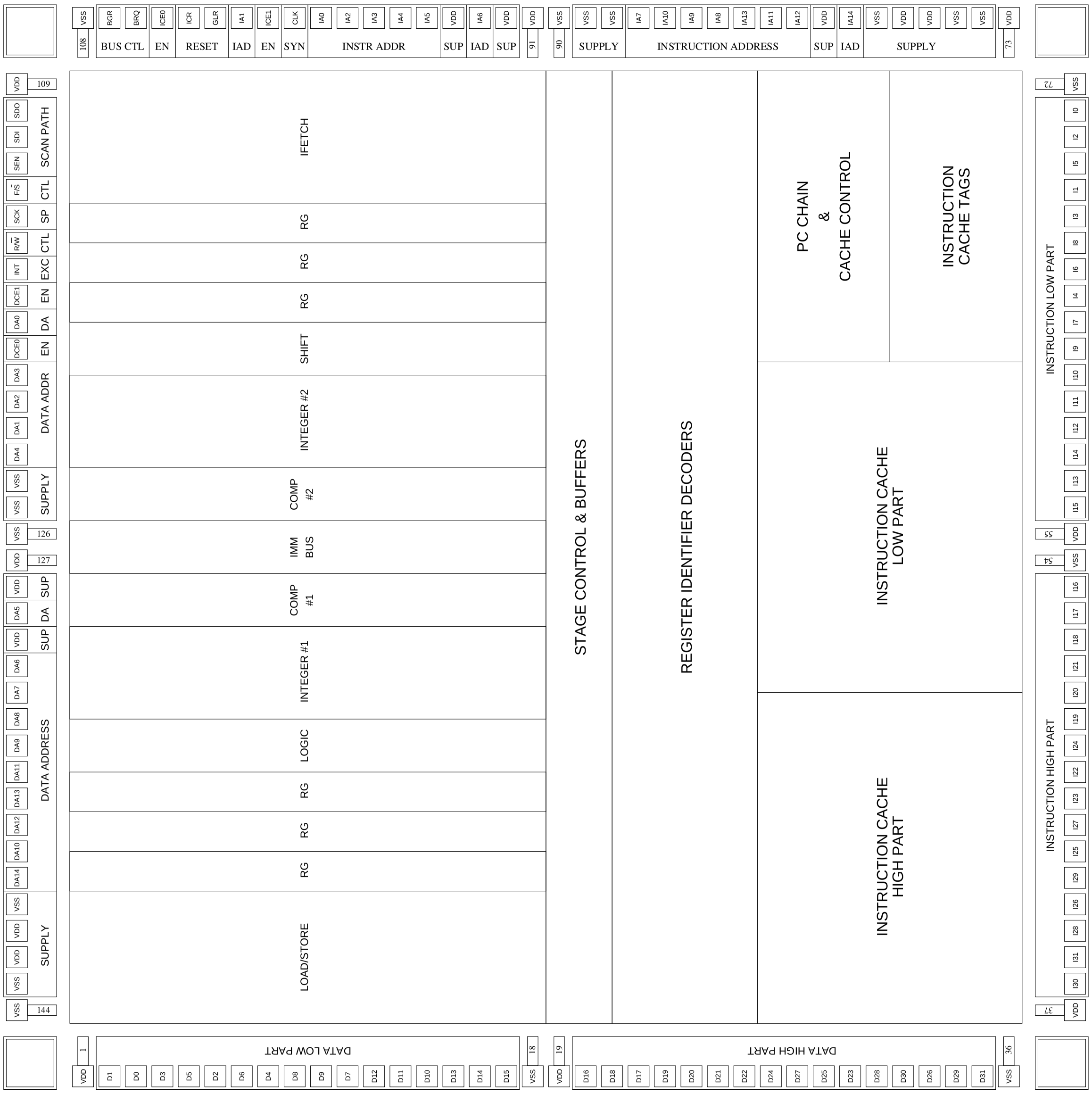


Figure E.2: Floorplanning of *MOVE32INT*

on top of which most of the function units are positioned. The lower part contains Icache data and tags parts. In between the 4 control busses (mainly source and destination-id busses) are placed.

Figure E.3 shows the bottom view of the 144 pin ceramic pingrid array in which *MOVE32INT* is packaged.

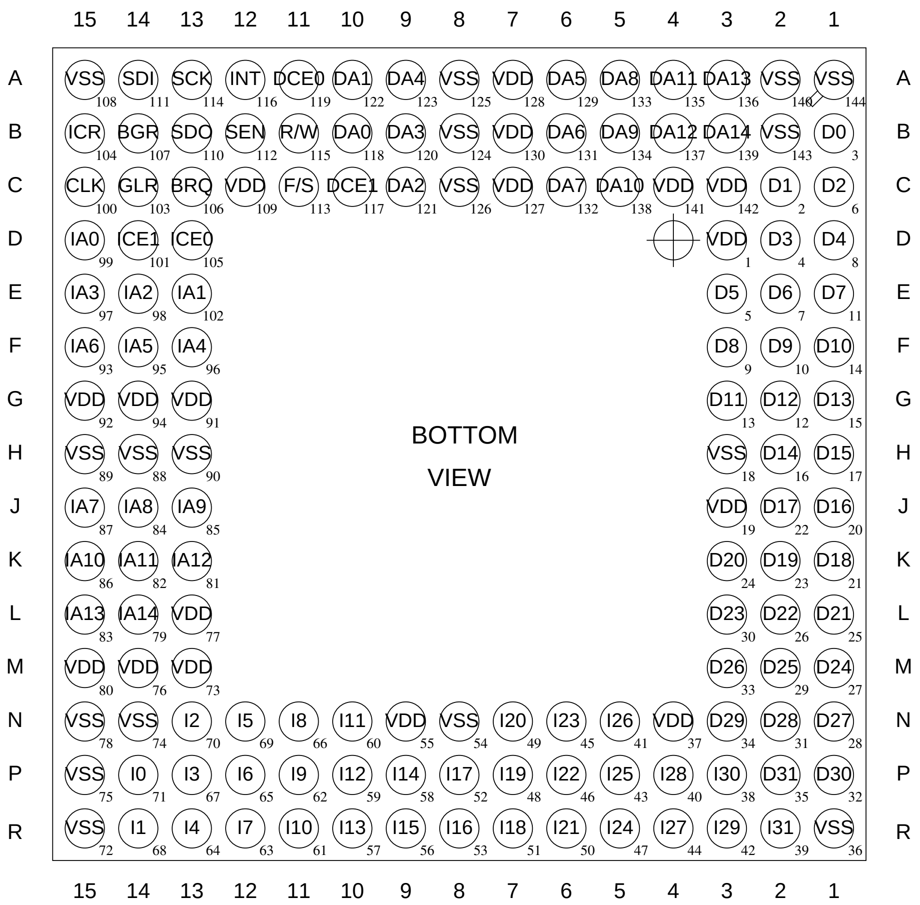


Figure E.3: Bottom view of the pingrid array of *MOVE32INT*