Mapping the SISO module of the Turbo decoder to a FPFA

Gerard J.M. Smit, Paul M. Heysters, Paul J.M. Havinga, Lodewijk T. Smit, Jaap Smit
John Dilessen, Jos Huisken[1]

University of Twente, dept. of Computer Science, Enschede, the Netherlands
smit@cs.utwente.nl

Abstract - In the CHAMELEON project[2] a *reconfigurable systems-architecture,* the *Field Programmable Function Array* (FPFA) is introduced. FPFAs are reminiscent to FPGAs, but have a matrix of ALUs and lookup tables instead of Configurable Logic Blocks (CLBs). The FPFA can be regarded as a low power reconfigurable accelerator for an application specific domain. In this paper we show how the SISO (Soft Input Soft Output) module of the Turbo decoding algorithm can be mapped on the reconfigurable FPFA.

I. INTRODUCTION

Next generation personal mobile multimedia terminals support wireless multimedia communication. A key challenge of these terminals is that many attributes of the wireless environment vary *dynamically*. These devices need to be able to operate in environments that can change drastically in short term as well as long term in available resources and available services.

Programmability and *adaptability* is thus an important requirement for mobile computing, since the mobiles must be flexible enough to accommodate a variety of multimedia services and communication capabilities and adapt to various operating conditions in an (energy) efficient way. Merely algorithmic adaptations are not sufficient, but rather an entirely new set of protocols and/or algorithms may be required. For example, mobile users may encounter a complete different wireless communication infrastructure when walking from their office to the street. A possible approach to achieve adaptivity is to have a mobile device with a reconfigurable architecture that can adapt its operation to the current environment and operating condition. Research has shown that adapting continuously the system and protocols can significantly improve the energy efficiency while maintaining a satisfactory level of performance [9].

Reconfigurability also has another more economic motivation: it will be important to have a fast track from sparkling ideas to the final design. If the design process takes too long, the return on investment will be less. It would further be desirable for a wireless terminal to have architectural reconfigurability where its capabilities may be modified by downloading new functions from network servers. Such reconfigurability would also help in field upgrading as new communication protocols or standards are deployed, and in implementing bug fixes.

In the CHAMELEON project [2] we have designed a reconfigurable architecture that is suitable for many DSP-like algorithms and yet is energy-efficient. In this paper we will show how the SISO algorithm of Turbo-decoding [5] maps on this architecture.

---

[1] John Dilessen and Jos Huisken are at Philips Research Laboratories, Eindhoven, the Netherlands.

[2] 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 under project number TES.5004.

II. Field Programmable Function Array

Field Programmable Function Arrays (FPFAs) are reminiscent to FPGAs, but have a matrix of ALUs and lookup tables [8] instead of Configurable Logic Blocks (CLBs). Basically the FPFA is a low power, reconfigurable accelerator for an application specific domain. Low power is mainly achieved by exploiting locality of reference. High performance is obtained by exploiting parallelism. A FPFA consists of interconnected processor tiles. Multiple processes can coexist in parallel on different tiles. Within a tile multiple data streams can be processed in parallel. Each processor tile contains multiple reconfigurable ALUs, local memories, a control unit and a communication unit. Figure 1 shows a FPFA with 25 tiles; each tile has five ALUs.

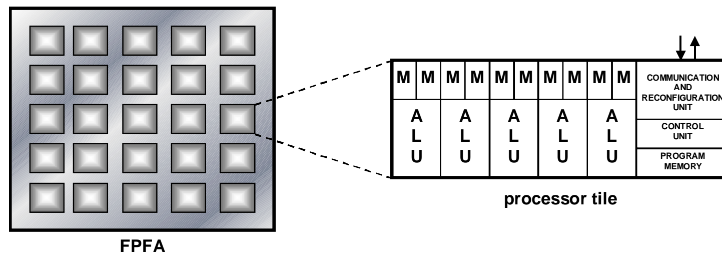

Figure 1: *FPFA architecture.*

The ALUs on a processor tile are tightly interconnected and are designed to execute the (highly regular) inner loops of an application domain. ALUs on the same tile share a control unit and a communication unit. The ALUs use the locality of reference principle extensively: an ALU loads its operands from neighboring ALU outputs, or from (input) values stored in lookup tables or local registers.

Processor tile

A FPFA processor tile in Figure 1 consists of five identical blocks, which share a control unit and a communication unit. An individual block contains an ALU, two memories and four register banks of four 20-bit wide registers. Because of the locality of reference principle, each ALU has two local memories. Each memory has 256 20-bit entries. A crossbar-switch makes flexible routing between the ALUs, registers and memories possible. Figure 2 shows the crossbar interconnect between five blocks. This interconnect enables an ALU to write-back to any register or memory within a tile.

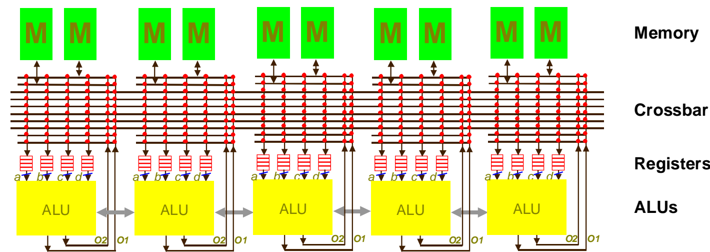

Figure 2: *Crossbar-switch.*

Five blocks per processor tile seems reasonable. With five blocks there are ten memories available. This is convenient for the Fast Fourier Transform algorithm, which has six inputs and four outputs [8]. Also, we now have the ability to use $5 \times 16 = 80$-bit wide numbers, which enable us to use floating-point numbers (although some additional hardware is required). Some algorithms, like the Finite impulse-response filter, can benefit substantially from additional

ALUs. With five ALUs, a five-tap FIR filter can be implemented efficiently. The fifth ALU can also be used for complex address calculations and other control purposes.

ALU datapath

The datapath of the FPFA-ALU is depicted in Figure 3. The ALU has 4 inputs (*a, b, c, d*) and 2 outputs (*OUT1, OUT2*). The in- and outputs are 20-bit wide and use a sign-magnitude representation. The internal datapaths are either 20 or 40-bit wide. The internal data representation is signed-magnitude for the multiplier and 2-complement for the adders. The datapath shown is only suitable for integer arithmetic.

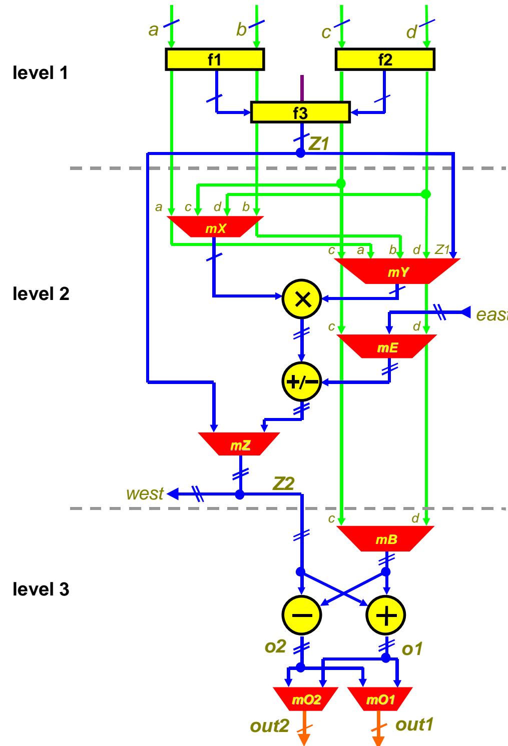

Figure 3: *ALU datapath design.*

In the ALU three different levels can be discriminated. We will now discuss the behavior of each level.

- *Level one* is a reconfigurable function block. Each function f_1, f_2 and f_3 in Figure 3 returns the following result:

$$
\begin{aligned}
f_n \quad = \quad & 0 \\
| \quad & [abs][-]Op_{left} \\
| \quad & [abs][-]Op_{right} \\
| \quad & [abs]+([-]Op_{left},[-]Op_{right}) \\
| \quad & [abs]\min([-]Op_{left},[-]Op_{right}) \\
| \quad & [abs]\max([-]Op_{left},[-]Op_{right})
\end{aligned}
$$

  The result of level one is:

$$Z1 = f_3\left(f_1(a,b), f_2(c,d)\right)$$

- *Level two* contains a 19×19-bit unsigned multiplier and a 40-bit wide adder. The *east-west interconnect* connects neighboring ALUs. A value in the *east* input can be added to the multiplier result. The result of level two *Z2* is used as input for level 3 and for the ALU connected to the *west* output. We can express the result of level two as:

  $$Z2 = Z1 \;|\; +(a|b|c|d \cdot a|b|c|d \,|\, Z1, [-]0 \,|\, C_{se} \,|\, d_{se} \,|\, east)$$

  The *se* subscript means that the numbers are *sign extended*. Note that it is possible to bypass level two.

- *Level three* can be used as a 40-bit wide adder or as a butterfly structure:

  $$o2 = +(0 \,|\, c_{se} \,|\, d_{se} \,|\, cd, -Z2)$$

  $$o1 = +(0 \,|\, c_{se} \,|\, d_{se} \,|\, cd, Z2)$$

  Two 20-bit words can be selected as the final result of the ALU:

  $$out2 = o1_{high} \;|\; o1_{low} \;|\; o2_{high} \;|\; o2_{low}$$

  $$out1 = o1_{high} \;|\; o1_{low} \;|\; o2_{high} \;|\; o2_{low}$$

III. TURBO DECODING

The introduction of *Turbo Codes* by Berrou et al. [1] in 1993 opened up new perspectives in channel coding theory. The outstanding bit error rate performances and the wide range of applications created a large interest in this coding scheme.

As an example we use the Turbo coding scheme for 3GPP-UMTS [3], it consists of two 8-state Recursive Systematic Coders (RSC) and an interleaver. Figure 4 shows the structure of the encoder. An 8-state RSC module is a 3-bit shift register with XOR functions in the loops. The output of the Turbo Encoder consists of the original message (systematic output *syst* from RSC 1) and the parity output *enc1* and *enc2* from both RSCs, resulting in a rate 1/3 code. This structure is also known under the name Parallel Concatenated Convolution Code (PCCC).

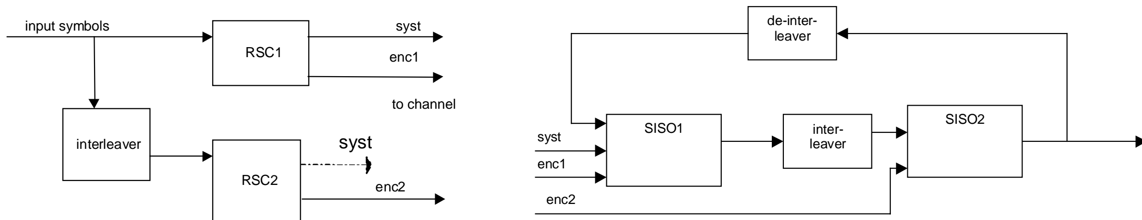

Figure 4: *Block diagram of the Turbo encoder and decoder*

Turbo Codes are block codes. The block-length of the message is determined by the interleaving length. In 3GPP-UMTS this interleaving length is specified between 40 and 5120 bits. In the interleaver the input bits are written row wise into a matrix. After inter- and intra- row permutations the matrix is read column wise. The total number of bits which are sent over the channel is approximate *3*m*, where *m* is the block-length of the message. This data (*syst*, *enc1* and *enc2* in Figure 4 and 5) is sent over the wireless link to the receiver. At the receiver side the errors of the channel have to be corrected, with a Turbo-decoding algorithm.

Figure 4 shows a simplified block-diagram of the Turbo-decoder. The Turbo Decoder consists of two types of modules: the (de-)interleaver and the Soft-Input Soft-Output (SISO) module. For SISO decoding there are two families: BCJR (named after its inventors Bahl, Cocke, Je-linek, and

Raviv) [7] and SOVA (Soft Output Viterbi Algorithm) [5]. BCJR-type algorithms have better performance and scalability, but are more computationally expensive. For example when using a BCJR-type SISO module we can reduce the average number of iterations resulting in lower implementation costs (energy and latency) [6]. In this paper we use the BCJR algorithm, in particular the maxLog-MAP algorithm. Turbo Codes got their name from the feedback structure shown in Figure 4 and its analogy to a turbo engine. The data iterates several times through the decoder, reducing the Bit-Error-Rate of the message in each iteration. The iteration can be stopped when all errors are corrected or when a predefined maximum number of iterations is reached. Eventual residual errors must be corrected in higher protocol layers.

Each SISO module passes information via a (de-)interleaver to the next SISO module, which in turn refines the estimated probability of the received information, using the information derived by the other decoder.

One of the parameters of the algorithm is the number of iterations of the decoder. The more iterations the less errors, but the more latency. The number of iterations can be used as a QoS parameter, e.g. when the number of bit errors is below a QoS specification the algorithm can be stopped. In this way we have a provision for an adaptable error mechanism, only when needed the algorithm will use the full number of iterations. As mentioned above adaptability is a crucial property for energy efficient mobile systems. When channel conditions are good, e.g. the receiver is close to the transmitter, the Turbo Decoding algorithm stops after only a few iterations.

The main disadvantage of Turbo decoding is the computational complexity of the algorithm. In this paper we will show how the SISO parts of the Turbo decoding algorithm can be efficiently mapped on our reconfigurable architecture.

IV. MAPPING OF SISO ALGORITHM

In this section we show how the SISO algorithm can be mapped on a single FPFA tile. There are several implementation alternatives for the SISO algorithm [7][6][5][4]. In our approach the algorithm first computes a *forward state metric A[0..7,0 ..m-1]*. Then it computes at the same time the *backward state metric B[0..7,0 ..m-1]* and the *output vector LLR[0..m-1]* of log-likelihood ratios. In this way the *B* state metric does not have to be stored. We explain the mapping of the computation of the forward state metric in detail, the mapping of the backward state metric calculations and the computation of the soft output is similar. It is important to note that we have not modified our FPFA-ALU nor the SISO algorithm for this particular mapping. Our intention was to find out whether a standard SISO algorithm could be mapped on our standard FPFA-tile, and to investigate what the possible problems and shortcomings of the architecture are.

Because we use a standard FPFA, the element size of the input vectors and the state metrics is 16 bits. For the calculation of the forward state metric we can start with the first column of the forward state metric *A[0..7,0]* and 2 input vectors *L[0..m-1]* and *P[0..m-1]*.Using this information and the 2 input vectors we can compute the second column of the forward state metric, after that the third, etc. In this way the entire forward state metric is computed iteratively and is saved in the matrix *A[0..7,0..m-1]* for block length *m* (see Program 1 below and Figure 5).

Similarly the backward state metric is computed after the last forward state metric computation. However, the soft output calculation starts immediately after the calculation of the corresponding backward state metric vector in the backward recursion. The corresponding forward state metric vector *A[0..7,0]* is retrieved from the memory in which it was saved during forward recursion. Note that in this way the soft output is generated in a backward fashion, i.e. the last metric of the block is computed first.

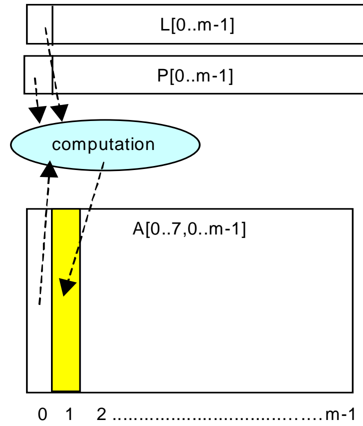

Figure 5: *Compute forward state metric A[0..7,1] using L[0], P[0] and A[0..7,0]*

To illustrate the mapping of the SISO algorithm the forward state metric is shown.

```
for (k=1;k<=m;k++)
 {
  LBut(A[0][k-1], A[1][k-1], 0.0,              L[k-1]+P[k-1],    AI[0][k], AI[4][k]);
  LBut(A[2][k-1], A[3][k-1], L[k-1],  P[k-1],             AI[1][k], AI[5][k]);
  LBut(A[4][k-1], A[5][k-1], P[k-1],  L[k-1],             AI[2][k], AI[6][k]);
  LBut(A[6][k-1], A[7][k-1], L[k-1]+P[k-1],   0.0,              AI[3][k], AI[7][k]);
}
```

with

```
TDMetric HLBut(TDMetric X1, TDMetric X2, TDMetric Y1, TDMetric Y2)
{ return Max(X1+X2,Y1+Y2); }

void LBut(TDMetric X, TDMetric Y, TDMetric H, TDMetric D, TDMetric &Z1,
        TDMetric &Z2)
{ Z1 = HLBut(X,H,Y,D);
  Z2 = HLBut(Y,H,X,D);}
```

Program 1: *The first part of the SISO algorithm in C*

From the algorithm (Program 1) we can see that for each iteration of the algorithm, the eight values of the state metric *AI[0..7,i]* can be computed in parallel. Furthermore we see that for each LBut function call we have two HLBut calls with 2 additions and 1 max operations each. In our FPFA-ALU we can do the 2 additions and the max operation of one HLBut function in one clock cycle. So for each LBut call we need two ALUs when we want to calculate them in parallel. In our FPFA tile we have five ALUs. Four out of these five ALUs are used to compute the forward state metric.

The remaining ALU is used to for other computations e.g. for computing *L[i]+P[i]*. So the 4 LBut function calls can be mapped on 4 ALUs and can be executed in 2 clock cycles. Figure 6 shows the detailed mapping of the forward state metric computation of the SISO algorithm on the 2 FPFA ALUs. The intermediate results are placed in registers and the forward state metric is placed in the memories [*M1..M8*]. Because the algorithm is executed in two clock-ticks, the intermediate results are stored in two different registers in two subsequent iterations, e.g. *A0* and *AI0*. It is assumed that there is a memory addressing function such that subsequent *A* values can

be stored in FIFO order in the memories. The backward recursion function needs the values in backward order, so the last stored value is used first.

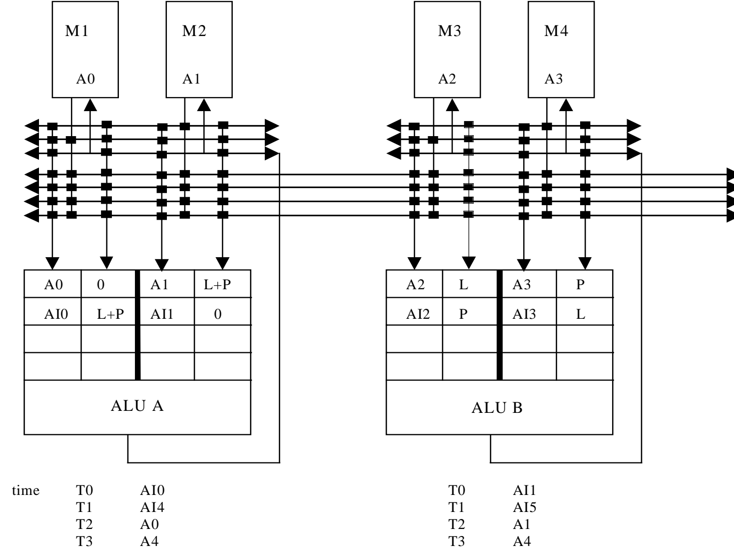

Figure 6: *Mapping of SISO to FPFA*

The calculation of the forward state metric takes *2m* clock cycles the calculation of the backward state metric and the soft output takes *7m* clock cycles, with *m* the length of the block. So in total *9m* clock cycles are needed to execute the SISO algorithm in a single FPFA tile.

V. DISCUSSION AND RESULTS

In this section we provide some discussions on the mapping of the SISO algorithm to the FPFA presented above.

1. For the mapping of the SISO algorithm we only use the level 1 of the ALUs. To use the ALUs in an energy efficient way, the results of level 1 have to be bypassed through the other ALU levels to the ALU output.
2. Currently our ALU memories are 256 word deep. In the herefore mentioned mapping we only can map a SISO algorithm with block size $m \leq 256$.
3. Dielissen and Huisken [6] suggested a way to circumvent the above mentioned memory restriction, but at the expense of using more FPFA tiles. They show that it is feasible to divide the input block into smaller portions of e.g. 256 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.

As already mentioned before the SISO algorithm is rather compute intensive. Measurements in the original Turbo-decoding algorithm showed that a Pentium executes 2.818.000 instructions in the SISO algorithm and 80.000 instructions in the (de-) interleaver for a block size of 5002 bits[3]. Note that these numbers are *executed* instructions and not compiler *generated* instruction. The table also shows that compiling with the optimization switches (-O3) on, has a tremendous impact. Table 2 shows the amount of instruction executed on a Pentium processor for the interleaver and SISO modules.

[3] Compiled on a gcc compiler version 2.7.2.3

Processor	Number of instructions executed for SISO algorithm	Number of instructions executed for Interleaver
Pentium with optimization -O3	2.818.000	80.000
Pentium without optimization	9.755.000	402.000
FPFA instructions	45.018	Not available

Table 2 *Number of executed instructions for block size m = 5002*

The standard FPFA tile does the SISO algorithm for a block size of $m = 5002$ bits in $9m = 45.018$ clock cycles. There are a few remarks in place here:

1. the FPFA runs at a much lower clock speed compared to a Pentium,
2. the standard FPFA can only deal with a block size up to 256 bits,
3. we assume that the input vectors and the first column of the forward state metric is already in the memories. This assumption was also made in the measurements of the SISO algorithm on a Pentium.

VI. CONCLUSION

In this paper we gave an overview of a reconfigurable architecture called Field Programmable Function Array (FPFA). We showed how the SISO module of the Turbo decoding algorithm can be mapped on this reconfigurable architecture. For the mapping of the SISO algorithm the limitation of the FPFA-ALU is the memory size. Therefore the block length is limited to 256 bits. With a proper partitioning of the input data block this limitation can be circumvented and more work can be done in parallel at the expense of using more FPFA-ALUs. In the mapping we assumed that the input vectors and the first column of the forward state metric is already in the memory of the FPFA-ALU. We did not include the reading of the input data and delivering of the result data in our calculations. The next step will be to compare the energy consumption of a FPFA with a standard processor (e.g. StrongARM) and an ASIC implementation.

REFERENCES

[1] C. Berrou, A. Glavieux, and P. Thitimajshima: "Near Shannon limit error-correcting coding and decoding: Turbo codes", In *IEEE Proceedings of ICC '93*, pages 1064–1070, May 1993.

[2] CHAMELEON project, http://chameleon.ctit.utwente.nl.

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

[4] G. Masera, G. Piccinini, M. Ruo Roch, and M. Zamboni:"VLSI architectures for turbo codes", *IEEE Transactions on VLSI Systems*, 7(3):369–378, September 1999.

[5] M.C. Valenti: "Turbo Codes and Iterative Processing", 1999, www.ee.vt.edu/valenti/turbo.html

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

[7] L.R.Bahl, J,Coeke, F.Jelinek, F.Raviv: "Optimal decoding of linear codes for minimizing symbol error rate", *IEEE Trans. Inform. Theory,* vol. 20, pp. 284-287, March 1974.

[8] Paul M. Heysters, Jaap Smit, Gerard J.M. Smit, Paul J.M. Havinga: " Mapping of DSP Algorithms on Field Programmable Function Arrays", *proceedings FPL2000*, Villach, Austria, August 28 - 30, 2000.

[9] P. Lettieri, M.B. Srivastava: "Advances in wireless terminals", *IEEE Personal Communications*, pp. 6-19, February 1999.

[10] Xilinx, "Virtextm-E 1.8V Field Programmable Gate Arrays", *Advance Product Specification*, December, 1999.