*International Journal of Parallel Programming, Vol. 34, No. 4, August 2006 (© 2006)*
DOI: 10.1007/s10766-006-0019-9

# Deterministic Parallel Processing

Gajinder Panesar,[1,2] Daniel Towner,[1] Andrew Duller,[1] Alan Gray,[1] Will Robbins[1]

*Received July 1, 2005; accepted June 1, 2006*

In order to address the problems faced in the wireless communications domain, picoChip has devised the picoArray™. The picoArray is a tiled-processor architecture, containing several hundred heterogeneous processors, connected through a novel, compile-time scheduled interconnect. This architecture does not suffer from many of the problems faced by conventional general purpose parallel processors and provides an alternative to creating an ASIC. The PC102 is the second generation device from picoChip containing 308 processors. The devices are designed to be connected together using a seamless extension of the internal interconnect structure. This enables multi-chip solutions to be easily realised for applications which require additional processing. This paper highlights some of the difficulties encountered when building parallel systems and goes on to show how the features of the picoArray allow deterministic processing to be achieved, how the tool chain allows programming to be performed effectively in a combination of high level assembly language and C, and how systems built around the picoArray are debugged in real-time. By handling a wide variety of types of processing within the picoArray a single design flow can be used to produce complex communications systems. The effectiveness of this approach is demonstrated through the use of the picoArray to build a 802.16 base-station for commercial deployment.

**KEY WORDS:** Deterministic; interconnect; wireless; hetrogeneous.

## 1. INTRODUCTION

In a field where no single standard exists, wireless communications systems are typically designed using a mixture of DSPs, FPGAs and custom ASICs, resulting in systems that are awkwardly parallel in nature. Due to

[1]picoChip Designs Limited, Riverside Buildings, 108 Walcot Street, Bath, UK.
[2]To whom correspondence should be addressed. E–mail: gajinder.panesar@picochip.com

the state of flux of standards, if there are any, it is very costly to enter the market with a custom ASIC solution. What is required is a scalable programmable solution, which can be used in most, if not all areas. To this end picoChip created the picoArray and a rich toolset.

The picoArray is a tiled processor architecture in which hundreds of processors are connected together using a deterministic interconnect.[1,2] The level of parallelism is relatively fine grained with each processor having a small amount of local memory. Each processor runs a single process in its own memory space and they use "signals" to synchronise and communicate. Multiple picoArray devices may be connected together to form systems containing thousands of processors using on-chip peripherals which effectively extend the on-chip bus structure.

In order to provide a commercially viable, massively parallel, scalable solution, picoChip has had to re-think methods of debug and verification in several areas.

## 2. OVERVIEW OF THE PICOARRAY ARCHITECTURE

The picoArray is a tiled processor architecture in which 308 heterogeneous processors are connected together using a deterministic interconnect as shown in Fig. 1. The interconnect consists of bus switches joined by picoBus$^{TM}$ connections. Each processor is connected directly to the picoBus above and below it (an enlarged view of part of the interconnect is shown in Fig. 2, to simplify the diagram only two of the four vertical bus connections are shown).

There are three RISC processor variants which share a common core instruction set, but have varying amounts of memory and additional instructions to implement certain wireless baseband control and digital signal processing functions. A brief description of the three processor variants and a breakdown of the internal memory distribution is given in Table I.

The routing strategy used was determined largely by the real-time nature of the intended applications where the indeterminate latency due to bus arbitration would be unacceptable. All of the communications are determined during the "compilation" of the system which means that the communications bandwidth can be guaranteed.

### 2.1. Interconnect

Within the picoArray, processors are organised in a two dimensional grid, and communicate over a network of 32-bit unidirectional buses (the picoBus$^{TM}$) and programmable bus switches. The physical interconnect

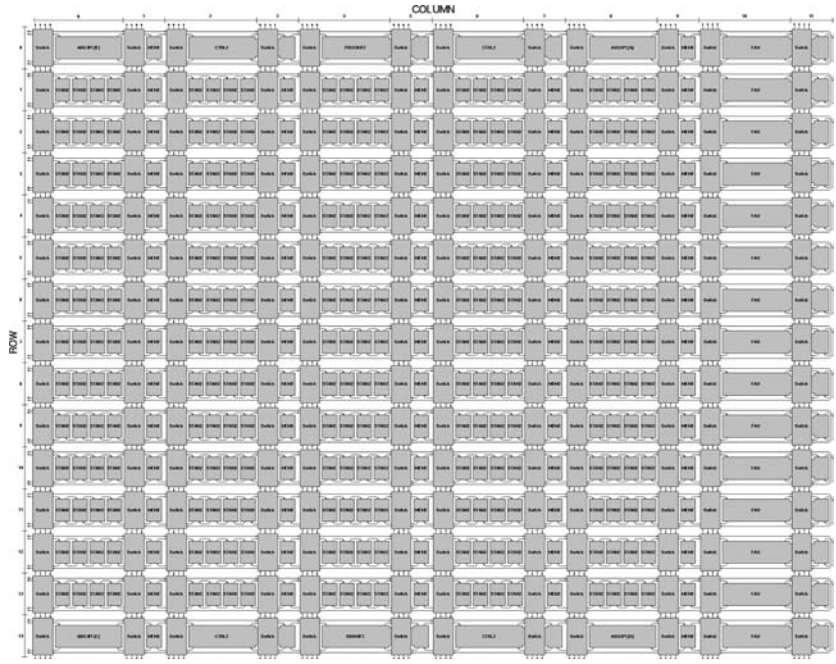


Fig. 1. Top-level diagram showing processors and interconnect.

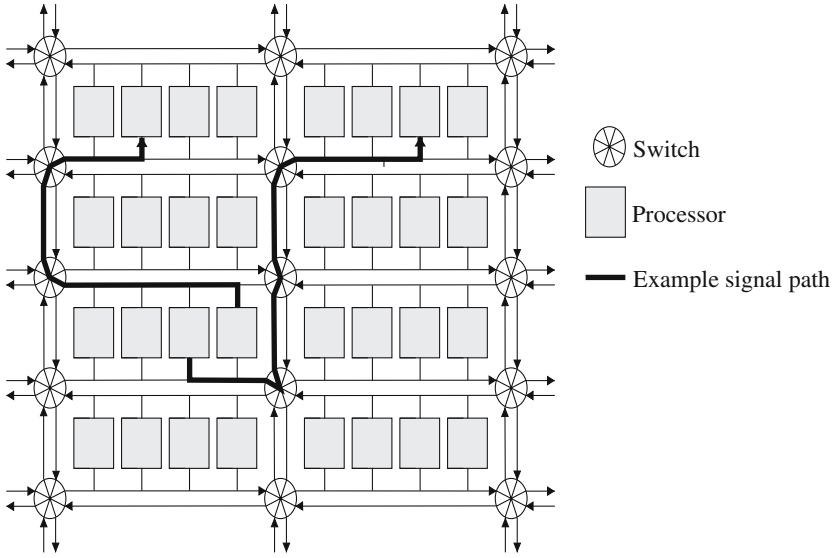


Fig. 2. Interconnect.

**Table I. PC102 Processor Variants and Memory Distribution**

| Type | Description | Number | Memory (bytes) |
|---|---|---|---|
| STAN | *Standard*: A standard processor optimised for CDMA spread and de-spread and other wireless base station signal processing functions. Includes a multiply accumulate unit and which supports additional multiply accumulate instructions | 240 | 768 |
| MEM | *Memory*: A processor having a multiply unit and additional data memory | 64 | 8,704 |
| CTRL | *Control*: A processor with a multiply unit and larger amounts of data and instruction memory optimised for the implementation of control functionality | 4 | 65,535 |

structure is shown in Fig. 2. The processors are connected to the picoBus by ports which contain internal buffering for signal data. These act as nodes on the picoBus and provide a simple processor interface to the bus based on *put* and *get* commands. The processors are essentially independent of the ports unless they specifically use a *put* or a *get* instruction.

The inter-processor communication protocol implemented by the picoBus is based on a time division multiplexing (TDM) scheme. There is no run-time bus arbitration, so communication bandwidth is guaranteed. Data transfers between processor ports occur during specific time slots, scheduled in software, and controlled using the bus switches. Figure 2 shows an example in which the switches have been set to form two different signals between processors. Signals may be point-to-point, or point-to-multi-point. Data transfer will not take place until all the processor ports involved in the transfer are ready.

Communication time slots throughout the picoBus architecture are allocated according to the bandwidth required. Faster signals are allocated time-slots more frequently than slower signals. The user specifies the required bandwidth for a signal by giving a rate at which the signal must communicate data. For example, a transfer rate might be described as @4, which means that every fourth time-slot has been allocated to that transfer.

The default signal transfer mode is synchronous; data is not transfered until both the sender and receiver ports are ready for the transfer. If either is ready before the other then the transfer will be retried during the next available time slot. If, during a *put* instruction no buffer space is available then the processor will sleep (hence reducing power consumption)

until space becomes available. In the same way, if during a *get* instruction there is no data available in the buffers then the processor will also sleep. Using this protocol ensures that no data can be lost.

## 2.2. Processors

All of the processors in the picoArray are 16-bit, and use 3-way VLIW scheduling. The basic structure of the processors is shown in Fig. 3. Each processor has its own small memory, which is organised as separate data and instruction banks (i.e. a Harvard architecture). The processor contains a number of communication ports, which allow access to the interconnect buses through which it can communicate with other processors. Each processor is programmed and initialised using a special configuration bus. The processors have a very short pipeline which helps programming, particularly at the assembly language level. The architecture of the three processor variants are shown in Fig. 4.

In addition to the general purpose processors, there are a number of special peripherals, including a host interface, an SRAM interface, asynchronous data and inter-picoArray interfaces. These peripherals are connected to the bus structure through ports, which enables them to be treated as though they are special purpose processors. Processors, peripherals and others with ports in a picoArray are referred to as Array Elements (AEs). The overall distribution of processors and peripherals is shown in Fig. 1 with the peripherals being placed in the top and bottom rows of the array.

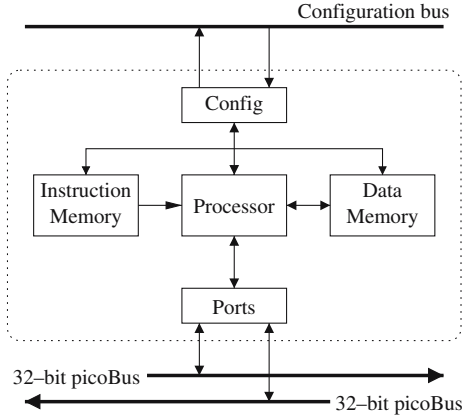


Fig. 3. Processor structure.

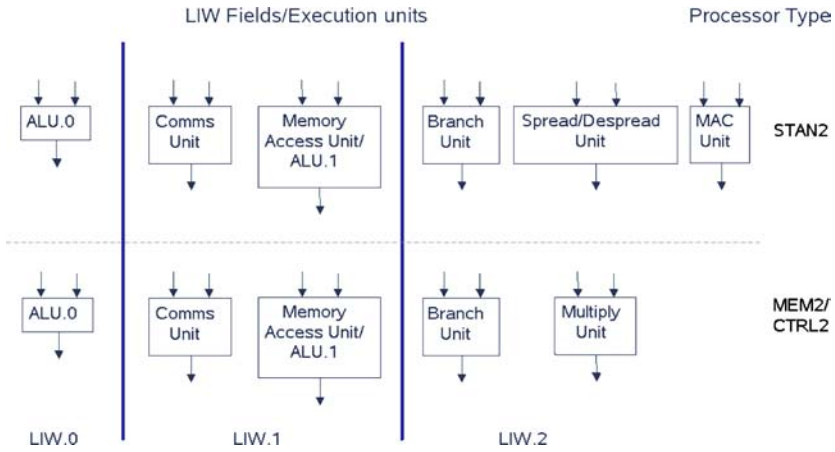


Fig. 4. VLIW and execution unit structure in each processor.

## 2.3. Host Interface

The Host or microprocessor interface is used to configure the picoArray device and to transfer data to and from the picoArray device using either a register transfer method or a DMA mechanism. The DMA memory-mapped interface has a number of ports mapped into the external microprocessor memory area. Two ports are connected to the configuration bus within the PC102 and the others are connected to the internal picoBus. These enable the external microprocessor to communicate with the internal AEs using signals.

## 2.4. SRAM Interface

Each picoArray has an amount of memory distributed amongst the processors for data and instruction storage. However, an external SRAM interface is provided to supplement the on-chip memory. This interface allows processors within the core of the picoArray to access external SRAM across the internal picoBus.

## 2.5. Asynchronous Data/Inter-picoArray Interfaces

There are four interfaces on each device which can be configured in one of two modes: either the inter picoArray interface (IPI) mode or the asynchronous data interface (ADI) mode. The choice of interface mode is made for each interface separately during device configuration.

### 2.5.1. Inter picoArray Interface

The four IPI interfaces are bidirectional and designed to allow each picoArray to exchange data with up to four others. Using this feature, a grid of picoArray devices can be constructed to implement highly complex and computationally intensive signal processing systems. The IPI interface operates in full duplex, sending and receiving 32-bit words. The 32-bit words on the internal picoBus are multiplexed as two 16-bit data on the interface itself.

### 2.5.2. Asynchronous Data Interface

The ADI allows data to be exchanged between the internal picoBus and external asynchronous data streams such as those input and output by data converters or control signals between the base band processor and the RF section of a wireless base station.

## 2.6. Functional Accelerator Units

Using experience gained from the first generation device, picoChip developed a new special purpose AE for PC102 called the Function Accelerator Unit (FAU). There are 14 FAUs present on the PC102. Each FAU includes configurable hardware for accelerating a number of compute-intensive tasks, including correlation and trellis processing. When configured in trellis-processing mode, the FAU provides support for log MAP decoding, Viterbi decoding, trellis coded modulation decoding and fast Hadamard transforms. Each FAU provides support for up to 64 trellis states, and multiple FAUs can be interconnected to support trellis codes with up to 512 states. Using the FAUs, a single PC102 can handle Viterbi decoding with output data rates in excess of 100 Mbit/s, leaving most of the picoArray free for other tasks.

## 2.7. Low-power Considerations

Potentially, a device such as the PC102, which contains 322 AEs and a TDM interconnect, could use a lot of power. A number of methods have been used to enable the power consumption of the picoArray to be reduced. For example, individual processors are able to ‘sleep’ when they are waiting for communications events, thus consuming minimal power, and parts of the picoArray which are not used in a particular design are switched off.

## 2.8. Array Layout

The target applications for the picoArray are fairly varied although they have many common attributes. The layout of the array in terms of quantities and locations of processors was determined to match these attributes as far as possible. Since any realistic system will make use of many picoArray devices, 4, 8, 16 or more, the layout has to be a compromise between the different types of processing that are required within a wireless infrastructure system. At the input to the system the data rates are very high but the processing is simple, as data flows through the system the data rates reduce, the control becomes more complex and the operations become more complex.

Figure 5 shows a typical piece of processing from the front end of the receive chain in a base station. At the input to the system, through the ADI, the data rate is high during which the incoming signals are filtered. This is then transformed into a symbol rate stream of data at a much lower rate, this rate can vary between the 960 kHz given, down to 15 kHz depending on the type of signal. It should be noted that this rate is for each user of the base station and typically there will be 64 users or more. In addition, multiple antennas will be used making the initial input data rate even higher.

The majority of the array, 240 processors (STAN processors), are designed for stream based processing and therefore have small amounts of local memory. The target applications tend to have a smaller requirement for block based processing and this is supported by the 64 MEM processors which have more local storage and can be used in conjunction with the SRAM interface. The low level control requirements of application systems are handled by the provision of 4 CTRL processors. Each processor has a performance comparable to an ARM 9 for control type functionality, or a TI C55XX for DSP tasks.

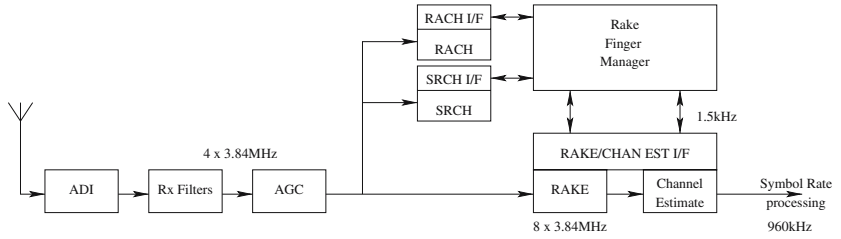


Fig. 5. Typical data rates.

## 3. PUSHING THE EDA ENVELOPE

This section highlights some of the practical considerations necessary to implement a large chip such as PC102 in small geometries like 130 nm.

### 3.1. Hierarchical Approach

PC102 contains over 200 million transistors; even using ASIC design techniques this causes large problems to EDA tools. This problem was considered up front in the design process. The massive parallelism of the picoArray was exploited to divide the device into so called tiles. Each tile consisted of five AEs. A tile was the atomic element to pass through the backend EDA flow. It was a suitable size to make all tasks from synthesis to design rule checking managable in terms of memory footprint and run time.

Ninety of these tiles were then assembled channelessly, i.e. they were butted, to form over 99.5% of the device. The remainder of the device was simply padring. In this way vast EDA tasks that could have taken days or weeks to run were removed from the design process.

### 3.2. Tiles

Tiles were designed to be complete encapsulated entities. They could be signed off for timing, DRC, LVS, power and clocking independently of the remainder of the device. They were self-contained P&R tasks, in fact the tiles were the largest gate count P&R tasks in the device. Their independence was achieved by making them all follow the same template for power routing, signal routing and timing constraints. Everything was included in the tile including feed though test circuity, configuration busses, all power routing and even a "gutter" to route the top level clock in, so that assembling tiles assembled the device. Tiles were effectively made self-contained in terms of power by connecting each of them to the power and ground in the package using flip-chip bumps. This reduced the need for chip wide power analysis and ensured minimal ground bounce. By considering the detailed floorplan when designing the logic ensured that high level routing and timing problems at the top level were solved at the simpler lower levels.

### 3.3. Clock Distribution

PC102's clocking is completely synchronous, that is to say all logic is running from a single clock. Broadcasting a single clock over a large

area with minimal skew is a challenge. The initial approach was to ensure that the circuitry at the tile interfaces was tolerant of moderate skews. This is inherent in the picoBus since timing analysis of the paths across it are done for every user design. However clock skew can have the effect of shortening apparent clock periods, this affects the distance a signal can travel on the picoBus which in turn can cause congestion on the buses. Hence the secondary approach was to reduce top level clock skew by laying it out as a balanced H-tree in the gutters between tiles. The regular tile power structure provided shielding from fast switching signals which could have "crosstalked" onto the clock causing further skew.

By considering all of the key circuits in terms of logical and physical aspects early on in the design, and understanding where the EDA tools should be given freedom and where they should be manually constrained, it was possible to produce a extremely large, powerful and highly utilised design.

## 4. PICOARRAY DEBUG AND ANALYSIS

The debug and analysis of parallel systems containing perhaps thousands of processors requires specialised tool support. This section describes a number of ways in which this has been achieved.

### 4.1. Language Features

The language features aid verification and integration through three main features: strong type checking, fixed process creation, and bandwidth allocation.

Strong type checking is used to ensure that whenever data is communicated from one process to another, the data will be interpreted by both producer and consumer in the same way. Types are selected from a library of built-in types, or by the users defining their own types. Types used in communication are limited to 32-bits, which is the maximum size which may be transferred in a single communication over the picoBus. At the structural level, processes will be defined with ports of specific types, and they will be connected with signals which must match the port types. Within a process, any data which is "put" or "get" from a port must be of the correct type. For processes written in C, this is achieved by synthesising the available types using C encoding rules (e.g., using typedef's, union's, and struct's), and hence tying into the C compiler's type system. Thus, end-to-end communication of data can only occur when all processes and signals agree on the type format. This makes integration of

independently developed components easy since any discrepancies in type formats will be detected at compile time, when they are easily fixed.

The structural VHDL used to define a system requires the number of processes, and their interconnections to be fixed at compile time. During compilation, the tools will allocate each process to its own processor, and schedule the signals connecting the processes onto the picoBus interconnection fabric. Because of this compile-time scheduling, non-deterministic runtime effects such as process scheduling, or bus contention have been eliminated. This makes it easier to integrate systems. If problems are found, it also makes the reproduction of the problems, their debugging and the verification of their fixes easier.

In addition to specifying fixed signals connecting processes, the signals are also allocated bandwidth. This is achieved using a language notation which allows the frequency of communication over the signal to be specified. Processes requiring high signal bandwidths will use high frequencies (e.g., every 4 cycles), while processes requiring low bandwidth will use low frequencies (e.g., every 1024 cycles).

## 4.2. Design Browser

The design browser is a tool which allows the user's logical design to be viewed graphically and can be used both during simulation and when executing a design on hardware. There is a number of different graphical views.

For example the hierarchical view mirrors the structural hierarchy that was created by the user and allows each level of this hierarchy to be explored. An example of this is shown in Fig. 6.

In addition to the static features the design browser can provide dynamic information about the each instance in a design, for example whether it is processing or waiting for a communications operation. An example of this display is shown in Fig. 7 (in fact the boxes are coloured, green for processing, red for waiting on communications, but in this paper are grey and dark grey respectively).

## 4.3. Simulation

The cycle accurate simulation system allows users to build, test and verify their entire design before moving to the hardware. The user is able to extract the state of the system (on a cycle-by-cycle basis) in order to check against the behaviour on hardware. Importantly, the same simulation system was used to provide a "golden reference" during the design and verification of the PC101 and PC102 chips.

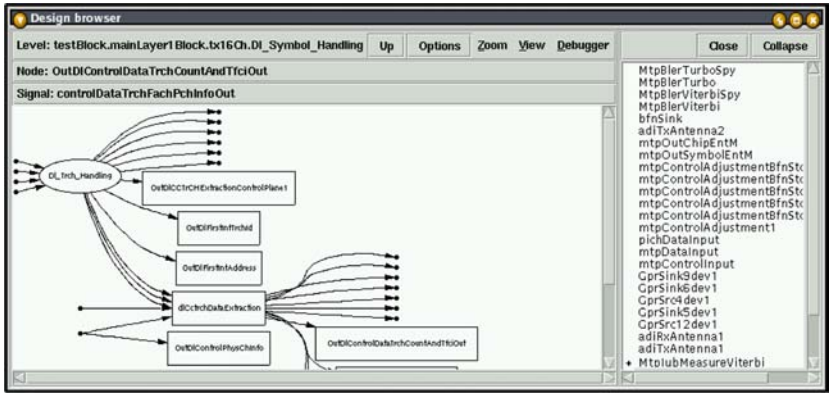


Fig. 6. Design browser hierarchical display.

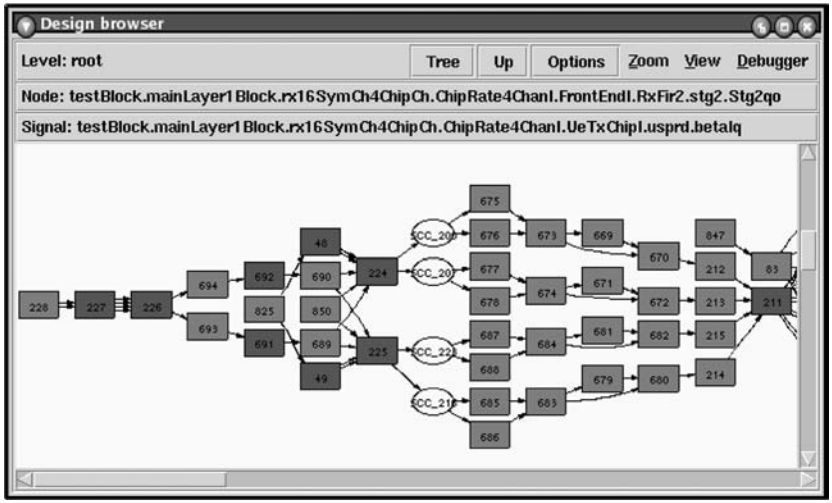


Fig. 7. Design browser strongly connected component display.

The same source-level debugging interface exists on the hardware as on the simulator enabling the user to migrate from one environment to the another without making any changes to their design or their testbenches.

## 4.4. Scripting

While debugging large parallel systems, operations such as viewing the source code or variable values of individual processes become too low

level; this is analogous to debugging a compiled process by inspecting its raw machine code and register values. For large parallel systems it is more convenient to be able to abstract the debugger to provide a higher system-level interface. Such an interface allows the details of individual processes to be hidden, and replaced by system-specific displays instead. Clearly, it is impossible for picoChip to provide interfaces for every possible system, so instead the debugger can be programmed using Tcl/Tk.[3] This allows the users to create their own system-specific interfaces, built on top of the picoChip debugger. Figure 9 shows an example of a WiMax system interface.

### 4.5. Traffic Analysis

Traffic analysis is used to monitor the state of the communications network. The AEs in a picoArray use signals to communicate data (and hence synchronise), and traffic analysis can indicate to the user the states of the communication at any particular time. The relevant data is extracted from selected AEs or from all AEs in a design and either displayed immediately or stored to a file.

For each signal the maximum bandwidth of a signal has to be specified at design time but of particular interest to the user is the actual bandwidth used on a given signal. Using the traffic analysis data can provide information on the statistics of the bandwidth used and can help in the analysis of deadlock and livelock problems.

### 4.6. FileIO

When testing and debugging it is common to wish to use Unix files in order to inject data into a system or to record intermediate results. This is achieved by providing an AE template which interfaces to the picoBus in the usual way using signals but which is also "connected" to a Unix file. The advantage of this method is that the same user's code can be used whether the system is running as a simulation or on hardware. The FileIO AE has two different implementations, one for simulation and one for hardware. In a simulation the connection to the file is trivial since the simulation simply consists of a piece of compiled C++. In hardware the data memory of the AE is used to buffer the data and when the AE requires it must request that the debugger either empty its memory (for an output FileIO) or fill its memory (for an input FileIO).

### 4.7. Traces

Traces allow specific types of data, such as register and memory contents, or signal values to be recorded during execution. The trace is stored as a sequence of tuples recording changes in value against the time at which that change occurred. This sequence can be saved to a file, and used by external programs such as gtkwave.[4] The tracing tool is used in a way that is similar to a hardware engineer using an oscilloscope to probe data paths in an electronic circuit. The trace allows a visual representation of the data to be shown with respect to time, which can make certain types of bug readily apparent. Tracing can also be used to perform code profiling, by tracing how the program counter changes over time, and post-processing the information to relate it to the original source code.

While many general purpose processors use special hardware to implement tracing (e.g., ARM Embedded Trace Macrocell[5]) the picoArray devices do not. One reason for this is that traces can generate huge quantities of data (e.g., tracing the program counter for a single processor would generate $3.2 \times 10^8$ bytes/s). While the picoArray devices have impressive internal communications bandwidth, it would be impossible to transfer this much data off chip without affecting the system being debugged.

Two mechanisms are used to perform tracing. Signals are traced using probes, which are described in Section 4.8. The probes mechanism allows signal traces to be performed while running a system at full hardware speed (160 Mhz) but the dumping of data to a file means that this speed cannot be sustained. All other types of data (general/special purpose registers, and blocks of memory) are traced using software. The debugger tool repeatedly single-steps the debug system, recording traced values after each step. This can be slow. Typically, the debugged system will be traced offline, and the results analysed using post-processing tools.

### 4.8. Probes

Probes are special purpose processes which the debugger inserts into the user's design by utilising unused processors. Probes can be connected to one or more signals, and can non-intrusively monitor all traffic which passes over the signals. They achieve this by using the bus interconnects ability to create 1-to-many connections. For example, suppose two processes in a system were connected by a 1-to-1 signal. If a probe is inserted during debugging to monitor that signal, the debug tools will change the 1-to-1 signal into a 1-to-many signal, with the probe acting as an extra destination. The original processes are unaffected by this change (both in

terms of latency and bandwidth), but the probe is now able to monitor all communication over that signal.

Probes are implemented as processes, and so can run at full hardware speed. This enables probes to be used to debug systems in real-time. One use for probes is to allow real-time signal traces to be performed. Other uses include signal assertions, and on-chip analysis.

Signal assertion probes can be used to check that the data passing over a signal conforms to some compile-time specified property. For example, all signals in picoArray devices have pre-allocated bandwidth. A signal assertion probe could be attached to a signal to record the bandwidth actually used, thus allowing signals with over-allocated bandwidth to be detected.

Probes can be used to perform on-chip analysis of signal data, rather than having to transport the data off-chip (e.g., using traces), for later analysis. For example, during the development of the picoChip base station, a probe was created which performed Bit-Error Rate (BER) computation on signals. These BER probes could be used to monitor the performance of the base station's Viterbi decoder's in real-time, under different system loads.

### 4.9. Activity Display

This is related to the trace facility but only looks at the type of activity being undertaken by an AE. This can be running, waiting on a communication or stalled on a memory pipeline fetch. This display allows the history of the activity of a number of AEs to be viewed.

## 5. PICOARRAY DESIGN METHODOLOGY

This section goes through a typical process that is used to create a picoArray based application.

### 5.1. System Decomposition

Typically this is done by hierarchically breaking down the problem into components consisting of processes connected by signals. Experience has shown that components generally contain a few tens of processes, however the number of processes required does not have to be specified at this stage. The boundaries of these components will also have signals defined and will eventually be connected to other parts of the system. The user will use knowledge of the real-time system being developed to specify signal properties, such as maximum bandwidth and signal type. The

properties can be checked during integration using signal assertions, which are described in Section 5.3.

## 5.2. Component Coding

Two approaches can be taken, the choice being dictated by the complexity of the component.

For small components in which the division into AEs can be determined easily these AEs can be coded using C or ASM and connected using appropriate signals.

For larger components it may be preferable to initially produce a functional representation using C. This can be simulated even when the code size exceeds the memory for any AE and allows functional testing of this component prior to its division into individual AEs.

Whichever approach is used the code can be tested by creating test harnesses using FileIO to mimic the external components. The symbolic debugger and its attendant tools can be used to find bugs within the AEs.

The migration of the code to hardware is eased by the fact that the same FileIO test harnesses produced for simulation can be used for verification. This highlights a huge advantage of the picoChip approach since testing on hardware can be performed at a very early stage which means that components can be tested for minutes or hours of real time which would be impossible using simulation.

Other components can be written in parallel by other developers, or sequentially by the same developer.

## 5.3. Small Scale Integration

As components are completed they can be integrated. The strong typing, bandwidth allocation, and fixed process creation ensure that components developed by different people will fit together properly. Signal assertions can be written to encode properties (such as signal value or minimum throughput) of the signals, and these can be checked during integration using assertion probes.

If integration fails (components fail to communicate properly), then this is caused by problems between components, rather than within a component (since the component has been verified in isolation, it has static processes, fixed local signals, etc.). The suite of system-wide tools (probes, traces, activity display, etc.) can be used to identify the problem.

### 5.4. Large Scale Integration and Performance Testing

This phase of development can only really be done on the hardware. At this stage all of the FileIO will have been replaced by real components.

It is important to be able to monitor aspects of performance in real-time and this can be done using customised probes which monitor various signals and compare data throughput against predetermined limits. In addition it is possible to monitor the behaviour of the system when processing real-world data, and to inject data by using the microprocessor interface. The results of the monitoring can be displayed using custom GUI's which the user can develop (an example of a custom GUI is shown in Fig. 9).

## 6. A DESIGN EXAMPLE

picoChip is a member of the WiMax (802.16) Forum[6] and is working with its customers to produce a 802.16 compliant system. The scope of the work is aimed at producing a system which can be used in either the base station or the consumer premises equipment (CPE) market. As part of this work, picoChip has developed the first part of this system solution — an 802.16 compliant PHY, whose functional decomposition can be seen in Fig. 8.

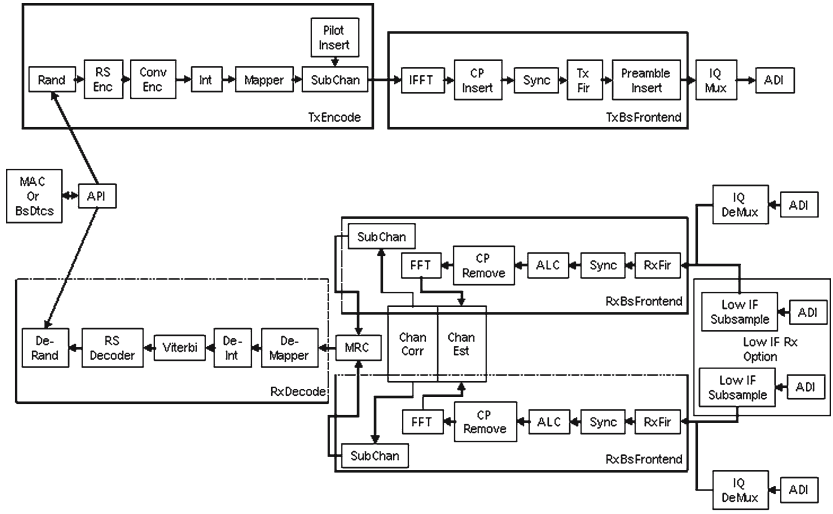


Fig. 8. Functional decomposition of an 802.16 PHY.

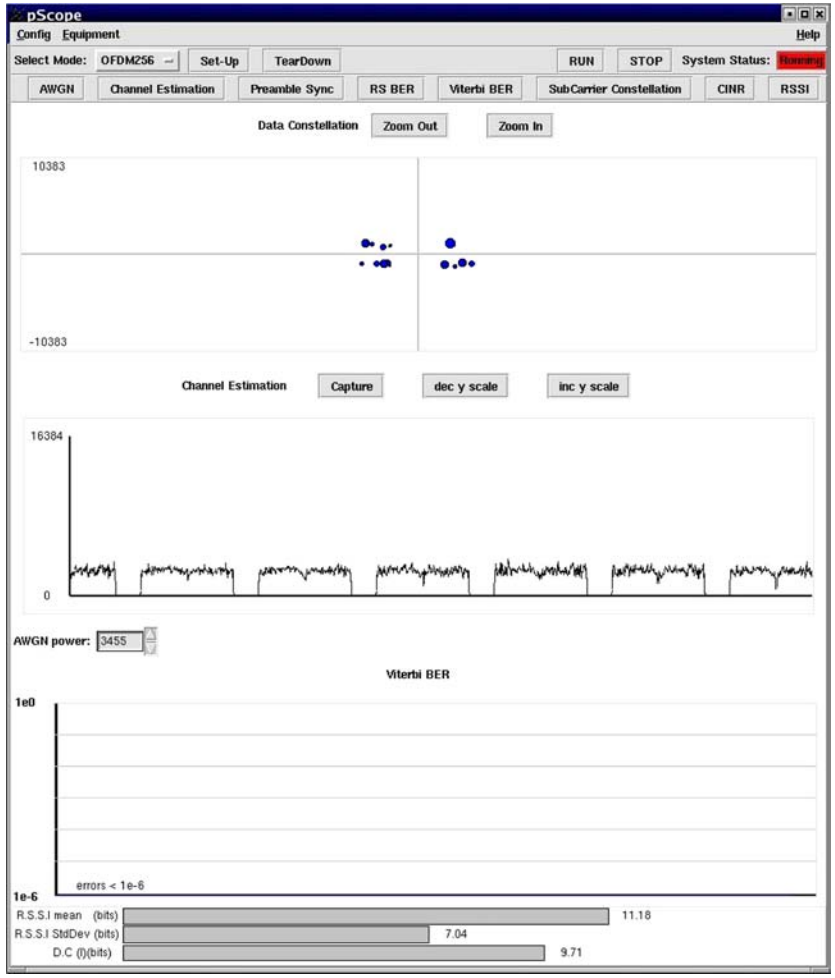


Fig. 9. Diagnostics output from 802.16 PHY.

Using the PC102 device the system's team at picoChip have used most of the debugging aids to implement this PHY. This includes developing the individual blocks, executing the implementation on the simulator and creating testbenches in order to verify the correct (compliant) operation before moving onto integration.

It is the final integration of the whole PHY system, for both transmit and receive, that illustrates the key aspects of the debugging environment. The result of this is best shown in Fig. 9 where the systems group

scripted an application specific GUI on top of the primitives provided by the toolkit and indeed the system debugging widgets. There are four areas of interest: three data output probes and one input probe.

The Data Constellation shows data captured by a probe at the output of the Channel Equaliser. This data is extracted in real-time and streamed out of the picoArray via the microprocessor interface.

The Channel Estimation shows the magnitude of preamble sub-carriers (consecutive preambles shown on plot) as captured by a probe at the output of the FFT.

The AWGN (additive white Gaussian noise) has been added to aid checking of the behaviour when there is noise in the channel. This injects data (noise) as input to the quad demodulation block.

The Viterbi BER display shows the BER at the output of the Viterbi, again in real-time, as captured by a probe.

Finally the RSSI (Received Signal Strength Indicator) display shows received signal statistics captured by a probe at the output of the ADI.

## REFERENCES

1. A. Duller, G. Panesar, and D. Towner, Parallel Processing – the picoChip way! In J.F. Broenink and G.H. Hilderink (eds.), *Communicating Processing Architectures 2003*, pp. 125–138, 2003.
2. P. Claydon, A Massively Parallel Array Processor, In *Embedded Processor Forum*, 2003.
3. J. K. Ousterhout, *Tcl and the Tk Toolkit*. May 1994.
4. The University of Manchester Advanced Processor Technologies Group. GTKWave Electronic Waveform Viewer. http://www.cs.man.ac.uk/apt/projects/tools/gtkwave/.
5. Embedded Trace Macrocell. http://www.arm.com/products/solutions/ETM.html.
6. Wimax forum. http://www.wimaxforum.org/home.