Sloppy Addition and Multiplication

Nannarelli, Alberto

Publication date:
2011

Document Version
Publisher's PDF, also known as Version of record

Link back to DTU Orbit

Citation (APA):
Nannarelli, A. (2011). Sloppy Addition and Multiplication. Kgs. Lyngby, Denmark: Technical University of Denmark (DTU). (IMM-Technical Report-2011; No. 14).

Sloppy Addition and Multiplication

IMM-Technical Report-2011-14

Alberto Nannarelli

Dept. Informatics and Mathematical Modelling
Technical University of Denmark
Kongens Lyngby, Denmark
Email: an@imm.dtu.dk

Abstract

Sometimes reducing the precision of a numerical processor, by introducing errors, can lead to significant performance (delay, area and power dissipation) improvements without compromising the overall quality of the processing. In this work, we show how to perform the two basic operations, addition and multiplication, in an imprecise manner by simplifying the hardware implementation. With the proposed "sloppy" operations, we obtain a reduction in delay, area and power dissipation, and the error introduced is still acceptable for applications such as image processing.

1 Introduction

In common language the adjective *"arithmetical"* usually indicates something very precise or error-free. However, also arithmetic operations have to be put in the *"context'*. There are several fields of application of computer arithmetic that can tolerate some imprecision. For example, in audio and image processing or in wireless communication, it might be desirable to get better performance (faster, smaller, less power-hungry systems) at expenses of some quality degradation.

Recently, a few papers have addressed this issue of designing imprecise hardware to save power [1, 2, 3, 4].

In this work, we introduce a systematic way of having imprecise arithmetic operations for the two most common operations: addition and multiplication. We liked the term *"sloppy"* introduced in [5], and we will use this term in the paper to refer to imprecise arithmetic operations.

2 Sloppy Addition

The idea is very simple. Do we need to propagate the carry for the whole word?

Assuming that we are operating on positive integers, and defining position k as the bit of weight 2^k in a n-bit word, we can ignore the carry up to position k when implementing the addition. The bit-level algorithm to implement this sloppy adder is the following:

```
c=0 // carry
if (i < k) then
  s_i = a_i XOR b_i;
else
  s_i = a_i XOR b_i XOR c;
  c = (a_i AND b_i) OR (a_i AND c) OR (b_i AND c);
endif
```

For example, addition $103 + 70$ ($n = 8$, $k = 4$):

```
       sloppy                 exact
A :    0110 0111 +         0110 0111 +
B :    0100 0110 +         0100 0110 +
c :    100- ---- =         0100 110- =
   --------------       --------------
S :    1010 0001           1010 1101
```

That is, the sloppy adder computes 161 (exact value is 173) introducing an error $\epsilon = 12$.

By looking at the bits of weight $< 2^k$, we notice that the XOR of two ones produces a zero sum bit ($1 \oplus 1 = 0$). Because the carry is not computed (or propagated), in position k an error 2^{k+1} is generated. The error can be halved to 2^k by computing the OR of the two bits in place of the XOR. For the example above we have:

```
       sloppy (OR-ing)
A :    0110 0111 +
B :    0100 0110 +
c :    100- ---- =
   --------------
S :    1010 0111
```

and the error is reduced from $\epsilon = 12$ to $\epsilon = 6$ (halved).

By simulating all possible combinations of the operands for the 8-bit addition ($k = 4$), we found that by obtaining the sum by OR-ing the k least-significant bits the average error is $\epsilon_{mean} = 3.75$, while by XOR-ing, it is $\epsilon_{mean} = 7.5$.

We show in Figure 1 the comparison of the hardware implementation of the sloppy adder used in the above example ($n = 8$, $k = 4$) and an error-free 8-bit carry-propagate adder (CPA). The data on delay, area and power are reported in Table 1.

In a rough evaluation, we considered lowering the supply voltage V_{DD} in the sloppy adder to match the delay of the error-free adder (1.0 ns). In our library, when V_{DD} is lowered from 1.0 V

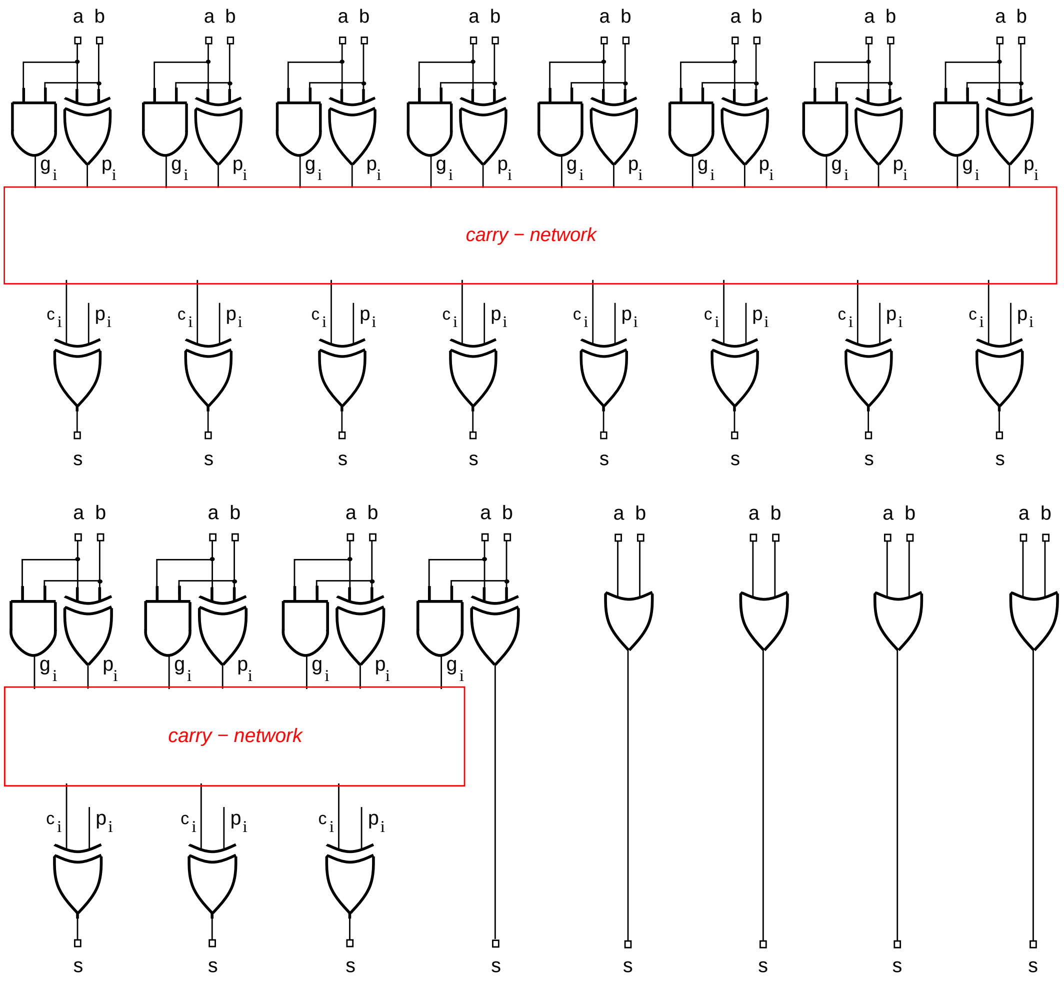

Figure 1: Implementation of 8-bit error-free (top) and sloppy $k = 4$ (bottom) adders.

to 0.7 V the delay doubles. Because the power dissipation is

$$P_{1.0V} = V_{DD}^2 f \cdot \sum^{N} a_i C_i \qquad \Rightarrow \qquad 20 = (1.0)^2 \cdot \mathcal{K}$$

we assume that the switching activity does not change when scaling V_{DD}. Therefore, $\mathcal{K} = 20$ is constant:

$$P_{0.7V} = (0.7)^2 \cdot 20 \simeq 10\ \mu W$$

That is, with the sloppy adder the power is reduced to 1/4 at same adder speed.

2.1 Example: sloppy adder in image filtering

We use the sloppy adder defined above ($k = 4$) to process two grayscale (each pixel is an unsigned 8-bit integer) images for the following bidimensional filters:

1. an averaging (low-pass) filter;
2. a sharpening filter;

	CPA 8-bit	sloppy	ratio
max. delay $[ps]$	999	495	2.00
Area $[\mu m^2]$	191	112	1.70
Power $[\mu W]$	42	20	2.10

Table 1: Synthesis data of adders in Figure 1.

	smoothing		*sharpening*		*edge det.*	
	ϵ_{max}	$\overline{\epsilon}$	ϵ_{max}	$\overline{\epsilon}$	ϵ_{max}	$\overline{\epsilon}$
uma	26	7.2	60	18.9	64	9.0
huse	28	7.8	59	17.5	68	9.2

Table 2: Error analysis of processed images.

3. an edge-detection unit.

The visual results are shown in Figure 2.

The maximum error (absolute value) ϵ_{max} and the average error $\overline{\epsilon}$ are reported in Table 2 for the different types of filtering. The results show that the degradation is independent of the image (**uma** is a portrait, while **huse** has greater detail). Depending on the filter mask, we can change the design of the sloppy adder to obtain larger savings. For example, for edge-detection, a sloppy adder with $k = 6$ has an average error $\overline{\epsilon} = 28$.

3 Sloppy Multiplication

Parallel multiplication $p = x \cdot y$ can be divided into three steps:

1. generation of Partial Products (PPs);
2. carry-free reduction from n PPs to 2 operands;
3. carry-propagate two operands addition.

We use a sloppy approach for step 1 only, as step 2 is quite delay-efficient (no carry propagation) and step 3 has been addressed in the previous section.

We consider radix-4 multiplication as for $n \times n$ bit operands $\frac{n}{2}$ PPs are generated and the unit is smaller. In radix-4 multiplication, the radix-4 digits of the multiplier y are recoded into signed-digit representation to avoid multiples of 3 and carry propagation as explained in [6]. The resulting architecture (for one digit) recoder plus PP generation (rec+PPgen) is sketched in Figure 3 (top).

Similarly to what was done for the addition, we have a sloppy rec+PPgen for the least-significant digits of y. The recoding is performed as shown in Table 3.

The resulting implementation is greatly simplified as shown in Figure 3 (bottom).

Clearly, a competitor of the sloppy multiplier is the truncated multiplier. To compare performance and error introduced, we implemented a 8×8-bit multiplier (two's complement) in the following schemes:

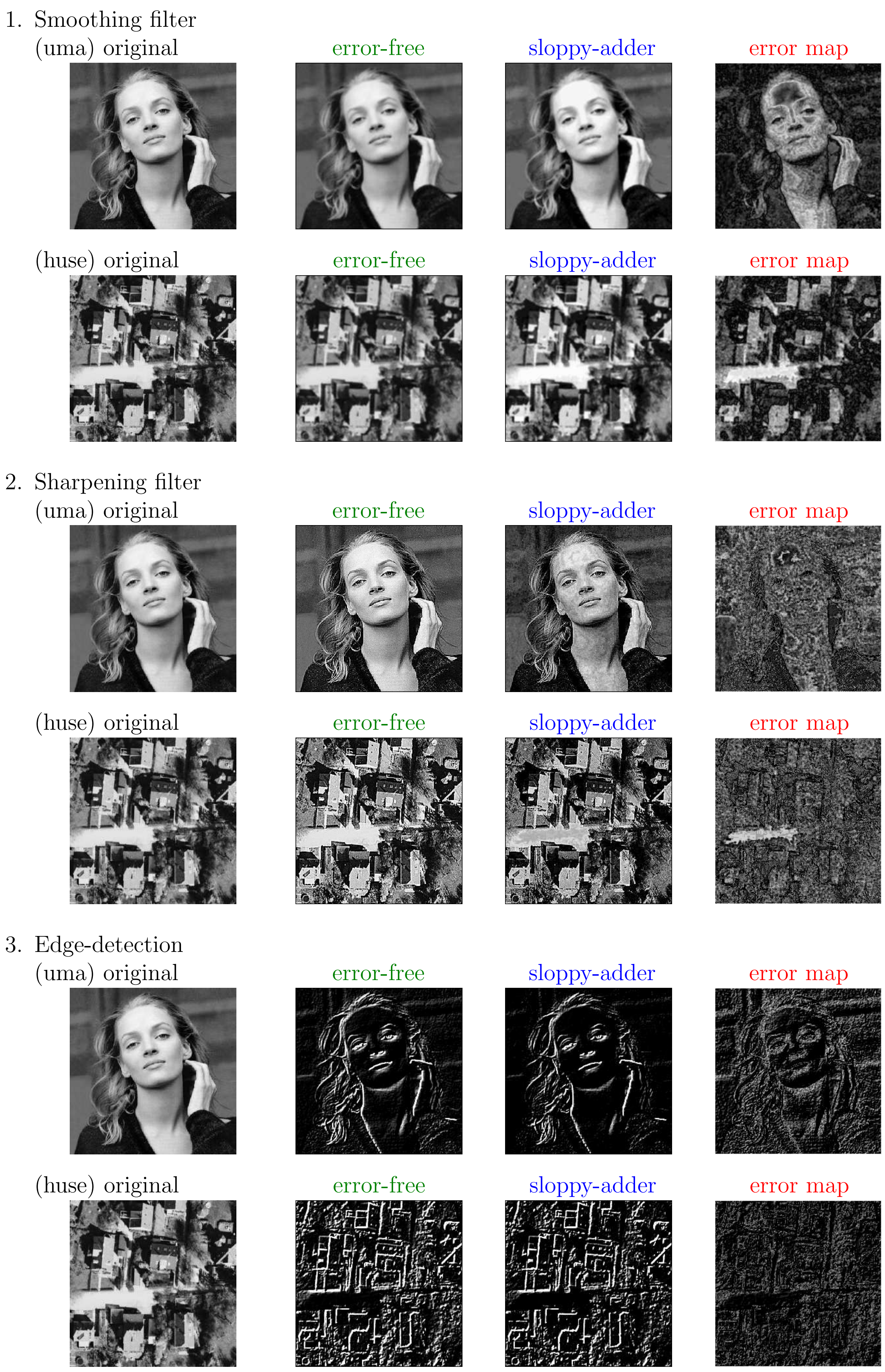

Figure 2: Visual result of sloppy addition in filtering.

		PP_k		
y_{2k+1}	y_{2k}	std.	sloppy	ϵ_k
0	0	0	0	0
0	1	$x \cdot 4^k$	$2x \cdot 4^k$	$x \cdot 4^k$
1	0	$2x \cdot 4^k$	$2x \cdot 4^k$	0
1	1	$3x \cdot 4^k$	$2x \cdot 4^k$	$-x \cdot 4^k$

Table 3: Sloppy radix-4 recoding.

unit	delay	power	area	error	
	$[ps]$	$[\mu W]$	$[\mu m^2]$	$\lvert\bar{\epsilon}\rvert$	$\lvert\epsilon_{max}\rvert$
r2-mult	900	70	2612	0	0
r4-mult	850	84	1842	0	0
r2-trunc	870	32	1426	256	897
r4-trunc	820	26	847	304	640
sloppy	490	21	1195	145	657

Table 4: Summary of result for 8×8-bit multiplier.

1. **r2-mult** a radix-2 standard multiplier;
2. **r4-mult** a radix-4 standard multiplier (with PPs generation as in Figure 3-top);
3. **r2-trunc** a r2-mult with k_t truncated bits;
4. **r4-trunc** a r4-mult with k_t truncated bits;
5. **sloppy** a radix-4 multiplier with PPs generation as in Figure 3-bottom for k_s digits.

We estimated a comparable error for $k_s = 2$ sloppy digits and $k_t = 8$ truncated bits. The results of the simulations on all 2^{16} combinations are reported in Table 4. The data do not include the contributions of the final carry-propagate adder.

4 Putting Everything Together

Now we combine the sloppy multiplier and adder in a multiply-add (and accumulate) unit (Figure 4) which can be used for the trivial implementation of the Inverse Discrete Cosine Transform (IDCT), which is part of the JPEG decompression algorithm.

We implemented the unit of Figure 4 with regular (R) and sloppy (S) operations as shown in Table 5. The multiplier is 12×12 bit, the adder is 24 bits. By C simulation, we found a sloppiness limit of $k_m = 3$ digits (6 bits) for the multiplier and $k_a = 8$ bits for the adder. The results in Table 5 are obtained by implementation in a 90 nm standard cells library (clock rate is 100 MHz). The errors are computed with respect to a floating-point software implementation. The visual results are shown in Figure 5.

The results show that the larger reduction in power is obtained when the sloppy multiplier is used. The contribution of the sloppy adder is little with respect to the power, but it is

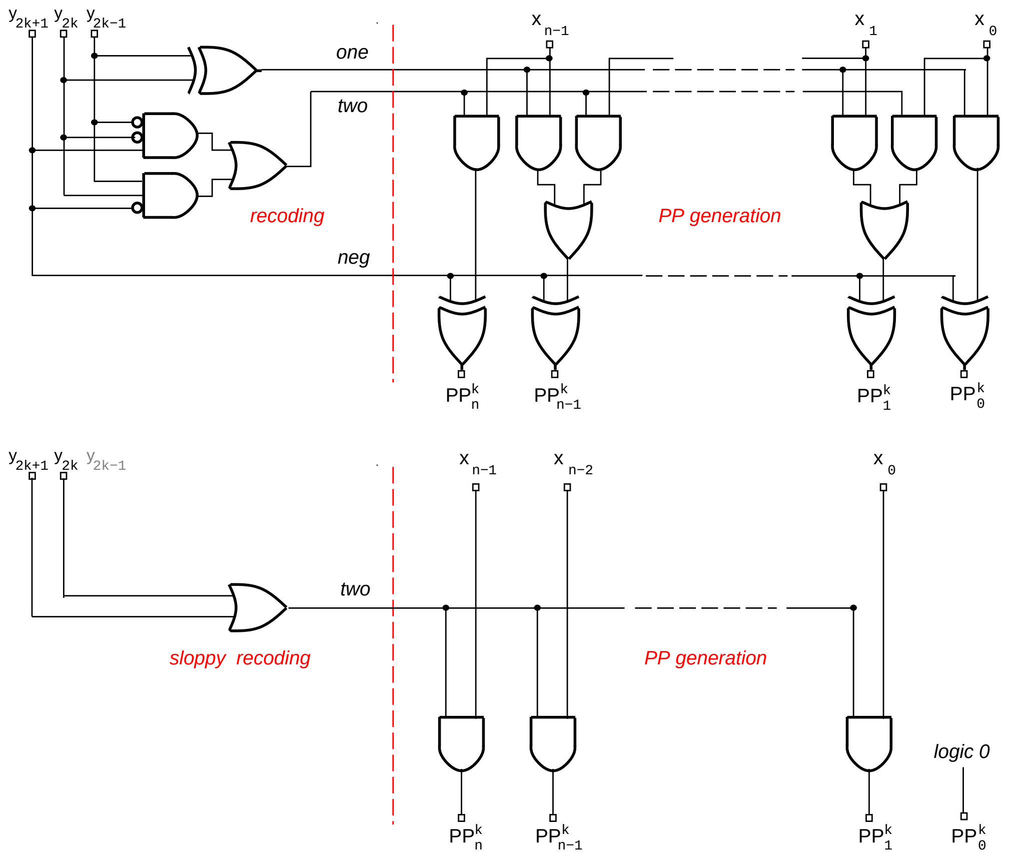

Figure 3: Implementation of error-free (top) and sloppy (bottom) rec+PPgen.

significant in delay reduction[1] (about 40% faster) and the slack can be used for low power design.

The degradation due to the sloppy adder, in addition to that of the sloppy multiplier, is marginal.

5 Conclusions and Future Work

We have presented simple ways of performing addition and multiplication in an imprecise manner with the aim to get better performance (delay, area and power) at expenses of an increased error which can be tolerated in some applications. This is preliminary work, just the idea, which is going to be further developed.

References

[1] K. He, A. Gerstlauer, and M. Orshansky, "Controlled Timing-Error Acceptance for Low Energy IDCT Design," *Proc. of 2011 Design, Automation and Test in Europe Conference*

[1]The synthesis was done with the minimum area constraint. Therefore, the adder is synthesized as a carry-ripple adder.

Unit		delay	area	**uma**			**huse**			power
MULT	ADD	$[ps]$	$[\mu m^2]$	P_{ave} $[\mu W]$	$\lvert\bar{\epsilon}\rvert$	$\lvert\epsilon_{max}\rvert$	P_{ave} $[\mu W]$	$\lvert\bar{\epsilon}\rvert$	$\lvert\epsilon_{max}\rvert$	ratio
R	R	3500	5580	128	3.7	9	185	3.8	10	1.00
S	R	3400	5090	107	5.0	34	155	6.0	39	0.84
R	S	3090	5440	125	3.8	18	181	5.0	21	0.98
S	S	2930	4950	106	5.0	35	153	6.6	36	0.83

Table 5: Summary of result for IDCT implementation.

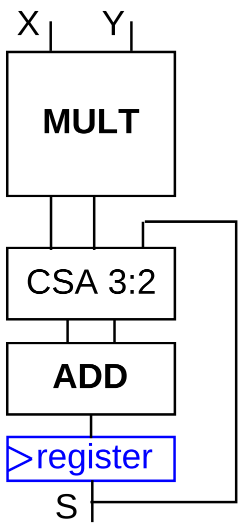

Figure 4: Scheme of multiply-accumulate used for IDCT.

(DATE), Mar. 2011.

[2] A. Lingamneni, J.-L. N. C. Enz, K. Palem, and C. Piguet, “Energy Parsimonious Circuit Design through Probabilistic Pruning,” *Proc. of 2011 Design, Automation and Test in Europe Conference (DATE)*, Mar. 2011.

[3] P. Krause and I. Polian, “Adaptive Voltage Over-Scaling for Resilient Applications,” *Proc. of 2011 Design, Automation and Test in Europe Conference (DATE)*, Mar. 2011.

[4] D. Mohapatra, V. Chippa, A. Raghunathan, and K. Roy, “Design of Voltage-Scalable Meta Functions for Approximate Computing,” *Proc. of 2011 Design, Automation and Test in Europe Conference (DATE)*, Mar. 2011.

[5] L. Hardesty. ”The surprising usefulness of sloppy arithmetic”. MIT News Office. [Online]. Available: http://web.mit.edu/newsoffice/2010/fuzzy-logic-0103.html

[6] M. Ercegovac and T. Lang, *Digital Arithmetic.* Morgan Kaufmann Publishers, 2004.

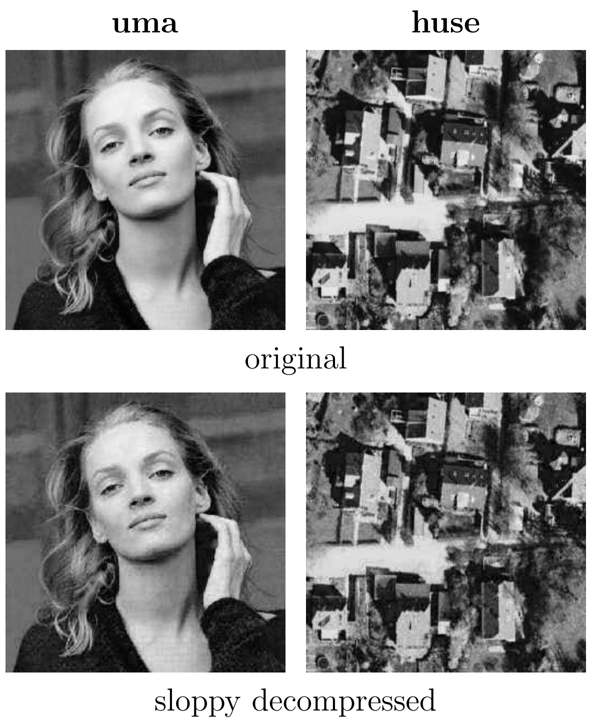

Figure 5: Original pictures (top) and after decoding by sloppy (S-S) IDCT (bottom).