

# CLEARSPPEED WHITEPAPER: CSX PROCESSOR ARCHITECTURE

## Abstract

This paper describes the architecture of the CSX family of processors based on ClearSpeed's multi-threaded array processor; a high performance, power-efficient, scalable, data-parallel processor.

The CSX processors have a simple programming model and are supported by a suite of software development tools, built around a C compiler, graphical debugger and visual profiling tools. High performance standard math libraries are also available.

Developed for applications with demanding performance requirements, the processor can be used as an application accelerator for high performance computing (HPC) and also as an embedded processor for Digital Signal Processing (DSP).

The use of the add-in CSX600-based Advance™ Board as an application coprocessor is also described. This PCI form factor board provides over 50 GFLOPS DGEMM sustained performance while typically consuming only 25 watts.

## Introduction

This whitepaper describes ClearSpeed's CSX family of coprocessors which are based around a multi-threaded array processor (MTAP) core. This processing architecture has been developed to address a number of problems in high performance, high data rate processing.

CSX processors can be used as application accelerators, alongside a standard processor such as those from Intel or AMD.

The architecture can be utilized with any application with significant data parallelism:

- Fine Grained e.g. vector operations
- Medium Grained e.g. unrolled independent loops
- Coarse Grained e.g. multiple simultaneous data channels

The CSX600-based ClearSpeed Advance board is used as an application accelerator for high performance computing (HPC). It works with standard processors (from Intel or AMD) to share the compute intensive parts of an application. By accelerating standard software libraries used by a number of applications, the ClearSpeed Advance board allows these applications to be accelerated "out of the box."

The ClearSpeed Advance accelerator board can be used to accelerate a single workstation, server or an entire cluster; multiple boards may be used in one computer.

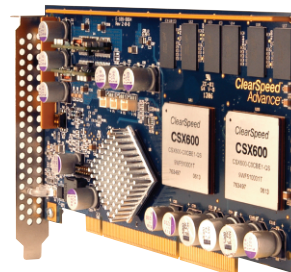


FIGURE 1: CLEARSPEED ADVANCE BOARD

## CSX Family

The CSX processors are designed as application accelerators and embedded processors. They provide a complete System on Chip (SoC) solution: integrating an MTAP processor with all the necessary interfaces and support logic.

The first product in the CSX family is the CSX600. This is optimized for high performance, floating point intensive applications. It includes an MTAP processor, DDR2 DRAM interface, on-chip SRAM and high speed inter-processor I/O ports.

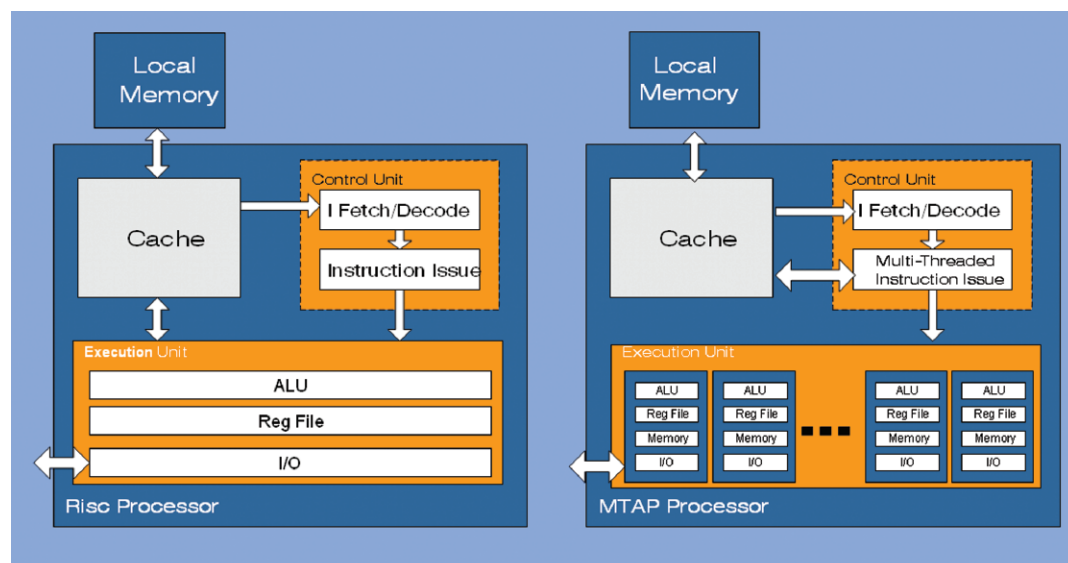


FIGURE 2: PROCESSOR EVOLUTION

## MTAP Architecture

The MTAP architecture defines a family of embedded processors with parallel data processing capability. Figure 1 compares a standard processor and the multi-threaded array processor. 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 architecture.

To simplify the integration of the processor into a variety of systems, the processor can be configured at boot time 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 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 processor also includes instruction and data caches to minimize the latency of external memory accesses.

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



FIGURE 3: CLEARSPD CSX600 COPROCESSOR ON THE ADVANCE BOARD

## Execution Units

The execution unit uses a number of poly execution (PE) cores. This allows it to process data elements in parallel. Each PE core consists of at least one ALU, registers, memory and I/O. The CSX600 also includes an integer MAC and a dual floating point unit (FPU) on every PE core.

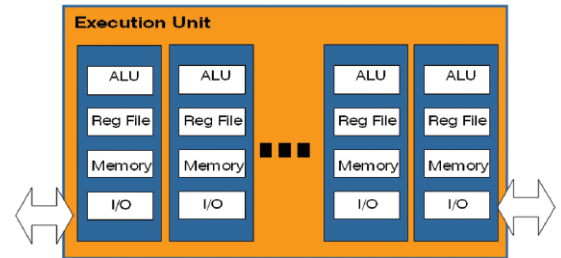


FIGURE 4: POLY EXECUTION UNIT

The execution unit can be thought of as two, largely independent, parts (see Figure 4)

One part 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 PE cores form the poly execution unit which processes parallel (poly) data.

The poly execution unit may consist of tens, hundreds or even thousands of PE cores. This array of PE cores operates in a synchronous manner, similar to a Single Instruction, Multiple Data (SIMD) processor, where every PE core executes the same instruction on its piece of data. Each PE core also has its own independent local memory; this provides fast access to the data being processed. For example, one PE core at 250 MHz has a memory bandwidth of 1 Gbytes/s. An array with 96 such PE cores has an aggregate bandwidth of approximately 100 Gbytes/s with single cycle latency.

## Programming Model

The control unit fetches, decodes and dispatches instructions to the execution units. The processor executes a fairly standard, three operand instruction set.

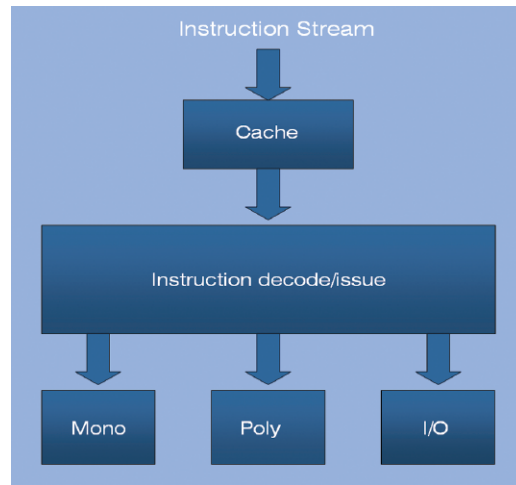


FIGURE 5: INSTRUCTION EXECUTION

From a programmer's perspective, the 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 processor cores, or can only access the processor via function calls or some other indirect mechanism.

The processor executes a single instruction stream; each instruction is sent to one of the functional units: this may be in the mono or poly execution unit, or one of the I/O controllers. The processor can dispatch 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 several cycles to complete). 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 core and are operated on in parallel: we call this *poly* data. This can be thought of as a data vector which is distributed across the PE core array.

Variables which only need to have a single instance (e.g. loop control variables) are known as mono variables, they behave exactly like normal variables on a sequential processor.

ClearSpeed 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 declaration to define data which exists, and is processed, independently on every PE core in the poly execution unit.

## Example

To give a feel for the way the processor is programmed, we use a simple example of evaluating the sine function.

```

#include <stdio.h>
#include <math.h>

#define PI 3.1415926535897932384f
#define SAMPLES 96

int main() {
    double sine, angle;
    int i;

    for (i = 0; i < SAMPLES; i++) {
        // convert to an angle in
        // range 0 to Pi
        angle = i * PI / SAMPLES;
        // calculate sine of angle
        sine = sin(angle);
    }
}
  
```

FIGURE 6: STANDARD C

```
#include <lib_ext.h>

#define PI 3.1415926535897932384f
#define SAMPLES 96

int main() {
    poly double sine, angle;
    poly int i;

    // get PE number: 0...n-1
    i = get_penum();
    // convert to an angle in range 0
    to Pi
    angle = i * PI / SAMPLES;
    // calculate all sine values
    simultaneously
    sine = sinp(angle);
}
```

FIGURE 7: PARALLEL C

Figure 6 shows a loop in standard C which iterates to calculate a number of values of the sine function.

Figure 7 shows an equivalent piece of code which simultaneously calculates 96 different values of the sine function across the PE core array.

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

Finally, the library function `sinp()` is called to calculate the sine of these values on all PE cores 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 core.

The result of running this program on a 96-PE core processor is shown graphically in Figure 8.

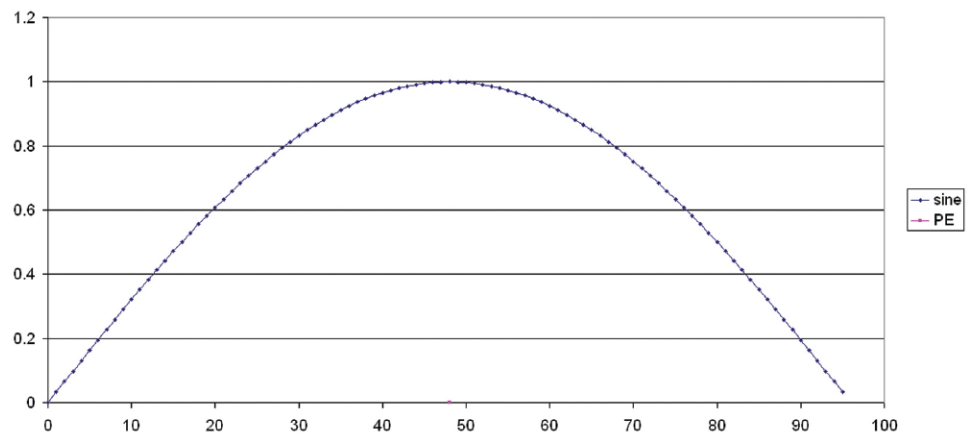


FIGURE 8: RESULTS FROM EXAMPLE PROGRAM

## Architecture Details

The following sections describe the multi-threaded array processor in more detail. A block diagram of the processor is shown in Figure 9. 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 interfaces to the ClearConnect NoC. There are three categories of interface:

**Mono data & instructions:** This interface is used for mono loads and stores, and for instruction fetching.

**Poly data:** There are one or more interfaces for I/O and data. The number of physical interfaces corresponds to the number of I/O channels implemented.

**Control:** The control interface is used for initialization and debug, and to allow the processor to generate interrupts on 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 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, exclusive-or, etc.

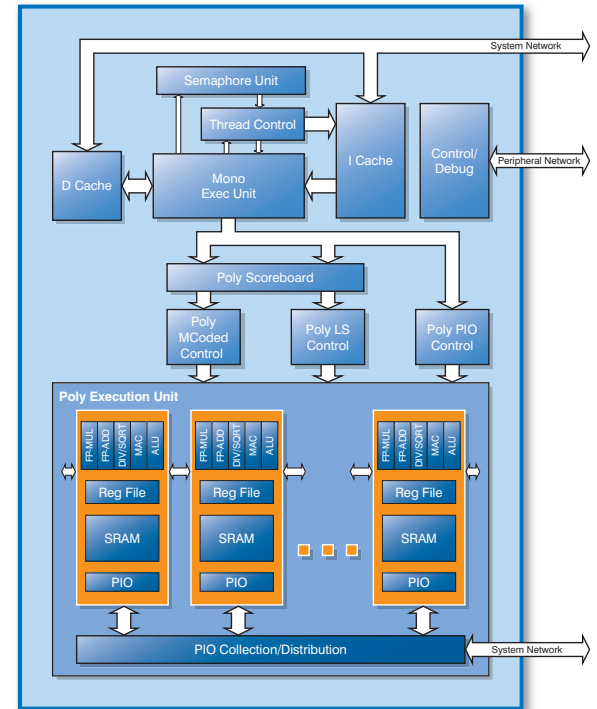


FIGURE 9: MULTI-THREADED ARRAY PROCESSOR

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

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

## Control Unit

The control unit fetches instructions from memory, decodes them and dispatches 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 processor. Because of the highly parallel architecture of the poly execution unit, there can be significant latencies if all PE cores need to read or write external data.

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 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 synchronized (with each other and with hardware such as I/O engines) via hardware 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 (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 Unit

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

## ALU Operations

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

- Various sizes: 1, 2, 4 and 8 bytes;
- Various data types: signed and unsigned integers, and single and double-precision floating point;
- Hardware support for floating point and DSP functions.

## 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
- Carry generated
- Overflow generated
- Negative result
- Zero result

## Register

To support operations on data of different widths, the registers in the PE cores 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 addresses and widths must be a multiple of 2 bytes. There are no alignment restrictions on poly register accesses.

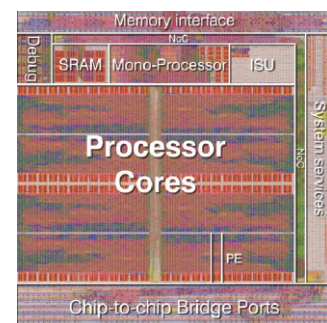


FIGURE 10: POLY-DIE CHIP

## 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 to the processor.

Poly loads and stores transfer data between the PE core register file and PE core memory. Data is transferred between the PE cores 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 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.

**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 way that conditional code is executed. 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 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 core will have its state changed by any instructions it executes. This provides per-PE core predicated operation. The enable state can be changed based on the result of a previous operation.

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

## Mono execution unit

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. Results from these operations are returned to registers in the mono unit.

## 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.

## Multi-threaded execution

The 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 Poly Execution (PE) cores. Each PE core in the poly execution unit is similar to a VLIW processor and consists of:

- Multiple function units such as:
  - Floating point unit (FPU)
  - Integer multiply-accumulate (MAC) unit
  - Integer arithmetic/logic unit (ALU)
- A register file
- Status and enable registers
- A block of fast, private memory
- An inter-PE core communication path
- One or more I/O channels

Load and store instructions move data between a PE core register file and PE core memory, while the function units operate on data in the register file. Data is transferred in to, and out of, the PE cores memory using I/O instructions.

The multiple functional units can operate concurrently so that, for example, an integer operation can be performed simultaneously with a floating point operation.

## Conditional behavior

The SIMD nature of the PE core array prohibits each PE core having its own branch unit (branching being handled by the mono execution unit). Instead, each PE core 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 core'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 core is enabled and executes instructions normally. If one or more of the enable bits is zero, then the PE core is disabled and most instructions it receives will be ignored (instructions on the enable state itself, for example, are not 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 enable 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 PE cores 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 all PEs where reg 32 is
// non-zero

if.eq 32:p1, 0 // pushing result
onto stack

// increment reg 8 on enabled PEs
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 core. If they are equal it pushes 1 onto the top of the enable stack. This leaves those PE cores 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 PE cores.

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

Normally, none of this is visible to the programmer writing code in C: conditional code just does what's expected.

### Forced loads and stores

Poly loads and stores are normally predicated by the enable state of the PE core. 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 core even if it is disabled.

### I/O mechanisms

I/O instructions transfer data between PE core memory and devices outside the CSX processor. Programmed I/O (PIO) extends the load/store model: it is used for transfers of small amounts of data between PE core memory and external memory.

### I/O architecture

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

**Controller:** The PIO controller decodes I/O instructions and coordinates with the rest of the control unit and the mono processor. The controller synchronizes 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 core. 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 PE cores.

PIO is closely coupled to program execution and is used to transfer data to and from the outside world (e.g. external memory, peripherals or the host processor).

The PIO mechanism provides a number of addressing modes:

**Direct addressed:** Each PE core provides an external memory address for its data. This provides random access to data.

**Strided:** The external memory address is incremented for each PE core.

In each case, the size of data transferred to each PE core is the same.

When multiple PE cores 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: because the bus transfers are packetized, even transfers from nearby addresses can be effectively consolidated.

### Swazzle

Finally, the PE cores are able to communicate with one another via what is known as the swazzle path that connects the register file of each PE core with the register files of its left and right neighbors. On each cycle, PE core<sub>*n*</sub> can perform a register-to-register transfer to either its left or right neighbor, PE core<sub>*n-1*</sub> or PE core<sub>*n+1*</sub>, while simultaneously receiving data from the other neighbor.

Instructions are provided to shift data left or right through the array, and to swap data between adjacent PE cores.

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

The data written into the registers of the PE cores at the ends of the swazzle path can be set by the mono execution unit. Alternatively, the two ends can be connected to form a circular path.

## Host Interface

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

### Initialization

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

First, the application code (including boot-strap) 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 host/debug 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 the PE cores (e.g. setting the PE core numbering before running the application code).

### Host services

Once the application program is running it will need to access host resources, such as the file system. To support this, a protocol is defined between the run-time libraries and a device driver on the host system. This is interrupt based and allows the code running on the processor to make calls to an application or the 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 control 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

To illustrate the use of the multi-threaded array processor in a real device, the architecture of the CSX600 is described in this section.

The CSX600 is fabricated on a  $0.13\mu$  process and runs at clock speeds between 200 MHz and 250 MHz.

### CSX600 architecture

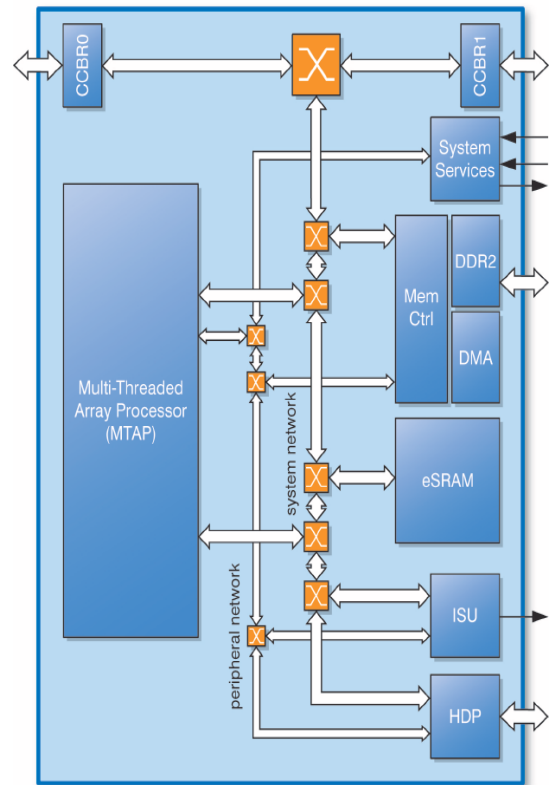


FIGURE 11: CSX600 SYSTEM-ON-CHIP (SOC) ARCHITECTURE

### Processor core

The MTAP processor in the CSX600 has the following specification:

#### General

- 8 Kbyte instruction cache: 4-way, 512 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 bit FPU
- 64 byte register file
- Support for 8 threads

### **Poly execution unit**

- Array of 96 PE cores
- Superscalar 64 bit FPU on each PE core
- 16 bit MAC per PE core
- 6 Kbytes SRAM per PE core
- 128 byte register file per PE core

### **Performance**

- Over 25 GFLOPS single or double precision
- 25,000 MIPS
- 25 GMAC/s integer multiply-accumulate
- 96 Gbytes/s internal memory bandwidth

### **Memory system**

External memory is connected via a 64 bit DDR2 SDRAM interface. When used with 72 bit wide DRAM modules this provides Error Checking and Correction (ECC). Each processor supports up to 4 Gbytes of local DRAM.

The processor supports 64 bit addressing so that large data sets can be processed. The 64 bit address space is flexibly mapped into a 48 bit physical address space. For embedded systems and backward compatibility, a simple 32 bit addressing mode is provided.

On-chip SRAM is included for fast access to frequently accessed code and data.

The integrated DMA controller can be programmed to transfer data to and from the external memory interface and any other device on the ClearConnect Network-on-Chip (NoC, whether on the same chip or elsewhere in the system).

### **Bridge ports**

The NoC is made available at two ports which can be interconnected with no glue logic to construct multi-processor systems.

This enables system performance to be scaled to meet the requirements of the application.

These ports use double data rate interfaces to minimize the pin count. If not fully used, they can be selectively turned off to reduce power consumption.

Data can be transferred directly from one bridge port to the other without impacting other transactions on the ClearConnect NoC. This allows efficient multi-processor systems to be constructed.

The bridge ports can also be used to connect to a Field Programmable Gate Array (FPGA) to provide other functions and interfaces.

### **Interrupt and Semaphore Unit (ISU)**

The Interrupt and Semaphore Unit supports the synchronization of threads with external events. Both pin and message signaled interrupts are supported for flexible support of multiple devices in various host environments.

### **Host/debug port (HDP)**

A host interface allows the CSX600 to communicate with, and be controlled by, the system's host processor. This port can also be used as a hardware and software debug port as it provides full access to all the internal registers on the device.

### **ClearConnect<sup>®</sup> NoC**

The various subsystems are interconnected using the ClearConnect on-chip network to interface the subsystems on the chip. This supports multiple independent data transfers; for example, the processor can access data in the on-chip SRAM at the same time as data is transferred to the DDR2 interface from one of the bridge ports.

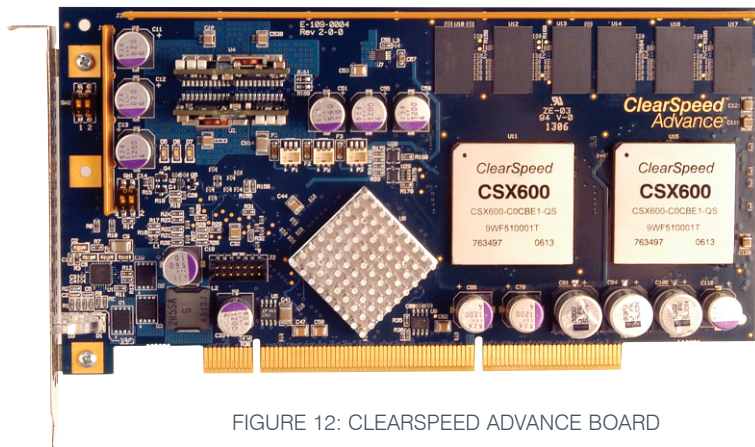


FIGURE 12: CLEARSPEED ADVANCE BOARD

## Applications

### High Performance Computing

ClearSpeed's CSX processors are used as application accelerators for HPC, delivering performance that unlocks the development of completely new algorithms for applications such as financial modeling and a host of scientific research disciplines. Because of its low power consumption, the technology can be applied to systems from workstations to clusters delivering compute performance beyond the limits of standard systems.

### Application Acceleration

In HPC systems, the CSX600 is typically used as a coprocessor to accelerate the inner loops of an application. The CSX600-based ClearSpeed Advance accelerator board is shown in Figure 12.

The use of the CSX600 as a coprocessor is illustrated in Figure 13. Here an application is running on the host system. Key functions of the application are ported to the CSX600 and the main program effectively calls the code on the CSX600 via a communication library. This sends the data to be processed from host memory to the ClearSpeed Advance board and waits for the results. By pipelining multiple operations, it is possible to overlap the communication with computation.

Libraries will be provided to allow the code on the coprocessor to be called from C, C++ or FORTRAN.

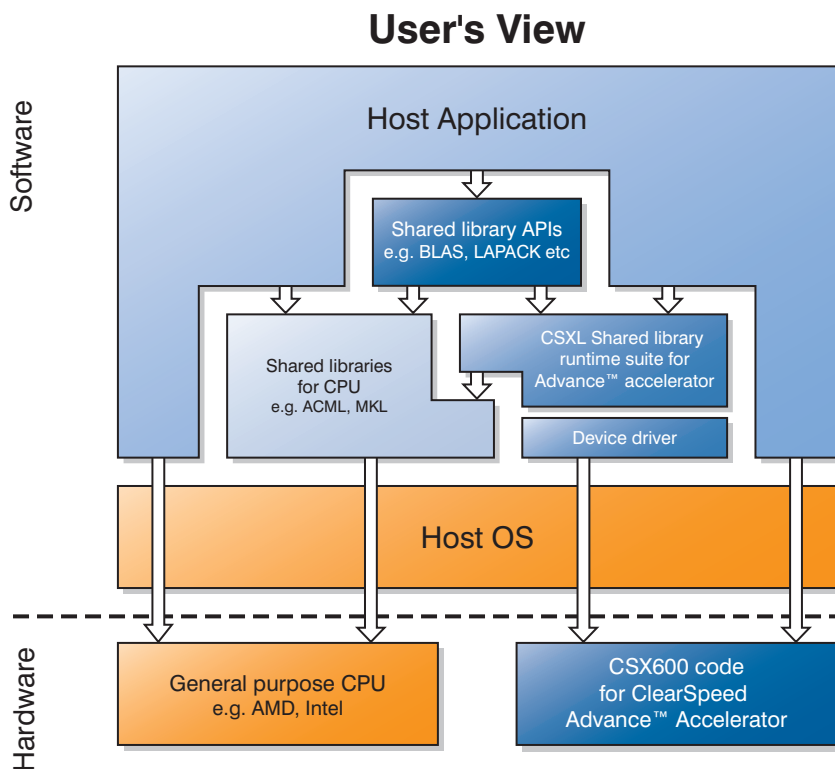


FIGURE 13: CLEARSPEED ADVANCE™ APPLICATION ACCELERATION

### **Example: Drug Docking**

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

Bristol University's Department of Biochemistry has produced their own software, called Dockit, for this application. This FORTRAN code performs molecular simulation of 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 naturally results in a massively data-parallel problem ideally suited to the CSX processor architecture.

ClearSpeed worked with the team at Bristol University to identify the functions in the program that take the most compute time. These were then rewritten in C for the CSX600 and the original FORTRAN sub-routines replaced with code to communicate with the CSX processor; passing the data to be processed and getting the results back.

On the CSX600, each PE core 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 core 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 atom data is effectively 'streamed' through the PE cores. The fetching of atoms can be efficiently overlapped with the atom-atom processing. However, even with the floating point accelerated processor the task is compute bound so I/O bandwidth is not an issue.

The calculations currently performed in most applications of this sort are a simplistic

model of the interaction of atoms. The processing power that the CSX600 makes available will allow more sophisticated models to be developed (e.g. taking into account quantum effects), improving the accuracy and thus value of the results.

### **Embedded systems**

The low power dissipation of the processor is also a key benefit in embedded systems where it can be used as a DSP in applications such as defense (e.g. radar and sonar processing) and medical imaging.

### **Communications**

The array processor architecture is ideally suited to packet processing for networking systems as well as portable and satellite communications.

### **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 done in real time.

In the network processing application, an I/O channel can be used for continuously streaming packets into, and out of, PE core memory. Within each PE core, 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.

PIO channels can be used by the PE cores to send requests to another subsystem that handles lookups. This could on-chip or off-chip, using table lookup hardware or a solution such as Content Addressable Memory (CAM).

## Software development kit

For customers who need greater flexibility or wish to port proprietary code, ClearSpeed provides a set of software development tools.

The ClearSpeed Software Development Kit (SDK) is a suite of tools and libraries designed to enable the rapid development of application software for CSX processors. The tools are centered around a C compiler and the industry-standard GDB symbolic debugger.

## Conclusion

The twin demands of functionality and ever increasing data volumes demand powerful yet flexible processing solutions. ClearSpeed's CSX processors supply a solution to these next generation needs in the form of a hardware and software platform that can be rapidly integrated with new and existing industry standard systems.

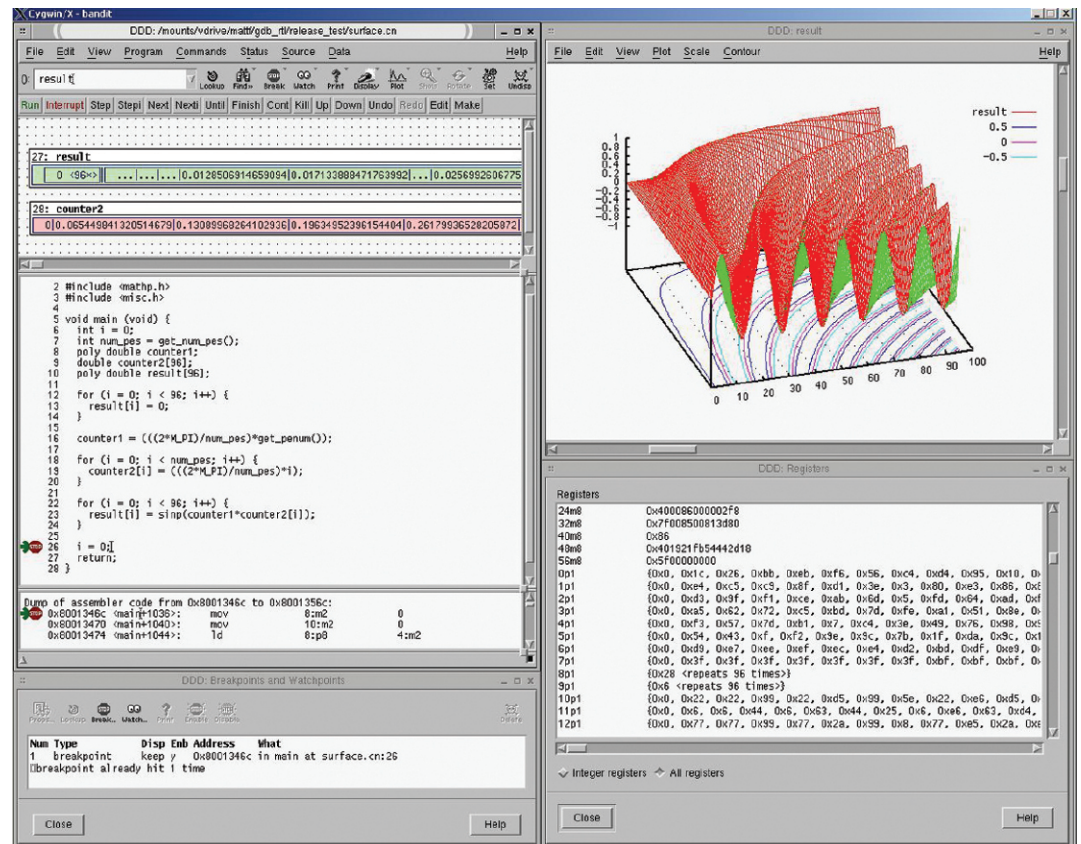


FIGURE 14: CLEARSPD DEBUGGER

Copyright 2007 ClearSpeed Technology plc ("ClearSpeed").

All rights reserved.

All information in this document is provided only as general information in connection with ClearSpeed products. Except as provided in ClearSpeed's terms and conditions of sale for such products, ClearSpeed assumes no liability whatsoever, and ClearSpeed disclaims any express or implied warranty relating to sale and/or use of ClearSpeed products, including liability or warranties relating to fitness for a particular purpose, merchantability, or infringement of any patent, copyright, or other intellectual property right. ClearSpeed may make changes to specifications, product descriptions, and plans at any time, without notice.

ClearSpeed, ClearConnect and Advance are trademarks or registered trademarks of ClearSpeed Technology plc or its group companies. All other marks are the property of their respective owners.

**PN-1110-0702**