
XPP Technologies

# XPP-III Processor Overview

White Paper

*For further information and questions, please contact* `support@pactxpp.com.`

Version 2.0.1 July 13, 2006

# 1 Introduction

The limitations of conventional processors are becoming more and more evident. The growing importance of stream-based applications makes reconfigurable architectures an attractive alternative. They combine the performance of ASICs with the flexibility of programmable processors. On the other hand, irregular control-flow dominated algorithms require high-performance sequential processor kernels for embedded applications. The XPP-III architecture combines novel sequential processor kernels optimized for these algorithms with a coarse-grain reconfigurable dataflow array. It is designed to support different types of parallelism: pipelining, instruction level, dataflow, and task level parallelism. Hence XPP-III meets the performance requirements of today's heterogeneous embedded applications. It is well suited for applications in multimedia, telecommunications, simulation, digital signal processing, cryptography and similar application domains.

This White Paper first gives an overview of the runtime-reconfigurable *XPP-III Processors*. It focuses on the *XPP-III Core* architecture, cf. section 2. It consists of sequential processor kernels called *Function-PAEs* (PAE = Processing Array Elements), cf. section 2.2, and a reconfigurable *XPP dataflow array* (ALU- and RAM-PAEs, cf. sections 2.3 and 2.4). The *XPP-III Reference Design* (cf. section 2.6) provides a full processor implementation consisting of the XPP-III Core and peripheral devices. Next, the XPP processing basics are presented in section 3.

More information on other aspects of the XPP architecture can be found in the accompanying White Papers *Programming XPP-III Processors* and *Reconfiguration on XPP-III Processors*.

# 2 Architecture

XPP (*eXtreme Processing Platform*) is a data processing architecture based on a hierarchical array of coarse-grain, adaptive computing elements called *Processing Array Elements (PAEs)*, cf. sections 2.1 – 2.4, and a *packet-oriented communication network*, cf. section 2.5.

Control-flow dominated, irregular code (without loop-level or pipelining parallelism) is mapped to one or several concurrently executing Function-PAEs (FNC-PAEs). They are sequential 16-bit processor kernels which are optimized for sequential algorithms requiring a large amount of conditions and branches like bit-stream decoding or encryption. A FNC-PAE executes up to eight ALU operations and one special operation (e. g. multiplication) in one cycle. Operations on up to four levels can be *chained*, i. e. the output of one operation is immediately fed to the input of the next operation in the chain. This can even be combined with *predicated execution*, i. e. conditional execution based on the result of input operations. In this way, nested if-then-else statements can be executed in one cycle. Furthermore, special mechanisms enable jumps (conditional

or non-conditional) in one cycle. The FNC-PAE offers high perfomance at a moderate clock frequency.

Regular streaming algorithms like filters or transforms are efficiently implemented on the dataflow part of the XPP-III array (ALU- and RAM-PAEs, see below). Flow graphs of arbitrary shape can be directly mapped to ALUs and routing connections, resulting in a parallel, pipelined implementation. Distributed *event* signals within the dataflow array add additional flexibility for less regular algorithms since events can be used to control the data streams. However, the real strength of the XPP dataflow array originates from the combination of parallel array processing with unique, powerful run-time reconfiguration mechanisms. PAEs can be configured while neighboring PAEs are processing data. Entire algorithms can be configured and run independently on different parts of the array. Reconfiguration is triggered by a controlling FNC-PAE or by special event signals originating within the dataflow array. By utilizing protocols implemented in hardware, data and event packets are used to process, generate, decompose and merge streams of data.

## 2.1 XPP-III Core Structure

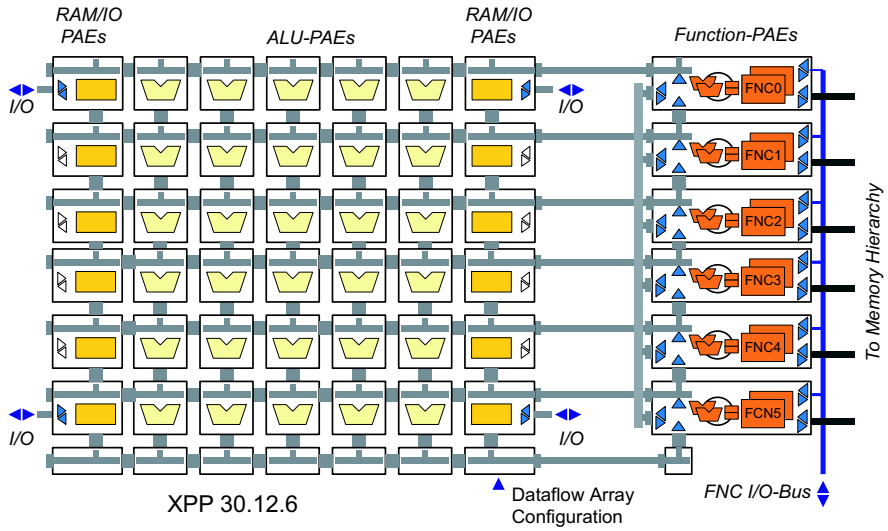


Figure 1: Structure of a sample XPP-III Core

A XPP Core contains a rectangular array of three types of PAEs: Those in the center of the array are *ALU-PAEs*. To the left and right side of the ALU-PAEs are *RAM-PAEs with I/O*. Finally, at the right side of the array, there is a column of *FNC-PAEs*. An ALU-PAE contains three ALUs. A RAM-PAE contains two ALUs, a small RAM, and an I/O object. A FNC-PAE contains a complete VLIW-like sequential processor kernel. The details of these PAE types are described in the following sections. Figure 1 shows a sample array with 30 ALU-PAEs, 12 RAM-PAEs and six FNC-PAEs.

The horizontal routing busses for point-to-point connections between XPP objects (ALUs, RAMs, I/O objects, etc.) are also integrated in the PAEs. They can be segmented by configurable *switch objects*. Seperate busses for n-bit data values and 1-bit events are available. The XPP Core data bitwidth (for ALUs, RAMs, and data busses) can be chosen from 16, 24 or 32 bit. Vertical routing connections are provided within the ALU- and RAM-PAEs. Between the FNC-PAEs, there is an additional dedicated vertical routing connection.

After resetting a XPP-III Processor or at power-up, the first FNC-PAE (FNC0) boots automatically from external memory. It then boots the other FNC-PAEs (if applicable). When the dataflow array is used, a FNC-PAE instructs a DMA controller to configure the ALU- and RAM-PAEs as well as the communication network from external memory. The configuration words enter the dataflow array through its *configuration interface* which is internally connected to a pipelined configuration bus. Each configuration word contains the address of the PAE and XPP object to be configured and the configuration value (ALU operator, bus connection etc.).

The I/O objects allow to cascade XPP Cores and to access external streaming data sources or destinations or external RAM. The XPP Core's data and event synchronization mechanism is extended to the I/O ports by means of handshake signals.

## 2.2 FNC-PAE

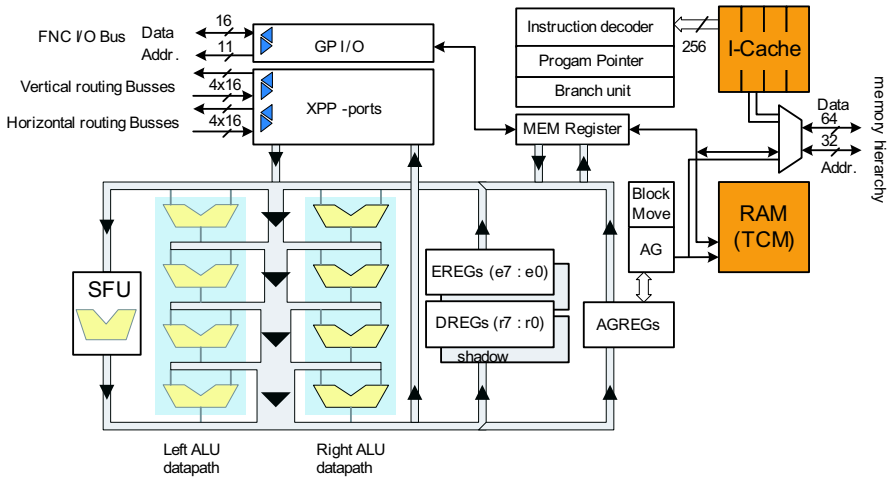


Figure 2: Structure of a FNC-PAE

Figure 2 shows the structure of a FNC-PAE. It comprises a 2x4 array of 16-bit ALUs, a Special Function Unit (SFU), a 16-bit register file, a 32-bit address generator (AG), a local instruction cache, a tightly coupled memory (TCM) and I/O ports which connect it tightly to the other FNC-PAEs, to the dataflow array and to a general purpose I/O bus.

The eight ALUs are designed to be small and fast because they are arranged in two non-pipelined (i. e. purely combinational) columns of four ALUs each. Therefore, the ALUs are restricted to a limited instruction set containing arithmetic, logic, comparison and barrel shift operations including conditional execution and branching.

Every ALU selects its operands from the register files (data registers DREG and extended registers EREG, both with shadow registers), from the address generator registers (AGREG), the memory register MEM or the ALUs of all rows above itsefl. Furthermore, the top-row ALUs have access to the I/O ports. All ALUs can store their results simultaneously to the registers.

The ALU datapath is not pipelined since the FNC-PAE is optimized for irregular code with many conditions and jumps. These code characteristics would continuously stall the operator pipeline, resulting in a low IPC (instructions per cycle) count. Instead, the FNC-PAE chains the ALUs and executes all instructions asynchronously in one cycle, even if there are dependences. Together with unique features[1] which enhance the condition execution and branching performance, this results in a very high IPC count. Despite the lower clock frequency, irregular algorithms are executed very efficiently.

The FNC-PAE also supports efficient procedure call and return, stack operations and branching. Up to three independent jump targets can be evaluated in a single cycle. The Special Function Unit (SFU) operates in parallel to the ALU datapath. It supports up to two 16x16-bit multiplications and functions such as bit-field extraction. The SFU delivers its results directly to the register file. By combining the SFU multiplications with the adders of the ALU array, it is possible to execute two pipelined multiply-accumulate (MAC) operations each cycle.

For efficient code access, a local 256x256-bit 4-way set-associative L1 instruction cache (I-cache) is provided. Cache lines can be locked for high priority code. The data memory is composed of a tightly-coupled memory (TCM, 1Kx16-bit) and access mechanisms to external RAM. The memory is addressed by the address generator or a block move unit which transfers blocks of data between external memory and the TCM in the background. Data read from memory is available for the ALUs in the MEM register. Code and data accesses utilize a 64-bit data and 32-bit address bus to connect to the external memory hierarchy.

## 2.3 ALU-PAE

Figure 3 shows the structure of an ALU-PAE. It contains three XPP objects (FREG, ALU and BREG object) and the routing busses. All objects have input registers which store the data or event packtets for one cycle, i. e. add a register delay. After the input register, a one-stage FIFO stores an additional packet if required. This feature is especially useful for pipeline balancing. The input registers and FIFOs can be preloaded during configuration. Additionally, the input registers can be set to “constant mode”. In this mode, they continuously generate packets with the same constant value. In contrast to input registers, output registers (DF-Register in Figure 3) do not add a delay.

[1] Patents pending.

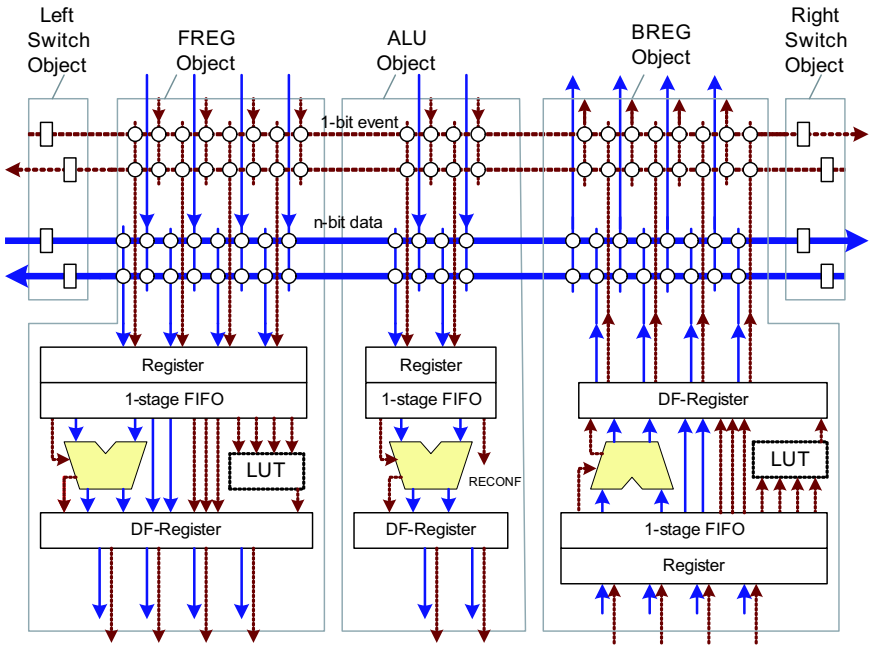


Figure 3: Structure of an ALU-PAE

They store the values only to guarantee that no packets are lost during pipeline stalls. The switch objects directly connect an input register, a one-stage FIFO, and an output register. Hence they also add a register delay and can store up to two additional packets (in a pipeline stall).

The *ALU object* in the center of the PAE provides the following functionality:

- logical operators
- basic arithmetic operators (i. e. adders and subtracters)
- special arithmetic operators including comparators and multipliers

An event input port is used to control the ALU execution and provide a carry-in value. An event output port is used to control subsequent computations and provide a carry-out value. The special `RECONF` event input triggers a reconfiguration. The ALU inputs can be connected to one or more busses above the ALU, and the outputs to one or more busses below it. Hence the processing direction is top-down.

The Forward Register (FREG) object on the left side and the Backward Register (BREG) object on the right side of the ALU-PAE are very similar. The main difference is the processing direction: top-down for the FREG and bottom-up for the BREG object. Both objects provide the following functionality:

- routing of data and events (top-down or bottom-up, respectively)

- dataflow operators (for stream merging, multiplexing, demultiplexing etc.)
- basic arithmetic operators (i. e. adders and subtracters)
- lookup table for boolean operations on events and event stream processing (LUT)

For both objects, the number of vertical routing connections is set by XPP Core parameters. If an ALU is not used for dataflow or arithmetic operations, the ports can be used as additional routing connections. The same holds for event ports not used by the LUT.

The following additional operators are only available in either the BREG or FREG object:

- **BREG:** barrel shifter, packing, unpacking and clipping operators
- **FREG:** counters and accumulators

The combination of the different ALU types in an ALU-PAE allows the efficient implementation of typical signal-processing functions (multiply-add or multiply-accumulate combinations) or fixed-point operations (multiply-add-shift combinations).

## 2.4 RAM-PAE

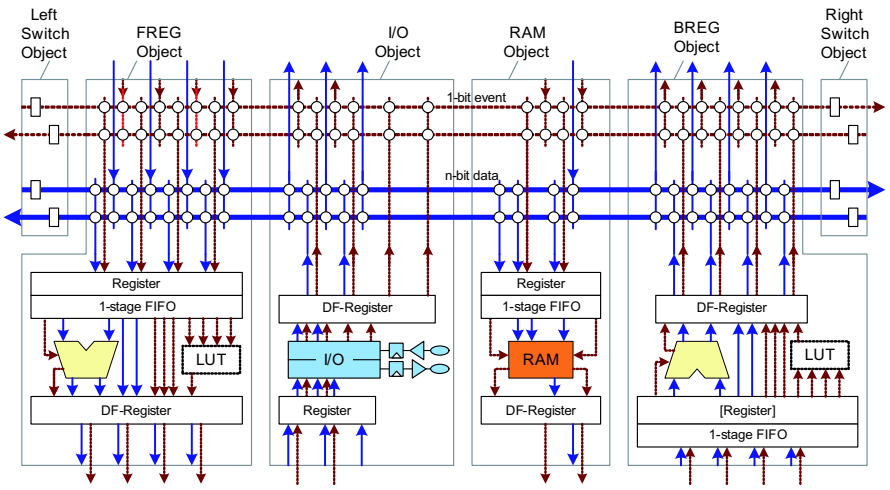


Figure 4: Structure of a RAM-PAE

Figure 4 shows a RAM-PAE. The FREG and BREG objects are identical to the ones in the ALU-PAEs, but the ALU object is replaced by a RAM object, and there is an additional I/O object. The RAM object contains a small bank of two-ported SRAM. The RAM size can be selected by a XPP Core parameter. The RAM words have the

same bitwidth as all other XPP objects and data busses. Two independent address ports enable simultaneous read and write operations. The third data input port is used for the write value, and the only data output port for the read value. The event ports control the RAM access. All ports use a ready-acknowledge protocol like the other XPP objects.

The RAM operates either in internal RAM (IRAM) or FIFO mode. In the latter mode, no explicit addressing is required. The FIFO uses internal read and write pointers and generates output packets as long as it contains data. RAMs and FIFOs can be preloaded during configuration. The content of RAMs is preserved during reconfiguration of the array.

The I/O object is integrated into the RAM-PAE, providing access to external data and event streaming sources or destinations or external RAMs (EXRAM mode).

## 2.5 Packet Handling and Synchronization

As explained above, XPP objects communicate through a packet-oriented network. In normal operation mode, XPP objects are self-synchronizing. An operation is performed as soon as all necessary data input packets are available. The results are forwarded as soon as they are available, provided the previous results have been consumed. Thus it is possible to map a dataflow graph directly to ALU objects, and to pipeline input data streams through it. The communication system is designed to transmit one packet per cycle. Hardware protocols ensure that no packets are lost, even in the case of pipeline stalls or during the configuration process. This simplifies application development considerably. No explicit scheduling of operations is required. The "constant mode" of input ports disables the synchronization. This allows to repeatedly reuse data values.

Event packets transmit state information. E. g., events generated by ALU objects depend on ALU results or exceptions, very similar to the state flags of a classical microprocessor. These events can be used to control the subsequent processing. E. g., they can be used to control the merging of data-streams or to deliberately discard data packets. Thus conditional computations depending on the results of earlier ALU operations are feasible. A counter, e. g., generates a special event only after it has terminated. It could be connected to the `RECONF` port of an ALU object, thus triggering the removal of the configuration from the device. Details are provided in the White Paper *Reconfiguration on XPP-III Processors*. Furthermore, events can be combined by the event LUTs and in the connections of the event busses.

## 2.6 XPP-III Reference Design

In addition to the XPP-III Core, the XPP-III Reference Design contains modules used for the integration of XPP-III in SoCs (Systems on Chip). The library of peripherals comprises the following components for the dataflow array: (a) a configuration controller which loads configurations from external memory to the dataflow array, and (b) a DMA controller combined with a four-dimensional address generator for the array

I/O ports. For the FNC-PAEs, the follwing components are provided: (a) an interrupt controller for all FNC-PAEs, (b) arbiters for code and data accesses of FNC-PAEs (which simplify the integration of XPP-III into the memory hierarchy), and (c) an optional data cache which reduces the required memory bandwidth in many applications.

The Reference Design also contains an interrupt controller which arbitrates the external interrupts and triggers the interrupt routines on the FNC-PAEs. Each FNC-PAE provides eight interrupt vectors; one of them is the reset vector. Interrupt sources are e.g. the DMA controllers, soft interrupts and event signals originating from the dataflow array.

# 3 XPP Processing Basics

Applications running on a XPP-III processor achieve a very high performance by exploiting parallelism on different levels. First of all, parallel tasks can be mapped to several FNC-PAEs and to the dataflow array. The dataflow array additionally exploits loop-level (pipelining) parallelism, and the FNC-PAEs exploit instruction-level parallelism and efficient branching. The following give an introduction to XPP data processing on FNC-PAEs and on the XPP dataflow array.

## 3.1 FNC-PAE Processing Basics

One of the essential features of Function-PAEs is their ability for conditional operations within one clock cycle. The following simple example shows the general principle.

```
if (r0 > r2)
   r0 = r2;
if (r0 < r1)
   r0 = r1;
r0 << 1;
```

The value in register r0 is first clipped to the lower and upper bounds defined in registers r1 and r2, respectively. Then, the result is shifted left by one bit. This computation can be mapped to the left and right data paths (columnL, columnR) of a FNC-PAE as shown in Figure 5. The figure shows that two comparisons followed by two conditional operations are required.

Initially, we assume that r0 is within the limits. Therefore, the top-right ALU left-shifts r0. Note that the value in r0 is only available in the next clock cycle. However, r0 may be overwritten by one of the subsequent conditional instructions: On the left path, r0 is compared with the lower limit r1. If r0 is smaller than r1 (LT), the left path is executed and r0 loaded with r1 left-shifted by one. Otherwise the left path is disabled while the right one is enabled. Note that the OPI condition disables the right path if the left path is active. In the left path we compare r0 with r2. If r0 is greater than r2 (GT),

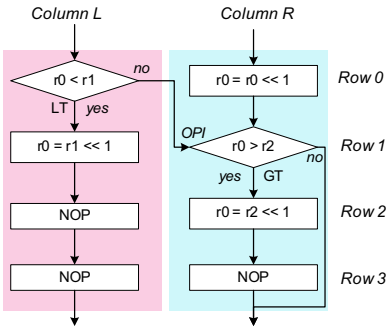


Figure 5: Example mapped to a FNC-PAE's ALU array

r0 is loaded with r2 left-shifted by one. Otherwise the left path is disabled. If both conditions are not true, i.e. r0 is within the limits, the result computed by the top-right ALU is valid. Figure 6 illustrates the runtime evaluation for three different values of r0.

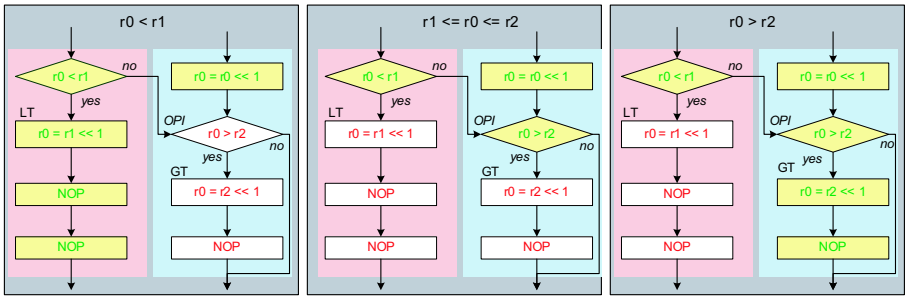


Figure 6: Three possible runtime paths (shaded blocks are enabled)

This feature — in combination with different branch targets for the left and right ALU paths — allows exit conditions for any kind of loops to be evaluated within a single cycle. Using two instruction memories, two branch targets can be reached in a single cycle. If a third branch target is used in the same opcode, an additional cycle is required.

## 3.2 Dataflow Array Processing Basics

The main idea is to combine data-stream processing in an arithmetic array with sophisticated run-time reconfiguration mechanisms. Array configurations are parallel computation modules derived from a dataflow graph of an algorithm. Nodes of the dataflow graph are mapped to fundamental machine operations such as multiplication, addition etc. The operations are implemented by configurable ALUs. After configuration, the routing connections between the ALUs are fixed. They provide an automatically synchronizing, packet-oriented communication network based on a ready-acknowledge protocol.

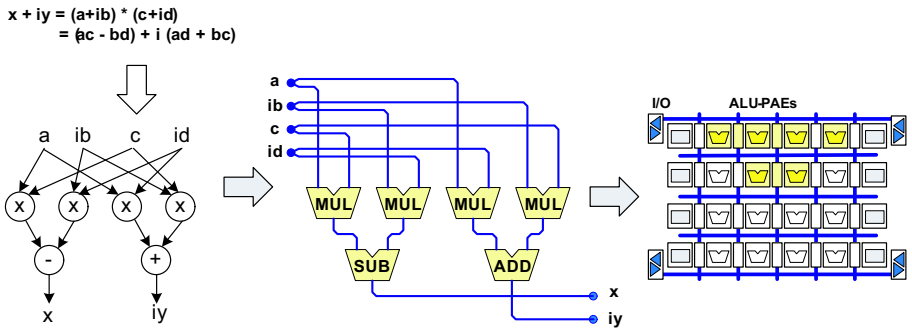


Figure 7: Complex multiplication dataflow graph and mapping to XPP Core

Figure 7 shows the definition of a complex multiplication, its dataflow graph, and the corresponding XPP dataflow configuration. In this example, data packets for successive multiplications continuously stream through the dataflow array. The configuration remains static during the entire computation, i. e. no ALU operator or connection is changed. No reconfiguration or instruction decoding is required, and all operators and input and output ports are active in each cycle, resulting in the optimal performance. After all computations are finished, the ALUs and connections are released and can be used for the next configurations performing the subsequent phases of a larger algorithm. The *reconfiguration time* between configurations can be reduced to a minimum by caching configuration data on-chip. If each configuration processes long enough data streams, the reconfiguration overhead is amortized over many parallel operations. For details, refer to the White Paper *Reconfiguration on XPP-III Processors*.

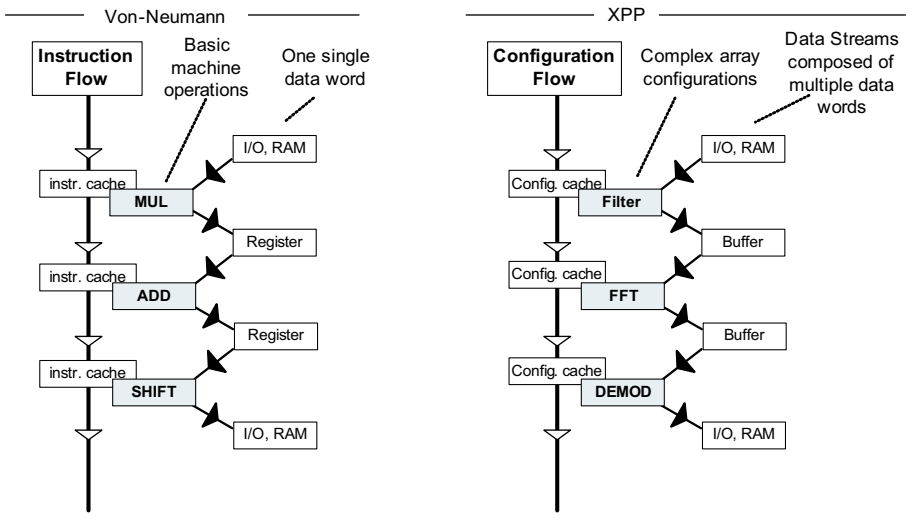


Figure 8: From instruction flow to configuration flow

Intermediate results of computations are stored in distributed memories or FIFOs for use by subsequent configurations. We call this programming paradigm *configuration flow*, as opposed to the *instruction flow* in a classical Von-Neumann architecture where data is moved and processed by ALUs which change their function (instruction) in each processing step. In the XPP dataflow array, no instruction sequencers and caches are continuously active. Since several ALUs process data simultaneously, the XPP Core runs at a relatively low clock speed. This results in a lower power consumption. The difference between the two approaches is illustrated in Figure 8.

The XPP dataflow array is highly suited to computation-intensive applications since many of them can be separated into smaller, inherently parallel phases which process a large number of data in a relatively uniform way.

# 4 Conclusions

This White Paper presented the main components and the structure of an XPP-III Processor. XPP-III is a fully programmable processing platform suitable for a wide range of heterogeneous applications. It is a low-cost and low-power architecture for both highly regular dataflow-oriented and irregular control-dominated tasks. Even the most challenging high-performance applications like H.264 and CABAC decoding at 1080i resolution with 40 Mbits/s data streams can be handled efficiently. The XPP-III provides a significant improvement in performance over standard processor and DSP implementations, and much more flexibility than ASIC implementations.

Refer to the accompanying White Paper *Reconfiguration on XPP-III Processors* for more information on XPP's reconfiguration mechanisms, and to the White Paper *Programming XPP-III Processors* for details on PACT's integrated tool chain for programming, simulating and debugging XPP-III processors.