Mapping of DSP Algorithms on the MONTIUM Architecture

Paul M. Heysters, Gerard J.M. Smit
University of Twente, Department EEMCS,
PO Box 217, 7500 AE, Enschede, The Netherlands
{heysters, smit}@cs.utwente.nl

Abstract

In battery operated mobile devices there is a growing need for flexible high-performance architectures due to the limited amount of available energy and the increasing demand of processing power. Course grained reconfigurable architectures could be the key to more energy-efficient, yet programmable systems. In this paper a course-grained reconfigurable architecture, called MONTIUM[1], is presented. Several mappings of commonly used digital signal processing algorithms are shown to demonstrate the flexibility of this architecture.

1 Introduction

Performance, flexibility and energy-efficiency are the most important requirements for the architecture of a future 3G/4G terminal. Conventional architectures tend to emphasize on only one of these requirements. General-purpose processors and digital signal processors are flexible yet inefficient whereas application specific integrated circuits are efficient yet inflexible. A reconfigurable architecture can obtain a better balance between flexibility and energy-efficiency. A reconfigurable architecture is both flexible and efficient for a particular algorithm domain. Since there are typically multiple algorithm domains required for future 3G/4G terminals, the system architecture should be heterogeneous. The energy-efficiency of the system can be improved significantly by executing computational kernels[2] on algorithm domain specific hardware.

In our CHAMELEON project a heterogeneous system-on-chip (SoC) is being designed [7]. The architectural template for the CHAMELEON SoC is depicted in Figure 1. It contains a general-purpose processor (i.e. an ARM core), a fine-grained reconfigurable part (consisting out of FPGA tiles) and a course-grained reconfigurable part (i.e. MONTIUM tiles). The reconfigurable parts execute highly regular computational kernels. The irregular parts of an algorithm run on the general-purpose processor. This paper focuses on the course-grained reconfigurable part and in particular on the architecture of the MONTIUM processor tile. The target algorithm domain of the MONTIUM comprises 16-bit digital signal processing (DSP) algorithms that contain multiply accumulate (MAC) operations such as FFT, FIR and linear interpolation. However, the MONTIUM is not limited to these algorithms, as will be shown in this paper.

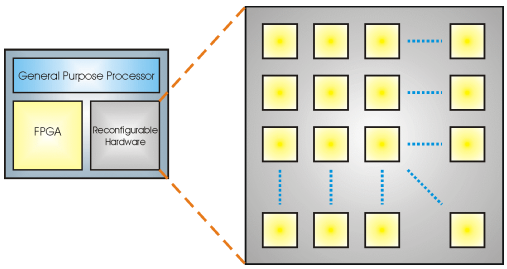

Figure 1: *CHAMELEON heterogeneous SoC*

2 MONTIUM architecture

Figure 2 depicts a single MONTIUM processor tile. The hardware organisation within a tile is very regular and resembles a very long instruction word (VLIW) architecture. The five identical arithmetic and logic units (ALU1…ALU5) in a tile can exploit spatial concurrency to enhance performance. This parallelism demands a very high memory bandwidth, which is obtained by having 10 local memories (M01…M10) in parallel. The small local memories are also motivated by the locality of reference principle. The ALU input registers provide an even more local level of storage. Locality of reference is one of the guiding principles applied to obtain energy-efficiency in the MONTIUM. A vertical segment that contains one ALU together with its associated input register files, a part of the interconnect and two local memories is called a processing part (PP). The five processing parts together are called the processing part array (PPA). A relatively simple sequencer controls the entire PPA. The communication and configuration unit (CCU) implements

[1] The MONTIUM is named after the mountain chameleon, which is indigenous to Cameroon.

[2] The inner loop of a computation, where most time is spent during execution.

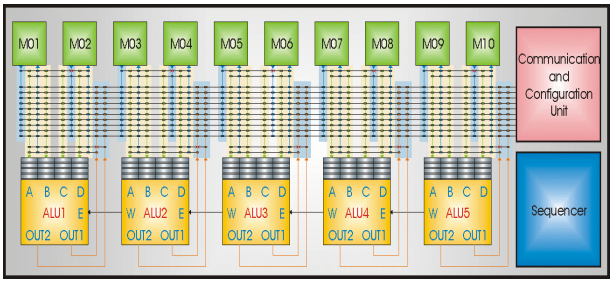

Figure 2: *MONTIUM processor tile*

the interface with the world outside the tile. The MONTIUM has a datapath width of 16-bits and supports both integer and fixed-point arithmetic.

Each local SRAM is 16-bit wide and has a depth of 512 positions, which adds up to a storage capacity of 8 Kbit per local memory. A memory has only a single address port that is used for both reading and writing. A reconfigurable address generation unit (AGU) accompanies each memory. The AGU contains an address register that can be modified using base and modify registers. A mask register is also provided, which can be used to implement cyclic buffers (with a length that is a power of two). It is also possible to use the memory as a lookup table for complicated functions that cannot be calculated using an ALU, such as sinus or division (with one constant). A memory can be used for both integer and fixed-point lookups.

The interconnect provides flexible routing within a tile. The configuration of the interconnect can change every clock cycle. There are ten busses that are used for inter-processing part communication. These busses are called global busses. Note that the span of global busses is only the PPA within a single tile. The CCU is also connected to the global busses. The CCU uses the global busses to access the local memories and to handle data in streaming algorithms. Communication within a PP uses the more energy-efficient local busses.

A single ALU has four 16-bit inputs. Each input has a private input register file that can store up to four operands. The input register file cannot be bypassed, i.e. an operand is always read from an input register. Input registers can be written by various sources via a flexible interconnect. An ALU has two 16-bit outputs, which are connected to the interconnect. The ALU is entirely combinatorial and consequentially there are no pipeline registers within the ALU.

The diagram of the MONTIUM ALU in Figure 3 identifies two different levels in the ALU. Level 1 contains four function units. A function unit implements the general arithmetic and logic operations that are available in languages like C (except multiplication and division). Level 2 contains the MAC unit and is optimised for algorithms such as FFT and FIR. Levels can be bypassed (in software) when they are not needed. If it is known that a level is never used in the aimed algorithm

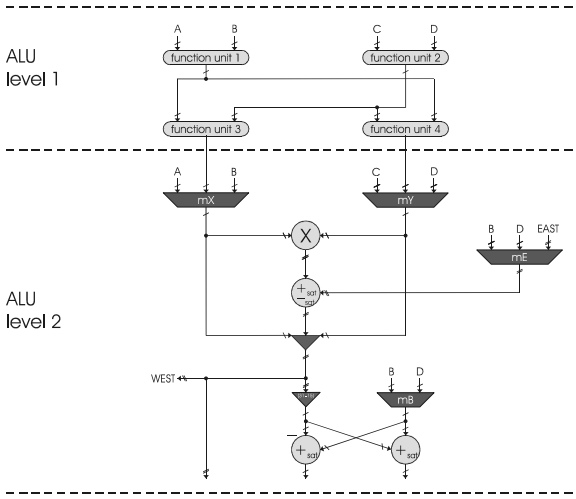

Figure 3: *MONTIUM ALU*

domain, then a MONTIUM can be made in which an entire level is removed. For example, if the algorithms domain is limited to FFT and FIR filters only, then level 1 is not used and a MONTIUM realisation with only level 2 will suffice. The level interface has been designed in such a way that a level can be left out easily. It is in principle also possible to replace a level or to insert a new one. The exact organisation of the ALU should be optimised for the aimed algorithm domain. The current level 1 of the MONTIUM ALU is rather general purpose, because we did not want to restrict the algorithm domain of our experimental processor too much.

Neighbouring ALUs can also communicate directly on level 2. The West-output of an ALU connects to the East-input of the ALU neighbouring on the left (the West-output of the leftmost ALU is not connected and the East-input of the rightmost ALU is always zero). The 32-bit wide East-West connection makes it possible to accumulate the MAC result of the right neighbour to the multiplier result (note that this is also a MAC operation). This is particularly useful when performing a complex multiplication, or when adding a large amount of numbers (up to 20 in one clock cycle). The East-West connection does not introduce a delay or pipeline, as it is not registered.

The implementation of the MONTIUM is still ongoing work. A prototype of the predecessor of the MONTIUM, the field programmable function array (FPFA) [5] was implemented. The architecture of the FPFA is virtually similar to the architecture of the MONTIUM. Key differences are that the datapath of the FPFA is 20-bit wide and that the local memories have a depth of only 256 20-bit words. Also, the control structure of the FPFA offers less flexibility in comparison with the MONTIUM. A single FPFA tile was specified in VHDL. The FPFA was implemented on an FPGA in order to verify the functionality. One FPFA tile fits in a Xilinx Virtex XCV1000 FPGA. The Virtex can run at least at 6.5 MHz.

An ASIC implementation of one FPFA tile in CMOS 0.18µm is predicted to have an area of 2.6mm^2 and to run at least at 23MHz. In this technology we can have approximately 20 FPFA tiles in the same area as an embedded PowerPC. The clock speeds might look a bit disappointing at first, but the FPFA can do an impressive amount of work in just one clock cycle. Also, the VHDL specification of the FPFA design was not optimised in any way.

The availability of high-level design entry tooling is critical for the viability of any reconfigurable architecture. The architecture of the MONTIUM has been kept simple and regular in order to bound the complexity of a compiler. A compiler for the MONTIUM is currently being implemented. Until now, the mapping of algorithms has been done by hand.

3 Typical MONTIUM algorithms

In this section the mapping of several widely used DSP algorithms on the MONTIUM datapath is presented. The mapping of a standard algorithm on the standard MONTIUM tile demonstrates the generality of the architecture or reveals possible problems and shortcomings. Mappings of the following algorithms are discussed: correlation, finite impulse response (FIR) filters, matrix and vector multiplication, Max-Log-MAP decoding, 8x8 point Discrete Cosine Transform (DCT) and Fast Fourier Transform (FFT).

3.1 Correlation

Correlation is a frequent operation in UMTS (universal mobile telecommunications service) receivers [1]. The input signal (*S*) is correlated with a known pseudo noise code (PN-code) sequence (*PN*). The length of the PN-code is equal to the spreading factor (*SF*). In UMTS receivers (i.e. a rake receiver) both the directly transmitted signal and several reflections of the signal are used. As a consequence, correlations with delayed versions of the input signal are also needed. Correlation is similar to a FIR filter with coefficients that have a value of +1 or –1. In pseudo code the correlation algorithm is expressed as follows:

```
for (i=1; i<SF; i++) {
     cor = cor + S[i+Δ]*PN[i]};
```

In this algorithm Δ denotes the delay of the input signal for a particular correlation. A MONTIUM tile can execute 5 correlations simultaneously using two clock-cycles per sample. The basic operation for the correlation can be done in two steps:

1. In the odd clock cycles the sample data (*S[i]*) is written into five local memories and the PN-code (*PN*) is stored in an ALU input register. At the same time (except for the first Δ iterations) the five ALUs calculate five MAC operations.

2. In the even clock cycles the (delayed) input sample (*S[i+Δ]*) is moved from the memories into the ALU input registers. A memory provides a Δ-delayed version of the input signal by using Δ as an offset in the memory address.

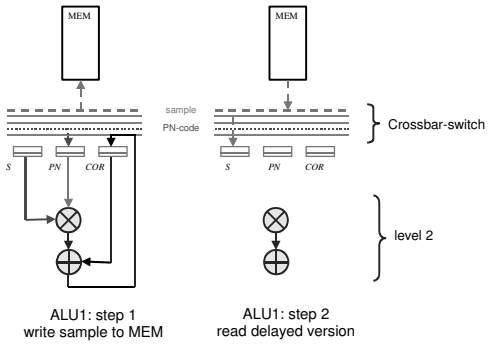

Figure 4*: Correlation*

Figure 4 shows the two steps for ALU 1. Some remarks on this mapping:

1. Steps 1 and 2 cannot be combined since the same memory cannot be read and written (on different addresses) in the same clock cycle.

2. Due to the local memory size of 512 entries, the maximum delay Δ can be at most 512/sample-frequency.

3. For clarity in this mapping only 5 memories are used. The ALUs are computing only during step 1. An optimised mapping that uses all 10 memories and computes a MAC operation every clock cycle is also possible. In this way, 10 correlations can be computed using 2 clock-cycles per sample.

3.2 Finite-impulse response filter

The finite impulse response filter (FIR) is a frequently used algorithm in digital signal processing applications. Figure 5 shows two implementations of a 4-tap FIR filter: the direct form and the transposed form. Each tap contains a multiplication, an addition and a delay. As shown in Figure 5, the direct form FIR filter uses three inputs and two outputs for every tap. The transposed form FIR filter uses three inputs and one output. However, in the latter case, the input is routed to all the taps.

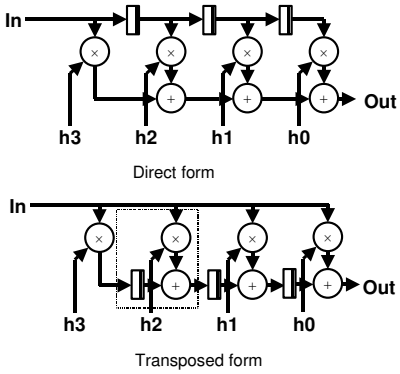

Figure 5: *4-tap FIR filter*

5-tap FIR filter

Figure 6 shows a straightforward mapping of a five-tap transposed for FIR filter onto one MONTIUM tile. Once the FIR filter is started, every clock cycle an additional ALU is used, until all five ALUs are in use. Thus, after a start-up delay of five clock cycles, every subsequent clock cycle returns a result. This implementation shows that every clock cycle 5 MACs are computed and that the overhead is negligible. This FIR filter can be used in a streaming mode as well as in block mode. In streaming mode the CCU provides a new sample on a bus every clock cycle. Similarly one result value has to be read from a bus in every clock cycle. In block mode the CCU first writes a block of input samples to one or more memories. Subsequently, the filter computation reads the samples from the memory and writes the results back to another memory. Finally, the results are read by the CCU and forwarded to another tile. Whether streaming or block mode is used depends on the (delay) requirements of the application.

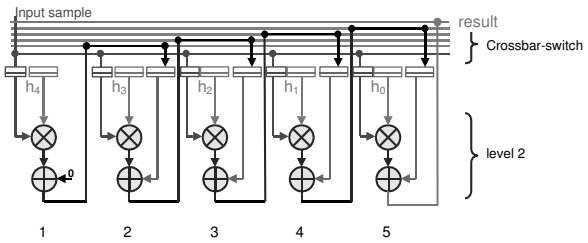

Figure 6*: Five-tap finite-impulse response filter*

n-tap FIR filter (using registers)

If more than 5 taps are needed, the mapping of the transposed form FIR filter shown above can be time multiplexed. The intermediate results and the filter coefficients are stored in the registers. Since the number of registers is limited, the maximum number of taps that can be implemented in this way is also limited. Each processing part has 16 registers. The input sample has to be stored in one register, so there are 15 registers left for the filter coefficients and intermediate results. Each additional coefficient that is computed by a particular ALU also requires an additional register for an intermediate result. So, the registers in each processing part can store up to 7 coefficients and 7 intermediate results. In this manner, the processing part array can compute a FIR filter of up to 7x5=35 taps. A 35-tap FIR filter has a delay of 7 clock cycles. If this delay is not acceptable the entire filter operation has to be partitioned over more tiles.

n-tap FIR filter (using memories)

FIR filters with more than 35 taps can be implemented by storing the delayed sample values and filter coefficients in the local memories. In this mapping the direct form of the FIR filter is used. The left memory of each processing part (PP) contains one-fifth of the (delayed) input samples and the right memory one-fifth of the filter coefficients. An *n*-taps filter is divided into 5 equal parts. Each PP computes the partial sum of *n/5* taps of the filter (see Figure 7). It takes one clock cycle to load the registers and *n/5* clock cycles are needed to compute the five partial sums in the five PPs. The five partial sums have to be added together to produce the final output. This requires another three clock cycles. However, loading the registers at the beginning of the partial sum calculation and the computation of the final output can overlap. In general the delay of a sample is *n/5+2* clock cycles. A new input sample "streams" into the left memory of PP1 and overwrites the oldest sample in this memory. The oldest sample in the left memory of PP2 is replaced by the sample that is still in an input register of PP1 (this is the sample that was removed from the left memory in PP1). In this way, the oldest sample of every memory is shifted to the right. This memory shift operation is also done in parallel with the computation of the final output.

For the memory addressing it is required that *n/5* is a power of two. This can easily be achieved by adding coefficients of value zero to a FIR filter. A FIR filter with at most 5x512= 2560 taps and a delay of 512+2=514 clock cycles can be implemented. Some variants are also possible; for example: the five PPs can also be used to compute five 512-tap FIR filters (each with a delay of 512 clock cycles).

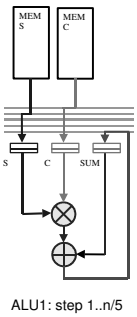

Figure 7: *Mapping of n-tap filter using memories*

3.3 Matrix/matrix multiplication and matrix/vector multiplication

A single MONTIUM tile can compute a 64×64 element matrix *A[1..64;1..64]* times a 64 element vector *B[1..64]*. The matrix is stored in the 8 memories of four PPs. The vector *B[1..64]* and the result *C[1..64]* (also a 64 element vector) is stored in the remaining two memories. Four PPs compute the MACs for the result. For each element of the result 64 MACs have to be computed. So, each element requires 64/4 = 16 clock cycles. The total execution time of a 64×64 matrix times a 64 element vector is 16x64=1024 clock cycles. The same mapping can be used for both block mode and streaming mode. In streaming mode a new *B* element has to be presented every 16 clock cycles. At the same time a new *C* element will be available. Note that the maximum size of the matrix is determined by the size of the local memories. Two 32×32 matrices can be multiplied in 8x32x32 = 8192 clock cycles.

3.4 Max-Log MAP algorithm

In [6] the SISO Max-Log-MAP algorithm that can be used for (UMTS) turbo decoding is mapped onto the FPFA/MONTIUM datapath. This algorithm only uses level 1 of the ALU. Calculation of the forward state metric is of *O(2m)* clock cycles. Calculation of the backward state metric and the soft output is of *O(7m)* clock cycles. The total execution time for the Max-Log-MAP algorithm on a single Montium tile is approximately *9m* clock cycles. The Montium memory depth of 512 positions limits the maximum block size *m* to 510 (the forward state metric requires *m+1* positions and the initialisation of the backward state metric requires one more position). However, Dielissen and Huisken [3] suggested a way to circumvent this memory restriction, at the expense of using more tiles. They show that it is feasible to divide the input block into smaller portions of e.g. 510 bits. In this way larger block sizes can be handled and more work can be done in parallel. This approach can also be used to speed up the algorithm.

3.5 8×8 point DCT algorithm

An 8×8 point discrete cosine transform (DCT) can be computed by eight iterations of an 8-point 1d-DCT algorithm. We used the fixed-point 1d-DCT implementation from [4]. A control dataflow graph (CDFG) was (automatically) generated from the unmodified C code. This CDFG was (manually) mapped onto the MONTIUM architecture. This algorithm comprises a mixture of multiplications, additions, subtractions and shift operations. Consequently, both level 1 and level 2 are used. The DCT algorithm is not as regular as the other algorithms discussed in this paper and needs all the flexibility of the ALUs. The 8-point 1d-DCT can be executed on the MONTIUM architecture in 6 clock cycles. The complete 8×8 DCT requires 8x6= 48 clock cycles.

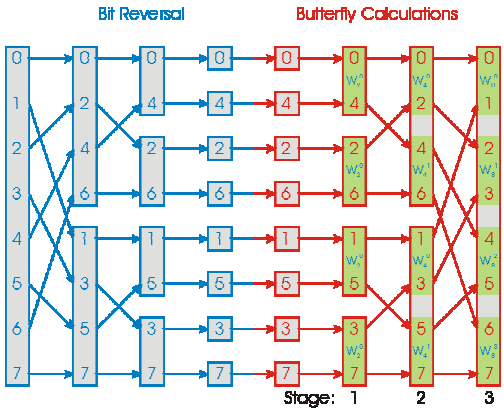

Figure 8*: Recursion of a radix 2 FFT with 8 inputs*

3.6 Fast Fourier Transform

The Fourier transform enables the conversion of signals from the time domain to the frequency domain (and vice versa). For digital signal processing, the Discrete Fourier Transform (DFT) is of particular interest. The Fast Fourier Transform (FFT) can be used to calculate a DFT efficiently [2]. The FFT recursively divides a DFT into smaller DFTs. Eventually only basic DFTs remain. These DFTs have a number of inputs that is equal to the radix of the FFT. This is illustrated in the left ("bit reversal") part of Figure 8 for a radix 2 FFT with 8 input signals.

The resulting basic DFTs can be calculated by a structure called a butterfly. The butterfly is the basic element of a FFT. An *n*-point FFT (in which *n* is a power of 2) requires $(n/2)\log_2(n)$ butterfly operations.

Figure 9 depicts the radix 2 butterfly; *a* and *b* are complex inputs and *A* and *B* are complex outputs. *W* is a complex constant called the twiddle factor. The radix 2 butterfly consists of a complex multiplication, a complex addition and a complex subtraction.

The FFT butterfly depicted in Figure 9 can be written as equations Eq. 1 and Eq. 2. A hardware algorithm for the radix 2 FFT butterfly has six inputs (a_{re}, a_{im}, b_{re}, b_{im}, W_{re}, W_{im}) and four outputs (A_{re}, A_{im}, B_{re}, B_{im}). Each input is used two times. Three subtraction, four multiplication and three addition operations are used.

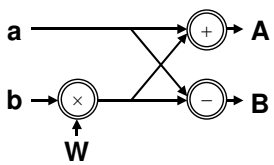

Figure 9: *The radix 2 FFT butterfly*

$$A = a + W \cdot b \equiv (a_{re} + a_{im}) + ((W_{re} \cdot b_{re} - W_{im} \cdot b_{im}) + (W_{re} \cdot b_{im} + W_{im} \cdot b_{re})_{im}) \quad \textbf{Eq. 1}$$

$$B = a - W \cdot b \equiv (a_{re} + a_{im}) + ((W_{re} \cdot b_{re} - W_{im} \cdot b_{im}) + (W_{re} \cdot b_{im} + W_{im} \cdot b_{re})_{im}) \quad \textbf{Eq. 2}$$

$$A_{re} = a_{re} + (W_{re} \cdot b_{re} - W_{im} \cdot b_{im}) \quad \textbf{Eq. 3}$$

$$B_{re} = a_{re} - (W_{re} \cdot b_{re} - W_{im} \cdot b_{im}) \quad \textbf{Eq. 4}$$

$$A_{im} = a_{im} + (W_{re} \cdot b_{im} + W_{im} \cdot b_{re}) \quad \textbf{Eq. 5}$$

$$B_{im} = a_{im} - (W_{re} \cdot b_{im} + W_{im} \cdot b_{re}) \quad \textbf{Eq. 6}$$

Equations Eq. 1 and Eq. 2 can be rewritten to equations Eq. 3, Eq. 4, Eq. 5 and Eq. 6. The latter equations can be mapped to the MONTIUM ALU directly. Figure 10 shows how the FFT butterfly can be calculated on four MONTIUM ALUs. The FFT butterfly is calculated in one clock cycle. There is only one clock cycle overhead per stage, i.e. an *n*-point FFT is computed in $(n/2)(log_2(n)+1)$ clock cycles. The loading of the registers at the beginning of each stage causes the extra cycle. In all subsequent cycles loading the data from memory, storing the intermediate results and executing the butterflies are done in parallel.

The computation of the FFT is very regular. The ALUs need to execute only one instruction. The input register files in a PP also need only one instruction, since always the same input registers are loaded. (Two instructions are needed for the input registers if, for energy-efficiency reasons, we do not want to load the registers in the last clock cycle of a stage.) The crossbar requires at most 8 instructions; an input value can be read from two possible memories, a result can be written to two possible memories and the read and write memories exchange every stage. The address generation is also very regular. This is accomplished by computing the butterflies within a stage in a particular order. The memory addressing of the twiddle factors is the most complicated and requires 11 different memory instructions for a 1024-point FFT. A 1024-point FFT is the largest FFT that can be computed within one MONTIUM tile (a 1024-point FFT is computed in 5130 clock cycles.). The maximum size is determined by the size of the local memories.

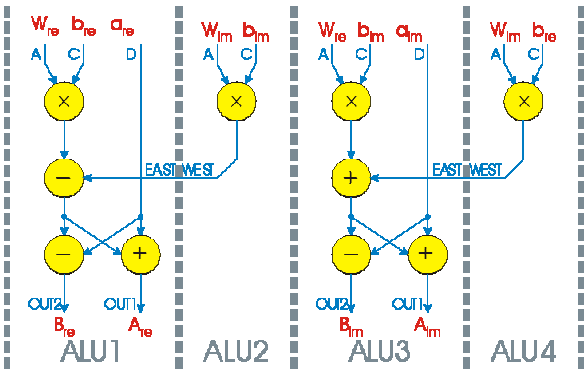

Figure 10: *The radix 2 FFT butterfly*

4 Conclusion

In this paper the MONTIUM course-grained reconfigurable architecture is outlined. Performance, flexibility and energy-efficiency are the most important design goals for the CHAMELEON heterogeneous SoC architectural template. A high performance can be obtained by the parallelism in a tiled architecture. The design of the MONTIUM tile focuses on a trade-off between flexibility and (energy-) efficiency. We believe that the MONTIUM architecture is energy-efficient, because the locality of reference principle is used extensively. In this paper, however, the focus is on the flexibility of the architecture. Mappings of a variety of DSP algorithms onto the MONTIUM architecture were presented. These mappings show the flexibility of the MONTIUM architecture.

Acknowledgements

This research is supported by PROGRESS, the embedded systems research program of the Dutch organisation for Scientific Research NWO, the Dutch Ministry of Economic Affairs and the Technology Foundation STW.

References

[1] 3GPP TSG RAN WG1: "TS 25.212 Multiplexing and Channel Coding (FDD) V3.1.1 (1999-12)", *http://www.3GPP.org*.

[2] Cooley J. W., Tukey J. W. : "An Algorithm For the Machine Calculation of Complex Fourier Series", *Mathematics of Computation*, Vol. 19, pages 297-301, April 1965.

[3] Dielissen J., Huisken J.: "Implementation issues of 3rd generation mobile communication Turbo decoding", *Proceedings 21st symposium on information theory in the Benelux*, 2000.

[4] http://www.ee.ucla.edu/~schaum/ee201a/jpeg_c_fix/dct_8c-source.html.

[5] Heysters P.M., Bouma H., et al.: "A Reconfigurable Function Array Architecture for 3G and 4G Wireless Terminals", *Proceedings of World Wireless Congress*, pages 399-404, San Francisco, USA, May 2002.

[6] Heysters P.M., Smit L.T., et al.: "Max-Log-MAP Mapping on an FPFA", *Proceedings of Engineering of Reconfigurable Systems and Algorithms*, pages 90-96, Las Vegas, USA, June 2002.

[7] Smit G.J.M., Havinga P.J.M., et al.: "Dynamic Reconfiguration in Mobile Systems", *Proceedings of Field-Programmable Logic and Applications*, pages 171-181, Montpellier, Fance, September 2002.