FACTA UNIVERSITATIS (NIŠ)
Series: Electronics and Energetics vol. 13, No.3, December 2000, 297-315

# PIPELINED PROCESSOR FOR PARALLEL INTERPRETATION

**Veljko Malbaša and Mark Manwaring**

**Abstract** The principles and design of a pipelined processor architecture for parallel interpretation of high-level machine languages is presented. A new instruction encoding method, that facilitates the design of pipelines, is used to design the instruction sets of the controller, memory, and execution units. Performance profiles of seven benchmark programs, obtained by using a cycle-level simulator, show the speedup of about two relative to equivalent processor for serial interpretation.

**Key words:** Pipelined processor architecture, high-level machine languages, minimally synchronized architecture.

## 1. Introduction

The goal of the parallel interpretation project is to overcome two serious problems imposed by traditional computer architectures: (i) inefficient support for a complex interpretation phase and (ii) classic interpretation techniques, based on a sequential fetch-execute model, are not very suitable for interpreting higher level, or abstract, machine languages, because of the phenomenon of "interpretive overhead", [4].

An innovative architecture for parallel interpretation of high-level machine languages has been described in [1]. This type of the architecture shares some commonality with a class of decoupled computer architectures, [3]. In the proposed architecture the interpretation process is decomposed into concurrent processes which are executed in parallel on a number of specialized processors which are connected by fast queues. The speedup resulting from the parallel operation is designed to counterbalance the penalty

Manuscript eingegangen April. 18, 2000.
The authors are with Schol of Electrical Engineering and Computer Scinece, Washington State University, Pullmann, WA, 99164-2752.

of interpretive overhead. This is a form of user transparent implicit parallelism. Details of the described architecture, called *Minimally Synchronized Architecture* (MSA) are given in the references [1], [5].

The performance of an MSA can be enhanced by incorporating other forms of parallelism, such as pipelining. In the paper we deal with the design and performance of a pipelined version of an MSA. A cycle level simulator for the pipelined MSA has been implemented. The simulator executes programs written in the machine code. Various architectural features, component speeds and queue lengths can be specified in a parameter file which is read by the simulator. The simulator generates various sets of data characterizing the execution process. These includes total parallel and serial execution time, execution time of each component, types and number of pipeline stalls, address traces et cetera. The estimate of serial time is based on the execution time of the same program on an equivalent serial machine. The benchmark programs that are run on the simulator are chosen to represent different types of loads. The performance measurements are presented and compared with equivalent serial machine.

The paper is organized in the following way. In the next section we analyze the problems imposed by pipelining to establish the background for the design of the pipelined MSA. An instruction encoding scheme, called *Distributed Encoding Scheme*, that strives to minimize structural complexity of the MSA instruction set while retaining the compactness of a complex instruction set is presented. After that we give some of the details of the pipelined MSA and the instruction set. At the end of the paper some of the performance measurement results obtained by running representative benchmark programs on the implemented cycle level simulator are presented and discussed.

## 2. Pipelining

Pipelining increases computer performance by overlapping the execution of multiple instructions, [2]. This feat is accomplished by dicing each instruction into basic operations and dedicating individual processing units (called *stages*) to each segment. The stages are connected with each other, usually via a staging register, and for a pipe. The number of stages present in a pipeline is called the depth of the pipeline. Assuming that the execution time in each pipeline stage is equal, the lower bound of the execution time of each instruction in the pipelined unit, called the *ideal execution time* is equal to the time per instruction on equivalent non-pipelined machine divided by the number of the pipe stages.

*Hazards* are pathological conditions in the operation of a pipeline where an instruction is prevented from executing at its designated clock cycle. Three kinds of hazards can be identified: *Data, Structural*, and *Control hazards*.

A **data hazard** occurs whenever some data object within the computer (e.g., register, memory locations, flag) is accessed or modified by two separate instructions that are close enough for their execution to be overlapped in the pipeline. Let the *domain* $D_i$ of an instruction $i$ be the set of all objects (registers, memory locations, flags, etc.) whose contents may affect the execution of the instruction $i$, and *range* $R_i$ be the set of all objects whose contents may be modified by the execution of the instruction $I$. Three types of data hazards are distinguished between an instruction $i$ and a successive instruction $i+1$.

- RAW (Read-After-Write) hazard is possible is $R_i \bigcap D_{i+1} \neq \emptyset$. RAW hazards occurs when the instruction $i+1$ attempts to read an object which is modified by the instruction $i$ before the modification is complete.
- WAR (Write-After-Read) hazard is possible if $D_i \bigcap R_{i+1} \neq \emptyset$. WAR hazard is possible if instruction $i+1$ modifies some object before it is read by the instruction i.
- WAW (Write-After-Write) hazard is possible if $R_i \bigcap R_{i+1} \neq \emptyset$. This type of hazard occurs when both instructions $i$ and $i+1$ attempt to modify the same object but the instruction $i$'s modification occurs after that of the $j$'s.

The simplest technique to resolve the data hazards is to "stall" the pipeline if a hazard is found, i.e. if the instruction $i+1$ is found to have a hazard possibility with a previously issued instruction $i$ then the issuing of $i+1$, and all subsequent instructions is halted until such time when the hazard condition ceases to exist. A more ambitious solution is to stall the instruction $i+1$, but let subsequent instructions $i+2, i+3, \cdots$ proceed, if they are free of potential hazards. A solution specific to RAW hazards, is to directly forward the data produced by the instruction $i$, which is required by the instruction $i+1$, to the instruction $i+1$ before the execution of the instruction $i$ is complete. This is called *forwarding* or *short circuiting* and requires extra hardware for its implementation.

A **structural hazard** might rise in a situation where two or more instructions compete for the same hardware resources at a given clock cycle. For example, at a given clock cycle, the instruction $i$ might want to write a

result to the register bank, while the instruction $i+1$ might need to read the contents of a register. The usual solution is to let on of the competing instruction proceed while stalling the others. Depending on the frequency of a particular hazard, the designer might want to duplicate the resource of contention.

The **control hazards** happen in the presence of branch and jump instructions that disrupt the sequential flow of the instruction execution. The problem is that the target of a branch instruction become available only after the instruction is well into the pipeline. At this point several subsequent instructions have also been issued to the pipeline under the assumptions of linearity of execution. However, if the branch is now taken, this would require that the pipeline be "flushed" and the effects (if any) of the instructions already in the pipe, be undone, before the execution can be resumed from the new address. The simplest solution to this problem is to freeze the pipeline as soon as a branch or jump instruction is detected, which usually occurs during the decode stage. A more sophisticated technique involve predicting the outcome of the branch instruction and continuing the execution from a point based on the prediction. In case of a false prediction the pipeline has to be flushed.

## 3. Pipelined MSA

Pipelined and superscalar architectures were both conceived to augment processor performance by exploiting implicit parallelism. The nature of the parallelism that each exploits is, however, very different. Pipelining offers an economical way to realize the *temporal* parallelism, that is inherent in the process of instruction execution, by segmenting the process into consecutive subprocesses. Superscalar architectures, on the other hand, are said to exploit *spatial* parallelism, i.e., instruction level parallelism. These two orthogonal approaches can also be combined in the same processor to provide a higher degree of potential parallelism. An example which carries this to the extreme is the SIMP processor, [48] and [58], which consists of four identical instruction pipelines, each of which consists of five pipe stages thus enabling the processor, in theory, to attain an overall speedup of 20.

The parallelism exploited by an MSA is very different in flavor from that of temporal or spatial parallelism, and can be termed *structural parallelism*. It exploits the parallelism inherent in the structure of the interpretation process. The performance of an MSA architecture can also be enhanced by incorporating other forms of parallelism within its architecture: superscalar MSA or pipelined MSA. Furthermore, for various reasons, pipelined MSAs

are much easier to implement than pipelined superscalar architecture. In a pipelined superscalar architecture, the detection of pipeline hazards are made more complicated, and their effect on processor performance is further exacerbated by multiple instruction issue and out of order execution. This is specially true for control hazards, which has been blamed for the poor performance of the SIMP processor.

The combination of structural and temporal parallelism in pipelined MSA is more benign because in machines that realize structural parallelism, the logical ordering of instructions is violated during execution. Structural and data hazards are not compounded by the presence of multiple pipelined units because (i) each pipeline is separate physical entity and do not share any common resources and(ii) the source and destinations of each pipeline are logically separated. Control hazards still impose a hefty penalty on the performance by introducing pipeline stalls, but this can be alleviated by a combination of software (delayed branches, software branch prediction etc.) and hardware (speculative execution, boosted execution etc) solutions.

The Table 1 presents the pipeline stages of the controller, memory, and the execution units of an MSA.

Table 1

| **Controller pipeline** | **Memory pipeline** | **Execution unit pipeline** |
|---|---|---|
| Instruction fetch | Instruction fetch | Instruction fetch |
| Insstruction decode | Instruction decode | Instruction decode |
| Instruction formation | Address generation | Execute |
| Send instruction | Cache address | Send result |
| | End of operation | |

Note that in all of the three pipelines the range and domain of the instructions do **not** overlap. The following table gives the domain and range objects for the three units. In the table IC stands for Instruction Cache, MCQ for Memory to Controller Queue, CMQ for Controller to Memory Queue, CXQ for Controller fo Execution unit Queue, DC for Data Cache, XMQ for Execution unit to Memory Queue, and MXQ for Memory to Execution unit Queue, see Fig. 1 and Fig. 1, [5].

Table 2

| **Pipeline** | **Domain** | **Range** |
|---|---|---|
| Controller | IC, MCQ | CMQ, CXQ |
| Memory | DC, CMQ, XMQ | MXQ, MCQ, DC |
| Execution | CXQ, MXQ | XMQ |

The consequences of the logical and physical separation of the range and domain structures are profound, because it implies that there are **no** data hazards in these pipelines, and thus they do **not** suffer from the crippling effect of this type of hazards.

A quick inspection of the pipeline also reveals that these pipelines are devoid of structural hazards. The controller unit accesses the instruction cache only during one stage in the pipeline. Similarly, the memory unit has a single read/write access to the data cache at a given clock cycle. Memory accesses therefore can not cause structural hazards. At a given clock cycle the controller unit can perform at most three write operations, on to the memory queue and two writes to the execution queue. The memory queue is provided with one write and one read port, while the execution queue is provided with two write and one read port. This precludes any possible structural hazard resulting from access to the resources. Most multiple-pipelined superscalar architectures, in contrast, are characterized by an aggravation in the complexity of data and structural hazards when compared to their scalar counterparts.

Control hazards, however, are still present in this architecture, although their effect on performance is less pronounced when compared to superscalar architectures. The two controller instruction that generate this type of hazard in this architecture are the $GOTO$ instruction, which denote an unconditional transfer of control, and the $LOOP$ instruction wich is a conditional transfer instruction and whose outcome is predicated by the result of a previously issued relational instruction. This relational instruction is evaluated by the memory unit and its result is passed back to the controller unit via the memory-to-controller queue MCQ. The $GOTO$ instruction can be detected in the ID stage of the pipeline and hence only the next instruction (which is in the IF stage) needs to be flushed from the pipeline. This introduces a pipe stall of one clock cycle. The effect of the $LOOP$ instruction is more severe. The instruction can be detected as early as the ID stage, however the resolution of the branch is dependent on the evaluation of the relational instruction in the memory unit, and in the case when the relational instruction immediately precedes the branch instruction the evaluation can take from six to thousands of clock cycles (in case of a data cache read miss in the memory unit). These harsh effects can, however, be largely mitigated by code optimization techniques like delayed branch and loop unrolling. In most cases the branch penalty can be reduced to 1 cycle. A further source of branch penalty are the $CALL$ and $RETN$ instructions. In each case the branch penalty is exactly one clock cycle.

The preceding discussion substantiates the claim that pipelined MSAs are significantly less prone to pipeline hazards than superscalar architectures.

## 4. Instruction Set Complexity and Pipeline Design

Another important aspect of pipeline design is the effect of the structural complexity of an instruction set on pipeline performance. An instruction encoding scheme, called *Distributed Encoding Scheme (DES)*, that strives to minimize structural complexity of MSA instruction set while retaining the compactness of a complex instruction set is presented in this section. The central idea in DES is to decompose a complex instruction into several constituent parts, each of which encodes several operations from the operation set of the complete instruction, and is complex enough to fully utilize the resources of the pipeline. This structural unit of a complex instruction is called an *instruction parcel*. In DES, unlike conventional instruction formats where the opcode that identifies the functionality of the instruction is usually confined to the leading bytes (one or two) of the encoding, the information is distributed among the constituent instruction parcels.

The instruction parcels can be differentiated into two categories, which are called *leading* and *dependent* parcels. The leading instruction parcel is the first instruction parcel in a multi-parcel instruction. Subsequent parcels in the instruction fall in the category of dependent parcels. A single bit in the encoding differentiates between the two types. The encoding of a leading instruction parcel consists of the two fields, the *major opcode* field that identifies the instruction, and when decoded, provides the relevant semantic information that determines the operations to be performed on the operands specified in the second *operand* field. A dependent instruction parcel also consists of two fields. The leading field is called the *minor opcode* field, which is usually smaller that the major opcode field, and when combined with a specified segment of the corresponding major opcode field, provides the information required to process the operands that are provided in the operand fields that follow. The leading and dependent instructions have a fixed field encoding which facilitates their decoding.

The DES concept is illustrated by an example design of the instruction PEXPR, that specifies a polish expression, and that is the most complex instruction in the MSA instruction set. The expression can be arbitrary long, and can consist of an arbitrary combination of operands and operators in arbitrary order. It is assumed that 4 bits are required to specify an operator and that 24 bits are required to specify a literal values or the address of an operand. The major opcode is specified by 8 bits, the last four bits of

which is combined with the 4 bit minor opcode to yield the working opcode for dependent parcels. the instruction parcel is 32 bit long, which is the usual length for modern RISC type instructions. The instruction parcels for PEXPR that can be formulated under these constraints are given in the Table 3, where for each instruction the mnemonic name and the name and type of operand fields are given.

Table 3

| **Opcode** | **Field 1** | **Field 2** | **Description** |
|---|---|---|---|
| PSWR | VAR | | leading parcel, variable operand |
| PSVI | @VAR | | leading parcel, indirect var. operand |
| PSVL | VAL | | leading parcel, literal operand |
| PAVR | BASE | | leading parcel, structured variable |
| POVL | OPER | VAL | dependent parcel, operator and literal |
| PVLO | VAL | OPER | dependent parcel, literal and operator |
| POVR | OPER | VAR | dependent parcel, operator and operand |
| PVRO | VAR | OPER | dependent parcel, variable and operator |
| POVI | OPER | @VAR | dep. parcel, operator and indirect operand |
| PVAL | VAL | | dependent parcel, literal operand |
| PVAR | VAR | | dependent parcel, variable operand |
| PVRI | @VAR | | dependent parcel, indirect variable |
| POPP | OPER | OPER | dependent parcel, two operators |
| PAOD | VAL | | dep. parcel, literal offset for string operand |
| PAOV | VAR | | dep. parcel, variable offset for string operand |

Note that the first four are leading instruction parcels while the rest belongs to the dependent category. A polish expression with be encoded as a combination of these parcels with the proviso that the encoding must start with on of the four possible leading parcels. Note that while a certain amount of independence can be ascribed to a parcel, it is not an independent instruction because its import is only valid in the context of the complete instruction of which it is a part.

As an example, the encoding of the representative polish expression (A + B) $\star C \star D \star E$ in the compact PEXPR for is given as PEXPR $AB+C\star DE\star\star$ and in the DES format as:

$$
\begin{aligned}
&PSVR\ A\\
&PVRO\ B\ +\\
&PVRO\ C\ +\\
&PVAR\ D\\
&PVAR\ E\\
&POPP\ +\ +
\end{aligned}
$$

If the size of the variables is 24 bits, and size of the operators is 4 bits, then the code size for the compact PEXPR form is $24 \cdot 5 + 4 \cdot 4 = 144$ bits, and the corresponding size for DES form requires 192 bits, that is about 32% more than the compact form. This overhead in code size is quite acceptable when compared to the more that twofold increase in RISC code size over that of CISC code size. More importantly, the performance of the pipeline has not been compromised because each instruction parcel exhibits the same structural simplicity that distinguishes RISC type instructions.

## 5. Pipelined MSA Architecture

The architecture details of the controller and memory unit of the proposed pipelined MSA are presented in the Fig. 1 and Fig. 2, respectively.

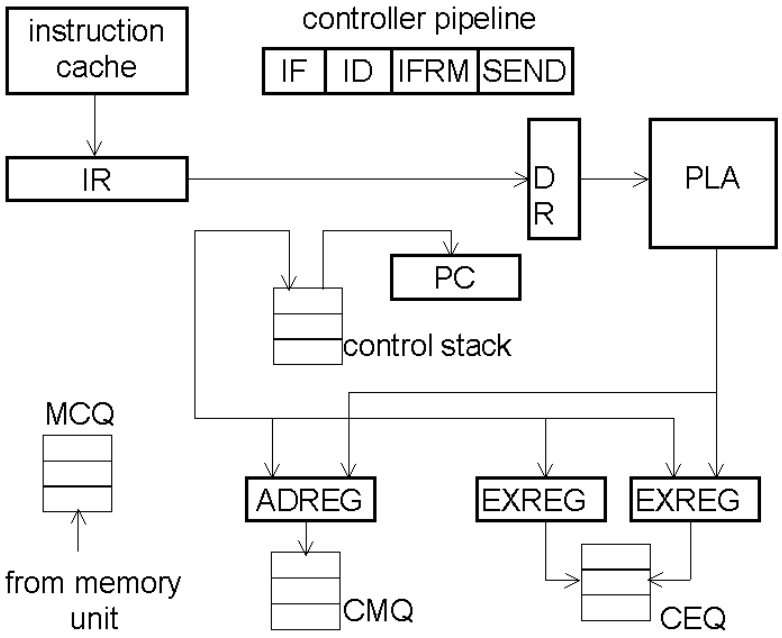


*Fig. 1. The pipelined controller unit of the MSA*

It is important to note that all of the three processors, the controller, the memory, and the execution unit, are pipelined. The instruction set architecture of each unit was designed in accordance with the DES concept.

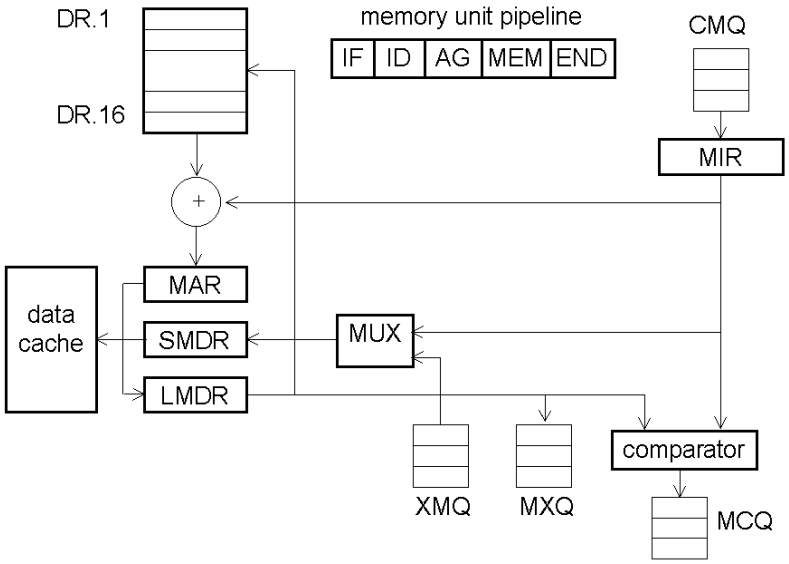


*Fig. 2. The pipelined memory unit of the MSA*

### 5.1 Controller unit

The controller unit, Fig. 1, communicates with the memory and execution processor via the CMQ and CEQ queues, respectively, and receives the results of relational operations from the memory unit through the MCQ. At each cycle a 32-bit instruction parcel is brought in from the instruction cache to the instruction register (IR). If this is a leading parcels, the first byte of the instruction, i.e. the opcode, is copied over to the decode register. In the case of a dependent parcel, only the first four bits are copied on to the last four bit of the DR. The contents of the DR is decoded by the PLA. When decoded, a controller instruction produces instructions for the memory and the execution units. The maximum number of memory and execution instructions that a controller instruction can be decomposed into is three, which includes on address unit instruction and two execution unit instructions. These are written into the address instruction register (ADREG) and the two execution instruction registers (EXREG1 and EXREG2), respectively.

The instructions are formed in the following fashion. The output of the PLA consists of up to a maximum of three fields, each of which is copied on to one of the three registers. The operand field of these registers are copied

from the operand field of the IR. When a controller instruction is in the write stage the contents of the non-empty ADREG, EXREG1 and EXREG2 are copied to the CMQ and CEQ.

### 5.2 Memory unit

The architecture of the memory unit is presented in the Fig. 2, and some of its instructions are described in the Table 4.

Table 4

| memory | instruction | | action implied | pipe delay |
|---|---|---|---|---|
| ACMVR0 | DR.M | OFFSET | compare variable with 0 | 0 cycle |
| ACMVIO | DR.M | OFFSET | compare indirect variable with 0 | 1 cycle |
| ACMVR1 | DR.M | OFFSET | compare variable with 1 | 0 cycle |
| ACMVI1 | DR.M | OFFSET | compare indirect variable with 1 | 1 cycle |
| ARLS | DR.M | OFFSET | relation operation of type *var rel var* | 0 cycle |
| RLGT | VALORVAR | | > comparison and second operand | 0 cycle |
| RLGE | VALORVAR | | >= comparison and second operand | 0 cycle |
| SNVR | DR.M | OFFSET | send variable to MEQ | 0 cycle |
| SNVI | DR.M | OFFSET | send indirect variable to MEQ | 1 cycle |
| STQV | DR.M | OFFSET | store variable from EMQ in address | 0 cycle |
| STQI | DR.M | OFFSET | store variable from EMQ in address ind. | 1 cycle |
| STIV | DR.M | OFFSET | store immediate data in address | 0 cycle |
| STII | DR.M | OFFSET | store immediate data in address indirect | 0 cycle |
| STID | VALUE | | literal value from the previous instruction | 0 cycle |
| SNAR | DR.M | OFFSET | send structured var to MEQ, 1st parcel | 0 cycle |
| SNOD | OFFSET | | 2nd parcel, offset is literal | 0 cycle |
| SNOV | DR.M | OFFSET | 2nd parcel, offset is variable | 0 cycle |
| STAR | DR.M | OFFSET | store structured var from EMQ, 1st parcel | 0 cycle |
| STOD | OFFSET | | 2nd parcel offset is literal | 0 cycle |
| STOV | DR.M | OFFSET | 2nd parcel offset is variable | 0 cycle |

The memory unit receives its instruction from the controller unit trough the CMQ. At each clock cycle an instruction is fetched form the CMQ and copied to the address instruction register (AIR). If the queue is empty, an NOP instruction is generated. This address is copied onto the memory address register (MAR). If the instruction requires a memory read, then the data is read from the data cache (DC) to the load memory data register (LMDR), and is subsequently written into MXQ. If memory write is indicated, the contents of store memory data register (SMDR) is written to the DC. The value in the SMDR has been copied from the AIR during an earlier cycle (ID). In both cases, the address of the load/store operation is fetched

from the MAR. The second category of instructions that the memory unit handles is the comparison instructions. In this case the instructions are fetched from the CMQ, the relation evaluated by the memory unit comparator, and the result is written back to the controller via the MCQ. The third category of instructions that is handled by this unit are the subroutine call and return. The processing of these instructions involves the creation or deletion of data stack frames and will be explained later.

### 5.3 Execution unit

The execution unit fetches its instruction from the CEQ. As in the case of the memory unit, an empty CEQ generates a NOP. The instruction set of the execution unit is shown in the Table 5.

Table 5

| instruction | | action implied |
|---|---|---|
| IVAL | VAL | push value operand on the expression stack |
| QVAL | | push a pointer to QITEM on the expression stack |
| OPER | <OPERATOR> | execute the operation specified by the OPERATOR field on the first two items on the expression stack |
| SEND | | send the value in RESREG to the EMQ |

### 5.4 Address encoding

MSA supports two types of memory objects: (i) simple scalar objects and (ii) structured objects. Al simple variable operands are encoded by a pair of values which are combined together to form the address of the variable, i.e. to bind the variable to a memory location. The 24 bit address consists of a four bit *display* field and a 20 bit *offset* field. The display field determines the environment in which the variable is defined. The display field is decoded to point to a display register in the memory unit which contains the base address of this environment. The actual address is calculated by adding the 20 bit offset to the base address specified by the relevant display register. Since the offset field is 20 bit long this implies that at most $2^{20}$ local variables are allowed in each environment. A 4 bit value for the display register implies that a maximum *static* nesting of 16 levels are supported by the MSA.

Structured variables are characterized by a base address, display and dimension. Each component of a structured object is further specified by an offset. The address of a component of a structured object is formed by adding three components: (i) base address of the data frame which contains the object and which is determined by the display register, (ii) base address

of the object, and (iii) offset of the component. In accordance to the DES principle, the base address and the display level of the structured object is specified by a leading parcel, and the offset is specified by a dependent parcel.

### 5.5 Procedure abstraction

The instruction set of MSA has been augmented to support the procedure abstraction. A distributed version of the Johnston's Contour model has been adopted for this purpose, [9], [10]. The management of the run-time environment necessary for this feature is distributed between the controller and memory units. The conventional activation stack is split into two stacks in the MSA. The control stack, which stores the return addresses, is maintained by the controller unit, and the *data stack*, which provides the space for the local variables is managed by the memory unit. Access to non-local variables is provided through the display mechanism. The displays are maintained in 16 display registers (DR.1 - DR.16) in the memory unit. Apart form the display registers, the memory unit also contains a HIGHMEM register, which points to the top of the data stack, and a CURDISP register which points to the display register associated with the current environment.

The layout of the activation record of a procedure in the data stack is organized in the following way. The first entry contains the old value of the display register $m$ which was saved at this location when the frame was created. The rest of the frame consists of parameters passed to the procedure and local variables, which includes both scalar and array type variables. Heap type storage is not required because dynamic variables are not supported in the MSA.

### 5.6 Control transfer instructions

Both unconditional and conditional control transfer instructions are present in the instruction repertoire of the MSA. GOTO implements control transfer. The conditional control transfer statement is formed of two sets of instructions, the LOOP instruction and the relational instructions transfer statement. First the condition on which the transfer is predicated must be evaluated. Once the value is known, actions pertaining the execution of control transfer can be undertaken, if required. The relational instructions are responsible for the evaluation of the branch condition. This set of instructions is DES encoded to alleviate structural complexity. The two leading parcels are RLS1 (for relation of the type VAR REL VAL) and RLS2 (for

the relations of the type VAR REL VAR). The rest of the instructions specify the kind of relational operation and the second operand of the expression. the interpretation of the second operation (variable or literal value) is contingent upon the preceding parcel. Also included in the category of relational instruction are four single word instructions which compare the immediate operand for equality with 0 or 1.

## 6. Performance Characteristics

The goal of this part of the project was to study the effects of the variation of different architectural parameters on the pipelined MSA performance. We first present the cycle level simulator that is used to execute the representative benchmark programs on the pipelined MSA, and then the results obtained by means of the simulator.

### 6.1 Simulator

To provide a testbed for exploring the performance of the proposed architecture a cycle level simulator of the described pipelined MSA has been implemented, [5]. The overall structure of the simulator is given in Fig. 3. The simulator reads a hardware specification file which defines various architectural attributes of the machine to be simulated. Parameters that can be specified include the various queue lengths and relative speeds of the different units. The program consists of four main modules, three of which simulate the controller, memory and execution units, and the fourth, the scheduler, coordinates the activities of the different units and also collects the data at every clock cycle. Several utility programs are also provided. A compiler translates the high level language programs into controller machine language programs. Several result analyzers process the large data files generated by the simulator and compute the performance parameters.

The simulator executes programs written in the machine code and then generates various sets of data characterizing the execution process. These includes total parallel and serial execution times, execution time of each component, types and number of pipeline stalls, address traces, etc. The estimate of serial time is based on the execution time of the same program on an equivalent serial machine.

### 6.2 Performance indicators

The measurements of performance are presented in comparison with some standard configurations. An *equivalent serial machine* of a pipelined

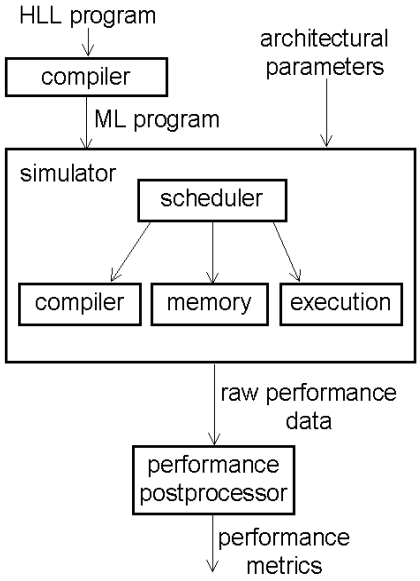


*Fig. 3. The srtucture of simulator*

MSA is a serial machine whose instruction set is identical to the instruction set of the controller unit of the MSA, and for which the execution time of each instruction is defined by the sum of the execution times of the semantic actions activated by the controller instruction in the MSA. Therefore, the equivalent serial machine does not incur the penalty of synchronization of an MSA. A *base MSA* is a machine, all of whose component IPs have a speed of unity and all the queues of which are of length one. A metric that measures the effectiveness of the MSA configuration is the ratio of execution time of a serial equivalent machine to that of a base MSA.

A *component speedup* is the speedup factor of a given component. This is an architectural attribute specified as a design parameter. The component speedup characterizes the computational capability of the component which can be augmented by different mechanisms, e.g. by investing more silicon in the unit (pipelined unit, faster circuits etc.) or by using a different technology (ECL, GaAs). The parameter component speedup thus denotes a quantitative characterization of the increase in the computational capabilities.

A *System speedup* for an MSA of a given configuration is the ratio of the execution time of a program on a base machine to the execution time of

the same program on an MSA of the given configuration. The speedup of the base machine over the equivalent serial machine for different benchmark programs is given in [Man].

### 6.3 Benchmark programs

The performance of the MSA relative to the equivalent serial machine has been extensively studied by means of the simulator. The workload of the simulation whose results are presented in this section consists of seven programs. The first five Livermore loops represent a load characterized by intensive floating point processing, and results in a high degree utilization of the execution unit, and also provide a moderate level of loading of the memory units. Loops 1 and 2 involve evaluation of long expressions which augments the level of parallel execution of the memory and execution units through the mechanism of use order renaming.

The next benchmark program involves the evaluation of the Ackerman's function. It is a memory intensive program, characterized by deep recursive calls and in which the execution unit is almost never used and results in a completely unbalanced loading of the components. The last program is a list insertion routine which inserts several data items in a doubly linked list. This program is characterized by shallow calling depths and heavy utilization of the memory unit at the expense of the execution unit, although not to the extent of the previous program.

### 6.4 Variation of the queue length

For this set of experiments the length of the different queues were varied over a range of 1 to 10 with increments of 1. The effects of the instruction queues (CMQ and CEQ) were studied with unbounded MXQ and XMQ, and vice versa, the effects of the MXQ and XMQ were studied with unbounded CMQ and CEQ. This was done in order to decouple the effects of the various queues on performance. The results obtained, with minor variations, were almost identical for all the programs in the benchmark suite.

Varying the length of the instruction queues from the minimum (1) to the maximum (10) value results in an improvement of the performance by a factor of 1.05 over the base machine. Moreover, this value was reached by a queue size of 3, after which increasing the queue size had no effect on the performance. This observation was mirrored in the case of study involving the other set of queues. The maximum speedup obtained was 1.2, and the saturation point was reached by a queue length of 3.

The reason for this minimal increase is easily explained. The controller unit for each instruction it executes produces data items (instructions) that must be processed by the execution and memory units. Therefore, a close lock-step type synchronization is established between the controller and the other two units which minimizes the effect of the capacity of the communication channels.

### 6.5 Variation of the component speedup

This set of experiments shows the effect of varying the speed of the memory and execution units over a range of 1 to 16 (with the controller speed as the base speed) on the program execution time.

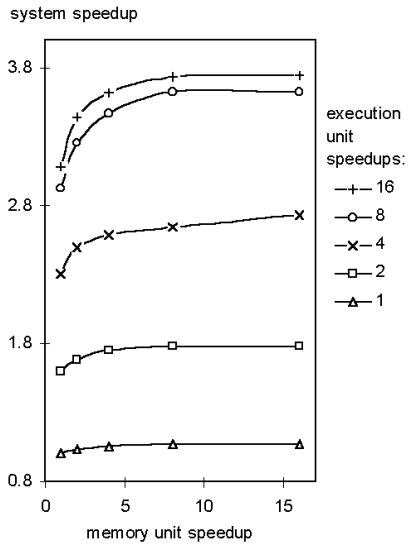


*Fig. 4. System speedup vs. memory unit speedup (Livermore #1)*

Different combinations of memory and execution unit speed are considered. An example of a typical diagram obtained running this set of experiment is given in Fig. 4 that shows the system speedup versus the memory unit speedup. The Livermore loop programs are characterized by intensive floating point computations and as such the greatest benefit is reaped by enhancing the processing power of the execution unit. This observation is completely corroborated by the system speedup curves. Compared to the gain obtained by increasing the speed of the execution unit, the net gain in

processing time obtained by increasing the processing power of the memory unit is relatively small.

The Ackerman benchmark and the List insertion benchmark provide a different workload and therefore radically different performance profiles. The base speedup in the case of the Ackerman function is 2.1 and for list is 1.9, considerably larger that the first Livermore loops (1.73). In view of the fact that both this benchmark create a highly unbalanced load this result might seem surprising. The reason for this apparent discrepancy can be explained by the fact that the exclusion of the execution unit from the processing also excludes all the various stalls associated with this unit and results in a configuration where the controller and memory units operate very much like a two stage pipeline which results in the doubling of the combined throughput of the system. The same observation applies to the case of the List insertion program, albeit to a lesser extent.

## 7. Conclusion

The parallelism exploited by the machine for parallel interpretation is very different in flavor from that of temporal or spatial parallelism, and can be termed *structural parallelism*. It exploits the parallelism inherent in the structure of the interpretation process. The performance of an architecture for parallel interpretation of high level machine languages (MSA) can also be enhanced by incorporating other forms of parallelism within its architecture: superscalar MSA or pipelined MSA.

The paper presents the design details of a pipelined MSA with three pipelined units: controller, memory and execution units. Note that in all of the three pipelines the range and domain of the instructions do **not** overlap. The consequences of the logical and physical separation of the range and domain structures are profound, because it implies that there are **no** data hazards in these pipelines, and thus they do **not** suffer from the crippling effect of this type of hazards.

The pipelined units were designed by using the Distributed Encoding Scheme (DES) that was derived from the observation that the most complex instructions consists of a number of operations which are executed in a sequential fashion and which usually over-utilize the resources of the pipeline. The DES retains structural simplicity of the instruction set at the cost of an acceptable increase in code size.

Performance measures, obtained by a cycle level simulator, show that the system speedup of about two is obtained relative to the equivalent se-

rial interpretation for five different benchmark programs. Analysis of the performance curves also reveals information about the optimal speed configurations. For example, for the Livermore loops, it can be seen that 90% of the maximal performance can be obtained by controller/memory/execution unit speed ratio of 1/14/8. Further increase in the component speed does not result in a proportionate increase in system performance. Of course, the optimal speed configuration is dependent on the load characteristics. In the case of the Ackerman function, the optimal speed ratio would be 1/4/1/ which results in 95% of the maximal attainable speed.

## REFERENCES


1. M.L. Manwaring and V.D. Malbaša: *A processor architecture for parallel interpretation of abstract machine languages.*, to appear in Facta Universitatis, Series Matematics and Informatics, 1998.
2. M. Flynn: *Computer Architecture: Pipelined and Processors, Parallel Processor Design.*, Jones and Bartlett, Boston, 1995.
3. J. Smith: *Decoupled access/execute computer architecture.*, ACM Trans. Computer Systems, Vol. 2, No. 4, April 1984, pp. 289-308.
4. D. Eddy and J. Campenhout: *Interpretation and Instruction Path Coprocessing.*, MIT Press, 1990.
5. M.F. Chowdhury: *Parallel interpretation of abstract machine language.* Ph.D. Thesis, Washington State University, 1993.
6. J. Henessy and D. Patterson: *Computer Architecture: A Quantitative Approach.*, Morgan Kaufman Publishers, 1996.
7. M. Kuga, M. Kazuaki and S. Tomita: *DSNS: Yet another superscalar processor architecture.*, Computer Architecture News, Vol. 19, No. 4, 1991, pp. 14-29.
8. M. Kazuaki, N. Irie, M. Kuga and S. Tomita: *SIMP: A novel high- speed single-processor architecture.*, Proc. 16th Annual Int'l Symposium on Computer Architecture, June 1989, pp. 78-83.
9. A. Aho, R. Sethi and J. Ullman: *Compiler Principles, Techniques, and Tools.*, Addison-Wesley, 1986.
10. J. Johnston: *The contour model of block structured processes.*, SIGPLAN Notices, Vol. 6, 1971, pp. 121-145.