

Hydra: An energy-efficient programmable cryptographic coprocessor supporting elliptic-curve pairings over fields of large characteristics

No Institute Given

Abstract. Bilinear pairings on elliptic curves have many applications in cryptography and cryptanalysis. Pairing computation is more complicated compared to that of other popular public-key cryptosystems. Efficient implementation of cryptographic pairing, both software- and hardware-based approaches, has thus received increasing interest. In this paper, we focus on hardware implementation and present the design of Hydra, an energy-efficient programmable cryptographic coprocessor that supports various pairings over fields of large characteristics. We also present several implementations of Hydra, among which the smallest only uses 116 K gates when synthesized in TSMC 90 nm standard cell library. Despite the extra programmability, our design is competitive compared even with specialized implementations in terms of time-area-cycle product, a common figure of merit that provides a good measure of energy efficiency. For example, it only takes 3.04 ms to compute an optimal ate pairing over Barreto-Naehrig curves when the chip operates at 200 MHz. To the best of our knowledge, our implementation has the smallest time-area-cycle product among all hardware implementations of cryptographic pairing in the literature.

1 Introduction

Bilinear pairings on elliptic curves were introduced in the middle of 1990's for cryptanalytic purposes, e.g., to break cryptographic protocols whose security is based on the hardness of the elliptic-curve discrete-logarithm problem [14, 9]. In 2000, Joux showed that they can also be used for tripartite key agreement [12]. Since then, many constructive pairing-based schemes were proposed, such as identity-based encryption [2], identity-based signatures [4], as well as short signatures [3].

Pairing computation is much more complicated compared to that of other popular public-key cryptosystems. Efficient implementation of cryptographic pairing has thus received increasing interest, both software- and hardware-based approaches, pursuing higher speed or, in the cases of hardware implementation, smaller time-area-cycle product (ATC product), a common figure of merit that provides a good measure of energy efficiency.

For example, Kammler *et al.* reported a hardware implementation of a cryptographic coprocessor for various pairings of 128-bit security [13]. Although this is not the first programmable architecture, it is the first one that supports 128-bit security level. Fan, Vercauteren, and Verbauwhe improved the multiplication algorithm for Barreto-Naehrig (BN) curves with special parameters to achieve a higher speed [8]. However,

as they took advantage of the chosen parameters, the result is only applicable to BN curves.

For field-programmable gate arrays (FPGAs), Ghosh, Mukhopadhyay, and Roychowdhury exploited the special arithmetic units in some FPGAs and gave the first FPGA pairing implementation with 128-bit security level [10]. In their proposed architecture, multiple homogeneous arithmetic units are used in parallel to exploit the inherent parallelism in pairing computation. Recently, Cheung *et al.* experimented with Montgomery multiplication in residue number systems [5], and Ghosh, Roychowdhury, and Das explored η_T pairing over binary curves [11]. Both implementations achieved good results with respect to speed.

In this paper, we will present the design and implementation of Hydra, an energy-efficient programmable cryptographic coprocessor that supports various pairings at 128-bit security level. Unlike the general architecture of, e.g., Kammler *et al.* [13], our design is optimized for carrying out pairing computation over fields of large characteristics. As a result, our design stays competitive in terms of ATC product even compared with specialized implementations such as that of Fan, Vercauteren, and Verbauwhede [8]. For example, our smallest implementation of Hydra only uses 116 K gates when synthesized in TSMC 90 nm standard cell library and has a latency of 3.04 ms when running at 200 MHz. To the best of our knowledge, our implementation has the smallest time-area-cycle product among all hardware implementations of cryptographic pairing in the literature.

The organization of the rest of this paper is as follows. In Section 2, we will first give a brief introduction cryptographic pairings for the subsequent exposition. We will present the main ideas behind Hydra’s architectural design in Section 3. We will then go through the detailed design of datapath in Section 4 and control in Section 5. To conclude, we will compare our implementation results with the state of the arts in Section 6.

2 Background

2.1 Bilinear pairings

Let G_1 , G_2 , and G_T be abelian groups. A bilinear pairing is a map $\alpha : G_1 \times G_2 \rightarrow G_T$ with the following properties.

1. Bilinearity: $\alpha(mP, Q) = \alpha(P, mQ) = \alpha(P, Q)^m$ for $m \in \mathbb{Z}$.
2. Non-degeneracy: For all nonzero $P \in G_1$, there exists $Q \in G_2$ such that $\alpha(P, Q) \neq 1$, and vice versa.
3. Computability: $\alpha(P, Q)$ can be efficiently computed.

Most cryptographic pairings work on elliptic curves. There are many choices for the map α as well as the curves. In this work, we use Barreto-Naehrig (BN) curves [1] and optimal ate pairing [19] as an example, but other pairings can also be accelerated by our Hydra coprocessor.

2.2 Barreto-Naehrig curves

Barreto-Naehrig curves are pairing-friendly elliptic curves over prime field \mathbb{F}_p with embedding degree $k = 12$ [1]. They are defined by the equation $E : y^2 = x^3 + b, b \neq 0$. Let n denote the group order of $E(\mathbb{F}_p)$. Then p and n can be parameterized as

$$\begin{aligned} p(u) &= 36x^4 - 36x^3 + 24x^2 - 6x + 1 \text{ and} \\ n(u) &= 36x^4 - 36x^3 + 18x^2 - 6x + 1, \end{aligned}$$

where $u \in \mathbb{Z}$ is an integer such that p and n are both prime numbers. We follow the work of Kammler *et al.* [13] and choose $b = 24$ and $u = 0 \times 60000000000001F2D$. This yields a key security parameter p of 256 bits.

2.3 Computing optimal ate pairing

Algorithm 1 shows how to compute an optimal ate pairing. We base our design on

Algorithm 1 Optimal ate pairing over Barreto-Naehrig curves

Require: $P \in G_1, Q \in G_2, s = 6t + 2 = \sum_{i=0}^{L-1} s_i 2^i, s_i \in \{0, 1\}$

Ensure: $\alpha_{opt}(P, Q)$

```

1:  $T \leftarrow Q; f \leftarrow 1;$ 
2: for  $i = L - 2$  to  $0$  do
3:    $f \leftarrow f^2 \cdot l_{T,T}(P); T \leftarrow 2T;$ 
4:   if  $s_i == 1$  then
5:      $f \leftarrow f \cdot l_{T,Q}; T \leftarrow T + Q$ 
6:   end if
7: end for
8:  $Q_1 \leftarrow \pi_p(Q); Q_2 \leftarrow \pi_{p^2}(Q);$ 
9:  $f \leftarrow f \cdot l_{T,Q_1}(P); T \leftarrow T + Q_1$ 
10:  $f \leftarrow f \cdot l_{T,-Q_2}(P); T \leftarrow T - Q_2$ 
11:  $f \leftarrow f^{\frac{(p^{12}-1)}{r}};$ 
12: return  $f$ 
```

Schwabe's high-quality software implementation [18] with further optimization on register usage.

The first part of Algorithm 1 is the Miller loop, first proposed by Miller [15] and later enhanced by others. It is a standard double-and-add loop performing the elliptic-curve arithmetic, including line addition and line doubling. After each line function, an $\mathbb{F}_{p^{12}}$ multiplication is performed.

The second part is the final exponentiation, which takes the output of the Miller loop $f \in \mathbb{F}_{p^{12}}$ and computes $f^{\frac{(p^{12}-1)}{r}}$. Here we use the method of Devegili, Scott, and Dahab [7] and split $\frac{p^{12}-1}{r}$ to $(p^6 - 1)(p^2 + 1)(\frac{p^4 - p^2 + 1}{l})$ to speed up the computation.

Overall, the main computations are those of the line functions and $\mathbb{F}_{p^{12}}$ operations. We construct $\mathbb{F}_{p^{12}}$ as a series of tower fields as follows.

$$\begin{array}{c}
\mathbb{F}_{p^{12}} = \mathbb{F}_{p^6}[Z]/(Z^2 - \tau) \text{ with } \tau = Y \\
\uparrow \\
\mathbb{F}_{p^6} = \mathbb{F}_{p^2}[Y]/(Y^3 - \xi) \text{ with } \xi = (X + 1) \\
\uparrow \\
\mathbb{F}_{p^2} = \mathbb{F}_p[X]/(X^2 - \sigma) \text{ with } \sigma = -2 \\
\uparrow \\
\mathbb{F}_p
\end{array}$$

In this case, an $\mathbb{F}_{p^{12}}$ operation can be implemented by a series of \mathbb{F}_p operations. Furthermore, the elliptic curve $E(\mathbb{F}_p)$ itself lives in \mathbb{F}_p . As a result, all computations can be decomposed into a set of \mathbb{F}_p multiplications and additions. As we will see subsequently, we exploit this fact to get a better speed at a moderate price in terms of control logic.

3 High-level architectural design

Fig. 1 shows the the five major blocks of the Hydra architecture, namely, Axy Engine, Data Cache, Decoder, Instruction Cache, and Top Control. The main design philosophy

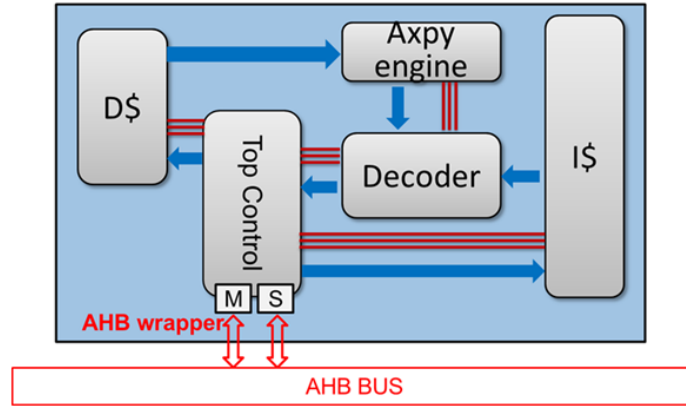


Fig. 1. High-level architectural design of Hydra

behind Hydra's architecture is the separation of data and control, a key enabler for both programmability and high-throughput data processing. As shown in Fig. 1, data flows from Data Cache to Axy Engine, the main computational unit of Hydra, and gets written back to Data Cache via Decoder and Top Control after being processed, as some of the results may be used by the latter two blocks for control purposes. On the other hand,

control information is mainly passed from Decoder to Data Cache via Top Control after instructions are fetched from Instruction Cache and decoded at Decoder. Moreover, Top Control also handles the input and output of the entire coprocessor, interfacing with the host processor via the AMBA (Advanced Microcontroller Bus Architecture) High-performance Bus (AHB), so Hydra can easily work with any host processor that speaks this protocol such as ARM microprocessors. As a result, a typical workflow might look as follows.

1. The host processor checks the status of the coprocessor until it is in the idle state.
2. The host processor sets the environment registers of the coprocessor via Slave Port (shown as the “S” at the bottom of Top Control in Fig. 1), including the entry point of the program, as well as the addresses of the source and target data.
3. The host processor activates the coprocessor by sending a signal to the related control register.
4. Through Master Port (shown as the “M” at the bottom of Top Control in Fig. 1), the coprocessor loads the instructions to Instruction Cache from external memory.
5. The coprocessor runs the program accordingly and loads the input data from external memory via Master Port.
6. After the computation finishes, the coprocessor notifies the host processor using an interrupt mechanism.

In synthesizing a cryptographic coprocessor for pairing, most of the silicon area is for implementing the arithmetic unit (AU) that computes finite-field multiplications and additions inside Hydra’s Axy Engine. Furthermore, multipliers like the one in Fig. 2 typically takes up the most area because multiplication tends to be much more expensive than addition.

Further analysis shows that to compute an optimal ate pairing, we need about $18500 \mathbb{F}_p$ multiplications and 85000 additions/subtractions. Typically, a multiplication operation takes about 5 times or more cycles than an addition/subtraction operation. This means that the time spent on multiplication is roughly the same as that on addition/subtraction, which is quite different from the typical multiplication-bound algorithms.

In this case, it is advantageous to use *heterogeneous* AUs, some of which have complete functionality (called full AU, or FAU), while others are only capable of computing addition and subtraction (called add-only AU, or AAU). Since an AAU is much smaller than an FAU, we expect a higher resource utilization if there is enough parallelism to be extracted by our scheduler.

Optimal resource allocation and scheduling on a set of heterogeneous computational units is in general a hard problem and has been intensely studied in the past. Here we are dealing with the most general cases because we are working with application-specific integrated circuits (ASIC). As we have seen earlier in this section, our architecture is similar to a general-purpose processor. For example, we use a *register file* to connect all AUs, which is actually implemented as part of Data Cache in Hydra. This greatly simplifies inter-AU communication, as all communication will need to go through the register file as the central hub. Also, as long as the register file is large enough, we can completely eliminate load/store operations from/to external memory. Furthermore, our compiler generates straight-line code, so there is no branch and loop, which further simplifies the analysis.

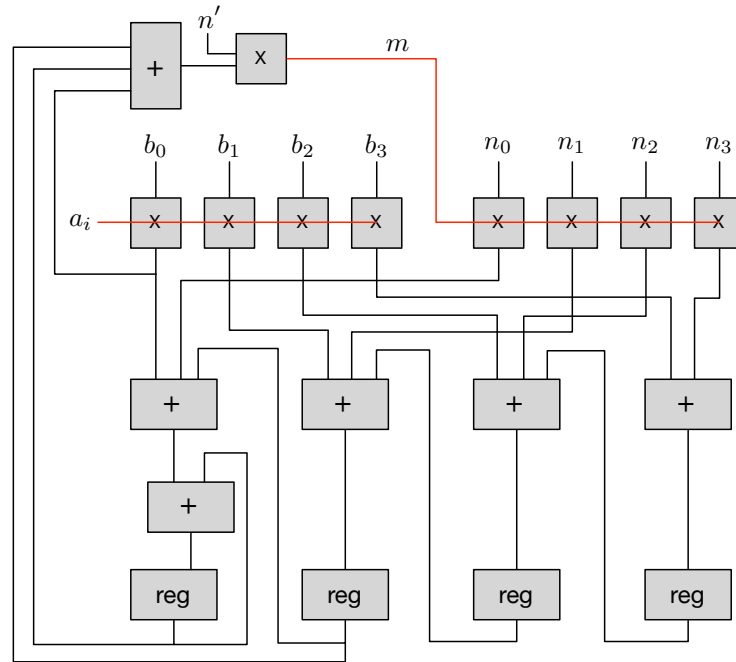


Fig. 2. A four-limb Montgomery multiplier

Now that we have a better understanding of our problem, we turn to the literature to seek solutions. We have tried without success several traditional methods, such as using critical path as the main heuristic for scheduling, as the memory required is unacceptably large. So far, most works in the literature focus on achieving the best speed performance. Although some of these works also take communication into account, none of them have considered memory constraints. The closest work to ours is perhaps that of Cordes, Marwedel, and Mallik [6], in which various constraints including a limit on the number of parallel tasks on the target system-on-chip (SoC) are taken into account, but the memory constraint was not considered.

In this paper, we experiment with one of the simplest methods, namely, the greedy method, to see if it can accelerate our pairing computations. That is, we use a simple in-order architecture in which an instruction is issued if all its dependencies have been resolved, and there is available AU to execute the instruction. This makes the hardware design relatively simple and shifts some work to compiler, which will need to come up with a plan of resource allocation and schedule. It bases its decision on information including the sequence of instructions to be executed and the resource constraints such as the composition of the heterogeneous AUs and the amount of working memory available to hold the intermediary results.

We briefly describe our design here. In each cycle each idle AU will sequentially search for a subsequent instruction whose input data is ready and can be done by this AU. If such an instruction exists and there is enough space in the working memory, it

will be issued to the AU. The AU will be set in a busy state for however many cycles required to execute the instruction, after which it will go back to idle state and try to grab more work to do.

In searching for instructions, an AU only looks for the type of instructions that it can execute and skip the other types. Naturally, there is a trade-off in computation vs. memory because each additional AU will require additional working memory, so it is not immediately clear whether more AUs would lead to a better ATC product. Here we resolve this issue by imposing a limit on register use, under which AAUs will only fire when there is still a reasonable amount of memory available. In other words, FAUs have higher priorities in instruction issuing whenever there are ready multiplications. However, if there is not enough memory space for executing a multiplication, FAUs can also be used for additions/subtractions.

There is another resource we need to consider, namely, the I/O bandwidth of the register file. When there are multiple AUs competing for reading respective input data from the register file, the bandwidth limitation introduces extra delay, which in turns imposes an upper bound for the amount of parallelism we can achieve.

Putting all these together, our compiler will generate a schedule for each of the AUs, dictating what instruction and when each of the AUs should execute, as well as a resource allocation plan, dictating which register should hold what intermediary result. Our compiler automatically searches through the solution space of all feasible schedules and resource allocations.

4 Datapath design

4.1 Full arithmetic unit

As we need to deal with general moduli, we use the well-known Montgomery method for computing modular multiplication $A \times B \bmod N$ [16]. The basic idea is to compute $\bar{A} = AR$ and $\bar{B} = BR$ first, and then compute $(\bar{A} \times \bar{B})R^{-1} \bmod N$ using the Montgomery reduction algorithm. With a good choice of R (usually a power of two), the operation could be computed efficiently.

Our implementation is a direct realization of Montgomery’s algorithm. Fig. 2 shows an example of a four-limb multiplier, and in our actual implementation there are 17 limbs. We use 256-bit operands in the arithmetic units. Also, we use 272 bits to represent a single \mathbb{F}_p element to allow further optimization such as lazy reduction.

Operands A , B , and N are divided into 17 limbs: a_i ’s, b_j ’s, and n_k ’s. It takes 17 cycles to compute the partial products and 3 more cycles, the carries. In the i -th cycle ($0 \leq i < 17$), the partial products of $a_i \times B$ and $m \times N$ are computed using 34 16-bit multipliers and then added with the shifted intermediate results from the previous cycle to produce the intermediate results r_j ’s. The value of m is obtained by summing the most significant part of r_0 and the least significant part of r_1 from the previous cycle, as well as the product $a_i \times b_0$, then multiplied by n' .

Carries are not propagated among the adders in Fig. 2 within the 17 multiply-and-add cycles. As a result, the r_j ’s could be at most $16 \times 2 + \lceil \log_2 17 \rceil = 37$ bits wide, which decides the width of the adders as well as that of the registers. After the 17

multiply-and-add cycles, 6 adders are reused to add up the carries in 3 cycles. For the sake of clarity, the wires and MUXs required for reusing the adders are omitted in Fig. 2. The critical path, as illustrated in Fig. 2, consists of 3 multipliers and 3 adders.

4.2 Add-only arithmetic unit

An AAU consists of an array of 17 adders and can calculate $A + B$ in a single cycle, as well as its carry in another cycle. Since we only add two 16-bit numbers per adder, the intermediate result will be at most 17-bit wide. Unlike the carry computation in FAU, AAU only need to carry 1 bit, so both the register size and the number of cycles can be significantly reduced. Using a design similar to carry-select adders, the carry computation in AAU is very fast compared with that in FAU.

The critical path of AAU is not important, as it is dominated by FAU. Synthesis results show that the area of an AAU is only about 1/10 of that of an FAU. This is largely due to lack of multipliers, use of narrow adders (17 vs. 37), and possibly a shorter critical path.

5 Control design

There are three essential parameters in Hydra's design: the numbers of FAUs and AAUs, as well as the register-file size. Through simulation, we found that the combination of one FAU and two AAUs delivers the best performance, which we will describe in more detail in Section 6. Fig. 3 illustrates conceptually all the relevant components, including datapath and control, for supporting heterogeneous AUs. The counter serves

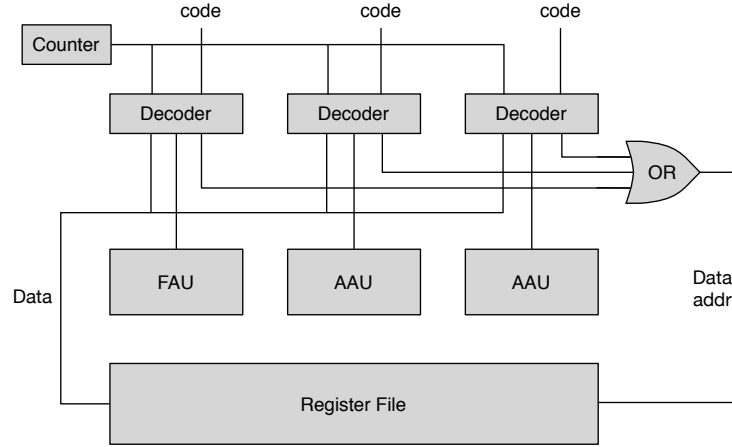


Fig. 3. Conceptual design of heterogeneous datapath and its control

as the synchronization unit by providing the cycle number. An operation code consists

of the addresses of the input and output data, as well as the cycle number that it should be issued. The decoder decides what operation need to be done in this cycle according to the operation code and the counter value.

Each AU has its own decoder and a separate list of operation codes. These decoders are independent of each other, i.e., there is no communication among them. Therefore, the compiler takes full responsibility of making sure there is no conflict among the operations of all the components.

In each cycle, the decoder (implemented inside Decoder) decides on one of the following operations according to the code that matches current counter value.

1. Idle.
2. Send the address for reading data from the register file.
3. Retrieve data and forward it to the AU. This could be done concurrently with operation 2 in the same cycle.
4. Start the computation of the corresponding AU. This could be done concurrently with operation 3 in the same cycle.
5. Retrieve data from an AU, send the address and the data to be stored to register file. This cannot happen concurrently with operations 2 and 3, as there is only one address port in each register file.

Below we describe in more detail Hydra's instruction and data scheduling mechanisms, which is responsible for instruction loading and data management for Axy Engine. It will prepare the corresponding instruction fetch and data loading of corresponding operation for Axy Engine. Typically, the scheduling unit would load and decode the instruction from Instruction Cache and send the corresponding addresses to Data Cache for data loading. Furthermore, the scheduling unit will handle instructions like jump and data rescheduling. This way Axy Engine can concentrate on data processing and avoid unnecessary idle states.

Hydra uses a data prefetch mechanism. To deal with data loading, a general-purpose processor usually uses either extra cycle or specific instruction to maintain. In both cases, the datapath will be in idle state, waiting for data loading. Hydra uses three instruction buffers. The first instruction buffer, the preloading buffer, stores the instruction after the next instruction. The second instruction buffer stores the next instruction itself, and we call the instruction stored in this buffer the "preparing instruction." The last instruction buffer is current buffer, which stores the instruction being executed in Axy Engine. In general, we can decode the addresses from the instruction in the preloading buffer and send them to Data Cache for data prefetching, so after the current instruction finishes execution, the data will be ready at Data Cache. While the preparing instruction is about finish, the input register of Axy Engine can be loaded with the corresponding data from Data Cache. This way the preloading instruction can be executed with the corresponding data already ready in input registers. In the case with single-cycle instructions followed by a multi-cycle instruction, the preparing instruction buffer serves as the temporal preloading instruction buffer, as the current instruction cannot finish in one cycle. With such scheduling mechanisms, we can overlap data loading with data processing in one instruction. Moreover, the prefetch technique avoids the conflict between data loading and processing. We use similar mechanisms for prefetching from and writing back to external memory.

6 Performance evaluation and concluding remarks

As mentioned in Section 5, there are three essential parameters in the proposed coprocessor design. In this section, we first investigate the settings with best performance via architectural exploration and then compare our results against the state of the arts.

All our designs are synthesized using Synopsys Design Compiler. In addition, SRAM blocks such as Data Cache and the register file in it are generated by Artisan’s memory compiler. For architectural exploration, we use Bluespec System Verilog [17]. These designs are then synthesized using TSMC standard cell libraries for their 90 nm and 130 nm technologies. However, Bluespec’s synthesis results are typically 2–3 times inefficient in terms of ATC product than hand-coded designs using low-level hardware description languages such Verilog or VHDL. Therefore, after finding out the best set of parameters, we hand-code a design using Verilog and synthesize it using TSMC 90 nm technology.

Table 1 shows the performance of Bluespec’s synthesis results using different numbers of FAUs and AAUs while keeping register-file size fixed. Using more AAUs should

Table 1. The performance achieved by a single-bank, 96-entry register file

Number of FAU	1	1	1	2	2	2	2
Number of AAU	1	2	3	1	2	3	4
Area (k gates)	144	157	168	246	256	268	280
Total time (ms)	4.34	3.60	3.62	3.23	2.72	2.57	2.55
ATC product	626.20	565.42	608.24	793.47	696.93	687.93	714.42

result in better timing results, but the marginal improvement diminishes as the following start to surface.

1. Multiplication-related operations become the bottleneck of the design, so adding extra AAUs can no longer shift any load from the FAU.
2. The bandwidth of the register file becomes fully occupied.
3. Last but not least, the register file itself becomes fully occupied.

From then on, the increase in area cannot bring in proportional improvement in speed, making it disadvantageous to add more AAUs. As seen in Table 1, with only one FAU, the bound is two AAUs.

Deploying more FAUs will increase this bound of adding more AAUs. However, we hit the limitation imposed by register-file bandwidth, and the extra FAU will compete for the bandwidth with other AUs. In this case, the bound becomes three AAUs, and the resultant ATC products are all worse than using only one FAU.

Table 2 shows the performance of Bluespec’s synthesis results using different sizes of register files in the setting of one FAU plus two AAUs. Increasing the number of registers means that there will be more spare registers for the extra AAUs to fetch next operations and thus improves the speed performance. This phenomenon becomes more

Table 2. The performance achieved by one FAU and two AAUs

Register-file size	80	88	96	128	160
Total time (ms)	5.14	4.67	4.60	4.58	4.57

noticeable when there are more AAUs. However, this comes at a price of a larger register file, so in general it does not improve the ATC product at all.

In Table 3, we compare the performance of our designs against that of two best works from the literature. We stress that it is in general very difficult to compare the

Table 3. Performance comparisons

	Technology	Total area	Cycle time	Total time	ATC product
This work (hand-coded Verilog)	90 nm	116 k gates	5 ns	3.04 ms	353.6
This work (Bluespec)	90 nm	157 k gates	5.92 ns	3.6 ms	565.2
This work (Bluespec)	130 nm	166 k gates	9.73 ns	5.88 ms	976.2
Kammler <i>et al.</i> [13]	130 nm	164 k gates	2.96 ns	15.8 ms	2591.2
Fan, Vercauteren, and Verbauwhede [8]	130 nm	183 k gates	4.9 ns	2.91 ms	532.5

performance of designs using different fabrication technologies. Furthermore, architectural differences also make fair comparison even more difficult, so here we can only try our best to compare apple to apple.

When compared against the design by Kammler *et al.* [13], it is clear that our designs achieve better ATC products. This is mainly because they have a scalar design in which a significant portion of the transistors are allocated to non-datapath components such as control/decoding logic and SRAM. Furthermore, a significant portion of the energy is typically consumed by the circuitry that supports execution of software programs in a processor architecture. A scalar design means that the density of the program code tends to be quite low, resulting in a much less efficient design in terms energy consumption. In return, the benefit of such an approach is a processor that is capable of executing a wide variety of programs not limited to cryptographic pairing. In contrast, our design is a coprocessor and would require the help from a microcontroller in a complete system.

Lastly, it is even more challenging to compare our result with that of Fan, Vercauteren, and Verbauwhede [8]. First, although our synthesis result at 90 nm seems better than theirs at 130 nm in terms of ATC product, it is difficult to tell whether this is merely due to advancement of fabrication technologies. There are several ways we can do a back-of-envelope estimation, and the best we can say is that these two designs are roughly on par with 10–15% of error. On the one hand, our design provides full connectivity with industry standard AMBA AHB as well as a high degree of programmability that would allow the support of, e.g., elliptic curve cryptography on curves over large characteristics. In order to achieve this flexibility, our design needs to load programs from external memories, while on the other hand, their design does not require inter-

facing with external memories because they are specialized for Barreto-Naehrig curves. At the end, the only conclusion we can draw from this apple-orange comparison is that the two designs are roughly on par with different directions of optimization.

References

1. P. S. L. M. Barreto and M. Naehrig. Pairing-friendly elliptic curves of prime order. In *SAC 2006*, pages 319–331. Montreal, QC, Canada, 2006.
2. D. Boneh and M. Franklin. Identity-based encryption from the Weil pairing. In *CRYPTO 2001*, pages 213–229, Santa Barbara, CA, USA, 2001.
3. D. Boneh, B. Lynn, and H. Shacham. Short signatures from the Weil pairing. *Journal of Cryptology*, 17(4):297–319, 2004.
4. J. C. Cha and J. H. Cheon. An identity-based signature from gap Diffie-Hellman groups. In *PKC 2003*, pages 18–30, Miami, FL, USA, 2002.
5. R. C. C. Cheung, S. Duquesne, J. Fan, N. Guillermin, I. Verbauwhede, and G. X. Yao. FPGA implementation of pairings using residue number system and lazy reduction. In *CHES 2011*, pages 421–441. Nara, Japan, 2011.
6. D. Cordes, P. Marwedel, and A. Mallik. Automatic parallelization of embedded software using hierarchical task graphs and integer linear programming. In *CODES+ISSS*, pages 267–276, Montreal, QC, Canada, 2010.
7. A. J. Devegili, M. Scott, and R. Dahab. Implementing cryptographic pairings over Barreto-Naehrig curves. In *Pairing 2007*, pages 197–207. Tokyo, Japan, 2007.
8. J. Fan, F. Vercauteren, and I. Verbauwhede. Faster \mathbb{F}_p -arithmetic for cryptographic pairings on Barreto-Naehrig curves. In *CHES 2009*, pages 240–253. Lausanne, Switzerland, 2009.
9. G. Frey and H.-G. Rück. A remark concerning m-divisibility and the discrete logarithm in the divisor class group of curves. *Mathematics of Computation*, 62:865–874, 1994.
10. S. Ghosh, D. Mukhopadhyay, and D. Roychowdhury. High speed flexible pairing cryptoprocessor on FPGA platform. In *Pairing 2010*, pages 450–466. Yamanaka Hot Spring, Japan, 2010.
11. S. Ghosh, D. Roychowdhury, and A. Das. High speed cryptoprocessor for η_T pairing on 128-bit secure supersingular elliptic curves over characteristic two fields. In *CHES 2011*, pages 442–458. Nara, Japan, 2011.
12. A. Joux. A one round protocol for tripartite Diffie-Hellman. In *Proceedings of the 4th International Symposium on Algorithmic Number Theory (ANTS-IV)*, pages 385–394, Leiden, the Netherlands, 2000.
13. D. Kammler, D. Zhang, P. Schwabe, H. Scharwaechter, M. Langenberg, D. Auras, G. Ascheid, and R. Mathar. Designing an ASIP for cryptographic pairings over Barreto-Naehrig curves. In *CHES 2009*, pages 254–271, Lausanne, Switzerland, 2009.
14. A. J. Menezes, T. Okamoto, and S. A. Vanstone. Reducing elliptic curve logarithms to logarithms in a finite field. *IEEE Transactions on Information Theory*, 39(5):1639–1646, 1993.
15. V. S. Miller. The Weil pairing, and its efficient calculation. *Journal of Cryptology*, 17(4):235–261, 2004.
16. P. L. Montgomery. Modular multiplication without trial division. *Mathematics of Computation*, 44:519–521, 1985.
17. R. Nikhil. Bluespec System Verilog: efficient, correct RTL from high level specifications. In *MEMOCODE 2004*, pages 69–70, San Diego, CA, USA, 2004.
18. P. Schwabe. Pairing computation on BN curves. <http://cryptojedi.org/crypto/#bnpairings>.

19. F. Vercauteren. Optimal pairings. *IEEE Transactions on Information Theory*, 56(1):455–461, 2010.