A System Solution for High-Performance, Low Power SDR

Yuan Lin[1], Hyunseok Lee[1], Yoav Harel[1], Mark Woh[1], Scott Mahlke[1],
Trevor Mudge[1] and Krisztián Flautner[2]

[1]Advanced Computer Architecture Laboratory
University of Michigan - Ann Arbor, MI
{linyz, leehzz, yoavh, mwoh, mahlke, tnm}@umich.edu

[2]ARM, Ltd.
Cambridge, United Kingdom
{krisztian.flautner}@arm.com

Abstract

One central challenge in the realization of Software Defined Radio (SDR) is to provide a programmable solution that meets the challenging high-performance, low-power requirements, while providing an efficient software development interface. In this paper, we present an overview of a fully programmable multi-core SIMD architecture for SDR. Our solution can support 2Mbps W-CDMA at about 270mW, and 24Mbps 802.11a at about 370mW in 90nm technology. This high computational efficiency is achieved by exploiting the vector characteristics of the algorithms, through a unique multi-core architecture that consists of tightly coupled scalar and wide SIMD pipelines. In addition, we present a software design flow that supports efficient DSP programming and implementation through a set of signal processing extensions to C, referred to as SPEX.

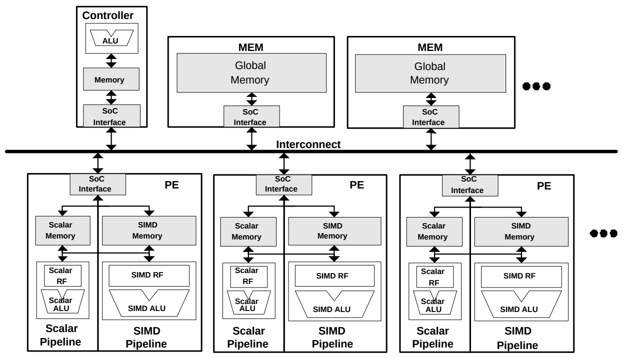
1 Introduction

Software Defined Radio (SDR) promises to revolutionize the communication industry by delivering low-cost, flexible software solutions for wireless mobile communication protocols. Wireless protocols are systems consisting of a collection of distinct DSP algorithms. The difficulties of implementing a complete system in software include challenges for both DSP hardware and software designers. In this paper, we present a system solution for SDR that includes a novel DSP processor architecture that is designed specifically for SDR, and a programming model that allows efficient DSP software development. We have developed the complete W-CDMA and 802.11a protocols' physical layers, programmed them onto our system, and shown that they achieve the required bandwidth and the power efficiency for mobile terminals.

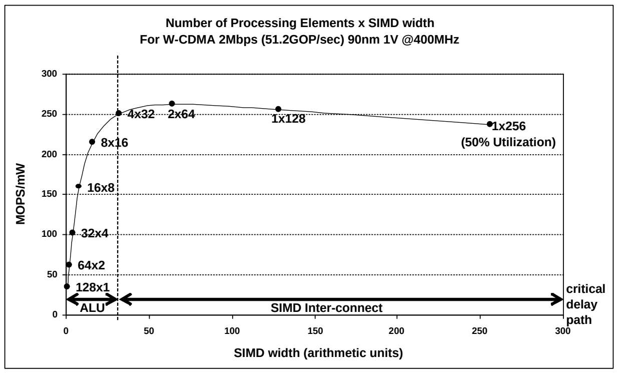
The two major challenges in SDR are the design of efficient hardware systems and software development environments.

- *Hardware requirements* for current and next generation wireless protocols are extremely high. SDR processors must achieve supercomputer-like computational throughput, maintain ASIC-like power consumption, meet the protocols' latency requirements, and support real-time systems with dynamically changing control states [4]. Existing and next generation DSP processors, such as the TI TMS320C64x [1] and the Freescale StarCore [2] are not designed specifically for SDR. They either consume too much power or do not meet the performance requirements. In addition, because wireless protocols are complex systems of many DSP kernels, it is also desirable for the hardware designers to provide an easy design interface for DSP software programmers. Some of the emerging SDR processor solutions meet the performance requirements, but are very difficult to develop or debug. Morpho Technology's RC Array [3] consists of 2D array of processing elements, which cannot be expressed efficiently in traditional C-like programming languages. Similarly, the PicoArray [5] also consists of an array of processing elements on which it is difficult to map applications.

- *DSP programming support* needs to provide an easy system development flow for the software developers. At the system-level, the software needs to provide a development flow similar to existing SoC development flows, where the inter-algorithm communication and protocol state control are developed and debugged. At the algorithm-level, programming support must generate efficient machine code for individual DSP kernels written in a high-level programming language. It also needs to be flexible enough to allow DSP programmers to develop hand-written assembly code for the extra optimizations. C is perhaps the most popular programming language in the DSP community even though it lacks some necessary features for properly describing the application domain. For example, it does not support first-class SIMD or concurrent function ob-



(a) Multi-core system architecture for SDR



(b) Effect of SIMD width on computational efficiency for W-CDMA 2Mbps

Figure 1: System Overview

jects. Matlab is another language which is very popular among DSP programmers. It provides SIMD-centric first class data structures and pipeline-level concurrency that can be expressed using Simulink. However, Matlab does not support explicit object definitions, including SIMD variable types, concurrent threads, and communication channels. The lack of this information makes it very hard for compilers to produce efficient assembly code.

In order to verify the efficiency of our processor architecture, we first implemented both the complete physical layer of a transmitter and receiver for W-CDMA and 802.11a in C. We then compiled both of these two protocols onto our processor architecture, and shown that our architecture was able to meet the performance and power requirements: 2Mbps W-CDMA at 270mW, 24Mbps 802.11a at 370mW. Our DSP system is a modular, multi-core DSP architecture where each DSP algorithm can be designed and verified individually and separately from the system-level development. Unlike other proposed multi-core SDR architectures, (e.g. [6] and [7]), each hardware component in our system has a standardized interface. DSP algorithms are mapped onto individual processors, not across multiple processors. Thus, the DSP kernel implementations can be developed and verified individually. System-level development can view these kernel codes as software ASICs, and control different kernels through a predefined standardized SoC interface.

We present our software development flow, which includes both the system-level and algorithm-level development flows. The central element of our software design is SPEX (Signal Processing EXtension), a set of language extensions for C, which narrow the semantic gap between DSP algorithm descriptions and their implementations. Like Matlab, SPEX provides built-in support for vectors (through the use of SIMD data variables), as well as SIMD data operations, such as vector permutation and vector predication. However, SIMD data variables carry additional attributes, such as data bitwidth for efficient implementation. SPEX also provides thread and communication objects, called kernels and channels. Kernel objects support dynamic thread spawning and deletion to account for dynamically changing workloads. SPEX channel objects are generalized FIFO (first-in, first out) structures that support random read access and SIMD objects as queue entries. In addition, global variables are disallowed, as all communication must be performed through channels. We believe these extensions provide an intuitive programming model for expressing high-throughput DSP applications, as well as an efficient interface for compiling to multi-core DSP processors.

2 Architecture Overview

2.1 Multi-core System

Our system is a heterogeneous multiprocessor architecture, shown in Figure 1(a). The system consists of multiple high throughput SIMD-based processing elements (PEs), a low throughput scalar controller, and global scratchpad memories (MEM). These components are all connected through a shared bus. PEs consist of tightly-coupled scalar and SIMD pipelines. The SIMD pipelines are generally used for computationally heavy DSP algorithms, such as filter, FFT, and channel decoders. The scalar pipelines are used for the sequential portions of algorithms and address generation for the SIMD pipelines. The controller is used for overall system management, such as power control. MEM is mainly used to buffer intermediate data transfers between DSP algorithms.

PEs are the main computation units in this system. They take the most area and consume the most power. The number of PEs and the architectural organization of these PEs are one of the main design considerations. Figure 1(b) shows an approximate efficiency trade-off for running W-CDMA protocols with multiple PE configurations, from left to right with increasingly bigger, but fewer processing elements. All of the configurations have constant computation throughput and meet the real-time W-CDMA requirements. As shown in the graph, configurations with a small number of wide SIMD units – 4x32 to 1x128 appear to be the most efficient. However, a wider SIMD architecture has greater of programming challenges. In most programs, it is very hard to find 128 independent data elements to compute in parallel. Signal processing algorithms have much inherent parallelism, ie, the taps of a filter, that can be calculated in parallel. But there are also many signal processing algorithms that do not have wide parallelism. In our case study with W-CDMA and 802.11a, we choose a design point near the inflection point of the graph: 4 PEs, each with 32-wide SIMD units.

In typical commercial wireless system solutions, low computation algorithms are handled by DSPs, high computation algorithms are designed with ASICs, and the whole system is an integrated SoC with a simple controller, such as an ARM processor. Given the complexity of these real-time systems, we want to separate the design of individual DSP algorithms from the design of the protocol system. In our SDR solution, each DSP algorithm is designed independently as a "software ASIC", with internal states and variables, and a communication interface to the outside world. System-level development, consists of linking these DSP algorithms together, mapping algorithms onto PEs, and defining real-time deadline requirements. Low computation algorithms, like filters and FFTs, may be combined together onto one PE. High computation algorithms, such as searchers, Viterbi decoders and Turbo decoders, generally require their own PE.

In order to support such a design methodology efficiently, each hardware component is designed with a standardized system interface. This interface includes both hardware requirements and software programming specifications. Any hardware units that are connected to the system has to support this interface. This is shown in Figure 1(a) as the "SoC Interface". The software specification is defined as a set of assembly instructions, including communication, synchronization, and memory access instructions. All processing elements must support these instructions with pre-defined timing requirements. The hardware implementation of the interface consists of a DMA (Direct Memory Access) unit, a real-time clock, and hardware synchronization registers.

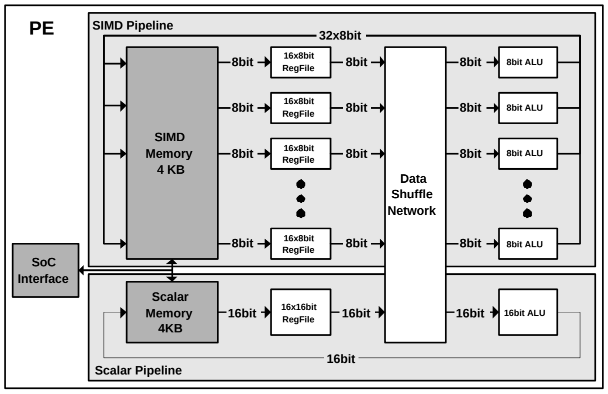


Figure 2: PE Architectural Diagram

2.2 Processing Element

Figure 2 shows the architectural detail of a PE. The PE consists of two coupled parts: a scalar pipeline and a SIMD pipeline. The scalar pipeline contains the address generation unit (AGU) and is a single issue, in-order, 16-bit RISC architecture. Its main purpose is to generate memory addresses for the SIMD pipeline, handle the kernel's control flow, and process scalar DSP algorithms (such as the interleaver). In most DSP algorithms, the core kernels are made up of shallow nested loops (one or two levels). Because of this, we choose not to implement a branch predictor, but add loop counter-based branch instructions instead. In addition, DSP kernels process data in stream buffers, thus most of the memory access are from data queues, which are directly supported by the AGU.

The SIMD pipeline consists of 32 8-bit clusters. Through the implementation of W-CDMA and 802.11a protocols, we found that most DSP algorithms have high degree of SIMD parallelism. The core operations of filter, FFT, Viterbi/Turbo decoder, and rake receiver all are based on wide vector variables of narrow data-width. Therefore, with the support of conditional operations on the clusters, we can efficiently utilize 32 clusters of 8-bit ALU computations. The PE's local scratchpad memory is divided into two clusters: one for the SIMD unit and the other for the scalar unit. Both memories have two read/write ports and there is a DMA engine that serves both memories.

Many DSPs have support for 8- and 16-bit operations. However, their clock cycle time is optimized for 32-bit arithmetic operations. This leads to lower power efficiency for 8- and 16-bit operations. In wireless protocols, the majority of the algorithms operate on 1- to 8-bit data, some algorithms operate on 16-bit data, and few operate on 32-bit data. Therefore, our system is optimized for 8-bit operations in the SIMD unit and 16-bit operations in the scalar unit. 16-bit support is provided in the SIMD unit by treating two register entries as one and using two cycles for 16-bit ALU operations (along with special hardware support for the carry in/out bits). The AGU registers are 12-bit, but only support 8-bit addition and subtraction. This is because AGU is mainly used for software management of data buffers, in which 8 bits are sufficient. The higher 4 bits are used to address different PEs, as well as different memory buffers within PEs.

3 System Evaluation

3.1 Wireless Protocol Mapping

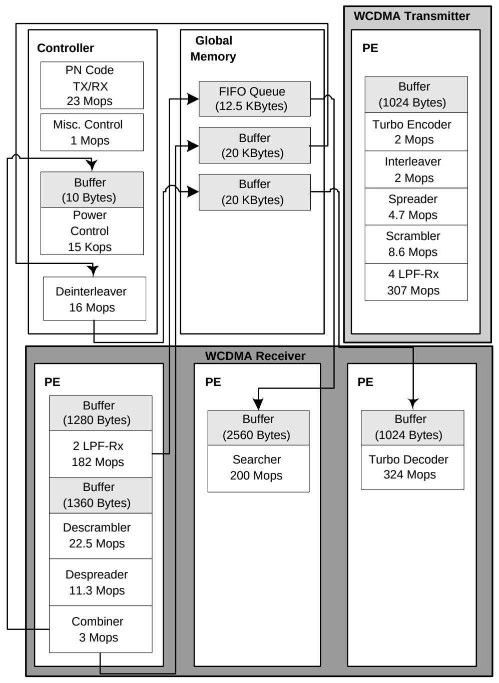
Figure 3 shows the mapping of W-CDMA and 802.11a onto our 4 PE system. As W-CDMA is a full duplex protocol, the receiver and transmitter are running at the same time. Because of this, the transmitter and receiver are mapped onto their own PEs for W-CDMA. This contrasts with 802.11, where the transmission and reception phases are disjoint in time and thus the kernels for these modes can share PEs. This provides for a more balanced task allocation.

W-CDMA Mapping. In W-CDMA, the receiver requires much more computation than the transmitter. As shown in Figure 3(a), the receiver is assigned to 3 PEs, and the transmitter is assigned to 1 PE. Global memory contains three buffers. The FIFO buffer is used to buffer results between the receiver FIR filter and the searcher. The other 2 buffers are used to store intermediate results between the Rake receiver and the interleaver, and the interleaver and the Turbo decoder.

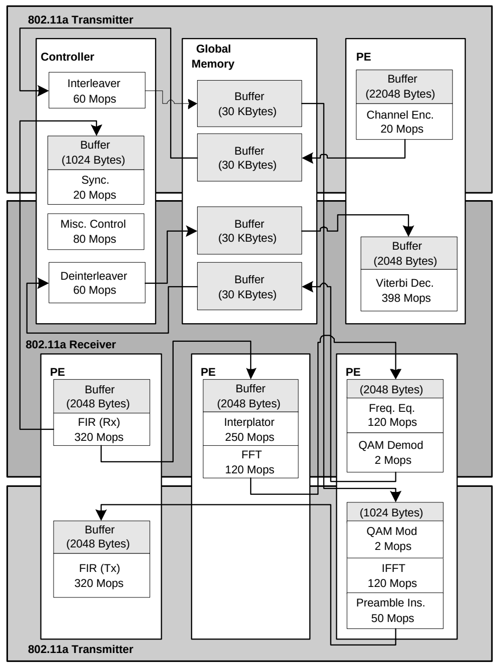
802.11a Mapping. In 802.11a, both receivers and transmitters are mapped onto the same set of hardware. Similar to W-CDMA case, global memory is mainly used to buffer the intermediate data traffic of the interleaver. Unlike most other algorithms, the interleaver is a highly sequential algorithm. It requires a whole frame to be buffered before it can output its results.

3.2 Area and Power Results

Table 1 shows the power consumption and area break down for a 2Mbps throughput W-CDMA and a 24Mbps throughput 802.11a. The overall power results were 1,381mW and 1,909mW for W-CDMA and 802.11a, respectively. This assumed 180nm technology at 1.8V and 400MHz. Scaling these results to 90nm technology at 1V and 400MHz results in power values of 268mW and 370mW. Three components consume the majority of the power: 1) the register file which consumes 34% for W-CDMA and 30% for 802.11a; 2) the



(a) W-CDMA Protocol Mapping



(b) 802.11a Protocol Mapping

Figure 3: Mapping of W-CDMA and 802.11a onto the processing elements

			Area		Power: W-CDMA 2Mbps		Power: 802.11a 24Mbps	
	Components	Units	Total Area mm^2	Area %	Total Power mW	Power %	Total Power mW	Power %
PE	Memory (12KB)	4	3.6	21%	515.6	37.3%	618.6	32.4%
	Register File	4	0.6	2%	562.4	40.7%	674.8	35.4%
	ALU, Shifter & Mult.	4	4.7	29%	103.2	7.5%	297.6	15.6%
	Other Logic	4	1.6	11%	183.2	13.3%	202.3	10.6%
System	Controller	1	0.6	3%	4.8	0.4%	9.6	0.5%
	Main Mem (64KB)	1	2.9	18%	10.0	0.7%	80.0	4.2%
	Inter-processor Bus	1	0.2	1%	2.5	0.2%	25.0	1.3%
	DMA	1	0.1	1%	0.1	0.0%	1.0	0.1%
	Routing	1	1.9	12%	0.0	0.0%	0.0	0.0%
Total	180nm (1.8V @400MHZ)		16.1	100%	**1381.7**	100%	**1909.0**	100%
	90nm (1V @400MHZ)		3.8		**268.1**		**370.4**	

Table 1: System area and power summary for W-CDMA and 802.11a

local memory which consumes 16% for W-CDMA and 14% for 802.11a; and 3) the scalar pipleine which consists of the scalar memory, instruction queue and miscellaneous logic which consumes 10%, 9%, and 10%, respectively, for both W-CDMA and 802.11a.

This table also shows the area break-down. Unlike traditional processor architectures, the biggest component is the arithmetic units, not memory units. This indicates that this processor architecture has a very high computation efficiency. By having small local memories, we are able to reduce the power consumption as well as decrease the die area. The global memory is shared between processors. Its size is required to store enough data for buffering frames during interleaving processing. Unless a more efficient interleaver algorithm is found, this global memory space is unavoidable.

4 Software Development Flow

4.1 Overall Design Flow

Figure 4 shows our software design flow. Algorithms are first debugged and verified functionally through either Matlab/Simulink or floating-point C implementations. In a manner similar to traditional SoC design, the development flow then separates into system-level and kernel-level design. They are both implemented in fixed-point format in SPEX (Signal Processing EXtensions for C). SPEX is a Matlab-like programming extension which offers first-class vector and matrix variables and operations. Unlike Matlab, SPEX also allows explicit variables declarations, including variable bitwidth and saturation mode operations. SPEX is explained in further detail in next section. From this point on, the compiler can automatically generate assembly code, but programmers can also choose to handcode the DSP kernel assembly files for further optimizations.

Machine code is generated in three steps from SPEX descriptions. First machine and timing independent assembly code is generated. At this level, kernels are not assigned to processors, and system protocol descriptions are not incorporated with kernel assembly code. Second machine-dependent assembly code is generated, where kernels are mapped into processors, and system protocol descriptions are translated into real DMA and control instructions. Finally, real machine code is generated by merging the system-level and kernel-level assembly code. Programmers are given the flexibility to develop and debug code during any stage of the compilation. The efficiency of SPEX means our compiler does not need complex code-transformation techniques, making the assembly code easily accessible to DSP developers.

4.2 High-Level Programming Model

We proposed a multi-core, wide SIMD processor architecture. Given the difficulty in programming traditional DSPs, this new processor architecture provides even greater challenges for the programmers and compilers. In this section, we briefly describe our C language extension called SPEX (Signal Processing EXtension), which is aimed at narrowing the semantic gap between the description of high-end signal processing algorithms and their implementation. SPEX contains two major components: SIMD variables and concurrent kernel support. The former is suitable for expressing data parallelism within algorithms by providing SIMD data structures and explicit SIMD operations. The latter

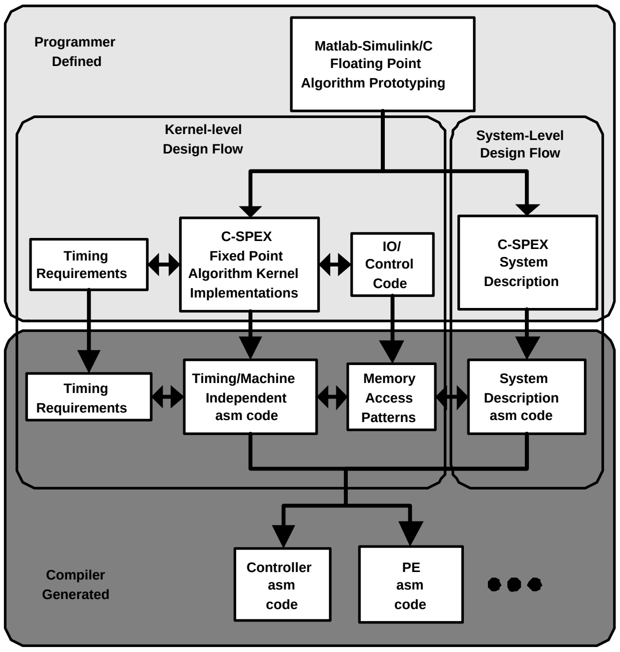

Programmer Defined
Matlab-Simulink/C Floating Point Algorithm Prototyping
Kernel-level Design Flow
System-Level Design Flow
Timing Requirements
C-SPEX Fixed Point Algorithm Kernel Implementations
IO/ Control Code
C-SPEX System Description
Timing Requirements
Timing/Machine Independent asm code
Memory Access Patterns
System Description asm code
Compiler Generated
Controller asm code
PE asm code

Figure 4: Software Development Flow

is suitable for expressing thread-level parallelism within algorithms through the use of concurrent kernel objects that communicate through channel objects. The intent is to separate coarse grain communication from fine grain communication. Coarse grain communication is best represented using the kernel extensions, and fine grain communication is best represented using the SIMD variable extension. We have not explicitly supported instruction-level parallelism in SPEX because modern parallelizing compilers are good at discovering it automatically.

Variable Extensions. SPEX contains a first class SIMD data type and an attribute mechanism for specifying implementation details. The SIMD extensions consist of two major data structures: one for describing scalars, another for describing SIMD data. The SIMD data structure is constructed internally as an array of scalar variables, which supports both vector and matrix objects. Both data structures can be further elaborated through attributes, which allow the programmer to specify implementation details at the point of variable declaration. These variable attributes are then treated internally as compiler directives, which are interpreted by the compiler based on the specifics of the target DSP architecture.

Kernel Extensions. SPEX kernel extensions consist of two types of data objects: kernel objects and channel objects. Conceptually, kernel objects represent functions that can be executed concurrently, and channel objects are communication interfaces between kernels. With traditional C, DSP programmers have to manually implement these communication memory structures. This results in difficult-to-read code for humans and compilers alike. In SPEX, kernels are written independently and communicate through virtual channel objects. This separation removes memory management tasks from the programmers.

5 Conclusion

In this paper, we have presented a hardware and software solution for SDR. The hardware system is composed of a set of dual-issue asymmetric processing elements that each contain a scalar and wide SIMD pipeline. A 4 processor version of this system is shown to meet the performance requirements of W-CDMA and 802.11a physical layer processing, and have the power characteristics needed for mobile terminals. To support software development on this system, we provide a modular programming environment that includes separate system and kernel level specification and debugging. Programming is carried out using SPEX, a set of extensions to C for specifying vector/matrix objects and operators, along with virtualized inter-kernel communication. Our future work includes the implementation of a larger variety of protocols and a deeper exploration of efficiency trade-offs of programmable signal processing architectures.

6 Acknowledgement

Yuan Lin is supported by a Motorola University Partnership in Research Grant. This research is also supported by ARM Ltd., the National Science Foundation grant CCR-0325898, and equipment donated by Intel Corporation.

References

[1] Dsp developers' village, Texas Instruments, http://dspvillage.ti.com.

[2] Freescale starcore, http://www.freescale.com.

[3] Morpho technologies: http://www.morphotech.com/.

[4] T. Austin, D. Blaauw, S. Mahlke, T. Mudge, C. Chakrabati, and W. Wolf. Mobile supercomputers. *Communications of the ACM*, May 2004.

[5] R. Baines and D. Pulley. The picoArray and reconfigurable baseband processing for wireless basestations. In *Software Defined Radio*, 2004.

[6] J. Glossner, E. Hokenek, and M. Moudgill. The Sandbridge Sandblaster Communications Processor. 2004.

[7] B. Mohebbi and F. Kurdahi. Reconfigurable parallel DSP - rDSP. In *Software Defined Radio*, 2004.

A System Solution for High-Performance, Low Power SDR

Yuan Lin[1], Hyunseok Lee[1], Yoav Harel[1], Mark Woh[1], Scott Mahlke[1], Trevor Mudge[1] and Krisztian Flautner[2]

[1]Advanced Computer Architecture Laboratory, University of Michigan

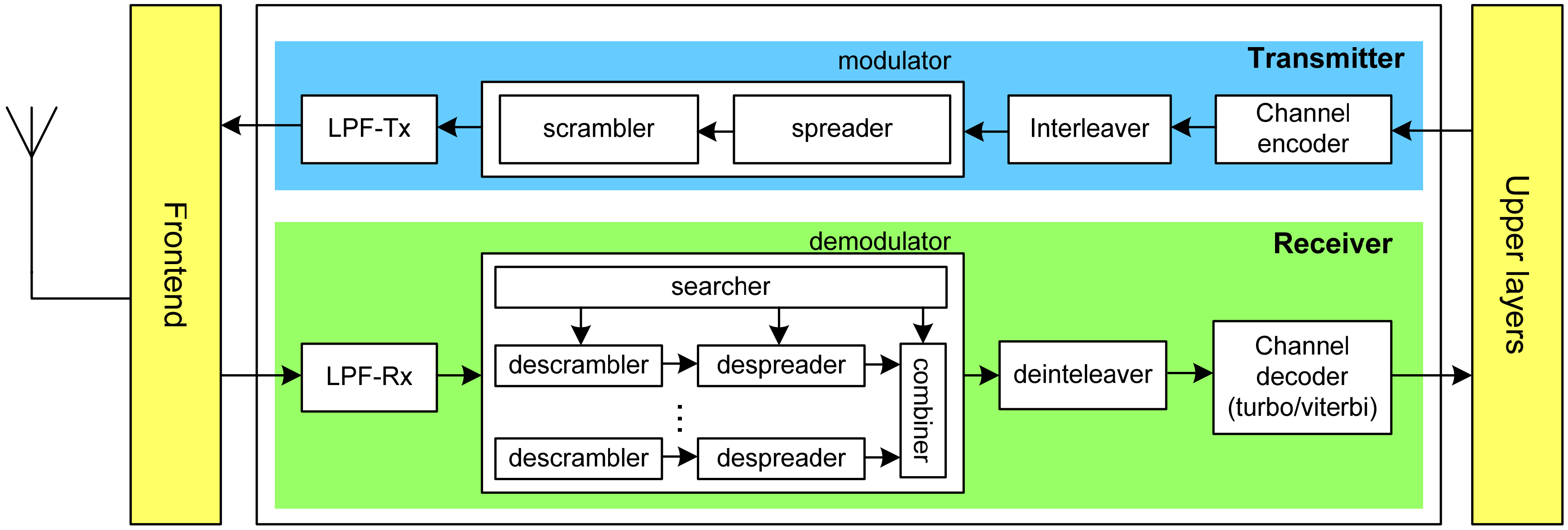
[2]ARM, Ltd.

SDR Design Challenges:

- Hardware Design Challenges
  - High computational throughput
  - Low power consumption
  - Meet real-time requirements
- DSP Programming Support
  - System-level development
    - Inter-algorithm communication
  - Algorithm-level development
    - Efficient DSP representations

SDR Benchmark Design & Analysis

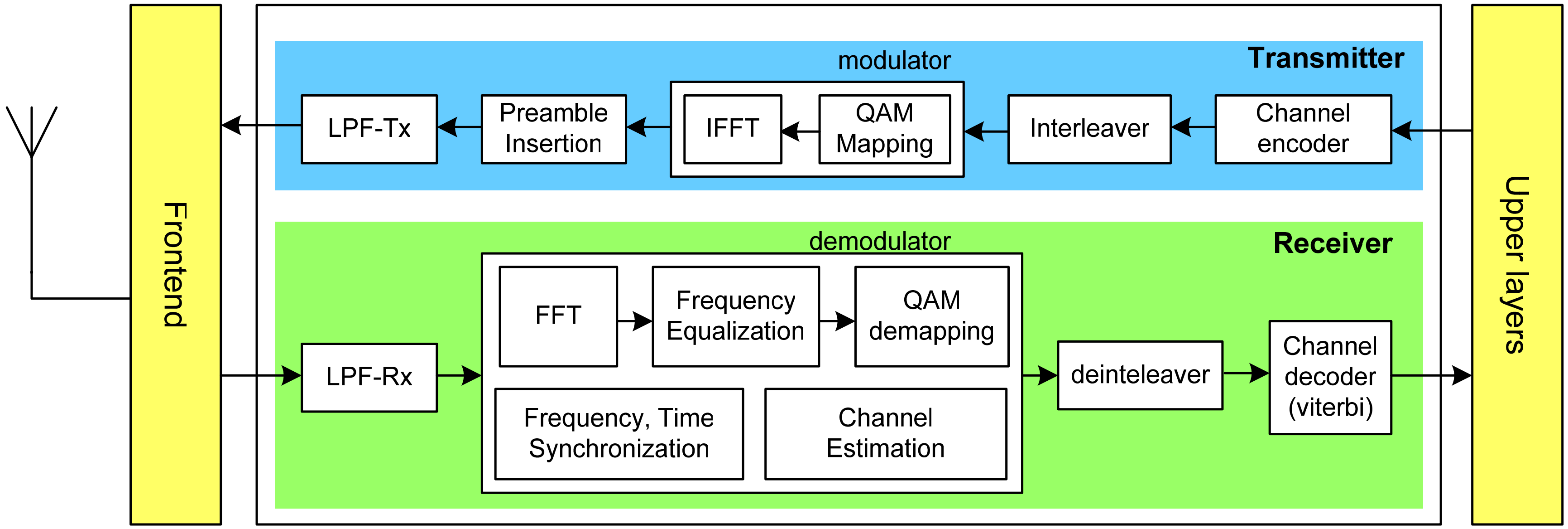
WCDMA Protocol



W-CDMA characteristics

- Mixture of various algorithms having different workload and timing requirement
- Plenty of vector parallelism
- 8~16 bits suffice
- Multiplication is not dominant
- No floating point operation
- Small instruction/data memory

802.11a Protocol

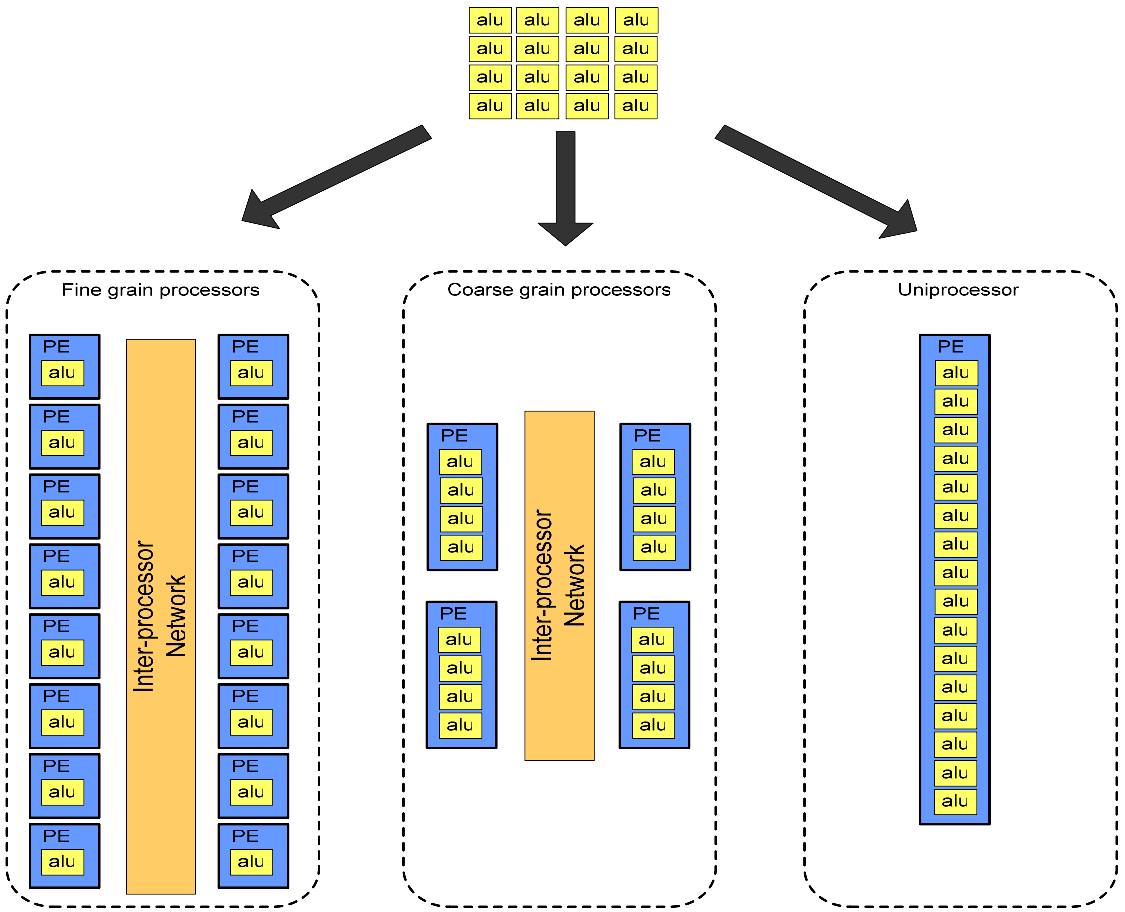


OFDM characteristics

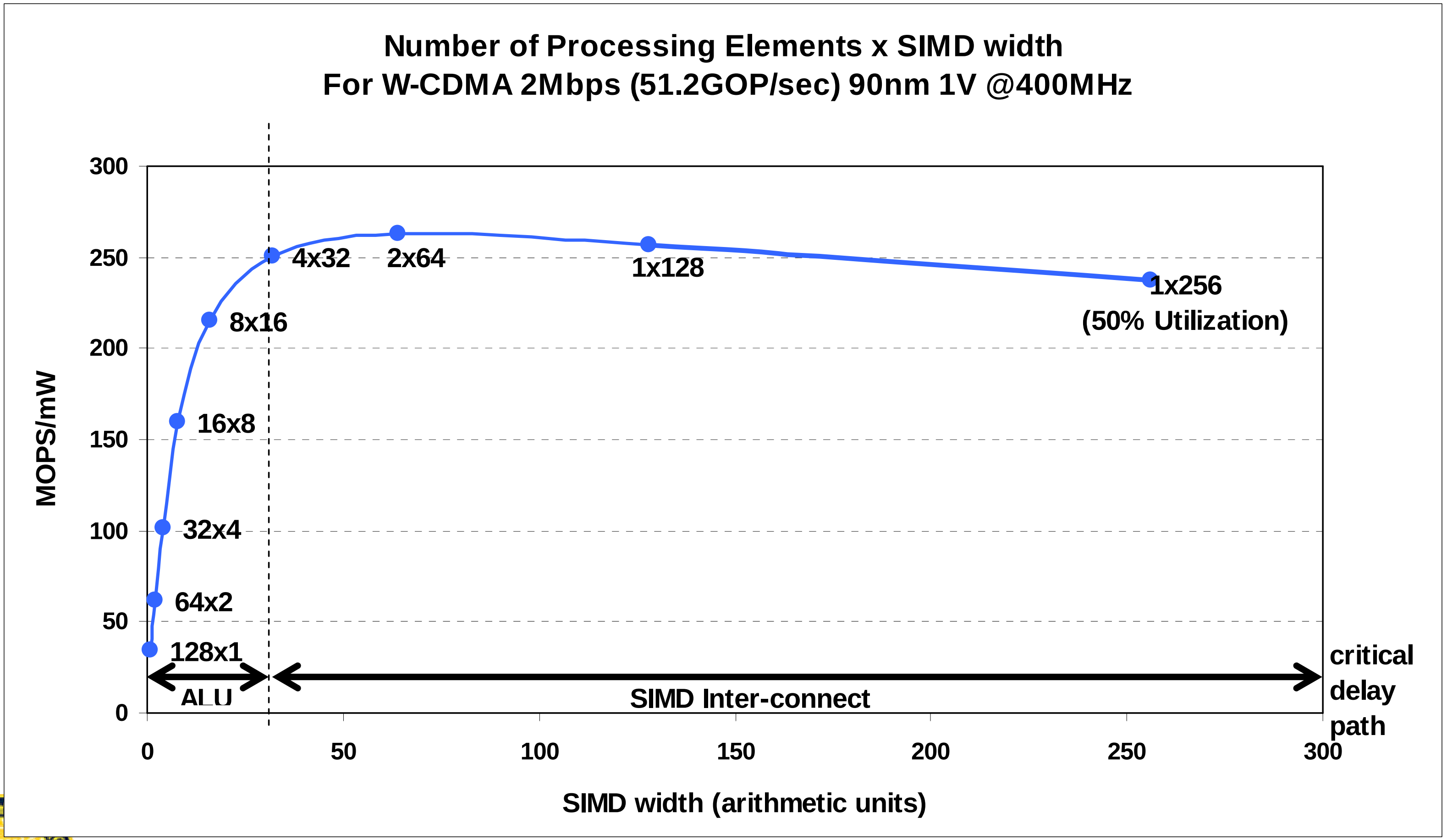
- Viterbi decoder, Filters are dominants
- Vector parallelism of Viterbi decoder is crucial for performance enhancement
- No periodic hard realtime tasks
- 8bit (Viterbi), 16bit(Demodulation, Synch, LPF)

SDR Processor Architecture Design

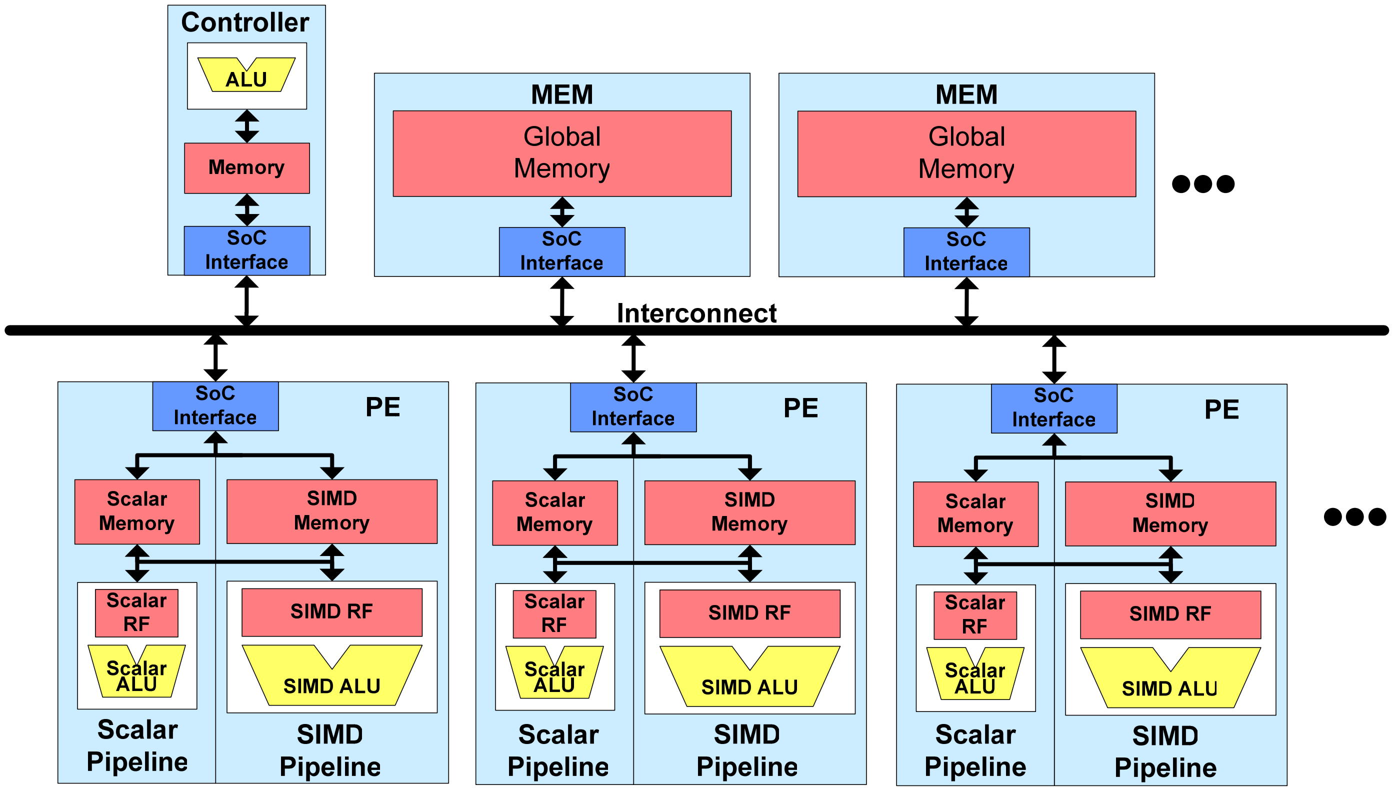
System Architecture Design Tradeoffs



System Architecture Design Tradeoffs



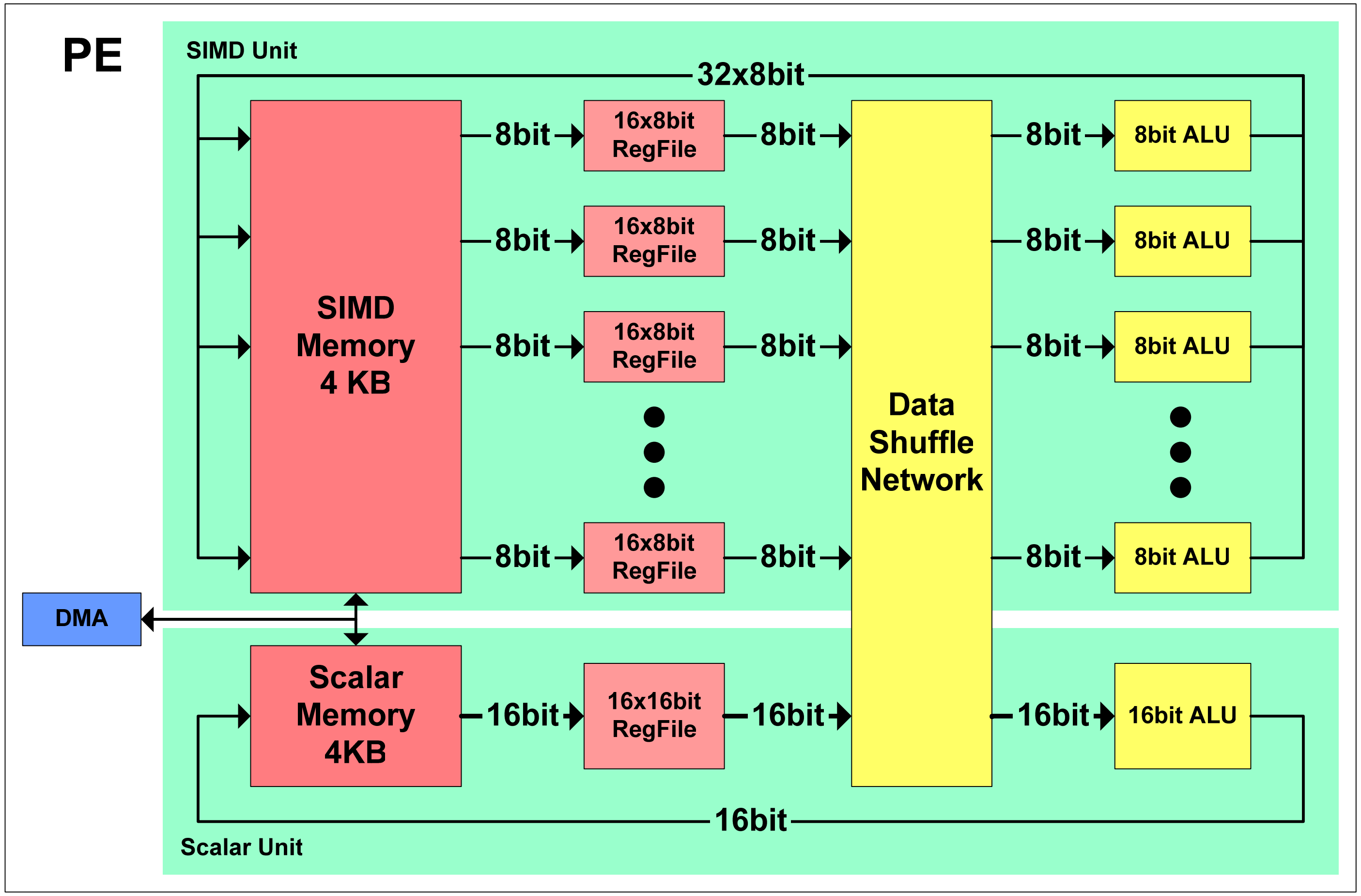
System Architecture Design



Our SDR System Architecture Design

- Scalable system design
  - Standardized SoC interface
  - System interface supports multiple (potentially) heterogeneous PEs and memories
- For WCDMA & 802.11a
  - 4 homogeneous processing elements (PEs)
    - Dual pipelines: scalar pipeline & SIMD pipeline
    - Local scratchpad memory (no data cache)
  - Global scratchpad memory (64KB)
  - Controller -- ARM general purpose processor

PE Design

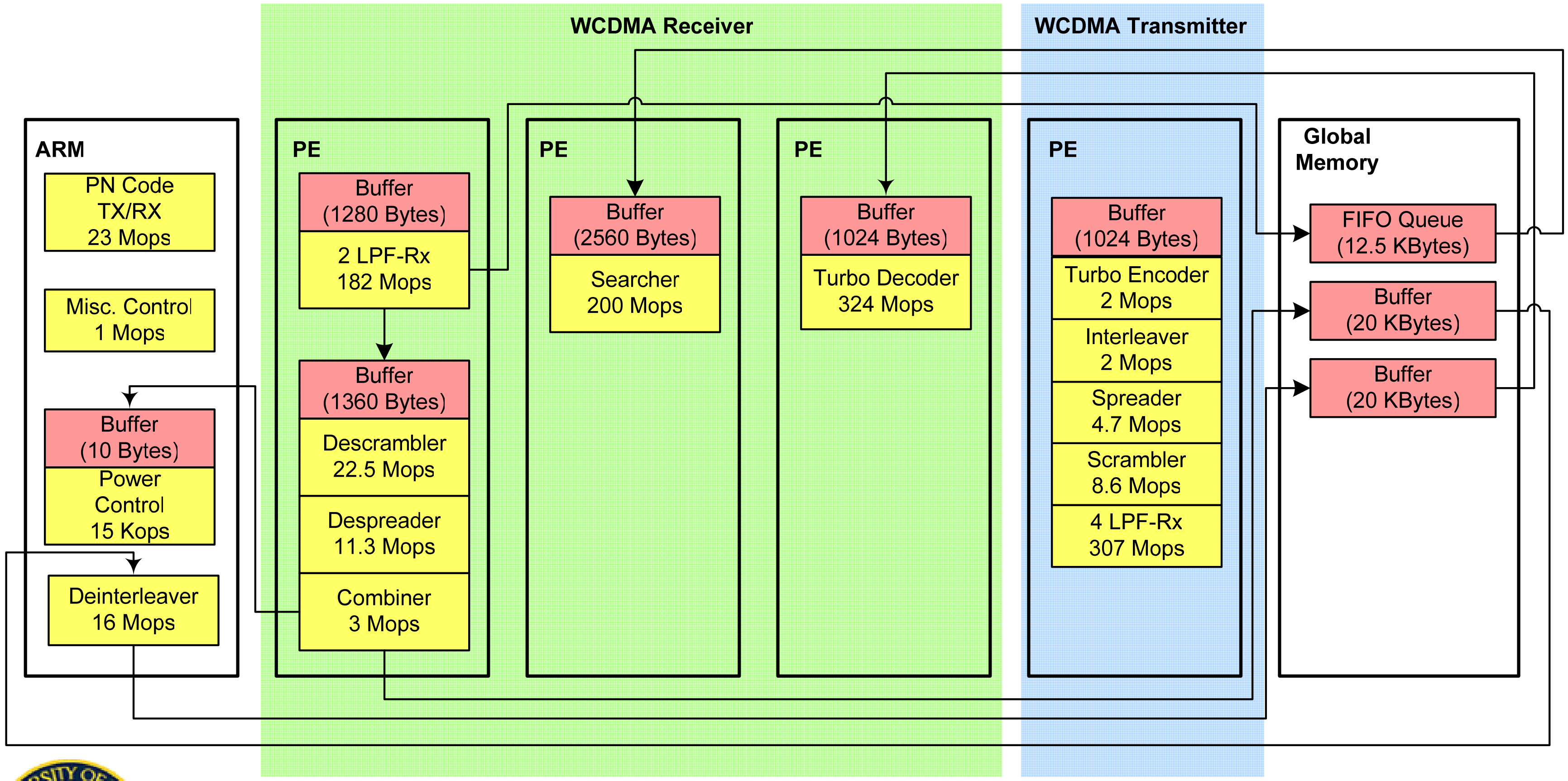


PE
SIMD Unit
32x8bit
SIMD Memory 4 KB
8bit
16x8bit RegFile
8bit
Data Shuffle Network
8bit
8bit ALU
DMA
Scalar Memory 4KB
16bit
16x16bit RegFile
16bit
16bit
16bit ALU
16bit
Scalar Unit

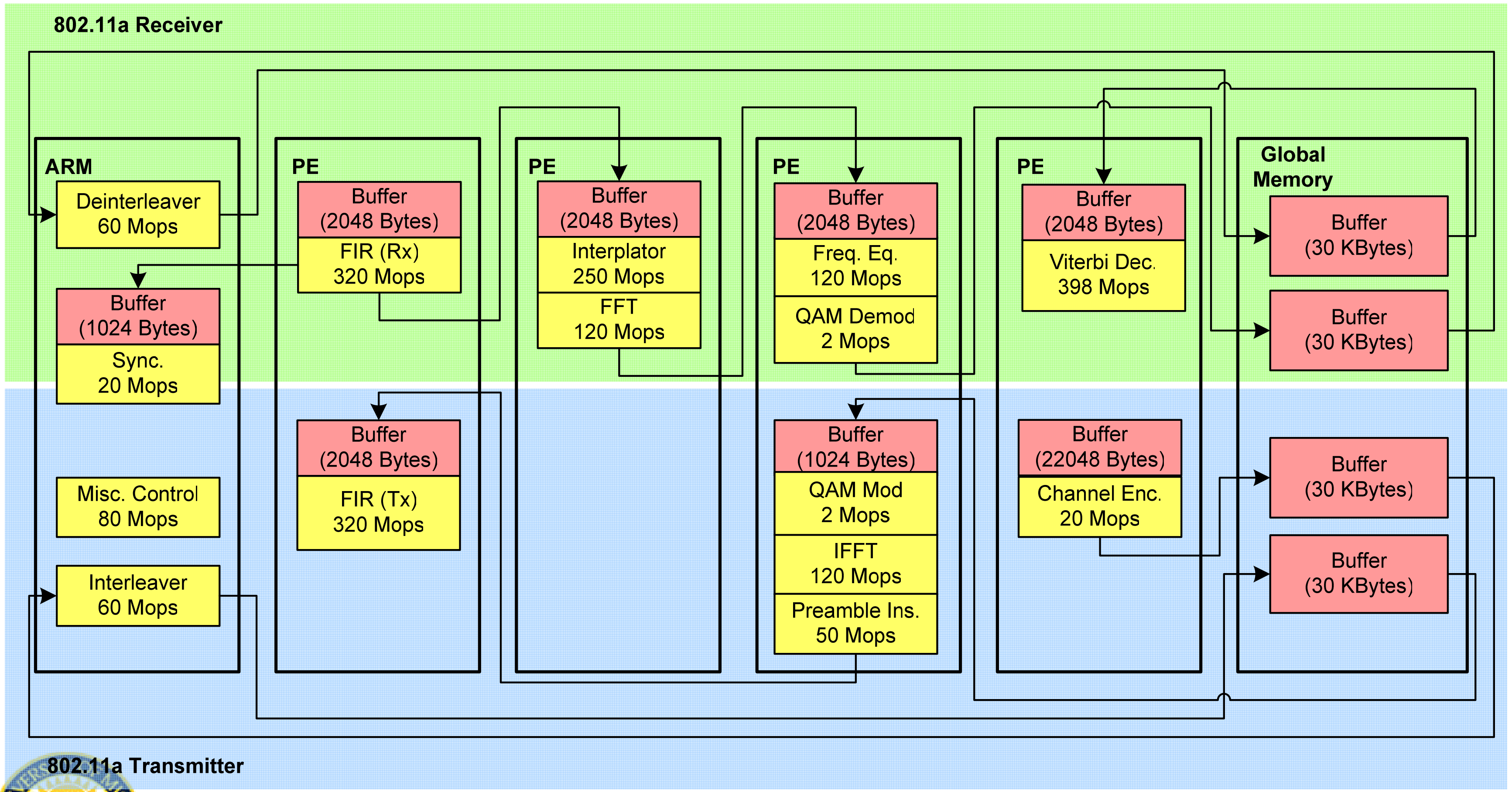
Processing Element (PE) Design

- Scalar pipelines
  - 16bit data path
- SIMD pipeline
  - 8 bit data path
  - 32x8 SIMD ALU
- Software controlled local scratchpad memory
  - 4KB scalar memory
  - 4KB SIMD memory
- Inter-PE communication through DMA

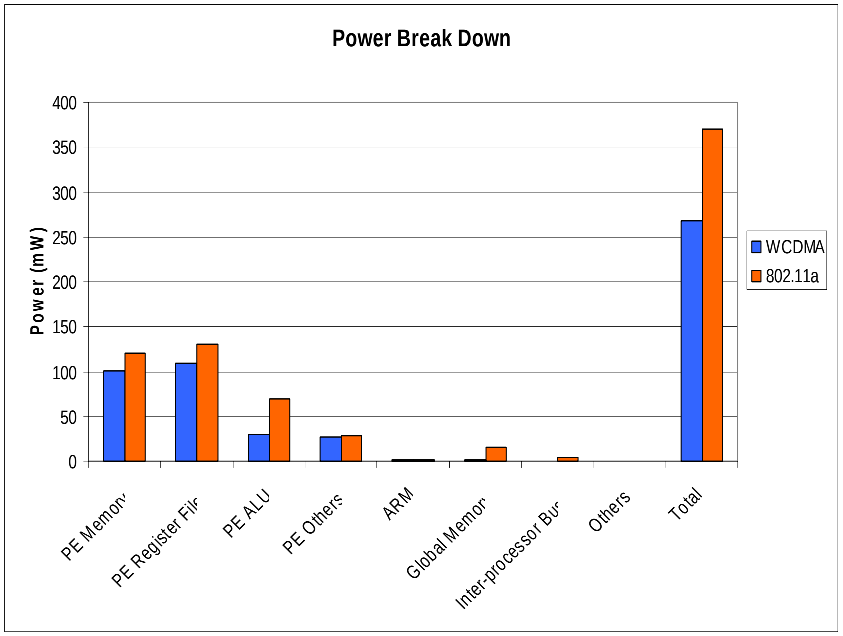
WCDMA PE Mapping

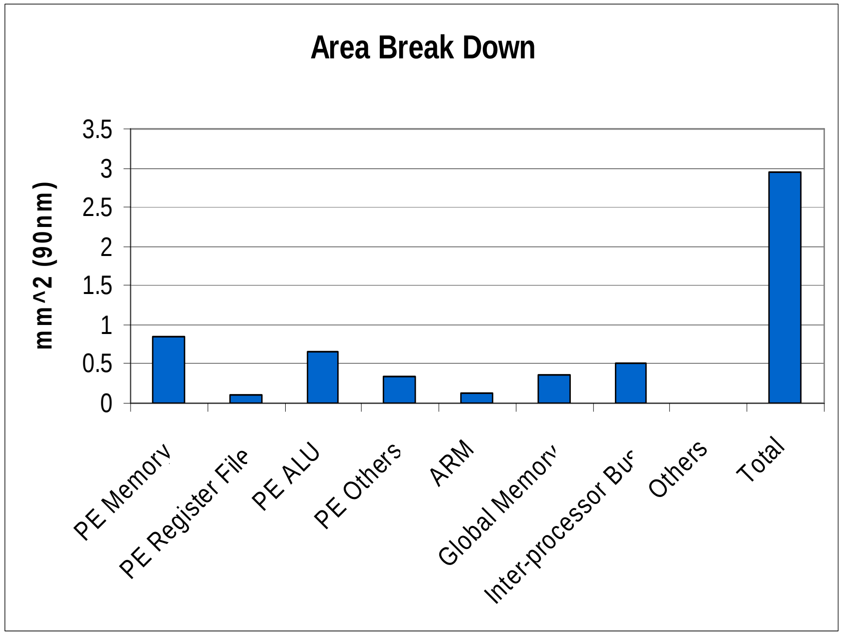


802.11a PE Mapping



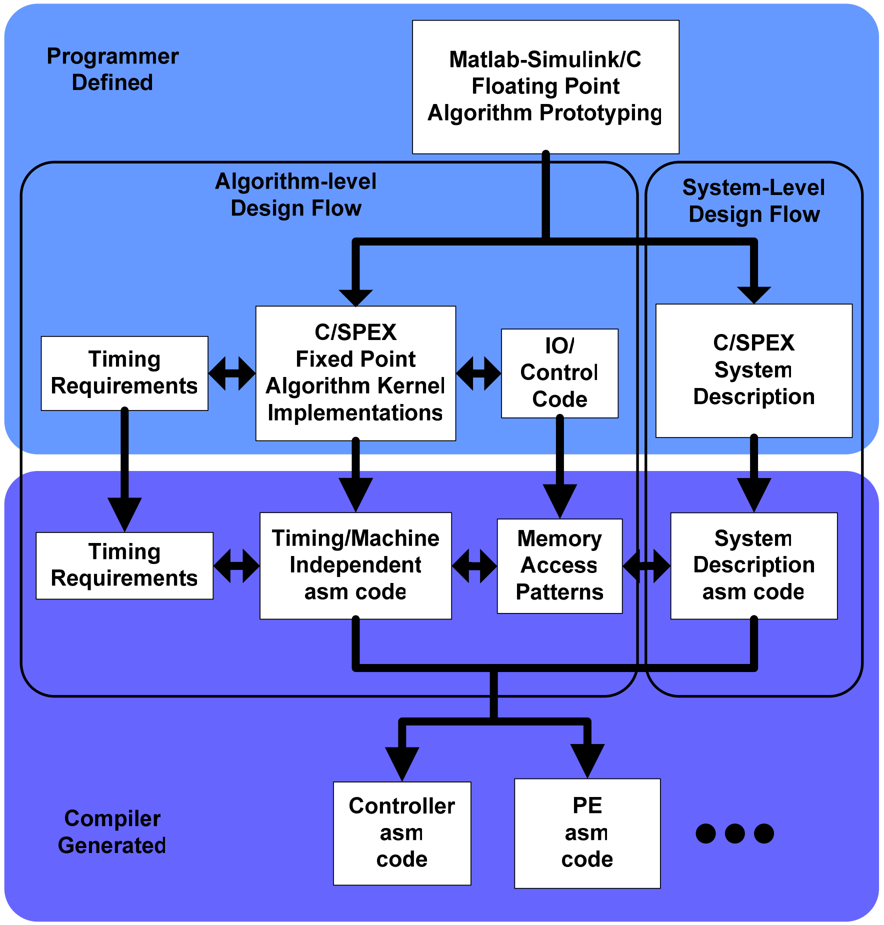
Power and Area Results





SDR Programming Language Support

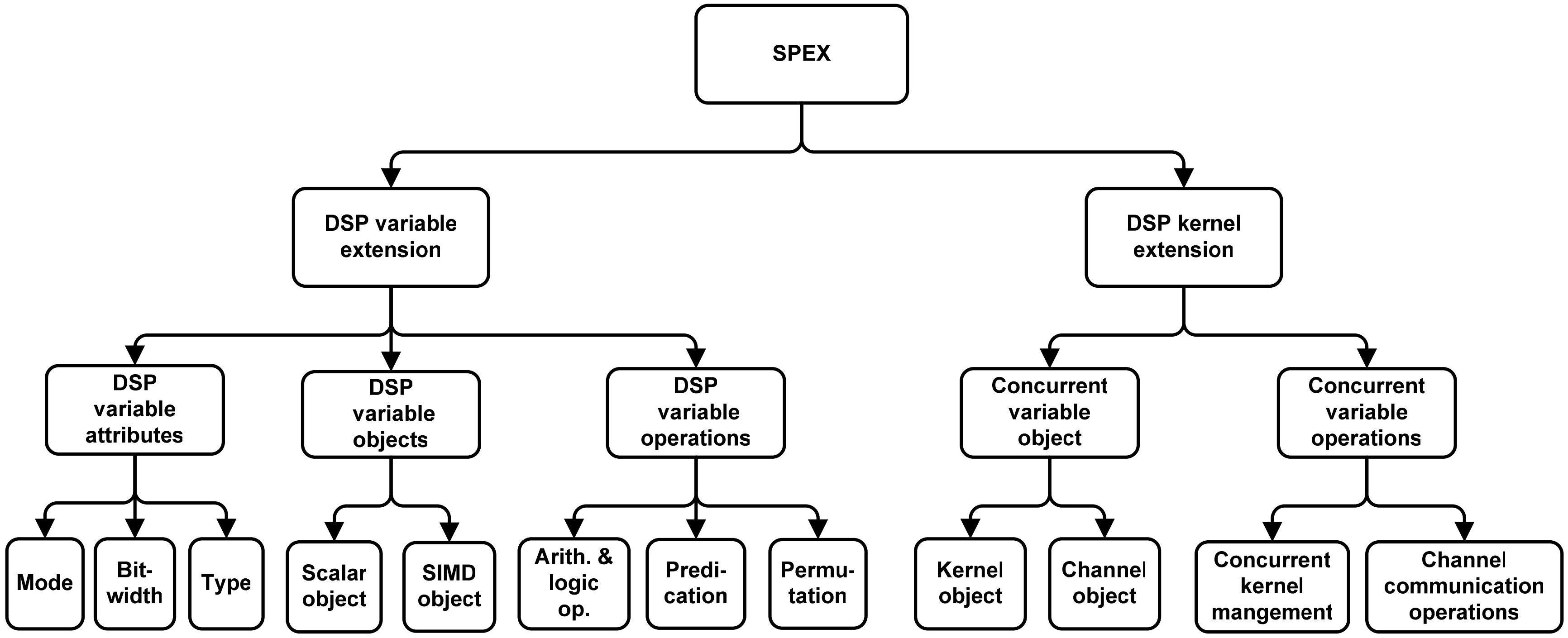
Software Development Flow


Programmer Defined
Matlab-Simulink/C Floating Point Algorithm Prototyping
Algorithm-level Design Flow
System-Level Design Flow
Timing Requirements
C/SPEX Fixed Point Algorithm Kernel Implementations
IO/ Control Code
C/SPEX System Description
Timing Requirements
Timing/Machine Independent asm code
Memory Access Patterns
System Description asm code
Compiler Generated
Controller asm code
PE asm code
●●●

SPEX (Signal Processing EXtension)

- Implemented as a library extension to C
- System-level development
  - Support concurrent DSP kernel function definitions
  - Channel variables for inter-kernel communications
- Algorithm-level development
  - Native vector & matrix variables
  - Explicit DSP variable attribute definition
  - Native vector & matrix operations

SPEX Overview



SPEX Example Code: Viterbi ACS

```
void* acs(void*) {
  /* variable declaration */
  saturated char<64> metrics1, metrics2;
  saturated char<64> states;
  saturated char<64> t1, t2;

  while (!viterbi.stop()) {
    /* receiving data from BMC */
    metrics1 = bmc_to_acs.receive();
    metrics2 = bmc_to_acs.receive();

    /* add */
    metrics1 += states;
    metrics2 += states;

    /* compare and select */
    t1 = (metrics1(0,2,62),metrics2(0,2,62));
    t2 = (metrics1(1,2,63),metrics2(1,2,63));
    states(t1<t2)  = t1;
    states(t1>=t2) = t2;

    /* sending data to TB */
    acs_to_tb.send(states);
  }
}
```

SPEX Example Code: Viterbi ACS

Concurrent DSP kernel definitions →

```
void* acs(void*) {
  /* variable declaration */
  saturated char<64> metrics1, metrics2;
  saturated char<64> states;
  saturated char<64> t1, t2;

  while (!viterbi.stop()) {
    /* receiving data from BMC */
    metrics1 = bmc_to_acs.receive();
    metrics2 = bmc_to_acs.receive();

    /* add */
    metrics1 += states;
    metrics2 += states;

    /* compare and select */
    t1 = (metrics1(0,2,62),metrics2(0,2,62));
    t2 = (metrics1(1,2,63),metrics2(1,2,63));
    states(t1<t2)  = t1;
    states(t1>=t2) = t2;

    /* sending data to TB */
    acs_to_tb.send(states);
  }
}
```

SPEX Example Code: Viterbi ACS

Native SIMD variable definition with explicit attributes

SPEX variable supports
1. saturated/overflow mode
2. various variable bit-width
i.e. *int2* is a 2 bit integer
3. vector & matrices

```
void* acs(void*) {
  /* variable declaration */
  saturated char<64> metrics1, metrics2;
  saturated char<64> states;
  saturated char<64> t1, t2;

  while (!viterbi.stop()) {
    /* receiving data from BMC */
    metrics1 = bmc_to_acs.receive();
    metrics2 = bmc_to_acs.receive();

    /* add */
    metrics1 += states;
    metrics2 += states;

    /* compare and select */
    t1 = (metrics1(0,2,62),metrics2(0,2,62));
    t2 = (metrics1(1,2,63),metrics2(1,2,63));
    states(t1<t2)  = t1;
    states(t1>=t2) = t2;

    /* sending data to TB */
    acs_to_tb.send(states);
  }
}
```

SPEX Example Code: Viterbi ACS

Inter-kernel communication through channel operations

Channel types:
1. FIFO queue
2. Broadcast queue
3. Merge queue
4. Sync/control channel
5. Random-read FIFO queue

```
void* acs(void*) {
  /* variable declaration */
  saturated char<64> metrics1, metrics2;
  saturated char<64> states;
  saturated char<64> t1, t2;

  while (!viterbi.stop()) {
    /* receiving data from BMC */
    metrics1 = bmc_to_acs.receive();
    metrics2 = bmc_to_acs.receive();

    /* add */
    metrics1 += states;
    metrics2 += states;

    /* compare and select */
    t1 = (metrics1(0,2,62),metrics2(0,2,62));
    t2 = (metrics1(1,2,63),metrics2(1,2,63));
    states(t1<t2)  = t1;
    states(t1>=t2) = t2;

    /* sending data to TB */
    acs_to_tb.send(states);
  }
}
```

SPEX Example Code: Viterbi ACS

SPEX vector operations supports

1. **SIMD arithmetic operations**
2. **SIMD permutation**
3. **SIMD predication**

```
void* acs(void*) {
  /* variable declaration */
  saturated char<64> metrics1, metrics2;
  saturated char<64> states;
  saturated char<64> t1, t2;

  while (!viterbi.stop()) {
    /* receiving data from BMC */
    metrics1 = bmc_to_acs.receive();
    metrics2 = bmc_to_acs.receive();

    /* add */
    metrics1 += states;
    metrics2 += states;

    /* compare and select */
    t1 = (metrics1(0,2,62),metrics2(0,2,62));
    t2 = (metrics1(1,2,63),metrics2(1,2,63));
    states(t1<t2)  = t1;
    states(t1>=t2) = t2;

    /* sending data to TB */
    acs_to_tb.send(states);
  }
}
```

SPEX Example Code: Viterbi ACS

SPEX allows efficient DSP algorithm implementation

C code:

```
/* compare and select */
for (i = 0; i < 32; i++) {
  if (metrics1[i*2] < metrics1[i*2+1])
    states[i] = metrics1[i*2];
  else
    states[i] = metrics1[i*2+1];

  if (metrics2[i*2] < metrics2[i*2+1])
    states[i+32] = metrics2[i*2];
  else
    states[i+32] = metrics2[i*2+1];
}
```

$\longleftrightarrow$

```
void* acs(void*) {
  /* variable declaration */
  saturated char<64> metrics1, metrics2;
  saturated char<64> states;
  saturated char<64> t1, t2;

  while (!viterbi.stop()) {
    /* receiving data from BMC */
    metrics1 = bmc_to_acs.receive();
    metrics2 = bmc_to_acs.receive();

    /* add */
    metrics1 += states;
    metrics2 += states;

    /* compare and select */
    t1 = (metrics1(0,2,62),metrics2(0,2,62));
    t2 = (metrics1(1,2,63),metrics2(1,2,63));
    states(t1<t2)  = t1;
    states(t1>=t2) = t2;

    /* sending data to TB */
    acs_to_tb.send(states);
  }
}
```

Summary

- Hardware & software solutions for SDR
  - Hardware
    - 4 dual-issue asymmetric SIMD processing elements
    - Consumes 200~300mW for 90nm
    - Meets the performance requirements for WCDMA & 802.11a
  - Software
    - SPEX provides efficient DSP algorithm and system implementation