VDF FPGA Competition Round 1 Report

Prepared for: VDF Alliance
By: Supranational
Date: 10/14/2019
Version:Final[1]

[1] This report makes no statements or warranties and is for discussion purposes only.

Executive Summary

Round 1 had 30 individuals sign up, formed into 18 teams. In the end we received 10 submissions from 5 teams, with the best design reducing the latency per squaring from 50 ns/sq to 28.6 ns/sq, in line with our expectation of getting to around 30 ns/sq.

All entries were derivatives of the Ozturk baseline design. Optimizations focused on making better use of FPGA resources and were along the lines of the suggested areas. Common themes were switching to 6:2 compression trees to better use LUT6 resources, fully utilizing the CARRY8 primitive, re-pipelining to increase parallelism given the smaller (1024) bitwidth, and using alternative memory resources for the lookup tables (e.g., LUTRAMs).

Of the 5 teams that submitted only 2 had fully working designs. In our testing and reviews it became clear that most teams never ran on F1 hardware and many had build repeatability issues. The working designs were from professionals with prior hardware, FPGA, and infrastructure experience. Feedback from contestants post-competition is that while the baseline design, test portal, office hours, etc. were very well done and significantly lowered the barrier to entry, building hardware is simply hard and time consuming. Some viewed round 1 as a chance to learn and ramp, with round 2 being where more ambitious optimizations can come in given the additional time.

Overall we're pleased with the turnout and results for this first round. Given the difficulty of the endeavor 5 teams submitting is good and the overall speedup exceeded our expectations. We would like to have seen alternative algorithms such as Chinese Remainder Theorem, but it is still possible we'll see alternatives in round 2.

In order to maximize continuity and efficiency for contestants who have already ramped, round 2 will reuse most of the infrastructure from round 1 as-is. The main changes will consist of a longer run time for measuring performance (t=2^33, ~4 minutes) to ensure precomputations and alternate algorithms are not disincentivized, and the prize will increase to $5000 / ns, as planned.

Resources Offered

Due to the complex nature of building and testing an FPGA design, several resources were offered to contestants to enable efficient work, simplify the overall process, and minimize any out of pocket cost. Hosting a contest on AWS F1 in particular offers many benefits in the form of unlimited instances, no capital or long term cost, fully supported infrastructure, etc. However there is an associated cost to use AWS EC2 instances ($0.744/hr for development, $1.65/hr for FPGA) and a learning ramp for those not familiar with the Xilinx SDAccel environment.

For additional formation see https://supranational.atlassian.net/wiki/spaces/VA/pages/42336279/Offered+Resources.

Xilinx Vivado Trial License

Xilinx provided contestants with a trial license of Vivado for use in the competition. For contestants with appropriate hardware at home, such as a high end desktop, this enabled an on-premise replica of the AWS F1 environment for development purposes at no cost.

For round 1 we provisioned 15 Vivado licenses.

AWS Promotional Credits

AWS provided contestants with access to a $200 credit for AWS EC2 services. Teams could use this to provision either development (z1d) or FPGA (f1) EC2 resources.

For round 1 we provisioned 11 AWS coupons.

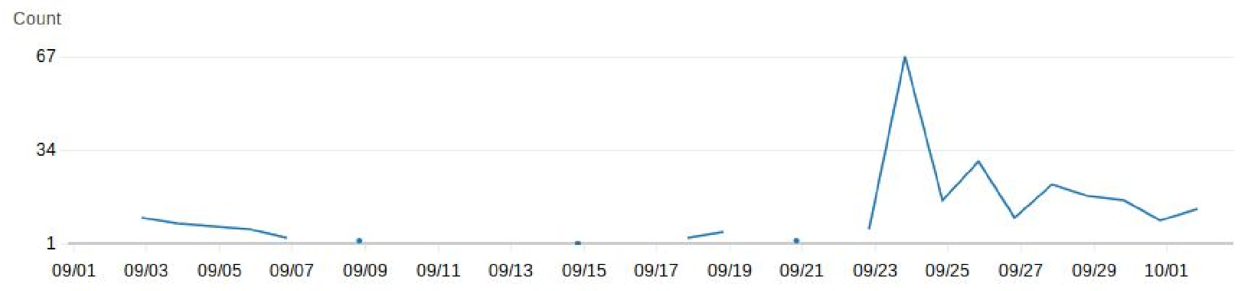
Automated Test Portal on AWS

To lower the barrier to entry as much as possible Supranational developed an automated test portal contestants could use to build, test, and synthesize designs directly on AWS infrastructure.

The test portal links to a team's privately shared GitHub repository. Upon push of a new revision to the "vdf-portal" branch the test portal provisions a new EC2 z1d instance, performs the requested test, and returns log files and other collateral to the contestant via an encrypted file in S3.

The portal uses AWS API Gateway, Lambda, EC2, Relational Database, S3, and Simple Email Services to operate in a serverless manner with minimal fixed infrastructure. For additional information see https://supranational.atlassian.net/wiki/spaces/VA/pages/44204033/Automated+Test+Portal.

The following graph shows test portal invocations over the contest period. Several teams used the test portal extensively, especially late in the contest cycle. Aside from the initial spike, daily average invocations were in the 15-20 per day range in late September.



Evaluation Criteria

The contest submission requirements and judging process are documented on the VDF Alliance wiki (https://supranational.atlassian.net/wiki/spaces/VA/pages/36601857/Contest+Judging). The majority of the evaluation process consisted of ensuring that the design results were fully reproducible from source files and measuring the design performance on AWS F1 hardware.

This consisted of:

- Reviewing general submission requirements (git commit signoff, submission_form.txt, documentation)
- Running Vivado behavioral simulation
- Running hardware emulation
- Synthesizing the design and generating a bitstream
- Running for 2^30 iterations on F1 FPGA hardware using both the public test input and the secret test input using the provided modulus. The secret input was used to ensure the design was not tailored to the public input.

	Input	Result (2^30 squarings)
Public	0x2	0x9782776834334634490446343758704728706980122657033141222406929631982781114105293252444979173994924549755313289718816652420124314107449156688222852673024696927113240716169907514261823484008194829047317452425855361884165852504086556390349991640188347831084926001670580437428161157316196941905575574310934275893
Secret	0x73757072616e6174696f6e616c	0x6174455735878898184800653437346193295490567128789082553532855498184947856454608565547721441367784705391665842255379498418396571810496631359407541596093058826306522770295733784008844388005290016646888021676887735177385955637383202217508672362921833073047830670275894368152356093110984504216077400752307254681 2

Contestants

30 individuals signed up for the competition, formed into 18 teams. Signups were pretty well distributed throughout the competition period.

Overall the entrants' occupations were along the lines of what we expected going into the competition, including:

- Academics
- Enthusiasts
- Professionals from various startups and other companies
- Consultants in the silicon space

Candidates backgrounds also covered a variety of areas of expertise:

- FPGA
- EDA tools
- Algorithm design
- Modular multiplication

Submission Summary

There were a total of 10 design submissions across 5 teams. Multiple submissions by a person or team are explicitly allowed by the rules, which say each submission will be evaluated independently.

In all cases the reason for submitting multiple designs fell into one of two categories:

- To lock in a conservative time and take risk on a more aggressive target
- To fix issues identified during the evaluation process

The following table briefly summarizes all received submissions. “Directory” refers to the path for the archived design in repository https://github.com/supranational/vdf-fpga-round1-results, “Notes” provides a brief summary of the purpose of the submission.

Team Name	**Directory**	**Notes**
Eric Pearson	eric_pearson-1	Targets latency of 31.5ns/sq
Eric Pearson	eric_pearson-2	Reduces latency target to 28.6 ns/sq
FPGA Enthusiast	fpga_enthusiast-1	Targets 45.7ns/sq, provided feedback on clocking issues
FPGA Enthusiast	fpga_enthusiast-2	Incorporates MMCM (PLL), provided feedback fails routing
FPGA Enthusiast	fpga_enthusiast-3	Resolves routing issue
Silicon Tailor	silicon_tailor-30	Targets 30ns/sq
Silicon Tailor	silicon_tailor-291	Targets 29.1ns/sq
Silicon Tailor	silicon_tailor-296	Targets 29.6ns/sq
Geriatric Guys with Gates	geriatric_guys_with_gates	Single submission from team targeting 44.96ns/sq
Andreas Brokalakis	andreas_brokalakis	Single submission from team targeting 24ns/sq

Results

The evaluation process identified Eric Pearson as the clear winner. His design simulated, synthesized, and ran at the target latency out of the box and achieved an impressively low latency of 28.6 ns/sq.

Qualifies	**Team Name**	**Directory**	**Notes**
Yes	Eric Pearson	eric_pearson-1	Fully functional at targeted 31.5 ns. Contestant did not use Vivado behavioral simulation, required additional instructions.
Yes	Eric Pearson	eric_pearson-2	Fully functional at targeted 28.6 ns/sq.
No	FPGA Enthusiast	fpga_enthusiast-1	Clocking issues on AWS F1 caused design to run at 56ns/sq instead of 45.7.
No	FPGA Enthusiast	fpga_enthusiast-2	Design fails routing due to congestions
No	FPGA Enthusiast	fpga_enthusiast-3	Design fails to close timing
No	Silicon Tailor	silicon_tailor-30	Faults FPGA, crashes host. Likely caused by PLL locking issue.
No	Silicon Tailor	silicon_tailor-291	Faults FPGA, crashes host. Likely caused by PLL locking issue.
No	Silicon Tailor	silicon_tailor-296	Faults FPGA, crashes host. Likely caused by PLL locking issue.
Yes	Geriatric Guys with Gates	geriatric_guys_with_gates	48ns / sq
No	Andreas Brokalakis	andreas_brokalakis	Design did not simulate or synthesize. After adding missing files hit resource constraint problems. After adjusting pblocks and target clock frequency did not meet target timing. Achieved 40ns/sq (but got wrong answer).

Of the 5 teams that submitted, only 2 teams had fully working designs. This speaks to the overall difficulty of building FPGA based designs. Through interacting with contestants, reviewing designs, and the types of issues seen some themes emerge for the low success rate:

- Teams started late. General activity didn't really pick up until late September.
- Lack of testing on F1 hardware. Some of the failures would have been caught and likely fixed by running candidate designs on real hardware.
- Clocking issues. While this did not affect the outcome, some teams encountered issues running valid designs at AWS supported frequencies.
- Tool runtime. Some teams reported long tool runtime (6+ hours). For teams that adapted the baseline design to use the entire FPGA runtimes in the 6-10 hour range are what we would expect.

Other Observations

- Silicon Tailor was a close 2nd from a latency point of view. It seems likely that he could have resolved the crashing issue given a little more time.
- The design from Andreas targeted a significantly lower latency (24ns/sq) but had many issues. It’s unclear from initial assessment of his design if the implementation is actually viable at such a low latency.
- Overall submissions with competitive latencies came from teams background in FPGAs (Eric Pearson, SiliconTailor, Andreas Brokalakis).

Submissions Detail

The following pages contain detailed information from the contestants about the final or best submission from each team along with a short summary of the approach.

For round 1 all of the submissions started from the Ozturk design and applied FPGA specific implementation optimizations. Common themes do emerge from the entries, all in line with the recommended areas to explore provided at the start of the competition:
- Unroll the pipeline
- Optimize the multiplier for squaring rather than general exponentiation
- Optimize the compressor trees to better use FPGA resources
- Optimize the reduction lookup tables to use alternative storage (LUTRAMs)
- Fully utilize the FPGA resources to increase parallelism

Eric Pearson

Summary

The Pearson design started from the baseline Ozturk design and fully unrolled it to complete the square and reduction in 2 phases, 2 cycles per phase, making heavy use of multi-cycle paths. For the compression trees it uses the hardened CARRY8 primitives. The reduction lookup tables were moved into LUTRAMs and distributed throughout the fabric.

Detailed Description of Approach from Contestant

Expected result (avg ns/square): 28.5 nS, 4 cyc @ 140.1 Mhz. This design must timing close as auto clock adjust not available with MMCM.
Design documentation (below):
- Used same numeric representation, word length, and interface as in the provided reference.
- unrolled as 2 stage pipeline, with lut address = square product register. see: modular_square/model/pictures.txt
- Stage1: Parallel 1Kbit square unit based on 17bit DSP multiplier, with native CARRY8 adder trees
- Stage2: Parallel 2Kbit product modulus using 5/6/6bit LUTRAM based word reduction roms, and native CARRY8 adder trees
- State machine controlled stage register enables and valid signals to give 2 clock cycles per stage. Multi-cycle timing constraints
- State machine initialization delay of 4 cycles for starting value paths. Multi-cycle timing constraints
- Power savings by reset of modsqr unit upon while IDLE or upon completion of iterations.
- Power ramp up of 1ms by integrated ramp pulse modulated delay of state machine register enables.
- MMCM pll 91:102 clock multiplier of 125 Mhz ref to give 140.1 Mhz or 28.5ns. clock domain crossing logic in the wrapper. Max-delay timing
constraints.
- For simulation modify POWER_RAMP to 'h100 (normally 'h1 for synth). See line 57 in modular_square_8_cycles.sv

FPGA Enthusiast

Summary

This FPGA Enthusiast design primarily changed the 3-to-2 compression trees to 6-to-3 to make better use of LUT6 primitives, thereby reducing the gate depth.

Detailed Description of Approach from Contestant

Expected result (avg ns/square): 45.7 ns (8 clks/square @ 175 MHz)
Design documentation (below):
1. The Squarer is based on the Project from Ozturk, from the pll_cdc branch

2. The adder three compression is improved: Originally, the adder tree is based on 3-to-2 CSA. Now it uses 6-to-3 CSA (which is better suited for 6-input LUTs). This reduces the number of logic levels in the adder three.
The following files were added to the primitives to enable 6-to-3 CSA:
\primitives\rtl\carry_save_adder_6_3.sv
\primitives\rtl\carry_save_adder_tree_6_3_level.sv
\primitives\rtl\compressor_tree_6_to_3.sv
\primitives\rtl\full_adder_6.sv
Due to the change, also some adjustments in the main squaring file (modular_square_8_cycles.sv) were required. The final addition after the CSA adders has now three instead of two inputs (a three-input adder can be implemented in one carry-chain, it is therefore only little costlier than a two-input adder).

3. Small improvements in the existing main squaring file (modular_square_8_cycles.sv):
 - max_fanout attribute
 - "Pipelining" the main FSM so that the "next_state" signal is already a FF output. This way, control logic has easier timing.

4. Due to the modifications, the core clock frequency could be increased from 161 to 175 MHz

Silicon Tailor

Summary

The Silicon Tailor design includes several of the optimizations seen in Pearson design along with a different approach to the squaring phase. First the entire square/reduce operation is unrolled into a single long clock cycle. Squaring is accomplished using a recursive multiplier optimized for squaring rather than exponentiation. The recursive approach uses fewer DSPs but increases the gate depth and requires signed numbers. This is still potentially a latency saving change in FPGAs if it reduces slow SLR crossings. Beyond that it moves the reduction lookup tables into LUTRAM and increases the compression cell size to 5:2 and 4:1 to make better use of FPGA resources.

Detailed Description of Approach from Contestant

The overall architecture used is similar to Ozturk, a central loop circuit is used to perform squaring and modulo using a redundant number format. This loop is optimized to have the minimum latency possible.

Extra logic is needed to convert the initial value into the redundant number format, and extra logic is also needed to convert the redundant number format back into a twos complement form. The extra logic has negligible effect on the overall performance on the circuit as this is dominated by the loop latency as the loop is iterated for 2^30 cycles. Hence this extra pre and post processing logic can be pipelined to amply meet the required clock frequency.

There are several key design choices that have been used to achieve lower latency than OzTurk:
(i) A recursive squarer is used rather than Ozturk's tabular multiplier.
(ii) DSP blocks are combined to perform 34x34 or 42x42 squaring (rather than as isolated 18x18s)
(iii) The modulo circuit is implemented using 6-LUTs rather than RAMs.
(iv) Adder trees use 5:2 and 4:1 adders that can be much more efficient on a Xilinx FPGA
(v) The loop circuit is only registered once, and each iteration of the loop takes one clock cycle (albeit at a slow clock)

As with Ozturk, all large numbers are split into a number of 'symbols', where each symbol is mostly processed in isolation, with carrys between adjacent symbols being resolved only when needed. This means that the loop logic does not contain any large carry chains.

All loop logic is placed in SLR2 (together with the PLL). All non-loop logic is placed in SLR1, this ensures that Vivado place-and-route has the maximum routing and logic resources available to optimize the core loop. Apart from these SLR placement constraints, no other placement constraints are used.

The design has been constructed to be a drop-in replacement for the OzTurk design, (and in fact the modulo coefficients are now generated using verilog macros rather than the python scripts).

The sequence of functions for the Core-loop are as follows:

(1) Multisymbol-Squarer
(2) Signed Carry Correction
(3) Modulo lookup tables
(4) 200:1 Adder tree for Modulo terms
(5) Unsigned Carry Correction
(6) Register --> connected to (1)

The loop invariant is that at any iteration the Register contents can be converted to two's complement form and modulo'd to yield the expected result.

Note that the register value is in a redundant form, meaning that it may appear a lot larger than the 1024 bit number it represents, however as the Multisymbol-squarer has been designed to work with this larger number form it is not necessary to reduce it down to a 1024 bit format inside the loop (which would be costly in latency!). Only the post processing pipeline needs to do this.

PLL design

To maximize the loop latency, the fewest number of pipeline stages are used, even though this reduces the clock frequency at which the loop can iterate. The reasoning behind this is that pipeline registers would unnecessarily constraint the timing of the loop logic (and potentially add delay too), and are not required as (i) no resource sharing or reuse is needed, and (ii) the small LUT based memories can be read asynchronously.

The modular_square_metzgen_wrapper module must transfer data to a slower clock domain where the VDF function will run. The host clockdomain (chosen to be 125MHz) is used to transfer data from the PC host, but the VDF clock runs at a much slower rate to match the VDF latency (~33MHz). Dual clock fifos are used to transfer data between clock domains, and special care is taken to ensure that these are reset before use, and no data is transferred until the VDF-PLL has achieved 'lock', and the Fifos have been initialized after reset. A holding register is used to hold the first reset and start value until the VDF clock domain is ready.

The existing interface does not allow the parent block to be stalled to wait for a reset to complete. Because of the remote clock is slower than 'clk' then it cannot be guaranteed that valid sq_out values will not continue to be received from a previous run even after reset has been asserted. This design uses a sequence-number system to work around this where all sq_out values are tagged with the current sequence-number. The sequence number is only incremented whenever a start pulse is asserted, and then all outputs with a different sequence-number can then be ignored. This ensures that all values still in the pipeline or Fifos from an old run continue to flush out even after a reset pulse.

Alternative vdfpll.sv files have been provided that can be used to fine tune the latency of the VDF function. In experiments, a latency close to 29ns is achievable (after 11 hours!), but 30ns is much easier to achieve (in just 8 hours).

Number formats

Numbers are represented in the Core-loop Register as a polynomial of 32 symbols with radix 2^33, and each symbol is an unsigned 34-bit value. This means a number, X, is represented as:

    X = Sum_i (Symbol[i] * (2^33)^i) for i = 0..31

A radix of 2^33 is used (instead of 2^32) to provide extra bits for numbers larger than 1024-bits. A minimum of (1024 + 8) bits are needed in total to store the 200:1 sum from the Modulo-term-adder. Symbols also store an extra bit that stores any carry-save information, this is necessary to avoid long carry chains.

The Register format is also used as the input to the Multisymbol Squarer.

The square operation yields a result that is twice the number of symbols (and one extra symbol to store carry information). The Multisymbol squarer uses both addition and subtraction and so must use a slightly different number format that is based on Signed-symbol values. The size of each symbol grows as it progresses through the multisymbol squarer but it never exceeds twice the original symbol size. This means that a signed carry correction stage can add the excess bits in each symbol to the next symbol yielding a smaller final number format where each symbol is just 35 bits (including 1 sign bit and 1 carry save bit).

The Modulo lookup tables yield almost 200 unsigned 1024-bit terms. These terms are each split up into 32 symbols with radix 2^33. As the adder tree adds, carrys are saved in the symbols without propagating them, so each symbol grows up to 8-bits longer. A final unsigned carry correction stage adds these extra 8-bits of each symbol into the next symbol to return to the original number format. Note that any carry from the last digit can be safely discarded as the final sum would never be able to reach that high.

Squarer Architecture

The Ozturk implementation use a 'rectangular' multiplier to implement a large squarer, multiplying every 18-bit symbol with every other. This requires O(N^2) multipliers which is about 3,250 18x18 multipliers -- much too large to fit in a single SLR. Because the OzTurk multiplier is too large to fit in an SLR and the latency penalty of crossing SLRs is very high; the baseline implementation circumvents this by reusing hardware (over 8 clock cycles). So whilst this should be a very low latency implementation in theory, the size of the hardware means that latency must be sacrificed for area reduction.

This implementation uses a 'recursive' squarer instead which only needs just O(N) multipliers and O(N log N) adder-logic. This reduces the resources needed for a large 1024-bit squarer significantly (to just 940 DSP blocks). Although this implementation theoretically has double the number of logic levels compared to the OzTurk squarer, it fits comfortably within a single SLR logic region, yielding a reduced latency overall.

A recursive squarer is built by splitting the input symbols in half (upper half and lower half sets). This can be written as:

X = A * 2^(N/2) + B

where A is the upper half set of symbols and B is the lower half.

X^2 = (A * 2^(N/2) + B) * (A * 2^(N/2) + B)
    = (A^2) * 2^N + (2*A*B) * 2^(N/2) + (B^2)

Note that: (A+B)^2 = A^2 + 2*A*B + B^2, and there 2*A*B can be replaced by ((A+B)^2 - A^2 - B^2)

Hence an N-symbol number, X, can be squared by computing A^2, B^2, (A+B)^2 and combining the results using addition and subtraction. Note that A, B, A+B are all N/2-symbol numbers, so N sized problem has been reduced to an 3 N/2-sized problems.

This can be shown pictorially as:

```
      Square({ A, B }):
                    [ A ][ B ]
             * [ A ][ B ]
      ========================
                    [  B*B   ]
             [  2*A*B   ][ 0 ]
  +   [  A*A   ][ 0 ][ 0 ]
```

```
       ========================
       [  A*A   ][  B*B   ]
  +           [  2*A*B   ][ 0 ]
       ========================
       [  A*A   ][  B*B   ]       Therefore: 2*A*B = (A+B)^2 - A^2 - B^2
  +           [ (A+B)^2 ][ 0 ]
  -           [   A*A   ][ 0 ]
  -           [   B*B   ][ 0 ]
       ========================
       [Upper][  Middle ][Lower] <-- Sections
       ========================
```

Note that because this technique uses subtraction, each symbol must be capable of representing a signed-number. Also (A+B) requires the middle symbols to grow by one-bit (at each recursive stage) to prevent loss of precision.

This recursive squaring reduction is performed repeatedly until the base case where we only need to square a single symbol. Because some symbols have grown in bitwidth slightly, this last symbol will range from a 32-bit to a 40-bit number. A 42x42 unsigned squarer is built from 4 DSP blocks, and where possible a smaller 34x34 unsigned squarer that only requires 3 DSP blocks is used instead.

Modulo Architecture

The Modulo lookup tables generate the adder terms needed. The first three adder terms have special significance, whereas the remaining terms are computed in groups of 6 using lookup tables to their equivalent 1024-bit value when Moduloed. The first modulo term is a constant that is used to cater for the sign bits of all Symbols ((sum_n -(2^n))%M for n = set of all sign bit positions), this allows us to invert all the signbits and treat them as positive 2^n terms thereafter. The second term is the lower 33 bits of the first 32 Symbols, and the third term is the adjusted carry-save bits for those first 32 Symbols.

By restricting the lookup tables to be 6-input or less, native 6-LUTs are used rather than the larger, slower RAMs. The number of additional LUTs required to do this is small compared to the Modulo-adder tree, and the routing delays tend to be lower for 6-LUTs compared to RAMs.

Adder Trees

Both the squarer and the 200:1 adder-tree used by the Modulo use a large number of adders. A very compact adder design can be used to reduce the area and the latency of these adder trees.   The Xilinx FPGA architecture can support 2:1 adders in one level of logic or 3:1 adders that are theoretically as compact but require two levels of logic. This design uses a combination of 5:2 and 4:1 adders which can make use of the LUT5s that preceed the Xilinx CLB carry chain to achieve much lower latency than conventional 2:1/3:1 adder trees.

Experiments were also done with using unused DSP-blocks in the SLR as large adders (eg: 5:1), but the routing delays were found to be worse.

Post-processing

The loop register holding the current VDF value is continually propagated back to SLR1 where a 2-stage pipeline is able to convert it back into twos complement form. The latency of this circuit is not so important as it is only really necessary on the final iteration.

This is achieved in two stages, the first stage is to use modulo-looukup tables to reduce the 32x34-bit symbols into a set of 8 1024-bit numbers each of which is less than the MODULUS. This is done is a similar fashion to the modulo-lookup function in the VDF loop. The second stage is to sum these 8 values, but now we take care to ensure that the sums do not exceed the MODULUS, so we use
modulo-adders.

Although it is possible to use 1024-bit adders, for this sized adder it is lower latency to use carry-select adders. This is achieved by splitting the 1024-bit values into 32-bit sized symbols, symbols are then summed independently, but we compute the sum for two cases, one assuming carry-0 and the other assuming carry-1. Multiplexors are used to select the correct sum to output based on the carry-outs from previous symbols. So the final symbol is a circuit composed of two 32-bit adders selected by a mux whose control input is a (simple) function of the 31-previous carry-outs, and has lower latency than a full 1024-bit carry chain.

Modulo adders are implemented by computing two values, the sum and the sum-minus-the-modulus. If the sum-minus-the-modulus is negative, then the sum is the correct answer, otherwise sum is too large and the sum-minus-the-modulus is chosen. Note that although the sum-minus-the-modulus is technically a 3-way adder, and because the modulus is constant the code has been written so that the logic is packed into just 1 level of logic on the Xilinx FPGA architecture.

Post-processing could also be done in software, however this implementation is not too large (and much faster!).

Geriatric Guys with Gates

Summary

Like the other entrants, the Geriatric design used Ozturk as a base. The main optimization implemented is to replace the 3:2 compressor with a 6:3 compressor. Further changes were made to the main state machine and synthesis settings to achieve improve clock frequency.

Detailed Description of Approach from Contestant

We started with the Ozturk 1024b 8-cycle baseline design. The critical path is through the compressor, so that was an obvious place to start.

Ozturk used a simple 3-to-2 compressor. The selected Xilinx part supports LUTs with six inputs, thus a 6-to-3 compressor is suitable. This can be implemented with three LUT6. Each LUT6 is a single logic level, which overall saves much time on the critical path.

In the main state machine, several critical control signals were re-coded to come straight from flops. These signals have heavy fanouts and need to be generated as fast as possible.

Several changes were made to the synthesis and implementation settings. phys_opt is run post-placement and post-route.

The design is overconstrained (tight clock period) to get optimum results.

Andreas Brokalakis

Summary

The Brokalakis design also started from Ozturk. Like Silicon Tailor, the multiply operation was unrolled and optimized for squaring. Unlike Silicon Tailor, Brokalakis used the same tabular algorithm as the baseline. As a result the pipeline was reduced from 8 to 6 cycles. Finally the lookup tables were modified to fully utilize the BlockRAM address space by storing two elements per BlockRAM.

Detailed Description of Approach from Contestant

The implementation provided in this repository is based on the low latency algorithm provided by Erdinc Ozturk of Sabanci University. However it replaces the main implementation provided in a number of ways in an effort to produce a more optimized (in terms of latency) final FPGA implementation.

In the following subsections, I provide the different ways this can be achieved.

** Attacking the generality of the multiplier implementation **

INITIAL CONCEPT

The following were the initial idea behind the multiply that was provided by
vdf alliance.

Multiply two arrays element by element
The products are split into low (L) and high (H) values
The products in each column are summed using compressor trees
Leave results in carry/sum format

Example A*B 4x4 element multiply results in 8 carry/sum values

```
                                            |-----------------------------------|
                                            |    B3   |    B2   |    B1   |    B0  |
                                            |-----------------------------------|
                                            |-----------------------------------|
                                    x       |    A3   |    A2   |    A1   |    A0  |
                                            |-----------------------------------|
  ------------------------------------------------------------------------------
   Column
```

```
    Row        7      6      5      4      3      2      1      0
        0                                         A00B03L  A00B02L  A00B01L  A00B00L
        1                                  A00B03H  A00B02H  A00B01H  A00B00H
        2                                  A01B03L  A01B02L  A01B01L  A01B00L
        3                    A01B03H  A01B02H  A01B01H  A01B00H
        4                    A02B03L  A02B02L  A02B01L  A02B00L
        5             A02B03H  A02B02H  A02B01H  A02B00H
        6             A03B03L  A03B02L  A03B01L  A03B00L
        7 + A03B03H  A03B02H  A03B01H  A03B00H
        ----------------------------------------------------------------------------
        C7,S7  C6,S6  C5,S5  C4,S4  C3,S3  C2,S2  C1,S1  C0,S0
```

IMPROVED VERSION

Since we actually want to do a squaring, therefore it is AxA rather than AxB. So then we can make a lot of improvements.

1.
Basic observation: since A and B are the same, in the picture above notice that for example A01B00 will be the same as B01A00. Therefore we can seriously reduce all multipliers required from n^2 to (n^2+n)/2

2.
Based on the previous observation, in each column the terms, for example, A00B01L and A01B00L are the same. As a result, we can shorten significantly the depth of the CSA tree, by using only the upper half of the grid and using all numbers multiplied by 2 (which is not a true multiplication but a shift and in actual implementation just a wiring issue), except for the numbers in the main diagonal which are unique. The result is that in the tree now we have to add at most n numbers instead of 2n-1

3.
Reducing the tree depth by as much, means that we can follow a much wider multiply strategy without sacrificing clock frequency as the critical path becomes significantly smaller.

4.
Problem: the above works only for squaring and NOT for general multiplications. This means that the strategy described cannot be generally applied in the reference source code provided by vdfalliance.
Solution: It is now possible to complete the whole squaring operation (for all elements + redundant elemements, i.e. no segmentation) in 2 cycles and the synthesis/implementation prove that this can be done safely @250MHz.

Note: the drawback is that significantly more DSPs are now required.

** Simpler pipeline **

By using the aforementioned approach, the whole NUM_ELEMENTS*NUM_ELEMENTS squaring can be carried out in two (2) cycles and synthesis and implementation results demonstrate that this can be achieved with a 250MHz clock. The result is that the code becomes much simpler significantly reducing wiring and improving routability despite the fact that much more resources are required (DSPs kinds explode). The circuit is more regular and the wiring simpler. This way the routing-to-logic delay becomes smaller (routing delay is approx. 2x the logic delay while in reference implementation it is far worse).

** Stressing the LUTs for reduction **

The shortcoming of the aforementioned approach is that significantly more LUTs are required, as the optimizations of the reference code cannot be applied in this implementation.

A simple trick has been devised in order to reduce the stress on memory resources: since blockRAMs as primitives have a fixed address width (the data stored per memory slot can be expanded by parallel cascading more BRAMs) then it becomes "free" to use a memory with 512 elements instead of 256 (for example for the case of LUT8 as described in Ozturk's paper). As such, in each instantiated LUT, we use all available address space and load the precomputed values for two elements instead of one.

** Simpler pipeline leads to shorter pipeline **

Since all previous operations are carried for the whole number that we have to deal with, this means that the pipeline can be reduced from 8 to 6 cycles following this structure:

CYCLE 0:
Core multiplications

These are all carried out in parallel using 17x17 multipliers implemented in DSP resources

CYCLE 1:
CSA adder tree

Using the construction mentioned above (using a grid of elements multiplied by 2), the Carry Save Adder as provided by the reference code (compressor tree) can be used directly without modifications.

CYCLE 2:
Carry propagate addition

Using DSP resources again, the carry propagate addition can be realized in a single cycle.

CYCLE 3:
LUT-based reduction

The upper part of the results of the CPA are reduced through the look-up tables. Notice that we do not employ the lookup tables provided by the reference code, but we generate the ones that are appropriate. Also we implement two modules that handle the LUT8 and LUT9 lookup tables.
The result of this process requires a single cycle.

CYCLE 4:
CSA adder tree

The results are accumulated in a single cycle.

CYCLE 5:
Carry propagate addition

This is practically the same as Cycle 3.

Cycle 5 completes the process providing the final result.

Performance Expectations

As mentioned above the implementation is designed to take 6 cycles to complete.

Each pipeline stage is constructed with a 250MHz clock constraint. This clock frequency is one of the supported by the AWS F1 and therefore has been deemed as a valid target.

Synthesis and implementation results demonstrate that this clock can be achieved by the design.

As such, the expected latency for modular square can be expected to be: 6 cycles * 4ns/cycles = 24ns.

This improves the baseline model by 50ns-24ns = 26ns.

Post Mortem

Following round 1 we contacted several teams to get feedback on the competition - what went well, difficulties encountered, time spent, resources used.

Positive feedback

- Office hours were really helpful. About the right frequency. Recordings were great when couldn't attend live.
- Test portal was essential, person who did that is a genius. Brilliant idea
- The baseline design was great - like a class project at university. Got an amazing result. People don't appreciate how difficult that part of that design is. Clearly spent a lot of time on it.
- AWS Coupon worked well. Credit was right sized. Spent $203, used the $200 credit and only cost $3. Would like another for round 2.
- Great job making the project approachable for non-hardware people

Areas with friction

- Building hardware is hard. Really hard. Huge learning curve for non-industry hardware people.
- Figuring out SDAccel, Makefiles, etc is a huge part of the contest which takes a lot of work. Without working on commercial design you don't get it. Repeatability and scripting is a big part of the game.
- Vivado and SDAccel results did not align as well as hoped, especially in the sub-nanosecond range
- Lots of ways to communicate - emails, github, confluence, telegram, discourse. Not always clear where to look. Concern there could be inconsistencies between them. It would be nice to narrow it down some.

Next Steps

We are now preparing for the launch of Round 2 of the competition with the following parameters:

Start Date	17:00 EDT Oct 15
End Date	23:59:59 EDT Dec 31
Baseline Latency	28.6 ns / sq
Prize	$5,000 / ns

Round 2 will for the most part operate using the same infrastructure as round 1, including contest rules, evaluation process, offered resources, tool versions, etc. By minimizing change we hope to maximize continuity and efficiency for existing contestants.

Key steps/changes for round 2:

- Release (open source) the submitted models from round 1.
- Highly recommend the use of an MMCM to contestants to avoid clocking issues seen in round 1. Note the winning entry from round 1 does use an MMCM.
- Increase number of simultaneous jobs allowed per team in the testing portal.
- Emphasize that the RTL interface between the software, the MSU and the multiplier is allowed to change.
- Longer runtime (t=2^33?) for timing the results