HPSm, a High Performance Restricted Data Flow Architecture Having Minimal Functionality

Wen-mei Hwu and Yale N. Patt

Computer Science Division,

University of California, Berkeley

Berkeley, CA 94720

ABSTRACT

Our recent work in microarchitecture has identified a new model of execution, restricted data flow, in which data flow techniques are used to coordinate out-of-order execution of sequential instruction streams. We believe that the restricted data flow model has great potential for implementing very high performance computing engines. This paper defines a minimal functionality variant of our model, which we are calling HPSm. The instruction set, data path, timing and control of HPSm are all described. A simulator for HPSm has been written, and some of the Berkeley RISC benchmarks have been executed on the simulator. We report the measurements obtained from these benchmarks, along with the measurements obtained for the Berkeley RISC II. The results are encouraging.

1. Introduction

Irregular parallelism in a program exists both locally and globally. Our mechanism exploits local parallelism, but disregards global parallelism. We are searching for a proper tradeoff between the size of hardware involved and the amount of parallelism exploited. Limiting the scope of exploitation is one way to keep the required hardware small in size and fast in speed. The task of exploiting global parallelism is left to the higher levels, i.e., algorithm and software. We refer to [1] for much of the theoretical basis of our research.

1.1. Restricted Data Flow.

Our model of the microengine (i.e., a restriction on classical fine granularity data flow [2,3]) applies data flow techniques to only a local section of the program at any time of execution. The microengine keeps a subset of the program, which we call the "active window", in the form of data dependency graph. Operations defined by the instructions within the active window are executed when their operands are ready. As execution goes on, new instructions join the active window and old instructions retire from the window. As the active window scans through the dynamic instruction stream, HPS executes the entire program.

We believe that a high performance computing engine should exhibit the following characteristics. First, there must be a high degree of concurrency available in hardware design, such as multiple paths to memory, multiple processing elements, and some form of pipelining. Second, concurrency provided in hardware design must be well utilized. There must be few stalls, both in the flow of information (i.e., the paths to memory, the paths to registers, etc.) and in the processing of information (i.e., the function units).

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, with the help of branch prediction, feeds the microengine at a constant rate with few stalls. Data memory and I/O supply extract data at constant rates with few stalls. Function units are kept busy by operations that can fire. Somewhere in this system, there has to be "slack." The slack is in the operations waiting in the node tables. Since operations can execute out-of-order, an operation does not block the others when its input data is not available. Decoding inserts operations to the node tables and execution removes them. The node tables tend to grow in the presence of data dependencies, and shrink as these dependencies become fewer. We use the node table to buffer the effect of data dependencies and thus to smoothly utilize the concurrency provided in hardware design.

1.2. Outline of this report

This paper is organized into six sections. Section 2 describes the generic HPS (High Performance Substrate) [4,5,6] model of execution. Section 3 defines the architecture of HPSm, a minimal functionality variant of the HPS model. Section 4 specifies its implementation. Section 5 reports the results of executing some of the RISC benchmarks on an HPSm simulator, and compares and analyzes these measurements in the context of the Berkeley RISC II. Section 6 offers a few concluding remarks.

0884-7495/86/0000/0297$01.00 © 1986 IEEE

2. The HPS Model of Execution.

2.1. Overview.

Instructions are fetched, with the help of branch prediction, from a sequential control flow program. Each instruction defines several *operations* which may be ALU operations and memory accesses, etc. The output of the decoder is a data dependency graph connecting all operations defined by one instruction.

A very important part of HPS is the notion of the active window. Unlike the classical data flow machines, HPS does not keep the entire program in the the form of data dependency graph. We define the active window as the set of ISP instructions whose operations are currently being worked on in the data-driven microengine.

New instructions join the active window after being fetched and decoded. Old instructions retire from the window after all the corresponding activities are done. Active window scans the dynamic instruction stream by pulling in new instructions at the front and shelving off old ones at the rear. As the active window scans through the dynamic instruction stream, HPS executes the entire program. Parallelism within the active window is fully exploited by the data-driven microengine.

The *merger* takes the data dependency graph from the decoder and merges it into the data dependency graph for the active window. The links between operations in the incoming graph and the operations in the active window are established with a generalized Tomasulo algorithm [7], Each node of the incoming data dependency graph is shipped to one of the node tables where it remains until being ready for execution.

When all operands for an operation are ready, the scheduling logic transmits the operation to the appropriate function unit. The function unit (an ALU, memory, or I/O device) executes the operation and distributes the result, if any, to those locations where it is needed for subsequent processing: the node tables (for enabling subsequent operations), the merger (for resolving subsequent dependencies) and the Instruction Unit (for bringing new instructions into the active window). When all the operations defined by a particular instruction have been executed, the instruction is said to be done. An instruction is retired from the active window when it is done and all the instructions before it have been retired. All side-effects to memory are made permanent at retire time. This is essential for the precise handling of exceptions [8].

2.2. 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.

An operation B depends on another operation A if B has to be executed after A in order to produce the correct result. There are three ways in which an operation can depend on another operation through register usage: flow, anti, and output dependencies [9].

A flow (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 be allocated to the same register. In that case, anti and output dependencies would never occur. So, a proper renaming mechanism and extra buffer registers would remove anti and output dependencies. Then, the only type of register dependency that could delay operation execution would be a data dependency. In other words, a operation could be executed as soon as its input operands are generated. This is exactly the description of a data-driven execution model.

Data dependencies due to main memory usages also fall into the three categories described above. However, the data dependencies through main memory are more difficult to detect (and thus more difficult to resolve) because the memory location to be accessed is not in general known at instruction issue time. The address may be from a register whose up-to-date value has not be generated yet. Because we can not detect all memory dependencies at instruction issue time, some memory accesses may have to be withheld until potential dependencies are cleared. For example, one may want to withhold all memory reads following a memory write until the address of that memory write is known. This scheme, of course, effectively introduces delays to memory reads that do not actually depend on the memory write.

2.3. Exception and Branch Prediction Miss

Handling branch prediction miss and exceptions is also fundamental to the correct, fast out-of-order execution of operations. However, the two types of repairs do impose different requirement of speed. Branch prediction miss, which occur frequently, requires efficient repair mechanism. For example, if the average number of instructions between branches is five, and if the the branch prediction provides a hit ratio of 90%, we have, on the average, only sixty instructions between branch prediction miss occurrences. Only conditional branch instructions can cause such repair. In order to perform such repair, we need only to save the machine state each time a conditional branch instruction enters the machine.

Exceptions occur far less often than branch prediction miss does. The average number of instructions between exceptions is usually on the order of five thousand or more. Thus repair from exceptions impose less speed requirement. Almost all instructions can potentially cause exceptions. In order to repair from exceptions, we need to save the machine state when every instruction enters the machine.

We do provide different repair mechanism for these two different types of repairs. Different speed/space tradeoffs are made in the two repair mechanism. Please see section 4.4 for more information.

3. Architecture of HPSm.

HPSm is a minimal functionality variant of the HPS model targeted for single chip VLSI implementation. In the architecture aspect, we limit the number of operations defined by each instruction to be small. The number of operations defined in each instruction is two in the preliminary design and simulation. We may extend this number to be three or four after examining substantial applications.

We describe in this section the data types, registers, instruction set, and procedure calling convention of HPSm.

3.1. HPSm Data Types and Registers.

HPSm supports signed and unsigned memory data of sizes byte (8 bits), short (16 bits), and long (32 bits). These data types are supported by appropriate memory operations available in the HPSm instruction set. Proper sign or zero extension is applied when a byte or short data is read from memory. All memory data must be aligned to long word boundaries. If two memory data items partially overlap, the execution result is unpredictable. Within the data path, however, all HPSm operations operate only on 32-bit values.

There are sixteen registers in the architecture. Each register holds a 32-bit data value. Register 15 provides the program counter of the current instruction when used as input operand. When register 15 is written, there is no immediate effect on the actual program counter. A subsequent **SWAP** instruction will swap the program counter with the content of register 15. Register 14 is stack pointer and register 13 is frame pointer. Register 12 is reserved for forwarding purposes. Please see the section on addressing modes for details of using register 12.

3.2. HPSm Instruction Set.

Each HPSm instruction is logically a data-dependency graph with two operations. The two operations which comprise each instruction are of types **ALU/Con** (primary **ALU** or Control transfer), and **Mem/S_ALU/Lit** (Memory access, Secondary **ALU**, or Literal value) respectively (see figure 1). Operations within an instruction may or may not depend on each other. Dependencies between operations are expressed through register 12 which will be described below. Dependencies crossing instruction boundaries are expressed in terms of reading from and writing to registers other than register 12.

3.2.1. HPSm Instruction Format.

The instruction size is fixed at 32-bits. If a control operation other than **RETURN** occupies the **ALU/Con** field, the **Mem/S_ALU/Lit** field contains a 13-bit branch displacement in terms of number of instructions. Otherwise, the **Mem/S_ALU/Lit** may contain a memory access operation, a two address format **ALU** operation, or a 13-bit literal value as an input operand to the primary **ALU** operation.

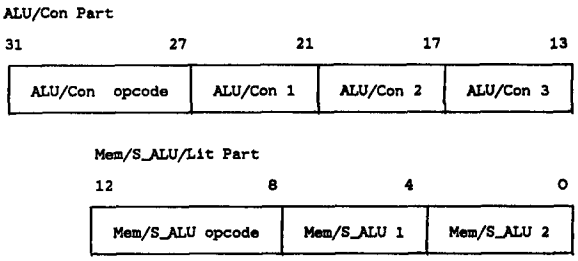

Figure 1. HPSm Instruction Format

3.2.2. HPSm Addressing Modes.

There are three addressing modes for source operands of the HPSm operations: *register, short literal,* and *long literal.* All addressing mode information is specified in the opcode of each operation. That is, the opcode of an operation determines the interpretation of all its operand fields.

An operand in the *register* mode is interpreted as a register number which selects one of 16 registers. When a source operand is register 12, the execution result of the other operation in the same instruction will be directly forwarded as the operand value. In this case, the other operand may or may not write into other register 12. If it does, the result is only forwarded within the same instruction. If it writes to some other register, the result is written to the specified register as well as forwarded within the same instruction. This is used by the compiler to express the dependencies within the an instruction.

A *short literal* operand is sign extended to provide the input value. *Short literal* operands in **ALU** and **Mem/S_ALU** operations range from -32 to +31 and from -8 to +7 respectively. In *long literal* mode, the operand presented in the operand field of the operation is ignored. The sign extended result of the **Mem/S_ALU/Lit** field is used as the input value which ranges from -4096 to +4095.

3.2.3. HPSm Instruction Opcodes.

There are two groups of operations in each HPSm instruction: **ALU/Con** and **Mem/S_ALU**. Each type of operations has its own opcode. The **ALU** operations are three-address arithmetic/logic operations on 32-bit integers. **TESTGT, TESTEQ,** and **TESTLT** operations generate a 1 or 0 depending on whether or not the input operand is greater than, equal to, and less than zero. There is no floating point operation as yet in the HPSm data path. However, an interface to a floating point coprocessor is being designed.

Con operations perform control transfer. **BTRUE** branches when the input register contains a 1. **BFALSE** works in the complementary way. **BRANCH** is unconditional branch. **CALL** and **RETURN** are procedure calling and returning instructions. **SWAP** can be used to implement jumps and calls to locations specified in a memory table. Control transfer has *forward semantics*, i.e., the instruction right after the control transfer instruction in the static code belongs to the path where the transfer is taken.

Memory operations read from and write to memory locations. Secondary **ALU** operations are two-address arithmetic/logic operations on 32-bit integers.

The I/O structure for HPSm is still undergoing change. The I/O operations will be multiplexed with the **ALU/Con** operations. I/O instructions will be performed sequentially to maintain the desired I/O behavior. This is different from memory operations, since memory reads can be executed out of order.

3.3. Procedure Calling Convention.

We divided the register file into three sections: special, safe and unsafe. The special registers are register 15 (potential program counter), register 14 (stack pointer), and register 13 (frame pointer). These are always saved during a procedure call. The safe region consists of registers 8 through 11 and they are specified to remain the same after the return. The called procedure is responsible for saving and restoring the safe registers it modifies, by means of explicit memory operations.

Registers 0 through 7 comprise the unsafe region. The returned value is in register 0. The calling procedure passes parameters via registers 0 through 7. Procedures can also use these registers as local or temporary variables. If the calling procedure needs any information in these registers after the procedure returns, the calling procedure itself is responsible for saving and restoring the registers.

The rationale behind adopting this calling convention is threefold. First, the parameters are passed through registers instead of through memory. This cuts down the memory traffic and increases the parallelism because we can handle register dependencies more efficiently than we can handle memory dependencies. Second, the compiler has the opportunity to reduce the amount of the register saving-restoring of unsafe registers. In tail recursion, for example, where unsafe registers are not used after the procedure returns, the called procedure can use the unsafe registers for free. Third, we do not have to deal with a register window mechanism [10], where large register file can increase the cycle time.

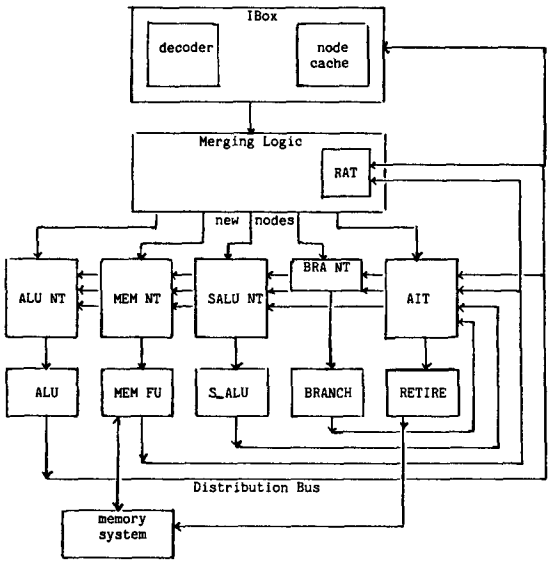

Figure 2. HPSm Data Path

4. Implementation of HPSm.

HPSm is also a minimal functionality variant of the HPS model in implementation. The restrictions on functionality here are (1) number pending branch prediction is limited to one, (2) memory dependency is resolved by a conservative algorithm, (3) only two **ALU**'s and one data memory port are provided. The implementation of HPSm is divided into three major parts: the Instruction unit, the main data path, and the memory system (see figure 2). The instruction unit is responsible for supplying instructions to the data path. The main data path schedules and executes operations in instructions and retire the finished instructions. The memory system does the cache and virtual memory management.

The entire HPSm engine can be viewed as a ten stage pipeline where four stages can be skipped (see figure 3). A trivial HPS instruction spends at least six cycles in the microengine. The pipeline view of the HPSm is somewhat misleading because within the stages, the execution of operations are controlled by data flow firing rules.

4.1. Instruction Unit Organization.

The instruction unit implements an instruction buffer and a fetch algorithm. The instruction buffer caches HPSm instructions in decoded form. The size of

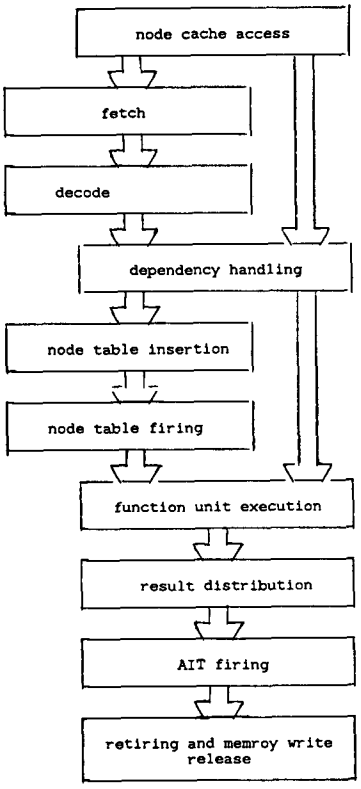

Figure 3. Pipelined View of HPSm

the instruction buffer is 1K entries in current design. When instruction buffer hit, the instruction unit can deliver one instruction each cycle, which is equivalent to delivering two operations each cycle.

On a instruction buffer miss, the instructions are fetched from cache memory and decoded both for execution and for refilling the instruction buffer. Figure 4 shows the decoding logic for ALU operations. The process is pipelined between consecutive instructions. In a instruction buffer miss sequence without cache memory miss, the instruction unit can deliver one instruction every two cycles.

The forward branch mechanism (described in section 3.2.3) is adopted to assist the pipeline design described above. HPSm performs one level of branch prediction. The instruction unit always predicts the conditional branch to be taken. There can be one pending branch prediction in the microengine. When the instruction unit encounters a second conditional branch before the first one is confirmed, the instruction unit stalls. This restriction on the number of pending branch predictions is due to the requirement of efficient repairing from branch prediction miss (we have to save only one set of machine state for this type of repair).

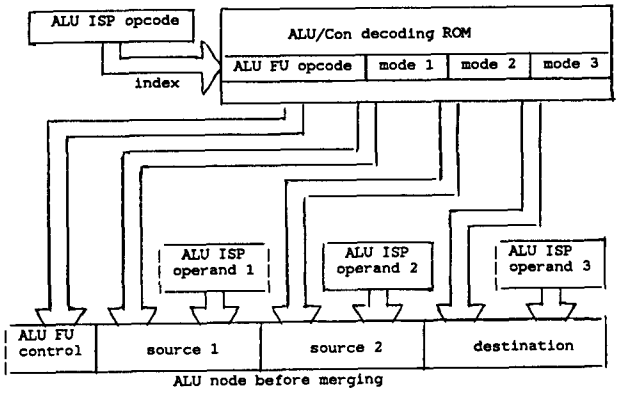

Figure 4. ALU Operation Decoding

4.2. Dependency Graph Handling.

Every HPSm instruction is decoded into a data dependency graph. The entire HPSm microengine maintains a data dependency graph connecting all active operations. The *merger* of HPSm takes the decoder output and uses a modified Tomasulo algorithm [7] to merge it into the active dependency graph. Physically, this is done by translating the source operands in the decoder output into operands of node table entry using the *register alias table* and the tag assignment logic as shown in figure 5(a). The merging process also changes the register alias table contents to reflect the fact that some operations write to registers, as shown in figure 5(b).

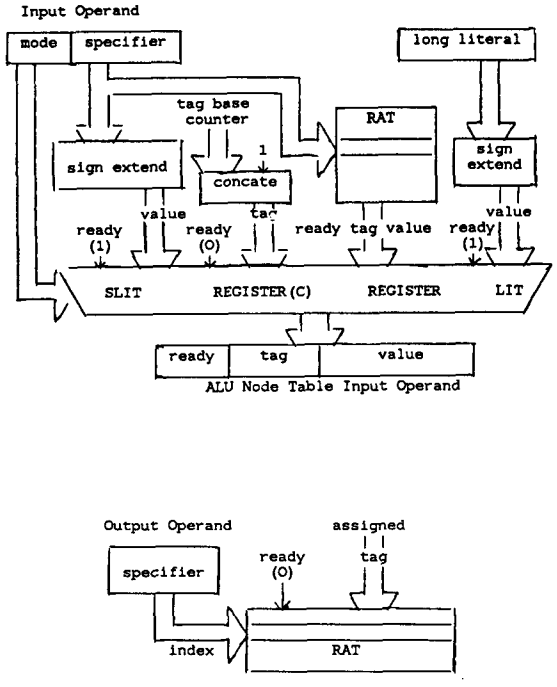

Figure 5. ALU Operation Merging

If operands of an operation are ready and the appropriate function unit is available, the operation is directly transmitted for execution. Otherwise it is inserted to the node table.

Each register alias table entry consists of two back-to-back parts: current and backup. Whenever the previous instruction invokes a branch prediction, the current part of each entry is saved in the backup part in one cycle. The backup part is maintained (distributed, see section 4.5) so that it keeps a version of the register alias table where none of the instructions ever entered the machine after the branch went into effect. When a branch prediction repair is signaled, the register alias table can be restored to its correct state in one cycle.

4.3. Node Scheduling.

When both (we use only two-input operations in HPSm) operands are ready, an operation is firable. The scheduling logic of ALU node table fires the oldest firable operation in each node table.

Memory operation scheduling is one of the critical issues in HPSm. The difficulty comes from the fact that memory dependency can not be resolved at instruction merging time due to unknown addresses. The current algorithm used is as follows:

(1) Fire the oldest firable memory write. If there is no firable memory write, try (2).

(2) Fire the oldest firable memory read if there is no younger waiting memory write in the node table.

It can be shown that this is not the optimal firing algorithm in terms of firing as many operations as the dependencies allow. However, it is one which we can easily implement with reasonable performance.

The active instruction table is a special node table used to detect when an HPSm instruction completes execution. An instruction completes execution when the results of all its operations have been distributed. The active instruction table entries monitor the distribution buses. There is one (ready bit, tag) pair for each operation in the corresponding HPSm instruction. The distribution logic sets the ready bit and gates in the distributed exception flags when the stored tag match the distributed tag. When both ready bits for operations are set, the active instruction table entry becomes firable. In other words, the corresponding HPSm instruction becomes retirable. The active instruction table entries are fired strictly in order to enforce the sequential semantics crossing HPSm instruction boundaries.

4.4. Node Execution.

All ALU operations execute in one cycle. Addition, subtraction, and, or, shifting, exclusive_or, and condition testing are provided in ALU.

The Memory function unit executes a memory write by performing the virtual address translation to detect potential exceptions and inserting it into the memory write buffer. Reads are executed by checking the memory write buffer and accessing the data cache memory. If there is a hit in the memory write buffer, the read result is forwarded from the memory write buffer. Otherwise the result comes from the memory system.

Branch confirmation is performed for conditional branches. The opcode (BTRUE, BFALSE) and the condition operand data together determine whether or not the branch prediction was correct. If the branch prediction was correct (hit), then the HPSm microengine performs the following operations in the next cycle: (1) Reset the branch prediction pending flag so that the next conditional branch can enter the data path, (2) Set the ready bit in the corresponding active instruction table entry so that the instruction can retire as far as branch prediction is concerned.

When a branch prediction turns out to be incorrect (miss), the HPSm microengine enters a repair cycle, as follows: (1) Redirect the instruction supply stream, (2) Invalidate all operations (node table entries) younger than the branch, (3) Reset the branch pending flag so that the next branch can enter the data path, (4) Restore the register alias table from the backup part, (5) Invalidate the memory write buffer entries younger than the branch, and (6) Invalidate all active instruction table entries younger than the branch. The repair is currently designed to take a single cycle. A less demanding design could be used if we wish because we have at least two cycles before the redirected instruction stream enters the data path.

When an active instruction table entry is fired into the retirement function unit, it performs the following operations: (1) If any exception flag is raised in the operation execution result, the entire microengine enters an exception handling sequence. This guarantees precise interrupts for memory management and debugging purposes. (2) If there is no exception, release the corresponding memory write buffer entry to the memory system. Since the memory write buffer release has a lower priority than the memory function unit, the retirement function may take more than one cycle waiting for the cache to become accessible. Note that we have performed virtual address translation when the memory write enters the memory write buffer. There will be no memory management exceptions when the write buffer entry is released. This simplifies the exception handling for access violations and page faults.

4.5. Result Distribution.

The results generated by the ALU and memory function units have to be distributed to their proper register alias table locations. There is a distribution bus for each function unit. All potential receivers of the FU result are connected to the distribution bus. If the distribution bus is valid and the distributed tag matches the tags stored in the register alias table entry, the ready bit is set and the distributed value is gated in when the distributed tag matches the stored tag. Distribution to the register alias table is done both for the current part and the backup part.

Operands of node table entries also monitor the relevant distribution bus. The schematic is shown in figure 6. The amount of logic involved in distributing into each node table operand argues for a small window size and thus small node table.

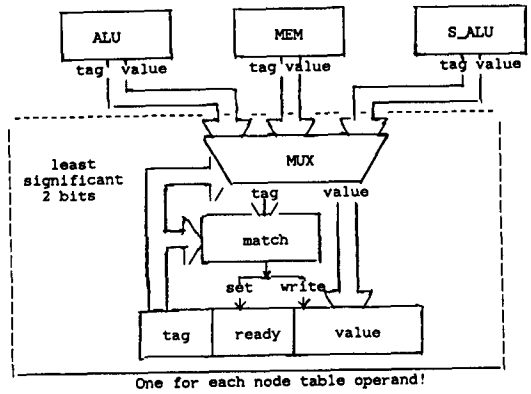

Figure 6. Distribution Logic

5. Performance Evaluation.

We have developed a register transfer level simulator to evaluate the design decisions and the performance of HPSm. A project to construct a C compiler for HPS [11] is now being conducted. The benchmarking reported here was performed by taking the VAX code generated by the UNIX* C compiler and hand translating it into HPSm code. We first translated the VAX† code into HPSm code in a straight forward way. Then, we arranged the code to reduce stalls and repairs due to conditional branches. Measured performance from both approaches are reported.

We also ran the same benchmarks on the RISC II simulator. The RISC codes from both optimizing and non-optimizing Berkeley RISC compilers were used. For each benchmark, we report the performance on the two simulated machines.

Table 1 summarizes the benchmarks we have measured so far. These benchmarks were chosen first for several reasons:

(1) The static code size of these benchmarks are sufficiently small that hand translation is doable, while we are awaiting completion of our C compiler.

(2) The selected benchmarks are both procedure intensive and conditional branch intensive. These control transfers are particularly difficult for machines like HPSm which try to exploit parallelism.

*UNIX is a trademark of Bell Laboratories.
†VAX is a trademark of Digital Equipment Corporation.

Table 1. Summary of measured RISC benchmarks	
Benchmark	Description
Towers(18)	Towers of Hanoi with 18 disks
Acker(3,6)	Ackerman's Function with input (3,6)
Qsort	Recursive version of quicksort
BenchE	String matching
BenchF	Setting, testing, and clearing bits in a bit vector
BenchH	inserting into linked list

(3) We wished to compare the performance of HPSm with another popular execution model, the Berkeley RISC II. According to reports on the Berkeley RISC II, it performed well on these benchmarks. Thus these benchmarks seem to be a good starting point for HPSm performance evaluation.

Table 2 summarizes the time and cycle comparison of HPSm and RISC II on the measured benchmarks. Columns *RISC A* and *RISC B* reports the measurement with non-optimizing and optimizing Berkeley RISC compilers respectively. Columns *HPSm A*and *HPSm B* reports the measurements with naive translation and translation with code rearrangement respectively. Code rearrangement techniques used were reducing the length of dependency chain, combining consecutive conditional branches to enlarge basic block size, rearranging branching structure to favor the more possible path, and to unravel tight loops. Note that we did not employ more conventional code optimizations techniques, such as lifting operation out of loop, deleting redundant operations, etc., to measurements reported under *HPSm B*. Thus we potentially have opportunity to further reduce the execution time.

There are three major factors contributing to the measured performance: clock cycle, concurrency, and size of dynamic program.

Tbale 2.a Performance Comparison (Time)				
Benchmark	RISC A	RISC B	HPSm A	HPSm B
Towers(18)	3,729ms	2,171ms	1,075ms	852ms
Acker(3,6)	3,164ms	2,484ms	230ms	207ms
Quicksort	867ms	764ms	175ms	175ms
BenchE	492us	337us	198us	116us
BenchF	178us	106us	50us	50us
BenchH	188us	148us	67us	56us

Table 2.b Performance Comparison (Cycle)				
Benchmark	RISC A	RISC B	HPSm A	HPSm B
Towers(18)	11.30M	6.59M	10.75M	8.52M
Acker(3,6)	9.39M	7.12M	2.30M	2.07M
Quicksort	2.63M	2.31M	1.75M	1.75M
BenchE	1.50K	1.02K	1.98K	1.16K
BenchF	0.54K	0.32K	0.50K	0.50K
BenchH	0.57K	0.45K	0.67K	0.56K

Cycle Time. Our estimated cycle time is 100ns assuming HPSm is going to be implemented in the same technology as RISC II whose cycle time was 330ns. The clock cycle of HPSm is kept small through pipelining the microengine and using small register file. Each HPSm instruction travels through at least 6 logical stages shown in figure 3. At each stage, we have one of the following happening at each stage: (1) one register access plus some simple gating, (2) one ALU operation (3) one cache memory access.

In RISC II, operations executed in one cycle are (1) reading from register file and then executing an ALU operation, (2) forwarding the result to the next instruction and writing back to register, (3) one cache memory access.

The very large register file in the RISC II, due to the overlapped window mechanism, also contributes to its longer cycle time. Of course, HPSm has to be equipped with faster cache memory than is RISC II in order to achieve projected cycle time.

Concurrency. HPSm exploits parallelism by (1) shipping more than one operation into machine in each cycle and (2) pipelining the microengine. In each cycle, we ship one of the following units into the machine.

- Two ALU operations, both can read from and write to register file.
- One ALU operation and one memory access, both can read from and write to register file.
- One ALU operation and one 13-bit literal constant.
- One control transfer operation with a 13-bit displacement.

Because we consistently handle dependencies both between operations within the same instructions and between operations across instructions, filling two operations into one instruction is not as difficult for us as for MIPS[12]. As we go on measuring substantial work-load benchmarks, we may expand each instruction to three or four operations. Such expansion had very little effect on the RISC benchmarks we have measured on due to very few operations available between control transfers.

In order to achieve enough parallelism to keep the stages busy, we perform branch prediction and data driven control. Also, to reduce the penalty for procedure returns, we separated program counter stack from the call frame stack.

The *BP hit* columns in table 3 give the branch prediction hit ratio, which is not very high for most of the benchmarks. The *M/R* columns in the same table gives the ratio of number instructions merged into the machine against number of instructions retired from the machine. The ratio is greater than one because some instructions were merged into HPSm due to incorrect branch prediction.

Since our branch repair mechanism is particularly efficient, we did not suffer badly from these branch prediction misses. If the branch prediction hit ratio had been higher, our performance gains would have been even more dramatic. This remains an ongoing research project for HPS.

In *I_S* columns, we provide measurement of two major stalls in instruction unit. The first number in such column is the percentage of time instruction unit stalled due to pending branch prediction. HPSm allows only one level of branch prediction. If another conditional branch arrives before the pending one is confirmed, the instruction unit stalls. If a program has tight loops, stalling of this kind becomes a performance problem. The code rearrangement in measurements under *HPSm B* tries to combine conditional branches (to enlarge basic block size) and to unravel the tight loops (to acquire the condition ahead of time). With these two techniques, we reduce the stalls at the cost of executing more instructions and probably lower branch prediction hit ratio.

The second number in *I_S* columns is the percentage of time the instruction unit stalls due to instruction buffer miss. In benchF and benchH, each path of the this case, instruction buffer miss becomes a major performance problem.

Number of Operations Executed. Table 4 summarizes the operations executed by the two simulators on the benchmarks. The number of HPSm operations executed was roughly comparable to the number of normal RISC II instructions executed.

In non-recursive but procedure-intensive benchmarks, HPSm tends to execute more ALU and memory operations to save and restore registers. This penalty is reduced by the safe-unsafe register saving convention of HPSm as described in section 3.3. When procedure calling reuse the same parameter, HPSm can save register moves to prepare the parameter. This effect is especially pronounced in the recursive benchmarks. However,

Table 3. Major Causes of Performance Degradation in HPSm

Bench	BP hit A	BP hit B	M/R A	M/R B	I_S A	I_S B
Towers	50%	50%	81/60	83/60	10%- 0%	2%- 0%
Acker	67%	66%	17/16	20/15	4%- 0%	3%- 0%
Qsort	54%	54%	89/63	103/78	44%- 0%	38%- 0%
BenchE	98%	96%	70/69	81/78	59%- 4%	21%- 8%
BenchF	40%	40%	27/21	27/21	3%-41%	3%-41%
BenchH	67%	69%	37/27	29/24	13%-29%	6%-40%

Table 4.a Operation Count: Towers(18)					
	ALU	MEM	CON	NOP	LIT
HPSm A	5.77M	3.15M	1.31M	1.05M	0.79M
HPSm B	5.77M	3.15M	1.31M	1.05M	0.79M
RISC A	7.50M	*0.28M	1.83M	1.31M	4.46M
RISC B	3.93M	*0.28M	1.31M	0.79M	1.84M

* 0.28M register window underflow/overflow operations.

Table 4.b Operation Count: Acker(3,6)					
	ALU	MEM	CON	NOP	LIT
HPSm A	1.12M	0.52M	0.52M	0.60M	0.34M
HPSm B	1.20M	0.52M	0.52M	0.34M	0.34M
RISC A	2.79M	*2.69M	0.83M	0.60M	1.72M
RISC B	1.46M	*2.69M	0.60M	0.09M	0.95M

* 2.69M register window underflow/overflow operations.

Table 4.c Operation Count: Qsort					
	ALU	MEM	CON	NOP	LIT
HPSm A	0.71M	0.19M	0.08M	0.21M	0.07M
HPSm B	0.99M	0.23M	0.08M	0.19M	0.07M
RISC A	1.66M	*0.18M	0.47M	0.17M	1.04M
RISC B	1.48M	*0.18M	0.46M	0.05M	1.02M

* 0.03M register window underflow/overflow operations.

Table 4.d Operation Count: BenchE					
	ALU	MEM	CON	NOP	LIT
HPSm A	584	224	219	134	221
HPSm B	918	322	113	100	115
RISC A	513	218	225	224	583
RISC B	366	218	222	12	572

Table 4.e Operation Count: BenchF					
	ALU	MEM	CON	NOP	LIT
HPSm A	259	108	24	12	21
HPSm B	259	108	24	12	21
RISC A	383	32	54	45	263
RISC B	332	31	33	0	211

Table 4.f Operation Count: BenchH					
	ALU	MEM	CON	NOP	LIT
HPSm A	241	104	66	18	103
HPSm B	252	112	35	13	72
RISC A	244	99	68	63	297
RISC B	179	93	58	25	260

register moves in RISC II, due to windowing, are required for preparing parameters even they stay in the same logical position.

Overall, HPSm execute more memory accesses. However, the write-back data cache with valid bit for each long (32-bit) data in HPSm saved most of the data cache memory miss penalty. Window overflow/underflow hurt RISC II performance substantially in Acker(3,6) where deeply nesting pattern of procedure calling prevailed. Each window overflow/underflow instruction takes at least two cycles (one for instruction fetch, one for execution). This degraded RISC II performance by 123% on Acker(3,6).

6. Concluding Remarks.

In this paper we have defined HPSm, a minimal version of HPS. HPS is our new restricted data flow microarchitecture, which we believe has significant potential for implementing high performance computing machines. We have specified the instruction set, data path, and timing for HPSm. We have reported the preliminary measurements obtained by executing several RISC benchmarks on our HPSm simulator, and we have compared our results with those obtained for the Berkeley RISC II. We intentionally restricted the functionality available to HPSm to what could reasonably fit in a single 1.5 micron NMOS VLSI chip. In all cases, our model of execution far outperformed the RISC II.

The above results auger well for the HPS restricted data flow model of execution. We hasten to add, however, that our results are very preliminary. They were performed on a simulation model, not real hardware. They report on toy benchmarks, not substantive applications. Still the results are very promising, and encourage us to continue our research in restricted data flow microarchitecture.

Acknowledgements.

The authors wish to acknowledge first the Digital Equipment Corporation for there generous support of our research. Linda Wright, formerly Head of Digital's Eastern Research Lab in Hudson Massachusetts, provided an environment during the summer of 1984 where our ideas could flourish; Bill Kania, formerly with Digital's Laboratory Data Products Group, was instrumental in DEC's providing major capital equipment grants that have greatly supported our ability to do research; Digital's External Research Grants Program, also provided major capital equipment; and Fernando Colon Osorio, head of Advanced Development with Digital's High Performance Systems/Clusters Group, provided funding of part of this work and first-rate technical interaction with his group on the tough problems. We also acknowledge that part of this work was sponsored by Defense Advance Research Projects Agency (DoD), Arpa Order No. 4871, monitored by Naval Electronic Systems Command under Contract No. N00039-84-C-0089. 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.

References.

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

[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] Patt, Y.N., Hwu, W., and Shebanow, M.C., "HPS, A New Microarchitecture: Rationale and Introduction" Proceedings of the 18th International Microprogramming Workshop, Asilomar, CA, December, 1985.

[5] 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.

[6] Hwu, W., Melvin, S., Shebanow, M.C., Chen, C., Wei, J., and Patt, N.Y., "An HPS Implementation of VAX; Initial Design and Analysis," Proceedings of the 19th Annual Hawaii International Conference on System Sciences, 1986.

[7] 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.

[8] 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.

[9] Kuck, D.J., et al, "Dependency Graphs and Compiler Optimizations," Proceedings of 8th POPL, January 1981.

[10] Patterson, D.A. and Sequin, C.H., "A VLSI RISC," Computer, 15, 9, September, 1982, 8-21.

[11] Shebanow, M.C., Patt, N.Y., Hwu, W., and Melvin, S., "A C Compiler for HPS I, a Highly Parallel Execution Engine," Proceedings of the 19th Annual Hawaii International Conference on System Sciences, 1986.

[12] Hennessy, J.L., "VLSI Processor Architecture," IEEE Transaction on Computers, C-33(12), December 1984, pp1221-1246.