HPS, A NEW MICROARCHITECTURE: RATIONALE AND INTRODUCTION

Yale N. Patt, Wen-mei Hwu, and Michael Shebanow

Computer Science Division
University of California, Berkeley
Berkeley, CA 94720

ABSTRACT

HPS (High Performance Substrate) is a new microarchitecture targeted for implementing very high performance computing engines. Our model of execution is a restriction on fine granularity data flow. This paper introduces the model, provides the rationale for its selection, and describes the data path and flow of instructions through the microengine.

1. Introduction

A computer system is a multilevel structure, algorithms at the top, gates and wires at the bottom. To achieve high performance, one must optimize at all levels of this structure. At most levels, the conventional wisdom suggests exploiting concurrency. Several proposals have been put forward as to how to do this. We also argue for exploiting concurrency, focusing in particular on the microarchitecture level.

1.1. Restricted Data Flow.

We are calling our engine HPS, which stands for High Performance Substrate, to reflect the notion that what we are proposing should be useful for implementing very dissimilar ISP architectures. Our model of the microengine (i.e., a restriction on classical fine granularity data flow) is not unlike that of Dennis [3], Arvind [2], and others, but with some very important differences. These differences will be discussed n detail in section 3.

For the moment, it is important to understand that unlike classical data flow machines, only a small subset of the entire program is in the HPS microengine at any one time. We define the "active window" as the set of ISP instructions whose corresponding data flow nodes are currently part of the data flow graph which is resident in the microengine. As the active window moves through the dynamic instruction stream, HPS executes the entire program.

1.2. Potential Limitations of Other Approaches.

We believe that an essential ingredient of high performance computing is the effective utilization of a lot of concurrency. Thus we see a potential limitation in microengines that are limited to one operation per cycle. Similarly, we see a potential limitation in a microengine that underutilizes its bandwidth to either instruction memory or data memory. Finally, although we appreciate the advantages of static scheduling, we see a potential limitation in a microengine that purports to execute a substantial number of operations each cycle, but must rely on a non-run-time scheduler for determining what to do next.

1.3. Outline of this paper.

This paper is organized in four sections. Section 2 delineates the fundamental reasons which led us to this new microarchitecture. Section 3 describes the basic operation of HPS. Section 4 offers some concluding remarks, and describes where our research in HPS is heading.

2. Rationale.

2.1. The Three Tier Model.

We believe that irregular parallelism in a program exists both locally and globally. Our mechanism exploits the local parallelism, but disregards global parallelism. Our belief is that the execution of an algorithm should be handled in three tiers. At the top, where global parallelism can be best identified, the execution model should utilize large granularity data flow, much like the proposal of the CEDAR project [4]. In the middle, where forty years of collected experience in computer processing can be exploited probably without harm, classical sequential control flow should be the model. At the bottom, where we want to exploit local parallelism, fine granularity data flow is recommended. Our three tier model reflects our conception that the top level should be algorithm oriented, the middle level sequential control flow ISP architecture oriented, and the bottom level microengine oriented.

2.2. Local Parallelism.

We feel obliged to re-emphasize the importance of local parallelism to our choice of execution model. Indeed, we chose this restricted form of data flow specifically because our studies have shown that the parallelism available from the middle control flow tier (i.e., the sequential control flow architecture) is highly localized. We argue that, by restricting the active instruction window, we can exploit almost all of the inherent parallelism in the program while incurring very little of the synchronization costs which would be needed to keep the entire program around as a total data flow graph.

2.3. Stalls, Bandwidth, and Concurrency.

We believe that a high performance computing engine should exhibit a number of characteristics. First, all its components must be kept busy. There must be few stalls, both in the flow of information (i.e., the path to memory, loading of registers, etc.) and in the processing of information (i.e., the functional units). Second, there must be a high degree of concurrency available, such as multiple paths to memory, multiple processing elements, and some form of pipelining, for example.

In our view, the restricted data flow model, with its out-of-order execution capability, best enables the above two requirements, as follows: The center of our model is the set of node tables, where operations await their operands. Instruction memory feeds the microengine at a constant rate with few stalls. Data memory and I/O supply and extract data at constant rates with few stalls. Functional units are kept busy by nodes that can fire. Somewhere in this system, there has to be "slack." The slack is in the nodes waiting in the node tables. Since nodes can execute out-of-order, there is no blocking due to unavailable data. Decoded instructions add nodes to the node tables and executed nodes remove them. The node tables tend to grow in the presence of data dependencies, and shrink as these dependencies become fewer. Meanwhile, our preliminary measurements support, the multiple components of the microengine are kept busy.

3. The HPS Model of Execution.

3.1. Overview.

An abstract view of HPS is shown in figure 1. Instructions are fetched and decoded from a dynamic instruction stream, shown at the top of the figure. The figure implies that the instruction stream is taken from a sequential control flow ISP architecture. We need to emphasize that this is not a necessary part of the HPS specification. Indeed, we are investigating having HPS directly process multinode words (i.e., the nodes of a directed graph) which would be produced as the target code of a (for example) C compiler. What is necessary is that, for each instruction, the output of the decoder which is presented to the Merger for handling by HPS is a data flow graph.

A very important part of the specification of HPS is the notion of the active instruction window. Unlike classical data flow machines, it is not the case that the

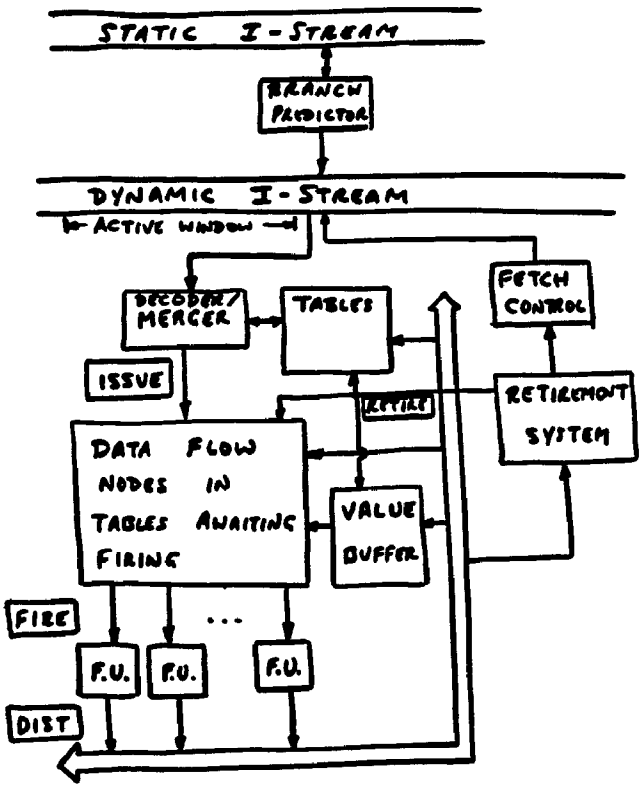

FIGURE 1.

data flow graph for the entire program is in the machine at one time. We define the active window as the set of ISP instructions whose corresponding data flow nodes are currently being worked on in the data flow microengine.

As the instruction window moves through the dynamic instruction stream, HPS executes the entire instruction stream. Parallelism which exists within the window is fully exploited by the microengine. This parallelism is limited in scope; ergo, the term "restricted data flow."

The Merger takes the data flow graph corresponding to each ISP instruction and, using a generalized Tomasulo algorithm to resolve any existing data dependencies, merges it into the entire data flow graph for the active window. Each node of the data flow graph is shipped to one of the node tables where it remains until it is ready to fire.

When all operands for a data flow node are ready, the data flow node fires by transmitting the node to the appropriate functional unit. The functional unit (an ALU, memory, or I/O device) executes the node and distributes the result, if any, to those locations where it is needed for subsequent processing: the node tables, the Merger (for resolving subsequent dependencies) and the Fetch Control Unit (for bringing new instructions into the active window). When all the data flow nodes for a particular instruction have been executed, the instruction is said to have executed. An instruction is retired from the active window when it has executed

and all the instructions before it have retired. All side effects to memory are taken care of when an instruction retires from the active window. This is essential for the correct handling of precise interrupts [1].

The instruction fetching and decoding units maintain the degree of parallelism in the node tables by bringing new instructions into the active window, which results in new data flow nodes being merged into the data flow node tables.

3.2. Instruction Flow

Figure 2 shows the global data path of HPS. Instructions enter the data path as input to the Merger. This input is in the form of a data flow graph, one per instruction. The data flow graph can be the result of decoding an instruction in a classical sequential instruction stream, or it can be the output of a non-conventional compiler. In either case, the Merger sees a set of data flow nodes (and data dependencies), one for each operation that must be performed in the execution of that instruction. Operations are, for example, reads, writes, address computations and ALU functions. In the example of figure 3, the data flow graph corresponding to the VAX instruction ADDL3 #1000,A,B consists of three nodes: a memory read, memory write, and an ALU operation. Figure 3 also shows the structure of the three nodes and the five value buffer entries required for the instruction.

The Merger, using the Register Alias Table to resolve data dependencies not explicit in the individual instruction, forms the set of data flow nodes which are necessary to execute the instruction. Nodes are then transmitted to the appropriate node tables. Node tables, as we shall see, are content addressible memories, and thus should be kept small. The size of each node table is a function of the size of the active window and the decoding rate of the Von Neumann instruction stream. In our experiments with the VAX architecture, for example, an active window of 16 instructions, coupled with a decoding rate of eight nodes per cycle, required at most a 35 entry node table.

For each node, a slot is reserved in the global multi-port value buffer for storing the result of the operation of that node. The index of each slot is designated as a tag for the corresponding node, and is carried along with the node until it completes its execution. Value buffer slots are assigned in a circular queue, the size of the buffer being large enough to guarrantee retirement of an instruction before its value buffer slot is again needed. (In the case of our simulated implementation of the VAX architecture, an active window of 16 instructions, having approximately four nodes per instruction, means that a value buffer of 128 entries is more than adequate.)

A node remains in its node table until all of its operands are available, at which point it is ready to fire (i.e., it is executable). A node is fired by transmitting its operator, tag, and set of operands to one of the functional units associated with that node table. When execution completes, the result and its tag are distributed to each port of the value buffer. In the case of a result destined for a general purpose register, the corresponding tag is also transmitted to the Register Alias Table to update information stored there. The corresponding tag is also transmitted to the node tables for the purpose of setting the ready bits in those nodes awaiting this result.

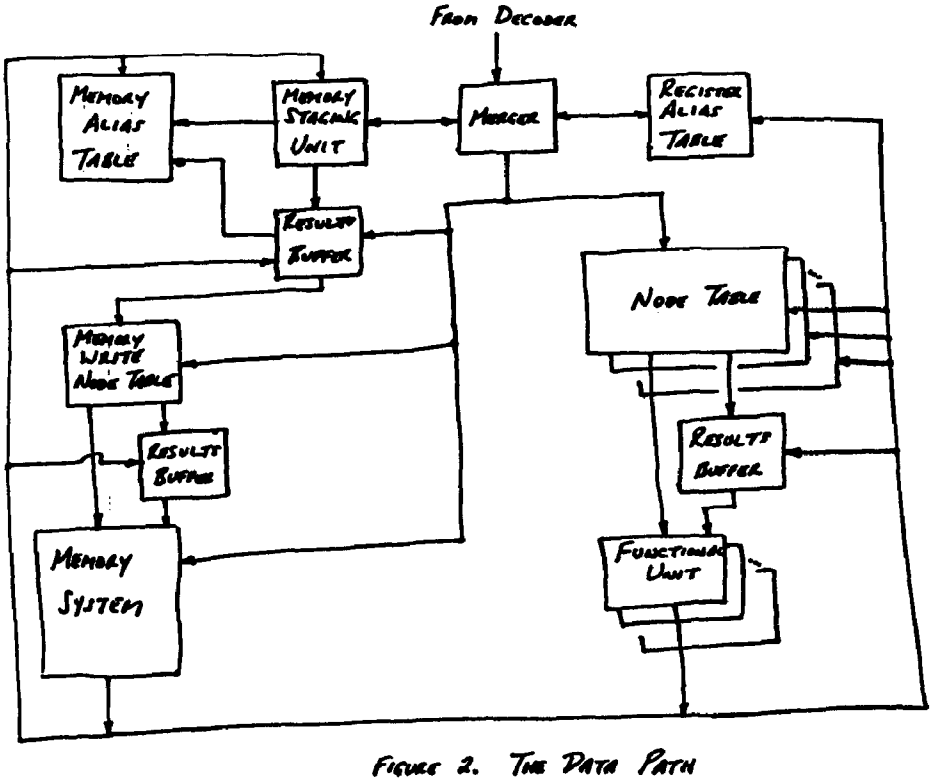

FIGURE 2. THE DATA PATH

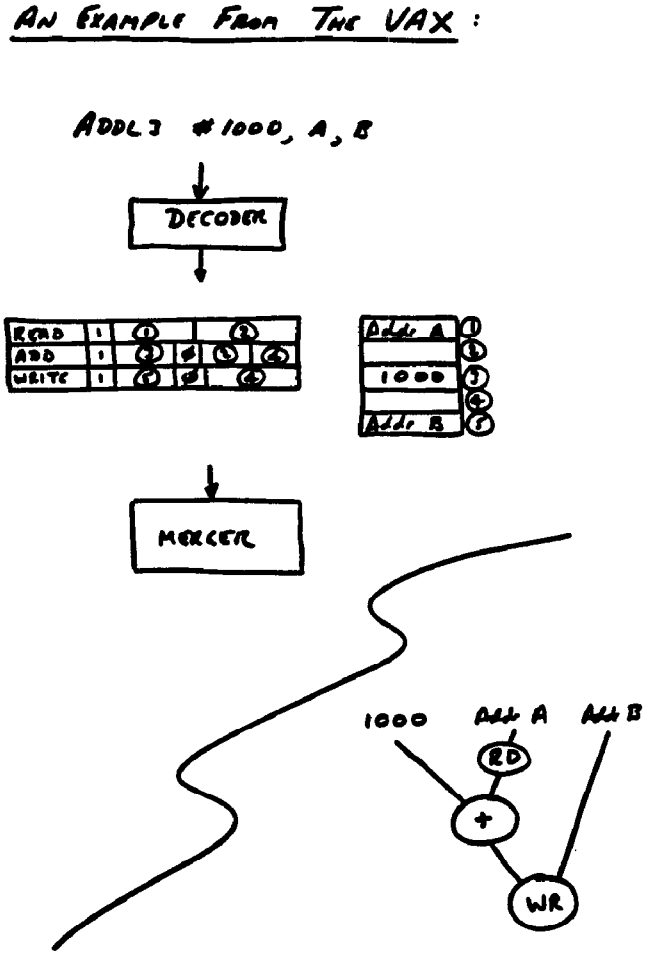

FIGURE 3.

Memory read and write nodes present additional complications. Although these will be discussed in greater detail in [7], a few observations here are in order. First is the fact that at the time memory access nodes are issued by the Merger (depending of course on the addressing structure of the target architecture), the address of the memory access may be unknown, and the addresses of other memory accesses which could block the node being issued may also be unknown. A Memory Alias Table and a Read Staging Unit are provided to handle these problems. Second is the fact that writes can occur out of order coupled with our requirement that exception handling must allow the machine state to be recovered "precisely." A Write Buffer and an algorithm for retiring instructions are provided for handling this problem.

One final observation about the processing of nodes must be made. The stages that a node goes through (i.e., merging, waiting for operands, firing, executing, and distributing its results) is independent of the other nodes in the node tables. That is, for example, the number of nodes firable in a given cycle is limited by the ability to detect that multiple nodes are firable and the number of functional units available for concurrent processing of nodes. The number of results that can be distributed in a single cycle is a function of the bus structure and the organization of the node tables. The intent is that in each cycle, multiple nodes will be in each stage of the process.

3.3. Data Dependencies and their Resolution.

Fundamental to the correct, fast, out-of-order execution of operations in HPS is the handling of data dependencies and, as we will see, the absence of blocking in those cases where blocking is unnecessary. Since our locally concurrent implementation model has to conform to the target architecture, the local concurrency exploited must not cause incorrect execution results.

3.3.1. Data, Anti, and Output Dependencies.

A micro-operation *B* depends on another micro-operation *A* if *B* has to be executed after *A* in order to produce the correct result. There are three ways in which a micro-operation can depend on another micro-operation through register usage: data, anti, and output dependencies.

A data (read-after-write) dependency occurs when *A* is going to write to the register from which *B* is going to read. In this case, *A* supplies information essential to the execution of *B*. An anti (write-after-read) dependency occurs when *A* is going to read from the register to which *B* is going to write. An output (write-after-write) dependency occurs when *A* and *B* are going to write to the same register.

In the last two cases, the execution of *A* does not supply any information necessary for the execution of *B*. The only reason *B* depends on *A* is that a register has been allocated to two different temporary variables due to a shortage of registers. In fact, if we had an unlimited number of registers, different temporary variables would never have to be allocated to the same register and the second and the third dependencies would never occur. So, a proper renaming mechanism and extra buffer registers would remove anti and data dependencies. Then, the only type of dependency that could delay micro-operation execution would be a data dependency. In other words, a micro-operation could be executed as soon as its input operands are properly generated. This is exactly the description of a data flow execution model.

3.3.2. Our Modified Tomasulo Algorithm.

Our algorithm for enforcing data dependencies and removing anti and output dependencies is similar to the Tomasulo algorithm which was used in the Floating Point Unit of the IBM 360/91 [6]. During execution, the algorithm manages two major data structures: a register alias table and a set of node tables. Each entry in the register alias table keeps track of the dynamic information for a register necessary either to supply an input operand value or to establish dependency arcs. There are two fields in each register alias table entry. The first is a *ready* bit. This bit, if cleared, indicates that there is an active micro-operation which is going to supply the register value. The second field is the *tag* field which provides an index into a result buffer. This indicates where the register value can be found if the ready bit is set.

Each entry in a node table corresponds to a micro-operation and has an *operation* field, a *result tag* field, and two operand records. The *operation* field specifies the action that will be performed on the input operands. The *result tag* field provides the location in the result buffer that the result value will be shipped to after the execution of the micro-operation. Each operand record consists of two fields. The first is a *ready* bit. This bit is set when the input operand has been properly produced. The second field is the *tag* field which contains an index into the operand buffer. This indicates where the operand can be found if the ready bit is set.

A data path designed for our modified Tomasulo algorithm is presented in figure 2. There are two phases in each machine cycle: merging/scheduling and distribution. Initially all register alias table entries are ready and the initial register values are in result buffer entries whose index is in the *tag* fields of the corresponding register alias table entries.

Merging/Scheduling

A new micro-operation is assigned an entry in a node table and is given a unique *result tag* . First, the contents of both fields in the register alias table for each input operand are copied to the corresponding fields in the operand records of the new node table entry. Second, the *ready* bit of the register to be written by the micro-operation is reset and the *result tag* for the micro-operation is written into the *tag* field of the alias table entry. If both operands of a node are marked ready, this node can fire. The tags in the operand records are used to index into the result buffer and obtain the operand values. The *operation* and the operand values are sent to the function unit for execution. The *result tag,* which will be used to distribute the result, is also sent to the function unit.

Distribution

When a function unit finishes executing a micro-operation, the *result tag* of that micro-operation is used to select a result buffer entry and the result value is stored into the entry. The result tag is also distributed to the register alias table and the node table. Both the register alias table and the node table are content addressable memories. Entries in these tables are addressed by the value of the result tag. All of the register alias table entries and all of the operand records in the node table entries set their ready bit if the distributed *result tag* matches their *tag* field contents.

After execution, all the register alias table entries are ready and the corresponding register values are in the result buffer entries whose indices are in the *tag* fields.

The algorithm described above is a win on at least two counts. First, it removes anti and output dependencies without producing incorrect results. In fact, it can be shown that reservation schemes without renaming can not remove anti and output dependencies without producing incorrect results. Second, unlike Scoreboarding (for example), the issuing process never has to stall due to dependencies. It can also be shown that any reservation scheme without renaming will have to stall for some dependencies.

4. Concluding Remarks.

The purpose of this paper has been to introduce the HPS microarchitecture. Current research at Berkeley is taking HPS along four tracks. First, we are attempting to design, at high performance, three very dissimilar architectures: the microVAX, a C machine, and a Prolog processor. Equally important, we are investigating the limits of this microarchitecture, both from the standpoint of a minimal implementation and from the standpoint of a cadillac version.

As is to be expected, there are issues to be resolved before an effective HPS implementation can be achieved. For example, if HPS is to implement a sequential control based ISP architecture, then there are decoding issues, including the question of a node cache, which need to be decided. Second, HPS requires a data path that (1) has high bandwidth and (2) allows the processing of very irregular parallel data. Third, HPS needs a scheduler which can determine, in real-time, which nodes are firable and which are not. Fourth, the out-of-order execution of nodes requires additional attention to the design of the memory system, the instruction retirement and repair mechanisms, and the I/O system. These issues are the subject of a companion paper [7] in these Proceedings.

Acknowledgement.

The authors wish to acknowledge first the Digital Equipment Corporation for supporting very generously our research in a number of positive ways: Linda Wright, formerly Head of Digital's Eastern Research Lab in Hudson Massachusetts for providing an environment during the summer of 1984 where our ideas could flourish; Bill Kania, formerly with Digital's Laboratory Data Products Group, for providing major capital equipment grants that have greatly supported our ability to do research; Digital's External Research Grants Program, also for providing major capital equipment to enhance our ability to do research; and Fernando Colon Osorio, head of Advanced Development with Digital's High Performance Systems/Clusters Group, for providing funding of part of this work and first-rate technical interaction with his group on the tough problems. We also acknowledge the other members of the HPS group, Steve Melvin, Chien Chen, and Jia-juin Wei for their contributions to the HPS model as well as to this paper. Finally, we wish to acknowledge our colleagues in the Aquarius Research Group at Berkeley, Al Despain, presiding, for the stimulating interaction which characterizes our daily activity at Berkeley.

5. References.

1. Anderson, D. W., Sparacio, F. J., Tomasulo, R. M., "The IBM System/360 Model 91: Machine Philosophy and Instruction - Handling," IBM Journal of Research and Development, Vol. 11, No. 1, 1967, pp. 8-24.

2. Arvind and Gostelow, K. P., "A New Interpreter for Dataflow and Its Implications for Computer Architecture," Department of Information and Computer Science, University of California, Irvine, Tech. Report 72, October 1975.

3. Dennis, J. B., and Misunas, D. P., "A Preliminary Architecture for a Basic Data Flow Processor," Proceedings of the Second International Symposium on Computer Architecture, 1975, pp 126-132.

4. Gajski, D., Kuck, D., Lawrie, D., Sameh, A., "CEDAR -- A Large Scale Multiprocessor," Computer Architecture News, March 1983.

5. Keller, R. M., "Look Ahead Processors," Computing Surveys, vol. 7, no. 4, Dec. 1975.

6. Tomasulo, R. M., "An Efficient Algorithm for Exploiting Multiple Arithmetic Units," IBM Journal of Research and Development, vol. 11, 1967, pp 25 - 33. Principles and Examples, McGraw-Hill, 1982.

7. Patt, Y.N., Melvin, S.W., Hwu, W., and Shebanow, M.C., "Critical Issues Regarding HPS, a High Performance Microarchitecture, Proceedings of the 18th International Microprogramming Workshop, Asilomar, CA, December, 1985.