MOVE: A framework for high-performance processor design

Henk Corporaal Hans (J.M.) Mulder

Delft University of Technology, Department of Electrical Engineering
POBox 5031, 2600 GA Delft, The Netherlands

Abstract

CPU architectures allowing exploitation of instruction-level parallelism (pipelining, VLIW, superscalar) exhibit poor average hardware utilization and complex hardware organizations. We propose a different architecture with a less complex hardware organization and significantly better hardware utilization. This architecture allows short cycle implementations, and is easily configurable in functionality and scalable in performance. Besides being suitable for high-performance general-purpose computing, the architecture is particularly useful as framework for the (semi) automatic generation of high-performance processors for specific applications.

The key feature of this architecture is the separation of data transport and data operations. This separation yields a degree of autonomicity for the different operational units which allows, for example, different units to implement different pipelining schemes. Furthermore this separation allows the machine cycle to be reduced to the transport delay. The resulting programming model is quite unusual and provides scheduling possibilities unexploitable in the aforementioned architectures. We believe this architecture will be a favorable candidate for future powerful computing systems.

1 Introduction

One of the most influential developments in designing computing systems has been the change from CISC to RISC architecture principles [1]. The primary lesson we learned was that extra hardware functionality (e.g. for implementing complex instructions) does not always increase execution speed; on the contrary, it may slow down the overall computation. Possible reasons for this slow down are:

1. Extra functionality may add to the critical timing path within a processor, increasing the cycle time.
2. Complex instructions are seldom used in general applications. Hardware spent on these instructions may be more effective for other enhancements.
3. Complex instructions are difficult to pipeline.
4. Sometimes software solutions using simpler instructions are faster than corresponding hardware implementations; this is especially the case where compile time optimizations may eliminate part of the software overhead.
5. Complex hardware increases the design time and makes use of the latest technological improvements difficult.

Apart from this, the implementation of complex functionality decreases design flexibility. Complex hardware may have a major impact on the total design, making future enhancement more difficult.

By now, most manufacturers deliver RISC based computer systems with more or less similar performance, which is primarily dominated by the implementation technology used. Because the core of a RISC processor is rather small, it can also easy be integrated in a VLSI design environment for the creation of application specific processors (e.g., see [2]). Theoretically a RISC-based processor can sustain a performance of 1 CPI (cycle per instruction). In practice however, the CPI is somewhat greater because of branching delay and load interlocking. The efficient use of RISC systems requires advanced compilers, which reduce possible load and branch penalties by scheduling instructions into delay slots. This can be viewed as a limited form of parallel instruction scheduling.

In order to get even more performance from a single processor than by just adapting to faster technologies we distinguish two important architectural techniques for implementing parallelism a the instruction level: 1) the use of deeply pipelined functional units and 2) the use of more independent functional units. The first technique reduces the cycle time, the latter reduces CPI below one. Both techniques require extensive parallelization of code at the instruction level. Advanced compiler scheduling techniques are under development to allow for this degree of parallelization [3,4,5,6,7,8,9,10,11]. The next section will discuss these architectural developments. As will be explained, the resulting architectures show a number of disadvantages which severly limit their applicability.

The purpose of this paper is to introduce a totally different architecture, called the MOVE architecture, which fully separates data transport and data operation. Section three explains the concepts of this architecture. Next we compare the MOVE architecture with the recent architectural developments (introduced in section 2) and show its superiority for many different aspects. The MOVE architecture is part of an architecture framework for the automatic generation of application specific processor systems containing one or more specialized MOVE processor nodes. The final section describes the goals and status of this framework and states several concluding remarks.

2 Recent developments in computer architecture

In order to increase execution speed of single processing nodes several architectural developments take place. They are based on two design techniques [12]:

Superpipelining: Change the architecture in order to decrease the achievable cycle time by subdividing the already existing pipeline stages. This is an extension of the pipelining principle used in RISC architectures. While in RISC machines the execution stage of most instructions takes 1 cycle, superpipelined machines split up this cycle. RISC principles brought down the CPI close to one. Superpipelining however, will more or less increase CPI, because of added delay slots. The idea is that this slight increase is more than compensated for by the decrease in cycle time (e.g., Cray, Intel I860).

Functional parallelism: Add independent functional units in order to increase operation level parallelism and decrease effective CPI below one. Two representatives using functional parallelism are Superscalar and Very Long Instruction Word (VLIW) architectures. The former aims at dynamic detection of operation level parallelism (e.g., RS6000, Intel 960). The latter is totally dependent on compile time scheduling of operations. VLIW examples are described in [13,14,15,16,17]. In principle a Superscalar architecture can run unscheduled code and is therefore object code compatible with the same architecture without detection of runtime parallelism. However, getting a reasonable amount of parallelism requires hand coding or compiler support.

Superscalars are severly limited in the amount of exploitable parallelism. The hardware needed for runtime parallelism detection limits the allowable window detection size to a few instructions. Further hardware reduction is achieved by restricting the possible combinations of parallel instructions (e.g. one integer, one floating point, one load/store and one control instruction). However, still quite a bit of instruction control hardware is spent on work for which the compiler could do a far better job. A possible exception is the disambiguation of memory references. A favorable consequence of dynamic scheduling is that code size does not increase by unnecessary empty instruction slots. This means that superscalar techniques can certainly be used to upgrade existing architectures; they give moderate speed improvements, but need active compiler support in order to get this speedup.

VLIW and superpipelining architectures have a higher potential for parallel execution than superscalars. Note that from a compilers perspective a VLIW and superpipeline behave similar in the sense that for both architectures the compiler has to search for independent operations which can be scheduled together into one (VLIW) instruction or in a pipelined fashion. The advantage of these architectures for general purpose computation will probably not be big, because of the large sequential code component of those computations. However, in vector and other special purpose applications they may show large speedups.

Ideally we would like to combine VLIW and superpipelining principles, which would lead to architectures which deliver compatible performance for vector applications (as compared to vector machines), may be tailored to special applications, and which also give adequate scalar performance (which is a must for vector applications too). However, as we will explain after introducing the MOVE concept, these architectures exhibit some disadvantages which result in complex organizations, poor hardware utilization, difficult to change functionality, and poor performance scalability. This means that they are not the ideal object for being part of a framework for high-performance processor design.

3 The MOVE concept

For VLIW the number and type of function units (FUs) and for superpipelined the depth and type are determined according to the expected need of the application domain, the performance required, and the allowable design and fabrication costs. The communication bandwidth between registers and FUs depends solely on the number of FUs which can accessed concurrently. Similarly the instruction bandwidth required is based on this communication bandwidth and is therefore also designed to service worst-case requirements. Even when all function units are utilized, however, the worst-case communication bandwidth rarely occurs.

In this section we propose an architecture concept in which the problems of underutilization are strongly reduced. In the following subsection we develop the MOVE concept from VLIW and superpipelining. Subsequent subsections deal with the issues of pipelining function units, the generation and handling of exceptions, and with support mechanisms to allow efficient mapping of high-level languages and operating systems.

3.1 From VLIW to MOVE

Taking VLIW and superpipelining as point of departure, the development of the MOVE concept can be viewed as a six stage process: 1) reducing redundant register-file communication; 2) programming the bypass; 3) reducing bypass read-and-write connections; 4) separating transport from operation; 5) reducing transport capacity; 6) programming the transport.

Reducing redundant register-file communication Figure 1a shows the internal communication structure of a superpipelined architecture. A set of FUs is connected to a bypass unit containing operand and bypass latches. The bypass unit is connected to a register file. A small clock cycle provides a high communication throughput. A superpipelined VLIW organization is shown in Figure 1b; this organization provides communication throughput by means of both width (multiple busses) and time (small clock cycle). In the following discussion we will use this VLIW organization as example.

The bypass unit consists of operand latches, and bypass latches organized as a result FIFO for every FU. The operand latches can be written by the register file, by the FIFO, and by the FU output

Figure 1: Register-FU-bypass communication for (super) pipelined and VLIW organizations.

ports. The FU output ports also write the result FIFO. Writing can occur in all FIFO latches to allow FUs of different latency to deliver their result. Writing a result also implies writing the register identifier of the result. This identifier is used to deliver the result in the correct register en to be able to identify the result in the FIFO. A result which leaves the FIFO is written back in the register file. Reading the bypass is architecturally transparant: the identifier of the requested register is matched with the identifiers in the bypass; if a match occurs the bypass value is transferred to the operand latch instead of the value from the register file.

With respect to the bandwidth between register file and bypass we can make three observations:

1. Values written from the bypass into the register file may be dead. Measurements indicate that many generated results are consumed immediately. Deeper FIFOs enlarge this number. The bandwidth of the write port into the register file is therefore largely redundant.
2. Operations requiring one or two operands under utilize the register-bypass bandwidth (e.g., monadic operations and branches).

However, the compiler is able to detect all cases of under-utilization. An organization can be constructed which only saves replaced bypass values under program control. As shown in Figure 2a, the register file now becomes an FU (called the Regiser Unit, RU).

Programming the bypass. Now it becomes more efficient to make the bypass visible in the architecture. Instead of a transparent register cache, the bypass becomes an explicitly addressed first-level of the memory hierarchy. Spilling of bypass values into the register FU all happens under program control. Figure 2b shows the resulting organization: no FIFO replacement, all registers[1] can be written and read by all FUs. The main advantages of this organization are:

- more efficient use of bypass storage. If necessary temporaries can stay longer in the bypass. This may reduce the number of ports of the register file (see Figure 2b).
- simplified design of the register file.
- simplified design of the bypass (no FIFO, no id matching).
- smaller operand-identifier fields in the instructions.

Reducing bypass read-and-write connections. Because reading from and writing into the bypass is part of the FU execution, this communication must be as efficient as possible. One way of reducing the impact of reading and writing on the machine cycle is reducing the read and/or write load. Applying this principle on the visible bypass in Figure 2b for read connections and write connections yields Figure 2c and d respectively.

The simplest way to reduce read connections is to provide the operand inputs of every FU with a private set of registers which can be written by all FUs (Figure 2c). The LIFE processor [14] incorporates this organization. To avoid having multiple write connections on its register file LIFE allows only a single FU to write into a file at a given time.

Another way of reducing write connections is to provide separate bypass registers for every FU. The CYDRA-5 processor [17] incorporates this organization. Figure 2d shows an organization with two private result registers per FU[2].

Separating transport from operation. Reducing both read and write connections is shown in Figure 3a. All bypass registers are made private as FU operand and result latches and are integrated with the FUs. Because operand and result latches are local to a FU, the whole of the operations is now concentrated in the FU. As a result operations and transport are separated.

Reducing transport capacity. The transport network in Figure 3a is still under utilized when a particular function unit does not provide a result at a certain time. To improve utilization the number of busses can be reduced. Depending on the scheduling freedom required an FU can write on a single or on multiple

[1] Note that now the bypass latches are called bypass registers

[2] CYDRA-5 also reduces the number of physical read ports by providing each FU with a private copy of the bypass registers; this is of course an hardware-expensive solution

Figure 2: Making the bypass architecturally visible.

busses. Figure 3b shows an organization where three FUs share two general-purpose interconnecting busses.

The most significant implication of this organization is that both scheduling of operations and of transport is required. When we view this organization as derived from VLIW, the operations are scheduled by the compiler on specific FUs and the transport network is scheduled dynamically by the hardware.

Programming transport. The next logical step is to make the compiler do the scheduling of the transport. The compiler potentially outperforms scheduling hardware because of a larger scheduling scope than (practically) possible in hardware. Making the transport visible in the architecture, implies that the operations can be hidden: data transports (or moves) can trigger the operations. Now we have mirrored the traditional programming model:

Traditional:	Programmed operations Operation-triggered transport.
MOVE:	Programmed transport Transport-triggered operations.

Traditional architectures are *operation triggered* on the other hand MOVE-based architectures can be called *transport triggered*. The MOVE architecture based on the MOVE concept can be viewed as a register set interconnected by means of a point-to-point and/or point-to-multi network (see Figure 3c). The register set contains FU-input and FU-output registers, and general-purpose registers. The input registers are split in two types: operand registers and trigger registers. The first can be viewed as a general-purpose registers except for the fact that its contents is used as operand when its corresponding trigger is written. Consequently, the MOVE programming model only contains one instruction: the *move*. A *move* contains at least a source and destination identifier. A *move* always specifies a transport from one register to another, potentially causing the triggering of an operation. Viewed traditionally, a register-to-register addition takes three *moves*; the first to set the operand, the second to trigger the addition, the third to read the result. Most of the moves, however, will not be from register to operand/trigger, but from result to operand/trigger.

The network is not limited in any sense as long as register-to-register transports can be specified. The network is not required to have full connectivity. It is the task of the compiler to optimize the transport given a certain topology. Limiting the connectivity is an important tool in reducing bus loads, yielding faster transport times.

The MOVE prototype, which is currently under construction, contains a regular fully interconnected (multi) bus structure, allowing four concurrent *moves*.

3.2 Pipelining alternatives

The MOVE architecture does not constrain the pipelined implementation of FUs in any way. Optimal FU usage means maximal FU pipelining. In implementing pipelines there are some alternatives:

- Continue always. In this case data stored in a pipeline always continues to the following stage on each clock cycle. This is implemented in the Cydra 5 VLIW processor [17] and all current vector machines. This is easy to implement but requires large register usage as will be explained in the section on exceptions.
- Push or pull. Here the pipeline only continues in case a new instruction is issued which uses the pipeline (e.g., Intel i860). Note that the distinction between push and pull only makes sense for transport-triggered architectures.
- Hybrid pipeline. We can also implement a hybrid solution, where pipeline values may continue always, except when the result values are not read in time. Pipeline stages block as late as possible without overwriting results from previous started executions.

In the MOVE architecture we have chosen the hybrid alternative. The first solution requires that results have to be read exactly at

Figure 3: Separating transport from operations.

the right cycle. Besides severly limiting the scheduling freedom, on which we will come back in section 4, this alternative has to be excluded because results get lost in case of locking (e.g. instruction cache miss) or an exception. The FU has no idea where to leave its results[3].

The second alternative leads to large code sizes for scalar applications. If the pipelines are not used at maximum throughput extra instructions have to be scheduled in order to get the results out of the pipeline.

The last alternative is more difficult to implement because stages may continue only in case there is an empty stage ahead, but combines easy exception handling with small code size and easy scheduling of instructions.

3.3 Exceptions

Exceptions can be divided in three categories:

- 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).
- Imprecise exceptions. These do not require source operands to be available upon recovery. Examples are integer arithmetic exceptions [4].
- Fatal exceptions. In contrast to the former two, these exceptions are not recoverable.

Precise operations are partly implemented by the use of the so-called inexact exception condition. Most FUs try as soon as possible to verify if possibly an exception might occur. Before it knows this condition, and also incase this condition is met, it has to be certain that source operands are intact. This can be arranged by a combination of methods: (1) by compiler convention, (2) by locking directly until it is known that either the inexact condition has not been met, or it is known that a real exception happens, (3) by locking only when source operands are about to being modified, or (4) by saving source operands in the FU itself. It will be clear that solution 3, together with a quick generation of the inexact exception condition is preferable, which is what we implement in the MOVE prototype.

In recovering from exceptions there are in general 3 strategies: (1) restart, (2) complete and (3) block the current instruction.

Restart When we have multicycle operations with out of order completion, restarting unfinished instructions is almost impossible. This would need hardware to backup the PC and processor state for many cycles.

Completion Many superpipelined processors are designed to continue always. They are not able to block their FUs during exceptions [5]. As a consequence, they have to complete all unfinished instructions. This implies that the compiler has to enforce that a destination register of a multicycle operation is not used by other operations during execution of its value. The situation gets even worse for precise exceptions, in which case we must also require that source operands are not overwritten during execution of the operation on these source operands. In Cydra 5 [17] this is implemented by means of a compiler convention: a register is defined to be in use, from the beginning of the operation that writes to it until the latest completion time of all operations using that value. For software pipelined loops this leads to a disastrous

[3] A solution would be the use of *run out* registers. They are advantageous both for fast exceptions, and they allow the use of wave-pipelined FUs.

[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] An exception is the Intel i860, where pipelines have to be pushed or pulled by instructions in order to continue.

Figure 4: Support mechanisms for the MOVE architecture.

The MOVE concept separates transport from operation and implements programmed execution solely by specifying register-register moves. A processor based on the MOVE concept will require additional mechanisms to support high-level languages and operating systems. Figure 4 shows both the move mechanism and the supporting mechanisms. The move mechanism consists of the datatransport network and the move-identifier bus. The supporting mechanisms are the guard, locking, and exception mechanisms.

Conditional execution Implementing conditional execution solely by means of a compare and branch mechanism is less acceptable for VLIW and superpipelined machines. A better approach concerns the use of guards, allowing every move to be executed conditionally. Using this approach, every move includes a guard selector. This selector can select a boolean register or an expression of multiple boolean registers. The interpretation of the guard is FU dependent. Usually the semantics of a move with a false guard is that the move did not occur.

Setting the boolean (or condition) registers occurs by means of compare operations in the guard FU. The boolean values are combined according to the guard specifier into a single boolean value (the guard) per move. The MOVE prototype contains two boolean registers (x, y), and a three-bits guard specifier allowing $true, x, \overline{x}, y, \overline{y}, x\&y, \overline{x}\&y, x\&\overline{y}$ as guards. The guard FU implements $a = b, a = 0, a > 0, a \geq 0$ to generate x and y. More complex compares (e.g., $a > b$) require a preceding subtract (e.g., $a - b > 0$).

Locking 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 locks the consumer until the result is ready.

Within the move concept locking reduces to two types of transport locks: 1) read lock, a result is still pending; 2) write lock, a FU is not yet available. The first allows the scheduling of result reads independent of the FU latency; the second allows efficient implementation of precise exceptions. Locking stretches the move cycle until the locking condition disappears.

Exceptions The exception mechanism allows FUs to signal the three types of exceptions described in Section 3.3. A recoverable exception implies that the current move is not completed, but can potentially be restarted after the handling of the exception. The trap and return mechanism depends on the implementation of the exception FU. The exception trap and dispatch overhead for many exceptions can be handled very quickly by using the move read-or-write identifier for immediate generation of the trap addresses.

4 Comparing architecture trends with MOVE

The former two sections described the recent architectural developments and our alternative, the MOVE architecture. In this section we will compare these architectures from different points of view in order to justify our claims made in the introduction about the MOVE architecture. Comparisons are made on 1) hardware utilization, 2) performance aspects like speed and code size, 3) suitability for retargeting MOVE for specific applications,

4.1 Hardware utilization

In judging the MOVE concept w.r.t. hardware usage we have to look at the different components of a MOVE node. As illustrated

in Figure 3c MOVE node consists of a transport network and function units. Registers can be described as function units, but we will look at their usage separately.

Transport Network. As explained in section 3, a MOVE processor uses its internal datatransport capacity far more efficient then a comparable traditional VLIW processor. Roughly this means that with the same amount of metal spent on the transport network we can serve more function units and, therefore, allow more parallelisme to be exploited. This also leads to easier integration onto a single chip.

Function Units. In the MOVE architecture function units (FUs) are separated almost completely from the datatransport network. They have to obey only to the FU interface description. This means that we can pipeline each FU in the most optimal way. The only restriction is that the FU injection cycle time must be a multiple of the data-transport cycle time[6]. Remark that a traditional VLIW has to pipeline their FUs to reach single cycle throughput in order to use both its busses and instruction-code slots optimally.

Register Use. The amount of registers needed can be reduced drastically in MOVE because of three reasons:

- Less temporaries have to be stored. Many temporaries go directly from FU to FU without being stored into a general purpose register.
- Pipeline stages and operand registers are used as temporary registers. Because in the MOVE concept an elementary RISC instruction is decoupled into 3 components which can be scheduled independently, a result produced by a FU, but not directly usable by another FU can temporarily be left in the producing FU, at least if it does not block other results in the same FU which are needed earlier. In the same fashion we can allocate one of the source operand registers for a temporary which is used for an operation later on.
- Blocking exceptions. We have chosen for pipeline stages within a FU to block automatically when results are not removed in time. Therefore, we do not have to enforce a compiler convention which declares registers to be in use for a long time. As explained earlier, in Cydra 5 [17], each register file had to be increased with many registers in order to guarantee precise exceptions.

4.2 Performance issues

The proof of the pudding is the eating; a real performance comparison of architectures should be based on real benchmarks and real machines. However since we have not yet finished our prototype implementation, we can only give a qualitative comparison. We will look at cycle time constraints, scheduling freedom, code size, and discuss the suitability for scalar and vector code.

[6]This even makes wave-pipelined FUs possible

Cycle time Most RISC based computer systems use separate pipeline stages for instruction fetch, ALU, and memory access. The minimal cycle time of these computer systems is constrained by either cache access or by the ALU stage. Both the cache access and ALU stage can be superpipelined (split into more stages). The cache access can easily be split into a decode, lookup and tag compare stage. We could even use an interleaved cache organization if cache lookup becomes a bottleneck (because of to many cache lines).

The ALU stage consists of reading the source operand latches, going through the combinatorial logic, and next through the bypass multiplexers. In principal we can put latches inside the logic and before the multiplexers. As noted by Kunkel and Smith [18], the minimum amount of logic between stages depends on the amount of data and clock skew. The time to write through the multiplexer depends linearly on the load to be driven , and the load is proportional to the number of drivers; so this time is $\theta(n)$, where n equal the number of drivers [7]. Driving a value from n sources could well turn out to be the bottleneck if we really go to heavily pipelined machines. This means reading the register file (which is also a n to 1 transport) or going through a complex bypass circuit dictates the cycle time. Besides careful designing in order to reduce data and clock skew to an absolute minimum, the size of the register file and bypass have to be reduced. As noted, the MOVE architecture is superior in reducing the number of necessary registers. The bypass circuit has been replaced by the transport network. The number of connections in this network is far less as compared to an equivalent VLIW architecture. So we think that in case this transport will become the bottleneck (which still has to be verified with our prototype), the MOVE architecture has no real contender.

Scheduling freedom The performance of a VLIW processor is critically dependent on the quality of code scheduling. In the MOVE architecture, we have separated a RISC operation into 2 or more data transport operations (called moves). E.g. a 3 operand RISC instruction can be compared to 3 moves, one to move the first source operand (operand move), one to move the second (and final) source operand (trigger move), and one to move the result to another destination (result move). The instruction which moves the final source operand must also trigger the operation. Combined with the hybrid pipeline solution for FUs, this opens new scheduling degrees of freedom not seen before (except in microarchitectures); both operand and result moves are decoupled from the trigger move. Separating a traditional instruction into its transport components gives several advantages:

1. Common subexpression elimination can be applied to these moves, that is, if source operands do not change in two successive calculations (using the same FU) we can eliminate the second operand move. For loops this often means that we can put several operand moves outside the loop body (e.g., operand moves used for address calculations and loop-condition comparison.).

[7]Remark it does not help to choose larger drivers, because driving power is proportional to its capacitive load. Using a driver tree however, reduces the time to $\theta(log(n))$, but a tree is far more difficult to route on VLSI.

2. Removing unnecessary moves to general purpose registers. This happens when a result can not directly be used for the next operation, e.g. when the second source operand of this operation is not yet available, the result may sometimes be moved to one of the source registers of the next operation, instead of being put in a register (this also saves a register). It may also be possible to postpone the result move. This gives the same advantages.

3. Getting more optimal initiation intervals in software pipelined loops. As noted in by Kogge [19], it is often desirable to add delay stages into pipeline reservation tables in order to reach the optimal initiation interval. For the MOVE architecture this translates into delaying result moves. This can of course only happen for those FUs which are not used at maximum throughput.

Code size

In going from CISC to RISC computers, code size increased approximately 50 %. This was acceptable given (1) the increased density of memory RAMs, and (2) the use of large instruction caches which compensate for the increased instruction traffic. However the use of VLIW architectures can easily lead to a code size explosion, because of both the large number of unused instruction slots and the code added by advanced scheduling techniques. In principal move instructions are even less dense than their 32-bit RISC counterparts; e.g., a three-operand instruction translates into 3 moves, where each move decodes in some way the corresponding FU. In our MOVE prototype, a move instruction takes 16 bits (see figure 5), which gives 48 bits for three moves; a code size increase of 50 %. Fortunately there are several factors in favor of the MOVE architecture:

1. Several RISC instructions expand in only two (a=-b) or one move (a=b), resulting in equal or half the RISC instruction size.

2. Many moves transport data directly from FU to FU. Ideally this gives 1.5 move per 3 operand RISC instruction; a code size decrease of 25 %.

3. Common subexpression elimination may remove unecessary operand moves.

4. Flexible division of move-idspace (imagine a RISC with 33 registers).

Our first prototype compiler, using only basic block scheduling without the mentioned elimination of operand moves, translated several large benchmarks with an average move count of 2.2 per RISC instruction. We expect future versions of the compiler to decrease this number below 2, which means that MOVE and RISC code densities are comparable.

Compared to VLIW systems the MOVE architecture is superior considering code size. First the number of unused slots is reduced because of the shared tranport network (see section 3), and second, completely empty instructions can often be removed because of the blocking pipeline concept. This is essential for scalar code.

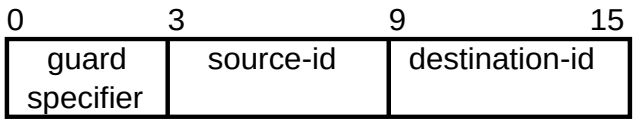

Figure 5: Move format of the MOVE prototype; instructions contain four moves.

Scalar and Vector applicability

The MOVE architecture is designed to be easy scalable in the number and type of the number of FUs. Combined with the almost unconstrained implementation possibilities of the FUs and the efficient use of the data transport network, this gives an ideal machine for vector code applications and all other kind of special purpose applications. A problem with many vector machines is that their scalar performance does not match their vector performance. Amdahl's law therefore restricts their usability for general vector applications. VLIW architectures combined with fancy compilers are able to increase both scalar and vector performance, although the scalar performance increase is limited to about a factor of 2 to 4.

The MOVE architecture improves on a standard VLIW in 2 aspects, (1) as noted, code size increase for scalar code is far less dramatic, and (2) extra FUs can easily be added without any change in transport capacity (and instruction format change). In standard VLIWs we tend to combine in a single FU multiple hardware-unrelated functions which are less often used in parallel (e.g., one unit for integer, logic, shift and control operations). In MOVE we can split them almost without costs. This creates extra opportunities for parallelism in scalar code[8]

Compared to a standard RISC processor it looks like the MOVE concept has an inherent disadvantage for scalar code.

RISC code	MOVE code
ADD R1, R2, R3	R2 → Radd-operand, R3 → Radd-trigger
SUB R4, R5, R1	.....
	Radd-result → Rsub
	.... etc.

Figure 6: Comparing scalar RISC and MOVE code

As shown in figure 6, between dependent operations we loose at least one cycle. Remind however, that the total amount of work to be done in both cases is about the same; going through the combinatoric logic and latch the result (see the discussion on cycle time). The difference is that in the MOVE architecture we latch 2 times i.s.o. 1 time. However the FU result latch can easily be combined with combinatoric logic (e.g., by using Earl or Polarity hold latch). Further, as already noted, going through the transport network is easier than going through the bypass. And

[8] Initial measurements indicate that the splitting of FUs increases the exploitable parallelism for several large benchmarks upto 30%.

Figure 7: MOVE architecture without the "lost cycle"

4.3 Suitability as Application Specific Processor (ASP)

Given the state-of-the-art silicon compilers, with all kinds of facilities of using parameterized VLSI cells, quick retargeting of a processor design becomes a realistic option. A processor is suitable for retargeting for a specific application if its architecture is both flexible and scalable, and its design is easy.

Flexibility Retargeting assumes flexible functionality, that is, the functionality can be easy adapted. Current VLIWs have a bad score on this point; adding function units implies: (1) change of instruction format, (2) adding extra busses (changing the whole register layout). The MOVE architecture does not show these disadvantages. There are also no constraints on the number of source and destination operands for a specific FU.

Performance Scalability Besides changing functionality, it may be desirable to adapt the performance. The MOVE architecture offers an extra degree of freedom in this respect. Besides adding FUs, we can (independently) increase the transport capacity. E.g. the transport network could be implemented as a couple of parallel busses which can be generated by a parameterized generic VLSI library cell. Adding a bus is easy, but changes the instruction format, although in a predictable way. We also may increase the pipelining degree of a specific (underpipelined) FU for getting more power.

Design Time In making ASPs the ultimate question is: what performance-cost ratio do we get. Design time highly influences this ratio. A long design time results into high design costs, and also a lower performance because the design is based on out of date implementation technology. By now the reader should be convinced that the number of design constraints dictated by the MOVE concept is low. Changing FUs, adding FUs, and changing the transport network can all be done independently. We therefore believe that this concept is ideal for quickly generating ASPs, using available VLSI compiler environments.

Quick Prototyping Within the MOVE concept, quick prototyping can be aided by means of an emulation mechanism. This mechanism allows systems to be built with some FUs emulated in software instead of implemented in hardware. This mechanism takes care of generating exceptions for all moves to and from not-implemented FUs. The mechanism is built into an FU consisting solely of an operand register per specifyable move bus. Both emulation exceptions already exist in the exception mechanism. The operand exceptions can be treated as a precise exception (source operands are available for emulation), the result exception is treated as an imprecise exception. By using quick dispatch on the move identifiers, emulation overhead can be reduced.

5 Status, Research and Conclusions

The former sections introduced the MOVE concept and compared it with current architectural trends for exploiting instruction level parallelism. The main conclusions which can be drawn are:

1. There is no general agreement about how to exploit instruction level parallelism.
2. Limited parallelism can be exploited using Superscalar architectures. In a sense these architectures do not adhere to the RISC principle that no hardware should be spent where a compiler could do a better job.
3. VLIW architectures in combination with superpipelined function units have a much higher parallelism potential, and can be adapted to boost performance of specific applications. However we have explored several disadvantages:
   - Change of functionality requires major architectural changes.
   - VLIWs require far to much transport capacity.
   - Code size may dramatically increase.
   - The architectural organization limits FU splitting, and therefore the exploitable parallelism.
4. The MOVE architecture does not suffer from the aforementioned disadvantages. Besides being flexible and scalable, it combines good scalar performance with tight code size. Its design is relative easy due to its separation of operations and transport. It is therefore an ideal architecture for the design of application specific processors.

5. The MOVE concept introduces an unusual programming model with extra scheduling possibilities not seen before. This may speed up both scalar and software-pipelined loops.

The MOVE architecture is part of a framework for the automatic generation of Application Specific Processors (ASPs). The process of generating ASPs from an application description should be viewed as a semi-automatic iterative process. An analyzing compiler detects parameters like exploitable parallelism, memory requirements, required functionality, number of registers, guards, and function units, and transport capacity and connectivity. Initial design parameters are chosen and used to automatically generate compiler and hardware. Evaluating the application with this compiler and hardware provides feedback for the next design iteration step.

Our compiler uses the GCC/G++ frontend to generate move code for a generic move architecture (with unlimited resources and a given functionality). Using list-scheduling techniques adapted for transport triggered architectures, we generate parallel code obeying resource constraints. The current compiler only schedules basic blocks. Developments are ongoing to extend scheduling to handle extended basic blocks. Finally we have to include some form of software pipelining as well. Available techniques have to be adapted to the move programming model.

References

[1] John L. Hennessy and David A. Patterson. *Computer Architecture, a quantitative approach*. Morgan Kaufmann publishers, 1990.

[2] J.M. Mulder et al. A framework for application-specific architecture design. In *Proceedings of the 14th International Symphosium on Computer Architecture*, May 1989.

[3] Alexander Aiken and Alexandru Nicolau. Optimal Loop Parallelization. In *Proceedings of the SIGPLAN'88 conference on Programming Language Design and Implementation*, pages 308–317, Atlanta, Georgia, June 1988.

[4] Kemal Ebcioğlu. A Compilation Technique for Software Pipelining of Loops with Conditional Jumps. In *Proceedings of the 20th Annual Workshop on Microprogramming*, December 1987.

[5] John R. Ellis. *Bulldog: A Compiler for VLIW Architectures. ACM Doctoral Dissertation Awards*, MIT Press, Cambridge, Massachusetts, 1986.

[6] Joseph A. Fisher. Trace Scheduling: A Technique for Global Microcode Compaction. *IEEE Transactions on Computers*, C-30(7):478–490, July 1981.

[7] Joseph A. Fisher, John R. Ellis, John C. Ruttenberg, and Alexandru Nicolau. Parallel Processing: A Smart Compiler and a Dumb Machine. In *Proceedings of the ACM SIGPLAN '84 Symposium on Compiler Construction*, pages 37–47, June 1984. SIGPLAN Notices Vol. 19, No. 6.

[8] Monica Lam. *A Systolic Array Optimizing Compiler. The Kluwer International Series in Engineering and Computer Science*, Kluwer Academic Publishers, Norwell, Massachusetts, 1989.

[9] Alexandru Nicolau. *Percolation Scheduling: A Parallel Compilation Technique*. Technical Report TR 85-678, Cornell University, Department of Computer Science, Cornell University, Ithaca, NY 14853, USA, May 1985.

[10] 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*, 12–35, January 1989.

[11] U. Schwiegelshohn, F. Gasperoni, and K. Ebcioğlu. On Optimal Loop Parallelization. In *Proceedings of the 22nd Annual International Workshop on Microprogramming and Microarchitecture*, Dublin, Ireland, August 1989.

[12] N.P. Jouppi and D.W. Wall. Available instruction-level parallelism for superscalar and superpipelined machines. In *Proc. 3th Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 272–282, April 1989.

[13] Kemal Ebcioğlu. Some Design Ideas for a VLIW Architecture for Sequential Natured Software. In *Proceedings of the IFIP WG 10.3 Working Conference on Parallel Processing*, pages 1–21, Pisa, Italy, April 1988.

[14] Junien Labrousse and Gerrit A. Slavenburg. A 50MHz microprocessor with a Very Long Instruction Word architecture. In *ISSCC '90*, February 1990.

[15] Trace: technical summary. Multiflow Computer Inc., June 1987.

[16] S. Borkar et al. Iwarp: an integrated solution to high-speed parallel computing. In *Proceedings of Supercomputing '88*, 88.

[17] B.R. Rau et al. The cydra 5 departmental supercomputer, design philosophies, decisions, and trade-offs. *IEEE Computer*, January 89.

[18] Steven R. Kunkel and James E. Smith. Optimal pipelining in supercomputers. In *ISCA-13*, pages 404–414, Tokyo, Japan, June 1986.

[19] Peter M. Kogge. *The Architecture of Pipelined Computers*. McGraw-Hill, 1981.