

(12) INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

(19) World Intellectual Property
Organization
International Bureau



(43) International Publication Date
28 April 2005 (28.04.2005)

PCT

(10) International Publication Number
WO 2005/037326 A2

(51) International Patent Classification⁷: **A61K 51/04**,
A61P 35/00, A61K 31/16

Bristol BS16 1WR (GB). **DAY, Timothy** [GB/GB]; 33
Rosebery Avenue, St. Werburghs, Bristol BS2 9TW (GB).

(21) International Application Number:
PCT/GB2004/004377

(74) **Agent: VIGARS, Christopher, Ian**; Haseltine Lake, Red-
cliff Quay, 120 Redcliff Street, Bristol BS1 6HU (GB).

(22) International Filing Date: 13 October 2004 (13.10.2004)

(81) **Designated States** (*unless otherwise indicated, for every
kind of national protection available*): AE, AG, AL, AM,
AT, AU, AZ, BA, BB, BG, BR, BW, BY, BZ, CA, CH, CN,
CO, CR, CU, CZ, DE, DK, DM, DZ, EC, EE, EG, ES, FI,
GB, GD, GE, GH, GM, HR, HU, ID, IL, IN, IS, JP, KE,
KG, KP, KR, KZ, LC, LK, LR, LS, LT, LU, LV, MA, MD,
MG, MK, MN, MW, MX, MZ, NA, NI, NO, NZ, OM, PG,
PH, PL, PT, RO, RU, SC, SD, SE, SG, SK, SL, SY, TJ, TM,
TN, TR, TT, TZ, UA, UG, US, UZ, VC, VN, YU, ZA, ZM,
ZW.

(25) Filing Language: English

(26) Publication Language: English

(30) **Priority Data:**
0323950.6 13 October 2003 (13.10.2003) GB
0409815.8 30 April 2004 (30.04.2004) GB

(84) **Designated States** (*unless otherwise indicated, for every
kind of regional protection available*): ARIPO (BW, GH,
GM, KE, LS, MW, MZ, NA, SD, SL, SZ, TZ, UG, ZM,
ZW), Eurasian (AM, AZ, BY, KG, KZ, MD, RU, TJ, TM),
European (AT, BE, BG, CH, CY, CZ, DE, DK, EE, ES, FI,
FR, GB, GR, HU, IE, IT, LU, MC, NL, PL, PT, RO, SE, SI,
SK, TR), OAPI (BF, BJ, CF, CG, CI, CM, GA, GN, GQ,
GW, ML, MR, NE, SN, TD, TG).

(71) **Applicant** (*for all designated States except US*): **CLEAR-
SPEED TECHNOLOGY PLC** [GB/GB]; 3110 Great
Western Court, Hunts Ground Road, Stoke Gifford, Bristol
BS34 8HP (GB).

(72) **Inventors; and**

(75) **Inventors/Applicants** (*for US only*): **STUTTARD, David**
[GB/GB]; 28 St Georges Avenue, St George, Bristol
BS5 8DD (GB). **WILLIAMS, David** [GB/GB]; 20
Cousins Way, Emersons Green, Bristol BS16 7DG (GB).
PACKER, James [GB/GB]; 43 Whitesfield Road, Nailsea,
Bristol BS48 2DY (GB). **DAVIDSON, Colin** [GB/GB]; 46
Magdelene Street, Glastonbury, Somerset BA6 9EJ (GB).
HICKEY, Neil [GB/GB]; 34 Casson Drive, Stoke Park,

Published:

— *without international search report and to be republished
upon receipt of that report*

*For two-letter codes and other abbreviations, refer to the "Guid-
ance Notes on Codes and Abbreviations" appearing at the begin-
ning of each regular issue of the PCT Gazette.*

(54) **Title:** UNIFIED SIMD PROCESSOR

(57) **Abstract:** A data processor comprises a single control unit and an integral execution unit adapted to operate on both parallel and scalar data under the same instruction set. The execution unit comprises a SMO part comprising an array of processing elements adapted to operate on the parallel data and a scalar part adapted to operate on the scalar data. Substantially all the instructions are operable on the parallel and/or the scalar data. Preferably, both the scalar part and the SIMD part have a complete range of addressing modes that allow them to support: scalar variables pointing to data in scalar memory; scalar variables pointing to data in memory in the SIMD part of the execution unit; SHVM variables pointing to memory in the SIMD part of the execution unit; and SIMD variables pointing to data in scalar memory. The scalar part, the SIMD part, and I/O means are adapted to be operated in parallel by the same instruction stream.



WO 2005/037326 A2

Unified SIMD Processor

Background to the Invention

The invention consists of a significant improvement to a well known type of parallel processor. Single-Instruction Multiple-Data (SIMD) architectures are widely used in a number of computing applications. They have the advantage of high performance, low power consumption and easy scalability. A SIMD processor differs from the more commonly used multi-processor system (where a number of processors, each independently executing their own program code, are used together to solve a problem) in that it executes a single stream of instructions but each instruction operates on multiple data items in parallel.

There are a range of SIMD processor architectures. These range from an *intelligent memory* type of approach where each *column* of a memory array has a simple 1-bit Arithmetic Logic Unit (ALU) through to more complex architectures where a, usually smaller, array of more complex processing elements are fed with a single instruction stream. They have, however, been difficult to program in the past.

In all these cases the array of SIMD processing elements are fed with instructions and data by a dedicated controller. Often the controller is the largest and most complex part of the design because of the need to match the data processing rate achieved by the SIMD array. The array controller is not itself 'intelligent' and is, in turn, controlled by a general purpose processor which also runs the bulk of the application software. This *host* processor will execute most of the application code; it will send data to be processed to the SIMD array with an indication of the function to be performed, typically a pointer to the start of a sequence of SIMD array instructions or microinstructions. An exemplary prior art SIMD processor is outlined in Figure 1.

This introduces two programming problems. Firstly, the array controller can only be programmed at a very low level: a series of microcode, or possibly assembler, instructions to be executed by the array. This is because the array controller is not designed to work with a compiler. Secondly, this programming is completely separated from the main application code running on the host processor. This may be 'wrapped up' by providing C functions or C++ classes which makes them appear more integrated and reduces the complexity of passing control between the host processor and the SIMD array. Also, in the simplest form, the host processor and the SIMD array will have completely separate memory spaces for code and data. Any of

the standard shared memory techniques can be used to address this with all the resulting problems: complex arbitration, cache coherency issues, contention for memory bandwidth, inter-processor synchronization, greater cost, etc.

A preferable solution is to provide a fully integrated architecture where the same processor can transparently process both *scalar* and SIMD operations in a fully integrated fashion. This means that a single program can be written, in a standard high-level language such as C, which includes all parts of the application – both ‘normal’ (scalar) data and parallel data processing.

Prior Art

The basic SIMD processing model has been in use for many years. Some of the earliest examples are the ILLIAC IV developed in 1972, which had 64 processors with floating point and memory, and Goodyear’s STARAN (1975) which was a 1-bit architecture. Many variants have been developed since then.

There is also a recent trend to put small-scale SIMD extensions in standard processor architectures. This is exemplified by the Intel MMX and SSE extensions to the x86 architecture. This allows, for example, the values in two 32-bit registers to be added as if each register contained four independent 8-bit values. This is not considered relevant to the current discussion for a number of reasons:

- the SIMD processing is done on a very small scale;
- only a limited range of SIMD functionality is provided;
- there are distinct instructions for operating on SIMD data;
- the same hardware is used for SIMD and for scalar operations (although separate registers may be provided for the SIMD operands)

The idea of providing indexed addressing modes in a SIMD processor is disclosed in our copending patent application GB9908225.7. This invention extends that idea by using a more flexible implementation of that idea in combination with a more powerful array controller to provide a unified processor architecture.

Summary of the Invention

Applicant’s approach, after many years of experience with developing systems based on SIMD coprocessors, was to create a new architecture where instructions are decoded and despatched to the appropriate part of the execution unit: either the scalar (mono) execution unit or the SIMD (poly) execution unit. Instruction execution on these are decoupled so that, for example, a mono instruction followed by a poly instruction will execute concurrently. The support for mono and poly data is, as far as

possible, identical – this makes it practical to write a compiler to target the processor. This greatly simplifies the task of programming the processor.

This means a single program runs on the processor which does both mono and poly operations as required – typically it will do both at the same time. A single C program
5 can be written which manages program control flow, operates on mono data, and operates on poly data.

To support this, and further simplify programming, this unity of the architecture is reflected in the instruction set and programming model. The processing elements in the SIMD array and the mono execution unit have a common instruction set. For
10 instance, an **add** instruction can be used with operands which are either poly registers or mono registers (or immediate values); the appropriate execution unit will execute the instruction. Mono and poly operands can also be mixed to allow, for example, a single mono value to be added to a poly register which holds a different value on every processing element.

15 It is appropriate to note that this does not require the mono execution unit to have exactly the same micro-architecture as the poly execution unit. For example, the poly processing elements could have an 8-bit ALU, while the mono execution unit has a 32-bit ALU.

A block schematic diagram of a processor in accordance with the invention is shown
20 in Figure 2.

The most significant features contributing to the novelty and inventiveness of this invention

include the following concepts:

- The processor provides a unified programming model: a single program runs on
25 a processor providing both scalar and SIMD operations.
- This is supported with a unified instruction set: e.g. an **add** instruction can work with either mono or poly operands (or a mixture of the two)
- Poly operations can freely use mono registers in expressions, and more importantly for addressing memory.
- 30 • Having a full range of addressing modes and completely separate memory blocks in each PE make it simple to support poly data structures and poly pointers on the SIMD array (previous SIMD architectures either provide a single address to all PEs or use sequences of instructions to construct an address in memory).

- Mono and poly operations are of equal standing: wherever possible, all instructions can be used with either mono or poly operands (there are some exceptions because a small set of operations, mainly for control, only make sense for one or the other).
 - 5 • The mono and poly execution units are loosely coupled so that their operation can be overlapped. Compilers are extremely good at scheduling instructions to take advantage of this flexibility - maximizing the efficiency of the system.
 - There are some compiler optimizations specific to this architecture: for example it may be better, in some instances, to move an operation that would normally
10 be done on the mono execution unit on to the poly execution unit - either because it can then be overlapped with other operations on the mono execution unit or because the result is needed on the poly execution unit anyway. Again, the compiler can detect such cases and make the appropriate optimizations.
 - A common code and data space is available to both mono and poly operations
15 making it easy to generate code for, and share data between, the two execution units.
 - The overall architecture is suitable for targeting by a high level language compiler - again for both mono and poly operations.
- 20 To this end, the invention provides, in a first aspect a data processor comprising a single control unit and an integral execution unit adapted to operate on both parallel and scalar data under the same instruction set.
- According to a second aspect, the invention provides a data processor comprising a single control unit and an integral execution unit adapted to operate on both parallel
25 and scalar data under the same instruction set, and wherein the execution unit comprises a SIMD part comprising an array of processing elements adapted to operate on said parallel data and a scalar part adapted to operate on said scalar data.
- According to a third aspect, the invention provides a data processor comprising a single control unit and an integral execution unit adapted to operate on both parallel
30 and scalar data under the same instruction set, and wherein substantially all said instructions are operable on said parallel and/or said scalar data.
- According to a fourth aspect, the invention provides a data processor comprising a single control unit and an integral execution unit adapted to operate on both parallel and scalar data under the same instruction set, and wherein the execution unit

comprises a SIMD part comprising an array of processing elements adapted to operate on said parallel data and a scalar part adapted to operate on said scalar data, and wherein substantially all said instructions are operable on said parallel and/or said scalar data.

- 5 The single control unit and the integral execution unit may be adapted to operate asynchronously on both the parallel and scalar data under the same instruction set.

The execution unit preferably comprises a SIMD part comprising an array of processing elements adapted to operate on the parallel data and a scalar part adapted to operate on the scalar data.

- 10 Preferably, the SIMD part and the scalar part differ only in that each processing element of the array contains local memory. The SIMD part and the scalar part preferably have a common memory area for instructions and scalar data. Each PE may contain an ALU and one or more multiply-accumulate function units or an ALU and one or more floating point function units. Preferably, each PE has multiple
15 enable bits to support nested conditional code.

The SIMD and the scalar execution unit conveniently have a similar set of status/result flags which can be used to control branching on the scalar part of the execution unit and conditional execution on the SIMD part of the execution unit in a similar manner.

- 20 There is preferably a communication path between PEs in the SIMD part and means are preferably provided for data to be transferred to and from the scalar part of the execution unit and the SIMD part of the execution unit.

Advantageously the control unit is adapted to fetch and execute multiple instruction threads simultaneously. A semaphore unit may use semaphores to synchronize

- 25 between instruction threads. The semaphores may be used to synchronize between I/O operations and instruction threads. The control unit may be adapted to schedule and execute threads based on their priority and the state of any semaphores they are waiting on and may include a set of the semaphores for use in synchronizing threads.

- Each processing element of the SIMD array may comprise one or more function units,
30 a register file, said local memory, and I/O means.

Both the scalar part and the SIMD part preferably have a complete range of addressing modes that allow them to support: scalar variables pointing to data in scalar memory; scalar variables pointing to data in memory in the SIMD part of the

execution unit; SIMD variables pointing to memory in the SIMD part of the execution unit; and SIMD variables pointing to data in scalar memory.

Substantially all the instructions are preferably operable on either the parallel data or the scalar data or a combination of both the scalar data and the parallel data.

- 5 The processor may further comprise a compiler adapted to produce code from a single source program to operate on both the scalar and the parallel data. The compiler is preferably adapted to detect parallel data automatically.

The scalar part, the SIMD part, and the I/O means are preferably adapted to be operated in parallel by the same instruction stream.

- 10 A programmer or the compiler is adapted to use the multi-threaded execution to schedule I/O operations such that they occur concurrently with computation and data is always available when required.

- The I/O means may comprise programmed I/O means adapted to enable each PE to transfer data to and from external scalar memory and wherein each PE provides the
15 address it wants to read data from or write data to. The memory accesses may be consolidated to minimize the number of memory transfers when multiple PEs are accessing the same memory area.

- Each PE is preferably adapted to transfer data to and from external scalar memory, in which case the address for each PE is generated automatically based on the PE
20 number and the amount of data being transferred.

- The I/O means may comprise streaming I/O means which distributes an incoming data stream to all the PEs in the SIMD part and collects data from all the PEs to generate an output stream independently of program execution on the SIMD part of the execution unit. The streaming I/O means may only use enabled PEs to take part in
25 the streaming I/O operation. The size of data distributed to, and collected from each PE may be different for each PE.

Brief Description of the Drawings

The invention will be described with reference to the following drawings, in which:

- Figure 1 shows a prior art SIMD processor;
30 Figure 2 shows the architecture in accordance with the present invention;
Figure 3 is a comparison between RISC and MTAP architecture;
Figure 4 is a schematic diagram of an MTAP execution unit;
Figure 5 indicates schematically the execution of an instruction;
Figure 6 is a graph indicating the power efficiency of the MTAP architecture;

Figure 7 is a block schematic diagram depicting an exemplary MTAP processor; and

Figure 8 is a block diagram of a specific implementation of the MTAP architecture as applied to an evaluation chip.

5 Detailed Description of the Illustrated Embodiments

The design of this processor architecture was driven by the desire to make it practical and efficient to program in a high level language. In order to make this possible a number of features are required. First, the overall programming model needs to be simple and 'regular' – i.e. the same operations and addressing modes can be applied to
10 all data types.

The execution unit for mono data is based on well known principles used in many RISC processors. This makes it simple for a compiler to target and easy for a user to understand.

The functionality supported by each PE is then made as similar as possible to the
15 mono execution unit: each PE has an ALU, a register file and memory, and supports a range of addressing modes for transferring data between memory and registers.

It is not necessary for the PEs to be identical to the mono execution unit in every detail: e.g. the number of registers and the width and complexity of the ALU are likely to be different in practice. The poly execution unit gets its performance largely
20 from the number of PEs brought to bear on the problem rather than the processing power of the individual PEs. To balance this, the mono execution unit is likely to require a wider and more complex ALU. This is reasonable because this is only instantiated once while the PEs are replicated many times and need to be efficiently implemented.

The number of clock cycles required to execute an instruction on the mono and poly
25 execution units is highly variable. Both will execute some instructions in a single cycle. But some instructions could take several cycles on the mono execution unit or on the poly execution unit. It is therefore essential that the two execution units are only loosely coupled rather than operating in lock-step; this allows them to overlap
30 their execution of instructions. The hardware will synchronize their operations when necessary: e.g. when they need to access a shared resource, or one execution unit requires data from the other.

Transfers between the registers and memory, and other I/O transfers, can take place concurrently with ALU operations. To simplify the task for the compiler, these

interactions are all interlocked using register scoreboarding (see our copending patent application GB 9908203.4).

An important aspect of the PE architecture to support compiled code is the existence of a full set of addressing modes: direct, indirect and indexed. Indirect and indexed
5 addressing by the PEs is described in the previously mentioned copending patent application GB 9908225.7.

Reference may also be had to our copending patent application GB 0321186.9 for details of the "ClearConnect" bus referred to in the context of the present invention.

In a high level language compiler for this architecture, it is only necessary to
10 introduce a simple modification; the keyword 'poly' used in a declaration identifies data that is distributed across the SIMD array. Similar methods have been used before, for example the 'shape' keyword used in C*, a language used for programming the "Connection Machine". However, it is only with the present architecture that poly variables become 'first class' objects – i.e. they can be used everywhere that 'normal'
15 mono data can be: in expressions, as function arguments, in conditional statements, etc. Pointers to both mono and poly data are allowed, as are pointers of both mono and poly types. Furthermore, poly variables can be mixed quite freely with mono data in all these contexts. This provides a very simple and regular programming model for the user.

20 Note that it would also be possible for a compiler to detect the parallelism in the program or data automatically: This is frequently done with 'vectorizing' Fortran compilers used with supercomputers, for example. The same techniques could be applied to this architecture. The architecture will now be described in more detail.

The Multi-Threaded Array Processing (MTAP) architecture has been developed to
25 address a number of problems in high performance, high data rate processing.

A number of applications are characterized by very high data rates, flexible processing requirements and hard real-time constraints. We use the term *data flow processing* to describe this class of problem.

The MTAP processor delivers on the three primary requirements for data flow
30 applications:

1. It directly addresses the data bandwidth, and has a clear scalability path for future requirements.

2. It provides the raw horsepower for the processing functions required, on the maximum data rate that the system will encounter. That processing power can scale with increasing function demands.
3. It stores the data close to the processing core to maximize bandwidth and minimize latency.

MTAP Overview

The MTAP architecture defines a family of embedded processors with parallel data processing capability. Figure 3 compares a standard processor and the MTAP architecture. As can be seen, the MTAP processor has a standard, RISC-like, control unit with instruction fetch, caches and I/O mechanisms. This is coupled to a highly parallel execution unit which provides the performance and scalability of the MTAP architecture.

The processor is designed in a highly modular fashion which allows many details of a specific implementation to be easily customized for the target application.

To simplify the integration of the processor into a variety of systems, the processor can be also configured to be big or little-endian.

Control unit

The control unit fetches, decodes and dispatches instructions to the execution units. The processor executes a fairly standard, three operand instruction set. The manner in which the same instruction stream is decoded and issued in parallel to the mono part, the poly part and the I/O part is schematically indicated in Figure 5.

The control unit also provides hardware support for multi-threaded execution, allowing fast swapping between multiple threads. The threads are prioritized and are intended primarily to support efficient overlap of I/O and compute. This can be used to hide the latency of external data accesses.

The MTAP processor can also include instruction and data caches to minimize the latency of memory accesses. The size and type of these is a configurable option. Alternatively, the caches may be replaced with SRAM to provide a complete embedded solution.

The control unit also includes a control port which is used for initializing and debugging the processor; it includes support for breakpoints, single stepping and the examination of internal state.

Execution unit

The execution unit consists of a number of *processing elements* (PEs). This allows it to process data elements in parallel. Each PE consists of an ALU, registers, memory and I/O.

- 5 The number of PEs in a processor core is a configurable parameter which allows performance to be scaled to meet the needs of the application (see Figure 6). The execution unit can be thought of as two, largely independent, parts (see Figure 4).

One PE forms the *mono* execution unit; this is dedicated to processing mono (i.e. scalar or non-parallel) data. The mono execution unit also handles program flow control such as branching and thread switching. The rest of the PEs form the *poly* execution unit which processes parallel (poly) data.

The poly execution unit may consist of tens, hundreds or even thousands of PEs. This array of PEs operates in a synchronous manner, similar to SIMD, where every PE executes the same instruction on its piece of data. Each PE also has its own independent local memory; this provides fast access to the data being processed. For example, one PE at 400 MHz has a memory bandwidth of 1.6 Gbytes/s. An array with 256 such PEs has an aggregate bandwidth of over 400 Gbytes/s, with single cycle latency. The number of registers and the amount of memory in each PE are configurable options.

20 ***Input / output***

The MTAP processor core has two basic I/O mechanisms. The first, *Programmed I/O* (PIO), is the normal mechanism used for accessing memory external to the MTAP core: it supports random accesses to variable sized data by the PEs. The second mechanism is *Streaming I/O* (SIO) which allows chunks of contiguous data to be streamed directly into the memory within PEs.

Each of these supports a variety of addressing modes, described in more detail later.

Programming model

From a programmer's perspective, the MTAP processor appears as a single processor running a single C program. This is very different from some other parallel processing models where the programmer has to explicitly program multiple independent processors, or can only access the processor via function calls or some other indirect mechanism.

The MTAP processor executes a single instruction stream; each instruction is sent to one of the functional units: this may be the mono or poly execution unit or one of the

I/O controllers. The processor can despatch an instruction on every cycle.

For multi-cycle instructions, the operation of the functional units can be overlapped. So, for example, an I/O operation can be started with one instruction and on the next cycle the mono execution unit could start a multiply instruction (which requires
 5 several cycles to execute). While these operations are proceeding, the poly execution unit can continue to execute instructions.

The main change from programming a standard processor is the concept of operating on parallel data. Data to be processed is assigned to variables which have an instance on every PE and are operated on in parallel: we call this *poly* data. This can be
 10 thought of as a data vector which is distributed across the array.

Variables only requiring a single instance (e.g. loop control variables) are known as *mono* variables – they behave exactly like normal variables on a sequential processor. Applicant provides a compiler which uses a simple extension to standard C to identify data which is to be processed in parallel. The new keyword **poly** is used in a
 15 declaration to define data which exists, and is processed, on every PE in the poly execution unit.

Example

To give a feel for the way the MTAP processor is programmed, the following represents a fragment of code which will calculate 64 values of a sine function across
 20 the PE array in a single operation.

```
#include <fnext.h>
#include <math.h>
#define PI 3.1415926535897932384
#define NUMBER_OF_PES 64
25
poly float angle, sine;
poly int pe;

/* get PE number: 0...n-1 */
30 pe = get_penum();
/* convert to an angle in range 0 to Pi */
angle = pe * PI / NUMBER_OF_PES;
/* calculate sine of angle on each PE */
sine = sinp(angle);
```

This code uses the library function `get_penum()` to get a unique value in the variable `pe` on each PE. This is scaled to give a range of values for `angle` between 0 and π across the PEs.

Finally, the library function `sinp()` is called to calculate the sine of these values on all PEs simultaneously. The `sinp` function is the poly equivalent of the standard sine function; it takes a poly argument and returns the appropriate value on every PE.

Performance

The performance of an MTAP core with 64 PEs, on a 0.13 μ process and running at 400 MHz is shown in Table 1. The columns show the performance achieved with the basic PE, the PE with an additional Multiply-Accumulate (MAC) unit, and the PE with an FPU extension.

Table 1 - Performance Comparison

	Base Architecture	8-bit MAC	FPU
MIPS	25,600	25,600	25,600
MMACS (x 10 ⁶)	—	25,600	25,600
MFLOPS	—	—	51,200
Memory Bandwidth (Gbytes / s)	100	100	100
Inter-PE Bandwidth (GBytes / s)	50	50	50
Relative Area	1.0	1.1	1.3

However, the core can be scaled well beyond this number of PEs, to 256 or even thousands of PEs. It also scales down efficiently to lower numbers of PEs, or lower clock rates, enabling low cost and low power systems to be built.

Figure 6 shows how performance and power scale with the number of PEs. By choosing the number of PEs and the clock frequency, the designer has a great deal of flexibility to tune performance, bandwidth, cost and power for the needs of a given application.

Architecture Details

The following sections describe the MTAP processor in more detail. A block diagram of the processor is shown in Figure 7. This is a conceptual view, which reflects the programming model. In practice, the control and mono execution units form a tightly-coupled pipeline which provides overall control of the processor.

Interfaces

The processor has a number of external interfaces. The widths of each of these are configurable. All these interfaces currently use the Virtual Component Interface Standard defined by the VSI Alliance.

There are three categories of interface:

Mono data & instructions

This is a single *Advanced* VCI (AVCI) interface which is used for mono loads and stores, and for instruction fetching.

Poly data

This uses one or more AVCI interfaces for PIO and SIO data. The number of physical interfaces corresponds to the number of PIO and SIO channels implemented.

Control

There are two control interfaces which use the *Peripheral* VCI (PVCi) protocol. One is used for initialization and debug. The other allows the MTAP processor to generate interrupts to the host system.

Instruction set

The processor has a fairly standard RISC-like instruction set. Most instructions can operate on mono or poly operands and are executed by the appropriate part of the execution unit. Some instructions are only relevant to either the mono or poly execution unit, for example all program flow control is handled by the mono unit.

The instruction set provides a standard set of functions on both mono and poly execution units:

- Integer and floating point adds, subtracts, multiplies, divides
- Logical operations: and, or, not, xor
- Arithmetic and logical shifts
- Data comparisons: equal, not equal, greater than, less than, etc.
- Data movement between registers
- Loads and stores between registers and memory

To give a feel for the nature of assembly code for the MTAP processor, a few lines of code are shown below. This simple example loads a value from memory into a mono register, gets the PE number into a register on each PE and then adds these two values together producing a different result on every PE.

```

5   ld 0:m4, 0x3000    // load mono reg 0 from mem
   penum 8:p4          // PE number into poly reg
   add 4:p4, 0:m4, 8:m4 // add; result in reg 4

```

Control unit

The control unit fetches instructions from memory, decodes them and despatches them to the appropriate functional unit.

The controller includes a scheduler to provide hardware support for multi-threaded code. This is a vital part of the architecture for achieving the performance potential of the MTAP processor. Because of the highly parallel architecture of the poly execution unit, there can be significant latencies if all PEs need to read or write external data.

15 When part of an application stalls because it is waiting for data from external memory, the processor can switch to another code thread that is ready to run. This serves to hide the latency of accesses and keep the processor busy.

The number of threads is a configurable option. The threads are prioritized: the processor will run the highest priority thread that is ready to run; a higher priority thread can pre-empt a lower priority thread when it becomes ready to run. Threads are
 20 synchronized – with each other and with hardware, such as I/O engines – via semaphores.

In the simplest case of multi-threaded code, a program would have two threads: one for I/O and one for compute. By *pre-fetching* data in the I/O thread, the programmer
 25 (or the compiler) can ensure the data is available when it is required by the execution units and the processor can run without stalling.

Execution units

This section describes the common aspects of the poly and mono execution units.

ALU operations

30 Instructions for arithmetic and logical operations are provided in several versions:

- Various sizes: 1, 2, 3 and 4 bytes, and combinations of sizes
- Various data types: signed and unsigned integers, and floating-point
- Mixes of signed/unsigned operations

Status register

Associated with the ALU is a status register; this contains five status bits that provide information about the result of the last ALU operation. When set, these bits indicate:

- Most significant bit set
- 5 • Carry generated
- Overflow generated
- Negative result
- Zero result

Registers

- 10 To support operations on data of different widths, the registers in the PEs can be accessed very flexibly. The register files are best thought of as an array of bytes which can be addressed as registers of 1 to 8 bytes wide.

The mono and poly registers are addressed in a consistent way using byte addresses and widths specified in bytes. The mono register file is 16 bits wide and so all
15 addresses and widths must be a multiple of 2. There are no alignment restrictions on poly register accesses.

Addressing modes

- Load and store instructions are used to transfer data between memory and registers. In the case of the mono execution unit, these transfer data to and from memory external
20 to the MTAP processor.

Poly loads and stores transfer data between the PE register file and PE memory. Data is transferred between the PEs and external memory using the I/O functions described later.

- There are three addressing modes for loads and stores which can be used for both
25 mono and poly data. These are:

Direct

The address to be read/written is specified as an immediate value.

Indirect

The address is specified in a register.

- 30 **Indexed**

The address is calculated from adding an offset to a base address in a register. The offset must be an immediate value.

Conditional code

The main difference between code for the mono and the poly execution units is the handling of conditional execution.

The mono unit uses conditional jumps to branch around code, typically based on the result of the previous instructions. This means that mono conditions affect both mono
5 and poly operations. This is just like a standard RISC architecture.

The poly unit uses a set of enable bits (described in more detail below) to control whether each PE will have its state changed instructions it executes. This provides per-PE predicated operation. The enable state can be changed based on the result of a
10 previous operation.

The following sections describe the architectural features of the two execution units in more detail.

Mono execution unit

The mono execution unit is a 16-bit processing element consisting of:

- 15 • A 16-bit ALU with optional extensions
- A register file of configurable size
- Status and control registers

The mono ALU extensions include a multiplier, a barrel shifter and a normalizer, which is important for accelerating software implementations of floating point
20 operations.

As well as handling mono data, the mono unit is responsible for program flow control (branching), thread switching and other control functions. The mono execution unit also has overall control of I/O operations of poly data. Results from these operations are returned to a register in the mono unit.

25 Conditional execution

The mono execution unit handles conditional execution in the same way as a traditional processor. A set of conditional and unconditional jump instructions use the result of previous operations to jump over conditional code, back to the start of loops, etc.

30 Multi-threaded execution

The MTAP processor supports several hardware threads. There is a hardware scheduler in the control unit and the mono execution unit maintains multiple banks of critical registers for fast context switching.

The threads are prioritized (0 being highest priority). Control of execution between the threads is performed using semaphores under programmer control. Higher priority threads will only yield to lower priority threads when they are stalled on yielding instructions (such as semaphore wait operations). Lower priority threads can be pre-empted at any time by higher priority ones.

Semaphores are special registers that can be incremented or decremented with atomic (non-interruptible) operations called *signal* and *wait*. A *signal* instruction will increment a semaphore. A *wait* will decrement a semaphore unless the semaphore is 0, in which case it will stall until the semaphore is signalled by another thread.

Semaphores can also be accessed by hardware units (such as the I/O controllers) to synchronize these with software.

Poly execution unit

The poly execution unit is an array of Processing Elements (PEs). Each PE in the poly execution unit consists of:

- An 8-bit ALU with optional extensions such as a multiply-accumulate (MAC) unit
- A register file of configurable size
- Status and enable registers
- A block of memory of configurable size
- An inter-PE communication path
- One or more I/O channels

Load and store instructions move data between a PE's register file and memory, while the ALU operates on data in the register file. Data is transferred in to, and out of, the PE's memory using I/O instructions.

ALU

The poly ALU is used for performing arithmetic and logical operations on values held in the PE register file.

While the ALU is only 8 bits wide, instructions exist for multi-byte arithmetic which is handled by iteration. ALU extensions exist to accelerate functions for floating point, DSP, etc. For example, an optional integer multiply-accumulate unit (MAC) can be included in the ALU. This can deliver an 8 x 8 bit MAC result every cycle, or a 16 x 16 bit MAC every four cycles. The accumulated result can be up to 64 bits wide. The standard ALU includes basic hardware support to accelerate floating point operations.

Conditional behaviour

The SIMD nature of the PE array prohibits each PE having its own branch unit (branching being handled by the mono execution unit). Instead, each PE can control whether its state should be updated by the current instruction by *enabling* or *disabling* itself; this is rather like the predicated instructions in some RISC CPUs.

Enable state

A PE's enable state is determined by a number of bits in the enable register. If all these bits are set to one, then a PE is *enabled* and executes instructions normally. If one or more of the enable bits is zero, then the PE is *disabled* and most instructions it receives will be ignored (instructions on the enable state itself, for example, are not be disabled).

The enable register is treated as a stack, and new bits can be pushed onto the top of the stack allowing nested predicated execution. The result of a test, either a 1 or a 0, is pushed onto the enable stack. This bit can later be popped from the top of the stack to remove the effect of that condition. This makes handling nested conditions and loops very efficient. Note that, although the enable stack is of fixed size, the compiler handles saving and restoring the state automatically, so there are no limitations on compiled code. When programming at the assembler level, it is the programmer's responsibility to manage the stack.

Instructions

Conditional execution on the poly execution unit is supported by a set of poly conditional instructions: **if**, **else**, **endif**, etc. These manage the enable bits to allow different PEs to execute each branch of an **if...else** construct in C, for example. These also support nested conditions by pushing and popping the condition value on the enable stack.

As a simple example, consider the following code fragment:

```
// disable PEs where reg 32 is non-zero
if.eq 32:p1, 0 // push result onto stack
// increment reg 8 on enabled PEs
30 add    8:p4, 8:p4, 1
// return all PEs to original enable state
endif           // pop enable stack
```

Here, the initial **if** instruction compares the two operands on each PE. If they are equal it pushes 1 onto the top of the enable stack – this leaves those PEs enabled if they

were previously enabled and disabled if they were already disabled. If the two operands are not equal, a 0 is pushed onto the stack – this disables the corresponding PEs.

The following **add** instruction is sent to all PEs, but only acted on by those that are still enabled. Finally, the **endif** instruction pops the enable stack, returning all PEs to their original enable state.

Forced loads and stores

Poly loads and stores are normally predicated by the enable state of the PE. However, because there are instances where it is necessary to load and store data regardless of the current enable state, the instruction set includes *forced* loads and stores. These will change the state of the PE even if it is disabled.

I/O mechanisms

There are two I/O mechanisms provided for transferring data between PE memory and devices outside the MTAP core. *Programmed I/O* (PIO) extends the load/ store model: it is used for transfers of small amounts of data between PE memory and external memory. *Streaming I/O* (SIO) is used to efficiently transfer contiguous chunks of data in and out of PE memory.

The number of PIO and SIO channels in a processor is a configurable parameter. A processor will always have at least one PIO channel as this is required for normal program execution.

Multiple I/O channels can run simultaneously.

I/O architecture

The I/O systems consist of three parts: Controller, Engine and Node.

Controller

The PIO and SIO controllers decode I/O instructions and coordinate with the rest of the control unit and the mono processor. The controllers synchronize with software threads via semaphores.

Engine

The I/O engines are basically DMA engines which manage the actual data transfer.

There is a Controller and Engine for each I/O channel. A single Controller can manage several I/O Engines.

Node

There is an I/O Node in each PE. The I/O Engine activates each Node in turn allowing to serialize the data transfers. The Nodes provide buffering of data to minimize the impact of I/O on the performance of the PEs.

5 **Programmed I/O (PIO)**

PIO is closely coupled to program execution and is the normal way for the processor to transfer data to and from the outside world (e.g. external memory, hardware accelerators or a host processor).

The PIO mechanism provides a number of addressing modes:

10 **Direct addressed**

Each PE provides an external memory address for its data. This provides random access to data.

Strided

15 The external memory address is incremented for each PE. In each case, the size of data transferred to each PE is the same.

When multiple PEs are accessing memory then the transfers can be *consolidated* so as to perform the minimum number of external accesses. So, for example, if half the processors are reading one location and the other half reading another, then only two memory reads would be performed. In fact, consolidation can be better than that:
20 because the bus transfers are *packetized*, even transfers from nearby addresses can be effectively consolidated.

Streaming I/O (SIO)

SIO is used for streaming high-bandwidth data directly to and from PE memory. This is less flexible but very efficient for transferring blocks of data in and out of the
25 system. This is typically used to stream data to and from memory mapped I/O devices. Each PE can transfer a different size block of data.

Streaming I/O can be made very efficient in a number of ways. For example, multi-threaded code allows data I/O and compute to be fully overlapped. Also, multiple data buffers can be allocated in PE memory so that different chunks of data can be input,
30 processed and streamed out simultaneously.

Swazzle

Finally, the PEs are able to communicate with one another via what is known as the *swazzle* path that connects the register file of each PE with the register files of its left and right neighbours. On each cycle, PE_n can perform a register-to-register transfer of

16 bits to either its left or right neighbour, PE_{n-1} or PE_{n+1} , while simultaneously receiving data from the other neighbour.

Swizzle instructions use multiples of 2 bytes in their arguments, so the source and destination registers must be 2-byte aligned. Instructions are provided to shift data left
5 or right through the array, and to swap data between adjacent PEs.

The enable state of a PE affects its participation in a swizzle operation in the following way: if a PE is enabled, then its registers may be updated by a neighbouring PE, regardless of the neighbouring PE's enable state. Conversely, if a PE is disabled, its register file will not be altered by a neighbour under any circumstance. A disabled
10 PE will still provide data to an enabled neighbour.

The data written into the registers of the PEs at the ends of the swizzle path can be set by the mono execution unit.

Host interface

The interfaces to the host system are used for 3 basic purposes: initialization and
15 booting, access to host services and debugging.

Initialization

There are a number of stages of initialization required to start code running on the MTAP processor. These are normally handled transparently by the development tools, but an overview is provided here as background information.

20 First, the application code (including bootstrap) is loaded into memory.

Next, the host system initializes the state of the control unit, caches and mono execution unit by a series of writes to the PPCI port. The last of these specify the start address of the code to execute and tell the processor to start fetching instructions.

Finally, the boot code does any remaining initialization of the processor including any
25 set-up of the PEs (e.g. setting the PE number) before running the application code.

Host services

Once application program is running it will need to access host resources such as the file system.

A protocol is defined between the run-time libraries and a device driver on the host
30 system. This is interrupt based and allows the code running on the MTAP processor to make calls to the application or operating system running on the host.

Debugging

The processor includes hardware support for breakpoints and single-stepping. These are controlled via registers accessed through the PPCI interface.

This interface also allows the debugger, running on the host, to interrogate and update the state of the processor to support fully interactive debugging.

Case Study: Device EV1

To illustrate the use of an MTAP processor in a real device, the architecture of Applicant's evaluation chip EV1 is described in this section. This device is a simple embodiment of Applicant's MTAP processor using Applicant's bus to interface the various ports of the processor to a memory block and to external pins.

EV1 architecture

Figure 9 shows the top level architecture of the EV1 device. The five ports of the MTAP processor and one port of the embedded memory block are connected to a single **ClearConnect** channel (**ClearConnect** is Applicant's bus structure) of six nodes and two lanes in opposing directions. The two external ports at each end of the bus allow the **ClearConnect** bus to be connected from one chip to another. This means a system can be built from multiple EV1 devices to provide the required performance.

The bus ports can also be used to connect to an FPGA to provide other functions, such as peripherals, an external memory controller or a host interface.

The EV1 chip also includes on-chip SRAM which provides fast access to code and data.

MTAP processor

The MTAP processor core in the EV1 has the following specification:

- General
 - Implemented on 0.13 μ process
 - Clock speed: 200 MHz
 - 25 • 4 Kbyte instruction cache: 4-way, 256 lines x 4 instructions, with manual and auto pre-fetch
 - 4 Kbyte data cache, 4-way, 256 lines x 16 bytes
- Mono execution unit
 - 64 byte register file
 - 30 • Support for 8 threads
- Poly execution unit
 - Array of 48 PEs
 - MAC extension to the ALU
 - 4 Kbytes SRAM per PE

- 64 byte register file
- One PIO channel
 - AVCI port: 32-bit address, 64-bit data
 - Transfer size: 4, 8, 16, 32 bytes per PE
- 5 • Address modes: direct and strided
- One SIO channel
 - AVCI port: 32-bit address, 64-bit data
 - Transfer size: up to 128 bytes per PE
- Control interfaces
- 10 • Interrupt port: 32-bit address, 32-bit data PVCI
- Register port: 32-bit address, 32-bit data PVCI

This gives a performance of:

- 9,600 MIPS
- 9.6 billion 8x8 MACs / second
- 15 • 38 Gbytes/s memory bandwidth
- 18 Gbytes/s inter-PE bandwidth

Example Applications

Here we give a couple of examples of how the MTAP processor can be applied to specific applications.

20 ***Network processing***

In this case, the data to be processed consists of a stream of *packets* of variable size. Each packet consists of header information and a data payload. In its simplest form, the problem is to examine various fields in the header, do some sort of lookup function and route the packet to the appropriate output port. This processing must be
25 done in real time.

In the network processing application, the SIO channels are used for continuously streaming packets into, and out of, PE memory. Within each PE, the software maintains several buffers so that, while one set of packets is being processed, the previous set can be output and, simultaneously, the next set can be loaded.

- 30 The PIO channels are used by the PEs to send requests to another subsystem that handles lookups. This could on-chip, for example Applicant's Table Lookup Engine, or an off-chip solution such as CAMs.

Bio-informatics

The developing field of *in-silico* drug discovery uses computer simulation rather than the 'wet science' of test tubes to explore the behaviour of potential new drugs.

This includes molecular simulation of individual molecules and the interactions of molecules. For example, the docking of a ligand molecule (drug) into a protein. This simulation requires the calculation of the interaction energies of all atoms of one molecule with all the atoms of the other. This process is repeated for thousands of configurations of the molecules. This results in an embarrassingly parallel problem ideally suited to the MTAP architecture.

Each PE is allocated a different configuration of protein and ligand, and performs all of the atom-atom interaction energy calculations. The memory available within a PE is insufficient to hold the details of an entire molecule, however the molecule can be split into pieces and each piece processed in turn. The fetching of atoms can be overlapped with the atom-atom processing and, even with the floating point accelerated MTAP processor, is compute bound. The processing currently performed in most production code is a simplistic model of the interaction of atoms. The processing power that the MTAP architecture makes available allows more sophisticated models to be considered, improving the accuracy and thus value of the results.

The MTAP processor is also suited to other bio-informatic tasks such as genetic sequence comparison and pharmacophore fingerprinting.

Software Development Kit

The Software Development Kit (SDK) provides a complete set of development tools: C compiler, assembler linker, debugger, profiler, etc. The software tools use a central configuration file to define the attributes of the target processor that code is being generated for.

Simulation tools

Cycle and bit-accurate simulations of the core are available in C and Verilog. These can be parameterized for the number of PEs, size of memory, and other options. A generic configuration of the C simulator is shipped with the SDK. Once the target system is defined, the simulator can be configured to match that specification.

This allows application development to start before the system architecture is fully defined, and then proceed in parallel with the silicon implementation.

Once the target hardware (or a simulation) is available then the application code can be run and debugged on the target hardware.

Debugger

A major challenge when developing code for a parallel architecture is understanding
5 the state of the machine when the code is stopped at a breakpoint or error. The data-parallel nature of the MTAP processor minimizes the difficulty here: a single instruction stream is executing and only the data is different on each PE.

Applicant's 2nd generation debugger builds on previous experience and provides a variety of ways to visualize the state of PE memory and registers. As well as
10 traditional source-level symbolic views of data, and low-level dumps of memory and data, the debugger provides user-defined *picture* views which show the contents of PE memory or registers in graphical form. This provides a quickly understood view of the overall state of the system.

The debugger works identically with all of the simulators and with target hardware. It
15 supports all the features expected from a modern development tool: graphical user interface and command line operation, breakpoints, watchpoints, single-stepping, etc.

Claims

1. A data processor comprising a single control unit and an integral execution unit adapted to operate on both parallel and scalar data under the same instruction set.
5
2. A data processor comprising a single control unit and an integral execution unit adapted to operate on both parallel and scalar data under the same instruction set, and wherein the execution unit comprises a SIMD part comprising an array of processing elements adapted to operate on said parallel data and a scalar part adapted
10 to operate on said scalar data.
3. A data processor comprising a single control unit and an integral execution unit adapted to operate on both parallel and scalar data under the same instruction set, and wherein substantially all said instructions are operable on said parallel and/or said
15 scalar data.
4. A data processor comprising a single control unit and an integral execution unit adapted to operate on both parallel and scalar data under the same instruction set, and wherein the execution unit comprises a SIMD part comprising an array of
20 processing elements adapted to operate on said parallel data and a scalar part adapted to operate on said scalar data, and wherein substantially all said instructions are operable on said parallel and/or said scalar data.
5. A data processor as claimed in claim 1, wherein said scalar part and said
25 SIMD part are adapted to operate asynchronously.
6. A data processor as claimed in any of claims 1, 3 or 5, wherein the execution unit comprises a SIMD part comprising an array of processing elements (PEs) adapted to operate on said parallel data and a scalar part adapted to operate on said scalar data.
30
7. A data processor as claimed in any of claims 2, 4 or 6, wherein the SIMD part and the scalar part differ only in that each processing element of the SIMD part contains local memory.

8. A data processor as claimed in claim 6, wherein said SIMD part and said scalar part have a common memory area for instructions and scalar data.
9. A data processor as claimed in claim 6, wherein each PE contains an ALU and
5 one or more multiply-accumulate function units.
10. A data processor as claimed in claim 6, wherein each PE contains an ALU and one or more floating point function units.
- 10 11. A data processor as claimed in claim 6, wherein each PE has multiple enable bits to support nested conditional code.
12. A data processor as claimed in claim 6, wherein said PEs and said scalar part each have a similar set of status/result flags which can be used to control branching,
15 on the scalar part of the execution unit, and conditional execution, on the SIMD part of the execution unit, in a similar manner.
13. A data processor as claimed in claim 6, further comprising a communication path between PEs in the SIMD part.
- 20 14. A data processor as claimed in claim 6, wherein means are provided for data to be transferred to and from the scalar part of the execution unit and the SIMD part of the execution unit.
- 25 15. A data processor as claimed in any of the preceding claims, wherein the control unit is adapted to fetch and execute multiple instruction threads simultaneously.
16. A data processor as claimed in any of the preceding claims, further comprising
30 a semaphore unit adapted to use semaphores to synchronize between instruction threads.
17. A data processor as claimed in claim 16, wherein said semaphores are adapted to synchronize between I/O operations and instruction threads.

18. A data processor as claimed in claim 16, wherein said control unit is adapted to schedule and execute threads based on their priority and the state of any semaphores they are waiting on.
- 5 19. A data processor as claimed in claim 16, wherein said control unit includes a set of said semaphores adapted to be used to synchronize threads.
20. A data processor as claimed in either of claims 2 or 3, wherein each
10 processing element of the SIMD array comprises one or more function units, a register file, said local memory, and I/O means.
21. A data processor as claimed in any of claims 2, 4, 6 or 7, wherein both said scalar part and said SIMD part have a complete range of addressing modes that allow
15 them to support:
- scalar variables pointing to data in scalar memory;
 - scalar variables pointing to data in memory in the SIMD part of the execution unit;
 - SIMD variables pointing to memory in the SIMD part of the execution unit; and
 - 20 SIMD variables pointing to data in scalar memory.
22. A data processor as claimed in any of claims 1, 2 or 5-21, wherein substantially all said instructions are operable on said parallel data or on said scalar data or on a combination of both said parallel data and said scalar data.
- 25 23. A data processor as claimed in claim 22, further comprising a compiler adapted to produce code from a single source program to operate on both said scalar data and said parallel data.
- 30 24. A data processor as claimed in claim 22, wherein said compiler is adapted to detect parallel data automatically.
25. A data processor as claimed in claim 23 or 24, wherein a programmer or said compiler is adapted to use the multi-threaded execution to schedule I/O operations

such that they occur concurrently with computation and data is always available when required.

26. A data processor as claimed in claim 20, wherein said scalar part, said SIMD
5 part and said I/O means are adapted to be operated in parallel by the same instruction stream.

27. A data processor as claimed in claim 26, wherein said I/O means comprises
programmed I/O means adapted to enable each PE to transfer data to and from
10 external scalar memory and wherein each PE provides the address it wants to read data from or write data to.

28. A data processor as claimed in claim 27, wherein memory accesses are
consolidated to minimize the number of memory transfers when multiple PEs are
15 accessing the same memory area.

29. A data processor as claimed in claim 26, wherein each PE is adapted to
transfer data to and from external scalar memory and wherein the address for each PE
is generated automatically based on the PE number and the amount of data being
20 transferred.

30. A data processor as claimed in claim 26, wherein said I/O means comprises
streaming I/O means which distributes an incoming data stream to all the PEs in the
SIMD part and collects data from all the PEs to generate an output stream
25 independently of program execution on the SIMD part of the execution unit.

31. A data processor as claimed in claim 30, wherein the streaming I/O means
only uses enabled PEs to take part in the streaming I/O operation.

30 32. A data processor as claimed in claim 30, wherein the size of data distributed to, and collected from each PE is different for each PE.

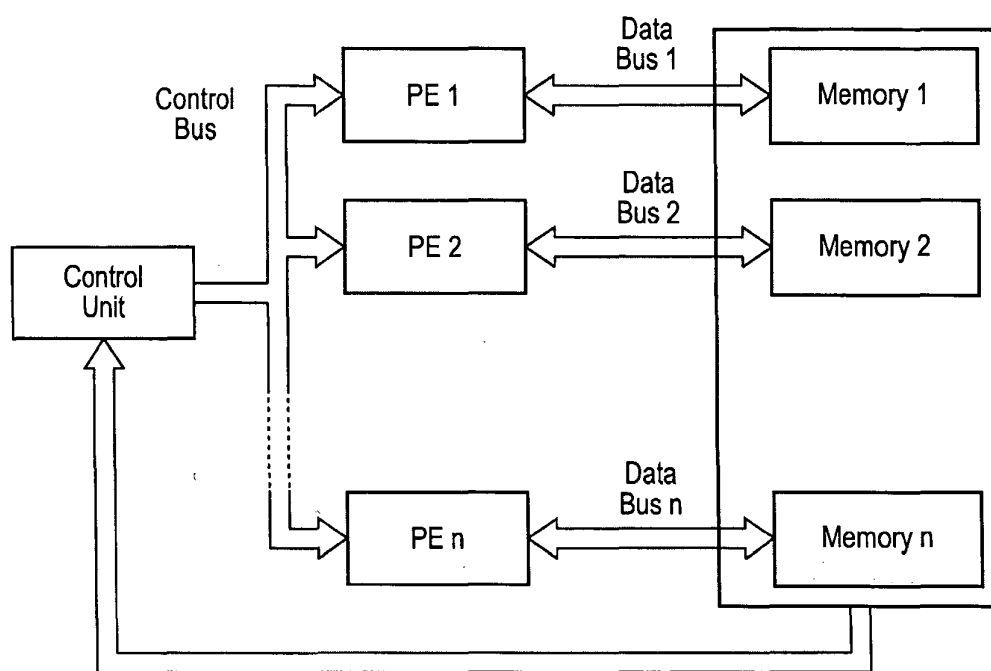


Fig. 1

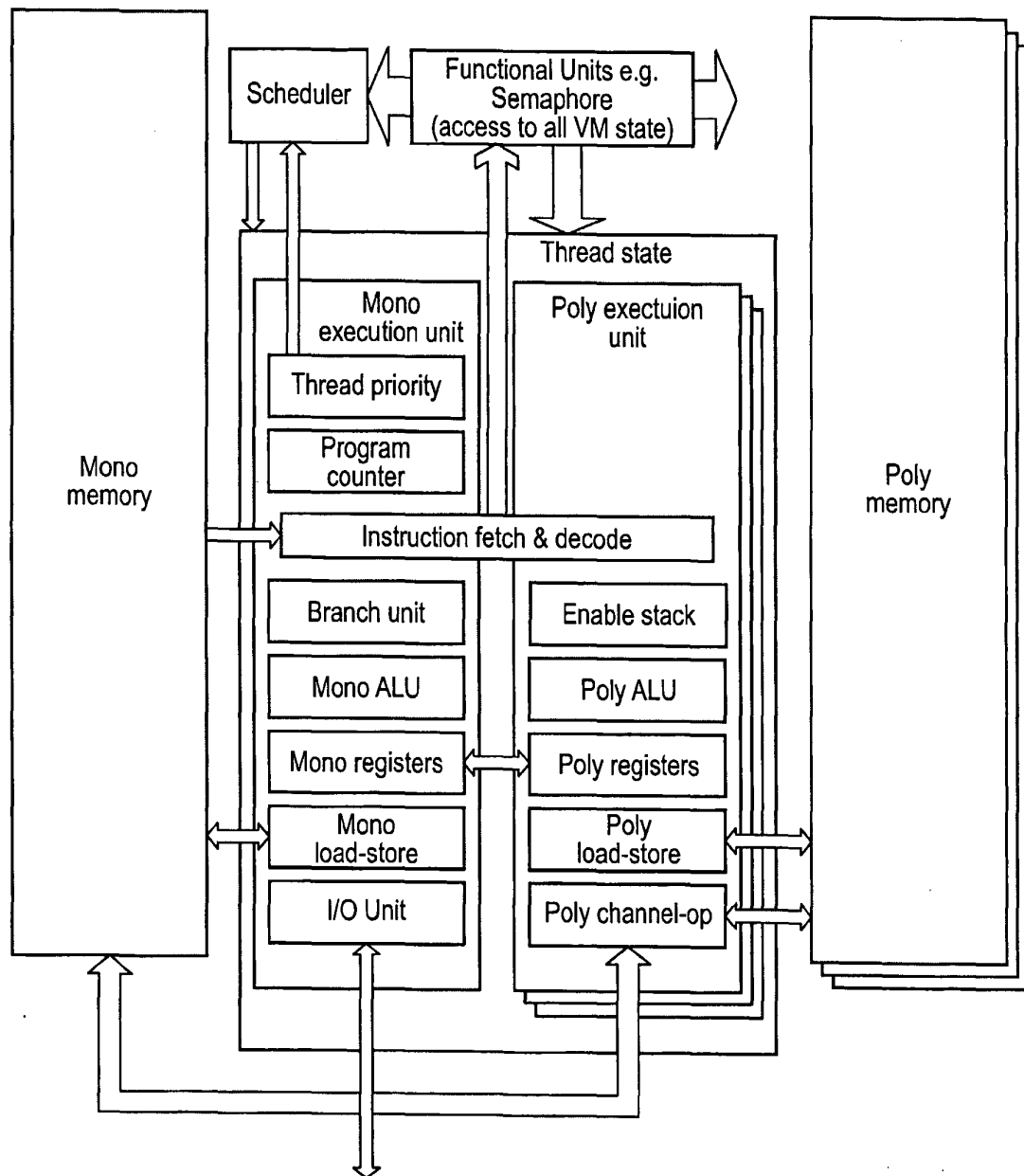


Fig. 2

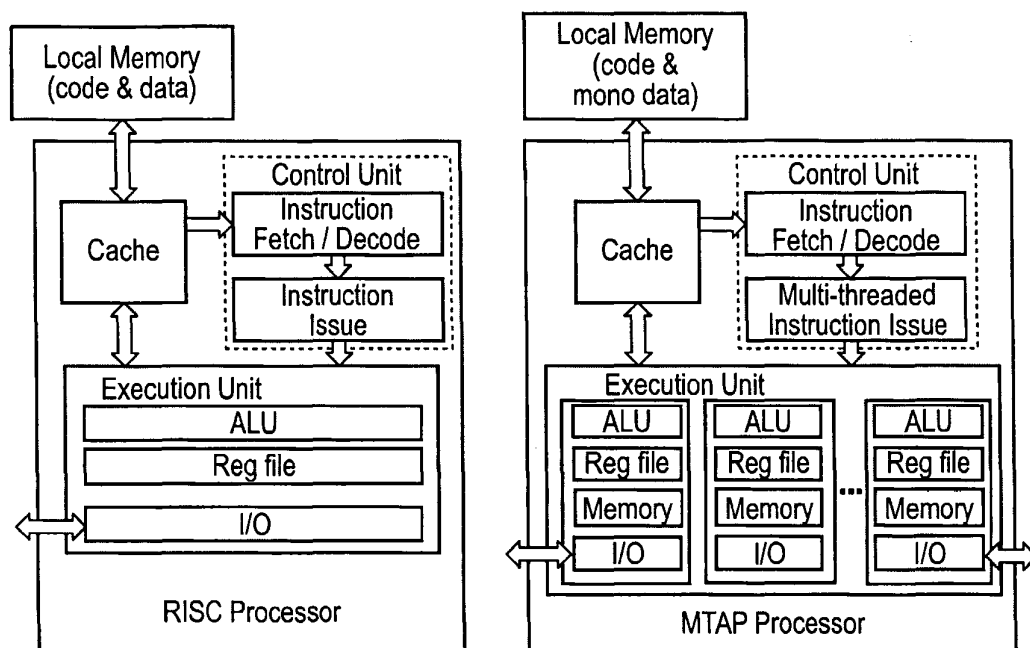


Fig. 3

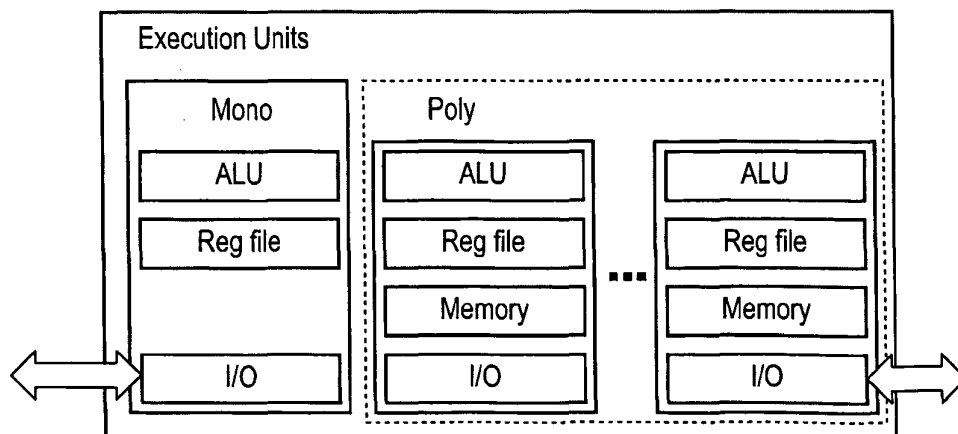


Fig. 4

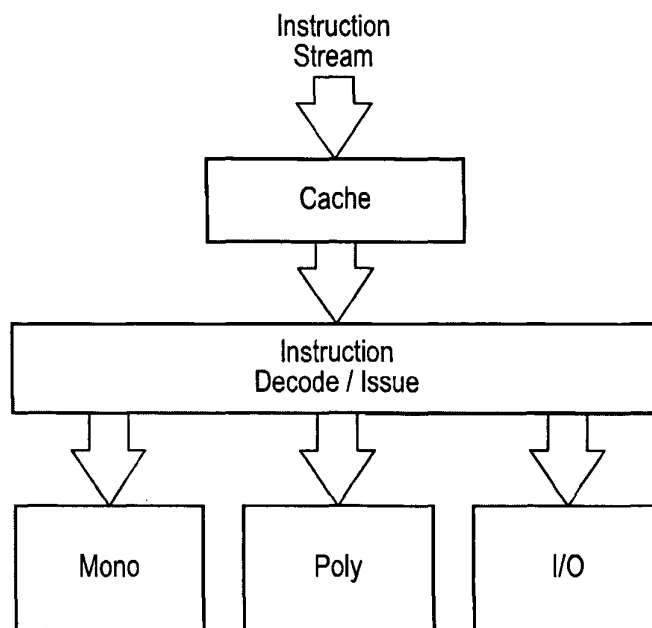


Fig. 5

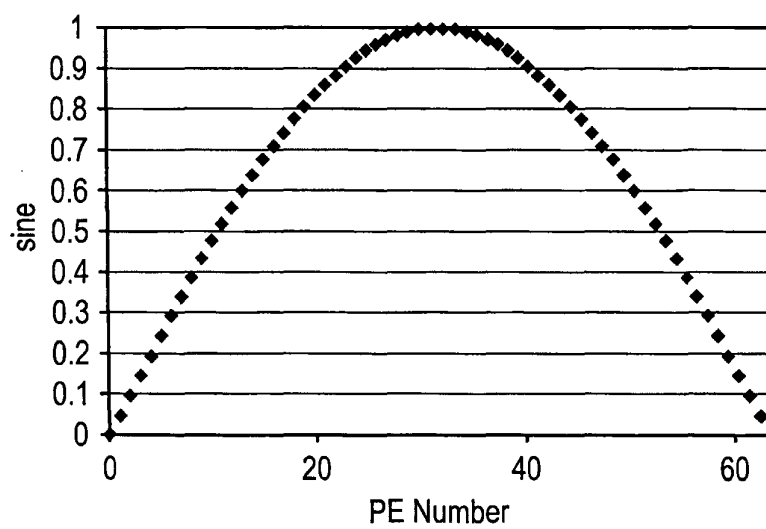


Fig. 6

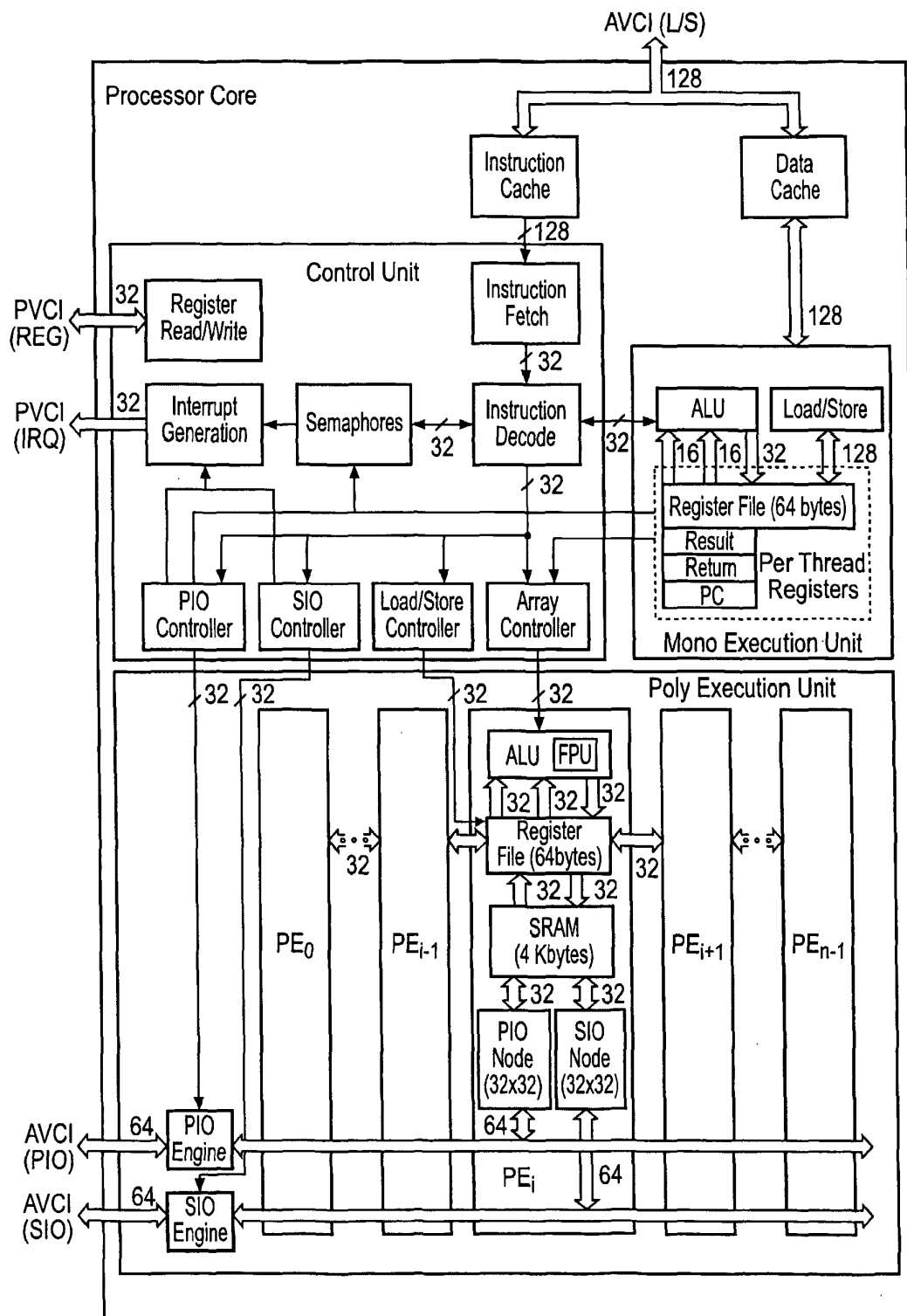


Fig. 7

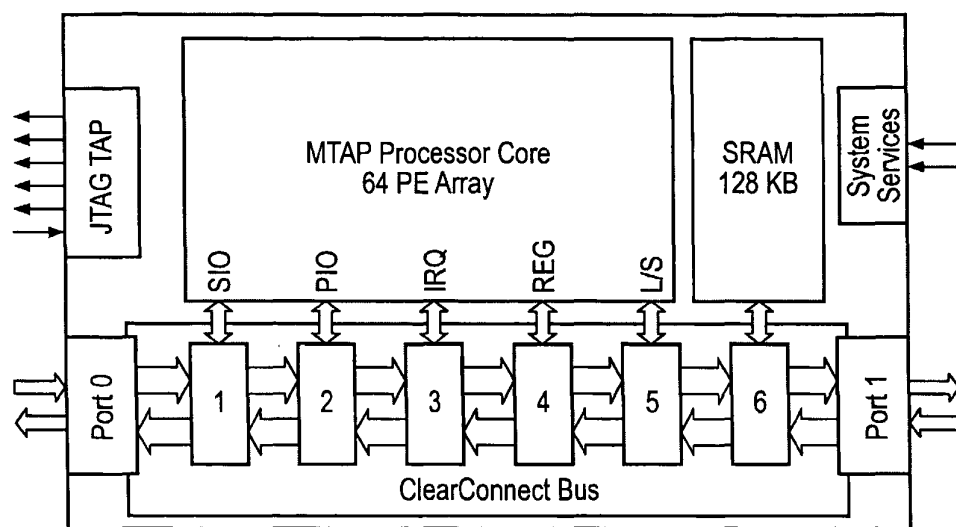


Fig. 8