ECSE 682 VLSI Signal Processing Implementation Report: The FastICA Algorithm

Loren Lugosch (260404057)
Department of Electrical and Computer Engineering
McGill University
Montreal, Quebec, Canada
loren.lugosch@mail.mcgill.ca

***Abstract*—This report describes my implementation of the FastICA one-unit algorithm.**

***Index Terms*—unsupervised learning, BSS, HLS, VLSI signal processing.**

I. Introduction

A. Problem statement

Signals available for processing often do not represent a phenomenon of interest directly, but rather represent the phenomenon along with some interference from some other signal sources. Not removing this interference can reduce the quality of subsequent processing operations.

For example, a microphone which is intended to record the sound of a guitar in a band will also pick up the sounds of the other musical instruments in the band. An audio engineer might want to make the guitar louder, but the recording from the guitar microphone will have sound components from the other instruments, so making that track louder will make the other instruments louder as well. This problem is called the "blind source separation" (BSS) problem.

B. The ICA model

The Independent Component Analysis (ICA) model is an approach to the BSS problem [1]. In ICA, we assume that the signals we have received, **x**(t), are a linear combination of the source signals, **s**(t), where **x** and **s** are n-dimensional random variables.

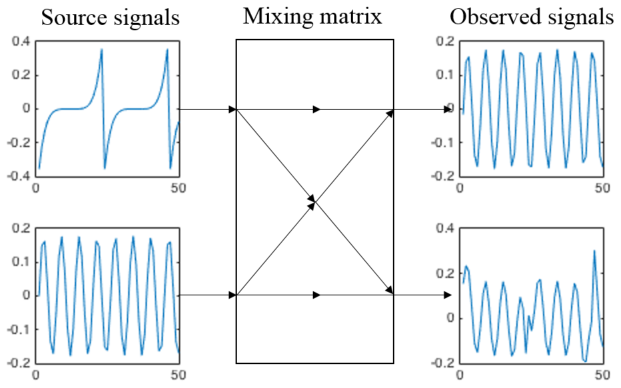

Fig. 1: Source signal mixing in the ICA model

The linear combinations resulting in each observed signal can be thought of together as a matrix multiplication by some mixing matrix, **A**. Figure 1 depicts the ICA mixing process.

Since matrix multiplications are invertible operations, if the first ICA assumption holds, we can recover **s** by finding an "unmixing matrix", **W**, and simply multiplying **x** by this unmixing matrix to obtain a set of estimated signals, $\mathbf{s}_{est}$.

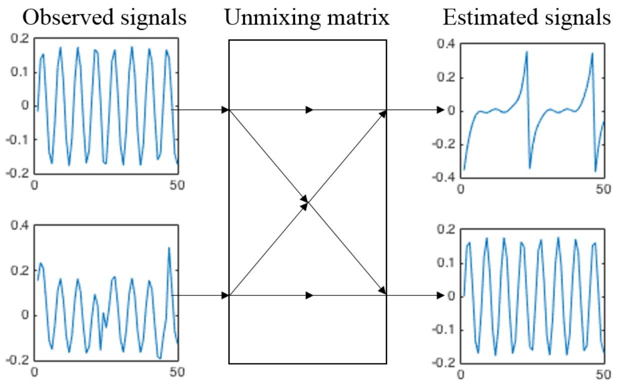

Fig. 2: Unmixing the observed signals to produce estimated signals

The second assumption of ICA is that the components of **s** are independent random variables with non-Gaussian distributions. From the Central Limit Theorem in probability, we know that the sum of two non-Gaussian random variables is more Gaussian than either of those random variables. Thus, when the mixing matrix adds source signals together, if those source signals are non-Gaussian, then the observed signals will be more Gaussian than the sources. Therefore, multiplying the observed signals by a good unmixing matrix will produce estimated signals which are more non-Gaussian than the observed signals.

C. The FastICA algorithm

FastICA is an efficient algorithm for solving a BSS problem in real-time using the ICA model [1]. The algorithm consists in repeated applications of the "one-unit" computation, which produces one vector (**w**) of the unmixing matrix **W**. The one-unit computation is composed of two stages: a preprocessing stage and an iterative stage.

1) Preprocessing stage: In this stage, the algorithm centers and whitens the input signals (i.e., makes them zero-mean and identity-variance). (For my implementation, I assume that the signals have already been centered and whitened.)

2) Iterative stage: In this stage, the algorithm performs four steps (which I have taken verbatim from [1]) to find an unmixing vector which maximizes the non-Gaussianness of the resulting estimated signals:

1) Choose an initial (e.g. random) weight vector **w**
2) Let $\mathbf{w}^+$ = E{**x**g($\mathbf{w}^T\mathbf{x}$)} - E{g'($\mathbf{w}^T\mathbf{x}$)}**w**
3) Let **w** = $\mathbf{w}^+$ / $||\mathbf{w}^+||$
4) If not converged, go back to 2.

where E{} represents the expected value of a random variable, and g() and g'() represent some non-Gaussianness function and its derivative, respectively. Step 2 is a vector rotation, and step 3 is a vector normalization.

D. Related work

(For related work on implementing FastICA, see my Literature Review.)

E. Structure of the report

For my project, I implemented the iterative stage of the FastICA one-unit computation. Section II describes modifications to the algorithm to make it more suitable for real-time implementation. Section III describes rapid prototyping of different architectural ideas using the LegUp HLS tool. Section IV describes my handwritten HDL implementation. In Section V, I present the results of my design space exploration. Finally, in Section VI, I discuss some future work.

(See README for instructions on how to compile my code and run my tests.)

II. Algorithmic exploration

The first step towards hardware implementation is writing out the single assignment form of the algorithm (Algorithm 1) and experimenting with changes to the algorithm.

A. Dependence graph analysis

We first need to expand the E{} operator in the FastICA pseudocode. Because we don't have access to the expected value of the product of two signals a priori, we estimate it by calculating the product and averaging together all the samples. The more observed signal samples the algorithm uses, the better estimates it can make of the signal statistics, but the longer the duration of a single iteration. N is the number of channels in the input signals, and T is the number of samples taken.

The single assignment algorithm can be translated into a dependence graph (Figure 3). For my project, I used N = 2 and T = 64. However, for a more intelligible dependence graph, I use N = 3 and T = 4.

The dependence graph can be decomposed into three systolic arrays: the upper half (the rotation stage), the left side of the lower half (the normalization stage), and the right side of the lower half, boxed in red (the convergence check).

Algorithm 1 Single assignment FastICA

```
1: function (ws)        ▷ ws: T N-dimensional whitened observations
2:     $w \leftarrow <$ random vector $>$
3:     $w_{next} \leftarrow <$ random vector $>$
4:     converged ← False
5:     while not converged do
6:         for $t \leftarrow 1, T$ do
7:             $p1[t][0] \leftarrow 0$
8:             for $n \leftarrow 1, N$ do
9:                 $p1[t][n] \leftarrow p1[t][n-1] + w[n] * ws[n][t]$
10:            end for
11:        end for
12:        for $t \leftarrow 1, T$ do
13:            $tp[t] \leftarrow (g(p1[t][N]))/T$
14:        end for
15:        for $n \leftarrow 1, N$ do
16:            $p2[n][0] \leftarrow 0$
17:            for $t \leftarrow 1, T$ do
18:                $p2[n][t] \leftarrow p2[n][t-1] + ws[n][t] * tp[t]$
19:            end for
20:        end for
21:        $s1[0] \leftarrow 0$
22:        for $t \leftarrow 1, T$ do
23:            $sp[t] \leftarrow (g'(p1[t][N]))/T$
24:            $s1[t] \leftarrow s1[t-1] + sp[t]$
25:        end for
26:        $s2[0] \leftarrow 0$
27:        for $n \leftarrow 1, N$ do
28:            $p3[n] \leftarrow w[n] * s1[T]$
29:            $diff[n] \leftarrow p2[n][T] - p3[n]$
30:            $s2[n] \leftarrow s2[n-1] + diff[n] * diff[n]$
31:        end for
32:        $w\_rnorm \leftarrow rsqrt(s2[N])$
33:        for $n \leftarrow 1, N$ do
34:            $w_{next}[n] \leftarrow diff[n] * w\_rnorm$
35:        end for
36:        $dot[0] \leftarrow 0$
37:        for $n \leftarrow 1, N$ do
38:            $dot[n] \leftarrow dot[n-1] + w[n] * w_{next}[n]$
39:        end for
40:        if $abs(dot[N]) - 1 < \epsilon$ then
41:            converged ← True
42:        end if
43:    end while
44:    return $w$        ▷ w: N-dimensional unmixing vector
45: end function
```

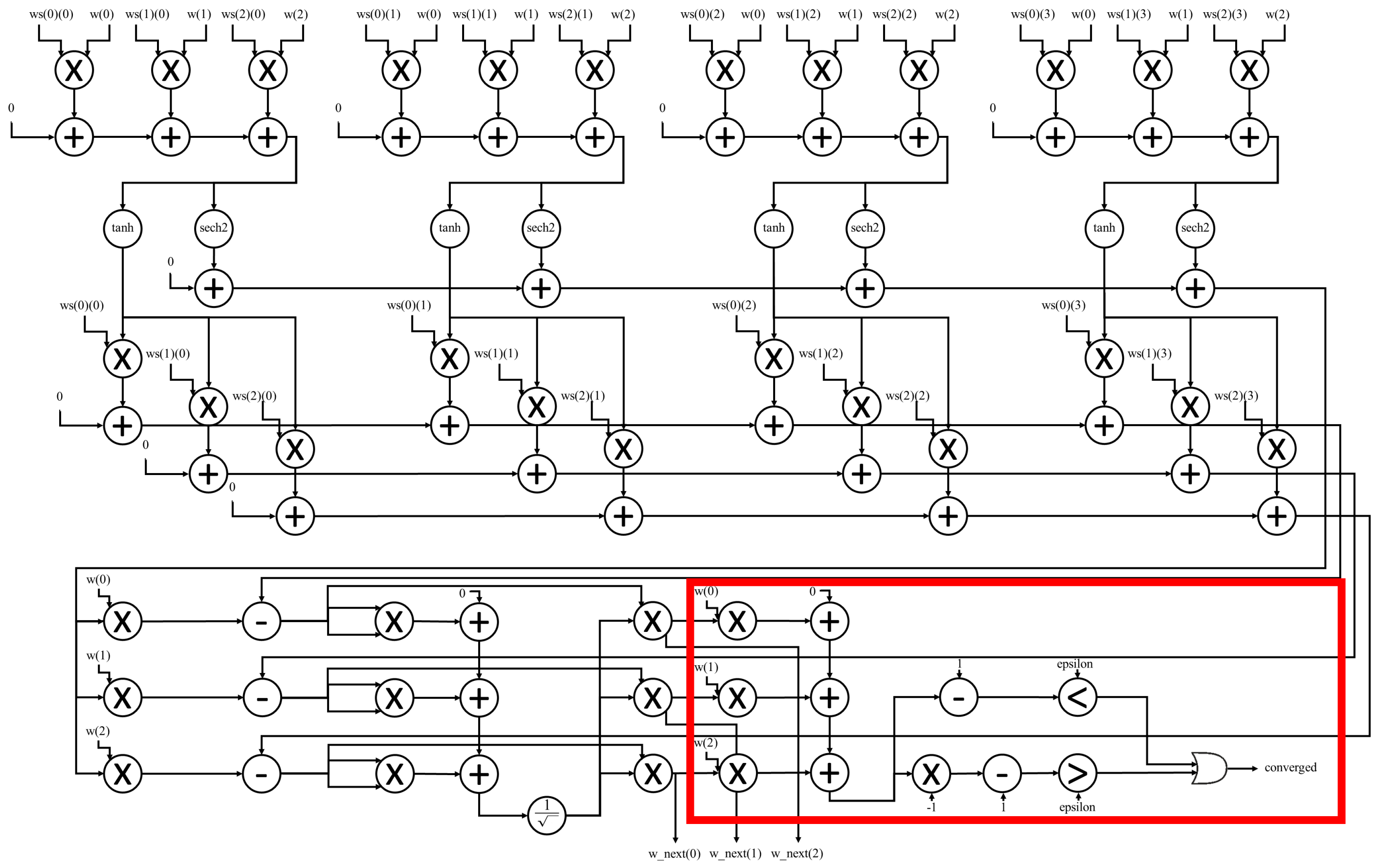

Fig. 3: FastICA dependence graph for N = 3, T = 4 (convergence check highlighted in red)

The normalization stage produces the next version of the unmixing vector, which feeds back into the rotation stage during the next iteration. Because this feedback does not depend on the convergence check, we can perform a few interesting optimizations, which I will discuss later.

B. No random number generation

I found while trying different inputs to the algorithm that a single initial vector was adequate for any mixing matrix and any set of source signals. By using a fixed vector, I avoid the latency and area incurred by performing random number generation before running the iterative algorithm.

C. Non-Gaussianness functions

The authors of [1] describe a few non-Gaussianness functions which will work for FastICA. However, I decided to use hyperbolic tangent (tanh()) because all of the papers I read for this project used that function, and I wanted to be able to compare my designs with those in the literature.

The derivative of tanh() is hyperbolic secant squared (sech^2()). LegUp is able to synthesize hyperbolic tangent and hyperbolic secant from the C math library. However, the C math implementation provides much more precision than our application needs. Furthermore, verifying hardware which uses

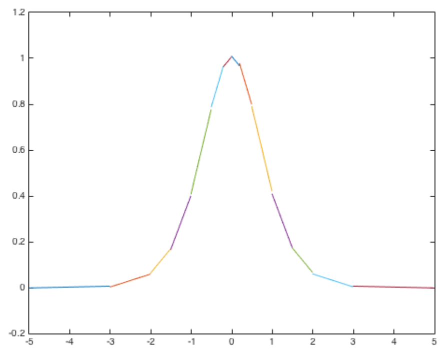

Fig. 4: Piecewise linear approximation of hyperbolic secant squared (each segment colored differently)

these functions in Modelsim is prohibitively slow, with a single iteration of the FastICA loop taking 3 hours to simulate. Instead of the C math implementations, I use a piecewise linear approximation for these functions. A piecewise linear approximation uses one add, one multiply, one multiplexer, and one decoder to figure out which constants to select. I take the approximation for tanh() from [2]. Using linear regression in Matlab, I created my own approximation for $\mathrm{sech}^2()$ (Figure 4).

D. Normalizing the rotation vector

The normalization step requires taking the square root of the next unmixing vector. If the LegUp compiler sees “sqrtf()” in the code, it will synthesize an Altera floating-point square root. The square root (ALTFP_SQRT) IP core, like the Taylor series implementations of tanh() and $\mathrm{sech}^2()$, is prohibitively slow to simulate and requires an enormous amount of FPGA resources [3]. I created another piecewise approximation which can be used in both the fixed-point and floating-point implementations. Instead of square root, I implemented reciprocal square root (1/sqrt() or “rsqrt()”), thus replacing the subsequent division with a multiplication. Because the convergence check needs a relatively precise square root, I had to increase epsilon (the convergence constant) from 0.000001 to 0.02.

E. Dithering

Dithering is a technique used in audio and image processing to reduce the effect of quantization noise resulting from data compression. By adding noise to the signal before quantizing, the quantization error is “spread” over the signal more evenly (Figure 5).

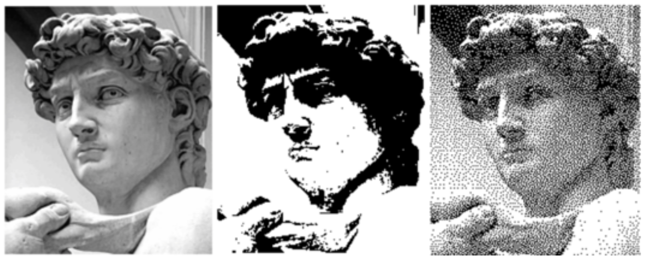
Fig. 5: The latter two images use only 1 bit per pixel, but the image on the right looks better than the middle image because it was created using dithering. (Image source: [4])

I experimented with using dithered constants for the hyperbolic approximations instead of multiply-adds and was able to maintain a good signal-to-interference-ratio (SIR, or how strongly the other source components are present in the estimated signal corresponding to a given source) in the final output. However, to be able to compare my architecture with the existing implementations of FastICA more easily, I chose to leave dithering as future work and use the multiply-add approximations.

III. Architectural exploration

After finalizing the exact form of the algorithm to be implemented (FastICA iterative stage, N = 2, T = 64, linear piecewise approximations for tanh(), $\mathrm{sech}^2()$, and rsqrt()), I explored different mappings for the dependence graph of Figure 3 using the LegUp high-level synthesis (HLS) tool.

A. Parallel processing

The OpenMP parallel-for directive hints LegUp that iterations of a for-loop can be parallelized. Using parallel-for gave great results for loops over n. For any amount of parallelism (i.e. parallelizing one loop, two loops, etc.), the latency was lower (and the area was higher) than for the baseline fixed-point implementation.

One of my Pareto-optimal points comes from parallelizing a block with operations over n and t (line 15). When I added a second (n,t) block (line 6), the number of clock cycles decreased yet again, so the design was still faster than the fixed-point baseline. However, the clock frequency decreased, and the resulting latency was worse, so the design with two (n,t) blocks was dominated by the design by the design with only one (n,t) block. I found it interesting that the latency started to increase at this point. It may be that the extra parallelism causes extra fan-out, which reduces the clock frequency.

The hyperbolic tangent and secant calculations on every sample product (lines 12 and 23 in Algorithm 1) are completely independent, so they can theoretically be parallelized. However, parallelizing these loops causes too much hardware to be generated, and the Quartus compile fails. I tried varying the number of threads between 2 and T, thinking that T parallel tanh() and $\mathrm{sech}^2()$ blocks would be too much hardware, but even “num_threads(2)” resulted in more ALMs than the target FPGA could support. This would seem to be a bug in LegUp since the hardware required for 2 of these blocks should be much smaller than that required for T blocks.

B. Reduction

An associative operation (such as addition, multiplication, maximum, and minimum) on a group of operands can be performed in any order, as long as the number of operations stays the same. By shuffling around associative operations so that as many occur as can fit into the available hardware, we can decrease the length of the critical path. This technique is called “reduction”. Figure 6 shows an example of a reduction sum.

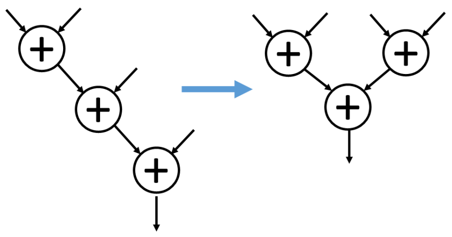

Fig. 6: Reduction of addition of four operands

Any code of the form:

```
for(i = 0; i++; i < N) {
    result (ASSOCIATIVE OP) a[i];
}
```

can be reduced. Thus, lines 18, 24, 30, and 38 in the algorithm are all candidates for reduction, as they involve summing over an array.

LegUp supports reduction by using the OpenMP reduction directive as a compiler hint. However, I was unable to produce a working design with reduction- LegUp support for reduction is incomplete. Reduction on integer types often produces incorrect results (reducing line 24 caused a single print statement elsewhere in the code to print thousands of times and made the simulation last overnight, suggesting that more of the synthesized HDL is being duplicated than ought to be; reducing the other lines produced results which were different from those from the C code compiled with the OpenMP option). Reduction on floating-point operations simply doesn't compile.

C. Loop pipelining

Because the rotation step at the start of the iterative loop does not depend on the convergence check at the end of the previous loop, the outer loop as a whole is theoretically loop-pipelineable. However, loop pipelining either the outer loop or one of the the inner loops results in the same number of clock cycles, suggesting to me the LegUp didn't actually manage to build the pipelines, and that the increase in latency is simply due to the "LOCAL_RAMS" parameter.

D. Loop unrolling

The final technique I explored was loop unrolling. First, I tried unrolling every loop in my code. Quartus was unable to synthesize the design. This also seems to be a bug: if the operations are schedulable as part of a loop, they should also be schedulable when unrolled!

Next, I tried unrolling every loop over n. This design actually dominated the parallel-for over n only (but not over the (n,t) blocks), which was unexpected.

Fully unrolling over t also produced too much hardware, so instead I unrolled the loop 8 times. This design was slightly faster than the baseline, but uses more hardware than any other design.

E. Summary of LegUp results

Tables I and II show the results of synthesis for my LegUp designs. I targeted the Cyclone V 5CSEMA5F31C6 FPGA.

TABLE I: LegUp floating-point designs

Design	F_{max}	CCs	ALMs	Regs	Mem.	DSPs
(Baseline)	127.26	60697	2729	4019	2048	1
Parallel-for n	115.13	61068	2817	4171	2176	1
Reduction p2	n/a	n/a	n/a	n/a	n/a	n/a
Reduction s1	n/a	n/a	n/a	n/a	n/a	n/a
Parallel-for t	n/a	n/a	n/a	n/a	n/a	n/a
Loop pipelining	107.57	65577	2725	4354	2304	1
Fully unrolled	n/a	n/a	n/a	n/a	n/a	n/a
Unrolled n	106.89	66909	2989	4457	2176	1
Unrolled t (x8)	101.1	65061	4329	5500	4352	1

TABLE II: LegUp fixed-point designs

Design	F_{max}	CCs	ALMs	Regs	Mem.	DSPs
(Baseline)	99.21	12822	1443	1933	2048	38
Parallel-for n	108.6	11936	3545	4648	2304	72
Reduction p2	n/a	n/a	n/a	n/a	n/a	n/a
Reduction s1	n/a	n/a	n/a	n/a	n/a	n/a
Parallel-for t	n/a	n/a	n/a	n/a	n/a	n/a
Parallel-for n, t	105.61	8404	3825	5169	2304	76
Parallel n, t (x2)	92.91	7940	4922	6659	2304	87
Loop pipelining	113.62	15115	1958	2403	4268	25
Fully unrolled	n/a	n/a	n/a	n/a	n/a	n/a
Unrolled n	112.96	11072	1775	2343	2048	46
Unrolled t (x8)	96.1	12190	6417	5108	4160	68

IV. Handwritten HDL implementation

A. Mapping the dependence graph

Mapping the dependence graph directly to hardware produces an impractical design with low hardware utilization. Instead of a direct mapping, I mapped the first systolic array in the dependence graph (the rotation stage) to a rotation unit for one set of signal samples (Figure 7). The normalization unit and convergence unit are directly mapped from the dependence graph. Each stage is pipelined. A "start" token (green) and "stop" token (blue) are passed through the pipeline to initialize the accumulations to 0 and allow feedback only once valid operands reach the lower adders and multipliers.

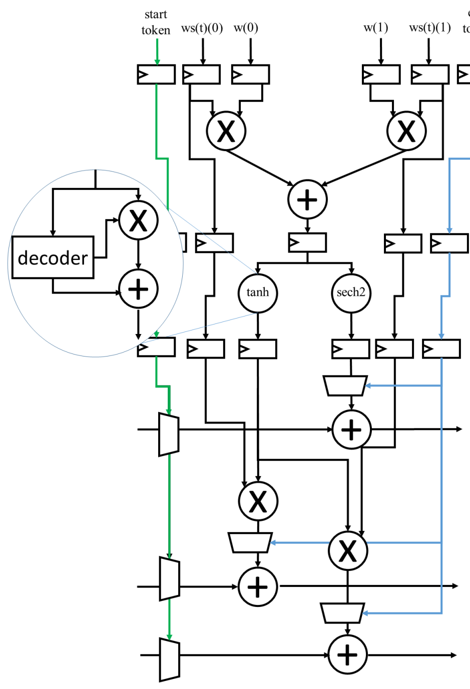

Fig. 7: Pipelined rotation unit

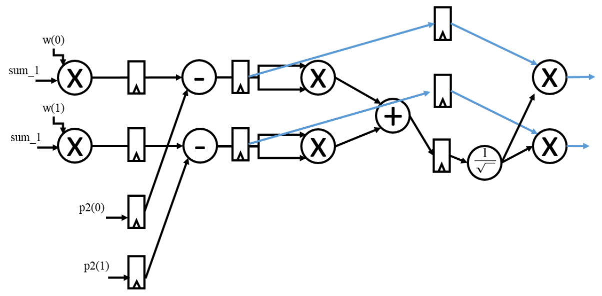

Fig. 8: Pipelined normalization unit

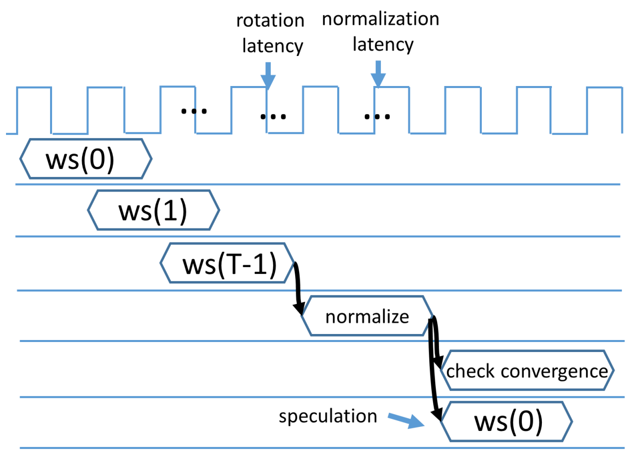

Fig. 9: Schedule of whitened signals (ws) entering the pipeline, followed by normalization and convergence checking

B. Scheduling with speculation

The next iteration is "speculative", i.e. it begins before the convergence check has a chance to complete, saving as many clock cycles per iteration as there are pipeline stages in the convergence check.

Table III shows the metrics for my handwritten VHDL design on the Cyclone V 5CSEMA5F31C6.

TABLE III: VHDL design

F_{max}	CCs	ALMs	Regs	Mem.	DSPs
72.6 MHz	286	669	768	2048	27

V. Results

I devised two metrics to evaluate my designs: "latency" and "area". The latency score represents the number of microseconds between first input and final output for a given implementation. Latency is defined in Equation 1:

$$\text{Latency} = \frac{\text{\# of clock cycles for completion}}{\text{maximum operating frequency } (F_{max})} \quad (1)$$

The area score is somewhat more heuristic: it represents roughly how much of the FPGA is used by a design. The area score is defined in Equation 2:

$$\text{Area} = \text{\# ALMs} + \text{\# registers} + \text{\# mem. bits} + \text{\# DSPs} \quad (2)$$

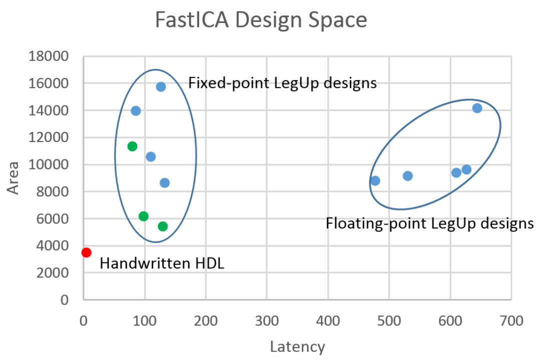

Fig. 10: Scatter plot of designs by latency and area

Figure 10 shows a scatter plot of the valid design points' latency and area scores. The green points represent the Pareto-optimal designs from LegUp, namely 1) the baseline fixed-point implementation, 2) the implementation with n unrolled, and 3) the implementation with a single parallel-for (n, t) block. Unsurprisingly, every floating-point implementation is dominated. The red point in the graph represents my design. My design has a latency of 4 microseconds and an area score of 3512, dominating every design produced by LegUp. (Note that although 4 microseconds is much faster than any of the implementations described in the Literature Review, my design only performs the one-unit computation, and not the centering and whitening or decorrelation.)

VI. Conclusion and Future Work

This report described the implementation of FastICA using LegUp HLS and VHDL. Future work might include adding the preprocessing stage, dithering, and estimating a number of components not equal to the number of received signal channels.

References

[1] A. Hyvärinen and E. Oja, "Independent component analysis: algorithms and applications," *Neural networks*, vol. 13, no. 4, pp. 411–430, 2000.

[2] C.-H. Yang, Y.-H. Shih, and H. Chiueh, "An 81.6 uw fastica processor for epileptic seizure detection," *Biomedical Circuits and Systems, IEEE Transactions on*, vol. 9, no. 1, pp. 60–71, Feb 2015.

[3] altera.com, "Floating-point ip cores user guide," 2015. [Online]. Available: https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/ug/ug_altfp_mfug.pdf

[4] Verypdf.com, "Verypdf ocr user guide," 2015. [Online]. Available: http://www.verypdf.com/app/ocr-to-any-converter-cmd/user-guide.html