Superconducting Processors for HTMT: Issues and Challenges

Kevin B. Theobald, Guang R. Gao
Computer Architecture and Parallel Systems Lab.
Dept. of Electrical and Computer Engineering
University of Delaware
Newark, Delaware 19716 USA
theobald@capsl.udel.edu, ggao@capsl.udel.edu

Thomas L. Sterling
NASA Jet Propulsion Laboratory /
Center for Advanced Computing Research
California Institute of Technology
Pasadena, California 91109 USA
tron@cacr.caltech.edu

Abstract

The Hybrid Technology Multi-Threading project is a long-term study of the feasibility of combining several emerging technologies to reach 1 petaFLOPS within ten years. HTMT will combine high-speed superconductor processors, semiconductor memories with built-in processors, high-speed optical interconnects, and high-density holographic storage.

While there are major challenges in all aspects of this project, those in processor architecture are the focus of this paper. Fundamental differences between RSFQ circuits and conventional semiconductor circuits, including a radical jump in clock speed, make today's processor design approaches inappropriate for HTMT. Sequential instruction dispatching, even within the lowest programming unit (a strand), will lead to unacceptably high latencies, hence poor performance. We propose alternative processor designs which use fine-grain synchronizations between individual instructions in order to avoid these bottlenecks.

1. Introduction

The Hybrid Technology Multi-Threading (HTMT) project [2, 4] is an ambitious, long-term study of the feasibility of combining several emerging technologies to produce, within ten years, a computer with a sustained speed of 1 petaFLOPS (10^{15} floating point operations per second) and 1 petabyte of memory. HTMT will combine high-speed superconductor processors, semiconductor memories with built-in processors, high-speed optical interconnects, and high-density holographic storage. It is believed that by combining the strengths of these diverse technologies, the speed and memory capacity goals can be achieved much sooner than through evolutionary improvements in current semiconductor technology alone.

The top-level architecture of an HTMT computer is shown in Figure 1. The main computation is performed by superconducting Rapid Single Flux Quantum (RSFQ) processors submerged in liquid helium and running at approximately 150GHz. A complete petaFLOPS machine is expected to have 4,096 processors executing an average of 2 floating point ops per clock cycle (6.7ps).

Recent studies suggest such a system may require a wide range of memory capacities, depending on the application [10]. While many applications can run at petaFLOPS speeds with as little as 10 terabytes, some important data-intensive problems will demand on the order of a petabyte of main or near-main memory. The processors are connected to a memory hierarchy consisting of

- Superconducting cryogenic RAM (CRAM) attached to the RSFQ processors;
- Static RAM buffers (SRAM), cooled by liquid nitrogen;
- Dynamic RAM memory (DRAM);
- Holographic memory (HRAM).

As one moves down the hierarchy, both memory capacity and access latency increase. The SRAM and DRAM incorporate Processor-In-Memory (PIM) technology [7] to provide intelligent control of the memory hierarchy. The processors are connected by a superconducting interconnection network, and an optical network called the "Data Vortex" provides complete interconnection between the SRAM and DRAM modules.

Extreme differences in memory access latencies across the hierarchy are anticipated, ranging from one or a few processor clock cycles for processor registers to the order of a million cycles for HRAM. For this reason, it has been recognized that conventional approaches to parallel processing are not suitable for HTMT. Preliminary work has been done

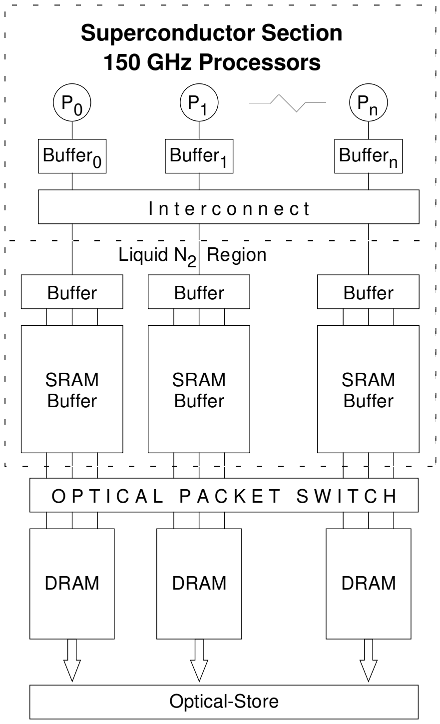


Figure 1. The HTMT architecture

on a program execution model that will allow HTMT to utilize all resources in spite of these huge latencies [5].

Briefly, the HTMT model is based on two key concepts: multithreading and percolation. Multithreading divides a program into separate units of computation, which can be run simultaneously to exploit parallelism and overlapped to hide latencies. The HTMT execution model has a multi-level hierarchy in which programs are divided into *threads*, each of which has its own local context, and *strands,* which share most memory with other strands in the same thread. Threads and strands are executed in the superconducting processors, where synchronization counters are used to ensure that a thread or strand only runs when all its dependences have been satisfied. Thus, strand scheduling can adapt to runtime conditions, reducing stalls due to unexpected latencies.

Outside the superconducting region, threads are combined into coarser units which are handled by the PIM processors. These processors do less computation, due to their lower speed, but they control the program at a higher level. They decide which threads to run on the superconducting processors, and use their highly-parallel access to the memory to collect data and contexts and "percolate" them to and from the superconducting memory. (Our previous work describes these ideas in detail [5].)

This previous work was based on general assumptions about the different components of HTMT. In this paper, we take a closer look at the superconducting processors. Like the HTMT as a whole, the superconducting processors use devices radically different from conventional semiconductors. The RSFQ processors will have latency problems analogous to the memory hierarchy. An examination of these issues has led us to the conclusion that conventional approaches to processor architecture are similarly inappropriate for HTMT, and that new techniques in instruction scheduling and execution are needed to ensure maximum performance of the processors.

This paper presents our preliminary views on HTMT processor design. It is based on our understanding of RSFQ technology, and relies both on general RSFQ papers and on papers dealing specifically with HTMT processor design [1, 11]. Performance and latency estimates come from these sources, where available. This paper presents general design issues, but does not discuss details such as the sizes of register sets, specific register sharing policies, etc. Such specific issues are closely tied both to hardware capabilities and to application needs. We believe detailed functional simulations of HTMT applications, coupled with feedback from further RSFQ implementation efforts, are necessary to come up with a meaningful design at that level of detail.

In the next section, we discuss the differences between RSFQ and CMOS technology, focusing on those which have an impact on architecture design. These observations lead to some general statements about logic design in RSFQ, from which we conclude that von Neumann-style sequential execution is not suitable for HTMT. In Section 3, we describe a top-level view of an RSFQ processor and discuss the functionality of each component. In Section 4, we focus on what is probably the most critical issue in processor design, the strand execution unit. We argue that a conventional RISC pipeline for each strand would still suffer from excess latencies, and propose alternative implementations that would yield better performance by reducing these latencies.

2. RSFQ versus CMOS

The renewed interest in using superconducting Josephson junctions for very high speed logic, in spite of its earlier disappointments, is due to a relatively recent breakthrough in circuit design, which allows Josephson junction circuits to run much faster than the few gigahertz achieved in previous efforts. The breakthrough is called Rapid Single Flux Quantum (RSFQ) technology. RSFQ circuits represent digital values using pulses corresponding to single quanta of magnetic flux, which circulate in superconducting loops controlled by Josephson junctions. The energy in one RSFQ pulse is much lower than the energy used to charge a node in a semiconductor. This gives RSFQ several advantages.

First, switching is much faster. RSFQ circuits fabricated today with a feature size of 3.5μm can switch in roughly 4 picoseconds (flip-flops running at 240GHz have recently been demonstrated). It is projected that by 2007, RSFQ circuits will be fabricated at 0.4–0.8μm and switch in less than 1ps. Allowing for several layers of logic per stage, this enables clock speeds of 150GHz. Smaller devices may allow clock speeds up to 200GHz. (High-temperature superconductors may run even faster, but this is still very speculative at this time.)

Second, transmission is much faster. RSFQ pulses can be transmitted along impedance-matched microstrip lines ballistically, as opposed to the diffusive signal propagation in semiconductor circuits. Signal transmission rates are expected to be 150 gigabits per second along internal lines and 30Gbps between adjacent superconducting chips in an MCM package.

Finally, power dissipation is significantly reduced. Since the pulses travel along resistance-free superconducting wires, both switching and transmission require little or no energy. In fact, the major sources of power dissipation in RSFQ circuits are static resistors used to bias the Josephson junctions and thermal transfer through the wires connecting the processors to the outside world. It is estimated that the total power dissipation will be around a kilowatt. Even though a closed cooling system could require up to 1,000 times as much power to remove the heat from the cryostat, an RSFQ-based system would still consume far less power than a pure semiconductor-based machine.

However, there are fundamental differences between RSFQ circuits and conventional semiconductor circuits, both in the density of the circuits and in the behavior of RSFQ pulses and circuits. These must be taken into account even at a high level of design. Because of these differences, one cannot simply "transliterate" existing CMOS designs into RSFQ technology; all levels from circuits up to high-level architecture must be completely rethought.

2.1. Circuit density

The first important difference is the density, in terms of clock speed. In a 1GHz processor, a signal can travel the entire length of the chip in a fraction of a clock cycle. This is not true for a 150GHz processor, in which a signal can only travel about 0.67mm in one clock cycle (assuming a speed of $\frac{1}{3}c$). With Cartesian circuit paths, a pulse transmitted from a single point can only reach an area of less than 0.9mm^2, as shown in Figure 2. Only a small portion of the chip can be reached in one clock cycle.

Although *any* chip running at 150GHz will have this limitation, the problem is worse for RSFQ circuits because of their lower density. With a junction size lower bound of 0.4μm expected in 2007,[1] only $5.5 \cdot 10^6 \lambda^2$ are reachable.

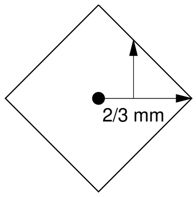


Figure 2. Area reachable in one clock cycle

Both RSFQ and CMOS logics are expected to have about the same circuit density, roughly 1 transistor or junction per $150\lambda^2$. This means that fewer than 37,000 junctions can be reached in one clock cycle; similar computations for 0.1μm CMOS transistors (expected in 2007 [9]) yield a "coverage" of over half a million transistors for the same period. If one considers round-trip time, only about 9,000 junctions, or about 900 gates, are reachable in one cycle, and this becomes even worse if one accounts for gate delays at either end.

This has important ramifications for the architecture of an RSFQ processor, for both timing and control must be decentralized. Global clocks are infeasible, and much of the interaction between modules will be asynchronous. Also, since signals will take multiple clock cycles to travel across even a fraction of the chip, it is impossible for one module to affect the operation of a distant module in the same or next cycle. For instance, if the execution unit and fetch unit are far apart, then a conditional branch may take many cycles to reach the fetch unit, which may consequently fetch many wrong instructions.

This means that the techniques processor designers have come to rely on to increase IPC (instruction-per-cycle) counts – out-of-order execution, register renaming, branch prediction, speculative execution, etc. – will simply be unavailable. It is physically impossible to place that much logic within one clock cycle of the control unit. Even instruction caches or large register sets are out of the question. On the other hand, it will be necessary to develop techniques to handle large latencies in on-chip components such as FP units, which may take 60 cycles or more due to their many stages of logic and long data paths.

2.2. Circuit behavior

Besides timing, there are key behavioral differences between RSFQ and CMOS circuits. These stem from the pri-

[1] The lower density of Josephson junction technology is not inherent, but merely reflects lack of development. It is currently not known what the limits are. The current state of RSFQ technology is that 3.5μm is used commercially and 1.5μm is used experimentally. Without the demand to justify spending billions of dollars for a state-of-the-art fabrication line and accelerated research, it is expected that RSFQ technology will need to go through its own learning curve, and use equipment from retired CMOS foundries for years.

mary difference in the way data is represented: quantized pulses in RSFQ versus charge differences in CMOS. Magnetic flux in superconductors is quantized, and an RSFQ pulse contains the energy corresponding to a single quantum of magnetic flux trapped in a superconducting loop. Because it is quantized, a pulse can't be split. When a pulse reaches a fork, it must go one way or the other. This is actually one of the key factors responsible for RSFQ logic's tremendous power savings, for energy is not disspated in transmitting data from one circuit stage to the next. However, it means that if one *wants* a pulse to split, a "pulse splitter" circuit must be used to *replicate* the pulse, introducing a small gate delay.

Consequently, large fan-ins or fan-outs (e.g., data or address lines in a register file) require more than simply tying a row of inputs or outputs to a common bus. (Buses can be implemented by converting RSFQ pulses to voltage levels, but this takes time and is therefore only practical for a large bus structure such as a RAM.) This problem is not unique to RSFQ; any circuit running in the hundred gigahertz range will have difficulties with the common conductor approach. But it means that the standard building blocks of digital logic may have substantially different performance characteristics. For instance, an RSFQ register file would most likely do the reads and writes through binary trees of splitting and merging circuits, which would introduce extra delays. As a result, tried-and-true techniques at all levels of the logic hierarchy may present design tradeoffs radically different from today's technology, and some approaches which work fine for CMOS may be unsuitable for RSFQ.

2.3. Implications for high-level architecture

The preceding generalities about circuit and logic design affect the way we think about the program execution model. We believe the von Neumann execution model, with its sequential semantics, is unsuitable for RSFQ technology.

First, in sequential execution, instruction fetching is more tightly coupled with execution than would be practical given our constraints. Any instruction cache large enough to be useful (especially given the need for concurrent instructions) would, by virtue of its size, take several clock cycles just to cross, and would have to be at least several clock cycles away from a central control unit. (One RSFQ design, based on a conventional instruction fetching unit, has an estimated fetch latency of 24 cycles [1].) Instruction fetching would need to be pipelined, and there would be so many stages between the I-cache and the branch unit that a conditional branch would incur a large branch penalty. Branch prediction hardware would consume too much area and would not be feasible.

Second, sequential execution often involves chains of operations with successive data dependences. Conventional processors handle these well with bypass logic, because the latencies of the functional units are usually 1–2 cycles for integer ops and 3–4 cycles for FP ops. Simple bypass logic won't cover the considerably higher latencies expected for RSFQ units. In order for an execution pipeline to execute one instruction per cycle from a single instruction stream, without dynamically reordering instructions, the stream would need to have many independent sequences of instructions interleaved at compile time. This has been difficult to do, even at the modest level of parallelism in today's VLIW machines, because basic blocks tend to be small, and interleaving multiple blocks statically requires that the compiler be able to predict whether two streams of instructions can always run in parallel without having one of them stalled by unpredictable dependences.

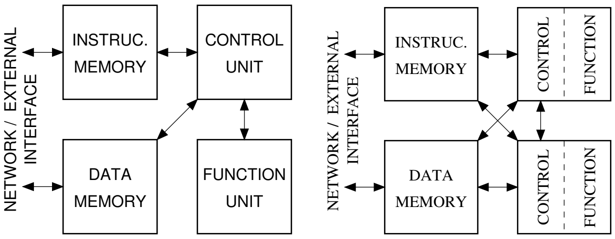
Finally, current superscalar processors can extract parallelism dynamically from a sequential program through scoreboarding and out-of-order execution. However, the hardware required to achieve even the modest gains of today's processors (typically less than 2 IPC) is a significant fraction of the total processor die area (around 20%). This would be prohibitively expensive for an RSFQ processor; any parallelism it could find would be wiped out by its own internal latencies.

For these reasons, a petaFLOPS computer based on RSFQ technology needs a processor designed with the constraints of RSFQ in mind. This generalization led us to the HTMT program execution model [5], in which execution latencies are hidden by interleaving concurrent strands (see Section 1. In the next two sections, we present a preliminary design for such a processor, beginning with a modular overview and finishing with a discussion of strand execution.

3. A processor architecture for RSFQ

This section presents a module-level architecture for an RSFQ processor supporting the HTMT program execution model. This description covers chip decomposition and the function of each module. The execution pipeline is perhaps the most critical component of this processor, and the one which has the greatest impact on the program execution model. We defer a discussion of that component to the next section.

It is expected that by 2007, chips containing 0.8μm Josephson junctions will be reliably fabricated up to 2 centimeters on a side, with a density of 1 million Josephson junctions per square centimeter for logic. A 2cm$\times$2cm logic chip would contain about 3 million junctions, allowing space for bonding pads. Individual chips can be combined using multi-chip modules, with inter-chip transmission rates reduced by a factor of 5 (30Gbps instead of



a) Separate functional unit

b) Functional units next to dispatch

Figure 3. Two designs for HTMT processors

150Gbps). A package with four 2cm×2cm chips would have a gate count comparable to today's largest microprocessors.

Two possible organizations for a 4-chip MCM are shown in Figure 3. In both designs, all memory except for registers and other temporary values are implemented on separate chips using cryogenic RAM (CRAM) technology. Data and instructions are stored in separate CRAM chips. The remaining two chips contain the execution pipelines and functional units. In configuration (a), they are kept on separate chips, which reduces inter-module connections but increases operation latencies because instructions must go from the control unit to the functional unit chip. In configuration (b), functional units are kept closer to the control units for reduced latency. Distributing control in this way requires more links between the units, including communication between the two control modules.

Each CRAM chip can store 256 kilobytes of memory. HTMT CRAM is organized similar to CMOS RAM, except that it is heavily pipelined and interleaved to support the high bandwidth requirements of the RSFQ processors [1]. In addition, each CRAM chip has additional logic specifically designed to support its role in the HTMT program execution model. Since waiting for individual fetches of instructions and data would cause long latencies, the CRAM chips will include support for more efficient data transfer operations, including streams and blocks of instructions and data. In addition, the instruction CRAM will be involved in the scheduling of threads and strands so that it can start fetching future blocks of instructions even while the RSFQ processor is finishing up the previous one. The CRAM chips also connect to the cryogenic interconnection network and the interface leading to the SRAM buffers and percolation logic.

In order to support an average issue rate of 2 FP operations per clock cycle, the functional unit will consist of a pipelined FP adder and a pipelined FP multiplier. In configuration (b), they would be divided between the two control/function chips. Depending on space, there may be other FP functions, such as divide, and long-latency integer operations such as multiply.

Logic blocks for low-latency operations, such as integer add, bitwise logic and shifting, can be replicated and placed near the execution pipelines. This would be more efficient than putting them in central modules, because the extra area would be relatively small, but would reduce the latencies of integer operations. Such operations are often on critical paths (e.g., loop counters, indices). Explicit support for short integer sizes (e.g., 16 and 32 bits) will speed up execution even more, because shorter integers take less time to add, and many computations on critical control paths (again, counters and indices) only need small ranges.

It might even be beneficial to replace one or two central pipelined FP units with a large set of small, non-pipelined units which reuse data paths to save area, in order to place them closer to the control units. However, load balancing among these units would be harder than with a central unit; simulation would have to be used to compare these two arrangements.

In either configuration, there will be several functional units, a moderate number of separate memory blocks, and many register sets storing the contexts of different strands. These all must be interconnected. Since buses have been ruled out (in Section 2.2), networks will have to be used. Networks currently under consideration for communication within a processor are variants of multistage interconnection networks [1, 11].

4. The execution pipeline

In the HTMT execution model outlined in Section 1, and in the architecture described in Section 3, each processor executes many strands concurrently. These strands are independent streams of computations. The compiler must explicitly identify the dependences for each strand and associate a counter with each to count the number of dependences which have not yet been satisfied. When another strand produces data or satisfies a dependence in some other way, it signals the processor to decrement that count. This strand synchronization mechanism guarantees that concurrently executing strands have no unsatisfied dependences with other strands. The scheduling of strands, utilizing this synchronization mechanism, uses independent hardware and can therefore run in parallel with strand execution. Thus, the only dependences that can stall an instruction must be between instructions in the same strand.

The question remains: what is the best way to execute individual strands in an RSFQ processor? We should try to minimize average overall instruction latency, because this will reduce the number of concurrent strands needed to achieve a given average IPC rate. The design tradeoffs in this section aren't likely to affect the latencies of func-

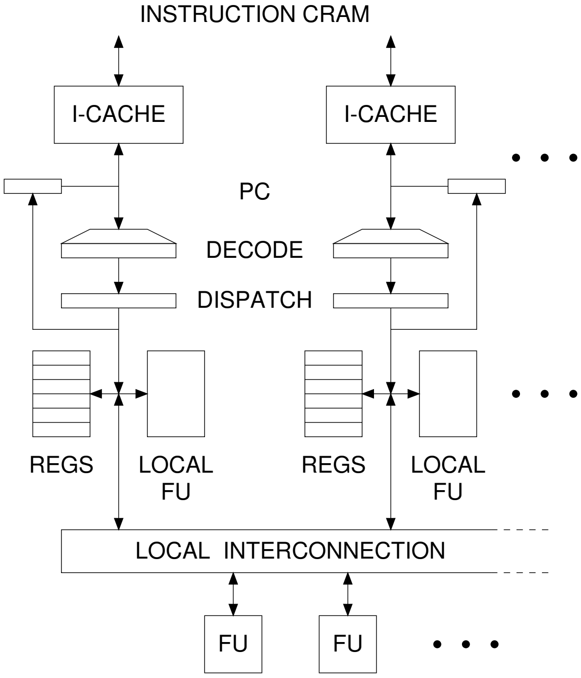


Figure 4. Multiple sequential HTMT pipelines

tional units, but reducing the overhead of fetching, decoding and issuing instructions to the functional units decreases latency overall, and is especially critical for integer operations. Most integer ops have smaller functional latencies, so instruction execution overheads are relatively greater. Furthermore, many integer ops are in critical paths (e.g., conditional control and indexing operations).

In this section, we compare four alternative approaches, beginning with a conventional multithreading approach and gradually moving to more radical schemes which apply the HTMT concept of inter-strand synchronization to *intra*-strand synchronization.

4.1. Implementations with sequential strands

The EARTH model [6], on which the HTMT model is based, assumes that strands are executed sequentially (at least semantically; superscalar techniques may be used for limited ILP). An EARTH-like implementation of a multiple-strand control unit might look something like Figure 4. Each strand controller has its own instruction fetch stage, instruction cache, decode logic, register file, and local functional unit. Instructions fetched from the I-cache go through one or more stages of decode logic. A dispatcher checks the next instruction for dependences to see if it is OK to issue it. When the instruction is released, the registers are fetched, and a packet consisting of the register contents, operator type, and destination register, is either passed to the local functional unit, or to the network for transfer to one of the pipelined functional units.

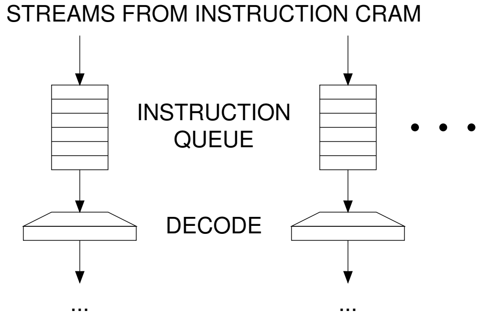


Figure 5. Improved instruction fetching

We believe that this design will yield poor performance. First, the conventional instruction fetch mechanism was shown in Section 2.3 to be a bottleneck in sequential execution. The standard mechanism to improve speed, the instruction cache, is too big for RSFQ. This will still be true for multiple strands. If all the strands share a common cache, latencies will be even worse due to increases in transit distances.

The bottlenecks with instruction fetch and branch penalties can be removed by modifying the instruction fetch mechanism as shown in Figure 5. In this implementation, each pipeline has a FIFO of undecoded instructions which are written in stream fashion by the instruction CRAM. This buffering allows the CRAM to send blocks of instructions rather than one at a time. Branches are eliminated entirely, and conditionals are handled either through predicated instructions (for short conditional clauses), or conditional synchronization commands (using the strand synchronization mechanism). This requires separating conditional clauses into separate strands. This idea has also been proposed as a way to reduce branch penalties in conventional processors, and compiler techniques have been developed to support it [8].

The standard way to prevent instructions from executing prematurely is to use a “scoreboard” in the dispatcher. The scoreboard keeps track of which registers are currently awaiting results from functional units. This can be very effective in systems where instruction latencies are mostly predictable; the compiler can try to schedule the code optimally so that independent instructions can run in parallel, with the scoreboard there to guard against unforeseen errors in the compiler’s prediction. Unfortunately, the instruction latencies in HTMT are expected to vary widely, mainly because the networks connecting the control units with the functional units can have contentions. If runtime conditions don’t match the scheduled instruction order, one instruction could stall the pipeline even though instructions behind it are eligible to go. The problem is that the dispatcher and

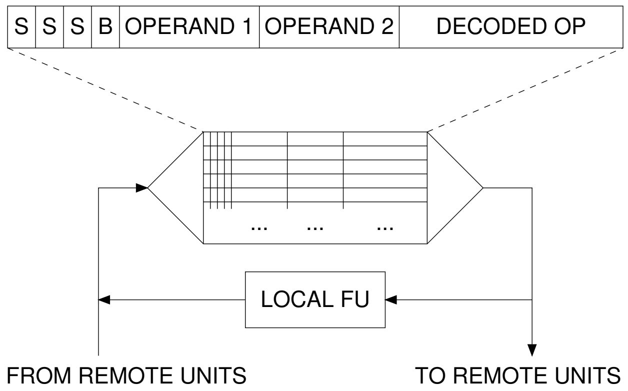


Figure 6. HTMT parallel execution mechanism

scoreboard are centralized; if we want those later instructions to "go around" the stalled instruction, then the processor must include costly out-of-order mechanisms, which were shown to be impractical for RSFQ in Section 2.1.

4.2. Implementations with intra-strand parallelism

Therefore, we believe that alternatives to sequential issuing of instructions should be considered, even within a strand. The HTMT model already establishes a mechanism for synchronizing strands to run in proper order, according to their dependences. The same technique can also be applied to individual instructions *within* a strand.

Instead of representing a strand as a series of instructions, we propose to represent a strand as a *partially ordered* graph in which the ordering is determined solely by data and control dependences. Any instruction whose dependences have been satisfied can be issued; no ready instruction can be blocked by a stalled instruction unless there is a dependence between the two. This increases the potential for parallelism. This *dependence-driven* instruction synchronization is somewhat similar to dataflow [3], but our strands are not restricted to functional (side-effect-free) program execution models like dataflow.

A possible implementation of our dependence-driven intra-strand synchronizer, suitable for HTMT, is shown in Figure 6. Instructions are loaded from the instruction CRAM into a series of instruction slots, being decoded along the way. Each slot contains the decoded op, which indicates the function desired and the identity of one or more destination slots. Each slot can hold up to three arguments: two 64-bit values to hold numerical quantities (up to double-precision FP), and a single bit for a boolean value, which can be used for conditional operators (marked B in the figure). Each of the three operands is guarded by a "synchronization" bit S. The bit S is set to indicate there is a valid operand in the corresponding register field. Bits corresponding to unused operands are kept set.

Each slot checks its own synchronization bits. When all three bits are turned on, a packet is formed from the entire row in the table: the opcode, operands, and where the results will go. This packet is dispatched, and all synchronization bits corresponding to operands actually used are reset. The logic to dispatch an instruction is simple and replicated for every slot, so slots may be initiated at random.

Given the problems with buses discussed in Section 2.2, the packets have to pass through a local multiplexing tree. Though slots may execute at random, the multiplexing tree will cause the packets coming out to be ordered. At the output of the tree, additional hardware checks the contents of the opcode field to determine which FU to use, either the local FU or a remote, pipelined unit. If the latter is used, the packet is entered into the network leading to the pipelined units. The functional unit creates a new packet containing the value of the operand and the destination slots. When the packet gets back from the functional unit, it enters a de-multiplexing tree, where it is routed to the destination slot. The specified register is filled and the appropriate synchronization bit is set. Multiple slots may be specified if the de-multiplexing tree has the ability to replicate packets.

This method is highly parallel, but has one shortcoming. Operand storage must be allocated for each instruction (129 bits apiece in our example). Yet each instruction may be receiving data for only a short period of time. Fewer storage registers may be required if they can be shared by several instructions.

An alternative mechanism, shown in Figure 7, addresses this shortcoming. This approach also uses synchronization bits to enforce dependences. However, both the processor architecture and the code of an argument fetch machine are more like conventional architectures. There is a register file, and each instruction specifies registers as in an ordinary register-to-register RISC instruction. Here, each instruction only has synchronization bits and an opcode; the latter specifies the functional unit desired, a pair of source registers and a destination register, and a list of slots whose instructions will use the result. Packets are formed and dispatched as before, but in this case, the packets first go to a set of registers, where the values are read and attached to the packet. The new packet goes to the local or remote functional unit, as before. When the result packet comes back, the value must go to the register file to be written, and the synchronization bits in the specified slot(s) are set. The hardware must ensure the registers are written before the synchronization bits.

This architecture permits greater efficiency in storage. Registers can be reused, much as they are in conventional processors, using compiler techniques such as register coloring. However, there is a tradeoff. The register file re-

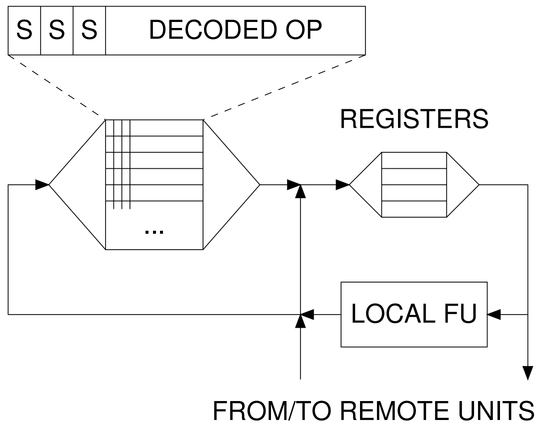


Figure 7. Register-efficient parallel execution

quires the same kind of access trees as the instruction slots. This means that register access is likely to take more than one additional clock cycle. This increases the latency of individual instructions, which means more parallel threads are required to utilize the functional units fully. Given this tradeoff, additional RSFQ design studies and HTMT simulations will be required to choose which technique to use; at this time no decision has been made.

5. Conclusions

RSFQ technology has the potential to give HTMT the benefits of both extremely fast clock speeds and minute power consumption. However, there are fundamental differences between RSFQ and semiconductor circuits, and conventional processor design techniques are not appropriate for this new technology. In particular, only a small amount of logic is reachable within a few clock cycles. This means that familiar processor features, such as reordering logic and instruction caches, may not be practical in HTMT.

A conventional sequential issuing of instructions within one strand, as in the EARTH model on which the HTMT execution model is based, will suffer high latencies between instructions within a strand, which means that more strands are needed to hide these latencies. A processor which allows fine-grain synchronization of individual instructions, even among those within the same strand, will have lower latencies, which will reduce the total strand count requirements.

Acknowledgements

This work was performed for the Jet Propulsion Laboratory, California Institute of Technology, and was sponsored by the Defense Advanced Research Projects Agency (DARPA) and the National Security Agency (NSA) through an agreement with the National Aeronautics and Space Administration. The authors would also like to thank Konstantin Likharev (SUNY RSFQ Group) and Andrés Márquez (CAPSL) for their valuable suggestions.

References

[1] P. Bunyk, M. Dorojevets, K. Likharev, P. Litskevich, Y. Pogudin, S. Polonsky, and D. Zinoviev. HTMT SBC processor design concept. Internal report, RFSQ Group, State University of New York at Stony Brook, October 1997. Draft.

[2] California Institute of Technology and NASA Jet Propulsion Laboratory. *Proceedings of the First Workshop on Hybrid Technology Multithreaded Architecture for Very High Performance Computing*, Pasadena, California, February 1997.

[3] J. B. Dennis. The evolution of "static" data-flow architecture. In J.-L. Gaudiot and L. Bic, editors, *Advanced Topics in Data-Flow Computing*, chapter 2, pages 35–91. Prentice-Hall, Englewood Cliffs, New Jersey, 1991. Book contains papers presented at the First Workshop on Data-Flow Computing, Eilat, Israel, May 1989.

[4] G. Gao, K. K. Likharev, P. C. Messina, and T. L. Sterling. Hybrid technology multi-threaded architecture. In *Proceedings of Frontiers '96: The Sixth Symposium on the Frontiers of Massively Parallel Computation*, pages 98–105, Annapolis, Maryland, October 1996.

[5] G. R. Gao, K. B. Theobald, A. Márquez, and T. Sterling. The HTMT program execution model. CAPSL Technical Memo 09, Department of Electrical and Computer Engineering, University of Delaware, Newark, Delaware, July 1997. In ftp://ftp.capsl.udel.edu/pub/doc/memos.

[6] H. H. J. Hum, O. Maquelin, K. B. Theobald, X. Tian, G. R. Gao, and L. J. Hendren. A study of the EARTH-MANNA multithreaded system. *International Journal of Parallel Programming*, 24(4):319–347, August 1996.

[7] P. M. Kogge, S. C. Bass, J. B. Brockman, D. Z. Chen, and E. Sha. Pursuing a petaflop: Point designs for 100 TF computers using PIM technologies. In *Proceedings of Frontiers '96: The Sixth Symposium on the Frontiers of Massively Parallel Computation*, pages 88–97, Annapolis, Maryland, October 1996.

[8] A. Márquez, K. B. Theobald, X. Tang, and G. R. Gao. A superstrand architecture. CAPSL Technical Memo 14, Department of Electrical and Computer Engineering, University of Delaware, Newark, Delaware, December 1997. In ftp://ftp.capsl.udel.edu/pub/doc/memos.

[9] Semiconductor Industry Association, San Jose, California. *The National Technology Roadmap for Semiconductors*, 1994. Revised.

[10] R. Stevens. Applications of petaFLOPS computers. In *Proceedings of the First Workshop on Hybrid Technology Multithreaded Architecture for Very High Performance Computing* [2].

[11] SUNY RSFQ Group. SCOPE: Superconductor processing element for petaFLOPS-scale computing. Internal report, RFSQ Group, State University of New York at Stony Brook, June 1997. Draft.