POD: A Parallel-On-Die Architecture

Dong Hyuk Woo[1] Joshua B. Fryman[2] Allan D. Knies[3] Marsha Eng[2] Hsien-Hsin S. Lee[1]

[1]School of Electrical and Computer Engineering
Georgia Institute of Technology
Atlanta, GA 30332
{dhwoo, leehs}@ece.gatech.edu

[2]Microprocessor Technology Labs
Intel Corporation
Santa Clara, CA 95052
{joshua.b.fryman, marsha.eng}@intel.com

[3]Intel Research Berkeley
Berkeley, CA 94704
allan.knies@intel.com

Technical Report GIT-CERCS-07-09

May 2007

Abstract

As power constraints, complexity and design verification cost make it difficult to improve single-stream performance, parallel computing paradigm is taking a place amongst mainstream high-volume architectures. Most current commercial designs focus on MIMD-style CMPs built with rather complex single cores. While such designs provide a degree of generality, they may not be the most efficient way to build processors for applications with inherently scalable parallelism. These designs have been proven to work well for certain classes of applications such as transaction processing, but they have driven the development of new languages and complex architectural features.

Instead of building MIMD-CMPs for all workloads, we propose an alternative parallel on-die many-core architecture called POD based on a large SIMD PE array. POD helps to address the key challenges of on-chip communication bandwidth, area limitations, and energy consumed by routers by factoring out features necessary for MIMD machines and focusing on architectures that match many scalable workloads. In this paper, we evaluate and quantify the advantages of the POD architecture based its ISA on a commercially relevant CISC architecture and show that it can be as efficient as more specialized array processors based on one-off ISAs. Our single-chip POD is capable of best-in-class scalar performance up to 1.5 TFLOPS of single-precision floating-point arithmetic. Our experimental results show that in some application domains, our architecture can achieve nearly linear speedup on a large number of SIMD PEs, and this speedup is much bigger than the maximum speedup that MIMD-CMPs on the same die size can achieve. Furthermore, owing to synchronized computation and communication, it shows that POD can efficiently suppress energy consumption on the novel communication method in our interconnection network.

1 Introduction

Although several commercial processors offer multiple cores on a single die, many challenging issues need to be addressed before these designs can make efficient use of their parallel resources. The open questions of how they will be used, what architectural features they will have, and how they will be programmed remain unclear. A simple solution will be fitting a proven parallel architecture such as a MIMD-based multiprocessor onto a single die. However, there are several new challenges that need to be overcome for such implementation, including power constraint, efficiency in the area usage and interconnection network, different target users and applications, etc., thus it is worth investigating other architectures as an alternative design in future many-core era.

MIMD-based large scale multiprocessors were more popular than massive SIMD machines because they offer two advantages: first, they leverage off-the-shelf microprocessor economies of scale; second, the flexibility of either running a larger number of independent workloads or a smaller number of parallel workloads. However, in an age when an entire SIMD array can be placed on a single processor die and this processor is sold to high-volume many-core processor market, the economies of scale becomes possible. Furthermore, lower manufacturing cost will make it feasible for one to have both a MIMD-based CMP and a SIMD array on the same chip, thus providing flexibility may become less an issue.

Besides, on-die massive SIMD machines provide substantial advantages in both area and power efficiency for future many-core processors. Unlike traditional large scale MIMD-based MP systems where space occupied by machines are not practically important, in a many-core era, the area of each core will determine the achievable performance. The larger each core is, the more likely the performance/area efficiency will go down. A Processing Element (PE) of a massive SIMD machine is typically much smaller than a processor node of a MIMD-based MP system. For example, one PE of our proposed architecture consumes only 12% of die area compared to a full-fledged Intel 64[1] core due to the lack of complex CISC instruction decoding logic, branch predictors, TLBs, and instruction caches in each SIMD PE. Consequently, 64 SIMD PEs can be integrated onto the same die with one host Intel 64 core while only eight Intel 64 cores can be integrated with the same area based on the 45nm process technology. Similarly, a SIMD PE will consume a lot less energy than an Intel 64 core for executing the same operation. Such on-die SIMD PE array enables the possibility of exploiting large data parallelism to achieve supercomputing performance with highly economical area and power efficiency.

Another major difference between a large scale MP system and a single-die many-core processor is in the design of interconnection network (ICN). Different types of interconnection network, e.g. hypercube or crossbar, can be implemented on massive MP systems whereas many cores on a chip may not have such luxury due to floorplanning constraint and limited area and power budget. According to MIT RAW [34, 16], area and power consumed by wires and routers account for 40% of the die area and 38% of the overall chip power. Recently, the packet switched mesh based MIMD-CMP is found to consume unsustainable energy in its router logic, and its unpredictable communication pattern prevents designers from using common low-power techniques such as clock gating [6]. In contrast, as all computation and communication are synchronized on a SIMD array, routers can be eliminated and the overall power consumption can be better harnessed.

Last but not least, programmers and workloads for single-chip many-core system will be different. Users of traditional MP systems are typically a small number of well-trained programmers who are well-versed in writing, debugging and optimizing their parallel code. Nonetheless, it is difficult to anticipate that programmers of high-volume many-core processor market would be able to handle sophisticated and subtle parallelization issues such as debugging data racing, balancing parallelism and data locality, hiding hard-to-predict communication overheads, etc. Furthermore, future killer applications for high-volume many-core processors will be content-centric applications such as 3D graphics and rich multimedia that are more SIMD-friendly. A simple SIMD programming model is not only easier to debug, but also well-matched to data-parallel applications.

[1]"Intel 32" was previously known as IA32 or x86. "Intel 64" was previously known as IA32e, EM64T, or x86_64.

In this paper, we propose an architecture that can efficiently support scalable parallel applications while minimizing the complexity of programming a many-core system. To achieve this goal, we revisit the designs of data-parallel SIMD computers [5, 36], but we focus on two realities: (1) current industry trends in ISAs and programming languages, and (2) to satisfy the new on-chip requirements. Since broad acceptance and software compatibility nearly necessitate Intel 64 compatibility, we accept this as a basic starting point for our work.

Our research goals are to provide best in class *performance per watt* across the space of many-cores, graphics processors, and media accelerators while maintaining ISA compatibility and providing a computing model that is more general than the specialized graphics and media processors. The primary contributions of this work over the prior efforts in SIMD machine research are:

- We propose a new Parallel-On-Die (POD) many-core architecture based on a large SIMD PE array.
- Our SIMD PE array represents a first-class citizen (rather than co-processors) in the system with respect to virtual memory and the host core.
- We address the on-chip wire latency problems for lock-step execution and communications.
- We eliminate complex global networks and propose an efficient communication topology in terms of energy, area and latency to address the on-die wiring limitations in our architecture.
- We reduce the need of using thousands of SIMD PEs to dozens while attaining multi-TFLOP performance.
- Our architecture maintains semantic compatibility with an existing CISC architecture.

The rest of this paper is organized as follows. In Section 2, we discuss the background. Section 3 explores the contemporary challenges with respect to large-scale SIMD architectures. In Section 4, we propose a modification of a current Intel 64-based microprocessor platform, and in Section 5, we describe ISA support and programming model for POD architecture. Section 6 describes our simulator, and analyzes the performance results using several benchmark programs. Section 7 concludes.

2 Background

Our work revisits the SIMD concepts, expands and interprets them with modern requirements and technological constraints. The closest architectures to our design are the Maspar MP-1 [5, 22] and the Thinking Machines CM-2 [36]. Both of these machines were centered on the concept of a very large number of processing tiles for parallel calculations. This style of SIMD machine was broadly characterized by having a front-end system that consists of a host processor. Both machines took advantage of the relatively "free" wire latency compared to the transistor switching speed, and had a low communication latency between nearest neighbors (4 cycles for CM-2 and 8 for Maspar). The nearest-neighbor connections were supplemented with sophisticated networks to allow all-to-all, unstructured, and long-distance communication. The immense number of PEs led to an elaborated global network design with many layers of switches and crossbars.

Other important SIMD machines include the Solomon [32], the IBM GF-11 [17], and the Illiac IV [8]. Some hybrid efforts between MIMD and SIMD were undertaken [33], but remain on the fringe. Systolic arrays [7, 19] resemble SIMD machines, but were tailored for applications whose computations fit a narrower structure. In most cases, to be highly efficient, the programmers need to map the application's execution and data flow in details to the target machine. Tarantula [11] extended Alpha ISA with a slew of new instructions and state. EV8 understands entire Vector ISA for renaming/retirement/speculation issue and supports deep conditionals via masks, but does not support intercommunication among vector units except for gather-scatter ops.

Imagine [4, 15, 29] is a stream-model processor, but uses a normal host and acts as a coprocessor. Imagine uses a 128KB stream register file to contain data, while each attached FPU has a local register file and several dedicated ALUs to operate on the stream data in a producer-consumer model, unlike our architecture which uses private SRAMs to allow local data reuse. Imagine also uses instruction memories and fetch/decode patterns, but capitalized on an 8-wide SIMD ability inside each full ALU tile. The drawback is that each of the eight sub-ALU blocks is wired with a crossbar to the full SRF, and that applications must be ported to a stream-based model for exposing the parallelism.

More recently, non-SIMD tile-based architectures, e.g. the MIT RAW [35, 34] and the UT-Austin TRIPS [30], were proposed. The RAW processor provides local instruction and data caches, contains 64KB of RAM for each processor tile to program a dynamic/static routers, and has its ALU bypass network tied directly into the interconnect network (ICN). To support its programming model, the RAW also has large memories and additional modes dedicated to the routing logic for use based on application needs for either static or dynamic routing. This approach requires the extra logic and power compared to a SIMD design. TRIPS is tile-based, but uses more sophisticated ISA mechanisms relying on compiler's analysis and static placement of computation. It also provides separate ICNs for data and instruction movement and each tile has a complete CPU. The entire design is intended to support highly-speculative parallelization techniques. The IBM Cell processor [13] is an alternative to tile-based designs by hosting eight Synergistic Processor Elements (SPEs) on a PowerPC host. While the Cell processor is similar in concept, these SPEs, MIMD in pattern, are complete with instruction fetch, decode, branch control, and load-store queues. Setting aside the complexity of MIMD programming when compared to our SIMD model, the peak performance of the Cell is below what our POD can attain.

Finally, the PicoChip [10] and the Connex Machine [3] are on-chip massively parallel machine implementation, but they are special-purpose processors, and not SIMD machine. The Morphosys [31] proposed dynamically reconfigurable SoC architecture. It includes an array of reconfigurable cells working in SIMD fashion and contains a sophisticated programmable tri-level ICN.

3 Modern Considerations

One primary reason attributed to the commercial failure of SIMD machines hinges on the rate of growth of microprocessor performance relative to the time to market for SIMD machines [25]. With a concept-to-market time of 36+ months, a new SIMD machine would be released with a scalar performance pegged to the state of the art 1.5-3 years prior to the first sales. Meanwhile, commercial microprocessors leapt ahead by up to a 4x performance improvement following along Moore's Law. Considering the cost of early SIMD machines versus microprocessors, most consumers who could have benefited from parallelization chose not to, letting Moore's law carry on their evolutionary growth as opposed to accepting a major architectural change.

Today, while single stream performance has not reached its limit, its progress has slowed dramatically due to performance per watt and complexity-effectiveness issues [24]. While process technology continues to advance, industry leaders look to many-core architectures as the roadmap of the future. With the resultant slowdown in single core improvements, we call in to question the reasons for SIMD machine failure of the past and explore how the original ideas might fit into today's changing landscape.

Our work is based on a very different set of constraints than those that existed when the original SIMD machines were built. First, since we wish to provide a backward compatible processor (e.g. Intel 64) without dramatically changing its ISA, we cannot overly simplify the processing elements. Second, we design both computation and communication architectures simultaneously so that they only consume sustainable amount of energy, which is a new requirement for on-chip massively parallel machines. Third, prior SIMD machines (e.g. the Maspar MP-1 and CM-2) were not limited to planar interconnects because their processing elements were connected across multiple boards via backplanes and wires. When they were built, wire delays were less critical compared to transistor switching speeds. This reduced sensitivity to wire latency is even more pronounced the farther back one goes in the SIMD genealogical tree.

Since we propose to place an entire implementation of a host processor core and a large SIMD array on a single die, we are limited to planar networks [9, 18] and thus do not have the luxury of high-dimensionality networks.

This is a key difference in technology over the past two decades. In our design, the wire delay to cross a single PE in a straight line is a function of the manufacturing process, and is intended to be very fast (1 - 3 cycles). In addition to having to adopt to the infeasibility of a global data network, we are also faced with the problem of synchronously broadcasting the SIMD instruction stream across the PE array in a power-efficient manner. At all stages of design and consideration, wire delays dominated our thinking and drove simplification of the PEs. For our design, we have restricted the PE size so that a signal can propagate across it in a single cycle.

To simplify the design and minimize complexity, we borrow a similar concept from the interconnect of the MIT RAW processor. The interconnect was directly wired to the pipeline of each RAW processor tile, supporting only nearest-neighbor communications and using fixed algorithms to implement more complex routing. However, because our switches only support single-hop routing and every switch is always communicating in the same direction, the design/size of our routers is much smaller and simpler and provides communications limited only by wire delay. We require no substantial buffering or extra support to handle deadlock, livelock, or drain requirements. We do not use the RAW model of multiple ICN modules within a tile to handle alternatives of static or dynamic routing, and instead propose simple algorithms for non-nearest-neighbor communications.

In addition to the interconnect simplification, our design uses far fewer cores than the thousands of cores supported by the MP-1 and CM-2. In which, each PE was only a 1- or 4-bit ALU internally, any given arithmetic operation (e.g., 64-bit integer add) required the use of several PEs and/or multiple cycles to compute it. Since process technology allows full 128-bit SSE units to be constructed in a small area, this allows us to provide semantic compatibility with the host processor and to reduce the number of SIMD PEs needed to achieve high rates of computation. Finally, the SIMD architecture we present is a first-class citizen with respect to the rest of the system. The proposed SIMD array directly interfaces the rest of the system through standard virtual memory interfaces so each PE in the SIMD array can directly access main memory. This is in addition to the private SRAM each PE has for local data and computation results.

4 Parallel-On-Die Architecture

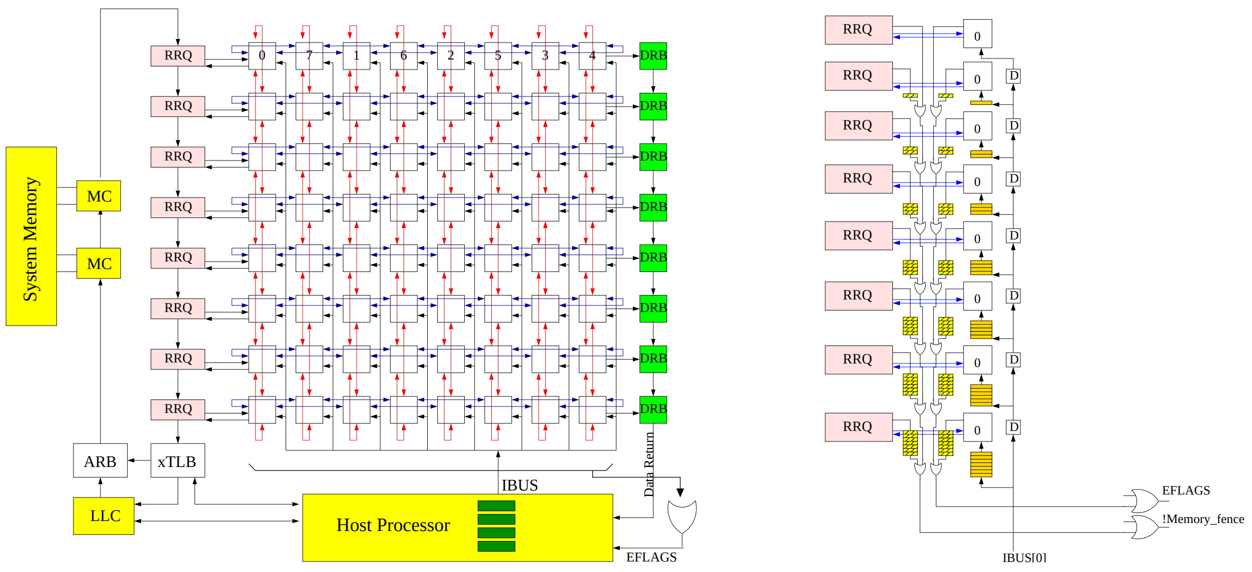
In this section, we propose a new massively parallel processor architecture called *Parallel-On-Die* or *POD*. POD is a fully integrated processing fabric on a single die based on the Intel 64 ISA and provides best-in-class single-stream performance for scalar applications as well as a robust parallel SIMD PE array for scalable parallel application execution. The high-level block diagram of the POD architecture is illustrated in Figure 1(a). The POD system will fully boot a normal OS and run every legacy application under that OS without problem, thereby presenting the SIMD PE array we attach to it as a pseudo-coprocessor.

4.1 Host Processor Core

The principal claim to best-in-class scalar performance is provided by a high performance host core such as a core from the Intel Core 2 Duo processor. This provides not only flawless single application execution, but also presents a known, compatible platform to the OS, programmers, and applications to reduce the complexity of bootstrapping new functionality and applications.

The target SIMD PE array is a sea of $n \times n$ tiles, where we show $n = 8$ in Figure 1(a). The host processor core is capable of broadcasting instructions, each of which has the same fixed size, as well as broadcasting 64-bit register values as might be needed for immediates or loop conditions. The PE array generates a flag-tree output which is tied together logically via *OR* gates, and routed back to the host.

To allow the addition of a SIMD array while minimally altering the existing Intel 64 ISA, we propose to add a new instruction *prefi x byte* on existing opcodes to denote a parallel-instruction. When this prefix byte is encountered, the host core could implement the instruction by one of several methods. The simplest one is to broadcast it blindly to the POD, but this would require that each PE be able to decode the CISC ISA. A more



(a) POD Architecture (b) One Detailed POD Column

Figure 1: POD Architecture

flexible mechanism is to run a dynamic binary translator or JIT to capture such instructions and selectively decode, broadcast, and optimize them.

In this work, to avoid the complexity of supporting a new parallel prefix, we instead chose to implement a handful of new instructions that allow us to send native PE instructions from the host. The details of the instruction extension will be described in Section 5.1.

4.2 SIMD Processing Element

Each PE tile consists of a high performance arithmetic unit with its own private registers and local SRAM memory space. To provide a baseline performance level and to support a subset of the host instruction set, we chose to modify an existing 128-bit SSE engine (including SSE, SSE2 and SSE3) from a contemporary Intel processor. This approach provides 4-wide SIMD execution units for single-precision IEEE floating point operations, or 2-wide for double-precision. We also added a fused multiply-add instruction to the SSE instructions to improve efficiency of the PE resources. By assuming an existing SSE engine design (with extension), we only need to add the surrounding logic to complete a standalone PE and it minimizes the difference between the host ISA and the PE ISA. The PE microarchitecture is shown in Figure 2.

As the Figure shows, each PE has two groups of registers including a 32-entry 64-bit general-purpose register file (`r0 - r31`) and a 32-entry 128-bit SSE register file (`xmm0 - xmm31`). While this exceeds the size of Intel Architecture register files, additional resources may or may not be exposed to a programmer directly. In the future, a dynamic binary translator could optimize the host processor's use of the original 16 `xmm` registers to make use of the PEs 32 `xmm` registers. For the purposes of our evaluation, all PE registers are exposed in the POD ISA during our hand-coded assembly optimizations.

On the input side of the Figure, the PE also contains a *Mask Stack*, which is to be used for conditional execution such as *if-then-else* clauses. Our PE implements a novel way to efficiently execute nested if-then-else clauses, examples of which are in Section 5. On the execution path, there are three individual pipelines within each PE tile

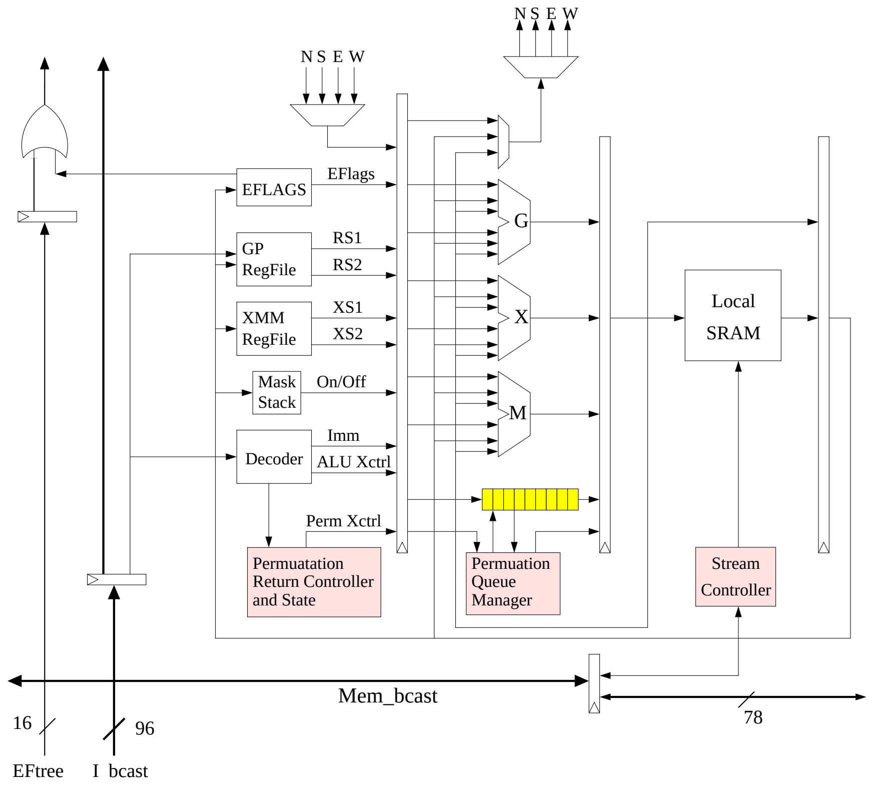


Figure 2: A Processing Element Tile

— one for memory instructions (load, store, etc.) called *M-pipeline*, one *X-pipeline* for all SSE instructions, and one *G-pipeline* for generic integer non-SSE arithmetic (address calculation, constant generation, mask operations). Based on a 5 to 7 cycle latency for basic integer and floating point operations in the SSE pipeline and 3 cycles to local memory, we require between 15-35 registers to keep each PE fully running. Our selection of 32 xmm registers satisfies the majority of this range, and requires only one extra bit per source-destination register field in the PE instruction.

Also shown on the top of the Figure, each SIMD PE consists of four unidirectional input point-to-point links from the North, South, East, and West neighbors and four unidirectional output point-to-point links to the same neighbors. Each link is 144 bits wide, capable of latching up to 128 bits of data or register value every clock cycle. The rest of the 16 bits are used only during permutation routing to specify the coordinates of the source PE and destination PE. The permutation routing will be discussed in Section 4.4. These eight point-to-point links comprise the data torus for the POD communication patterns. To communicate with main memory, each PE is further enhanced with two unidirectional memory buses, discussed in Section 4.5.

The PE instructions are broadcast from the host via a special instruction with an immediate data field of 12 bytes. These 12 bytes forms a partially pre-decoded VLIW packet of three instructions (4 bytes per instruction) that eliminate CISC decoding overhead. Each VLIW packet has a fixed format of one G, one X, and one M pipeline instruction. Since the execution of PE instructions are broadcast and orchestrated by the host, there is no need for an instruction cache or associated blocks. Furthermore, since each PE is executing the same instruction and there is no instruction equivalent to a branch, no branch predictor or associated flush/control logic is required, keeping the PE small and simple. The salient features of the PE instruction set will be discussed in Section 5.

The needed control logic is made up of processing arriving PE instructions, register file and state access, the nearest-neighbor North-South-East-West interconnects via muxes, and a private SRAM accesses. Additional modules are included for the permutation routing control logic to implement complex routing patterns between PEs.

4.3 POD Interconnection Network

In the design of the interconnection network, we investigated two topologies — a 2D mesh and a full torus. We found that while a 2D mesh connect neighboring PEs is straightforward, it has certain drawbacks related to our SIMD routing control. Specifically, when an application needs to communicate from edge to edge, it will take $n - 1$ hops. Many parallel algorithms naturally rotate data across PEs (see results from the Cannon's algorithm used in DenseMMM in Section 6.2). This is a specific problem related to our simplification of the communication network: because all the switches route in the same direction at the same time, and because there is no dynamic routing, the extra flexibility of the torus was required.

As shown in Figure 1(a), our proposed design adopts a modified torus network. To minimize latency and maximize packing, each PE is designed to take less than one clock cycle for a signal to cross the entire PE itself. Ideally, each PE will be no larger in any direction than 95% of the wire distance in one clock cycle with all surrounding line drivers, buffers, and so forth. The ordering of the number labels inside the PEs of the top row in Figure 1(a) indicates the nearest-neighbor connection pattern. In the same way, we lay out the communication links for each column in the POD (north-south direction). In addition to providing shorter links, such a layout also leads to a deterministic communication latency.

As mentioned earlier, there are eight physical point-to-point links connected to each PE. At any given moment, only one direction (input and output) needs to be enabled. Since each nearest-neighbor communication pattern has a known latency, the links are not enabled during periods of pure computation or during periods when links in the other direction are not being used. This reduced power profile allows growth of the POD array to be limited only by the average power consumption of each PE and the manufacturing die reticle. This approach is compared to other tiled designs such as MIT RAW or TRIPS or where any of the ICN links could be active at the same time due to dynamic routing.

To enable SIMD-style instruction execution where every PE executes each instruction at the same global clock cycle, there are two options: (1) execute an instruction immediately upon arrival to a POD row, leading to a North-South *timezone effect*, or (2) buffering each arriving instruction for sufficient time such that every PE will execute the same instruction at the same instant.

The timezone effect can be challenging to work around for programmers and architects, as any given row will be executing instruction j, while the preceding row is executing $j + 1$ and the successor row is executing $j - 1$. To avoid undesired complexity for programmers, architects, and compilers, we use a buffering model to enable lock-step execution without suffering from the timezone effect.

Figure 1(b) shows such a model for one single column in the POD. Instructions are broadcast using the IBUS and are queued before being executed by the PE. It takes f_0 cycles to uniformly reach the first row ($f_0 = 4$ when $n = 8$), and for n rows, it takes $n - 1$ cycles before every PE executes the instruction. As shown, the queue size shrinks monotonically as the location of a PE gets farther away from the host processor. For an $n \times n$ POD, where $n = 8$, there are 7 entries for the bottom-most core while no queue is needed for the top-most core. The delay units (D block) are inserted to delay each instruction broadcast in order to synchronize the SIMD execution. Similarly, when gathering results (*e.g.,* EFLAGS) from PEs, the results from the cores closer to the host processor need to be delayed and wait in their queue till the farther results arrive for combining. These are depicted in the propagation paths with correct delay queues on the left-side of Figure 1(b). Compared to previous immediate execution model, there is zero overhead to the PEs with this implementation, excepting a buffer to hold the instructions broadcast. The round-trip latency for the host to evaluate conditional loop also remains same, which is, for n rows, $2 \times (n + f_o)$ cycles where $2n$ cycles is consumed for instruction and EFLAGS propagation, and $2f_o$ is fan-in and fan-out latency between the host and PEs in the first row.

4.4 Interaction Among PEs

As mentioned in Section 4.2, each PE has 4 uni-directional input point-to-point links and 4 uni-directional output point-to-point links. Each link pair implements a nearest-neighbor direct link. These eight links are also arranged such that they are glue-less drop-in components, with each neighboring PE only requiring direct wiring to complete the layout. This allows for dense packing, although there is a very high wire count. Note that neighbor-to-neighbor communication does require neither arbitration nor routing, because it is fully controlled by software. Consequently, each PE does not need to have buffers for communication, which is known to consume large energy on packet-switched on-chip interconnection [6].

Each PE can communicate with its nearest neighbor by either directly moving a register value of up to 128 bits, or by transferring memory in 64-bit chunks. In order to support streaming memory behavior between PEs, we support both single load-store style transfers as well as block-based transfers, with and without striding. Since the nearest neighbor latency for an interleaved torus is targeted to be two cycles or less, this allows for high throughput computation even when the algorithm requires neighboring registers and memory values. This is a major contrast to typical shared-cache interface implementations, where it can take 10 or more cycles to move a value between cores.

When one PE needs to communicate to another PE in a non-nearest-neighbor fashion, we use the k-permutation routing [12] in our interconnect design. Rather than provide dynamic wormhole routing hardware support for a relatively infrequent operation, we propose dedicated algorithms to drive the collective POD muxes into a series of sweeps to migrate all data to the intended targets. These algorithms require each PE to support n hardware buffer slots of the bit-size matching the point-to-point link width in an $n \times n$ SIMD array.

The basic algorithm proceeds by all PEs send messages to the East, with each message stopping when it reaches its target column. This takes $n - 1$ hops and at the end, at most n messages will be buffered in any one PE. At the end of this sweep, every message in every POD row will be in its target column. If we now apply the same algorithm to the North, we may require as many as $n^2 - 1$ steps respectively until all the buffered messages reach their target PE. As messages reach their target PE, they are processed (stored into the appropriate memory location). This two-phase sweeping algorithm ensures that for *any* permutation of routing, even all-to-one, all messages are delivered after a fixed latency. This fixed routing would not be an optimal solution, but each PE only needs to enable only one link at the same time, which is more energy efficient.

While the fixed latency may be high for such generic routing support, we have made the trade-off to keep nearest-neighbor communications fast, which is much more frequent event than generic routing. More optimized row-only and column-only sweeps of just $(n - 1)$ steps are possible for more structured communication to reduce the high latency of a full any-to-any communication. More details on the permutation routing can be found in Section A.

4.5 POD and System Memory Interaction

Aside from a 128KB private local SRAM dedicated to each PE, applications must also be able to communicate with the system memory through normal loads and stores. To manage this interaction, each PE is further enhanced with two unidirectional buses (MBUS) to the main memory via an interface called the *Row Response Queue* (RRQ). One bus streams data back from main memory to the PEs in the row, while the other bus streams data from the PEs in the row to the main memory. Because system memory operations of all PEs are synchronized, PEs can safely disable their MBUS and its related logic, to minimize energy consumption, while they are not communicating with the system memory. The RRQ is the queuing point for transactions in both directions, and in turn is connected to a memory ring with the host core's last level cache (LLC) and all memory controllers (MCs).

In our work, the conceptualized ring is composed of four separate rings as shown in Figure 3. There is one shared data ring, at 66 bytes wide, which represents the actual data to or from the MC one line at a time. There is a corresponding address ring of eight bytes to indicate what address a request or response payload corresponds to.

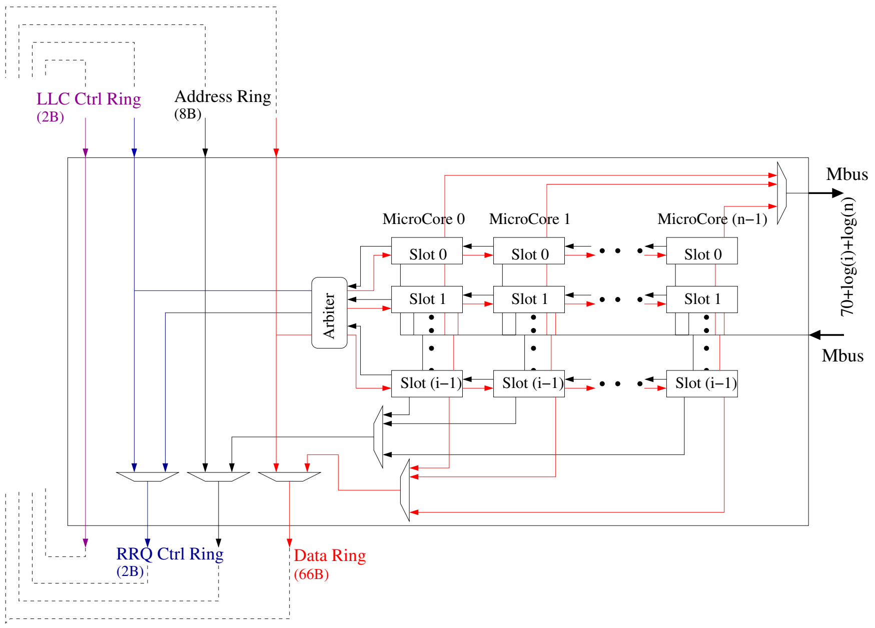


Figure 3: The RRQ state machine for queuing PE system memory requests and streaming responses from the MCs.

Then there are two control rings, one for the LLC and one for all RRQs to share. The premise is that the POD will always be a first-class participant in the memory hierarchy, but a second-class participant to the host core cache misses. Since every request from an RRQ must be acknowledged, whether positively or negatively, when the LLC needs to take over the data and/or address ring for higher priority traffic, an arbiter will set the necessary bits in the RRQ ring for failure and allow the original message to return to the originator for a later retry effort.

One PE in a row can generate up to i requests in the form of load-store traffic to system memory. Therefore, the RRQ must buffer each request from each PE and service them as it finds free slots on the ring. Each PE will only be able to use the i buffers reserved for it, since in traditional SIMD execution every PE will generate the same number of memory access requests at the same moment, varied only by masking controls.

There are separate instructions for loading and storing to local PE memory and for the global system memory. Different instruction flavors are provided to load/store single words and contiguous or strided block moves. System memory access use virtual addresses acquired from the host. In order to translate the given virtual address among all n^2 PEs, we share one pipelined TLB external to the host that the host manages. This $xTLB$ in Figure 1(a) need not be organized along traditional lines since the TLB lookup is not as critical as it is in the host processor – this allows for a super-pipelined, very high capacity xTLB to be implemented. In the event of a fault or miss event in the TLB, the host is notified and the request in the RRQ control ring is flagged as a TLB failure. When the host updates any TLB entry, a dedicated control signal in the RRQ control ring is set to indicate any prior TLB failure may now retry.

Typically, some form of coherence is essential between the collective POD PE SRAM storage regions and the system memory. To reduce the complexity and to evaluate the basic performance potential of the proposed architecture, we avoid coherence problems by requiring that any memory region that may be loaded into the private POD collective SRAM space to be marked as uncacheable to the host and associated cache hierarchy. While this leads to lower performance it provides sufficient simplification for our models, and can be improved in our further work.

4.6 Challenges in Integrating with Host Processor

Unlike conventional massive SIMD machines, our POD architecture integrates a massive SIMD PE array with a modern out-of-order host processor. To ensure the execution correctness, there are two major challenges to be addressed in the host processor: recovery from mis-speculation and out-of-order dispatch of POD instructions.

To support speculative execution, some recovery mechanism is required to roll the machine back to the correct architectural state. Unfortunately, implementing recovery mechanism in each PE will add a substantial overhead to both the area and power. To not complicate the PE design, we enforce the host processor to broadcast POD instructions in a non-speculative manner. In other words, the POD instructions will not be dispatched from the host until its corresponding branches are resolved. From performance standpoint, as long as the code that runs on the host does not depend on the results from the POD, this approach will not degrade the performance.[2] Another issue is that the POD instructions might be re-ordered by the host processor. This will lead to correctness problem, because PE is ignorant of program order. To prevent this, POD instructions issued by the host are strongly ordered, similar to store instructions that are not re-ordered in most of the out-of-order implementations.

To address these issues, we propose an *IBits queue* which is inherently similar to the store queue in an out-of-order processor. When a SendBits instruction is issued, its 12-byte immediate field (encoding a VLIW POD instruction) is entered into the IBits queue. Upon the retirement of the SendBits instruction from the ROB, the corresponding 12-byte immediate value is latched onto the IBUS and broadcast to the POD.

With regard to multi-tasking support, as each PE's local SRAM is considered part of the architecture state, it needs to be saved and restored in-between context switches. It needs to be handled in the same way with other heterogeneous multicores such as the IBM Cell processor. This overhead on POD depends on several parameters including the size of the LLC, off-chip memory bandwidth, OS scheduling algorithm, etc. The Cell processor reports 20 μsec overhead for a context switching [2].

4.7 Physical Design Evaluations

In our implementation, we aim for a 3GHz clock speed assuming a 45nm or better process. For this target frequency, the memory ring is capable of up to 192GB/s bandwidth (servicing up to eight 24GB/s MCs before any modification is required). For an 8×8 POD array, with each PE containing 128KB of SRAM, connected in a torus, the peak performance of single-precision and double-precision IEEE FP operations is 1.5 TFLOPS and 768 GFLOPS, respectively.[3]

To estimate the overall die size, we use publicly accessible information (based on 65nm Intel Conroe processor) [28, 14]. First, we evaluate the size of each PE. Using Intel's Conroe die picture and floorplan, the integer, SIMD, and AGU pipeline occupies approximately 1.42, 1.36, and 0.14 mm^2, respectively, with a total of $2.92mm^2$ in size. The process scaling factor from 65nm to 45nm is 1.44 under perfect conditions, but we assume the scaling of these logic blocks is imperfect to an error of 50% for making conservative estimates. These same units will amount to approximately $2.1mm^2$ in 45nm.

For the area of each PE's local SRAM, according to Intel's published data [1], each SRAM cell at 45nm is approximately $0.346\mu m^2$. A single-ported 128KB SRAM will be roughly $0.363mm^2$. Since our basic 128KB PE SRAM contains 2 Read/Write ports, one Read port and one Write port, we estimate the entire SRAM to be

[2]Note that this is the case for all of our benchmark programs except k-means. The host processor does not issue any data-dependent instruction that reads data updated by the POD immediately for these benchmark programs. This event is extremely rare even in k-means simulation.

[3]Here are more analytical comparison between POD and IBM Cell. IBM's Cell has a theoretical SP floating-point capacity of 256 GFLOPS for 8 SPE units at 4GHz in a 90nm process [26]. For a fair comparison with our 8×8 POD, we assume that the SPE SRAM is halved to 128KB and has a perfect shrink with a 2x feature reduction for a 4x increase in number of SPEs, the maximum theoretical performance at 4GHz jumps to 1024 GFLOPs. However, again for fair comparison, the Cell speed should be reduced to our target 3GHz for a peak performance of 768 GFLOPs using all 8 SPEs. In contrast, we attain twice the performance at 1.5 TFLOPs, all while using a simpler PE design, clocking model, and programming model.

1.09mm^2. The areas of the two register files and one 32-entry RRQ with 75 bytes each compared to the local SRAM will be insignificant.

Based on these projections, one single PE will occupy around 3.2mm^2. In other words, the entire 8×8 SIMD PE array will amount to 205mm^2. Each RRQ, given the complexities of the various bus wirings and the ring interfaces, we allot an equal area on par with each PE. The total RRQ space is approximately 25.6mm^2. For the host processor, we simply scale the 36mm^2 of one single core in Conroe with the same scaling and fudge factors for 45nm process, the new core is approximately 25.9mm^2. The 3MB LLC is estimated 20mm^2 using the same 45nm SRAM data aforementioned. Therefore, the entire processor will amount to (205+25.6+25.9+20) = 276.5mm^2 without accounting for on-die integrated memory controllers.

5 ISA Support and Programming Model

5.1 ISA Support for Host Core

The SIMD execution inside the POD is completely managed by the host processor. To enable this, we propose extending the host core with five new instructions and modifying three others. Our new instructions are:

- *SendBits*, to broadcast instructions to the POD;
- *GetFlags*, to obtain the return status;
- *DrainFlags*, which assures that the initial setup of a known state in the flag tree is complete;
- *SendRegister*, to broadcast a host register value to every PE;
- *GetResult*, to obtain a return buffer value from the POD without using system memory as a go-between.

The three modified host instructions are the various *Fence* operations (Load, Store, and combined) that are extended to monitor the return status of the POD's memory interface system. Every other modification that our system requires is *external* to the host core and LLC.

5.2 ISA Support for POD PE

The instruction set of the PE supports typical integer ALU, memory, and SSE instructions. The integer and logical instructions operate on (32) 64-bit general-purpose registers while the SSE engine can address (32) 128-bit xmm registers. There are a variety of memory operations supported in the PE including regular load/store instructions, strided or contiguous block move instructions, and conditional-move instructions. Several versions of memory instructions are provided to allow data accesses from/to local SRAM, remote SRAM on another PE, and system memory. Details on the instruction set of POD can be found in Section E.

To allow multi-level conditional execution in the PE (nested if-then-else's and while-loops), we provide two types of masking instructions — pushmask and popmask. Inside each PE, there is a 64-bit mask register that the mask instruction modifies to keep track of the nested conditional state. The MSB of the mask register indicates the masking (on or off) for the current level of a nested control — this allows a PE to selectively turn itself on or off during the broadcast of instructions from the host. Conditions are determined by flag values which are generated by separate compare operations and their EFLAG results. By turning on and off PEs, it is possible to make only some of the cores execute a certain instruction. When entering a new conditional region, pushmask shifts down all the bits in the mask register for each PE and sets the MSB of the mask register to the new test condition. Whenever leaving a conditional region, the popmask instruction pops one bit out of the mask register and restores the prior state by shifting up. This provides up to a 63-levels of if-then-else or general conditional clauses. If necessary, the programmer or compiler can push or pop the mask register to the system memory or private SRAM to enable higher levels of nesting.

```
$ASM sub8sx r2 = r2, r2
for (int i=0; i<npeX; i++) {
  $ASM add8sx r2 = r2, r1
  $ASM xfer.e r1 = r1
}
$ASM sub8sx r1 = r1, r1
$ASM add8sx r1 = r1, r2
for (int i=0; i<npeY; i++) {
  $ASM xfer.n r1 = r1
  $ASM add8sx r2 = r2, r1
}
```

Figure 4: Code Example for POD (Reduction)

5.3 Mixed Instruction Stream

An example of the basic programming model for our POD prototype is shown in Figure 4. Lacking a comprehensive compiler for this architecture, we use pseudo-C code consisting of conventional C code for the host core and annotated inline POD assembly for the SIMD PE array.

Our current POD compiler (implemented with a pre-processing script and runtime library), captures this directive and generates the corresponding *SendBits* instructions as described in Section 5.1. The host processor is responsible for decoding normal CISC instructions. Once it detects a *SendBits* instruction, the following 96bits comprising our RISC-style VLIW instruction packet will be forwarded to the unit that is responsible for IA-POD instruction broadcast to the SIMD PE array. Programmers or compilers are required to explicitly generate the code for the PE array. An example code can be found in Section D.

Since the latency of all non-system-memory instructions, inter-core communication, and communication between the host and PE are all determined statically, this programming model is generally free from unrepeatable behavior, difficult debugging, locking, etc. The only unpredictable communication latency is the latency to or from system memory. When references are made to system memory, the host must issue a barrier instruction (one of the host's modified fence operations) before PEs can access the results.

5.4 Inter-PE Communication

Communication between PEs are explicitly specified by the programmer or compiler as shown in Figure 4. While this requires more up-front algorithmic work than an SMP model, it makes the resulting code much easier to debug. Additionally, since the latency of inter-PE communications is so low, Amdahl's law effects are much less prevalent than they are in longer-latency cache-based designs.

6 Experimental Results

6.1 Simulation Framework

A cycle-level POD simulator was developed to carry out our performance study. The simulator can sustain approximately 30 KIPS simulation throughput on a 3.4GHz Intel Xeon workstation. When simulating a full 8×8 POD, our effective simulation rate is approximately 2 MIPS on the same workstation. The feature of not having a complicated instruction fetch/decode mechanism, as well as the lack of control flow, branch prediction, and cache effects on the POD enables us to attain such high simulation speed.

Our compiler and simulator are tightly coupled — the compiler takes the application code and generates a native Intel binary. The x86 instructions of the host processor are natively executed on the Xeon workstation while the POD instructions are translated by a script, passed through a dependency checker, and simulated.

This library models every single feature of the PEs and memory subsystem, including an accurate modeling of on-chip and off-chip communication bandwidth. Yet there are certain limitations in our simulation framework. First, we did not measure the overheads incurred by the host processor such as Icache misses, branch mispredictions, TLB misses, etc. as they do not affect our results significantly. In general, the scalar code running on the x86 host processor should have negligible overheads to our target applications that exploit large data-parallelism on the PE array. Second, we did not model the LLC takeover of the MC ring for the host processor to access system memory, nor did we model the xTLB faults from address translations. Given these activities are very rare with the workloads and datasets we used, they should dramatically change our results. Details on PODSIM can be found in Section B.

6.2 Performance Evaluation

To evaluate the performance of POD architecture, we ported several data-parallel benchmark programs (Table 1) using inline assembly. Table 2 shows achieved GFLOPS[4] and relative performance improvement normalized to the performance result of 1×1 POD as the number of PEs increases. (Full simulation results can be found in Section C.) In this simulation, we aggressively model off-chip DRAM bandwidth as 4×32 GBps (four on-chip memory controllers where each can provide 32 GBps bandwidth.)[5] and DRAM latency as 50 ns. To factor out performance improvement due to larger on-chip memory as the number of PEs increases, we assume that aggregate size of the on-chip memory remains the same regardless of the number of PEs. For example, in our simulations, a PE of 1×1 POD has 8MB of local SRAM, while each PE of 8×8 POD has 128KB SRAM only. Our goal is to implement an 8×8 POD, and we conservatively assume that the access latency of 8MB SRAM is equivalent to that of a 128KB SRAM, which is 3 cycles for load or 1 cycle for store. Note that, in reality, the access time of an 8MB SRAM of our baseline, a 1×1 POD, will be slower, which will further boost the speedup of our results.

The first application, DenseMMM, which represents the main computation kernel in many linear system problems, shows very good scalability, although it requires a lot of communication between the neighboring PEs. This is because, at each computation stage, DenseMMM only requires one-hop communication, a much cheaper operation (2 cycles) on POD than on a MIMD-CMP. Moreover, this communication can be easily overlapped with the computation. DenseMMM is very computation-intensive, and it achieves overall 870.8 GFLOPS on 64 PEs.[6] The reason why the performance does not show an ideal linear speedup is that the efficiency of each PE goes down, although not severely, as the working set of each PE becomes smaller when we increase the number of PEs.

The second application, FFT, is a highly communication-intensive program. To demonstrate how effectively

[4]We count each add, sub, mul, div, max, min and cmp as one floating point operation, and fma (multiply and add) as two.

[5]Cell BE's on-chip memory controller supports 25.6 GBps of off-chip memory bandwidth.

[6]In fact, it achieves 1.06 TFLOPS of IEEE single precision floating point operations during main computation. The 870.8 GFLOPS result took the overhead of loading input and writing-back output into account.

Name	Description
DenseMMM	512×512 Dense Matrix-Matrix Multiplication (based on Cannon's algorithm [21])
FFT	1024-point 1D complex number Fast Fourier Transform
IDCT	IEEE 1180 8×8 Inverse Discrete Cosine Transform used in MPEG2 decoder of Media-Bench [20]
OptionPricing	A financial application that computes the risk of a portfolio by projecting future option prices
DownSampling	2:1 down-sampling over a 2112×2112 image
K-means	A mean-based data clustering application of Minebench [23] (Default input data set, 17695 data points in 18 dimensional space, is used.)

Table 1: Benchmark

Name	# of PEs	GFLOPS	Speedup
DenseMMM	1	16.3	1.0
	4	64.4	3.9
	16	248.4	15.1
	64	870.8	52.3
FFT	1	3.2	1.0
	4	16.3	5.1
	16	49.6	15.4
	64	113.6	35.3
IDCT[†]	1	3.9	1.0
	4	14.9	3.9
	16	51.8	13.4
	64	134.8	34.9
OptionPricing	1	13.4	1.0
	4	53.8	4.0
	16	215.1	16.0
	64	860.2	64.0
DownSampling	1	1.9	1.0
	4	7.4	4.0
	16	29.1	15.6
	64	97.2	52.1
K-means	1	8.3	1.0
	4	33.1	4.0
	16	131.6	15.8
	64	504.6	59.1

[†] Double precision floating point operations

Table 2: Performance Improvement

PEs exchange data, we choose a small input size (1024 points) so that the communication overhead cannot be hidden by the computation. Clearly, as the number of PEs increases, communication overhead becomes dominant, but we can still achieve very good performance improvement due to our high efficiency communication architecture. Another reason of the sub-linear speedup is that at each phase of the computation in our current FFT implementation, we only use one half of the PE array due to the nature of the algorithm. This inefficiency becomes more dominant as the number of PEs increases. The third reason is that overhead of loading input values becomes more outstanding as the main computation time decreases due to an increased number of PEs used. Note that the single chip performance improvement of our POD (64 PEs) is 35 times while that of MIMD-CMP with eight full-blown out-of-order cores will be at most 8 times. Figure 5 shows the active time of inter-PE point-to-point links with respect to the overall execution time for PODs with difference sizes. Only those applications that use inter-PE communication in our simulations are shown in the figure. As shown, although FFT is a well-known communication-intensive application, synchronized computation and communication model of POD makes it possible to disable its point-to-point links for more than 95% of the total execution time, thus minimizing the energy consumption of the communication links. Note that our MIMD-based many-core counterpart will not be able to disable their routers and wires due to the unpredictable nature of their interconnection.

The third application is IEEE 1180 8×8 2D IDCT, which is used by the MPEG2 decoder. According to our profiling result, more than 80% of the total execution time of the MPEG2 decoder are spent inside the IDCT block. As shown in Table 2, POD with 64 PEs can improve IDCT performance only by around 35 times even though this application does not have inter-PE communication and can fully utilize the PE array. This sub-linear speedup is caused by the fact that the working set of this application is very small, so time to broadcast input from the host

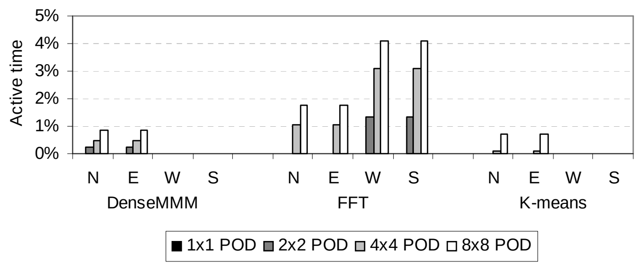


Figure 5: Point-to-point Links Active Time

processor to the PE array becomes dominant. Again, 35 times speedup with 64 PEs is still much bigger than eight times speedup that MIMD-CMP can achieve at best.

The fourth application, OptionPricing, is a computation-intensive application which shows very good data-level parallelism, and does not require any inter-PE communication. Furthermore, its computation is very heavy compared to system memory load overhead, thus its performance can be improved very well on POD as shown in Table 2, and it achieves 860.2 GFLOPS with 64 PEs.

In contrast, the performance of DownSampling does not scale well, in spite of its very good data-level parallelism and no inter-PE communication. Although it computes approximately 7 million 1×7 convolutions, this application becomes memory-intensive as the number of PEs increases. To quantify the effect of memory bandwidth, we also performed sensitivity study with different off-chip memory bandwidth. Although we performed this sensitivity study for all benchmark applications, here we only show the results of three applications, which are sensitive to the off-chip memory bandwidth. As shown in Figure 6, especially in DownSampling, we might not be able to efficiently utilize all 64 PEs, if the off-chip memory bandwidth is not high enough. Clearly, off-chip memory bandwidth is one of the biggest problems that we need to solve in future many-core architectures.

The last application, K-means, is arguably the most commonly used clustering algorithm in data mining [27]. This application is again very computation-intensive, while it requires large-scale global reduction to synchronize the computation results at the end of each computation phase. However, this communication is not significant compared to its heavy computation, thus it shows near-linear speedup as shown in Table 2, and achieves 504.6 GFLOPS with 64 PEs. The active time of point-to-point links is found to be less than 1% of the overall execution time as shown in Figure 5.

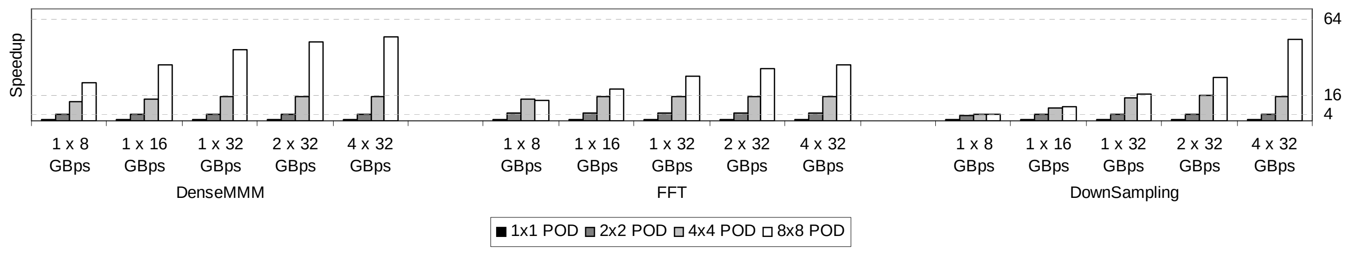


Figure 6: Memory Bandwidth Sensitivity

7 Conclusions

In this paper, we re-evaluate the SIMD computation paradigm in a new many-core architecture called Parallel-On-Die (POD) which integrates a sea of SIMD PE array into a host processor with minimally new instruction support to enable highly parallel processing. Our POD architecture fills a vital gap between the very general MIMD-style CMPs that work well on transactions and multi-programming workloads and the highly specialized processors used for media and graphics-oriented workloads. The SIMD designs also have the advantage that they are substantially good fits for implementing highly parallel versions of CISC instruction sets without having to pay the CISC penalty on every processing element. In other words, as one scales a SIMD array to larger sizes, the inefficiencies of the base architecture will be largely hidden, thus making the designs both compatible with existing ISAs and power/performance competitive with more specialized engines.

With the POD-style implementation, it becomes feasible to have both best-in-class scalar performance and extremely efficient scalable parallel performance on a single-die processor with minimally modified instruction set. As shown in our experimental results, single-chip performance of the POD is much higher than that of its MIMD counterpart for several applications ported onto our POD architecture, and POD can efficiently suppress energy consumption on its interconnection. As the industry moves toward the era of 10 billion transistor single-chip processor, the POD architecture will provide a highly scalable, energy/area efficient, and complexity-effective solution.

References

[1] Intel corporation, http://www.intel.com/technology/silicon/new_45nm_silicon.htm.

[2] Meet the Experts: Alex Chow on Cell Broadband Engine programming models, http://www-128.ibm.com/developerworks/power/library/pa-expert8/.

[3] Massively Parallel Digital Video. Microprocessor Report, January 2006.

[4] J. H. Ahn, W. J. Dally, B. Khailany, U. Kapasi, and A. Das. Evaluating the Imagine System Architecture. In *Proc. of the Int'l Symp. on Computer Architecture*, 2004.

[5] T. Blank. The MasPar MP-1 Architecture. In *Proceedings of COMPCON*, Spring 1990.

[6] S. Borkar. Networks for Multi-core Chip–A Controversial View. In *2006 Workshop on On- and Off-Chip Interconnection Networks for Multicore Systems*, 2006.

[7] S. Borkar, R. Cohn, G. Cox, S. Gleason, T. Gross, H. T. Kung, M. Lam, B. Moore, C. Peterson, J. Pieper, L. Rankin, P. S. Tseng, J. Sutton, J. Urbanski, and J. Webb. iWarp: An integrated solution to high-speed parallel computing. In *Supercomputing '88*, pages 330–339, 1988.

[8] W. J. Bouknight, S. A. Denenberg, D. F. McIntyre, J. M. Randall, A. H. Sameh, and D. L. Slotnick. The Illiac IV System. In *Proceedings of IEEE*, April 1972.

[9] A. A. Chien and J. H. Kim. Planar-adaptive routing: Low-Cost adaptive networks for multiprocessors. *Journal of the ACM*, 42(1):91–123, 1995.

[10] A. Duller, G. Panesar, and D. Towner. Parallel processing-the picochip way. *Communicating Processing Architectures*, pages 125–138, 2003.

[11] R. Espasa, F. Ardanaz, J. Emer, S. Felix, J. Gago, R. Gramunt, I. Hernandez, T. Juan, G. Lowney, M. Mattina, and S. A. Tarantula: a vector extension to the alpha architecture. In *Proc. of the Int'l Symp. on Computer Architecture*, 2002.

[12] M. D. Grammatikakis, D. F. Hsu, M. Kraetzl, and J. F. Sibeyn. Packet routing in fixed-connection networks: A survey. *Journal of Parallel and Distributed Computing*, 54(2):77–132, 1998.

[13] H. P. Hofstee. Power Efficient Processor Architecture and The Cell Processor. In *Proc. of the Int'l Symp. on High Performance Computer Architecture*, 2005.

[14] http://www.sandpile.org.

[15] U. J. Kapasi, W. J. Dally, S. Rixner, J. D. Owens, and B. Khailany. The Imagine Stream Processor. In *Proceedings of the International Conference on Computer Design*, 2002.

[16] J. S. Kim, M. B. Taylor, J. Miller, and D. Wentzlaff. Energy Characterization of a Tiled Architecture Processor with On-Chip Networks. In *Proceedings of the 8th International Symposium on Low Power Electronics and Design*, 2003.

[17] M. Kumar, Y. Baransky, and M. Denneau. The GF11 Parallel Computer. *Parallel Computing*, 19(12):1393–1412, 1993.

[18] R. Kumar, V. Zyuban, and D. M. Tullsen. Interconnections in Multi-core Architectures: Understanding Mechanisms, Overheads and Scaling. In *Proc. of the Int'l Symp. on Computer Architecture*, 2005.

[19] H. T. Kung. Why Systolic Architectures. *IEEE Computer*, 15(1):37–46, 1982.

[20] C. Lee, M. Potkonjak1, and W. H. Mangione-Smith. MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communications Systems. In *Proceedings of the International Symposium on Microarchitecture*, 1997.

[21] F. T. Leighton. *Introduction to parallel algorithms and architectures : arrays, trees, hypercubes*. Morgan Kaufmann, 1992.

[22] MasPar. Maspar programming language (ansi c compatible mpl) reference manual.

[23] R. Narayanan, B. Ozisikyilmaz, J. Zambreno, J. Pisharath, G. Memik, and A. Choudhary. MineBench: A Benchmark Suite for Data Mining Workloads. In *Proc. of the Int'l Symp. on Workload Characterization*, 2006.

[24] S. Palacharla, N. P. Jouppi, and J. E. Smith. Complexity-effective superscalar processors. In *ISCA*, pages 206–218, 1997.

[25] B. Parhami. SIMD Machines: Do They Have a Significant Future? In *Proceedings of SIGARCH*, 1995.

[26] D. Pham, S. Asano, M. Bolliger, M. N. Day, H. P. Hofstee, C. Johns, J. Kahle, A. Kameyama, J. Keaty, Y. Masubuchi, M. Riley, D. Shippy, D. Stasiak, M. Suzuoki, M. Wang, J. Warnock, S. Weitzel, D. Wendel, T. Yamazaki, and K. Yazawa. The Design and Implementation of a First-Generation CELL Processor. In *Proceedings of the 2005 IEEE International Solid-State Circuits Conference*, 2005.

[27] J. Pisharath, Y. Liu, W. keng Liao, G. Memik, and A. Choudhary. NU-MineBench: Understanding the Performance and Scalability Characteristics of Data Mining Algorithms. Technical Report CUCIS-2004-05-001, Center for Ultra-Scale Computing and Information Security, Northwestern Univ., 2004.

[28] K. Puttaswamy and G. H. Loh. Thermal Herding: Microarchitecture Techniques for Controlling HotSpots in High-Performance 3D-Integrated Processors. In *Proc. of the Int'l Symp. on High Perf. Computer Architecture*, 2007.

[29] S. Rixner, W. J. Dally, U. Kapasi, B. Khailany, A. Lopez-Lagunas, P. Mattson, and J. D. Owens. A Bandwidth-Efficient Architecture for Media Processing. In *Proc. of the Int'l Symp. on Microarchitecture*, 1998.

[30] K. Sankaralingam, R. Nagarajan, H. Liu, C. Kim, J. Huh, D. Burger, S. W. Keckler, and C. R. Moore. Exploiting ILP, TLP, and DLP with the Polymorphous TRIPS Architecture. In *Proc. of the 30th Int'l Symp. on Computer Architecture*, 2003.

[31] H. Singh, M. Lee, G. Lu, F. Kurdahi, N. Bagherzadeh, and E. Chaves Filho. MorphoSys: an integrated reconfigurable system for data-paralleland computation-intensive applications. *IEEE Transactions on Computers*, 49(5):465–481, 2000.

[32] D. L. Slotnick, W. C. Borck, and R. C. McReynolds. The Solomon Computer. volume 22, pages 97–107, 1962.

[33] M. Taveniku, A. Ahlander, M. Jonsson, and B. Svensson. The VEGA Moderately Parallel MIMD, Moderately Parallel SIMD, Architecture for High Performance Array Signal Processing. In *International Parallel Processing Symposium*, 1998.

[34] M. Taylor, S. Amarasinghe, and A. Agarwal. Scalar Operand Networks. In *IEEE Transactions on Parallel and Distributed Systems*, 2005.

[35] M. B. Taylor, J. Kim, J. Miller, D. Wentzlaff, F. Ghodrat, B. Greenwald, H. Hoffmann, P. Johnson, J.-W. Lee, W. Lee, A. Ma, A. Saraf, M. Seneski, N. Shnidman, V. Strumpen, M. Frank, S. Amarasinghe, and A. Agarwal. The Raw Microprocessor: A Computational Fabric for Software Circuits and General Purpose Programs. *IEEE Micro*, Mar/Apr, 2002.

[36] L. W. Tucker and G. G. Robertson. Architecture and Applications of the Connection Machine. In *IEEE Computer*, August 1988.

A Permutation Routing

The interconnection network of POD is designed to make frequently used communication, i.e., the neighbor-to-neighbor communication, fast to reduce the area and energy required. It is thus unoptimized for rarely used communication, which in turn is performed by using the *permutation routing* algorithm briefly explained in Section 4.3. In this appendix, we elaborate the permutation routing algorithm of POD in further details. Although it is assumed that PEs are connected with a 2D torus, their layout is shown in a normal order to simplify our explanation. Note that, however, they are laid-out as explained in Section 4.3.

The notation we used is shown in Figure 7. The numbers on the upper left corner of each PE is the ID of each PE. A small box on the upper right corner of each PE represents a communication message between 2 PEs and the two numbers inside the box represents an ID pair of the source and destination PE. The lower right side of each PE represents permutation queue shown in Figure 2, and the lower left side of each PE represents its local SRAM, where the transferred messages will eventually be stored. For example, PE15 in Figure 7 has received a store request from PE1 to PE15, has buffered one message from PE12 to PE3, and has already stored values from PE2 and PE3 to itself.

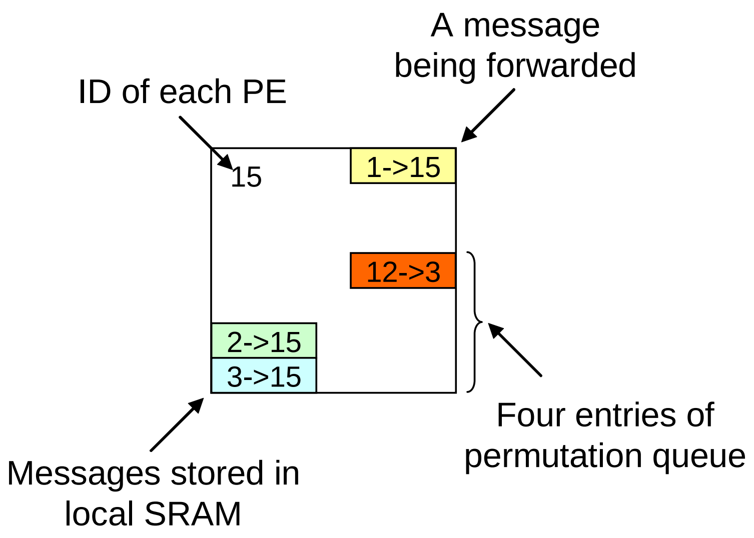

A message being forwarded
ID of each PE
15
1->15
12->3
2->15
3->15
Four entries of permutation queue
Messages stored in local SRAM

Figure 7: Notation for a PE

Figure 8(a) shows an example of a store operation among PEs. Initially, each PE starts to transfer a message to its own destination. In the first phase of the permutation routing algorithm, each PE sends their message toward the East, and each message will stop and be locally queued when it reaches its target column. Figure 8(b), Figure 8(c) and Figure 8(d) shows the location of messages at each stage of the first phase. For example, a message from PE1 to PE15 is transferred to PE2 at the first hop (Figure 8(b)), to PE3 (Figure 8(c)) at the second hop, and stops at PE3 (Figure 8(d)) after the second hop. In contrast, a message from PE2 to PE15 is transfered to PE3 (Figure 8(b)) and stops at PE3 and is queued in its permutation queue (Figure 8(c)) after the first hop.

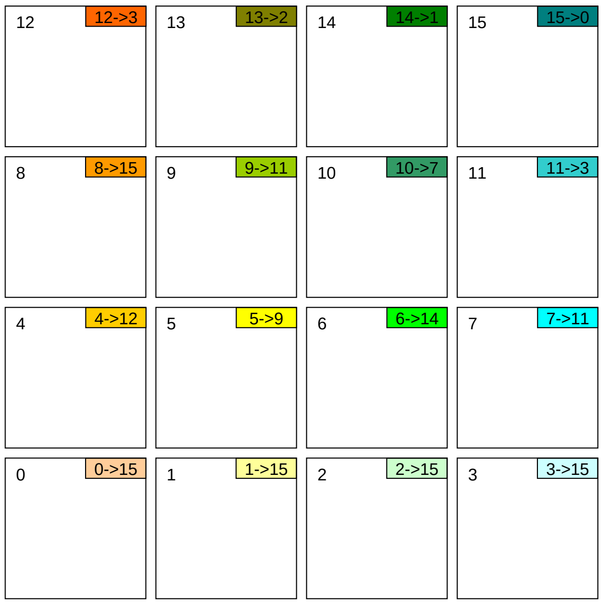
Note that only one point-to-point link (East link) is used during this first phase. Due to this characteristic, each PE only needs to enable East link during this phase, and it only needs to decode one message at a time. To buffer these messages, each PE on $n \times n$ PE array needs to have n entries in its permutation routing queue. Although this approach might not be optimal with respect to latency, this approach makes it possible to save both space and energy which might be consumed on a mesh-based MIMD-based CMP. For example, each PE on the POD does not need to keep track of messages coming from all different directions, handle link contention, have large buffer spaces to store them temporarily, and handle overflow problems of this buffer.

The second phase of this algorithm is similar to its first phase. Instead of sending messages to the East, now each PE sends messages to the North. Because each message has been forwarded to the column which its destination belongs to, this message can be transferred to the destination after this successive forwards to the North.

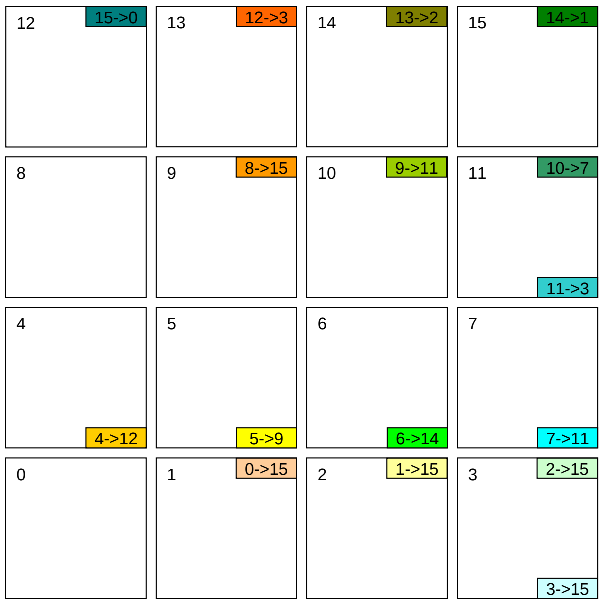
Figure 9 represents the communication pattern to forward messages buffered in the first slot of permutation queue. To forward these messages to the destination, it takes n steps of communication on $n \times n$ PE array as shown in Figure 9. Forwarding messages in the second, third and fourth slots of permutation queue takes another n steps respectively as shown in Figure 10, Figure 11, and Figure 12.

The reason why messages in different slots of the permutation queue are forwarded separately even though communication link often becomes idle is to avoid link contention. For example, PE3 can forward a message (PE2 to PE15) to PE7 at stage North3 (Figure 9(c)), because its communication link to PE7 is idle. If PE3 forwards it, it will eventually be forwarded PE11 and PE15 at stage North4 (Figure 9(d)) and North5 (Figure 10(a)). However, PE11 also wants to forward another message (PE10 to PE7) to PE11 and PE15 simultaneously as shown in Figure 9(d) and Figure 10(a). This means these two messages need to contend with each other to grab the communication link, and one of them needs to be stored somewhere. This will require another complicated control logic, which we want to avoid to make each PE small. The bigger a PE is, the longer neighbor-to-neighbor communication latency will become. Eventually, this will penalize the latency of our much more common communication.

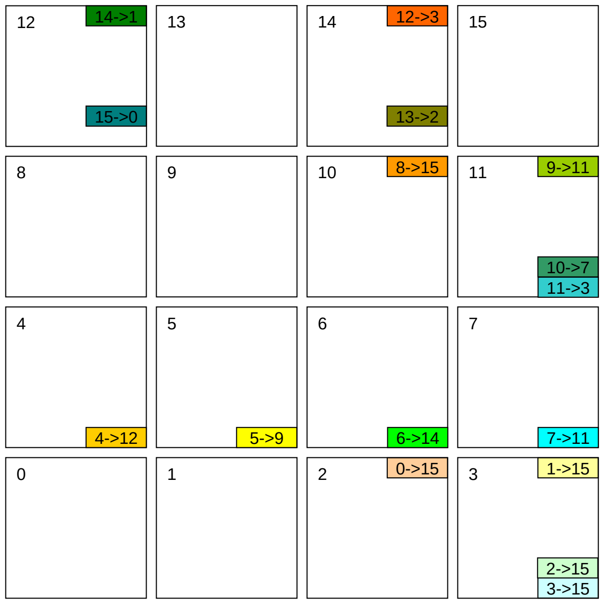
Note that this two-phase sweeping algorithm ensures that any permutation of routing, even all-to-one, is guaranteed to be finished after the same amount of a fixed latency. Although this might not be an optimal, this is highly space/energy efficient. More optimized row-only and column-only communication ($n - 1$ steps) for communication among PEs within same row or same column are possible to reduce the high latency of a full any-to-any communication.



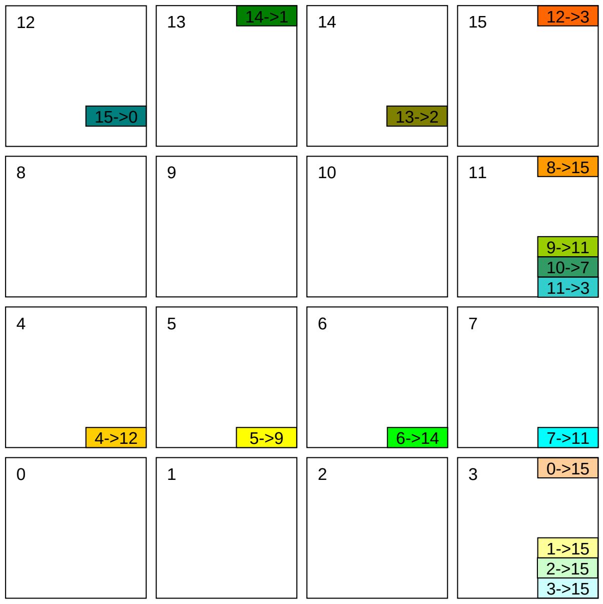
(a) Initial Request



(b) East1

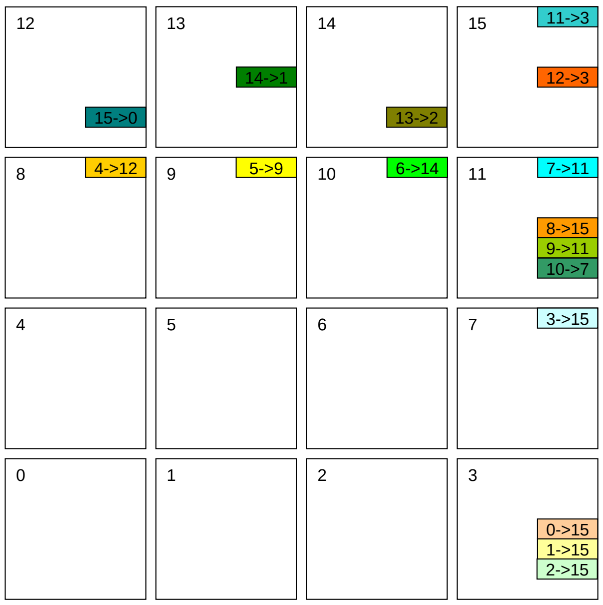


(c) East2

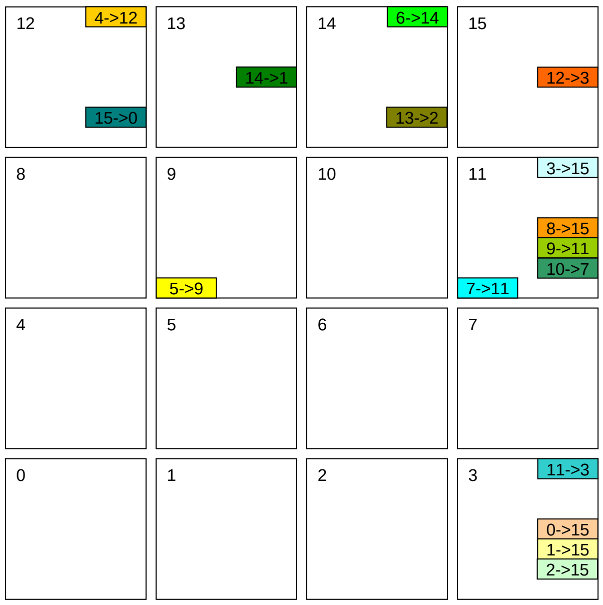


(d) East3

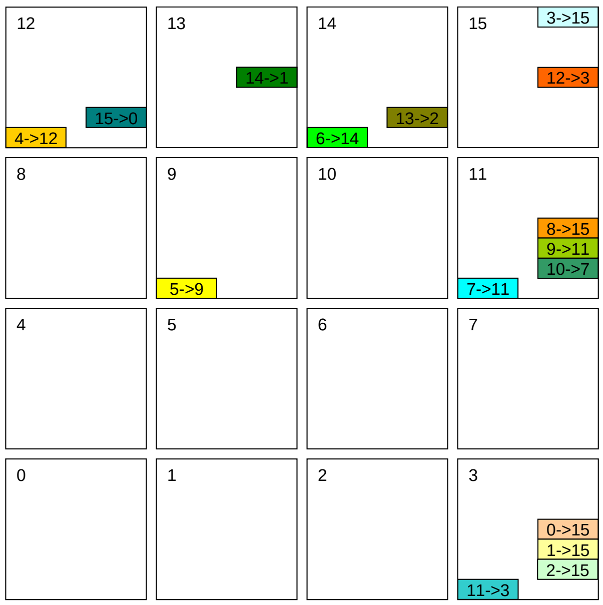
Figure 8: The First Phase



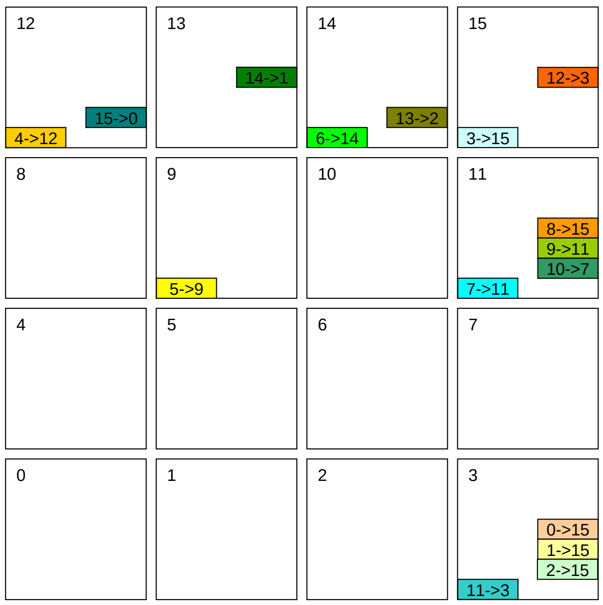
(a) North1



(b) North2

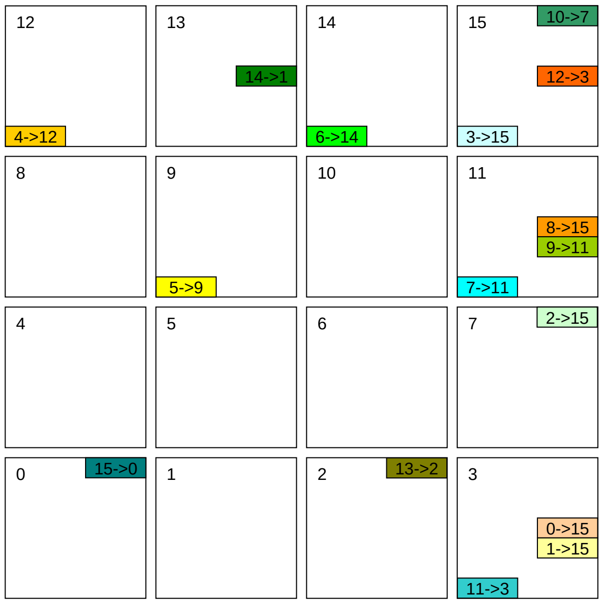


(c) North3

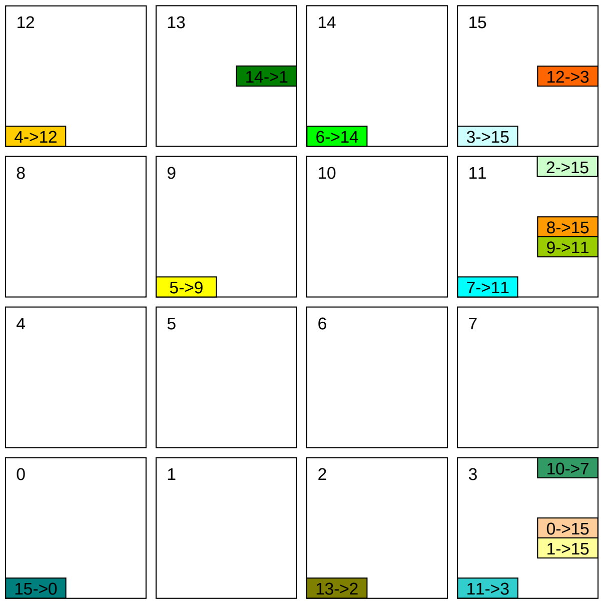


(d) North4

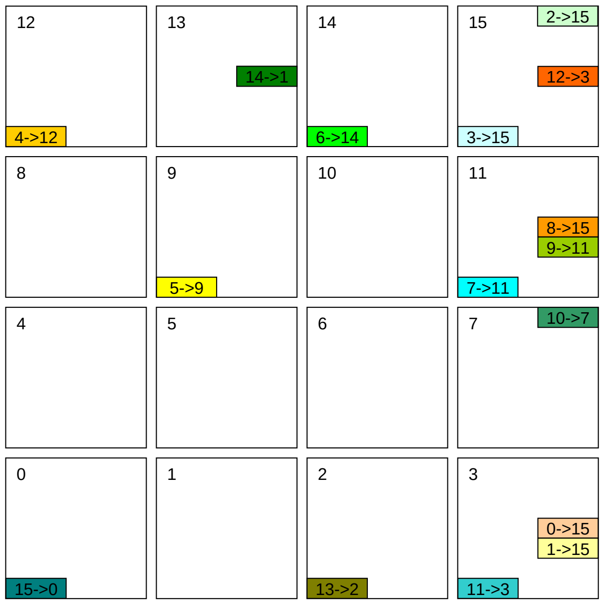
Figure 9: The Second Phase — for Permutation Queue Slot 0



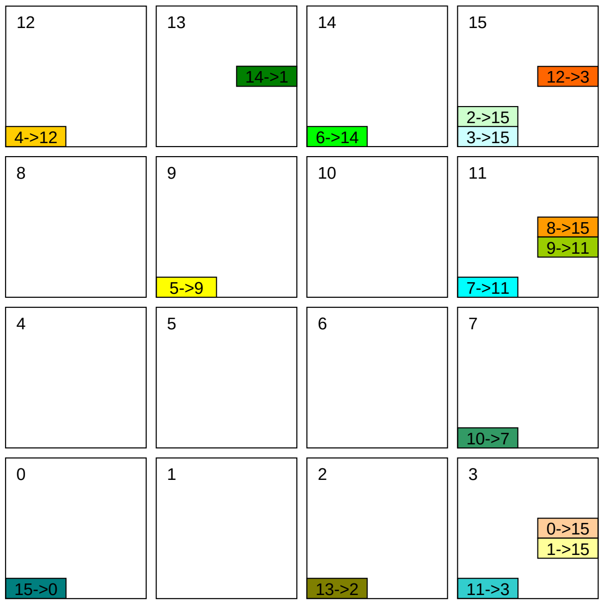
(a) North5



(b) North6

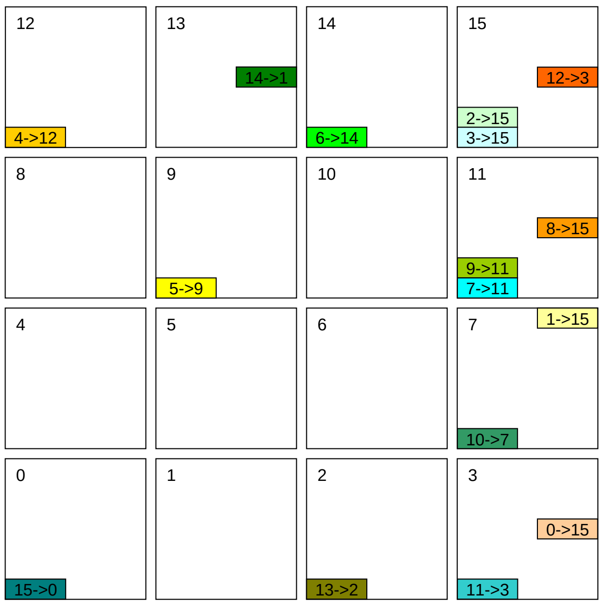


(c) North7

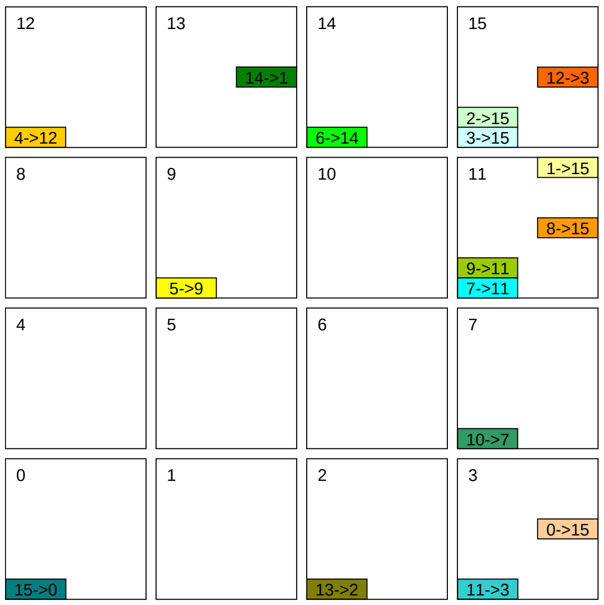


(d) North8

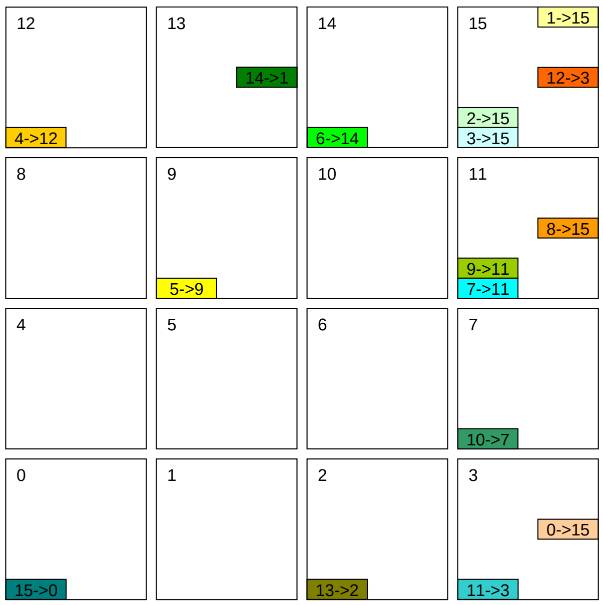
Figure 10: The Second Phase — Permutation Queue Slot 1



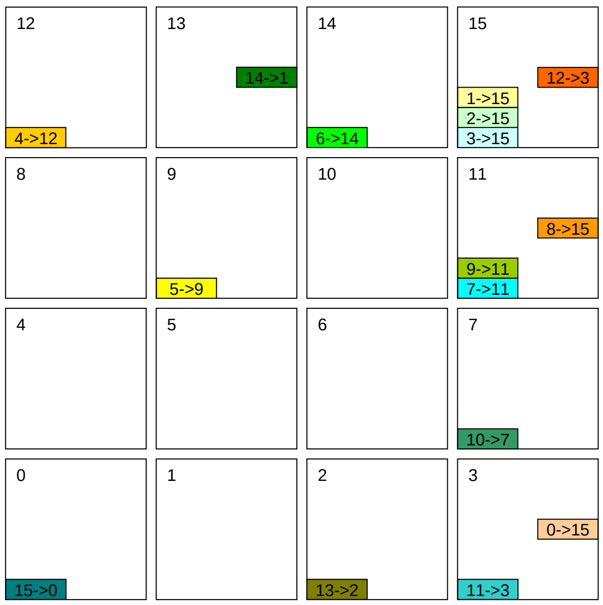
(a) North9



(b) North10

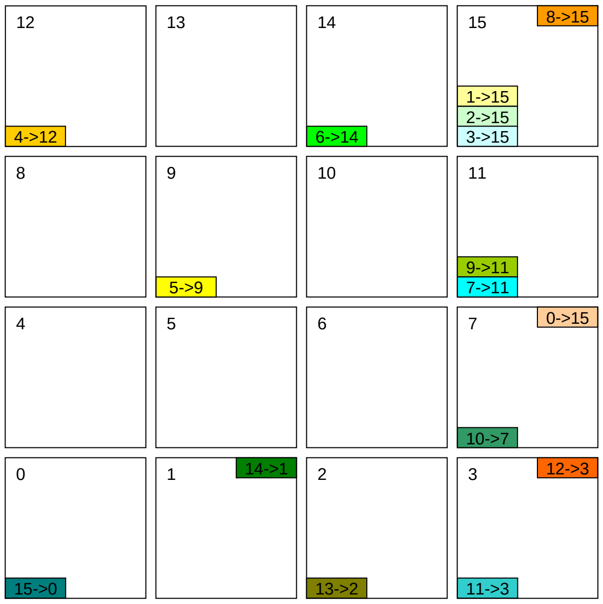


(c) North11

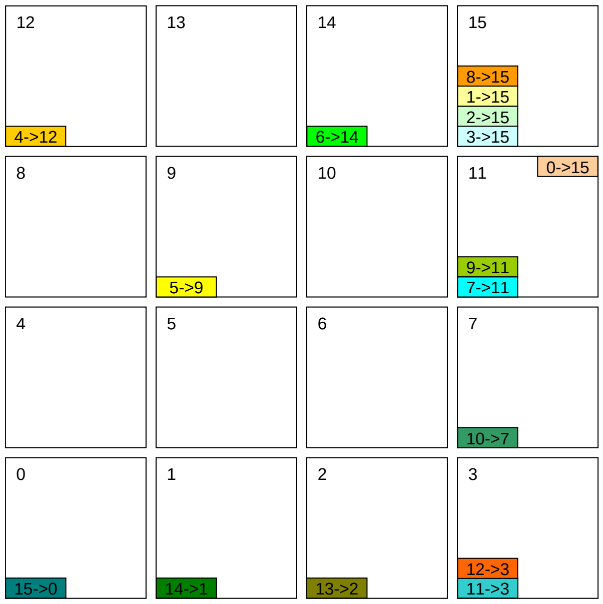


(d) North12

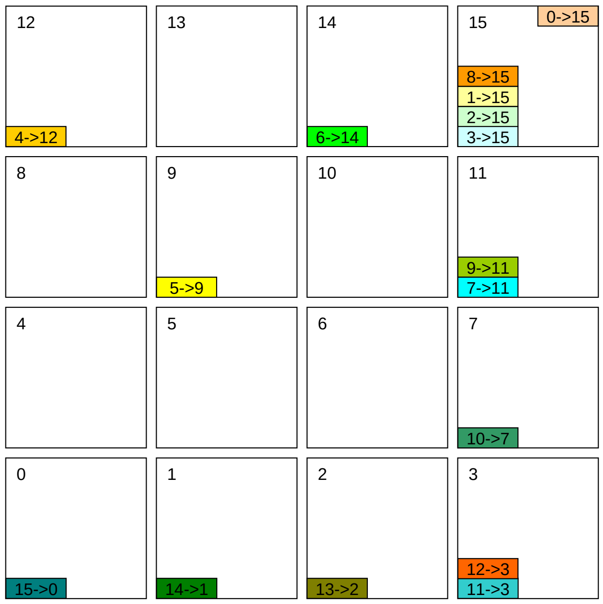
Figure 11: The Second Phase — Permutation Queue Slot 2



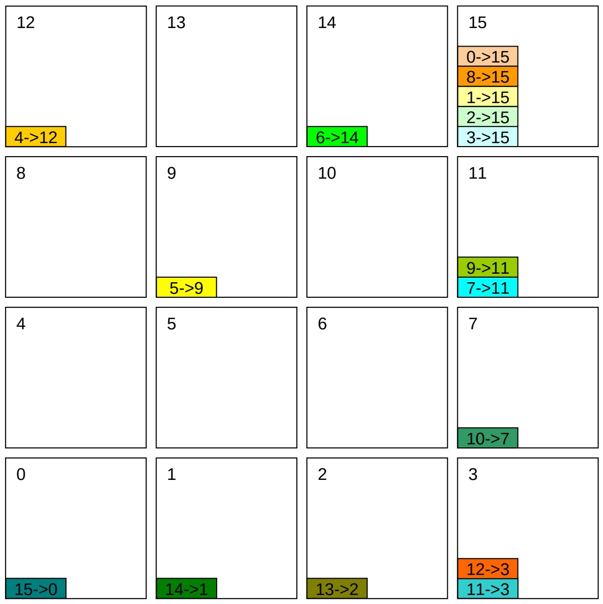
(a) North13



(b) North14



(c) North15



(d) Final result

Figure 12: The Second Phase — Permutation Queue Slot 3

B PODSIM

B.1 Objectives

Three major objectives were taken into consideration when PODSIM was initially designed. The first one was to provide a framework for cycle-level simulation. PODSIM simulates all behaviors of PEs' execution units, communication-relative logic, RRQs, and MCs. Furthermore, bandwidth of interconnections including iBus, inter-PE point-to-point links, Mbus, on-chip ring, and off-chip memory interconnection is modeled.

The second objective is to make it to be easily attached to an out-of-order host processor simulator. To make the porting easy, PODSIM provides an interface that mimics new and modified instructions of the host processor described in 5.1. All the host processor simulator required to do is to execute these instructions by calling these functions of PODSIM, which provides the execution latencies of these instructions.

The third objective is to make it as flexible as possible so that it can be easily configured for exploring the trade-off of design constraints, e.g. execution latency of instructions, the number of PEs, etc. Our current implementation provides this flexibility by using single unified header file that is used by all the simulator code. Every single feature that might be modified at the design-time is modeled as hash-defined variable, and the simulator can be easily reconfigured at the compile time.

B.2 IA-POD Translation

PODSIM is tightly coupled with its compiler — the compiler takes the application code and generates a native Intel binary. As shown in Table 3, PODSIM compiler translates assembly code written by a programmer into a function call, POD_sendibits, which drives PODSIM.[7] In addition to assembly translation, PODSIM compiler also bundles instructions into one VLIW instruction bundle, checks dependencies between VLIW instruction bundles, and generates warning messages if necessary.

B.3 Simulation Model and Limitation

PODSIM only simulates POD instructions, which run on the PEs. It does not simulate instructions executing on the host processor. For example, lines starting with the $ASM directive in Table 3 are POD instructions and are simulated by PODSIM. However, other normal C code will be executed by the host processor, which are not simulated by PODSIM. Instead, to drive our PODSIM, these instructions run on the native machine that simulates PODSIM. That means, instructions of the host processor are not simulated but executed to drive PODSIM (Figure 13). For example, translated code in Table 3 is the top-level code of PODSIM, and it runs on the native machine, but it performs simulation to calculate the latency of POD code whenever it executes POD_sendibits function.

Clearly, this simulation model has some limitations. First of all, current simulation model does not account for overheads incurred by the host processor. ICache misses of the host processor can make the host processor wait until the target instructions are ready. However, this inaccurate model is not likely affect our current simulation result much, because the instruction size of applications reported in this paper is not big enough to generate ICache misses. Branch mispredictions of the host processor is not modeled either. However, branches of the data-parallel applications that run on the host processors are usually loop-related branches or function calls, which are easily predicted. TLB misses are not modeled either, but this is not expected to hurt our simulation result because this is rare event.

Second, current PODSIM does not model the LLC takeover of the MC ring for the host processor to access system memory, because it does not have any host processor model. Given these activities are very rare with the workloads and datasets we used, they should not dramatically change our results.

[7]POD_sendibits takes three parameters, which are binary codes of three RISC-type instructions.

```
void euclid_dist_2_pod(int numdims) {
  int i;
  int loop_count;

  ...

  for ( i = 0; i < numdims; i+=4 ) {
    $ASM pfpsub.pack.sp pt0_dim = pt0_dim , cl_dim
    $ASM pfpsub.pack.sp pt1_dim = pt1_dim , cl_dim
    $ASM pfpsub.pack.sp pt2_dim = pt2_dim , cl_dim
    $ASM pfpsub.pack.sp pt3_dim = pt3_dim , cl_dim
    $ASM pfpsub.pack.sp pt4_dim = pt4_dim , cl_dim
    $ASM ldxmm++.pack cl_dim = local[ cl_ptr ], sixteen_gr

    ...

    $ASM pfpfma++.pack.sp distance4 += pt4_dim , pt4_dim
    $ASM ldxmm++.pack pt4_dim = local[ pt4_ptr ], sixteen_gr
  }

  ...

}
```

Before translation

```
void euclid_dist_2_pod(int numdims) {
  int i;
  int loop_count;

  ...

  for ( i = 0; i < numdims; i+=4 ) {
    POD_sendibits( 0x00000000, 0x21102102, 0x00000000 );
    POD_sendibits( 0x00000000, 0x21103182, 0x00000000 );
    POD_sendibits( 0x00000000, 0x21104202, 0x00000000 );
    POD_sendibits( 0x00000000, 0x21105282, 0x00000000 );
    POD_sendibits( 0x00000000, 0x21106302, 0x21101204 );


    ...

    POD_sendibits( 0x00000000, 0x1010b30c, 0x21106484 );

  }

  ...

}
```

After translation

Table 3: IA-POD Translation

B.4 Usage

Using PODSIM is quite simple. One just needs to write POD code starting with appmain() function[8], compile it using a provided shell script, and simulate it using a generated objective file.

gensim [npeX] [npeY] [nMC] [ifDebug] [file list]

npeX:	# of PEs in X dimension
npeY:	# of PEs in Y dimension
nMC:	# of Memory Controllers
ifDebug:	0 if non-debug mode, 1 if debug-mode

For example, to simulate FFT code on a 4×4 POD with 2 memory controllers in non-debugging mode, one needs to execute the following.

[8]instead of normal main() function

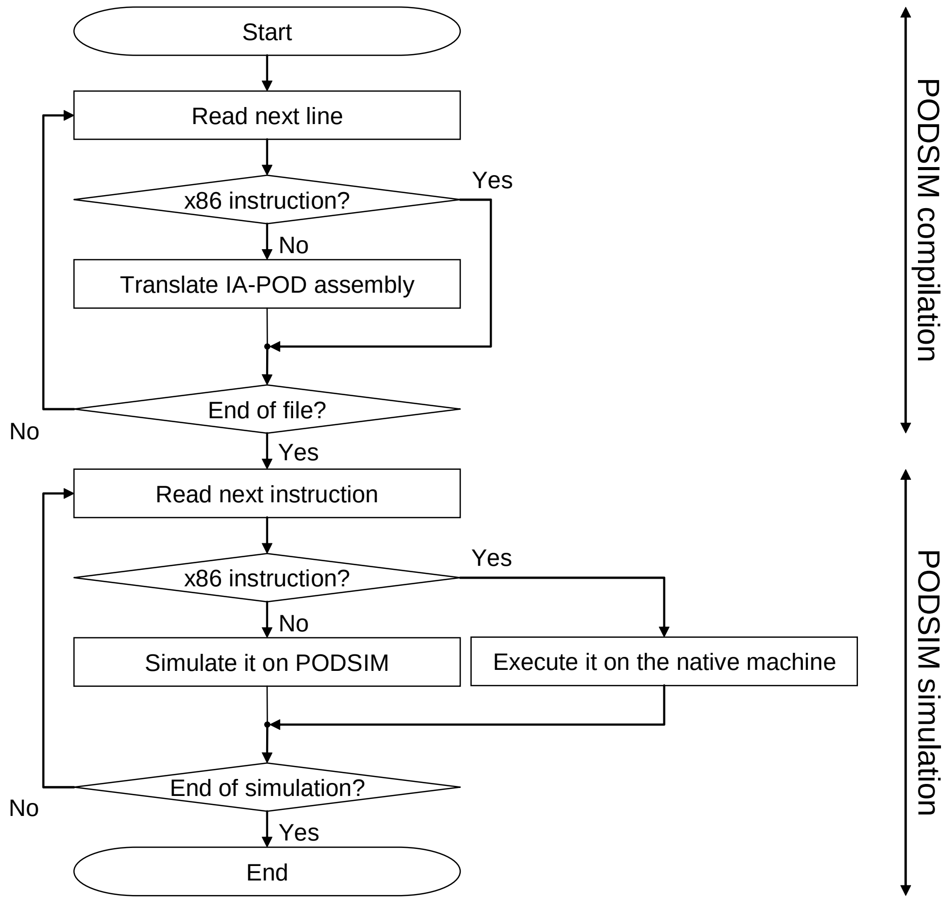


Figure 13: PODSIM Simulation Model

```
gensim 4 4 2 0 fft.c fft.h
```

To simulate the code, one just needs to run generated object code, called *podsim*.

B.5 Debugging Support

PODSIM also supports a debugging mode to help programmers debug their code easily. Easier debugging is another attractive feature of both POD itself and PODSIM. To run it in debugging mode, one needs to compile the code in debugging mode, and execute *podsim*.

PODSIM supports breakpointing, stepping through code, skipping code without breakpointing, and looking up register values and memory values. Details can be found once debugging-mode *podsim* is executed or in Table 4. By looking at register values or memory values, programmers can understand architectural status of all PEs at once.

B.6 Simulation Speed

Last, but not least feature of PODSIM is its FAST simulation! The simulator can sustain approximately 30 KIPS simulation throughput on a 3.4GHz Intel Xeon workstation. When simulating a full 8×8 POD, our effective simulation rate is approximately 2 MIPS on the same workstation. The feature of not having a complicated

h	To see this usage.
b<pc>	<B>reakpoint: To set a breakpoint (Currently, only one breakpoint is supported simultaneously.)
c	<C>ontinue: To execute one instruction bundle
g	<G>o: To execute instruction bundles until a breakpoint is met
s<cnt>	<S>kip: To skip<cnt>instruction bundles
v	<V>im: To vim the application code (tmpbuild/podsim.cpp)
gr<num>	<G>r: To read gr<num>s
x1<num>	<X>mm: To read xmm<num>s in char format
x2<num>	<X>mm: To read xmm<num>s in short format
x4<num>	<X>mm: To read xmm<num>s in int format
x8<num>	<X>mm: To read xmm<num>s in long long format
xs<num>	<X>mm: To read xmm<num>s in float format
xd<num>	<X>mm: To read xmm<num>s in double format
l1<addr>	<L>ocal SRAM: To read1—byte data from address<addr>of local SRAMs in char format
l2<addr>	<L>ocal SRAM: To read2—byte data from address<addr>of local SRAMs in short format
l4<addr>	<L>ocal SRAM: To read4—byte data from address<addr>of local SRAMs in int format
l8<addr>	<L>ocal SRAM: To read8—byte data from address<addr>of local SRAMs in long long format
ls<addr>	<L>ocal SRAM: To read4—byte data from address<addr>of local SRAMs in float format
ld<addr>	<L>ocal SRAM: To read8—byte data from address<addr>of local SRAMs in double format
s1<addr>	<S>ystem memory: To read1—byte data from address<addr>of the system memory in char format
s2<addr>	<S>ystem memory: To read2—byte data from address<addr>of the system memory in short format
s4<addr>	<S>ystem memory: To read4—byte data from address<addr>of the system memory in int format
s8<addr>	<S>ystem memory: To read8—byte data from address<addr>of the system memory in long long format
ss<addr>	<S>ystem memory: To read4—byte data from address<addr>of the system memory in float format
sd<addr>	<S>ystem memory: To read8—byte data from address<addr>of the system memory in double format
q	<Q>uit: To quit

Table 4: Debugging commands

instruction fetch/decode mechanism, as well as the lack of control flow, branch prediction, and cache effects on the POD enables us to attain such high simulation speed.

C Full Simulation Results

# of PEs	Off-chip memory bandwidth (GBps)	Achieved GFLOPS (single precision)	Normalized performance	North link active time (%)	East link active time (%)	West link active time (%)	South link active time (%)
1	1×8	16.3	1.0	0.1	0.1	0.0	0.0
	1×16	16.3	1.0	0.1	0.1	0.0	0.0
	1×32	16.3	1.0	0.1	0.1	0.0	0.0
	2×32	16.3	1.0	0.1	0.1	0.0	0.0
	4×32	16.3	1.0	0.0	0.0	0.0	0.0
4	1×8	63.1	3.9	0.3	0.3	0.0	0.0
	1×16	64.5	3.9	0.3	0.3	0.0	0.0
	1×32	64.5	3.9	0.3	0.3	0.0	0.0
	2×32	64.5	3.9	0.3	0.3	0.0	0.0
	4×32	64.4	3.9	0.3	0.3	0.0	0.0
16	1×8	191.8	11.6	0.4	0.4	0.0	0.0
	1×16	221.6	13.4	0.4	0.4	0.0	0.0
	1×32	246.3	14.9	0.5	0.5	0.0	0.0
	2×32	248.7	15.1	0.5	0.5	0.0	0.0
	4×32	248.4	15.1	0.5	0.5	0.0	0.0
64	1×8	403.0	24.2	0.4	0.4	0.0	0.0
	1×16	578.4	34.7	0.6	0.6	0.0	0.0
	1×32	741.4	44.5	0.7	0.7	0.0	0.0
	2×32	821.4	49.3	0.8	0.8	0.0	0.0
	4×32	870.8	52.3	0.9	0.9	0.0	0.0

Table 5: DenseMMM Simulation Result

# of PEs	Off-chip memory bandwidth (GBps)	Achieved GFLOPS (single precision)	Normalized performance	North link active time (%)	East link active time (%)	West link active time (%)	South link active time (%)
1	1×8	3.2	1.0	0.0	0.0	0.0	0.0
	1×16	3.2	1.0	0.0	0.0	0.0	0.0
	1×32	3.2	1.0	0.0	0.0	0.0	0.0
	2×32	3.2	1.0	0.0	0.0	0.0	0.0
	4×32	3.2	1.0	0.0	0.0	0.0	0.0
4	1×8	16.3	5.0	0.0	0.0	1.3	1.3
	1×16	16.3	5.0	0.0	0.0	1.3	1.3
	1×32	16.3	5.0	0.0	0.0	1.3	1.3
	2×32	16.3	5.0	0.0	0.0	1.3	1.3
	4×32	16.3	5.1	0.0	0.0	1.3	1.3
16	1×8	43.5	13.4	0.9	0.9	2.7	2.7
	1×16	48.1	14.8	1.0	1.0	3.0	3.0
	1×32	49.7	15.3	1.0	1.0	3.1	3.1
	2×32	49.6	15.3	1.0	1.0	3.1	3.1
	4×32	49.6	15.4	1.0	1.0	3.1	3.1
64	1×8	40.2	12.4	0.6	0.6	1.5	1.5
	1×16	65.1	20.1	1.0	1.0	2.4	2.4
	1×32	90.7	27.9	1.4	1.4	3.3	3.3
	2×32	104.5	32.3	1.6	1.6	3.8	3.8
	4×32	113.6	35.3	1.8	1.8	4.1	4.1

Table 6: FFT Simulation Result

# of PEs	Off-chip memory bandwidth (GBps)	Achieved GFLOPS (double precision)	Normalized performance	North link active time (%)	East link active time (%)	West link active time (%)	South link active time (%)
1	1×8	3.9	1.0	0.0	0.0	0.0	0.0
	1×16	3.9	1.0	0.0	0.0	0.0	0.0
	1×32	3.9	1.0	0.0	0.0	0.0	0.0
	2×32	3.9	1.0	0.0	0.0	0.0	0.0
	4×32	3.9	1.0	0.0	0.0	0.0	0.0
4	1×8	14.9	3.9	0.0	0.0	0.0	0.0
	1×16	14.9	3.9	0.0	0.0	0.0	0.0
	1×32	14.9	3.9	0.0	0.0	0.0	0.0
	2×32	14.9	3.9	0.0	0.0	0.0	0.0
	4×32	14.9	3.9	0.0	0.0	0.0	0.0
16	1×8	51.8	13.4	0.0	0.0	0.0	0.0
	1×16	51.8	13.4	0.0	0.0	0.0	0.0
	1×32	51.8	13.4	0.0	0.0	0.0	0.0
	2×32	51.8	13.4	0.0	0.0	0.0	0.0
	4×32	51.8	13.4	0.0	0.0	0.0	0.0
64	1×8	134.8	34.9	0.0	0.0	0.0	0.0
	1×16	134.8	34.9	0.0	0.0	0.0	0.0
	1×32	134.8	34.9	0.0	0.0	0.0	0.0
	2×32	134.8	34.9	0.0	0.0	0.0	0.0
	4×32	134.8	34.9	0.0	0.0	0.0	0.0

Table 7: IDCT Simulation Result

# of PEs	Off-chip memory bandwidth (GBps)	Achieved GFLOPS (single precision)	Normalized performance	North link active time (%)	East link active time (%)	West link active time (%)	South link active time (%)
1	1×8	13.4	1.0	0.0	0.0	0.0	0.0
	1×16	13.4	1.0	0.0	0.0	0.0	0.0
	1×32	13.4	1.0	0.0	0.0	0.0	0.0
	2×32	13.4	1.0	0.0	0.0	0.0	0.0
	4×32	13.4	1.0	0.0	0.0	0.0	0.0
4	1×8	53.8	4.0	0.0	0.0	0.0	0.0
	1×16	53.8	4.0	0.0	0.0	0.0	0.0
	1×32	53.8	4.0	0.0	0.0	0.0	0.0
	2×32	53.8	4.0	0.0	0.0	0.0	0.0
	4×32	53.8	4.0	0.0	0.0	0.0	0.0
16	1×8	214.9	16.0	0.0	0.0	0.0	0.0
	1×16	215.0	16.0	0.0	0.0	0.0	0.0
	1×32	215.1	16.0	0.0	0.0	0.0	0.0
	2×32	215.1	16.0	0.0	0.0	0.0	0.0
	4×32	215.1	16.0	0.0	0.0	0.0	0.0
64	1×8	857.1	63.7	0.0	0.0	0.0	0.0
	1×16	858.9	63.9	0.0	0.0	0.0	0.0
	1×32	859.7	63.9	0.0	0.0	0.0	0.0
	2×32	860.1	64.0	0.0	0.0	0.0	0.0
	4×32	860.2	64.0	0.0	0.0	0.0	0.0

Table 8: OptionPricing Simulation Result

# of PEs	Off-chip memory bandwidth (GBps)	Achieved GFLOPS (single precision)	Normalized performance	North link active time (%)	East link active time (%)	West link active time (%)	South link active time (%)
1	1×8	1.9	1.0	0.0	0.0	0.0	0.0
	1×16	1.9	1.0	0.0	0.0	0.0	0.0
	1×32	1.9	1.0	0.0	0.0	0.0	0.0
	2×32	1.9	1.0	0.0	0.0	0.0	0.0
	4×32	1.9	1.0	0.0	0.0	0.0	0.0
4	1×8	6.7	3.5	0.0	0.0	0.0	0.0
	1×16	7.5	4.0	0.0	0.0	0.0	0.0
	1×32	7.5	4.0	0.0	0.0	0.0	0.0
	2×32	7.5	4.0	0.0	0.0	0.0	0.0
	4×32	7.4	4.0	0.0	0.0	0.0	0.0
16	1×8	7.8	4.1	0.0	0.0	0.0	0.0
	1×16	14.4	7.6	0.0	0.0	0.0	0.0
	1×32	27.4	14.4	0.0	0.0	0.0	0.0
	2×32	29.4	15.6	0.0	0.0	0.0	0.0
	4×32	29.1	15.6	0.0	0.0	0.0	0.0
64	1×8	8.1	4.3	0.0	0.0	0.0	0.0
	1×16	16.2	8.5	0.0	0.0	0.0	0.0
	1×32	31.7	16.7	0.0	0.0	0.0	0.0
	2×32	50.3	26.7	0.0	0.0	0.0	0.0
	4×32	97.2	52.1	0.0	0.0	0.0	0.0

Table 9: DownSampling Simulation Result

# of PEs	Off-chip memory bandwidth (GBps)	Achieved GFLOPS (single precision)	Normalized performance	North link active time (%)	East link active time (%)	West link active time (%)	South link active time (%)
1	1×8	8.3	1.0	0.0	0.0	0.0	0.0
	1×16	8.3	1.0	0.0	0.0	0.0	0.0
	1×32	8.3	1.0	0.0	0.0	0.0	0.0
	2×32	8.3	1.0	0.0	0.0	0.0	0.0
	4×32	8.3	1.0	0.0	0.0	0.0	0.0
4	1×8	33.1	4.0	0.0	0.0	0.0	0.0
	1×16	33.1	4.0	0.0	0.0	0.0	0.0
	1×32	33.1	4.0	0.0	0.0	0.0	0.0
	2×32	33.1	4.0	0.0	0.0	0.0	0.0
	4×32	33.1	4.0	0.0	0.0	0.0	0.0
16	1×8	130.7	15.7	0.1	0.1	0.0	0.0
	1×16	131.5	15.8	0.1	0.1	0.0	0.0
	1×32	131.6	15.8	0.1	0.1	0.0	0.0
	2×32	131.6	15.8	0.1	0.1	0.0	0.0
	4×32	131.6	15.8	0.1	0.1	0.0	0.0
64	1×8	485.1	56.8	0.7	0.7	0.0	0.0
	1×16	495.8	58.0	0.7	0.7	0.0	0.0
	1×32	500.6	58.6	0.7	0.7	0.0	0.0
	2×32	504.2	59.0	0.7	0.7	0.0	0.0
	4×32	504.6	59.1	0.7	0.7	0.0	0.0

Table 10: K-means Simulation Result

D Example Code - K-means

```
#include "k-means.h"

#define MAX_FLOAT  0x7f7fffff

#ifndef FLT_MAX
#define FLT_MAX 3.40282347e+38
#endif

#define  CHECK  1

#define rdtsc(x)     __asm__ __volatile__ ("rdtsc" : "=A" (x))

void euclid_dist_2_pod(int numdims);
void find_nearest_point_pod(int nfeatures, int npts);
float** kmeans_clustering_pod(float **feature,
                          int     nfeatures,
                          int     originalnpoints,
                          int     npoints,
                          int     nclusters,
                          float   threshold,
                          int    *membership);
float euclid_dist_2(float *pt1,
                    float *pt2,
                    int    numdims);
int find_nearest_point(float  *pt,          /* [nfeatures] */
                       int     nfeatures,
                       float **pts,         /* [npts][nfeatures] */
                       int     npts);
float** kmeans_clustering(float **feature,
                          int     nfeatures,
                          int     npoints,
                          int     nclusters,
                          float   threshold,
                          int    *membership);
void readFromFile();

int select_initial_cluster(int nclusters, int i);
/*

1. allocate input data space in local SRAM
(# of points per PE) * (# of attributes) * 4

(attribute 0)(attribute 1)(attribute 2)...(attribute 17)(zero padding)(zero padding)
(attribute 0)(attribute 1)(attribute 2)...(attribute 17)(zero padding)(zero padding)
...
```

```
(attribute 0)(attribute 1)(attribute 2)...(attribute 17)(zero padding)(zero padding)

2. allocate local space for clusters
(# of clusters) * (# of attributes) * 4

(attribute 0)(attribute 1)(attribute 2)...(attribute 17)(zero padding)(zero padding)
(attribute 0)(attribute 1)(attribute 2)...(attribute 17)(zero padding)(zero padding)
...
(attribute 0)(attribute 1)(attribute 2)...(attribute 17)(zero padding)(zero padding)

3. allocate membership space in local SRAM
(# of points per PE) * 4

*/

float  *buf;
float **attributes;
float **attributesPOD;
int      numAttributes;
int      numObjects;
int localAttributesSize;

int objectGran;


void
appmain() {

  int effectiveNumObjects;
  int effectiveNumAttributes;
  int effectiveNumObjectsPerPE;
  int localClustersAttributesSize;
  float threshold = 0.001;
  int *membership_pod;
  int *membership;

  objectGran = npe*5;

  readFromFile();

  if ( ( numObjects % objectGran ) == 0 ) effectiveNumObjects = numObjects;
  else effectiveNumObjects = ( numObjects / objectGran ) * objectGran + objectGran;

  if ( (numAttributes % 20) == 0 ) effectiveNumAttributes = numAttributes;
  else effectiveNumAttributes = ( numAttributes / 20 ) * 20 + 20;

  effectiveNumObjectsPerPE = effectiveNumObjects / npe;
```

```
printf( "numObjects:     %d->%d\n", numObjects, effectiveNumObjects );
printf( "numAttributes: %d->%d\n", numAttributes, effectiveNumAttributes );

localAttributesSize = effectiveNumObjectsPerPE * effectiveNumAttributes * 4;

attributesPOD = (float**) malloc( effectiveNumObjects * sizeof(float*) );
attributesPOD[0] = (float*) malloc( effectiveNumObjects * effectiveNumAttributes * sizeof(float) );

membership_pod = (int*) malloc(effectiveNumObjects * sizeof(int));
membership = (int*) malloc(numObjects * sizeof(int));

      for (int i=1; i<effectiveNumObjects; i++)
          attributesPOD[i] = attributesPOD[i-1] + effectiveNumAttributes;

for ( int i = 0; i < effectiveNumObjects; i++ ) {
  memset( attributesPOD[i], 0, effectiveNumAttributes*sizeof(float) );
  if ( i < numObjects ) {
    memcpy( attributesPOD[i], attributes[i], numAttributes*sizeof(float) );
  }
}

$ASM sub4zx zero_gr = zero_gr, zero_gr
$ASM pintxor zero_xmm = zero_xmm, zero_xmm
$ASM add4zx four_gr = zero_gr, 4
$ASM add4zx sixteen_gr = zero_gr, 16
$ASM imul4 eighty_gr = sixteen_gr, 5

int alignmentIssue = effectiveNumObjectsPerPE * 4;
if ( ( alignmentIssue % 16 ) != 0 ) alignmentIssue = alignmentIssue / 16 * 16 + 16;

POD_movl( local_attributes_ptr, PODLIB_malloc( localAttributesSize ) );
POD_movl( local_membership_ptr, PODLIB_malloc( alignmentIssue ) );
POD_movl( local_index_ptr, PODLIB_malloc( alignmentIssue ) );
POD_movl( local_distance_ptr, PODLIB_malloc( alignmentIssue ) );


POD_movl( sys_attributes_ptr, (unsigned long long)  attributesPOD[0] );

$ASM movl npe_gr = PTR_NPE
$ASM movl my_pe = PTR_MY_PE
$ASM movl npex_gr = PTR_NPE_X
$ASM ld2.sxt npe_gr = local[ npe_gr + 0 ]
$ASM ld2.sxt my_pe = local[ my_pe + 0 ]
$ASM ld1.sxt npex_gr = local[ npex_gr + 0 ]

POD_movl( local_attributes_size_gr, localAttributesSize );
```

```
  $ASM sub4zx npe_minus_one_gr = npe_gr, 1
  $ASM sub4zx npex_minus_one_gr = npex_gr, 1
  $ASM imul4 tmp_gr = local_attributes_size_gr, my_pe
  $ASM nop.g
  $ASM add4zx sys_attributes_ptr = sys_attributes_ptr, tmp_gr
  $ASM nop.g
  $ASM copyblk local[ local_attributes_ptr ] = sys[ sys_attributes_ptr ], local_attributes_size_gr


  for ( int nclusters = /*min_nclusters*/ 2; nclusters <= /*max_nclusters*/ 10; nclusters++ ) {

    printf( "Clustering into %d clusters..\n", nclusters );

    kmeans_clustering_pod( attributesPOD, effectiveNumAttributes,
                           numObjects, effectiveNumObjects, nclusters, threshold, membership_pod );
    kmeans_clustering( attributes, numAttributes, numObjects, nclusters, threshold, membership );

    for ( int i = 0; i < numObjects; i++ ) {
      if ( membership[i] != membership_pod[i] )
        printf( "Different!!! %dth point (nclusters=%d): %d vs. %d\n",
                i, nclusters, membership[i], membership_pod[i] );
    }
  }

}
int select_initial_cluster(int nclusters, int i) {
  int random_initial_cluster[9][10] = {
    { 1450, 14699, 0, 0, 0, 0, 0, 0, 0, 0 },
    { 9626, 15691, 11941, 0, 0, 0, 0, 0, 0, 0 },
    { 10060, 7066, 10506, 13653, 0, 0, 0, 0, 0, 0 },
    { 15738, 16401, 6821, 15641, 479, 0, 0, 0, 0, 0 },
    { 6908, 275, 7087, 8508, 2693, 8400, 0, 0, 0, 0 },
    { 14662, 8792, 16509, 16062, 601, 15365, 15634, 0, 0, 0 },
    { 5243, 3629, 12618, 9511, 3338, 11411, 2394, 7131, 0, 0 },
    { 15739, 11396, 10158, 1394, 5169, 2135, 6704, 13119, 14140, 0 },
    { 8207, 3366, 2990, 9660, 3376, 9156, 13510, 17285, 17195, 11572 } };

  return random_initial_cluster[nclusters-2][i];
}
float** kmeans_clustering_pod(float **feature,    /* in: [npoints][nfeatures] */
                          int     nfeatures,
                          int     originalnpoints,
                          int     npoints,
                          int     nclusters,
                          float   threshold,
                          int    *membership) /* out: [npoints] */
```

```
{
  int       i, j, k, index, loop=0;
  int      *new_centers_len; /* [nclusters]: no. of points in each cluster */
  float     delta;
  float   **clusters;    /* out: [nclusters][nfeatures] */
  float   **new_centers;      /* [nclusters][nfeatures] */
  double    timing;

  int localClusterAttributesSize;
  localClusterAttributesSize = nclusters * nfeatures * 4;

  // need to convert gr<->xmm
  POD_movl( local_conversion_ptr, PODLIB_malloc( 16 ) );

  POD_movl( local_clusters_ptr, PODLIB_malloc( localClusterAttributesSize ) );
  POD_movl( local_new_centers_ptr, PODLIB_malloc( localClusterAttributesSize ) );
  POD_movl( local_new_centers_len_ptr, PODLIB_malloc( 4*nclusters ) );



  /* allocate space for returning variable clusters[] */
  clusters    = (float**) malloc(nclusters *             sizeof(float*));
  clusters[0] = (float*)  malloc(nclusters * nfeatures * sizeof(float));

  for (i=1; i<nclusters; i++)
    clusters[i] = clusters[i-1] + nfeatures;


  /* randomly pick cluster centers */
  for (i=0; i<nclusters; i++) {
    //int n = (int)random() % npoints;
    int n = select_initial_cluster(nclusters, i);
    for (j=0; j<nfeatures; j++)
        clusters[i][j] = feature[n][j];
  }


  /* start of POD code */

  POD_movl( sys_clusters_ptr, (unsigned long long)  clusters[0] );


  POD_movl( local_cluster_attributes_size_gr, localClusterAttributesSize );
  $ASM copyblk local[ local_clusters_ptr ] = sys[ sys_clusters_ptr ], local_cluster_attributes_size_gr

  // send 1.0 for delta increase
  POD_movl( tmp_gr, (unsigned long long) 0x3f800000 );
```

```
$ASM st4 local[ local_conversion_ptr + 0 ] = tmp_gr
$ASM ldxmm4.scalar one_xmm = local[ local_conversion_ptr + 0 ]

// initialize distance
// initialize membership
$ASM or membership_ptr = local_membership_ptr, local_membership_ptr
$ASM sub4zx minus_one_gr = zero_gr, 1
$ASM or distance_ptr = local_distance_ptr, local_distance_ptr
POD_movl( float_max_gr, (unsigned long long) MAX_FLOAT );
for ( i = 0; i < npoints/npe; i++ ) {
  $ASM st4++ local[ distance_ptr ] = float_max_gr, four_gr
  $ASM st4++ local[ membership_ptr ] = minus_one_gr, four_gr
}
// initialize new_centers_len
$ASM add4zx tmp_gr = local_new_centers_len_ptr, 0
for ( int i = 0; i < (nclusters); i++ ) {
  $ASM st4++ local[ tmp_gr ] = zero_gr, four_gr
}
// initialize new_centers
$ASM add4zx tmp_gr = local_new_centers_ptr, 0
for ( int i = 0; i < (localClusterAttributesSize/16); i++ ) {
  $ASM stxmm++.pack local[ tmp_gr ] = zero_xmm, sixteen_gr
}

POD_mfence();

do {
  $ASM or current_cl_ptr = local_clusters_ptr, local_clusters_ptr
  $ASM add4zx cluster_num_gr = zero_gr, zero_gr
  POD_movl( pt_size, (nfeatures*4) );

  for ( i = 0; i < nclusters; i++ ) {
    find_nearest_point_pod(nfeatures, npoints);
    $ASM sub4zx current_cl_ptr = cl_ptr, sixteen_gr
    $ASM add4zx cluster_num_gr = cluster_num_gr, 1
  }

  $ASM or index_ptr = local_index_ptr, local_index_ptr

  $ASM or membership_ptr = local_membership_ptr, local_membership_ptr
  $ASM ld4++.sxt index_gr = local[ index_ptr ], four_gr

  $ASM pintxor delta_xmm = delta_xmm, delta_xmm
  $ASM ld4.sxt membership_gr = local[ membership_ptr + 0 ]

  $ASM or current_object_ptr = local_attributes_ptr, local_attributes_ptr
```

```
$ASM or distance_ptr = local_distance_ptr , local_distance_ptr

int disableIndex = npoints−originalnpoints;
for ( i = 0; i < npoints; i+= npe ) {

  if ( ((npoints−i)/npe) <= disableIndex ) {
    $ASM sub4zx tmp_gr = my_pe, npe_minus_one_gr
    $ASM nop.g
    $ASM pushmask.and.not.e
  }
  // reset distance
  $ASM st4++ local[ distance_ptr ] = float_max_gr, four_gr

  // increase delta and set new membership
  $ASM sub4zx tmp_gr = membership_gr, index_gr
  $ASM nop.g
  $ASM pushmask.and.not.e
    $ASM pfpadd.scalar.sp delta_xmm = delta_xmm, one_xmm
  $ASM popmask
  $ASM st4++ local[ membership_ptr ] = index_gr, four_gr

  // increase new_centers_len
  // first shl should be overlapped
  // needs to be optimized!!!
  $ASM shl tmp_gr = index_gr, 2
  $ASM add4zx tmp_gr = local_new_centers_len_ptr, tmp_gr
  $ASM nop.g
  $ASM ld4.sxt new_centers_len_gr = local[ tmp_gr + 0 ]
  $ASM nop.g
  $ASM nop.g
  $ASM add4zx new_centers_len_gr = new_centers_len_gr, 1
  $ASM nop.g
  $ASM st4 local[ tmp_gr + 0 ] = new_centers_len_gr

  if ( ((npoints−i)/npe) <= disableIndex ) {
    $ASM popmask
  }


  // need to work on line 101
  POD_movl( new_centers_index, (unsigned long long) (nfeatures*4) );

  $ASM imul4 new_centers_index = index_gr, new_centers_index
  $ASM ldxmm++.pack pt_dim0 = local[ current_object_ptr ], sixteen_gr

  $ASM nop.g
  $ASM ldxmm++.pack pt_dim1 = local[ current_object_ptr ], sixteen_gr
```

```
    $ASM add4zx new_centers_index = new_centers_index , local_new_centers_ptr
    $ASM ldxmm++.pack pt_dim2 = local [ current_object_ptr ] , sixteen_gr

    $ASM ldxmm++.pack pt_dim3 = local [ current_object_ptr ] , sixteen_gr

    $ASM ldxmm++.pack pt_dim4 = local [ current_object_ptr ] , sixteen_gr

    $ASM ldxmm++.pack new_centers_dim0 = local [ new_centers_index ] , sixteen_gr
    $ASM ldxmm++.pack new_centers_dim1 = local [ new_centers_index ] , sixteen_gr
    $ASM ldxmm++.pack new_centers_dim2 = local [ new_centers_index ] , sixteen_gr
    $ASM ldxmm++.pack new_centers_dim3 = local [ new_centers_index ] , sixteen_gr
    $ASM ldxmm++.pack new_centers_dim4 = local [ new_centers_index ] , sixteen_gr
    for ( j = 0; j < nfeatures ; j+=20 ) {
      $ASM sub4zx new_centers_index = new_centers_index , eighty_gr
      $ASM pfpadd.pack.sp new_centers_dim0 = new_centers_dim0 , pt_dim0
      $ASM ldxmm++.pack pt_dim0 = local [ current_object_ptr ] , sixteen_gr
      $ASM pfpadd.pack.sp new_centers_dim1 = new_centers_dim1 , pt_dim1
      $ASM ldxmm++.pack pt_dim1 = local [ current_object_ptr ] , sixteen_gr
      $ASM pfpadd.pack.sp new_centers_dim2 = new_centers_dim2 , pt_dim2
      $ASM ldxmm++.pack pt_dim2 = local [ current_object_ptr ] , sixteen_gr
      $ASM pfpadd.pack.sp new_centers_dim3 = new_centers_dim3 , pt_dim3
      $ASM ldxmm++.pack pt_dim3 = local [ current_object_ptr ] , sixteen_gr
      $ASM pfpadd.pack.sp new_centers_dim4 = new_centers_dim4 , pt_dim4
      $ASM ldxmm++.pack pt_dim4 = local [ current_object_ptr ] , sixteen_gr

      $ASM stxmm++.pack local [ new_centers_index ] = new_centers_dim0 , sixteen_gr
      $ASM stxmm++.pack local [ new_centers_index ] = new_centers_dim1 , sixteen_gr
      $ASM stxmm++.pack local [ new_centers_index ] = new_centers_dim2 , sixteen_gr
      $ASM stxmm++.pack local [ new_centers_index ] = new_centers_dim3 , sixteen_gr
      $ASM stxmm++.pack local [ new_centers_index ] = new_centers_dim4 , sixteen_gr

      $ASM ldxmm++.pack new_centers_dim0 = local [ new_centers_index ] , sixteen_gr
      $ASM ldxmm++.pack new_centers_dim1 = local [ new_centers_index ] , sixteen_gr
      $ASM ldxmm++.pack new_centers_dim2 = local [ new_centers_index ] , sixteen_gr
      $ASM ldxmm++.pack new_centers_dim3 = local [ new_centers_index ] , sixteen_gr
      $ASM ldxmm++.pack new_centers_dim4 = local [ new_centers_index ] , sixteen_gr
    }


    $ASM sub4zx current_object_ptr = current_object_ptr , eighty_gr
    $ASM ld4++.sxt index_gr = local [ index_ptr ] , four_gr
    $ASM ld4.sxt membership_gr = local [ membership_ptr + 0 ]
    $ASM nop.g
    $ASM nop.g
  }
  // reduction!!!
```

```
$ASM add4zx tmp_gr = local_new_centers_len_ptr , 0
$ASM add4zx new_centers_index = local_new_centers_ptr , 0
$ASM add4zx cl_ptr = local_clusters_ptr , 0
for ( i = 0; i < nclusters ; i++ ) {
  // first , new_centers_len
  $ASM ld4.sxt comm_gr = local[ tmp_gr + 0 ]
  $ASM nop.g
  $ASM nop.g
  $ASM add4zx new_centers_len_sum = comm_gr, 0
  for ( j = 0; j < (npeX−1); j++ ) {
    $ASM xfer.wrap.e comm_gr = comm_gr
    $ASM nop.g
    $ASM add4zx new_centers_len_sum = new_centers_len_sum , comm_gr
  }
  $ASM add4zx comm_gr = new_centers_len_sum , 0
  for ( j = 0; j < (npeY−1); j++ ) {
    $ASM xfer.wrap.n comm_gr = comm_gr
    $ASM nop.g
    $ASM add4zx new_centers_len_sum = new_centers_len_sum , comm_gr
  }
  $ASM st4 local[ tmp_gr + 0 ] = new_centers_len_sum
  $ASM ldxmm4.scalar new_centers_len_xmm = local[ tmp_gr + 0 ]

  // then , new_centers!
  $ASM ldxmm++.pack comm_dim0 = local[ new_centers_index ], sixteen_gr
  $ASM ldxmm++.pack comm_dim1 = local[ new_centers_index ], sixteen_gr
  $ASM pfpshuffle.sp.aaaa new_centers_len_xmm = new_centers_len_xmm , new_centers_len_xmm
  $ASM ldxmm++.pack comm_dim2 = local[ new_centers_index ], sixteen_gr
  $ASM pfpadd.pack.sp new_centers_sum0 = comm_dim0 , zero_xmm
  $ASM ldxmm++.pack comm_dim3 = local[ new_centers_index ], sixteen_gr
  $ASM pfpadd.pack.sp new_centers_sum1 = comm_dim1 , zero_xmm
  $ASM ldxmm++.pack comm_dim4 = local[ new_centers_index ], sixteen_gr
  $ASM pfpadd.pack.sp new_centers_sum2 = comm_dim2 , zero_xmm
  $ASM pfpadd.pack.sp new_centers_sum3 = comm_dim3 , zero_xmm
  $ASM pfpadd.pack.sp new_centers_sum4 = comm_dim4 , zero_xmm
  $ASM pcvti2f.pack.sp new_centers_len_xmm = new_centers_len_xmm

  for ( k = 0; k < nfeatures ; k += 20 ) {
    for ( j = 0; j < (npeX−1); j++ ) {
      $ASM xferxmm.wrap.e comm_dim0 = comm_dim0
      $ASM xferxmm.wrap.e comm_dim1 = comm_dim1
      $ASM xferxmm.wrap.e comm_dim2 = comm_dim2
      $ASM xferxmm.wrap.e comm_dim3 = comm_dim3
      $ASM xferxmm.wrap.e comm_dim4 = comm_dim4
      $ASM pfpadd.pack.sp new_centers_sum0 = new_centers_sum0 , comm_dim0
      $ASM pfpadd.pack.sp new_centers_sum1 = new_centers_sum1 , comm_dim1
      $ASM pfpadd.pack.sp new_centers_sum2 = new_centers_sum2 , comm_dim2
  $ASM pfpadd.pack.sp new_centers_sum3 = new_centers_sum3, comm_dim3
  $ASM pfpadd.pack.sp new_centers_sum4 = new_centers_sum4, comm_dim4
}
$ASM pfpadd.pack.sp comm_dim0 = new_centers_sum0, zero_xmm
$ASM pfpadd.pack.sp comm_dim1 = new_centers_sum1, zero_xmm
$ASM pfpadd.pack.sp comm_dim2 = new_centers_sum2, zero_xmm
$ASM pfpadd.pack.sp comm_dim3 = new_centers_sum3, zero_xmm
$ASM pfpadd.pack.sp comm_dim4 = new_centers_sum4, zero_xmm
for ( j = 0; j < (npeY-1); j++ ) {
  $ASM xferxmm.wrap.n comm_dim0 = comm_dim0
  $ASM xferxmm.wrap.n comm_dim1 = comm_dim1
  $ASM xferxmm.wrap.n comm_dim2 = comm_dim2
  $ASM xferxmm.wrap.n comm_dim3 = comm_dim3
  $ASM xferxmm.wrap.n comm_dim4 = comm_dim4
  $ASM pfpadd.pack.sp new_centers_sum0 = new_centers_sum0, comm_dim0
  $ASM pfpadd.pack.sp new_centers_sum1 = new_centers_sum1, comm_dim1
  $ASM pfpadd.pack.sp new_centers_sum2 = new_centers_sum2, comm_dim2
  $ASM pfpadd.pack.sp new_centers_sum3 = new_centers_sum3, comm_dim3
  $ASM pfpadd.pack.sp new_centers_sum4 = new_centers_sum4, comm_dim4
}
$ASM pfpdiv.pack.sp new_centers_sum0 = new_centers_sum0, new_centers_len_xmm
PODLIB_NOPs( PFPDIV_LAT - 1 );
$ASM pfpdiv.pack.sp new_centers_sum1 = new_centers_sum1, new_centers_len_xmm
PODLIB_NOPs( PFPDIV_LAT - 1 );
$ASM pfpdiv.pack.sp new_centers_sum2 = new_centers_sum2, new_centers_len_xmm
PODLIB_NOPs( PFPDIV_LAT - 1 );
$ASM pfpdiv.pack.sp new_centers_sum3 = new_centers_sum3, new_centers_len_xmm
PODLIB_NOPs( PFPDIV_LAT - 1 );
$ASM pfpdiv.pack.sp new_centers_sum4 = new_centers_sum4, new_centers_len_xmm
$ASM nop.m

$ASM sub4zx new_centers_index = new_centers_index, eighty_gr
$ASM nop.g
// make new_centers zero
$ASM stxmm++.pack local[ new_centers_index ] = zero_xmm, sixteen_gr
$ASM stxmm++.pack local[ new_centers_index ] = zero_xmm, sixteen_gr
$ASM stxmm++.pack local[ new_centers_index ] = zero_xmm, sixteen_gr
$ASM stxmm++.pack local[ new_centers_index ] = zero_xmm, sixteen_gr
$ASM stxmm++.pack local[ new_centers_index ] = zero_xmm, sixteen_gr
// load new_centers for next iteration
$ASM ldxmm++.pack comm_dim0 = local[ new_centers_index ], sixteen_gr
$ASM ldxmm++.pack comm_dim1 = local[ new_centers_index ], sixteen_gr
$ASM ldxmm++.pack comm_dim2 = local[ new_centers_index ], sixteen_gr
$ASM ldxmm++.pack comm_dim3 = local[ new_centers_index ], sixteen_gr
$ASM ldxmm++.pack comm_dim4 = local[ new_centers_index ], sixteen_gr

PODLIB_NOPs( PFPDIV_LAT - 12 );
```

```
      // update clusters
      // and prepare new_centers_sum for next iteration
      $ASM pfpadd.pack.sp new_centers_sum0 = comm_dim0, zero_xmm
      $ASM stxmm++.pack local[ cl_ptr ] = new_centers_sum0, sixteen_gr
      $ASM pfpadd.pack.sp new_centers_sum1 = comm_dim1, zero_xmm
      $ASM stxmm++.pack local[ cl_ptr ] = new_centers_sum1, sixteen_gr
      $ASM pfpadd.pack.sp new_centers_sum2 = comm_dim2, zero_xmm
      $ASM stxmm++.pack local[ cl_ptr ] = new_centers_sum2, sixteen_gr
      $ASM pfpadd.pack.sp new_centers_sum3 = comm_dim3, zero_xmm
      $ASM stxmm++.pack local[ cl_ptr ] = new_centers_sum3, sixteen_gr
      $ASM pfpadd.pack.sp new_centers_sum4 = comm_dim4, zero_xmm
      $ASM stxmm++.pack local[ cl_ptr ] = new_centers_sum4, sixteen_gr

    }
    $ASM sub4zx new_centers_index = new_centers_index, eighty_gr
    // make new_centers_len zero
    $ASM st4++ local[ tmp_gr ] = zero_gr, four_gr
  }

  // global reduction for delta!!!
  $ASM pintor comm_dim0 = delta_xmm, delta_xmm
  $ASM pfpadd.scalar.sp delta_sum = delta_xmm, zero_xmm
  PODLIB_NOPs( PINTXOR_LAT - 2 );

  for ( j = 0; j < (npeX-1); j++ ) {
    $ASM xferxmm.wrap.e comm_dim0 = comm_dim0
    $ASM nop.x
    $ASM pfpadd.scalar.sp delta_sum = delta_sum, comm_dim0
    PODLIB_NOPs( PFPADD_LAT - 3 );
  }
  PODLIB_NOPs( 2 );
  $ASM pfpadd.scalar.sp comm_dim0 = delta_sum, zero_xmm
  PODLIB_NOPs( PFPADD_LAT - 1 );
  for ( j = 0; j < (npeY-1); j++ ) {
    $ASM xferxmm.wrap.n comm_dim0 = comm_dim0
    $ASM nop.x
    $ASM pfpadd.scalar.sp delta_sum = delta_sum, comm_dim0
    PODLIB_NOPs( PFPADD_LAT - 3 );
  }
  PODLIB_NOPs( 2 );

  $ASM stxmm4.scalar local[ local_conversion_ptr + 0 ] = delta_sum
  $ASM ld4.zxt delta_gr = local[ local_conversion_ptr + 0 ]
  $ASM nop.g
  $ASM nop.g
  $ASM sub4zx tmp_gr = my_pe, npex_minus_one_gr
```

```
    $ASM nop.g
    $ASM pushmask.and.e
      $ASM xferdrb delta_gr
    $ASM popmask

    int drb_value = POD_getdrb();
    memcpy( &delta, &drb_value, 4 );
    //printf( "delta = %f\n", delta );
    //getchar();
    delta /= originalnpoints;


  }
  while (delta > threshold && loop++ < 500);

  // write-back
  $ASM or membership_ptr = local_membership_ptr, local_membership_ptr
  POD_movl( tmp_gr, (npoints/npe*4) );
  $ASM imul4 membership_ptr = tmp_gr, my_pe
  POD_movl( sys_membership_ptr, (unsigned long long) membership );
  $ASM add4zx membership_ptr = membership_ptr, sys_membership_ptr
  $ASM nop.g
  $ASM copyblk sys[ membership_ptr] = local[ local_membership_ptr ], tmp_gr


  POD_mfence();

  PODLIB_free( 4*nclusters );
  PODLIB_free( localClusterAttributesSize );
  PODLIB_free( localClusterAttributesSize );

}

void euclid_dist_2_pod(int numdims) {
  int i;
  int loop_count;


  //$ASM pintxor distance0 = distance0, distance0
  $ASM pfpsub.pack.sp distance0 = distance0, distance0
  $ASM ldxmm++.pack cl_dim = local[ cl_ptr ], sixteen_gr

  //$ASM pintxor distance1 = distance1, distance1
  $ASM pfpsub.pack.sp distance1 = distance1, distance1
  $ASM ldxmm++.pack pt0_dim = local[ pt0_ptr ], sixteen_gr

  //$ASM pintxor distance2 = distance2, distance2
```

```
$ASM pfpsub.pack.sp distance2 = distance2 , distance2
$ASM ldxmm++.pack pt1_dim = local[ pt1_ptr ] , sixteen_gr

//$ASM pintxor distance3 = distance3 , distance3
$ASM pfpsub.pack.sp distance3 = distance3 , distance3
$ASM ldxmm++.pack pt2_dim = local[ pt2_ptr ] , sixteen_gr

//$ASM pintxor distance4 = distance4 , distance4
$ASM pfpsub.pack.sp distance4 = distance4 , distance4
$ASM ldxmm++.pack pt3_dim = local[ pt3_ptr ] , sixteen_gr

$ASM ldxmm++.pack pt4_dim = local[ pt4_ptr ] , sixteen_gr

for ( i = 0; i < numdims ; i+=4 ) {
  $ASM pfpsub.pack.sp pt0_dim = pt0_dim , cl_dim
  $ASM pfpsub.pack.sp pt1_dim = pt1_dim , cl_dim
  $ASM pfpsub.pack.sp pt2_dim = pt2_dim , cl_dim
  $ASM pfpsub.pack.sp pt3_dim = pt3_dim , cl_dim
  $ASM pfpsub.pack.sp pt4_dim = pt4_dim , cl_dim
  $ASM ldxmm++.pack cl_dim = local[ cl_ptr ] , sixteen_gr

  $ASM pfpfma++.pack.sp distance0 += pt0_dim , pt0_dim
  $ASM ldxmm++.pack pt0_dim = local[ pt0_ptr ] , sixteen_gr

  $ASM pfpfma++.pack.sp distance1 += pt1_dim , pt1_dim
  $ASM ldxmm++.pack pt1_dim = local[ pt1_ptr ] , sixteen_gr

  $ASM pfpfma++.pack.sp distance2 += pt2_dim , pt2_dim
  $ASM ldxmm++.pack pt2_dim = local[ pt2_ptr ] , sixteen_gr

  $ASM pfpfma++.pack.sp distance3 += pt3_dim , pt3_dim
  $ASM ldxmm++.pack pt3_dim = local[ pt3_ptr ] , sixteen_gr

  $ASM pfpfma++.pack.sp distance4 += pt4_dim , pt4_dim
  $ASM ldxmm++.pack pt4_dim = local[ pt4_ptr ] , sixteen_gr
}

$ASM pfphadd.sp distance0 = distance0 , distance0
$ASM pfphadd.sp distance1 = distance1 , distance1
$ASM pfphadd.sp distance2 = distance2 , distance2
$ASM pfphadd.sp distance3 = distance3 , distance3
$ASM pfphadd.sp distance4 = distance4 , distance4

// load max_distance for later computation
$ASM pfphadd.sp distance0 = distance0 , distance0
$ASM ldxmm4.scalar max_distance0 = local[ distance_ptr + 0 ]
```

```
  $ASM pfphadd.sp distance1 = distance1, distance1
  $ASM ldxmm4.scalar max_distance1 = local[ distance_ptr + 4 ]

  $ASM pfphadd.sp distance2 = distance2, distance2
  $ASM ldxmm4.scalar max_distance2 = local[ distance_ptr + 8 ]

  $ASM pfphadd.sp distance3 = distance3, distance3
  $ASM ldxmm4.scalar max_distance3 = local[ distance_ptr + 12 ]

  $ASM pfphadd.sp distance4 = distance4, distance4
  $ASM ldxmm4.scalar max_distance4 = local[ distance_ptr + 16 ]

}



void find_nearest_point_pod(int nfeatures, int npoints) {
  int i;

  // need to set max_dist_ptr somewhere

  $ASM or distance_ptr = local_distance_ptr, local_distance_ptr


  // need to set pt_size somewhere
  $ASM or pt0_ptr = local_attributes_ptr, local_attributes_ptr
  $ASM add4zx pt1_ptr = pt0_ptr, pt_size
  $ASM add4zx pt2_ptr = pt1_ptr, pt_size
  $ASM add4zx pt3_ptr = pt2_ptr, pt_size
  $ASM add4zx pt4_ptr = pt3_ptr, pt_size

  $ASM sub4zx index_ptr = local_index_ptr, 4

  // cluster_num_gr should be set somewhere outside
  for ( i = 0; i < npoints; i+=objectGran ) {

    $ASM or cl_ptr = current_cl_ptr, current_cl_ptr

    euclid_dist_2_pod(nfeatures);


    $ASM pfpcmp.lt.scalar.sp temp_xmm0 = distance0, max_distance0
    $ASM pfpcmp.lt.scalar.sp temp_xmm1 = distance1, max_distance1
    $ASM pfpcmp.lt.scalar.sp temp_xmm2 = distance2, max_distance2
    $ASM pfpcmp.lt.scalar.sp temp_xmm3 = distance3, max_distance3
    $ASM pfpcmp.lt.scalar.sp temp_xmm4 = distance4, max_distance4
```

```
// set max_distance and index

// to convert xmm type cmp result to gr type
$ASM stxmm4.scalar local[ local_conversion_ptr + 0 ] = temp_xmm0
$ASM ld4.zxt tmp_gr0 = local[ local_conversion_ptr + 0 ]

$ASM stxmm4.scalar local[ local_conversion_ptr + 4 ] = temp_xmm1
$ASM ld4.zxt tmp_gr1 = local[ local_conversion_ptr + 4 ]

$ASM sub4zx tmp_gr0 = tmp_gr0, zero_gr
$ASM add4zx index_ptr = index_ptr, 4
$ASM pfpmin.scalar.sp max_distance0 = distance0, max_distance0
$ASM pushmask.and.not.e
  $ASM st4 local[ index_ptr + 0 ] = cluster_num_gr
$ASM popmask

$ASM stxmm4.scalar local[ local_conversion_ptr + 0 ] = temp_xmm2
$ASM ld4.zxt tmp_gr0 = local[ local_conversion_ptr + 0 ]

$ASM sub4zx tmp_gr1 = tmp_gr1, zero_gr
$ASM add4zx index_ptr = index_ptr, 4
$ASM pfpmin.scalar.sp max_distance1 = distance1, max_distance1
$ASM pushmask.and.not.e
  $ASM st4 local[ index_ptr + 0 ] = cluster_num_gr
$ASM popmask

$ASM stxmm4.scalar local[ local_conversion_ptr + 4 ] = temp_xmm3
$ASM ld4.zxt tmp_gr1 = local[ local_conversion_ptr + 4 ]

$ASM sub4zx tmp_gr0 = tmp_gr0, zero_gr
$ASM add4zx index_ptr = index_ptr, 4
$ASM pfpmin.scalar.sp max_distance2 = distance2, max_distance2
$ASM pushmask.and.not.e
  $ASM st4 local[ index_ptr + 0 ] = cluster_num_gr
$ASM popmask

$ASM stxmm4.scalar local[ local_conversion_ptr + 0 ] = temp_xmm4
$ASM ld4.zxt tmp_gr0 = local[ local_conversion_ptr + 0 ]

$ASM sub4zx tmp_gr1 = tmp_gr1, zero_gr
$ASM add4zx index_ptr = index_ptr, 4
$ASM pfpmin.scalar.sp max_distance3 = distance3, max_distance3
$ASM pushmask.and.not.e
  $ASM st4 local[ index_ptr + 0 ] = cluster_num_gr
$ASM popmask

$ASM sub4zx tmp_gr0 = tmp_gr0, zero_gr
```

```
    $ASM add4zx index_ptr = index_ptr , 4
    $ASM pfpmin.scalar.sp max_distance4 = distance4 , max_distance4
    $ASM pushmask.and.not.e
      $ASM st4 local[ index_ptr + 0 ] = cluster_num_gr
    $ASM popmask

    // should be overlapped with previous nasty block
    $ASM sub4zx pt0_ptr = pt4_ptr , sixteen_gr
    $ASM stxmm4++.scalar local[ distance_ptr ] = max_distance0 , four_gr

    $ASM add4zx pt1_ptr = pt0_ptr , pt_size
    $ASM stxmm4++.scalar local[ distance_ptr ] = max_distance1 , four_gr

    $ASM add4zx pt2_ptr = pt1_ptr , pt_size
    $ASM stxmm4++.scalar local[ distance_ptr ] = max_distance2 , four_gr

    $ASM add4zx pt3_ptr = pt2_ptr , pt_size
    $ASM stxmm4++.scalar local[ distance_ptr ] = max_distance3 , four_gr

    $ASM add4zx pt4_ptr = pt3_ptr , pt_size
    $ASM stxmm4++.scalar local[ distance_ptr ] = max_distance4 , four_gr


  }
  $ASM nop.g
}


void readFromFile() {
  FILE* file;
  int i;
  char* filename = "edge";

/*
        if ((infile = open(filename, O_RDONLY, "0600")) == -1) {
            fprintf(stderr, "Error: no such file (%s)\n", filename);
            exit(1);
        }
*/
  file = fopen( "edge", "r" );
  if ( file == NULL ) {
            fprintf(stderr, "Error: no such file (%s)\n", filename);
            exit(1);
  }
        //read(infile, &numObjects,     sizeof(int));
        //read(infile, &numAttributes, sizeof(int));
  fread( &numObjects, sizeof(int), 1, file );
  fread( &numAttributes, sizeof(int), 1, file );

        /* allocate space for attributes[] and read attributes of all objects */
        buf           = (float*) malloc(numObjects*numAttributes*sizeof(float));
        attributes    = (float**)malloc(numObjects*             sizeof(float*));
        attributes[0] = (float*) malloc(numObjects*numAttributes*sizeof(float));
        for (i=1; i<numObjects; i++)
            attributes[i] = attributes[i-1] + numAttributes;

        //read(infile, buf, numObjects*numAttributes*sizeof(float));
  fread( buf, sizeof(float), numObjects*numAttributes, file );

        //close(infile);
  fclose(file);
  memcpy(attributes[0], buf, numObjects*numAttributes*sizeof(float));
}

float** kmeans_clustering(float **feature,    /* in: [npoints][nfeatures] */
                          int     nfeatures,
                          int     npoints,
                          int     nclusters,
                          float   threshold,
                          int    *membership) /* out: [npoints] */
{

    int      i, j, index, loop=0;
    int     *new_centers_len; /* [nclusters]: no. of points in each cluster */
    float    delta;
    float  **clusters;   /* out: [nclusters][nfeatures] */
    float  **new_centers;     /* [nclusters][nfeatures] */
    double   timing;

    /* allocate space for returning variable clusters[] */
    clusters    = (float**) malloc(nclusters *             sizeof(float*));
    clusters[0] = (float*)  malloc(nclusters * nfeatures * sizeof(float));
    for (i=1; i<nclusters; i++)
        clusters[i] = clusters[i-1] + nfeatures;

    /* randomly pick cluster centers */
    for (i=0; i<nclusters; i++) {
  //int n = (int)random() % npoints;
  int n = select_initial_cluster(nclusters, i);
        for (j=0; j<nfeatures; j++)
            clusters[i][j] = feature[n][j];
    }

    for (i=0; i<npoints; i++)
```

```
membership[i] = -1;

  /* need to initialize new_centers_len and new_centers[0] to all 0 */
  new_centers_len = (int*) calloc(nclusters, sizeof(int));

  new_centers    = (float**) malloc(nclusters *            sizeof(float*));
  new_centers[0] = (float*)  calloc(nclusters * nfeatures, sizeof(float));
  for (i=1; i<nclusters; i++)
      new_centers[i] = new_centers[i-1] + nfeatures;

  do {
      delta = 0.0;

      for (i=0; i<npoints; i++) {
    /* find the index of nestest cluster centers */
    index = find_nearest_point(feature[i],
              nfeatures,
              clusters,
              nclusters);
    /* if membership changes, increase delta by 1 */
    if (membership[i] != index) delta += 1.0;

    /* assign the membership to object i */
    membership[i] = index;

    /* update new cluster centers : sum of objects located within */
    new_centers_len[index]++;
    for (j=0; j<nfeatures; j++)
  new_centers[index][j] += feature[i][j];
      }

/* replace old cluster centers with new_centers */
      for (i=0; i<nclusters; i++) {
          for (j=0; j<nfeatures; j++) {
              if (new_centers_len[i] > 0)
      clusters[i][j] = new_centers[i][j] / new_centers_len[i];
  new_centers[i][j] = 0.0;   /* set back to 0 */
    }
    new_centers_len[i] = 0;   /* set back to 0 */
}

      delta /= npoints;
  }
  while (delta > threshold && loop++ < 500);

  free(new_centers[0]);
  free(new_centers);
```

```
    free(new_centers_len);

    return clusters;
}

int find_nearest_point(float  *pt,          /* [nfeatures] */
                       int     nfeatures,
                       float **pts,         /* [npts][nfeatures] */
                       int     npts)
{
    int index, i;
    float max_dist=FLT_MAX;

    /* find the cluster center id with min distance to pt */
    for (i=0; i<npts; i++) {
        float dist;
        dist = euclid_dist_2(pt, pts[i], nfeatures);  /* no need square root */
        if (dist < max_dist) {
            max_dist = dist;
            index    = i;
        }
    }
    return(index);
}

float euclid_dist_2(float *pt1,
                    float *pt2,
                    int    numdims)
{
    int i;
    float ans=0.0;

    for (i=0; i<numdims; i++) {
        ans += (pt1[i]-pt2[i]) * (pt1[i]-pt2[i]);
/*
  printf("%d: %f - %f = %f\n", i, pt1[i], pt2[i] );
  getchar();
*/
  }

    return(ans);
}
```

E PODISA (tentative)

E.1 G-format POD Instructions

ADC - Add with carry

Instruction	**Description**
adc1zx r1 = r2, r3	Add with carry unsigned 8-bit data r2 and r3
adc1zx r1 = r2, immed6	Add with carry unsigned 8-bit data r2 and unsigned 6-bit data immed6
adc1sx r1 = r2, r3	Add with carry signed 8-bit data r2 and r3
adc1sx r1 = r2, immed6	Add with carry signed 8-bit data r2 and signed 6-bit data immed6
adc2zx r1 = r2, r3	Add with carry unsigned 16-bit data r2 and r3
adc2zx r1 = r2, immed6	Add with carry unsigned 16-bit data r2 and unsigned 6-bit data immed6
adc2sx r1 = r2, r3	Add with carry signed 16-bit data r2 and r3
adc2sx r1 = r2, immed6	Add with carry signed 16-bit data r2 and signed 6-bit data immed6
adc4zx r1 = r2, r3	Add with carry unsigned 32-bit data r2 and r3
adc4zx r1 = r2, immed6	Add with carry unsigned 32-bit data r2 and unsigned 6-bit data immed6
adc4sx r1 = r2, r3	Add with carry signed 32-bit data r2 and r3
adc4sx r1 = r2, immed6	Add with carry signed 32-bit data r2 and signed 6-bit data immed6
adc8zx r1 = r2, r3	Add with carry unsigned 64-bit data r2 and r3
adc8zx r1 = r2, immed6	Add with carry unsigned 64-bit data r2 and unsigned 6-bit data immed6
adc8sx r1 = r2, r3	Add with carry signed 64-bit data r2 and r3
adc8sx r1 = r2, immed6	Add with carry signed 64-bit data r2 and signed 6-bit data immed6

ADD - Add

Instruction	Description
add1zx r1 = r2, r3	Add unsigned 8-bit data r2 and r3
add1zx r1 = r2, immed6	Add unsigned 8-bit data r2 and unsigned 6-bit data immed6
add1sx r1 = r2, r3	Add signed 8-bit data r2 and r3
add1sx r1 = r2, immed6	Add signed 8-bit data r2 and signed 6-bit data immed6
add2zx r1 = r2, r3	Add unsigned 16-bit data r2 and r3
add2zx r1 = r2, immed6	Add unsigned 16-bit data r2 and unsigned 6-bit data immed6
add2sx r1 = r2, r3	Add signed 16-bit data r2 and r3
add2sx r1 = r2, immed6	Add signed 16-bit data r2 and signed 6-bit data immed6
add4zx r1 = r2, r3	Add unsigned 32-bit data r2 and r3
add4zx r1 = r2, immed6	Add unsigned 32-bit data r2 and unsigned 6-bit data immed6
add4sx r1 = r2, r3	Add signed 32-bit data r2 and r3
add4sx r1 = r2, immed6	Add signed 32-bit data r2 and signed 6-bit data immed6
add8zx r1 = r2, r3	Add unsigned 64-bit data r2 and r3
add8zx r1 = r2, immed6	Add unsigned 64-bit data r2 and unsigned 6-bit data immed6
add8sx r1 = r2, r3	Add signed 64-bit data r2 and r3
add8sx r1 = r2, immed6	Add signed 64-bit data r2 and signed 6-bit data immed6

AND - Bitwise Logical AND

Instruction	Description
and r1 = r2, r3	Bitwise logical AND of r2 and r3
and r1 = r2, immed6	Bitwise logical AND of r2 and immed6

BT - Bit test

Instruction	Description
bt r2, immed6	Test immed6-th bit, and set the flags accordingly

CMOV - Conditional move

Instruction	Description
cmov.o r1 = r2, r3	Move if overflow
cmov.o r1 = r2, immed6	Move if overflow
cmov.not.o r1 = r2, r3	Move if not overflow
cmov.not.o r1 = r2, immed6	Move if not overflow
cmov.b r1 = r2, r3	Move if below
cmov.b r1 = r2, immed6	Move if below
cmov.not.b r1 = r2, r3	Move if not below
cmov.not.b r1 = r2, immed6	Move if not below
cmov.e r1 = r2, r3	Move if equal
cmov.e r1 = r2, immed6	Move if equal
cmov.not.e r1 = r2, r3	Move if not equal
cmov.not.e r1 = r2, immed6	Move if not equal
cmov.be r1 = r2, r3	Move if below or equal
cmov.be r1 = r2, immed6	Move if below or equal
cmov.not.be r1 = r2, r3	Move if not below or equal
cmov.not.be r1 = r2, immed6	Move if not below or equal
cmov.s r1 = r2, r3	Move if sign
cmov.s r1 = r2, immed6	Move if sign
cmov.not.s r1 = r2, r3	Move if not sign
cmov.not.s r1 = r2, immed6	Move if not sign
cmov.l r1 = r2, r3	Move if less
cmov.l r1 = r2, immed6	Move if less
cmov.not.l r1 = r2, r3	Move if not less
cmov.not.l r1 = r2, immed6	Move if not less
cmov.le r1 = r2, r3	Move if less or equal
cmov.le r1 = r2, immed6	Move if less or equal
cmov.not.le r1 = r2, r3	Move if not less or equal
cmov.not.le r1 = r2, immed6	Move if not less or equal

CMP - Compare two operands

Instruction	Description
cmp1 r2, r3	Compare signed 8-bit data r2 and r3, and set the flags accordingly
cmp1 r2, immed6	Compare signed 8-bit data r2 and signed 6-bit data immed6, and set the flags accordingly
cmp2 r2, r3	Compare signed 16-bit data r2 and r3, and set the flags accordingly
cmp2 r2, immed6	Compare signed 16-bit data r2 and signed 6-bit data immed6, and set the flags accordingly
cmp4 r2, r3	Compare signed 32-bit data r2 and r3, and set the flags accordingly
cmp4 r2, immed6	Compare signed 32-bit data r2 and signed 6-bit data immed6, and set the flags accordingly
cmp8 r2, r3	Compare signed 64-bit data r2 and r3, and set the flags accordingly
cmp8 r2, immed6	Compare signed 64-bit data r2 and signed 6-bit data immed6, and set the flags accordingly

IMUL4 - Multiply

Instruction	Description
imul4 r1 = r2, r3	Multiply signed 32-bit data r2 and r3
imul4 r1 = r2, immed6	Multiply signed 32-bit data r2 and signed 6-bit data immed6

NOT - Bitwise NOT

Instruction	Description
not r1 = r2	Reverse each bit of r2
not r1 = immed6	Reverse each bit of immed6

OR - Bitwise Logical OR

Instruction	Description
or r1 = r2, r3	Bitwise logical OR of r2 and r3
or r1 = r2, immed6	Bitwise logical OR of r2 and immed6

SAR - Arithmetic shift-right

Instruction	Description
sar r1 = r2, r3	Arithmetically shift r2 to right r3 bits
sar r1 = r2, immed6	Arithmetically shift r2 to right immed6 bits

SBB - Subtract with borrow

Instruction	Description
sbb1zx r1 = r2, r3	Subtract with borrow unsigned 8-bit data r2 and r3
sbb1zx r1 = r2, immed6	Subtract with borrow unsigned 8-bit data r2 and unsigned 6-bit data immed6
sbb1sx r1 = r2, r3	Subtract with borrow signed 8-bit data r2 and r3
sbb1sx r1 = r2, immed6	Subtract with borrow signed 8-bit data r2 and signed 6-bit data immed6
sbb2zx r1 = r2, r3	Subtract with borrow unsigned 16-bit data r2 and r3
sbb2zx r1 = r2, immed6	Subtract with borrow unsigned 16-bit data r2 and unsigned 6-bit data immed6
sbb2sx r1 = r2, r3	Subtract with borrow signed 16-bit data r2 and r3
sbb2sx r1 = r2, immed6	Subtract with borrow signed 16-bit data r2 and signed 6-bit data immed6
sbb4zx r1 = r2, r3	Subtract with borrow unsigned 32-bit data r2 and r3
sbb4zx r1 = r2, immed6	Subtract with borrow unsigned 32-bit data r2 and unsigned 6-bit data immed6
sbb4sx r1 = r2, r3	Subtract with borrow signed 32-bit data r2 and r3
sbb4sx r1 = r2, immed6	Subtract with borrow signed 32-bit data r2 and signed 6-bit data immed6
sbb8zx r1 = r2, r3	Subtract with borrow unsigned 64-bit data r2 and r3
sbb8zx r1 = r2, immed6	Subtract with borrow unsigned 64-bit data r2 and unsigned 6-bit data immed6
sbb8sx r1 = r2, r3	Subtract with borrow signed 64-bit data r2 and r3
sbb8sx r1 = r2, immed6	Subtract with borrow signed 64-bit data r2 and signed 6-bit data immed6

SHL - Logical shift-left

Instruction	Description
shl r1 = r2, r3	Logically shift r2 to left r3 bits
shl r1 = r2, immed6	Logically shift r2 to left immed6 bits

SHLADD - Shift left and add

Instruction	Description
shladd1 r1 = r2, r3	Shift signed 64-bit data r2 to left 1 bit and add this number and signed 8-bit data r3
shladd1 r1 = r2, immed6	Shift signed 64-bit data r2 to left 1 bit and add this number and signed 6-bit data immed6
shladd2 r1 = r2, r3	Shift signed 64-bit data r2 to left 2 bits and add this number and signed 8-bit data r3
shladd2 r1 = r2, immed6	Shift signed 64-bit data r2 to left 2 bits and add this number and signed 6-bit data immed6
shladd3 r1 = r2, r3	Shift signed 64-bit data r2 to left 3 bits and add this number and signed 8-bit data r3
shladd3 r1 = r2, immed6	Shift signed 64-bit data r2 to left 3 bits and add this number and signed 6-bit data immed6
shladd4 r1 = r2, r3	Shift signed 64-bit data r2 to left 4 bits and add this number and signed 8-bit data r3
shladd4 r1 = r2, immed6	Shift signed 64-bit data r2 to left 4 bits and add this number and signed 6-bit data immed6

SHR - Logical shift-right

Instruction	Description
shr r1 = r2, r3	Logically shift r2 to right r3 bits
shr r1 = r2, immed6	Logically shift r2 to right immed6 bits

SUB - Subtract

Instruction	Description
sub1zx r1 = r2, r3	Sub unsigned 8-bit data r2 and r3
sub1zx r1 = r2, immed6	Sub unsigned 8-bit data r2 and unsigned 6-bit data immed6
sub1sx r1 = r2, r3	Sub signed 8-bit data r2 and r3
sub1sx r1 = r2, immed6	Sub signed 8-bit data r2 and signed 6-bit data immed6
sub2zx r1 = r2, r3	Sub unsigned 16-bit data r2 and r3
sub2zx r1 = r2, immed6	Sub unsigned 16-bit data r2 and unsigned 6-bit data immed6
sub2sx r1 = r2, r3	Sub signed 16-bit data r2 and r3
sub2sx r1 = r2, immed6	Sub signed 16-bit data r2 and signed 6-bit data immed6
sub4zx r1 = r2, r3	Sub unsigned 32-bit data r2 and r3
sub4zx r1 = r2, immed6	Sub unsigned 32-bit data r2 and unsigned 6-bit data immed6
sub4sx r1 = r2, r3	Sub signed 32-bit data r2 and r3
sub4sx r1 = r2, immed6	Sub signed 32-bit data r2 and signed 6-bit data immed6
sub8zx r1 = r2, r3	Sub unsigned 64-bit data r2 and r3
sub8zx r1 = r2, immed6	Sub unsigned 64-bit data r2 and unsigned 6-bit data immed6
sub8sx r1 = r2, r3	Sub signed 64-bit data r2 and r3
sub8sx r1 = r2, immed6	Sub signed 64-bit data r2 and signed 6-bit data immed6

XFER - Transfer register value to a neighbor PE

Instruction	Description
xfer.n r1 = r2	Copy r2 to r1 of the northern neighbor PE (Northmost PE do nothing)
xfer.wrap.n r1 = r2	Copy r2 to r1 of the northern neighbor PE (Northmost PE copies r2 to r1 of southmost PE)
xfer.e r1 = r2	Copy r2 to r1 of the eastern neighbor PE (Eastmost PE do nothing)
xfer.wrap.e r1 = r2	Copy r2 to r1 of the eastern neighbor PE (Eastmost PE copies r2 to r1 of westmost PE)
xfer.w r1 = r2	Copy r2 to r1 of the western neighbor PE (Westmost PE do nothing)
xfer.wrap.w r1 = r2	Copy r2 to r1 of the western neighbor PE (Westmost PE copies r2 to r1 of eastmost PE)
xfer.s r1 = r2	Copy r2 to r1 of the southern neighbor PE (Southmost PE do nothing)
xfer.wrap.s r1 = r2	Copy r2 to r1 of the sourthern neighbor PE (Southmost PE copies r2 to r1 of northmost PE)

XFERDRB - Transfer register value to the data return buffer

Instruction	Description
xferdrb r2	Copy r2 to the data return buffer

XOR - Bitwise Logical Exclusive OR

Instruction	Description
xor r1 = r2, r3	Bitwise logical exclusive OR of r2 and r3
xor r1 = r2, immed6	Bitwise logical exclusive OR of r2 and immed6

E.2 X-format POD Instructions

PCVTF2I - Packed floating point number conversion

Instruction	Description
pcvtf2i.scalar.sp.mxcsr xmm1 = xmm2	Convert the low single-precision floating point value from xmm2 to a 32-bit integer value
pcvtf2i.pack.sp.mxcsr xmm1 = xmm2	Convert four single-precision floating point values from xmm2 to four 32-bit integer values
pcvtf2i.scalar.dp.mxcsr xmm1 = xmm2	Convert the low double-precision floating point value from xmm2 to a 64-bit integer value
pcvtf2i.pack.dp.mxcsr xmm1 = xmm2	Convert two double-precision floating point values from xmm2 to two 64-bit integer values

PCVTI2F - Packed integer conversion

Instruction	Description
pcvti2f.scalar.sp xmm1 = xmm2	Convert the low 32-bit integer value from xmm2 to a single-precision floating point value
pcvti2f.pack.sp xmm1 = xmm2	Convert four 32-bit integer values from xmm2 to four single-precision floating point values
pcvti2f.scalar.dp xmm1 = xmm2	Convert the low 64-bit integer value from xmm2 to a double-precision floating point value
pcvti2f.pack.dp xmm1 = xmm2	Convert two 64-bit integer values from xmm2 to two double-precision floating point values

PFPADD - Packed floating point add

Instruction	Description
pfpadd.scalar.sp xmm1 = xmm2, xmm3	Add the low single-precision floating point value in xmm2 and that in xmm3
pfpadd.pack.sp xmm1 = xmm2, xmm3	Add single-precision floating point values in xmm2 and those in xmm3
pfpadd.scalar.dp xmm1 = xmm2, xmm3	Add the low double-precision floating point value in xmm2 and that in xmm3
pfpadd.pack.dp xmm1 = xmm2, xmm3	Add double-precision floating point values in xmm2 and those in xmm3

PFPCMP - Packed floating point comparison

Instruction	Description
pfpcmp.lt.scalar.sp xmm1 = xmm2, xmm3	Compare the low single-precision floating point value in xmm2 to that in xmm3, and set the flags accordingly if less than
pfpcmp.lt.pack.sp xmm1 = xmm2, xmm3	Compare single-precision floating point values in xmm2 to those in xmm3, and set the flags accordingly if less than
pfpcmp.lt.scalar.dp xmm1 = xmm2, xmm3	Compare the low single-precision floating point value in xmm2 to that in xmm3, and set the flags accordingly if less than
pfpcmp.lt.pack.dp xmm1 = xmm2, xmm3	Compare double-precision floating point values in xmm2 to those in xmm3, and set the flags accordingly if less than
pfpcmp.le.scalar.sp xmm1 = xmm2, xmm3	Compare the low single-precision floating point value in xmm2 to that in xmm3, and set the flags accordingly if less than or equal to
pfpcmp.le.pack.sp xmm1 = xmm2, xmm3	Compare single-precision floating point values in xmm2 to those in xmm3, and set the flags accordingly if less than or equal to
pfpcmp.le.scalar.dp xmm1 = xmm2, xmm3	Compare the low single-precision floating point value in xmm2 to that in xmm3, and set the flags accordingly if less than or equal to
pfpcmp.le.pack.dp xmm1 = xmm2, xmm3	Compare double-precision floating point values in xmm2 to those in xmm3, and set the flags accordingly if less than or equal to
pfpcmp.le.scalar.sp xmm1 = xmm2, xmm3	Compare the low single-precision floating point value in xmm2 to that in xmm3, and set the flags accordingly if equal to
pfpcmp.le.pack.sp xmm1 = xmm2, xmm3	Compare single-precision floating point values in xmm2 to those in xmm3, and set the flags accordingly if equal to
pfpcmp.le.scalar.dp xmm1 = xmm2, xmm3	Compare the low single-precision floating point value in xmm2 to that in xmm3, and set the flags accordingly if equal to
pfpcmp.le.pack.dp xmm1 = xmm2, xmm3	Compare double-precision floating point values in xmm2 to those in xmm3, and set the flags accordingly if equal to
pfpcmp.ne.scalar.sp xmm1 = xmm2, xmm3	Compare the low single-precision floating point value in xmm2 to that in xmm3, and set the flags accordingly if not equal to
pfpcmp.ne.pack.sp xmm1 = xmm2, xmm3	Compare single-precision floating point values in xmm2 to those in xmm3, and set the flags accordingly if not equal to
pfpcmp.ne.scalar.dp xmm1 = xmm2, xmm3	Compare the low single-precision floating point value in xmm2 to that in xmm3, and set the flags accordingly if not equal to
pfpcmp.ne.pack.dp xmm1 = xmm2, xmm3	Compare double-precision floating point values in xmm2 to those in xmm3, and set the flags accordingly if not equal to
pfpcmp.unord.scalar.sp xmm1 = xmm2, xmm3	Compare the low single-precision floating point value in xmm2 to that in xmm3, and set the flags accordingly if unordered
pfpcmp.unord.pack.sp xmm1 = xmm2, xmm3	Compare single-precision floating point values in xmm2 to those in xmm3, and set the flags accordingly if unordered
pfpcmp.unord.scalar.dp xmm1 = xmm2, xmm3	Compare the low single-precision floating point value in xmm2 to that in xmm3, and set the flags accordingly if unordered
pfpcmp.unord.pack.dp xmm1 = xmm2, xmm3	Compare double-precision floating point values in xmm2 to those in xmm3, and set the flags accordingly if unordered

PFPDIV - Packed floating point divide

Instruction	Description
pfpdiv.scalar.sp xmm1 = xmm2, xmm3	Divide the low single-precision floating point value in xmm2 by that in xmm3
pfpdiv.pack.sp xmm1 = xmm2, xmm3	Divide single-precision floating point values in xmm2 by those in xmm3
pfpdiv.scalar.dp xmm1 = xmm2, xmm3	Divide the low double-precision floating point value in xmm2 by that in xmm3
pfpdiv.pack.dp xmm1 = xmm2, xmm3	Divide double-precision floating point values in xmm2 by those in xmm3

PFPFMA - Packed floating point multiply and add

Instruction	Description
pfpfma++.scalar.sp xmm1 += xmm2, xmm3	Do following operation on the low single-precision floating point number: $xmm1 = xmm1 + xmm2 \times xmm3$
pfpfma++.pack.sp xmm1 += xmm2, xmm3	Do following operations on four single-precision floating point numbers: $xmm1 = xmm1 + xmm2 \times xmm3$
pfpfma++.scalar.dp xmm1 += xmm2, xmm3	Do following operation on the low double-precision floating point number: $xmm1 = xmm1 + xmm2 \times xmm3$
pfpfma++.pack.dp xmm1 += xmm2, xmm3	Do following operations on two double-precision floating point numbers: $xmm1 = xmm1 + xmm2 \times xmm3$
pfpfma+-.scalar.sp xmm1 += xmm2, xmm3	Do following operation on the low single-precision floating point number: $xmm1 = xmm1 - xmm2 \times xmm3$
pfpfma+-.pack.sp xmm1 += xmm2, xmm3	Do following operations on four single-precision floating point numbers: $xmm1 = xmm1 - xmm2 \times xmm3$
pfpfma+-.scalar.dp xmm1 += xmm2, xmm3	Do following operation on the low double-precision floating point number: $xmm1 = xmm1 - xmm2 \times xmm3$
pfpfma+-.pack.dp xmm1 += xmm2, xmm3	Do following operations on two double-precision floating point numbers: $xmm1 = xmm1 - xmm2 \times xmm3$
pfpfma-+.scalar.sp xmm1 += xmm2, xmm3	Do following operation on the low single-precision floating point number: $xmm1 = -xmm1 + xmm2 \times xmm3$
pfpfma-+.pack.sp xmm1 += xmm2, xmm3	Do following operations on four single-precision floating point numbers: $xmm1 = -xmm1 + xmm2 \times xmm3$
pfpfma-+.scalar.dp xmm1 += xmm2, xmm3	Do following operation on the low double-precision floating point number: $xmm1 = -xmm1 + xmm2 \times xmm3$
pfpfma-+.pack.dp xmm1 += xmm2, xmm3	Do following operations on two double-precision floating point numbers: $xmm1 = -xmm1 + xmm2 \times xmm3$
pfpfma–.scalar.sp xmm1 += xmm2, xmm3	Do following operation on the low single-precision floating point number: $xmm1 = -xmm1 - xmm2 \times xmm3$
pfpfma–.pack.sp xmm1 += xmm2, xmm3	Do following operations on four single-precision floating point numbers: $xmm1 = -xmm1 - xmm2 \times xmm3$
pfpfma–.scalar.dp xmm1 += xmm2, xmm3	Do following operation on the low double-precision floating point number: $xmm1 = -xmm1 - xmm2 \times xmm3$
pfpfma–.pack.dp xmm1 += xmm2, xmm3	Do following operations on two double-precision floating point numbers: $xmm1 = -xmm1 - xmm2 \times xmm3$

PFPHADD - Packed floating point horizontal add

Instruction	Description
pfphadd.pack.sp xmm1 = xmm2, xmm3	Horizontal-add single-precision floating point values in xmm2 and those in xmm3
pfphadd.pack.dp xmm1 = xmm2, xmm3	Horizontal-add double-precision floating point values in xmm2 and those in xmm3

PFPMAX - Return maximum packed floating point values

Instruction	Description
pfpmax.scalar.sp xmm1 = xmm2, xmm3	Return the maximum scalar single-precision floating-point value between xmm2 and xmm3
pfpmax.pack.sp xmm1 = xmm2, xmm3	Return the maximum packed single-precision floating-point values between xmm2 and xmm3
pfpmax.scalar.dp xmm1 = xmm2, xmm3	Return the maximum scalar double-precision floating-point value between xmm2 and xmm3
pfpmax.pack.dp xmm1 = xmm2, xmm3	Return the maximum packed double-precision floating-point values between xmm2 and xmm3

PFPMIN - Return minimum packed floating point values

Instruction	Description
pfpmax.scalar.sp xmm1 = xmm2, xmm3	Return the minimum scalar single-precision floating-point value between xmm2 and xmm3
pfpmax.pack.sp xmm1 = xmm2, xmm3	Return the minimum packed single-precision floating-point values between xmm2 and xmm3
pfpmax.scalar.dp xmm1 = xmm2, xmm3	Return the minimum scalar double-precision floating-point value between xmm2 and xmm3
pfpmax.pack.dp xmm1 = xmm2, xmm3	Return the minimum packed double-precision floating-point values between xmm2 and xmm3

PFPMUL - Packed floating point multiply

Instruction	Description
pfpmul.scalar.sp xmm1 = xmm2, xmm3	Multiply the low single-precision floating point value in xmm2 and that in xmm3
pfpmul.pack.sp xmm1 = xmm2, xmm3	Multiply single-precision floating point values in xmm2 and those in xmm3
pfpmul.scalar.dp xmm1 = xmm2, xmm3	Multiply the low double-precision floating point value in xmm2 and that in xmm3
pfpmul.pack.dp xmm1 = xmm2, xmm3	Multiply double-precision floating point values in xmm2 and those in xmm3

PFPRCPSQRT - Packed floating point reciprocals of square roots

Instruction	Description
pfprcpsqrt.scalar.sp xmm1 = xmm2	Return the reciprocal of the square root of the low single-precision floating point value in xmm2
pfprcpsqrt.pack.sp xmm1 = xmm2	Return the reciprocals of the square roots of single-precision floating point values in xmm2
pfprcpsqrt.scalar.dp xmm1 = xmm2	Return the reciprocal of the square root of the low double-precision floating point value in xmm2
pfprcpsqrt.pack.dp xmm1 = xmm2	Return the reciprocals of the square roots of double-precision floating point values in xmm2

PFPSQRT - Packed floating point square roots

Instruction	Description
pfpsqrt.scalar.sp xmm1 = xmm2	Return the square root of the low single-precision floating point value in xmm2
pfpsqrt.pack.sp xmm1 = xmm2	Return the square roots of single-precision floating point values in xmm2
pfpsqrt.scalar.dp xmm1 = xmm2	Return the square root of the low double-precision floating point value in xmm2
pfpsqrt.pack.dp xmm1 = xmm2	Return the square roots of double-precision floating point values in xmm2

PFPSUB - Packed floating point sub

Instruction	Description
pfpsub.scalar.sp xmm1 = xmm2, xmm3	Subtract the low single-precision floating point value in xmm3 from that in xmm2
pfpsub.pack.sp xmm1 = xmm2, xmm3	Subtract single-precision floating point values in xmm3 from those in xmm2
pfpsub.scalar.dp xmm1 = xmm2, xmm3	Subtract the low double-precision floating point value in xmm3 from that in xmm2
pfpsub.pack.dp xmm1 = xmm2, xmm3	Subtract double-precision floating point values in xmm3 from those in xmm2

PINTADD - Packed integer add

Instruction	Description
pintadd1 xmm1 = xmm2, xmm3	Add sixteen 8-bit integer values in xmm2 and those in xmm3
pintadd2 xmm1 = xmm2, xmm3	Add eight 16-bit integer values in xmm2 and those in xmm3
pintadd4 xmm1 = xmm2, xmm3	Add four 32-bit integer values in xmm2 and those in xmm3
pintadd8 xmm1 = xmm2, xmm3	Add two 64-bit integer values in xmm2 and those in xmm3

PINTAND - Packed bitwise logical AND

Instruction	Description
pintand xmm1 = xmm2, xmm3	Bitwise logical AND of values in xmm2 and those in xmm3

PINTCMP - Packed integer comparison

Instruction	Description
pintcmp1.lt xmm1 = xmm2, xmm3	Compare sixteen 8-bit integer values in xmm2 to those in xmm3, and set the flags accordingly if less than
pintcmp2.lt xmm1 = xmm2, xmm3	Compare eight 16-bit integer values in xmm2 to those in xmm3, and set the flags accordingly if less than
pintcmp4.lt xmm1 = xmm2, xmm3	Compare four 32-bit integer values in xmm2 to those in xmm3, and set the flags accordingly if less than
pintcmp8.lt xmm1 = xmm2, xmm3	Compare two 64-bit integer values in xmm2 to those in xmm3, and set the flags accordingly if less than
pintcmp1.le xmm1 = xmm2, xmm3	Compare sixteen 8-bit integer values in xmm2 to those in xmm3, and set the flags accordingly if less than or equal to
pintcmp2.le xmm1 = xmm2, xmm3	Compare eight 16-bit integer values in xmm2 to those in xmm3, and set the flags accordingly if less than or equal to
pintcmp4.le xmm1 = xmm2, xmm3	Compare four 32-bit integer values in xmm2 to those in xmm3, and set the flags accordingly if less than or equal to
pintcmp8.le xmm1 = xmm2, xmm3	Compare two 64-bit integer values in xmm2 to those in xmm3, and set the flags accordingly if less than or equal to
pintcmp1.eq xmm1 = xmm2, xmm3	Compare sixteen 8-bit integer values in xmm2 to those in xmm3, and set the flags accordingly if equal to
pintcmp2.eq xmm1 = xmm2, xmm3	Compare eight 16-bit integer values in xmm2 to those in xmm3, and set the flags accordingly if equal to
pintcmp4.eq xmm1 = xmm2, xmm3	Compare four 32-bit integer values in xmm2 to those in xmm3, and set the flags accordingly if equal to
pintcmp8.eq xmm1 = xmm2, xmm3	Compare two 64-bit integer values in xmm2 to those in xmm3, and set the flags accordingly if equatl to
pintcmp1.ne xmm1 = xmm2, xmm3	Compare sixteen 8-bit integer values in xmm2 to those in xmm3, and set the flags accordingly if not equal to
pintcmp2.ne xmm1 = xmm2, xmm3	Compare eight 16-bit integer values in xmm2 to those in xmm3, and set the flags accordingly if not equal to
pintcmp4.ne xmm1 = xmm2, xmm3	Compare four 32-bit integer values in xmm2 to those in xmm3, and set the flags accordingly if not equal to
pintcmp8.ne xmm1 = xmm2, xmm3	Compare two 64-bit integer values in xmm2 to those in xmm3, and set the flags accordingly if not equatl to

PINTHADD - Packed integer horizontal add

Instruction	Description
pinthadd1 xmm1 = xmm2, xmm3	Horizontal-add sixteen 8-bit integer values in xmm2 and those in xmm3
pinthadd2 xmm1 = xmm2, xmm3	Horizontal-add eight 16-bit integer values in xmm2 and those in xmm3
pinthadd4 xmm1 = xmm2, xmm3	Horizontal-add four 32-bit integer values in xmm2 and those in xmm3
pinthadd8 xmm1 = xmm2, xmm3	Horizontal-add two 64-bit integer values in xmm2 and those in xmm3

PINTMUL - Packed integer multiply

Instruction	Description
pintmul4 xmm1 = xmm2, xmm3	Multiply four 32-bit integer values in xmm2 and those in xmm3

PINTNOT - Packed bitwise NOT

Instruction	Description
not xmm1 = xmm2	Reverse each bit of values in xmm2

PINTOR - Packed bitwise logical OR

Instruction	Description
pintor xmm1 = xmm2, xmm3	Bitwise logical OR of values in xmm2 and those in xmm3

PINTSAR - Packed integer arithmetic shift-right

Instruction	Description
pintsar1 xmm1 = xmm2, immed6	Arithmetically shift sixteen 8-bit integer values in xmm2 to right immed6 bits
pintsar2 xmm1 = xmm2, immed6	Arithmetically shift eight 16-bit integer values in xmm2 to right immed6 bits
pintsar4 xmm1 = xmm2, immed6	Arithmetically shift four 32-bit integer values in xmm2 to right immed6 bits
pintsar8 xmm1 = xmm2, immed6	Arithmetically shift two 64-bit integer values in xmm2 to right immed6 bits

PINTSHL - Packed integer logical shift-left

Instruction	Description
pintshl1 xmm1 = xmm2, immed6	Logically shift sixteen 8-bit integer values in xmm2 to left immed6 bits
pintshl2 xmm1 = xmm2, immed6	Logically shift eight 16-bit integer values in xmm2 to left immed6 bits
pintshl4 xmm1 = xmm2, immed6	Logically shift four 32-bit integer values in xmm2 to left immed6 bits
pintshl8 xmm1 = xmm2, immed6	Logically shift two 64-bit integer values in xmm2 to left immed6 bits

PINTSHR - Packed integer logical shift-right

Instruction	Description
pintshr1 xmm1 = xmm2, immed6	Logically shift sixteen 8-bit integer values in xmm2 to right immed6 bits
pintshr2 xmm1 = xmm2, immed6	Logically shift eight 16-bit integer values in xmm2 to right immed6 bits
pintshr4 xmm1 = xmm2, immed6	Logically shift four 32-bit integer values in xmm2 to right immed6 bits
pintshr8 xmm1 = xmm2, immed6	Logically shift two 64-bit integer values in xmm2 to right immed6 bits

PINTSUB - Packed integer substract

Instruction	Description
pintsub1 xmm1 = xmm2, xmm3	Subtract sixteen 8-bit integer values in xmm3 from those in xmm2
pintsub2 xmm1 = xmm2, xmm3	Subtract eight 16-bit integer values in xmm3 from those in xmm2
pintsub4 xmm1 = xmm2, xmm3	Subtract four 32-bit integer values in xmm3 from those in xmm2
pintsub8 xmm1 = xmm2, xmm3	Subtract two 64-bit integer values in xmm3 from those in xmm2

PINTXOR - Packed bitwise logical exclusive OR

Instruction	Description
pintxor xmm1 = xmm2, xmm3	Bitwise logical exclusive OR of values in xmm2 and those in xmm3

XFERXMM - Transfer XMM register value to a neighbor PE

Instruction	Description
xferxmm.n xmm1 = xmm2	Copy xmm2 to xmm1 of the northern neighbor PE (Northmost PE do nothing)
xferxmm.wrap.n xmm1 = xmm2	Copy xmm2 to xmm1 of the northern neighbor PE (Northmost PE copies xmm2 to xmm1 of southmost PE)
xferxmm.e xmm1 = xmm2	Copy xmm2 to xmm1 of the eastern neighbor PE (Eastmost PE do nothing)
xferxmm.wrap.e xmm1 = xmm2	Copy xmm2 to xmm1 of the eastern neighbor PE (Eastmost PE copies xmm2 to xmm1 of westmost PE)
xferxmm.w xmm1 = xmm2	Copy xmm2 to xmm1 of the western neighbor PE (Westmost PE do nothing)
xferxmm.wrap.w xmm1 = xmm2	Copy xmm2 to xmm1 of the western neighbor PE (Westmost PE copies xmm2 to xmm1 of eastmost PE)
xferxmm.s xmm1 = xmm2	Copy xmm2 to xmm1 of the southern neighbor PE (Southmost PE do nothing)
xferxmm.wrap.s xmm1 = xmm2	Copy xmm2 to xmm1 of the sourthern neighbor PE (Southmost PE copies xmm2 to xmm1 of northmost PE)

E.3 M-format POD Instructions

COPYBLK - Copy a block of data between the local memory and the system memory

Instruction	Description
copyblk sys[r1] = local[r2], r3	Copy a block (block size: r3) of data from the address r2 of the local memory to the address r1 of the system memory
copyblk strided sys[r1] = local[r2], r3	Copy blocks (block size: r3, the number of blocks: ar10, the block-stride of system memory space: ar11) of data from the address r2 of the local memory to the address r1 of the system memory
copyblk sys[r1] = strided local[r2], r3	Copy blocks (block size: r3, the number of blocks: ar10, the block-stride of local memory space: ar11) of data from the address r2 of the local memory to the address r1 of the system memory
copyblk local[r1] = sys[r2], r3	Copy a block (block size: r3) of data from the address r2 of the local memory to the address r1 of the system memory
copyblk strided local[r1] = sys[r2], r3	Copy blocks (block size: r3, the number of blocks: ar10, the block-stride of local memory space: ar11) of data from the address r2 of the system memory to the address r1 of the local memory
copyblk local[r1] = strided sys[r2], r3	Copy blocks (block size: r3, the number of blocks: ar10, the block-stride of system memory space: ar11) of data from the address r2 of the system memory to the address r1 of the local memory

LDLOCAL - Load data from the local memory to a general purpose register

Instruction	Description
ld1.zxt r1 = local[r2 + immed6]	Load 8-bit data from the address (r2 + immed6) of the local memory, zero-extend it, and save it at r1
ld1.sxt r1 = local[r2 + immed6]	Load 8-bit data from the address (r2 + immed6) of the local memory, sign-extend it, and save it at r1
ld2.zxt r1 = local[r2 + immed6]	Load 16-bit data from the address (r2 + immed6) of the local memory, zero-extend it, and save it at r1
ld2.sxt r1 = local[r2 + immed6]	Load 16-bit data from the address (r2 + immed6) of the local memory, sign-extend it, and save it at r1
ld4.zxt r1 = local[r2 + immed6]	Load 32-bit data from the address (r2 + immed6) of the local memory, zero-extend it, and save it at r1
ld4.sxt r1 = local[r2 + immed6]	Load 32-bit data from the address (r2 + immed6) of the local memory, sign-extend it, and save it at r1
ld8.zxt r1 = local[r2 + immed6]	Load 64-bit data from the address (r2 + immed6) of the local memory, zero-extend it, and save it at r1
ld8.sxt r1 = local[r2 + immed6]	Load 64-bit data from the address (r2 + immed6) of the local memory, sign-extend it, and save it at r1

LDLOCAL++ - Load data from the local memory to a general purpose register (post-increment type)

Instruction	Description
ld1++.zxt r1 = local[r2], r3	Load 8-bit data from the address r2 of the local memory, zero-extend it, save it at r1, and increase r2 by r3
ld1++.sxt r1 = local[r2], r3	Load 8-bit data from the address r2 of the local memory, sign-extend it, save it at r1, and increase r2 by r3
ld2++.zxt r1 = local[r2], r3	Load 16-bit data from the address r2 of the local memory, zero-extend it, save it at r1, and increase r2 by r3
ld2++.sxt r1 = local[r2], r3	Load 16-bit data from the address r2 of the local memory, sign-extend it, save it at r1, and increase r2 by r3
ld4++.zxt r1 = local[r2], r3	Load 32-bit data from the address r2 of the local memory, zero-extend it, save it at r1, and increase r2 by r3
ld4++.sxt r1 = local[r2], r3	Load 32-bit data from the address r2 of the local memory, sign-extend it, save it at r1, and increase r2 by r3
ld8++.zxt r1 = local[r2], r3	Load 64-bit data from the address r2 of the local memory, zero-extend it, save it at r1, and increase r2 by r3
ld8++.sxt r1 = local[r2], r3	Load 64-bit data from the address r2 of the local memory, sign-extend it, save it at r1, and increase r2 by r3

LDPODALL - Load data from the local memory of other PE to a general purpose register

Instruction	Description
ld1.zxt r1 = podall[r2 + immed6]	Load 8-bit data from the address (r2 + immed6)[19:0] of the local memory of ((r2+immed6)[27:24], (r2+immed6)[23:20]) PE, zero-extend it, and save it at r1
ld1.sxt r1 = podall[r2 + immed6]	Load 8-bit data from the address (r2 + immed6)[19:0] of the local memory of ((r2+immed6)[27:24], (r2+immed6)[23:20]) PE, sign-extend it, and save it at r1
ld2.zxt r1 = podall[r2 + immed6]	Load 16-bit data from the address (r2 + immed6)[19:0] of the local memory of ((r2+immed6)[27:24], (r2+immed6)[23:20]) PE, zero-extend it, and save it at r1
ld2.sxt r1 = podall[r2 + immed6]	Load 16-bit data from the address (r2 + immed6)[19:0] of the local memory of ((r2+immed6)[27:24], (r2+immed6)[23:20]) PE, sign-extend it, and save it at r1
ld4.zxt r1 = podall[r2 + immed6]	Load 32-bit data from the address (r2 + immed6)[19:0] of the local memory of ((r2+immed6)[27:24], (r2+immed6)[23:20]) PE, zero-extend it, and save it at r1
ld4.sxt r1 = podall[r2 + immed6]	Load 32-bit data from the address (r2 + immed6)[19:0] of the local memory of ((r2+immed6)[27:24], (r2+immed6)[23:20]) PE, sign-extend it, and save it at r1
ld8.zxt r1 = podall[r2 + immed6]	Load 64-bit data from the address (r2 + immed6)[19:0] of the local memory of ((r2+immed6)[27:24], (r2+immed6)[23:20]) PE, zero-extend it, and save it at r1
ld8.sxt r1 = podall[r2 + immed6]	Load 64-bit data from the address (r2 + immed6)[19:0] of the local memory of ((r2+immed6)[27:24], (r2+immed6)[23:20]) PE, sign-extend it, and save it at r1

LDPODALL++ - Load data from the local memory of other PE to a general purpose register (post-increment type)

Instruction	Description
ld1++.zxt r1 = podall[r2], r3	Load 8-bit data from the address r2[19:0] of the local memory of (r2[27:24], r2[23:20]) PE, zero-extend it, save it at r1, and increase r2 by r3
ld1++.sxt r1 = podall[r2], r3	Load 8-bit data from the address r2[19:0] of the local memory of (r2[27:24], r2[23:20]) PE, sign-extend it, save it at r1, and increase r2 by r3
ld2++.zxt r1 = podall[r2], r3	Load 16-bit data from the address r2[19:0] of the local memory of (r2[27:24], r2[23:20]) PE, zero-extend it, save it at r1, and increase r2 by r3
ld2++.sxt r1 = podall[r2], r3	Load 16-bit data from the address r2[19:0] of the local memory of (r2[27:24], r2[23:20]) PE, sign-extend it, save it at r1, and increase r2 by r3
ld4++.zxt r1 = podall[r2], r3	Load 32-bit data from the address r2[19:0] of the local memory of (r2[27:24], r2[23:20]) PE, zero-extend it, save it at r1, and increase r2 by r3
ld4++.sxt r1 = podall[r2], r3	Load 32-bit data from the address r2[19:0] of the local memory of (r2[27:24], r2[23:20]) PE, sign-extend it, save it at r1, and increase r2 by r3
ld8++.zxt r1 = podall[r2], r3	Load 64-bit data from the address r2[19:0] of the local memory of (r2[27:24], r2[23:20]) PE, zero-extend it, save it at r1, and increase r2 by r3
ld8++.sxt r1 = podall[r2], r3	Load 64-bit data from the address r2[19:0] of the local memory of (r2[27:24], r2[23:20]) PE, sign-extend it, save it at r1, and increase r2 by r3

LDPODCOL - Load data from the local memory of other PE in the same column to a general purpose register

Instruction	Description
ld1.zxt r1 = podcol[r2 + immed6]	Load 8-bit data from the address (r2 + immed6)[19:0] of the local memory of the PE in the (r2+immed6)[27:24]th row and in the same column, zero-extend it, and save it at r1
ld1.sxt r1 = podcol[r2 + immed6]	Load 8-bit data from the address (r2 + immed6)[19:0] of the local memory of the PE in the (r2+immed6)[27:24]th row and in the same column, sign-extend it, and save it at r1
ld2.zxt r1 = podcol[r2 + immed6]	Load 16-bit data from the address (r2 + immed6)[19:0] of the local memory of the PE in the (r2+immed6)[27:24]th row and in the same column, zero-extend it, and save it at r1
ld2.sxt r1 = podcol[r2 + immed6]	Load 16-bit data from the address (r2 + immed6)[19:0] of the local memory of the PE in the (r2+immed6)[27:24]th row and in the same column, sign-extend it, and save it at r1
ld4.zxt r1 = podcol[r2 + immed6]	Load 32-bit data from the address (r2 + immed6)[19:0] of the local memory of the PE in the (r2+immed6)[27:24]th row and in the same column, zero-extend it, and save it at r1
ld4.sxt r1 = podcol[r2 + immed6]	Load 32-bit data from the address (r2 + immed6)[19:0] of the local memory of the PE in the (r2+immed6)[27:24]th row and in the same column, sign-extend it, and save it at r1
ld8.zxt r1 = podcol[r2 + immed6]	Load 64-bit data from the address (r2 + immed6)[19:0] of the local memory of the PE in the (r2+immed6)[27:24]th row and in the same column, zero-extend it, and save it at r1
ld8.sxt r1 = podcol[r2 + immed6]	Load 64-bit data from the address (r2 + immed6)[19:0] of the local memory of the PE in the (r2+immed6)[27:24]th row and in the same column, sign-extend it, and save it at r1

LDPODCOL++ - Load data from the local memory of other PE in the same column to a general purpose register (post-increment type)

Instruction	Description
ld1++.zxt r1 = podcol[r2], r3	Load 8-bit data from the address r2[19:0] of the local memory of the PE in the r2[27:24]th row and in the same column, zero-extend it, save it at r1, and increase r2 by r3
ld1++.sxt r1 = podcol[r2], r3	Load 8-bit data from the address r2[19:0] of the local memory of the PE in the r2[27:24]th row and in the same column, sign-extend it, save it at r1, and increase r2 by r3
ld2++.zxt r1 = podcol[r2], r3	Load 16-bit data from the address r2[19:0] of the local memory of the PE in the r2[27:24]th row and in the same column, zero-extend it, save it at r1, and increase r2 by r3
ld2++.sxt r1 = podcol[r2], r3	Load 16-bit data from the address r2[19:0] of the local memory of the PE in the r2[27:24]th row and in the same column, sign-extend it, save it at r1, and increase r2 by r3
ld4++.zxt r1 = podcol[r2], r3	Load 32-bit data from the address r2[19:0] of the local memory of the PE in the r2[27:24]th row and in the same column, zero-extend it, save it at r1, and increase r2 by r3
ld4++.sxt r1 = podcol[r2], r3	Load 32-bit data from the address r2[19:0] of the local memory of the PE in the r2[27:24]th row and in the same column, sign-extend it, save it at r1, and increase r2 by r3
ld8++.zxt r1 = podcol[r2], r3	Load 64-bit data from the address r2[19:0] of the local memory of the PE in the r2[27:24]th row and in the same column, zero-extend it, save it at r1, and increase r2 by r3
ld8++.sxt r1 = podcol[r2], r3	Load 64-bit data from the address r2[19:0] of the local memory of the PE in the r2[27:24]th row and in the same column, sign-extend it, save it at r1, and increase r2 by r3

LDPODROW - Load data from the local memory of other PE in the same row to a general purpose register

Instruction	Description
ld1.zxt r1 = podrow[r2 + immed6]	Load 8-bit data from the address (r2 + immed6)[19:0] of the local memory of the PE in the same row and (r2+immed6)[23:20]th column, zero-extend it, and save it at r1
ld1.sxt r1 = podrow[r2 + immed6]	Load 8-bit data from the address (r2 + immed6)[19:0] of the local memory of the PE in the same row and (r2+immed6)[23:20]th column, sign-extend it, and save it at r1
ld2.zxt r1 = podrow[r2 + immed6]	Load 16-bit data from the address (r2 + immed6)[19:0] of the local memory of the PE in the same row and (r2+immed6)[23:20]th column, zero-extend it, and save it at r1
ld2.sxt r1 = podrow[r2 + immed6]	Load 16-bit data from the address (r2 + immed6)[19:0] of the local memory of the PE in the same row and (r2+immed6)[23:20]th column, sign-extend it, and save it at r1
ld4.zxt r1 = podrow[r2 + immed6]	Load 32-bit data from the address (r2 + immed6)[19:0] of the local memory of the PE in the same row and (r2+immed6)[23:20]th column, zero-extend it, and save it at r1
ld4.sxt r1 = podrow[r2 + immed6]	Load 32-bit data from the address (r2 + immed6)[19:0] of the local memory of the PE in the same row and (r2+immed6)[23:20]th column, sign-extend it, and save it at r1
ld8.zxt r1 = podrow[r2 + immed6]	Load 64-bit data from the address (r2 + immed6)[19:0] of the local memory of the PE in the same row and (r2+immed6)[23:20]th column, zero-extend it, and save it at r1
ld8.sxt r1 = podrow[r2 + immed6]	Load 64-bit data from the address (r2 + immed6)[19:0] of the local memory of the PE in the same row and (r2+immed6)[23:20]th column, sign-extend it, and save it at r1

LDPODROW++ - Load data from the local memory of other PE in the same row to a general purpose register (post-increment type)

Instruction	Description
ld1++.zxt r1 = podrow[r2], r3	Load 8-bit data from the address r2[19:0] of the local memory of the PE in the same row and r2[23:20]th column, zero-extend it, save it at r1, and increase r2 by r3
ld1++.sxt r1 = podrow[r2], r3	Load 8-bit data from the address r2[19:0] of the local memory of the PE in the same row and r2[23:20]th column, sign-extend it, save it at r1, and increase r2 by r3
ld2++.zxt r1 = podrow[r2], r3	Load 16-bit data from the address r2[19:0] of the local memory of the PE in the same row and r2[23:20]th column, zero-extend it, save it at r1, and increase r2 by r3
ld2++.sxt r1 = podrow[r2], r3	Load 16-bit data from the address r2[19:0] of the local memory of the PE in the same row and r2[23:20]th column, sign-extend it, save it at r1, and increase r2 by r3
ld4++.zxt r1 = podrow[r2], r3	Load 32-bit data from the address r2[19:0] of the local memory of the PE in the same row and r2[23:20]th column, zero-extend it, save it at r1, and increase r2 by r3
ld4++.sxt r1 = podrow[r2], r3	Load 32-bit data from the address r2[19:0] of the local memory of the PE in the same row and r2[23:20]th column, sign-extend it, save it at r1, and increase r2 by r3
ld8++.zxt r1 = podrow[r2], r3	Load 64-bit data from the address r2[19:0] of the local memory of the PE in the same row and r2[23:20]th column, zero-extend it, save it at r1, and increase r2 by r3
ld8++.sxt r1 = podrow[r2], r3	Load 64-bit data from the address r2[19:0] of the local memory of the PE in the same row and r2[23:20]th column, sign-extend it, save it at r1, and increase r2 by r3

LDSYS - Load data from the system memory to a general purpose register

Instruction	Description
ld1.zxt r1 = sys[r2 + immed6]	Load 8-bit data from the address (r2 + immed6) of the system memory, zero-extend it, and save it at r1
ld1.sxt r1 = sys[r2 + immed6]	Load 8-bit data from the address (r2 + immed6) of the system memory, sign-extend it, and save it at r1
ld2.zxt r1 = sys[r2 + immed6]	Load 16-bit data from the address (r2 + immed6) of the system memory, zero-extend it, and save it at r1
ld2.sxt r1 = sys[r2 + immed6]	Load 16-bit data from the address (r2 + immed6) of the system memory, sign-extend it, and save it at r1
ld4.zxt r1 = sys[r2 + immed6]	Load 32-bit data from the address (r2 + immed6) of the system memory, zero-extend it, and save it at r1
ld4.sxt r1 = sys[r2 + immed6]	Load 32-bit data from the address (r2 + immed6) of the system memory, sign-extend it, and save it at r1
ld8.zxt r1 = sys[r2 + immed6]	Load 64-bit data from the address (r2 + immed6) of the system memory, zero-extend it, and save it at r1
ld8.sxt r1 = sys[r2 + immed6]	Load 64-bit data from the address (r2 + immed6) of the system memory, sign-extend it, and save it at r1

LDSYS++ - Load data from the system memory to a general purpose register (post-increment type)

Instruction	Description
ld1++.zxt r1 = sys[r2], r3	Load 8-bit data from the address r2 of the system memory, zero-extend it, save it at r1, and increase r2 by r3
ld1++.sxt r1 = sys[r2], r3	Load 8-bit data from the address r2 of the system memory, sign-extend it, save it at r1, and increase r2 by r3
ld2++.zxt r1 = sys[r2], r3	Load 16-bit data from the address r2 of the system memory, zero-extend it, save it at r1, and increase r2 by r3
ld2++.sxt r1 = sys[r2], r3	Load 16-bit data from the address r2 of the system memory, sign-extend it, save it at r1, and increase r2 by r3
ld4++.zxt r1 = sys[r2], r3	Load 32-bit data from the address r2 of the system memory, zero-extend it, save it at r1, and increase r2 by r3
ld4++.sxt r1 = sys[r2], r3	Load 32-bit data from the address r2 of the system memory, sign-extend it, save it at r1, and increase r2 by r3
ld8++.zxt r1 = sys[r2], r3	Load 64-bit data from the address r2 of the system memory, zero-extend it, save it at r1, and increase r2 by r3
ld8++.sxt r1 = sys[r2], r3	Load 64-bit data from the address r2 of the system memory, sign-extend it, save it at r1, and increase r2 by r3

LDXMMLOCAL - Load data from the local memory to a XMM register

Instruction	Description
ldxmm1.scalar xmm1 = local[r2 + immed6]	Load 8-bit data from the address (r2 + immed6) of the local memory, and save it at xmm1
ldxmm2.scalar xmm1 = local[r2 + immed6]	Load 16-bit data from the address (r2 + immed6) of the local memory, and save it at xmm1
ldxmm4.scalar xmm1 = local[r2 + immed6]	Load 32-bit data from the address (r2 + immed6) of the local memory, and save it at xmm1
ldxmm8.scalar xmm1 = local[r2 + immed6]	Load 64-bit data from the address (r2 + immed6) of the local memory, and save it at xmm1
ldxmm.pack r1 = local[r2 + immed6]	Load 128-bit data from the address (r2 + immed6) of the local memory, and save it at xmm1

LDXMMLOCAL++ - Load data from the local memory to a XMM register (post-increment type)

Instruction	Description
ldxmm1++.scalar xmm1 = local[r2], r3	Load 8-bit data from the address r2 of the local memory, save it at xmm1, and increase r2 by r3
ldxmm2++.scalar xmm1 = local[r2], r3	Load 16-bit data from the address r2 of the local memory, save it at xmm1, and increase r2 by r3
ldxmm4++.scalar xmm1 = local[r2], r3	Load 32-bit data from the address r2 of the local memory, save it at xmm1, and increase r2 by r3
ldxmm8++.scalar xmm1 = local[r2], r3	Load 64-bit data from the address r2 of the local memory, save it at xmm1, and increase r2 by r3
ldxmm++.pack xmm1 = local[r2], r3	Load 64-bit data from the address r2 of the local memory, save it at xmm1, and increase r2 by r3

LDXMMPODALL - Load data from the local memory of other PE to a XMM register

Instruction	Description
ldxmm1.scalar xmm1 = podall[r2 + immed6]	Load 8-bit data from the address (r2 + immed6)[19:0] of the local memory of ((r2+immed6)[27:24], (r2+immed6)[23:20]) PE, and save it at xmm1
ldxmm2.scalar xmm1 = podall[r2 + immed6]	Load 16-bit data from the address (r2 + immed6)[19:0] of the local memory of ((r2+immed6)[27:24], (r2+immed6)[23:20]) PE, and save it at xmm1
ldxmm4.scalar xmm1 = podall[r2 + immed6]	Load 32-bit data from the address (r2 + immed6)[19:0] of the local memory of ((r2+immed6)[27:24], (r2+immed6)[23:20]) PE, and save it at xmm1
ldxmm8.scalar xmm1 = podall[r2 + immed6]	Load 64-bit data from the address (r2 + immed6)[19:0] of the local memory of ((r2+immed6)[27:24], (r2+immed6)[23:20]) PE, and save it at xmm1
ldxmm.pack xmm1 = podall[r2 + immed6]	Load 64-bit data from the address (r2 + immed6)[19:0] of the local memory of ((r2+immed6)[27:24], (r2+immed6)[23:20]) PE, and save it at xmm1

LDXMMPODALL++ - Load data from the local memory of other PE to a XMM register (post-increment type)

Instruction	Description
ldxmm1++.scalar xmm1 = podall[r2], r3	Load 8-bit data from the address r2[19:0] of the local memory of (r2[27:24], r2[23:20]) PE, save it at xmm1, and increase r2 by r3
ldxmm2++.scalar xmm1 = podall[r2], r3	Load 16-bit data from the address r2[19:0] of the local memory of (r2[27:24], r2[23:20]) PE, save it at xmm1, and increase r2 by r3
ldxmm4++.scalar xmm1 = podall[r2], r3	Load 32-bit data from the address r2[19:0] of the local memory of (r2[27:24], r2[23:20]) PE, save it at xmm1, and increase r2 by r3
ldxmm8++.scalar xmm1 = podall[r2], r3	Load 64-bit data from the address r2[19:0] of the local memory of (r2[27:24], r2[23:20]) PE, save it at xmm1, and increase r2 by r3
ldxmm++.pack xmm1 = podall[r2], r3	Load 64-bit data from the address r2[19:0] of the local memory of (r2[27:24], r2[23:20]) PE, save it at xmm1, and increase r2 by r3

LDXMMPODCOL - Load data from the local memory of other PE in the same column to a XMM register

Instruction	Description
ldxmm1.scalar xmm1 = podcol[r2 + immed6]	Load 8-bit data from the address (r2 + immed6)[19:0] of the local memory of the PE in the (r2+immed6)[27:24]th row and in the same column, and save it at xmm1
ldxmm2.scalar xmm1 = podcol[r2 + immed6]	Load 16-bit data from the address (r2 + immed6)[19:0] of the local memory of the PE in the (r2+immed6)[27:24]th row and in the same column, and save it at xmm1
ldxmm4.scalar xmm1 = podcol[r2 + immed6]	Load 32-bit data from the address (r2 + immed6)[19:0] of the local memory of the PE in the (r2+immed6)[27:24]th row and in the same column, and save it at xmm1
ldxmm8.scalar xmm1 = podcol[r2 + immed6]	Load 64-bit data from the address (r2 + immed6)[19:0] of the local memory of the PE in the (r2+immed6)[27:24]th row and in the same column, and save it at xmm1
ldxmm.pack xmm1 = podcol[r2 + immed6]	Load 64-bit data from the address (r2 + immed6)[19:0] of the local memory of the PE in the (r2+immed6)[27:24]th row and in the same column, and save it at xmm1

LDXMMPODCOL++ - Load data from the local memory of other PE in the same column to a XMM register (post-increment type)

Instruction	Description
ldxmm1++.scalar xmm1 = podcol[r2], r3	Load 8-bit data from the address r2[19:0] of the local memory of the PE in the r2[27:24]th row and in the same column, save it at xmm1, and increase r2 by r3
ldxmm2++.scalar xmm1 = podcol[r2], r3	Load 16-bit data from the address r2[19:0] of the local memory of the PE in the r2[27:24]th row and in the same column, save it at xmm1, and increase r2 by r3
ldxmm4++.scalar xmm1 = podcol[r2], r3	Load 32-bit data from the address r2[19:0] of the local memory of the PE in the r2[27:24]th row and in the same column, save it at xmm1, and increase r2 by r3
ldxmm8++.scalar xmm1 = podcol[r2], r3	Load 64-bit data from the address r2[19:0] of the local memory of the PE in the r2[27:24]th row and in the same column, save it at xmm1, and increase r2 by r3
ldxmm++.pack xmm1 = podcol[r2], r3	Load 64-bit data from the address r2[19:0] of the local memory of the PE in the r2[27:24]th row and in the same column, save it at xmm1, and increase r2 by r3

LDXMMPODROW - Load data from the local memory of other PE in the same row to a XMM register

Instruction	Description
ldxmm1.scalar xmm1 = podrow[r2 + immed6]	Load 8-bit data from the address (r2 + immed6)[19:0] of the local memory of the PE in the same row and (r2+immed6)[23:20]th column, and save it at xmm1
ldxmm2.scalar xmm1 = podrow[r2 + immed6]	Load 16-bit data from the address (r2 + immed6)[19:0] of the local memory of the PE in the same row and (r2+immed6)[23:20]th column, and save it at xmm1
ldxmm4.scalar xmm1 = podrow[r2 + immed6]	Load 32-bit data from the address (r2 + immed6)[19:0] of the local memory of the PE in the same row and (r2+immed6)[23:20]th column, and save it at xmm1
ldxmm8.scalar xmm1 = podrow[r2 + immed6]	Load 64-bit data from the address (r2 + immed6)[19:0] of the local memory of the PE in the same row and (r2+immed6)[23:20]th column, and save it at xmm1
ldxmm.pack xmm1 = podrow[r2 + immed6]	Load 64-bit data from the address (r2 + immed6)[19:0] of the local memory of the PE in the same row and (r2+immed6)[23:20]th column, and save it at xmm1

LDXMMPODROW++ - Load data from the local memory of other PE in the same row to a XMM register (post-increment type)

Instruction	Description
ldxmm1++.scalar xmm1 = podrow[r2], r3	Load 8-bit data from the address r2[19:0] of the local memory of the PE in the same row and r2[23:20]th column, save it at xmm1, and increase r2 by r3
ldxmm2++.scalar xmm1 = podrow[r2], r3	Load 16-bit data from the address r2[19:0] of the local memory of the PE in the same row and r2[23:20]th column, save it at xmm1, and increase r2 by r3
ldxmm4++.scalar xmm1 = podrow[r2], r3	Load 32-bit data from the address r2[19:0] of the local memory of the PE in the same row and r2[23:20]th column, save it at xmm1, and increase r2 by r3
ldxmm8++.scalar xmm1 = podrow[r2], r3	Load 64-bit data from the address r2[19:0] of the local memory of the PE in the same row and r2[23:20]th column, save it at xmm1, and increase r2 by r3
ldxmm++.pack xmm1 = podrow[r2], r3	Load 64-bit data from the address r2[19:0] of the local memory of the PE in the same row and r2[23:20]th column, save it at xmm1, and increase r2 by r3

LDXMMSYS - Load data from the system memory to a XMM register

Instruction	Description
ldxmm1.scalar xmm1 = sys[r2 + immed6]	Load 8-bit data from the address (r2 + immed6) of the system memory, and save it at xmm1
ldxmm2.scalar xmm1 = sys[r2 + immed6]	Load 16-bit data from the address (r2 + immed6) of the system memory, and save it at xmm1
ldxmm4.scalar xmm1 = sys[r2 + immed6]	Load 32-bit data from the address (r2 + immed6) of the system memory, and save it at xmm1
ldxmm8.scalar xmm1 = sys[r2 + immed6]	Load 64-bit data from the address (r2 + immed6) of the system memory, and save it at xmm1
ldxmm.pack xmm1 = sys[r2 + immed6]	Load 128-bit data from the address (r2 + immed6) of the system memory, and save it at xmm1

LDXMMSYS++ - Load data from the system memory to a XMM register (post-increment type)

Instruction	Description
ldxmm1++.scalar xmm1 = sys[r2], r3	Load 8-bit data from the address r2 of the system memory, save it at xmm1, and increase r2 by r3
ldxmm2++.scalar xmm1 = sys[r2], r3	Load 16-bit data from the address r2 of the system memory, save it at xmm1, and increase r2 by r3
ldxmm4++.scalar xmm1 = sys[r2], r3	Load 32-bit data from the address r2 of the system memory, save it at xmm1, and increase r2 by r3
ldxmm8++.scalar xmm1 = sys[r2], r3	Load 64-bit data from the address r2 of the system memory, save it at xmm1, and increase r2 by r3
ldxmm++.pack xmm1 = sys[r2], r3	Load 64-bit data from the address r2 of the system memory, save it at xmm1, and increase r2 by r3

MOVAR - Get or set a AR register value

Instruction	Description
mov8 r1 = ar2	Copy a register value of ar2 to r1
mov8 ar1 = r2	Copy a register value of r2 to ar1

POPMASK - Shift the mask register to left 1 bit

Instruction	Description
popmask	Shift the mask register to left 1 bit

PUSHMASK - Shift the mask register to right 1 bit and set top mask

Instruction	Description
pushmask.and.o	Shift the mask register to right 1 bit, and set top mask if the previous mask was set and if overflow
pushmask.and.not.o	Shift the mask register to right 1 bit, and set top mask if the previous mask was set and if not overflow
pushmask.and.b	Shift the mask register to right 1 bit, and set top mask if the previous mask was set and if below
pushmask.and.not.b	Shift the mask register to right 1 bit, and set top mask if the previous mask was set and if not below
pushmask.and.e	Shift the mask register to right 1 bit, and set top mask if the previous mask was set and if equal
pushmask.and.not.e	Shift the mask register to right 1 bit, and set top mask if the previous mask was set and if not equal
pushmask.and.be	Shift the mask register to right 1 bit, and set top mask if the previous mask was set and if below or equal
pushmask.and.not.be	Shift the mask register to right 1 bit, and set top mask if the previous mask was set and if not below or equal
pushmask.and.s	Shift the mask register to right 1 bit, and set top mask if the previous mask was set and if sign
pushmask.and.not.s	Shift the mask register to right 1 bit, and set top mask if the previous mask was set and if not sign
pushmask.and.l	Shift the mask register to right 1 bit, and set top mask if the previous mask was set and if less
pushmask.and.not.l	Shift the mask register to right 1 bit, and set top mask if the previous mask was set and if not less
pushmask.and.le	Shift the mask register to right 1 bit, and set top mask if the previous mask was set and if less or equal
pushmask.and.not.le	Shift the mask register to right 1 bit, and set top mask if the previous mask was set and if not less or equal
pushmask.or.o	Shift the mask register to right 1 bit, and set top mask if the previous mask was set or if overflow
pushmask.or.not.o	Shift the mask register to right 1 bit, and set top mask if the previous mask was set or if not overflow
pushmask.or.b	Shift the mask register to right 1 bit, and set top mask if the previous mask was set or if below
pushmask.or.not.b	Shift the mask register to right 1 bit, and set top mask if the previous mask was set or if not below
pushmask.or.e	Shift the mask register to right 1 bit, and set top mask if the previous mask was set or if equal
pushmask.or.not.e	Shift the mask register to right 1 bit, and set top mask if the previous mask was set or if not equal
pushmask.or.be	Shift the mask register to right 1 bit, and set top mask if the previous mask was set or if below or equal
pushmask.or.not.be	Shift the mask register to right 1 bit, and set top mask if the previous mask was set or if not below or equal
pushmask.or.s	Shift the mask register to right 1 bit, and set top mask if the previous mask was set or if sign
pushmask.or.not.s	Shift the mask register to right 1 bit, and set top mask if the previous mask was set or if not sign
pushmask.or.l	Shift the mask register to right 1 bit, and set top mask if the previous mask was set or if less
pushmask.or.not.l	Shift the mask register to right 1 bit, and set top mask if the previous mask was set or if not less
pushmask.or.le	Shift the mask register to right 1 bit, and set top mask if the previous mask was set or if less or equal
pushmask.or.not.le	Shift the mask register to right 1 bit, and set top mask if the previous mask was set or if not less or equal

SETTOPMASK - Set top mask

Instruction	Description
settopmask.and.o	Set top mask if the previous mask was set and if overflow
settopmask.and.not.o	Set top mask if the previous mask was set and if not overflow
settopmask.and.b	Set top mask if the previous mask was set and if below
settopmask.and.not.b	Set top mask if the previous mask was set and if not below
settopmask.and.e	Set top mask if the previous mask was set and if equal
settopmask.and.not.e	Set top mask if the previous mask was set and if not equal
settopmask.and.be	Set top mask if the previous mask was set and if below or equal
settopmask.and.not.be	Set top mask if the previous mask was set and if not below or equal
settopmask.and.s	Set top mask if the previous mask was set and if sign
settopmask.and.not.s	Set top mask if the previous mask was set and if not sign
settopmask.and.l	Set top mask if the previous mask was set and if less
settopmask.and.not.l	Set top mask if the previous mask was set and if not less
settopmask.and.le	Set top mask if the previous mask was set and if less or equal
settopmask.and.not.le	Set top mask if the previous mask was set and if not less or equal
settopmask.or.o	Set top mask if the previous mask was set or if overflow
settopmask.or.not.o	Set top mask if the previous mask was set or if not overflow
settopmask.or.b	Set top mask if the previous mask was set or if below
settopmask.or.not.b	Set top mask if the previous mask was set or if not below
settopmask.or.e	Set top mask if the previous mask was set or if equal
settopmask.or.not.e	Set top mask if the previous mask was set or if not equal
settopmask.or.be	Set top mask if the previous mask was set or if below or equal
settopmask.or.not.be	Set top mask if the previous mask was set or if not below or equal
settopmask.or.s	Set top mask if the previous mask was set or if sign
settopmask.or.not.s	Set top mask if the previous mask was set or if not sign
settopmask.or.l	Set top mask if the previous mask was set or if less
settopmask.or.not.l	Set top mask if the previous mask was set or if not less
settopmask.or.le	Set top mask if the previous mask was set or if less or equal
settopmask.or.not.le	Set top mask if the previous mask was set or if not less or equal

STLOCAL - Store a general purpose register value to the local memory

Instruction	Description
st1 local[r2 + immed6] = r1	Store 8-bit data of r1 to the address (r2 + immed6) of the local memory
st2 local[r2 + immed6] = r1	Load 16-bit data of r1 to the address (r2 + immed6) of the local memory
st4 local[r2 + immed6] = r1	Load 32-bit data of r1 to the address (r2 + immed6) of the local memory
st8 local[r2 + immed6] = r1	Load 64-bit data of r1 to the address (r2 + immed6) of the local memory

STLOCAL++ - Store a general purpose register value to the local memory (post-increment type)

Instruction	Description
st1++ local[r2] = r1, r3	Store 8-bit data of r1 to the address r2 of the local memory, and increase r2 by r3
st2++ local[r2] = r1, r3	Load 16-bit data of r1 to the address r2 of the local memory, and increase r2 by r3
st4++ local[r2] = r1, r3	Load 32-bit data of r1 to the address r2 of the local memory, and increase r2 by r3
st8++ local[r2] = r1, r3	Load 64-bit data of r1 to the address r2 of the local memory, and increase r2 by r3

STPODALL - Store a general purpose register value to the local memory of other PE

Instruction	Description
st1 podall[r2 + immed6] = r1	Store 8-bit data of r1 to the address (r2 + immed6)[19:0] of the local memory of ((r2+immed6)[27:24], (r2+immed6)[23:20]) PE
st2 podall[r2 + immed6] = r1	Load 16-bit data of r1 to the address (r2 + immed6)[19:0] of the local memory of ((r2+immed6)[27:24], (r2+immed6)[23:20]) PE
st4 podall[r2 + immed6] = r1	Load 32-bit data of r1 to the address (r2 + immed6)[19:0] of the local memory of ((r2+immed6)[27:24], (r2+immed6)[23:20]) PE
st8 podall[r2 + immed6] = r1	Load 64-bit data of r1 to the address (r2 + immed6)[19:0] of the local memory of ((r2+immed6)[27:24], (r2+immed6)[23:20]) PE

STPODALL++ - Store a general purpose register value to the local memory of other PE (post-increment type)

Instruction	Description
st1++ podall[r2] = r1, r3	Store 8-bit data of r1 to the address r2[19:0] of the local memory of (r2[27:24], r2[23:20]) PE, and increase r2 by r3
st2++ podall[r2] = r1, r3	Load 16-bit data of r1 to the address r2[19:0] of the local memory of (r2[27:24], r2[23:20]) PE, and increase r2 by r3
st4++ podall[r2] = r1, r3	Load 32-bit data of r1 to the address r2[19:0] of the local memory of (r2[27:24], r2[23:20]) PE, and increase r2 by r3
st8++ podall[r2] = r1, r3	Load 64-bit data of r1 to the address r2[19:0] of the local memory of (r2[27:24], r2[23:20]) PE, and increase r2 by r3

STPODCOL - Store a general purpose register value to the local memory of other PE in the same column

Instruction	Description
st1 podcol[r2 + immed6] = r1	Store 8-bit data of r1 to the address (r2 + immed6)[19:0] of the local memory of the PE in the (r2+immed6)[27:24]th row and in the same column
st2 podcol[r2 + immed6] = r1	Load 16-bit data of r1 to the address (r2 + immed6)[19:0] of the local memory of the PE in the (r2+immed6)[27:24]th row and in the same column
st4 podcol[r2 + immed6] = r1	Load 32-bit data of r1 to the address (r2 + immed6)[19:0] of the local memory of the PE in the (r2+immed6)[27:24]th row and in the same column
st8 podcol[r2 + immed6] = r1	Load 64-bit data of r1 to the address (r2 + immed6)[19:0] of the local memory of the PE in the (r2+immed6)[27:24]th row and in the same column

STPODCOL++ - Store a general purpose register value to the local memory of other PE in the same column (post-increment type)

Instruction	Description
st1++ podcol[r2] = r1, r3	Store 8-bit data of r1 to the address r2[19:0] of the local memory of the PE in the r2[27:24]th row and in the same column
st2++ podcol[r2] = r1, r3	Load 16-bit data of r1 to the address r2[19:0] of the local memory of the PE in the r2[27:24]th row and in the same column
st4++ podcol[r2] = r1, r3	Load 32-bit data of r1 to the address r2[19:0] of the local memory of the PE in the r2[27:24]th row and in the same column
st8++ podcol[r2] = r1, r3	Load 64-bit data of r1 to the address r2[19:0] of the local memory of the PE in the r2[27:24]th row and in the same column

STPODROW - Store a general purpose register value to the local memory of other PE in the same row

Instruction	Description
st1 podrow[r2 + immed6] = r1	Store 8-bit data of r1 to the address (r2 + immed6)[19:0] of the local memory of the PE in the same row and in the (r2+immed6)[27:24]th column
st2 podrow[r2 + immed6] = r1	Load 16-bit data of r1 to the address (r2 + immed6)[19:0] of the local memory of the PE in the same row and in the (r2+immed6)[27:24]th column
st4 podrow[r2 + immed6] = r1	Load 32-bit data of r1 to the address (r2 + immed6)[19:0] of the local memory of the PE in the same row and in the (r2+immed6)[27:24]th column
st8 podrow[r2 + immed6] = r1	Load 64-bit data of r1 to the address (r2 + immed6)[19:0] of the local memory of the PE in the same row and in the (r2+immed6)[27:24]th column

STPODROW++ - Store a general purpose register value to the local memory of other PE in the same row (post-increment type)

Instruction	Description
st1++ podrow[r2] = r1, r3	Store 8-bit data of r1 to the address r2[19:0] of the local memory of the PE in the same row and in the r2[27:24]th column
st2++ podrow[r2] = r1, r3	Load 16-bit data of r1 to the address r2[19:0] of the local memory of the PE in the same row and in the r2[27:24]th column
st4++ podrow[r2] = r1, r3	Load 32-bit data of r1 to the address r2[19:0] of the local memory of the PE in the same row and in the r2[27:24]th column
st8++ podrow[r2] = r1, r3	Load 64-bit data of r1 to the address r2[19:0] of the local memory of the PE in the same row and in the r2[27:24]th column

STSYS - Store a general purpose register value to the system memory

Instruction	Description
st1 sys[r2 + immed6] = r1	Store 8-bit data of r1 to the address (r2 + immed6) of the system memory
st2 sys[r2 + immed6] = r1	Load 16-bit data of r1 to the address (r2 + immed6) of the system memory
st4 sys[r2 + immed6] = r1	Load 32-bit data of r1 to the address (r2 + immed6) of the system memory
st8 sys[r2 + immed6] = r1	Load 64-bit data of r1 to the address (r2 + immed6) of the system memory

STSYS++ - Store a general purpose register value to the system memory (post-increment type)

Instruction	Description
st1++ sys[r2] = r1, r3	Store 8-bit data of r1 to the address r2 of the system memory, and increase r2 by r3
st2++ sys[r2] = r1, r3	Load 16-bit data of r1 to the address r2 of the system memory, and increase r2 by r3
st4++ sys[r2] = r1, r3	Load 32-bit data of r1 to the address r2 of the system memory, and increase r2 by r3
st8++ sys[r2] = r1, r3	Load 64-bit data of r1 to the address r2 of the system memory, and increase r2 by r3

STXMMLOCAL - Store a XMM register value to the local memory

Instruction	Description
stxmm1.scalar local[r2 + immed6] = xmm1	Store 8-bit data of xmm1 to the address (r2 + immed6) of the local memory
stxmm2.scalar local[r2 + immed6] = xmm1	Load 16-bit data of xmm1 to the address (r2 + immed6) of the local memory
stxmm4.scalar local[r2 + immed6] = xmm1	Load 32-bit data of xmm1 to the address (r2 + immed6) of the local memory
stxmm8.scalar local[r2 + immed6] = xmm1	Load 64-bit data of xmm1 to the address (r2 + immed6) of the local memory
stxmm.pack local[r2 + immed6] = xmm1	Load 128-bit data of xmm1 to the address (r2 + immed6) of the local memory

STXMMLOCAL++ - Store a XMM register value to the local memory (post-increment type)

Instruction	Description
stxmm1++.scalar local[r2] = xmm1, r3	Store 8-bit data of xmm1 to the address r2 of the local memory, and increase r2 by r3
stxmm2++.scalar local[r2] = xmm1, r3	Load 16-bit data of xmm1 to the address r2 of the local memory, and increase r2 by r3
stxmm4++.scalar local[r2] = xmm1, r3	Load 32-bit data of xmm1 to the address r2 of the local memory, and increase r2 by r3
stxmm8++.scalar local[r2] = xmm1, r3	Load 64-bit data of xmm1 to the address r2 of the local memory, and increase r2 by r3
stxmm++.pack local[r2] = xmm1, r3	Load 128-bit data of xmm1 to the address r2 of the local memory, and increase r2 by r3

STXMMPODALL - Store a XMM register value to the local memory of other PE

Instruction	Description
stxmm1.scalar podall[r2 + immed6] = xmm1	Store 8-bit data of xmm1 to the address (r2 + immed6)[19:0] of the local memory of ((r2+immed6)[27:24], (r2+immed6)[23:20]) PE
stxmm2.scalar podall[r2 + immed6] = xmm1	Load 16-bit data of xmm1 to the address (r2 + immed6)[19:0] of the local memory of ((r2+immed6)[27:24], (r2+immed6)[23:20]) PE
stxmm4.scalar podall[r2 + immed6] = xmm1	Load 32-bit data of xmm1 to the address (r2 + immed6)[19:0] of the local memory of ((r2+immed6)[27:24], (r2+immed6)[23:20]) PE
stxmm8.scalar podall[r2 + immed6] = xmm1	Load 64-bit data of xmm1 to the address (r2 + immed6)[19:0] of the local memory of ((r2+immed6)[27:24], (r2+immed6)[23:20]) PE
stxmm.pack podall[r2 + immed6] = xmm1	Load 128-bit data of xmm1 to the address (r2 + immed6)[19:0] of the local memory of ((r2+immed6)[27:24], (r2+immed6)[23:20]) PE

STXMMPODALL++ - Store a XMM register value to the local memory of other PE (post-increment type)

Instruction	Description
stxmm1++.scalar podall[r2] = xmm1, r3	Store 8-bit data of xmm1 to the address r2[19:0] of the local memory of (r2[27:24], r2[23:20]) PE, and increase r2 by r3
stxmm2++.scalar podall[r2] = xmm1, r3	Load 16-bit data of xmm1 to the address r2[19:0] of the local memory of (r2[27:24], r2[23:20]) PE, and increase r2 by r3
stxmm4++.scalar podall[r2] = xmm1, r3	Load 32-bit data of xmm1 to the address r2[19:0] of the local memory of (r2[27:24], r2[23:20]) PE, and increase r2 by r3
stxmm8++.scalar podall[r2] = xmm1, r3	Load 64-bit data of xmm1 to the address r2[19:0] of the local memory of (r2[27:24], r2[23:20]) PE, and increase r2 by r3
stxmm++.pack podall[r2] = xmm1, r3	Load 128-bit data of xmm1 to the address r2[19:0] of the local memory of (r2[27:24], r2[23:20]) PE, and increase r2 by r3

STXMMPODCOL - Store a XMM register value to the local memory of other PE in the same column

Instruction	Description
stxmm1.scalar podcol[r2 + immed6] = xmm1	Store 8-bit data of xmm1 to the address (r2 + immed6)[19:0] of the local memory of the PE in the (r2+immed6)[27:24]th row and in the same column
stxmm2.scalar podcol[r2 + immed6] = xmm1	Load 16-bit data of xmm1 to the address (r2 + immed6)[19:0] of the local memory of the PE in the (r2+immed6)[27:24]th row and in the same column
stxmm4.scalar podcol[r2 + immed6] = xmm1	Load 32-bit data of xmm1 to the address (r2 + immed6)[19:0] of the local memory of the PE in the (r2+immed6)[27:24]th row and in the same column
stxmm8.scalar podcol[r2 + immed6] = xmm1	Load 64-bit data of xmm1 to the address (r2 + immed6)[19:0] of the local memory of the PE in the (r2+immed6)[27:24]th row and in the same column
stxmm.pack podcol[r2 + immed6] = xmm1	Load 128-bit data of xmm1 to the address (r2 + immed6)[19:0] of the local memory of the PE in the (r2+immed6)[27:24]th row and in the same column

STXMMPODCOL++ - Store a XMM register value to the local memory of other PE in the same column (post-increment type)

Instruction	Description
stxmm1++.scalar podcol[r2] = xmm1, r3	Store 8-bit data of xmm1 to the address r2[19:0] of the local memory of the PE in the r2[27:24]th row and in the same column
stxmm2++.scalar podcol[r2] = xmm1, r3	Load 16-bit data of xmm1 to the address r2[19:0] of the local memory of the PE in the r2[27:24]th row and in the same column
stxmm4++.scalar podcol[r2] = xmm1, r3	Load 32-bit data of xmm1 to the address r2[19:0] of the local memory of the PE in the r2[27:24]th row and in the same column
stxmm8++.scalar podcol[r2] = xmm1, r3	Load 64-bit data of xmm1 to the address r2[19:0] of the local memory of the PE in the r2[27:24]th row and in the same column
stxmm++.pack podcol[r2] = xmm1, r3	Load 128-bit data of xmm1 to the address r2[19:0] of the local memory of the PE in the r2[27:24]th row and in the same column

STXMMPODROW - Store a XMM register value to the local memory of other PE in the same row

Instruction	Description
stxmm1.scalar podrow[r2 + immed6] = r1	Store 8-bit data of r1 to the address (r2 + immed6)[19:0] of the local memory of the PE in the same row and in the (r2+immed6)[27:24]th column
stxmm2.scalar podrow[r2 + immed6] = r1	Load 16-bit data of r1 to the address (r2 + immed6)[19:0] of the local memory of the PE in the same row and in the (r2+immed6)[27:24]th column
stxmm4.scalar podrow[r2 + immed6] = r1	Load 32-bit data of r1 to the address (r2 + immed6)[19:0] of the local memory of the PE in the same row and in the (r2+immed6)[27:24]th column
stxmm8.scalar podrow[r2 + immed6] = r1	Load 64-bit data of r1 to the address (r2 + immed6)[19:0] of the local memory of the PE in the same row and in the (r2+immed6)[27:24]th column
stxmm.pack podrow[r2 + immed6] = r1	Load 128-bit data of r1 to the address (r2 + immed6)[19:0] of the local memory of the PE in the same row and in the (r2+immed6)[27:24]th column

STXMMPODROW++ - Store a XMM register value to the local memory of other PE in the same row (post-increment type)

Instruction	Description
stxmm1++.scalar podrow[r2] = xmm1, r3	Store 8-bit data of xmm1 to the address r2[19:0] of the local memory of the PE in the same row and in the r2[27:24]th column
stxmm2++.scalar podrow[r2] = xmm1, r3	Load 16-bit data of xmm1 to the address r2[19:0] of the local memory of the PE in the same row and in the r2[27:24]th column
stxmm4++.scalar podrow[r2] = xmm1, r3	Load 32-bit data of xmm1 to the address r2[19:0] of the local memory of the PE in the same row and in the r2[27:24]th column
stxmm8++.scalar podrow[r2] = xmm1, r3	Load 64-bit data of xmm1 to the address r2[19:0] of the local memory of the PE in the same row and in the r2[27:24]th column
stxmm++.pack podrow[r2] = xmm1, r3	Load 128-bit data of xmm1 to the address r2[19:0] of the local memory of the PE in the same row and in the r2[27:24]th column

STXMMSYS - Store a XMM register value to the system memory

Instruction	Description
stxmm1.scalar sys[r2 + immed6] = xmm1	Store 8-bit data of xmm1 to the address (r2 + immed6) of the system memory
stxmm2.scalar sys[r2 + immed6] = xmm1	Load 16-bit data of xmm1 to the address (r2 + immed6) of the system memory
stxmm4.scalar sys[r2 + immed6] = xmm1	Load 32-bit data of xmm1 to the address (r2 + immed6) of the system memory
stxmm8.scalar sys[r2 + immed6] = xmm1	Load 64-bit data of xmm1 to the address (r2 + immed6) of the system memory
stxmm.pack sys[r2 + immed6] = xmm1	Load 128-bit data of xmm1 to the address (r2 + immed6) of the system memory

STXMMSYS++ - Store a XMM register value to the system memory (post-increment type)

Instruction	Description
stxmm1++.scalar sys[r2] = xmm1, r3	Store 8-bit data of xmm1 to the address r2 of the system memory, and increase r2 by r3
stxmm2++.scalar sys[r2] = xmm1, r3	Load 16-bit data of xmm1 to the address r2 of the system memory, and increase r2 by r3
stxmm4++.scalar sys[r2] = xmm1, r3	Load 32-bit data of xmm1 to the address r2 of the system memory, and increase r2 by r3
stxmm8++.scalar sys[r2] = xmm1, r3	Load 64-bit data of xmm1 to the address r2 of the system memory, and increase r2 by r3
stxmm++.pack sys[r2] = xmm1, r3	Load 128-bit data of xmm1 to the address r2 of the system memory, and increase r2 by r3

XFERBLK - Transfer a block of data from the local memory to the local memory of a neighbor PE

Instruction	Description
xferblk.n nn[r1] = local[r2], r3	Copy a block of data (block size: r3) from the local memory space, starting from the addresss r1, to the local memory space, starting from the address r2, of the northern neighbor PE (Northmost PE copies to the southmost PE)
xferblk.e nn[r1] = local[r2], r3	Copy a block of data (block size: r3) from the local memory space, starting from the addresss r1, to the local memory space, starting from the address r2, of the eastern neighbor PE (Eastmost PE copies to the westmost PE)
xferblk.w nn[r1] = local[r2], r3	Copy a block of data (block size: r3) from the local memory space, starting from the addresss r1, to the local memory space, starting from the address r2, of the western neighbor PE (Westmost PE copies to the eastmost PE)
xferblk.s nn[r1] = local[r2], r3	Copy a block of data (block size: r3) from the local memory space, starting from the addresss r1, to the local memory space, starting from the address r2, of the southern neighbor PE (Southmost PE copies to the northmost PE)
xferblk.n strided nn[r1] = local[r2], r3	Copy blocks of data (block size: r3, the number of blocks: ar10, the block-stride of neighbor's memory space: ar11) from the local memory space, starting from the addresss r1, to the local memory space, starting from the address r2, of the northern neighbor PE (Northmost PE copies to the southmost PE)
xferblk.e strided nn[r1] = local[r2], r3	Copy blocks of data (block size: r3, the number of blocks: ar10, the block-stride of neighbor's memory space: ar11) from the local memory space, starting from the addresss r1, to the local memory space, starting from the address r2, of the eastern neighbor PE (Eastmost PE copies to the westmost PE)
xferblk.w strided nn[r1] = local[r2], r3	Copy blocks of data (block size: r3, the number of blocks: ar10, the block-stride of neighbor's memory space: ar11) from the local memory space, starting from the addresss r1, to the local memory space, starting from the address r2, of the western neighbor PE (Westmost PE copies to the eastmost PE)
xferblk.s strided nn[r1] = local[r2], r3	Copy blocks of data (block size: r3, the number of blocks: ar10, the block-stride of neighbor's memory space: ar11) from the local memory space, starting from the addresss r1, to the local memory space, starting from the address r2, of the southern neighbor PE (Southmost PE copies to the northmost PE)
xferblk.n nn[r1] = strided local[r2], r3	Copy blocks of data (block size: r3, the number of blocks: ar10, the block-stride of local memory space: ar11) from the local memory space, starting from the addresss r1, to the local memory space, starting from the address r2, of the northern neighbor PE (Northmost PE copies to the southmost PE)
xferblk.e nn[r1] = strided local[r2], r3	Copy blocks of data (block size: r3, the number of blocks: ar10, the block-stride of local memory space: ar11) from the local memory space, starting from the addresss r1, to the local memory space, starting from the address r2, of the eastern neighbor PE (Eastmost PE copies to the westmost PE)
xferblk.w nn[r1] = strided local[r2], r3	Copy blocks of data (block size: r3, the number of blocks: ar10, the block-stride of local memory space: ar11) from the local memory space, starting from the addresss r1, to the local memory space, starting from the address r2, of the western neighbor PE (Westmost PE copies to the eastmost PE)
xferblk.s nn[r1] = strided local[r2], r3	Copy blocks of data (block size: r3, the number of blocks: ar10, the block-stride of local memory space: ar11) from the local memory space, starting from the addresss r1, to the local memory space, starting from the address r2, of the southern neighbor PE (Southmost PE copies to the northmost PE)

E.4 L-format POD Instructions

MOVL - Move a 64-bit immediate value to a general purpose register

Instruction	Description
movl r1 = immed64	Move a 64-bit immediate value to r1. This instruction consumes entire G/X/M slots.