NitroECC

Jonathan Harvey-Buschel, James Lovejoy
[jharveyb,jlovejoy]@mit.edu

2018/12/12

1 Introduction

As computer systems become more distributed, the need for fast and secure authentication in a trustless setting has increased dramatically. Specifically, Bitcoin is limited in transaction throughput by the speed at which the digital signatures it uses to validate payments can be validated on the devices it is intended to run on. In order to run a trustless Bitcoin instance on a mobile device, that device would be expected to validate all transaction signatures moving through the system. Thus, the Bitcoin community consciously limits the throughput of the network so that low-speed devices can realistically participate. This means your phone or a Raspberry Pi is capable of keeping up with the Bitcoin network, but that Bitcoin is limited to only around seven transactions per second (compared to in excess of forty-thousand for Visa).

We designed a Verilog core for performing arithmetic operations on the primitives used to perform signature generation and verification in Bitcoin and many other systems that use elliptic curve cryptography. We tested our system using the secp256k1 elliptic curve that Bitcoin uses but it is possible to use any Weierstrass curve. The core provides a 256-bit arithmetic logic unit for addition, subtraction, multiplication and division and takes the modulo of the former three with the curve prime. The core makes no assumptions about the particular curve or algorithms for your application and the prime number that addition, subtraction and multiply are modulo'ed over is a parameter.

We believe the core would be useful as part of a larger SoC project such as a Bitcoin full-node or wallet on a FPGA or ASIC. We designed the core to be as general as possible so that it both would be useful for applications other than Bitcoin, but also so that it can support future elliptic curve operations that are currently not in use on the Bitcoin network. Bitcoin presently uses ECDSA for signature verification but there are proposals to include support for the Schnorr signature algorithm on the live network in the future. While Schnorr uses the same elliptic curve and arithmetic primitives as ECDSA, the algorithms for signature generations and verification are different. Schnorr is not the only technology where this applies with new algorithms like zkSNARKs and confidential transactions also in the pipeline. Thus we hope our core will be continue to be useful in old hardware once there have been updates to the Bitcoin protocol, since backwards compatibility has always been a tenet of Bitcoin's design.

1.1 Elliptic curve arithmetic

When we wrote our project proposal we initially intended to use affine coordinates for representing the elliptic curve points used in the signature calculations. As such we produced an

accompanying C library for the project which implements point addition and multiplication in affine using OpenSSL's big number operations. These are the same operations that are available on our core and thus the code can help a user test and write code for the core.

However, while writing this library we quickly discovered that all signature operations would require multiple field inversions which in turn heavily use division. We have provided division on our core so that it is possible to use affine coordinates if desired, though this is not recommended due to the large performance hit. Instead we recommend using Jacobian coordinates which require no field inversions, only addition, subtraction and multiplication.

1.2 Division of labour

James wrote the math modules used in the final design, their test benches, the C implementation using OpenSSL, the processor state machine architecture and memory/stack management logic and the AXI-to-UART interface. Jonathan worked on some more pipelined math module designs using the DSPs on the FPGA, implemented the math input/output register logic in the processor and converted the processor to use 64-bit words rather than 8-bit words, as well as experimenting with faster methods of design like out-of-context synthesis.

1.3 Project outcomes

Although we were unable to complete our stretch goal of connecting the core to Bitcoin node software and using it for actual signature validation in time for the checkoff, this is certainly something that could be done with the core in the future once the development of driver software has taken place. Included with this writeup are example programs for the core for secp256k1 point doubling and Schnorr signature generation. Additionally an interface for the Xilinx AXI-UARTLite IP core that provides RS-232 communications via AXI is provided here, but was not working in time for checkoff. Nonetheless, the core provides embedded elliptic curve capability at low-power and would be an essential component of any IoT or mobile device that uses Bitcoin. Since we chose a more general, instruction-based design over the fixed point-addition and point-double modules described in our proposal, the core allows for a much larger scope of algorithms and curves, making it much more versatile than originally intended, at the cost of some performance reduction.

2 Instruction set

2.1 Architecture

NitroECC is a stack-based processor with separate stack and instruction memory. Operation codes and input data reside in the instruction memory and all temporary variables and outputs are stored in the stack memory. Instruction memory is read only and is decoded and executed sequentially from address 0 by the processor using a typical fetch-decode-execute cycle. The fetch-decode-execute cycle is not pipelined so every instruction takes a minimum of 2 cycles (some instructions do not need a separate execute phase). Each word of instruction memory is 64-bits wide, each word of stack memory is 256-bits wide and both

are unsigned. Thus a word of stack memory is equivalent to 4 words of instruction memory. This is to save space in the instruction code so opcodes only have to be 64-bits wide while satisfying the property of 256-bit wide math operations for elliptic curve points.

The processor does not have branching instructions and thus all instructions much be flat and cannot diverge in program flow. Therefore, for best performance it is recommended to batch many of the same type of operation (point double, point add, message sign etc) in a given program rather than run singular operations sequentially, relying on the host system for decision making, as performance will become limited by the communication speed with the host. In a production system one would load instruction memory in a pipeline with batches of data and instructions for a given operation and multiplex the instruction memory to a new operation and batch data when a previous batch has finished.

Each math operation has its own set of input and output registers. Thus the program flow to perform an arithmetic operation is to pop the inputs from the stack into the input registers for the given math operation, wait the number of cycles required for the math operation to complete and then push the result from the output registers onto the stack. Subtraction and addition operations only take 1 cycle to complete so there is no need to delay reading from their output registers. However, multiplication and division take 256 cycles to complete, so there are dedicated multiplication and division delay opcodes to pause the program while their output registers become valid. If a pop instruction to a math input register changes its value, the output registers for the operation immediately become invalid until the operation delay has elapsed. Multiplication, addition and subtraction all take place modulo the prime of the curve in use since the core is specialised for elliptic curve arithmetic. The prime can be set as a parameter in the source code if one different from the default is required.

The push and pop instruction to input and output registers do not change the stack pointer by themselves. Instead there are drop and forward instructions and decrement and increment the stack pointer by one word respectively. This feature makes navigating the stack to get the desired inputs much easier as stack values are not deleted unless overwritten by an OP_DATA or push instruction. There is no need to be concerned about where a particular value is on the stack when writing a program as one can use a sequence of OP_DROP and OP_FORWARD instructions to move the stack pointer to the desired data before using an OP_FORWARD sequence to return to the top of the stack.

The processor does not start until the "reset" input signal is made high for at least one cycle and then returned low. The processor begins executing from instruction memory address 0 once the "execute" input signal is brought high. Execution can be paused by making the execute signal low and resumed by making it high again. The processor continues executing until it reaches a stack-bounds error condition, a halt instruction or reaches the end of the instruction memory when it enters the halt state. In the halt state the "halt" output is asserted and the processor executes no more instructions irrespective of the value of the "execute" input. The halt state persists until the processor is reset using the "reset" input.

2.2 Instructions

2.2.1 OP_DATA (0x00, 6 cycles)

Copies the next 4 words of instruction data to the next word of stack data. This instruction is the only way to get inputs onto the stack.

2.2.2 OP_HALT (0x01, 3 cycles)

Halts the processor such that no more instructions are executed. Sets the "halt" output bit of the processor high.

2.2.3 OP_POPAA (0x02, 6 cycles)

Takes the top word of stack memory and puts it into the adder "A" input register. If the stack is empty, halts the processor.

2.2.4 OP_POPAB (0x03, 6 cycles)

Takes the top word of stack memory and puts it into the adder "B" input register. If the stack is empty, halts the processor.

2.2.5 OP_POPDA (0x04, 6 cycles)

Takes the top word of stack memory and puts it into the divider "A" input register (the dividend). If the stack is empty, halts the processor.

2.2.6 OP_POPDB (0x05, 6 cycles)

Takes the top word of stack memory and puts it into the divider "B" input register (the divisor). If the stack is empty, halts the processor.

2.2.7 OP_POPSA (0x06, 6 cycles)

Takes the top word of stack memory and puts it into the subtracter "A" input register (the number being subtracted from). If the stack is empty, halts the processor.

2.2.8 OP_POPSB (0x07, 6 cycles)

Takes the top word of stack memory and puts it into the subtracter "B" input register (the number being subtracted from A). If the stack is empty, halts the processor.

2.2.9 OP_POPMA (0x08, 6 cycles)

Takes the top word of stack memory and puts it into the multiplier "A" input register. If the stack is empty, halts the processor.

2.2.10 OP_POPMB (0x09, 6 cycles)

Takes the top word of stack memory and puts it into the multiplier "B" input register. If the stack is empty, halts the processor.

2.2.11 OP_DROP (0x0A, 2 cycles)

Drops the top word of stack memory by decrementing the stack pointer. Since the data is not gone until it is overwritten, it is still recoverable by a later call to OP_FORWARD. If the stack is empty, halts the processor.

2.2.12 OP_PUSHAO, (0x0B, 6 cycles)

Pushes the value in the output register of the adder onto the stack. If the stack is full, halts the processor.

2.2.13 OP_PUSHDQ, (0x0C, 6 cycles)

Pushes the value in the quotient output register of the divider onto the stack. If the stack is full, halts the processor.

2.2.14 OP_PUSHDR, (0x0D, 6 cycles)

Pushes the value in the remainder output register of the divider onto the stack. If the stack is full, halts the processor.

2.2.15 OP_PUSHSO, (0x0E, 6 cycles)

Pushes the value in the output register of the subtracter onto the stack. If the stack is full, halts the processor.

2.2.16 OP_PUSHMO, (0x0F, 6 cycles)

Pushes the value in the output register of the multiplier onto the stack. If the stack is full, halts the processor.

2.2.17 OP_FORWARD, (0x10, 2 cycles)

Restores the previously dropped stack value by incrementing the stack pointer. If the next stack value has not yet been used the value is undefined. If the stack is full, halts the processor.

2.2.18 OP_SWAP, (0x11, 18 cycles)

Swaps the values of the top two stack words. If there are fewer than two words on the stack, halts the processor.

2.2.19 OP_MUL, (0x12, 258 cycles)

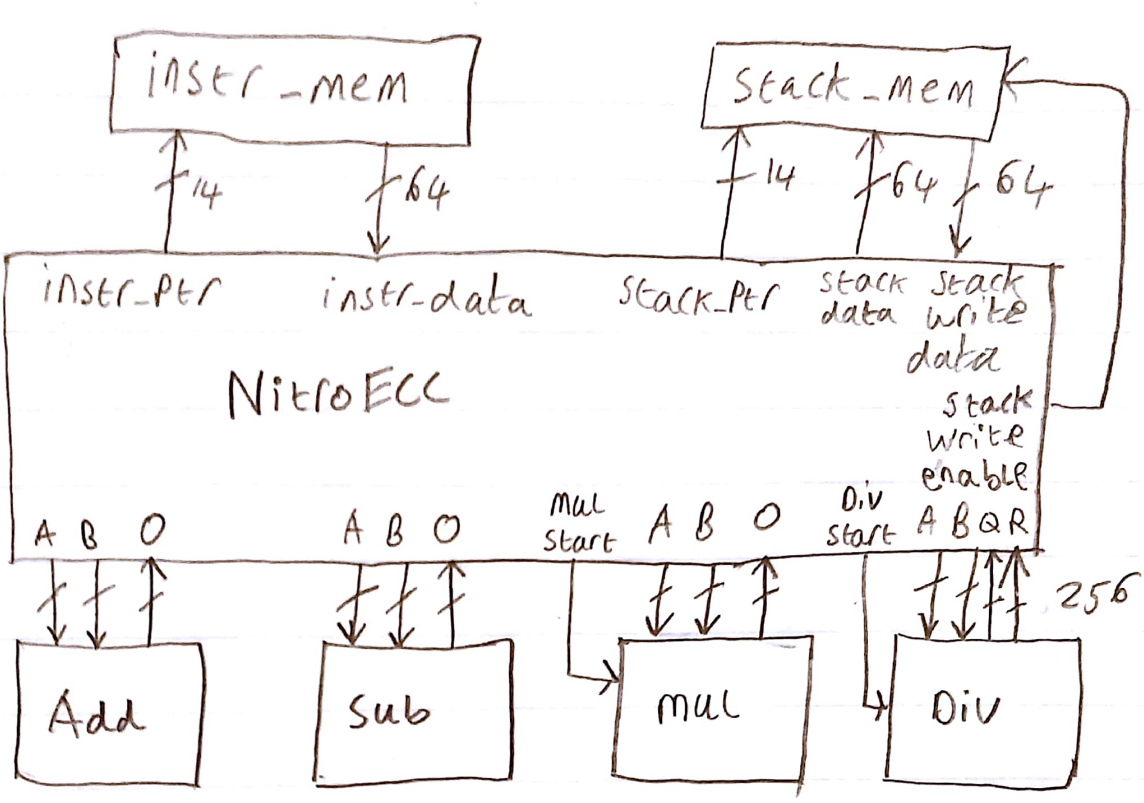
Starts the multiplier calculating on its current input register values and waits until its output register is valid.

2.2.20 OP_DIV, (0x13, 258 cycles)

Starts the divider calculating on its current input register values and waits until its output registers are valid.

3 Programming the accelerator

The general flow of programming the accelerator involves writing the instructions and data to the instruction memory, resetting the NitroECC module, setting the execute bit, waiting for a halt signal and finally reading the results from the stack memory. Instructions and data can be written at compile time by importing a COE file into Vivado or by some external circuit at runtime. It is important to ensure all programs end with an OP_HALT instruction, or the processor will continue executing until the end of the instruction memory which will likely contain random values that will be executed as instructions, and invalidate your outputs. Since all stack words are 256-bit wide, it is also important to ensure you pad smaller values with leading zeros (values are stored little-endian) to prevent alignment bugs in your programs.



4 Example programs

Each of the following examples provides a human readable version with each opcode on a line, as well as the raw bytecode representation. Each line represents a 64-bit instruction memory word.

4.1 Basic math operations

This example takes two big 256-bit numbers, pushes them onto the stack, then uses them as inputs to subtract and multiply operations, and pushes their results onto the stack. It also demonstrates how the OP_SWAP operation can be used to reverse the positions of values on the stack. The program leaves three values on the stack, the 0x0fd0... input, and the output from the subtraction followed by the output from the multiplication.

```
OP_DATA            // push 4 words onto the stack
0x09cc57f2ca39c2d8 // note the zero padding
0x1aed7e3d82af0b57
0x11863bd3403bb8f0
0x24c4c3b4ecf9652a
OP_DATA            // push 4 words onto the stack
0x0fd04ed02aef5778 // note the zero padding
0x9f1312d6817b6e9e
0x214fade46622a760
0xe692363e1843b3c2
OP_SWAP            // make 0x09cc... at the top of the stack
OP_POPSA           // push 0x09cc... into the subtract A reg
OP_POPMA           // push 0x09cc... into the mul A reg
OP_DROP            // move the stack ptr back to 0x0fd0...
OP_POPSB           // push 0x0fd0... into the subtract B reg
OP_POPMB           // push 0x0fd0... into the mul B reg
OP_PUSHSO          // push 0x09cc...-0x0fd0... onto the stack
OP_MUL             // start and wait for mul operation
OP_PUSHMO          // push 0x09cc...*0x0fd0... onto the stack
OP_HALT            // halt the processor
```

The above program in COE format can be found in the appendix.

4.2 Point double

This example program performs a point double on the secp256k1 elliptic curve in Jacobian coordinates which is a very common operation in any elliptic curve cryptography algorithm. The equations for the operation are as follows[1]:

$$S = 4XY^2$$

$$M = 3X^2 + Z^4$$

$$X' = M^2 - 2S$$
$$Y' = M(S - X') - 8Y^4$$
$$Z' = 2YZ$$

In our case for secp256k1, $Z = 1$ when starting from affine coordinates, so the second and last steps can be reduced to $M = 3X^2$ and $Z' = 2Y$ in that case. The code below handles the general case. The comments on each stack operation line show the current values on the stack immediately after that instruction has executed. Assume each line with a letter value such as X or numerical $0x1$ is padded to 256-bits or four instruction memory words and is preceded by an OP_DATA instruction.

```
Y          // Y
OP_POPMA   // Y
OP_POPMB   // Y
OP_MUL
OP_PUSHMO  // Y, Y^2
X          // Y, Y^2, X
OP_POPMA   // Y, Y^2, X
OP_SWAP    // Y, X, Y^2
OP_POPMB   // Y, X, Y^2
OP_MUL
OP_PUSHMO  // Y, X, Y^2, XY^2
0x4        // Y, X, Y^2, XY^2, 0x4
OP_POPMA   // Y, X, Y^2, XY^2, 0x4
OP_SWAP    // Y, X, Y^2, 0x4, XY^2
OP_POPMB   // Y, X, Y^2, 0x4, XY^2
OP_MUL
OP_PUSHMO  // Y, X, Y^2, 0x4, XY^2, S
X          // Y, X, Y^2, 0x4, XY^2, S, X
OP_POPMA
OP_POPMB
OP_MUL
OP_PUSHMO  // Y, X, Y^2, 0x4, XY^2, S, X, X^2
0x3        // Y, X, Y^2, 0x4, XY^2, S, X, X^2, 0x3
OP_POPMA
OP_SWAP    // Y, X, Y^2, 0x4, XY^2, S, X, 0x3, X^2
OP_POPMB
OP_MUL
OP_PUSHMO  // Y, X, Y^2, 0x4, XY^2, S, X, 0x3, X^2, M
OP_POPMA
OP_POPMB
OP_SWAP    // Y, X, Y^2, 0x4, XY^2, S, X, 0x3, M, X^2
OP_DROP    // Y, X, Y^2, 0x4, XY^2, S, X, 0x3, M
OP_SWAP    // Y, X, Y^2, 0x4, XY^2, S, X, M, 0x3
OP_DROP    // Y, X, Y^2, 0x4, XY^2, S, X, M
```

```
OP_SWAP   // Y, X, Y^2, 0x4, XY^2, S, M, X
OP_DROP   // Y, X, Y^2, 0x4, XY^2, S, M
OP_SWAP   // Y, X, Y^2, 0x4, XY^2, M, S
OP_MUL
OP_PUSHMO // Y, X, Y^2, 0x4, XY^2, M, S, M^2
OP_SWAP   // Y, X, Y^2, 0x4, XY^2, M, M^2, S
0x2       // Y, X, Y^2, 0x4, XY^2, M, M^2, S, 0x2
OP_POPMA
OP_SWAP   // Y, X, Y^2, 0x4, XY^2, M, M^2, 0x2, S
OP_POPMB
OP_SWAP   // Y, X, Y^2, 0x4, XY^2, M, M^2, S, 0x2
OP_DROP   // Y, X, Y^2, 0x4, XY^2, M, M^2, S
OP_SWAP   // Y, X, Y^2, 0x4, XY^2, M, S, M^2
OP_MUL
OP_PUSHMO // Y, X, Y^2, 0x4, XY^2, M, S, M^2, 2S
OP_POPSB
OP_SWAP   // Y, X, Y^2, 0x4, XY^2, M, S, 2S, M^2
OP_POPSA
OP_MUL
OP_PUSHSO // Y, X, Y^2, 0x4, XY^2, M, S, 2S, M^2, X'
OP_SWAP   // Y, X, Y^2, 0x4, XY^2, M, S, 2S, X', M^2
OP_DROP   // Y, X, Y^2, 0x4, XY^2, M, S, 2S, X'
OP_SWAP   // Y, X, Y^2, 0x4, XY^2, M, S, X', 2S
OP_DROP   // Y, X, Y^2, 0x4, XY^2, M, S, X'
OP_POPSB
OP_SWAP   // Y, X, Y^2, 0x4, XY^2, M, X', S
OP_POPSA
OP_DROP   // Y, X, Y^2, 0x4, XY^2, M, X'
OP_SWAP   // Y, X, Y^2, 0x4, XY^2, X', M
OP_PUSHSO // Y, X, Y^2, 0x4, XY^2, X', M, S - X'
OP_POPMA
OP_SWAP   // Y, X, Y^2, 0x4, XY^2, X', S - X', M
OP_POPMB
OP_MUL
OP_PUSHMO // Y, X, Y^2, 0x4, XY^2, X', S - X', M, M(S - X')
Y         // Y, X, Y^2, 0x4, XY^2, X', S - X', M, M(S - X'), Y
OP_POPMA
OP_POPMB
OP_MUL
OP_PUSHMO // Y, X, Y^2, 0x4, XY^2, X', S - X', M, M(S - X'), Y, Y^2
OP_POPMA
OP_MUL
OP_PUSHMO // Y, X, Y^2, 0x4, XY^2, X', S - X', M, M(S - X'), Y, Y^2, Y^3
OP_POPMA
OP_MUL
```

```
OP_PUSHMO // Y, X, Y^2, 0x4, XY^2, X', S - X', M, M(S - X'), Y, Y^2, Y^3, Y^4
0x8       // Y, X, Y^2, 0x4, XY^2, X', S - X', M, M(S - X'), Y, Y^2, Y^3, Y^4, 0x8
OP_POPMA
OP_DROP   // Y, X, Y^2, 0x4, XY^2, X', S - X', M, M(S - X'), Y, Y^2, Y^3, Y^4
OP_POPMB
OP_MUL
OP_PUSHMO // Y, X, Y^2, 0x4, XY^2, X', S - X', M, M(S - X'), Y, Y^2, Y^3, Y^4, 8Y^2
OP_SWAP   // Y, X, Y^2, 0x4, XY^2, X', S - X', M, M(S - X'), Y, Y^2, Y^3, 8Y^2, Y^4
OP_DROP   // Y, X, Y^2, 0x4, XY^2, X', S - X', M, M(S - X'), Y, Y^2, Y^3, 8Y^2
OP_SWAP   // Y, X, Y^2, 0x4, XY^2, X', S - X', M, M(S - X'), Y, Y^2, 8Y^2, Y^3
OP_DROP   // Y, X, Y^2, 0x4, XY^2, X', S - X', M, M(S - X'), Y, Y^2, 8Y^2
OP_SWAP   // Y, X, Y^2, 0x4, XY^2, X', S - X', M, M(S - X'), Y, 8Y^2, Y^2
OP_DROP   // Y, X, Y^2, 0x4, XY^2, X', S - X', M, M(S - X'), Y, 8Y^2
OP_SWAP   // Y, X, Y^2, 0x4, XY^2, X', S - X', M, M(S - X'), 8Y^2, Y
OP_DROP   // Y, X, Y^2, 0x4, XY^2, X', S - X', M, M(S - X'), 8Y^2
OP_POPSB
OP_DROP   // Y, X, Y^2, 0x4, XY^2, X', S - X', M, M(S - X')
OP_POPSA
OP_PUSHSO // Y, X, Y^2, 0x4, XY^2, X', S - X', M, M(S - X'), Y'
Z         // Y, X, Y^2, 0x4, XY^2, X', S - X', M, M(S - X'), Y', Z
Y         // Y, X, Y^2, 0x4, XY^2, X', S - X', M, M(S - X'), Y', Z, Y
0x2       // Y, X, Y^2, 0x4, XY^2, X', S - X', M, M(S - X'), Y', Z, Y, 0x2
OP_POPMA
OP_DROP   // Y, X, Y^2, 0x4, XY^2, X', S - X', M, M(S - X'), Y', Z, Y
OP_POPMB
OP_DROP   // Y, X, Y^2, 0x4, XY^2, X', S - X', M, M(S - X'), Y', Z
OP_MUL
OP_PUSHMO // Y, X, Y^2, 0x4, XY^2, X', S - X', M, M(S - X'), Y', Z, 2Y
OP_POPMA
OP_DROP   // Y, X, Y^2, 0x4, XY^2, X', S - X', M, M(S - X'), Y', Z
OP_POPMB
OP_MUL
OP_PUSHMO // Y, X, Y^2, 0x4, XY^2, X', S - X', M, M(S - X'), Y', Z, Z'
OP_HALT   // don't forget to halt!
```

The above program in COE format for calculating $2G$, converted to Jacobian from affine coordinates (by setting $Z = 1, X = X, Y = Y$) can be found in the appendix.

4.3 Schnorr signature generation

A Schnorr signature uses a more efficient elliptic curve algorithm than ECDSA and is currently being proposed for inclusion on the live Bitcoin network. NitroECC supports the algorithm and this example program demonstrates the creation of a Schnorr signature. The Schnorr signature algorithm given a message m, secret random number k and secret key a can generate a signature s as follows:

$$s = k - m * a$$

This translates into the following bytecode program:

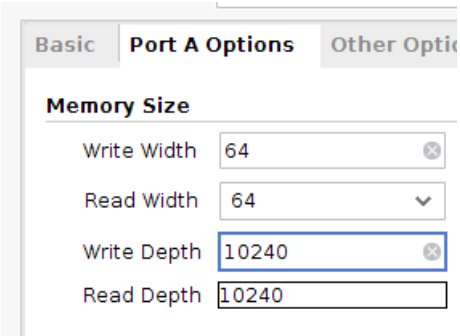
```
m          // padded to 256-bit and preceded by OP_DATA
a
OP_POPMA
OP_DROP
OP_POPMB
OP_MUL
OP_PUSHMO // now on the stack is: m, m*a
k
OP_POPSA
OP_DROP
OP_POPSB
OP_PUSHSO // now on the stack is: m, m*a, s
OP_HALT
```

The above program in COE format with dummy inputs can be found in the appendix.

5 Designing with the NitroECC module

5.1 Memory IP cores

The NitroECC module requires two BRAM IP cores for the stack and instruction memory. Both use a 64-bit word size and store 10,240 words by default, though the number of words is a parameter inside the core. The IP cores can be created from Vivado by opening the "IP Catalog" in the left pane under "Project Manager". Then in the directory search for "Block Memory Generator". Give the module a name in the "component name" field, leaving everything default on the "basic" pane. Click "Port A Options" and set the "Write Width and "Read Width" to 64-bits, and the "Write Depth" to 10,240, leaving everything else default. Finally click "OK" and create the IP core.



From the "Other Options" tab on the IP core creation window it is also possible to import the instructions into the instruction memory and have them included in the generated bitstream. Simply place the words of the program instructions into a COE file and select the file from the IP creation window. This is useful for testing where there is no external input source at runtime or if the set of instructions the core will be running is known and fixed ahead of time. In that case the instructions can be included in the memory as part of the bitstream at compile time, and only data words need to be rewritten at runtime.

5.2 Instantiating the module

Assuming you have created two BRAM IP cores in Vivado called "instr_mem" and "stack_mem" you would instantiate NitroECC using the following Verilog code snippet:

```
// these are wires as they're controlled by
// external modules
wire SYSCLK;
wire instr_ena;
wire stack_ena;
wire stack_we;
wire halt;

wire [13:0] instr_ptr, stack_ptr;
wire [63:0] instr_dout, stack_dout, stack_din;

// these are registers because we need to set
// them to 0
reg [63:0] instr_din;
reg instr_we;

// these are controlled by the host system
reg execute;
reg reset;

// instruction memory IP core
instr_mem instrs(.clka(SYSCLK),
                 .ena(instr_ena),
                 .wea(instr_we),
                 .dina(instr_din),
                 .douta(instr_dout),
                 .addra(instr_ptr));

// stack memory IP core
stack_mem sm(.clka(SYSCLK),
             .addra(stack_ptr),
             .ena(stack_ena),
```

```
            .wea(stack_we),
            .dina(stack_din),
            .douta(stack_dout));

// NitroECC core
bnvm vm(.clock(SYSCLK),
        .reset(reset),
        .instr_ptr(instr_ptr),
        .instr_ena(instr_ena),
        .instr_data(instr_dout),
        .stack_data(stack_dout),
        .stack_ptr_out(stack_ptr),
        .wstack_data(stack_din),
        .stack_ena(stack_ena),
        .stack_we(stack_we),
        .halt(halt),
        .execute(execute));
```

After implementation, timing analysis indicates the core system clock ("SYSCLK" in the above code snippet) can have a ≈25ns period in Vivado 2018.2 and uses approximately 23k LUTs.

From this point you would need to either load the data and instructions into the "instr_mem" module by using a COE file or writing to it using the "instr_we" and "instr_din" registers. Finally you would need to reset the NitroECC module and assert execute. Once the NitroECC module asserts "halt", the stack memory can be read for the program results.

Assuming you have a debounced reset input signal called "clean_reset":

```
always @(posedge SYSCLK) begin
    if(clean_reset) begin
        // set reset high to intialise NitroECC
        reset <= 1;
        // start with execution paused
        execute <= 0;
    end else if(!execute) begin
        // set reset back to low
        reset <= 0;
        // start executing instructions
        execute <= 1;
    end
end
```

6 Core internals

6.1 Math cores

Each elliptic curve math operation is composed from a sequence of arithmetic operations. Each math module has two 256-bit registers for input, and one for output. The multiplier and divider have an extra signalling bit for the NitroECC processor to start them and receive a signal when their output is present at their respective output registers.

We found that trying to share registers between math modules for input resulted in unreasonable synthesis and implementation times; we hypothesize that this is because routing becomes much more difficult, but regardless using independent registers did not add much complexity to the implementation of the math opcodes.

6.1.1 Adder and Subtracter

Both addition and subtraction are relatively small even for the 256-bit operands we were using; we found no need to use a custom design compared to what Vivado synthesized from just using the Verilog + and - operators. Both operations completed in a single cycle and the modules were not pipelined. We experimented with constructing an adder from multiple DSPs, but as discussed later in this paper this proved to be a challenge and not neccesary for improving the performance of our design.

6.1.2 Multiplier

The final design implements a Booth multiplier, which has a latency of 256 cycles but was also the most reliable design we found after trying a number of different options.

Using the Verilog multiplication operator return a large component that using 226 DSPs - however when we tried to implement this and use it on the board it did not return the correct results.

We spent a significant amount of time trying to use the DSPs manually for multiplication by implementing the Karatsuba algorithm. This technique involves separating each input into two halves, yielding four inputs half the size of the original.

$$x = x_1 * B^m + x_0$$

$$y = y_1 * B^m + y_0$$

This halves are then multiplied to produce four partial products, such that each half from one input is multiplied by both the halves of the second input.

$$z_2 = x_1 * y_1$$

$$z_1 = x_1 * y_0 + x_0 * y_1$$

$$z_0 = x_0 * y_0$$

The final result is a sum of these partial products, with each shifted by a multiple of the size of the halved input.

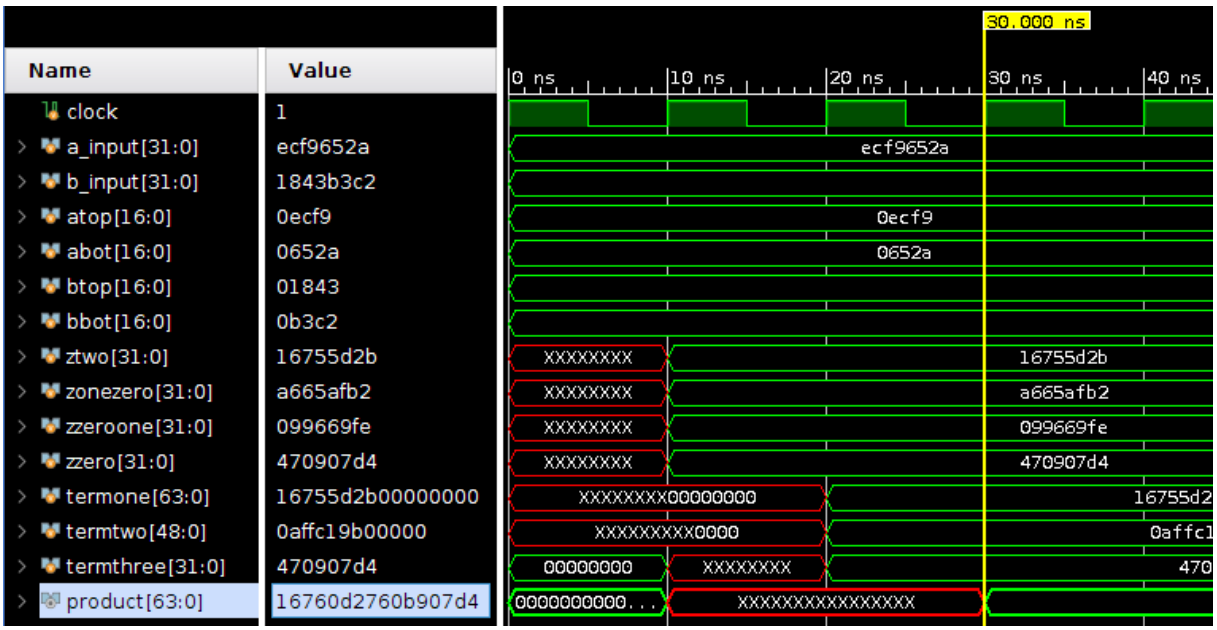
$$x * y = (x_1 * B^m + x_0) * (y_1 * B^m + y_0)$$
$$x * y = z_2 * B^{(2*m)} + z_1 * B^m + z_0$$

In Verilog, the unpipelined version translates to:

```
input clock,
input [31:0] a,
input [31:0] b,
output reg [63:0] out

parameter SIZE = 32;
parameter M = 16;
parameter SIZE_LESS = SIZE - 1;
parameter M_LESS = M - 1;
parameter OUT_SIZE_LESS = SIZE + SIZE_LESS;
parameter MID_SIZE = SIZE + M;

// partial products
reg [SIZE_LESS:0] ztwo = 0;
reg [SIZE_LESS:0] zonezero = 0;
reg [SIZE_LESS:0] zzeroone = 0;
reg [SIZE_LESS:0] zzero = 0;

// twobytemul is a DSP configured for 17x17 multiplication;
// The DSP requires a 3rd input, D, that is added to A
twobytemul top(.CLK(clock), .A({1'b0, a[31:16]}),
.B({1'b1, b[31:16]}), .D(2'b00) .P(ztwo));
twobytemul midone(.CLK(clock), .A({1'b0, a[31:16]}),
.B({1'b1, b[15:0]}), .D(2'b00), .P(zonezero));
twobytemul midzero(.CLK(clock), .A({1'b0, a[15:0]}),
.B({1'b1, b[31:16]}), .D(2'b00), .P(zzeroone));
twobytemul bottom(.CLK(clock), .A({1'b0, a[15:0]}),
.B({1'b1, b[15:0]}), .D(2'b00), .P(zzero));

// three operands used for computing output
reg [OUT_SIZE_LESS:0] termone = 0;
reg [MID_SIZE:0] termtwo = 0;
reg [SIZE_LESS:0] termthree = 0;

// 2 cycles, first for partials and second for combination
always @ (posedge clock) begin
        termone <= ztwo[M_LESS:0] << SIZE;
        termtwo <= ((zonezero[M_LESS:0] + zzeroone[M_LESS:0]) << M);
        termthree <= zzero[M_LESS:0];
```

```
        out <= termone + termtwo + termthree;
    end
```

In simulation the movement of data during the steps of a non-recursive Karatsuba multiplier is clearer:



This technique effectively halves the size of the multipliers needed, but requires 4 multipliers instead of one. The final combination of partial products requires a large adder and a few shifts, both of which require few resources on the FPGA. This algorithm can be applied recursively, with each stage using four times more multipliers than the one before. The DSP blocks support multiplication of inputs up to 18 bits wide, which for this application means a maximum input of 16 bits. We constructed a "base" 32-bit multiplier that used the DSPs for multiplication, and then built up to a 256-bit multiplier with 64-bit and 128-bit multipliers that used the smaller multiplier instead of the Verilog multiplication operator.

With an extra stage of pipelining for the DSPs, the 32-bit Karatsuba multiplier had a latency of 3 cycles; each level afterwards also took 3 stages, and the original 256-bit inputs took 4 cycles to reach the 32-bit multiplier, resulting in a total latency of 17 cycles.

This design should only require 256 DSPs, but after synthesis the module used 307 DSPs and more LUTs then the output of the Verilog multiplication operator. This design also did not function after implementation in hardware, so we had to result to the Booth multiplier.

6.1.3 Divider

Similar to the multiplier, the default Verilog operator also returned a module that consumed excessive resources. We switched to a sequential algorithm that take 256 cycles, but that used fewer LUTs. In the final version of the project, the divider is supported with OP_DIV and the appropriate registers, but is not used because of the switch to Jacobian coordinates.

6.2 Modulo

Originally we were planning to have a dedicated module for computing modulo; this function is essential for cryptography as the values need to stay within the field of the curve. The motivation for this separation was to save on resources compared to computing modulos in each math module. However, this function did not consume many LUTs and was therefore added to each math module to remove the complexity of routing all outputs through this one module.

6.3 Instruction and stack memory

// I thought we had 64-bit words for both stack and instruction memory?

Both instruction and stack memory was implemented with BRAM IP cores, which have a maximum size of 1 Mb, write port width of 4096 bits, read port width of 256 bits, and therefore a maximum address size of 20 bits. They also include a feature to write to individual bytes even when using widths larger than 8 bits, but this was not relevant for our use case.

Initially we had both memories configured for 8 bit reads and writes, but this resulted in reads or writes taking 32 cycles, as all of our data was 256 bits. We only planned to reserve up to 8 bits for opcodes, which meant using only 90% of our memory for data, assuming that 20% of the content was opcodes. To increase performance, we moved to 64 bit reads and writes. This led to opcodes with many leading zeroes, but since we were not pressed for space the performance increase was well worth it. Ideally we would use 256 bit reads and writes, but this was superceded by other performance optimizations such as the work on the multiplier.

7 Challenges

7.1 Using the DSPs

A major challenge we faced was correctly using the DSPs included on the Artix-7. Early on we realized that an efficient multiplier would use the onboard DSPs to save on LUTs, but the Artix on the DDR board we had used in class only included 240 DSPs. Since the default synthesized 256 bit multiplier used 226, we figured that using an FPGA with more DSPs would allow us to both actually implement our design as well as possibly accelerate other math operations. Since the DSPs can also be clocked higher than the FPGA fabric, we also wanted to explore using them at some multiple of the clock used for the fabric to further accelerate our math operations.

The only extra equipment we used for this project was the Nexys 4 Video, which uses a larger Artix-7 with slightly more LUTs and 740 DSPs. While we read the manual for the DSPs early on in the project, we did not take notice of a key detail; the DSP assumes that inputs are in two's complement form. All of our values were unisgned, so any use of the DSP where the input size and bus size were the same returned incorrect results. The multiplier inferred by Vivado also made this assumption, and we dismissed use of DSPs as an option. Much later, after re-reading the manual, we noticed this detail, which has a simple solution.

Padding any unsigned input with a leading zero will give the correct resuts, as long as the input and output widths are increased accoringly. However, we noticed this option too late and it did not make it into the final design.

7.2 Testing new programs

As mentioned, running a program on NitroECC involves initializing the BRAM for the instruction memory with the accompanying COE file. What we did not realize immediately was that the BRAM instance was tied to the COE file used during synthesis; we tried to edit the COE file and then test the design without re-synthesizing the IP core, but the output had not changed. After some time searching for a reason for this, we found that any change in the COE required re-synthesizing the IP core and subsequently the whole design. While this synthesis was relatively quick, it became a significant portion of our time spent waiting for synthesis and implementation to run before being able to test a new program on the board itself.

A better solution for loading the program would be to use UART and send it to the FPGA via another machine. but as our stretch goal we wanted to focus on having a working design first. In retrospect, it is unclear which choice would have been quicker, but we had many problems using the build in USB-to-Serial port and converter on the board.

7.3 Vivado quirks

Throughout the project, we had multiple problems with Vivado that slowed down our development. At one point, one of our attempts to synthesize a design consumed all of the RAM and swap space on a lab machine before hanging and eventually crashing. As a result, we switched to only using our personal machines for synthesis and implementation. We also switched to Vivado 2018.2, since that was what we already had on our machines. It is unclear if this made a difference on how well optimized our design was, but at times it would not recognize the use of the system clock, or request timing constraints for pins with no delay, such as switches and LEDs. With the respect to the attempt to use an AXI IP core for UART communication, at times Vivado would delete the core because some inputs which were unused were set to the constant 0. We also had to use macros such as {* "DONT_TOUCH" = true *} and {* "use_dsp" = yes *} to force Vivado to not optimize away modules or to enforce usage of DSPs in a module.

8 References

1. https://en.wikibooks.org/wiki/Cryptography/Prime_Curve/Jacobian_Coordinates
2. https://www.xilinx.com/support/documentation/sw_manuals/xilinx2012_2/ug901-vivado-synthesis.pdf
3. http://mathworld.wolfram.com/KaratsubaMultiplication.html

4. https://www.xilinx.com/support/documentation/user_guides/ug479_7Series_DSP48E1.pdf

A Example programs

A.1 Basic math operations

```
memory_initialization_radix=16;
memory_initialization_vector=
0000000000000000,
09cc57f2ca39c2d8,
1aed7e3d82af0b57,
11863bd3403bb8f0,
24c4c3b4ecf9652a,
0000000000000000,
0fd04ed02aef5778,
9f1312d6817b6e9e,
214fade46622a760,
e692363e1843b3c2,
0000000000000011,
0000000000000006,
0000000000000008,
000000000000000A,
0000000000000007,
0000000000000009,
000000000000000E,
0000000000000012,
000000000000000F,
0000000000000001;
```

A.2 Point double

```
memory_initialization_radix=16;
memory_initialization_vector=
0000000000000000,
483ADA7726A3C465,
5DA4FBFC0E1108A8,
FD17B448A6855419,
9C47D08FFB10D4B8,
0000000000000008,
0000000000000009,
0000000000000012,
000000000000000F,
0000000000000000,
```

```
79BE667EF9DCBBAC,
55A06295CE870B07,
029BFCDB2DCE28D9,
59F2815B16F81798,
0000000000000008,
0000000000000011,
0000000000000009,
0000000000000012,
000000000000000F,
0000000000000000,
0000000000000000,
0000000000000000,
0000000000000000,
0000000000000004,
0000000000000008,
0000000000000011,
0000000000000009,
0000000000000012,
000000000000000F,
0000000000000000,
79BE667EF9DCBBAC,
55A06295CE870B07,
029BFCDB2DCE28D9,
59F2815B16F81798,
0000000000000008,
0000000000000009,
0000000000000012,
000000000000000F,
0000000000000000,
0000000000000000,
0000000000000000,
0000000000000000,
0000000000000003,
0000000000000008,
0000000000000011,
0000000000000009,
0000000000000012,
000000000000000F,
0000000000000008,
0000000000000009,
0000000000000011,
000000000000000A,
0000000000000011,
000000000000000A,
0000000000000011,
```

000000000000000A,
0000000000000011,
0000000000000012,
000000000000000F,
0000000000000011,
0000000000000000,
0000000000000000,
0000000000000000,
0000000000000000,
0000000000000002,
0000000000000008,
0000000000000011,
0000000000000009,
0000000000000011,
000000000000000A,
0000000000000011,
0000000000000012,
000000000000000F,
0000000000000007,
0000000000000011,
0000000000000006,
0000000000000012,
000000000000000E,
0000000000000011,
000000000000000A,
0000000000000011,
000000000000000A,
0000000000000007,
0000000000000011,
0000000000000006,
000000000000000A,
0000000000000011,
000000000000000E,
0000000000000008,
0000000000000011,
0000000000000009,
0000000000000012,
000000000000000F,
0000000000000000,
483ADA7726A3C465,
5DA4FBFC0E1108A8,
FD17B448A6855419,
9C47D08FFB10D4B8,
0000000000000008,
0000000000000009,

0000000000000012,
000000000000000F,
0000000000000008,
0000000000000012,
000000000000000F,
0000000000000008,
0000000000000012,
000000000000000F,
0000000000000000,
0000000000000000,
0000000000000000,
0000000000000000,
0000000000000008,
0000000000000008,
000000000000000A,
0000000000000009,
0000000000000012,
000000000000000F,
0000000000000011,
000000000000000A,
0000000000000011,
000000000000000A,
0000000000000011,
000000000000000A,
0000000000000011,
000000000000000A,
0000000000000007,
000000000000000A,
0000000000000006,
000000000000000E,
0000000000000000,
0000000000000000,
0000000000000000,
0000000000000000,
0000000000000001,
0000000000000000,
483ADA7726A3C465,
5DA4FBFC0E1108A8,
FD17B448A6855419,
9C47D08FFB10D4B8,
0000000000000000,
0000000000000000,
0000000000000000,
0000000000000000,
0000000000000002,

```
0000000000000008,
000000000000000A,
0000000000000009,
000000000000000A,
0000000000000012,
000000000000000F,
0000000000000008,
000000000000000A,
0000000000000009,
0000000000000012,
000000000000000F,
0000000000000001;
```

A.3 Schnorr signature generation

```
memory_initialization_radix=16;
memory_initialization_vector=
0000000000000000,
09cc57f2ca39c2d8,
1aed7e3d82af0b57,
11863bd3403bb8f0,
24c4c3b4ecf9652a,
0000000000000000,
0fd04ed02aef5778,
9f1312d6817b6e9e,
214fade46622a760,
e692363e1843b3c2,
0000000000000008,
000000000000000A,
0000000000000009,
0000000000000012,
000000000000000F,
0000000000000000,
483ADA7726A3C465,
5DA4FBFC0E1108A8,
FD17B448A6855419,
9C47D08FFB10D4B8,
0000000000000006,
000000000000000A,
0000000000000007,
000000000000000E,
0000000000000001;
```

B Source code

B.1 labkit.v

```
‘timescale 1ns / 1ps

module labkit(
   input SYSCLK,
   input [7:0] SW,
   input uart_tx_in,
   output uart_rx_out,
   output[7:0] LED
   );

   parameter MEM_BUS = 63;

   reg start, reset, tx_ready, execute, instr_we;
   wire ready, stack_we, halt;

   wire instr_ena, stack_ena;

   wire [13:0] instr_ptr, stack_ptr;
   wire [MEM_BUS:0] instr_dout, stack_dout;

   wire [MEM_BUS:0] stack_din;
   reg [MEM_BUS:0] instr_din;

   reg [7:0] led_out;

   assign LED[7:0] = led_out;

   wire clean_reset, clean_execute;

   debounce dbreset(.clock(SYSCLK), .reset(SW[7]), .noisy(SW[7]),
                    .clean(clean_reset));
   debounce dbexecute(.clock(SYSCLK), .reset(SW[7]), .noisy(SW[6]),
                      .clean(clean_execute));

   wire [4:0] clean_sw;

   debounce dbs0(.clock(SYSCLK), .reset(SW[7]), .noisy(SW[0]), .clean(clean_sw[0]));
   debounce dbs1(.clock(SYSCLK), .reset(SW[7]), .noisy(SW[1]), .clean(clean_sw[1]));
   debounce dbs2(.clock(SYSCLK), .reset(SW[7]), .noisy(SW[2]), .clean(clean_sw[2]));
   debounce dbs3(.clock(SYSCLK), .reset(SW[7]), .noisy(SW[3]), .clean(clean_sw[3]));
   debounce dbs4(.clock(SYSCLK), .reset(SW[7]), .noisy(SW[4]), .clean(clean_sw[4]));
```

```
always @(posedge SYSCLK) begin
     instr_we <= 0;
     instr_din <= 64'd0;
     if(clean_reset) begin
         execute <= 0;
     end else if(clean_execute && !execute) begin
         execute <= 1;
     end else if(halt && execute) begin
         execute <= 0;
     end

     case (clean_sw[4:0])
         0: led_out <= stack_dout[7:0];
         1: led_out <= stack_dout[15:8];
         2: led_out <= stack_dout[23:16];
         3: led_out <= stack_dout[31:24];
         4: led_out <= stack_dout[39:32];
         5: led_out <= stack_dout[47:40];
         6: led_out <= stack_dout[55:48];
         7: led_out <= stack_dout[63:56];
         8: led_out <= stack_ptr[7:0];
         9: led_out <= {2'b00, stack_ptr[13:8]};
         10: led_out <= instr_dout[7:0];
         11: led_out <= instr_dout[15:8];
         12: led_out <= instr_dout[23:16];
         13: led_out <= instr_dout[31:24];
         14: led_out <= instr_dout[39:32];
         15: led_out <= instr_dout[47:40];
         16: led_out <= instr_dout[55:48];
         17: led_out <= instr_dout[63:56];
         18: led_out <= instr_ptr[7:0];
         19: led_out <= {2'b00, instr_ptr[13:8]};
         20: led_out <= {6'd0, execute, halt};
         default: begin end
     endcase
end

instr_mem instrs(.clka(SYSCLK),
                 .ena(instr_ena),
                 .wea(instr_we),
                 .dina(instr_din),
                 .douta(instr_dout),
                 .addra(instr_ptr));
stack_mem sm(.clka(SYSCLK),
```

```
                .addra(stack_ptr),
                .ena(stack_ena),
                .wea(stack_we),
                .dina(stack_din),
                .douta(stack_dout));

   bnvm vm(.clock(SYSCLK),
           .reset(clean_reset),
           .instr_ptr(instr_ptr),
           .instr_ena(instr_ena),
           .instr_data(instr_dout),
           .stack_data(stack_dout),
           .stack_ptr_out(stack_ptr),
           .wstack_data(stack_din),
           .stack_ena(stack_ena),
           .stack_we(stack_we),
           .halt(halt),
           .execute(execute)
           );

endmodule
```

B.2 bnvm.v

```
`timescale 1ns / 1ps

// Specify width of instr_data manually to avoid changing declaration of bnvm

module bnvm(
    input clock,
    input reset,
    input [63:0] instr_data,
    output [13:0] stack_ptr_out,
    output reg instr_ena,
    output reg [13:0] instr_ptr,
    input [63:0] stack_data,
    output [63:0] wstack_data,
    output stack_ena,
    output stack_we,
    output reg halt,
    input execute
    );

    // using LESS / MORE params to avoid multiple additions / subtractions on some lines - may wo
    // was easier to convert this way
```

```
parameter MEM_BUS = 63;
parameter MEM_BUS_MORE = MEM_BUS+1;
parameter MEM_BYTES = MEM_BUS_MORE/8;
parameter MEM_CYCLES = 256/(MEM_BUS+1);
parameter MEM_CYCLES_LESS = MEM_CYCLES-1;
parameter MEM_CYCLES_MORE = MEM_CYCLES+1;

parameter MUL_DELAY = 256;
parameter DIV_DELAY = 256;

parameter STACK_SIZE = 14'd10240;

// 1 byte bus width -> 32 cycles
// for loss in useful memory due to empty bits in opcodes: (1-((opcode frequency)*(1-(opcode
// assuming 4-bit opcodes & 20% opcodes, with 1 byte bus, loss is 10%, but tapers quickly;
// 17.5% for 4 bytes, ~19% for 32, ~20% for 64 (!)

parameter BEGIN = 0;
parameter FETCH_INSTRUCTION = 1;
parameter DECODE_INSTRUCTION = 2;
parameter FETCH_DATA = 3;
parameter HALT = 4;
parameter STACK_POP = 5;
parameter STACK_PUSH = 6;
parameter STACK_SWAP = 7;
parameter STACK_SWAP_CLOSE = 8;
parameter MUL_WAIT = 9;
parameter DIV_WAIT = 10;

parameter OP_DATA = {MEM_CYCLES{8'h00}};
parameter OP_HALT = {8'h01};
parameter OP_POPAA = {{MEM_CYCLES_LESS{8'h00}}, 8'h02};
parameter OP_POPAB = {{MEM_CYCLES_LESS{8'h00}}, 8'h03};
parameter OP_POPDA = {{MEM_CYCLES_LESS{8'h00}}, 8'h04};
parameter OP_POPDB = {{MEM_CYCLES_LESS{8'h00}}, 8'h05};
parameter OP_POPSA = {{MEM_CYCLES_LESS{8'h00}}, 8'h06};
parameter OP_POPSB = {{MEM_CYCLES_LESS{8'h00}}, 8'h07};
parameter OP_POPMA = {{MEM_CYCLES_LESS{8'h00}}, 8'h08};
parameter OP_POPMB = {{MEM_CYCLES_LESS{8'h00}}, 8'h09};
parameter OP_DROP = {{MEM_CYCLES_LESS{8'h00}}, 8'h0A};
parameter OP_PUSHAO = {{MEM_CYCLES_LESS{8'h00}}, 8'h0B};
parameter OP_PUSHDQ = {{MEM_CYCLES_LESS{8'h00}}, 8'h0C};
parameter OP_PUSHDR = {{MEM_CYCLES_LESS{8'h00}}, 8'h0D};
parameter OP_PUSHSO = {{MEM_CYCLES_LESS{8'h00}}, 8'h0E};
```

```
parameter OP_PUSHMO = {{MEM_CYCLES_LESS{8'h00}}, 8'h0F};
parameter OP_FORWARD = {{MEM_CYCLES_LESS{8'h00}}, 8'h10};
parameter OP_SWAP = {{MEM_CYCLES_LESS{8'h00}}, 8'h11};
parameter OP_MUL = {{MEM_CYCLES_LESS{8'h00}}, 8'h12};
parameter OP_DIV = {{MEM_CYCLES_LESS{8'h00}}, 8'h13};

reg [13:0] stack_ptr;
reg ena, we;

wire [MEM_BUS:0] dout;
wire [MEM_BUS:0] din;

reg din_source;
reg [3:0] input_select;

reg [MEM_BUS:0] push_data;
reg [255:0] push_tmp, push_tmp2;

assign din = din_source ? push_data : instr_data;

assign dout = stack_data;
assign wstack_data = din;
assign stack_ena = ena;
assign stack_we = we;
assign stack_ptr_out = stack_ptr;

reg [3:0] state;
reg [8:0] load_counter;

reg [255:0] AA, AB, DA, DB, SA, SB, MA, MB;
wire [255:0] AO, DQ, DR, SO, MO;
reg mulstart, divstart;
wire mulready, divready;

mod_add add(.clock(clock), .a(AA), .b(AB), .out(AO));
div div(.clock(clock), .start(divstart), .ready(divready), .a(DA),
.b(DB), .q(DQ), .r(DR));
mod_sub sub(.clock(clock), .a(SA), .b(SB), .out(SO));
mul_seq mul(.clock(clock), .start(mulstart), .ready(mulready),
.multiplicand(MA), .multiplier(MB), .out(MO));

always @(posedge clock) begin
    if (reset) begin
        state <= BEGIN;
    end else if(execute) begin
```

```
case (state)
    BEGIN: begin
        halt <= 0;
        ena <= 0;
        we <= 0;
        stack_ptr <= {14{1'b1}};
        instr_ptr <= 0;
        instr_ena <= 1;
        mulstart <= 0;
        divstart <= 0;
        AA <= 0;
        AB <= 0;
        DA <= 0;
        DB <= 0;
        SA <= 0;
        SB <= 0;
        MA <= 0;
        MB <= 0;
        state <= FETCH_INSTRUCTION;
    end

    FETCH_INSTRUCTION: begin
        // increment the stack pointer and move to the
        // decode instruction state
        we <= 0;
        instr_ena <= 1;
        instr_ptr <= instr_ptr + 1;
        if(instr_ptr == STACK_SIZE-1) begin
            state <= HALT;
        end else begin
            state <= DECODE_INSTRUCTION;
        end
    end

    DECODE_INSTRUCTION: begin
        // in this state dout now contains an instruction
        case (instr_data)
            OP_DATA: begin
                state <= FETCH_DATA;
                load_counter <= 0;
                instr_ptr <= instr_ptr + 1;
            end

            OP_HALT: begin
                state <= HALT;
```

```
end

OP_POPAA, OP_POPAB, OP_POPDA, OP_POPDB, OP_POPSA, OP_POPSB,
OP_POPMA, OP_POPMB: begin
    // assuming that opcode values for pops are contiguous
    // and start at 2
    input_select <= instr_data - 2'b10;
    load_counter <= 0;

    if (stack_ptr < MEM_CYCLES_LESS) begin
        state <= HALT;
    end else begin
        state <= STACK_POP;
    end

    stack_ptr <= stack_ptr - MEM_CYCLES_LESS;
end

OP_PUSHAO, OP_PUSHDQ, OP_PUSHDR, OP_PUSHSO, OP_PUSHMO: begin
    // here contiguous opcode values don't help since we have to select
    // before starting the pushes
    load_counter <= 0;
    case (instr_data)
        OP_PUSHAO: push_tmp <= AO;
        OP_PUSHDQ: push_tmp <= DQ;
        OP_PUSHDR: push_tmp <= DR;
        OP_PUSHSO: push_tmp <= SO;
        OP_PUSHMO: push_tmp <= MO;
    endcase
    if (stack_ptr == STACK_SIZE-1) begin
        state <= HALT;
    end else begin
        state <= STACK_PUSH;
    end
end

OP_DROP: begin
    // OP_POP* without the popping
    if (stack_ptr < MEM_CYCLES_LESS) begin
        state <= HALT;
    end else begin
        stack_ptr <= stack_ptr - MEM_CYCLES;
        state <= FETCH_INSTRUCTION;
    end
end
```

```
        OP_FORWARD: begin
            // Inverse of OP_DROP; limiting access to top of stack
            // via magic number here
            if (stack_ptr == STACK_SIZE-1) begin
                state <= HALT;
            end else begin
                stack_ptr <= stack_ptr + MEM_CYCLES;
                state <= FETCH_INSTRUCTION;
            end
        end

        OP_MUL: begin
            load_counter <= 0;
            state <= MUL_WAIT;
        end

        OP_DIV: begin
            load_counter <= 0;
            state <= DIV_WAIT;
        end

        default: begin
            state <= HALT;
        end

        OP_SWAP: begin
            load_counter <= 0;
            if (stack_ptr < (2*MEM_CYCLES_LESS)-1) begin
                state <= HALT;
            end else begin
                stack_ptr <= stack_ptr - (MEM_CYCLES_LESS*2)-1;
                state <= STACK_SWAP;
            end
        end
    endcase
end

STACK_SWAP: begin
    ena <= 1;

    if(load_counter < (MEM_CYCLES*2)) begin
        stack_ptr <= stack_ptr + 1;
    end
```

```
    if (load_counter < MEM_CYCLES_MORE*2) begin
        if(load_counter > MEM_CYCLES_MORE) begin
            push_tmp[MEM_BUS:0] <= dout;
            push_tmp[255:MEM_BUS_MORE] <= push_tmp[255-MEM_BUS_MORE:0];
        end else if(load_counter > 1) begin
            push_tmp2[MEM_BUS:0] <= dout;
            push_tmp2[255:MEM_BUS_MORE] <= push_tmp2[255-MEM_BUS_MORE:0];
        end
    end

    if (load_counter == MEM_CYCLES_MORE*2) begin
        state <= STACK_SWAP_CLOSE;
        load_counter <= 0;
        din_source <= 1;
        stack_ptr <= stack_ptr - MEM_CYCLES*2;
    end else begin
        load_counter <= load_counter + 1;
    end
end

STACK_SWAP_CLOSE: begin
    ena <= 1;
    we <= 1;

    if(load_counter > MEM_CYCLES_LESS) begin
        push_data <= push_tmp2[255:255-MEM_BUS];
        push_tmp2[255:MEM_BUS_MORE] <= push_tmp2[255-MEM_BUS_MORE:0];
    end else begin
        push_data <= push_tmp[255:255-MEM_BUS];
        push_tmp[255:MEM_BUS_MORE] <= push_tmp[255-MEM_BUS_MORE:0];
    end

    if (load_counter == MEM_CYCLES*2) begin
        we <= 0;
        state <= FETCH_INSTRUCTION;
    end else begin
        load_counter <= load_counter + 1;

        if(load_counter > 0) begin
            stack_ptr <= stack_ptr + 1;
        end
    end
end

FETCH_DATA: begin
```

```
    // copy 32 bytes from the instruction memory to
    // the stack

    we <= 1;
    din_source <= 0;
    ena <= 1;
    stack_ptr <= stack_ptr + 1;

    if (load_counter == MEM_CYCLES_LESS)
        state <= FETCH_INSTRUCTION;
    else begin
        load_counter <= load_counter + 1;
        instr_ptr <= instr_ptr + 1;
    end
end

HALT: begin
    halt <= 1;
end

STACK_POP: begin
    ena <= 1;

    if(load_counter < MEM_CYCLES_LESS) begin
        stack_ptr <= stack_ptr + 1;
    end

    if (load_counter > 1) begin
        case (input_select)
            0: begin
                AA[MEM_BUS:0] <= dout;
                AA[255:MEM_BUS_MORE] <= AA[255-MEM_BUS_MORE:0];
            end
            1: begin
                AB[MEM_BUS:0] <= dout;
                AB[255:MEM_BUS_MORE] <= AB[255-MEM_BUS_MORE:0];
            end
            2: begin
                DA[MEM_BUS:0] <= dout;
                DA[255:MEM_BUS_MORE] <= DA[255-MEM_BUS_MORE:0];
            end
            3: begin
                DB[MEM_BUS:0] <= dout;
                DB[255:MEM_BUS_MORE] <= DB[255-MEM_BUS_MORE:0];
            end
```

```
            4: begin
                SA[MEM_BUS:0] <= dout;
                SA[255:MEM_BUS_MORE] <= SA[255-MEM_BUS_MORE:0];
            end
            5: begin
                SB[MEM_BUS:0] <= dout;
                SB[255:MEM_BUS_MORE] <= SB[255-MEM_BUS_MORE:0];
            end
            6: begin
                MA[MEM_BUS:0] <= dout;
                MA[255:MEM_BUS_MORE] <= MA[255-MEM_BUS_MORE:0];
            end
            7: begin
                MB[MEM_BUS:0] <= dout;
                MB[255:MEM_BUS_MORE] <= MB[255-MEM_BUS_MORE:0];
            end
        endcase
    end

    if (load_counter == MEM_CYCLES_MORE) begin
        state <= FETCH_INSTRUCTION;
    end else begin
        load_counter <= load_counter + 1;
    end
end

STACK_PUSH: begin
    ena <= 1;
    we <= 1;
    din_source <= 1;
    stack_ptr <= stack_ptr + 1;

    push_data <= push_tmp[255:255-MEM_BUS];
    push_tmp[255:MEM_BUS_MORE] <= push_tmp[255-MEM_BUS_MORE:0];

    if (load_counter == MEM_CYCLES_LESS) begin
        state <= FETCH_INSTRUCTION;
    end else begin
        load_counter <= load_counter + 1;
    end
end

MUL_WAIT: begin
    if (load_counter == 0)
        mulstart <= 1;
```

```
                    else
                        mulstart <= 0;

                    if (load_counter == MUL_DELAY)
                        state <= FETCH_INSTRUCTION;
                    else
                        load_counter <= load_counter + 1;
                end

                DIV_WAIT: begin
                    if (load_counter == 0)
                        divstart <= 1;
                    else
                        divstart <= 0;

                    if (load_counter == DIV_DELAY)
                        state <= FETCH_INSTRUCTION;
                    else
                        load_counter <= load_counter + 1;
                end

            endcase
        end
    end

endmodule
```

B.3 axi_uart_manager.v

```
`timescale 1ns / 1ps

module axi_uart_manager(
    input clock,
    input reset,
    input uart_in,
    input [7:0] tx_byte,
    input tx_data,
    output uart_out,
    output reg [7:0] rx_byte,
    output reg rx_data,
    output reg tx_full
    );

    wire interrupt, arready, rvalid, bvalid, wready, awready;
    wire [31:0] rdata;
```

```
reg [31:0] wdata;

reg arvalid, rready, bready, awvalid, wvalid;

wire [1:0] bresp, rresp;

reg [3:0] araddr, awaddr, wstrb;

axi_uartlite_0 axiuart(.s_axi_aclk(clock),
                       .s_axi_aresetn(reset),
                       .interrupt(interrupt),
                       .rx(uart_in),
                       .tx(uart_out),
                       .s_axi_araddr(araddr),
                       .s_axi_arvalid(arvalid),
                       .s_axi_arready(arready),
                       .s_axi_rready(rready),
                       .s_axi_rdata(rdata),
                       .s_axi_rvalid(rvalid),
                       .s_axi_awaddr(awaddr),
                       .s_axi_awvalid(awvalid),
                       .s_axi_bready(bready),
                       .s_axi_wdata(wdata),
                       .s_axi_wstrb(wstrb),
                       .s_axi_wvalid(wvalid),
                       .s_axi_awready(awready),
                       .s_axi_bresp(bresp),
                       .s_axi_bvalid(bvalid),
                       .s_axi_rresp(rresp),
                       .s_axi_wready(wready));

parameter START_READ = 0;
parameter HANDSHAKE_WAIT = 1;
parameter HANDSHAKE_DONE = 2;
parameter CHECK_DATA = 3;
parameter POP_FIFO = 4;
parameter PUSH_FIFO = 5;

parameter STAT_REG = 4'h08;
parameter TX_FIFO_REG = 4'h04;
parameter RX_FIFO_REG = 4'h00;

parameter OKAY = 2'b00;
```

```
parameter RX_FIFO_DATA = 0;
parameter TX_FIFO_FULL = 3;

reg [2:0] rx_state, tx_state;

reg reading;

reg [31:0] out_data;

always @(posedge clock) begin
    if(!reset) begin
        out_data[31:0] <= 32'd0;
        rx_state[2:0] <= START_READ;
        araddr[3:0] <= 4'd0;
        arvalid <= 1'b0;
        rready <= 1'b0;
        awaddr[3:0] <= 4'd0;
        bready <= 1'b0;
        wstrb[3:0] <= 4'd0;
        awvalid <= 1'b0;
        rx_byte[7:0] <= 8'd0;
        wdata[31:0] <= 32'b0;
        wvalid <= 1'b0;
        rx_data <= 1'b0;
    end else begin
        case (rx_state)
            START_READ, POP_FIFO: begin;
                arvalid <= 1;
                rready <= 1;
                if(rx_state == START_READ) begin
                    araddr <= STAT_REG;
                    reading <= 0;
                end else begin
                    araddr <= RX_FIFO_REG;
                    reading <= 1;
                end
                rx_state <= HANDSHAKE_WAIT;
            end

            HANDSHAKE_WAIT: begin
                if(arready) begin
                    arvalid <= 0;
                    rx_state <= HANDSHAKE_DONE;
                end
            end
```

```
    HANDSHAKE_DONE: begin
        if(rvalid) begin
            rready <= 0;
            out_data <= rdata;
            rx_state <= CHECK_DATA;
        end

        if(rresp != OKAY) begin
            reading <= 0;
        end
    end

    CHECK_DATA: begin
        if(reading) begin
            rx_state <= START_READ;
            rx_byte <= out_data[7:0];
            rx_data <= 1;
        end else if(out_data[RX_FIFO_DATA]) begin
            rx_state <= POP_FIFO;
        end else begin
            rx_state <= START_READ;
        end

        if(!out_data[TX_FIFO_FULL] && tx_full) begin
            tx_full <= 0;
        end
    end
endcase

if(rx_data) begin
    rx_data <= 0;
end

case(tx_state)
    PUSH_FIFO: begin
        if(tx_data && !tx_full) begin
            awaddr <= TX_FIFO_REG;
            wdata <= {24'd0, tx_byte[7:0]};
            awvalid <= 1;
            wvalid <= 1;
            bready <= 1;
            tx_state <= HANDSHAKE_WAIT;
        end
    end
```

```
                HANDSHAKE_WAIT: begin
                    if(awready && wready) begin
                        awvalid <= 0;
                        wvalid <= 0;
                        tx_state <= HANDSHAKE_DONE;
                    end
                end

                HANDSHAKE_DONE: begin
                    if(bvalid) begin
                        bready <= 0;
                        if(bresp != OKAY) begin
                            tx_full <= 1;
                        end
                        tx_state <= PUSH_FIFO;
                    end
                end
            endcase
        end
    end

endmodule
```

B.4 mul_seq.v

```
    `timescale 1ns / 1ps

module mul_seq(
    output reg[255:0] out,
    output ready,
    input [255:0] multiplicand,
    input [255:0] multiplier,
    input start,
    input clock);

   parameter P = 256'hFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFEFFFFC2F;

   reg [9:0] bit;
   assign ready = !bit;

   reg [255:0] multiplier_copy;
   reg [511:0] multiplicand_copy;

   reg [511:0] product;
```

```
   initial bit = 0;

   always @(posedge clock) begin
     if(ready && start) begin
         bit <= 256;
         product <= 0;
         multiplicand_copy <= {256'd0, multiplicand};
         multiplier_copy <= multiplier;
     end else if(bit) begin
         if(multiplier_copy[0] == 1'b1) begin
            product <= product + multiplicand_copy;
         end

         multiplier_copy <= multiplier_copy >> 1;
         multiplicand_copy <= multiplicand_copy << 1;
         bit <= bit - 1;

         if(bit - 1 == 0) begin
            out <= product % P;
         end
      end
   end

endmodule
```

B.5 test_mul_seq.v

```
`timescale 1ns / 1ps

module test_mul_booth;
    reg clock, start;

    wire ready;

    reg [255:0] a_input = 256'h9cc57f2ca39c2d81aed7e3d82af0b5711863bd3403bb8f024c4c3b4ecf9652a;
    reg [255:0] b_input = 256'hfd04ed02aef57789f1312d6817b6e9e214fade46622a760e692363e1843b3c2;

    wire [255:0] product;

    always #5 clock = !clock;

    mul_seq multiplier(.clock(clock), .multiplicand(a_input), .multiplier(b_input),
```

```
.out(product), .ready(ready), .start(start));

parameter OUTPUT = 256'h98C40E99542A11A9EBB084E21DA7CB9397443C6A026097A892294AB8352668A1;
parameter OUTPUT2 = 256'h16E6559E8DEC319CAEFF16BD2FD4854FC5B51D92BD1FA9BE933ED17223728F4E;
parameter OUTPUT3 = 256'hDE1F385C39CE51CC35A84A986232F01935A81818CFB21E8B79861D6396CD55D4;

initial begin
    clock = 0;

    #10;

    start = 1;

    #10;

    start = 0;

    #2570;

    if (product != OUTPUT) begin
        $display("mod_mul failed: product = %x, expected %x", product, OUTPUT);
        $stop;
    end

    a_input = 256'hD32E1674426BB9251DF6E79F80D4518A2EBA853F9E0009656F2A2A56964903E4;
    start = 1;
    #10;

    start = 0;

    #2570;

    if (product != OUTPUT2) begin
        $display("mod_mul failed: product = %x, expected %x", product, OUTPUT2);
        $stop;
    end

    a_input = 256'hFFFFFFFFFFFFFFFFFFFFFFaFFFFFFFFFFFFFFFFF0FFFFFFFFFFFFFFFFFFFFFFFF;
    b_input = 256'hFF04ed02aef57789f1312d6817b6e9e214fade46622a760e692363e1843b3c2;
    start = 1;

    #10;

    start = 0;
```

```
        #2570;

        if (product != OUTPUT3) begin
            $display("mod_mul failed: product = %x, expected %x", product, OUTPUT3);
            $stop;
        end

        $display("mod_mul passed");
    end

endmodule
```

B.6 test_axi_uart_manager.v

```
‘timescale 1ns / 1ps

module test_axi_uart_manager;

    reg clk, uart_tx_in, reset;

    wire uart_rx_out;

    parameter INDATA = 10’b0111001101;

    reg [9:0] cur_data;

    always #33.333 clk <= !clk;

    initial begin
        clk = 0;
        reset = 0;
        uart_tx_in = 1;
        cur_data = INDATA;

        #200;

        reset = 1;

        #1000;

        #104166;

        uart_tx_in = cur_data[9];
```

```
    #104166;
    uart_tx_in = cur_data[8];
    #104166;
    uart_tx_in = cur_data[7];
    #104166;
    uart_tx_in = cur_data[6];
    #104166;
    uart_tx_in = cur_data[5];
    #104166;
    uart_tx_in = cur_data[4];
    #104166;
    uart_tx_in = cur_data[3];
    #104166;
    uart_tx_in = cur_data[2];
    #104166;
    uart_tx_in = cur_data[1];
    #104166;
    uart_tx_in = cur_data[0];
    #104166;
    uart_tx_in = 1;
end
axi_uart_manager uart(.clock(clk), .uart_in(uart_tx_in), .uart_out(uart_rx_out), .reset(reset
```

```
endmodule
```

B.7 Nexsys-Video-Master.xdc

```
    ## Clock Signal - Vivado appends all parameters it edits so this was moved
set_property -dict {PACKAGE_PIN R4 IOSTANDARD LVCMOS33} [get_ports SYSCLK]

## LEDs
set_property -dict {PACKAGE_PIN T14 IOSTANDARD LVCMOS25} [get_ports {LED[0]}]
set_property -dict {PACKAGE_PIN T15 IOSTANDARD LVCMOS25} [get_ports {LED[1]}]
set_property -dict {PACKAGE_PIN T16 IOSTANDARD LVCMOS25} [get_ports {LED[2]}]
set_property -dict {PACKAGE_PIN U16 IOSTANDARD LVCMOS25} [get_ports {LED[3]}]
set_property -dict {PACKAGE_PIN V15 IOSTANDARD LVCMOS25} [get_ports {LED[4]}]
set_property -dict {PACKAGE_PIN W16 IOSTANDARD LVCMOS25} [get_ports {LED[5]}]
set_property -dict {PACKAGE_PIN W15 IOSTANDARD LVCMOS25} [get_ports {LED[6]}]
set_property -dict {PACKAGE_PIN Y13 IOSTANDARD LVCMOS25} [get_ports {LED[7]}]

## Switches
set_property -dict {PACKAGE_PIN E22 IOSTANDARD LVCMOS12} [get_ports {SW[0]}]
set_property -dict {PACKAGE_PIN F21 IOSTANDARD LVCMOS12} [get_ports {SW[1]}]
set_property -dict {PACKAGE_PIN G21 IOSTANDARD LVCMOS12} [get_ports {SW[2]}]
set_property -dict {PACKAGE_PIN G22 IOSTANDARD LVCMOS12} [get_ports {SW[3]}]
set_property -dict {PACKAGE_PIN H17 IOSTANDARD LVCMOS12} [get_ports {SW[4]}]
set_property -dict {PACKAGE_PIN J16 IOSTANDARD LVCMOS12} [get_ports {SW[5]}]
set_property -dict {PACKAGE_PIN K13 IOSTANDARD LVCMOS12} [get_ports {SW[6]}]
set_property -dict {PACKAGE_PIN M17 IOSTANDARD LVCMOS12} [get_ports {SW[7]}]

## UART
set_property -dict { PACKAGE_PIN AA19  IOSTANDARD LVCMOS33 } [get_ports { uart_rx_out }]; #IO_L15
set_property -dict { PACKAGE_PIN V18   IOSTANDARD LVCMOS33 } [get_ports { uart_tx_in }]; #IO_L14P

## Configuration options, can be used for all designs
set_property CONFIG_VOLTAGE 3.3 [current_design]
set_property CFGBVS VCCO [current_design]

## Clock period & duty cycle + LED output delays
create_clock -period 66.666 -name sys_clk_pin -waveform {0.000 33.333} [get_ports SYSCLK]
set_output_delay -clock [get_clocks sys_clk_pin] -max -add_delay 10.000 [get_ports {LED[*]}]
set_output_delay -clock [get_clocks sys_clk_pin] -min -add_delay 30.000 [get_ports {LED[*]}]

set_output_delay -clock [get_clocks sys_clk_pin] -max 2 [get_ports uart_rx_out]
set_output_delay -clock [get_clocks sys_clk_pin] -min 1 [get_ports uart_rx_out]
set_input_delay -clock [get_clocks sys_clk_pin] -max 2 [get_ports uart_tx_in]
set_input_delay -clock [get_clocks sys_clk_pin] -min 1 [get_ports uart_tx_in]
```

B.8 debounce.v

```
module debounce (input reset, clock, noisy, output reg clean);

    reg [19:0] count;
    reg new;

    always @(posedge clock) begin
        if (reset) begin new <= noisy; clean <= noisy; count <= 0; end
        else if (noisy != new) begin new <= noisy; count <= 0; end
        else if (count == 1000000) clean <= new;
        else count <= count+1;
    end

endmodule
```

B.9 div.v

```
‘timescale 1ns / 1ps

module div(
    input clock,
    input [255:0] a,
    input [255:0] b,
    input start,
    output [255:0] q,
    output [255:0] r,
    output ready
    );

    reg [511:0] qr;

    assign r = qr[511:256];
    assign q = qr[255:0];

    reg [8:0] bit;
    assign ready = !bit;

    initial bit = 0;

    wire [256:0] diffstep;

    assign diffstep = qr[511:255] - {1’b0, b};
```

```
    always @(posedge clock) begin
        if (ready && start) begin
            bit <= 256;
            qr <= {256'd0, a};
        end else begin
            if (diffstep[256]) begin
                qr <= {qr[510:0], 1'd0};
            end else begin
                qr <= {diffstep[255:0], qr[254:0], 1'd1};
            end

            bit <= bit - 1;
        end
    end

endmodule
```

B.10 mod_add.v

```
`timescale 1ns / 1ps

module mod_add(
    input clock,
    input [255:0] a,
    input [255:0] b,
    output reg [255:0] out
    );

    parameter P = 257'h0FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFEFFFFC2F;

    always @(posedge clock) begin
        if ({1'b0, a} + {1'b0, b} >= P) begin
            out <= (a + b) - P;
        end else begin
            out <= a + b;
        end
    end

endmodule
```

B.11 mod_sub.v

```
    `timescale 1ns / 1ps

module mod_sub(
    input clock,
    input [255:0] a,
    input [255:0] b,
    output reg [255:0] out
    );

    parameter P = 256'hFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFEFFFFC2F;

    always @(posedge clock) begin
        if (b > a) begin
            out <= P - (b - a);
        end else begin
            out <= a - b;
        end
    end

endmodule
```

B.12 test_bnvm.v

```
`timescale 1ns / 1ps

module test_bnvm;

    parameter MEM_BUS = 63;
    parameter MEM_CYCLES = 256/MEM_BUS;
    parameter MEM_DELAY = 40*MEM_CYCLES;

    reg working;

    reg clock, we, reset, execute;

    wire ena, stack_ena, stack_we, halt;

    wire [13:0] instr_ptr, stack_ptr;
    wire [MEM_BUS:0] din, dout, stack_din, stack_dout;

    always #20 clock = !clock;
```

```
instr_mem instrs(.clka(clock), .ena(ena), .wea(we), .dina(din), .douta(dout),
.addra(instr_ptr));
stack_mem sm(.clka(SYSCLK), .addra(stack_ptr), .ena(stack_ena), .wea(stack_we),
.dina(stack_din), .douta(stack_dout));

bnvm vm(.clock(clock),
        .reset(reset),
        .instr_ptr(instr_ptr),
        .instr_ena(ena),
        .instr_data(dout),
        .stack_data(stack_dout),
        .wstack_data(stack_din),
        .stack_we(stack_we),
        .stack_ena(stack_ena),
        .stack_ptr_out(stack_ptr),
        .halt(halt),
        .execute(execute));

initial begin
    clock = 0;
    we = 0;
    reset = 0;
    working = 0;

    #100;
    reset = 1;

    #40;
    reset = 0;
    execute = 1;

    // 2-cycle delay before we initialize load_counter

    #80;
    working = ~working;

    // populating stack

    #MEM_DELAY;
    working = ~working;

    // 2 cycles to switch to stack popping + 2 to start reading, pop delayed by 1 cycle

    #160;
    working = ~working;
```

```
// popping stack into A

#MEM_DELAY;
working = ~working;

// same delays & situation as when popping A

#160;
working = ~working;

// now popping B
// though pop of A did not clear the stack..? Seems like we'd overflow our stack

#MEM_DELAY;
working = ~working;

// 2 cycles before we load push_tmp

#80;
working = ~working;

// A pushed, begin switching states for pushing D

#MEM_DELAY;
working = ~working;

// same delay

#80;
working = ~working;

#MEM_DELAY;
working = ~working;

// 3 cycles needed to halt

#120;
working = ~working;

#40;
working = ~working;

#80;
```

```
        $display("bnvm passed");
    end

endmodule
```

B.13 test_div.v

```
    `timescale 1ns / 1ps

module test_div;

    reg clock, start;

    reg [255:0] a_input = 256'h9cc57f2ca39c2d81aed7e3d82af0b5711863bd3403bb8f024c4c3b4ecf9652a;
    reg [255:0] b_input = 256'hfd04ed02aef57789f1312d6817b6e9e214fade46622a760e692363e1843b3c2;

    wire [255:0] q;
    wire [255:0] r;

    wire ready;

    always #5 clock = !clock;

    div divider(.clock(clock), .start(start), .ready(ready), .a(a_input),
    .b(b_input), .q(q), .r(r));

    parameter OUTPUTQ = 256'h0;
    parameter OUTPUTR = 256'h09CC57F2CA39C2D81AED7E3D82AF0B5711863BD3403BB8F024C4C3B4ECF9652A;

    parameter OUTPUT2Q = 256'h0D;
    parameter OUTPUT2R = 256'h059A15E21444480509FEF2BAED8FB3827DAEB0A66E3D8979B9BD692F5AD8E30A;

    parameter OUTPUT3Q = 256'h10;
    parameter OUTPUT3R = 256'hFB12FD510A88760ECECD97E849161DEB0520C99DD589F196DC9C1E7BC4C3DF;

    initial begin
        clock = 0;
        start = 1;

        #10;

        start = 0;

        #2560;
```

```
        if (q != OUTPUTQ || r != OUTPUTR || !ready) begin
            $display("div failed: q = %x, r = %x, ready = %b, expected %x %x", q, r,
            ready, OUTPUTQ, OUTPUTR);
            $stop;
        end

        a_input = 256'hD32E1674426BB9251DF6E79F80D4518A2EBA853F9E0009656F2A2A56964903E4;

        start = 1;

        #10;

        start = 0;

        #2560;

        if (q != OUTPUT2Q || r != OUTPUT2R || !ready) begin
            $display("div failed: q = %x, r = %x, expected %x %x", q, r, OUTPUT2Q, OUTPUT2R);
            $stop;
        end

        a_input = 256'hFFFFFFFFFFFFFFFFFFFFFaFFFFFFFFFFFFFFFFF0FFFFFFFFFFFFFFFFFFFFFFFF;
        b_input = 256'hFF04ed02aef57789f1312d6817b6e9e214fade46622a760e692363e1843b3c2;

        start = 1;

        #10;

        start = 0;

        #2560;

        if (q != OUTPUT3Q || r != OUTPUT3R || !ready) begin
            $display("div failed: q = %x, r = %x, expected %x %x", q, r, OUTPUT3Q, OUTPUT3R);
            $stop;
        end

        $display("div passed");
    end

endmodule
```

B.14 test_mod_add.v

```
    `timescale 1ns / 1ps

module test_mod_add;

    reg clock;

    reg [255:0] a_input = 256'h9cc57f2ca39c2d81aed7e3d82af0b5711863bd3403bb8f024c4c3b4ecf9652a;
    reg [255:0] b_input = 256'hfd04ed02aef57789f1312d6817b6e9e214fade46622a760e692363e1843b3c2;

    wire [255:0] sum;

    always #20 clock = !clock;

    mod_add adder(.clock(clock), .a(a_input), .b(b_input), .out(sum));

    parameter OUTPUT = 256'h199ca6c2f5291a50ba009114042a79f532d5e9b7a65e60510b56f9f3053d18ec;
    parameter OUTPUT2 = 256'hE2FE65446D5B109DBD09FA76024FC028500A33240422B0C655BC6094AE8CB7A6;
    parameter OUTPUT3 = 256'h0FF04ED02AEF57789F130DD6817B6E9E214FACF46622A760E692363F1843B792;

    initial begin
        clock = 0;

        #40;

        if (sum != OUTPUT) begin
            $display("mod_add failed: sum = %x, expected %x", sum, OUTPUT);
            $stop;
        end

        a_input = 256'hD32E1674426BB9251DF6E79F80D4518A2EBA853F9E0009656F2A2A56964903E4;

        #40;

        if (sum != OUTPUT2) begin
            $display("mod_add failed: sum = %x, expected %x", sum, OUTPUT2);
            $stop;
        end

        a_input = 256'hFFFFFFFFFFFFFFFFFFFFFaFFFFFFFFFFFFFFFFF0FFFFFFFFFFFFFFFFFFFFFFFF;
        b_input = 256'hFF04ed02aef57789f1312d6817b6e9e214fade46622a760e692363e1843b3c2;

        #40;
```

```
        if (sum != OUTPUT3) begin
            $display("mod_add failed: sum = %x, expected %x", sum, OUTPUT3);
            $stop;
        end

        $display("mod_add passed");
    end

endmodule
```

B.15 test_mod_mul.v

```
    `timescale 1ns / 1ps

module test_mod_mul;

    reg clock;

    reg [255:0] a_input = 256'h9cc57f2ca39c2d81aed7e3d82af0b5711863bd3403bb8f024c4c3b4ecf9652a;
    reg [255:0] b_input = 256'hfd04ed02aef57789f1312d6817b6e9e214fade46622a760e692363e1843b3c2;

    wire [255:0] product;

    always #5 clock = !clock;

    mod_mul multiplier(.clock(clock), .a(a_input), .b(b_input), .out(product));

    parameter OUTPUT = 256'h98C40E99542A11A9EBB084E21DA7CB9397443C6A026097A892294AB8352668A1;
    parameter OUTPUT2 = 256'h16E6559E8DEC319CAEFF16BD2FD4854FC5B51D92BD1FA9BE933ED17223728F4E;
    parameter OUTPUT3 = 256'hDE1F385C39CE51CC35A84A986232F01935A81818CFB21E8B79861D6396CD55D4;

    initial begin
        clock = 0;

        #10;

        if (product != OUTPUT) begin
            $display("mod_mul failed: product = %x, expected %x", product, OUTPUT);
            $stop;
        end

        a_input = 256'hD32E1674426BB9251DF6E79F80D4518A2EBA853F9E0009656F2A2A56964903E4;

        #10;
```

```
        if (product != OUTPUT2) begin
            $display("mod_mul failed: product = %x, expected %x", product, OUTPUT2);
            $stop;
        end

        a_input = 256'hFFFFFFFFFFFFFFFFFFFFFaFFFFFFFFFFFFFFFFF0FFFFFFFFFFFFFFFFFFFFFFFF;
        b_input = 256'hFF04ed02aef57789f1312d6817b6e9e214fade46622a760e692363e1843b3c2;

        #10;

        if (product != OUTPUT3) begin
            $display("mod_mul failed: product = %x, expected %x", product, OUTPUT3);
            $stop;
        end

        $display("mod_mul passed");
    end

endmodule
```

B.16 test_mod_sub.v

```
    `timescale 1ns / 1ps

module test_mod_sub;

    reg clock;

    reg [255:0] a_input = 256'h9cc57f2ca39c2d81aed7e3d82af0b5711863bd3403bb8f024c4c3b4ecf9652a;
    reg [255:0] b_input = 256'hfd04ed02aef57789f1312d6817b6e9e214fade46622a760e692363e1843b3c2;

    wire [255:0] sum;

    always #5 clock = !clock;

    mod_sub subber(.clock(clock), .a(a_input), .b(b_input), .out(sum));

    parameter OUTPUT = 256'hF9FC09229F4A6B5F7BDA6B6701339CB8F0368DEEDA19118F3E328D75D4B5AD97;
    parameter OUTPUT2 = 256'hC35DC7A4177C61AC7EE3D4C8FF58E2EC0D6AD75B37DD62048897F4187E055022;
    parameter OUTPUT3 = 256'hF00FB12FD510A88760ECE8297E849161DEB0512B99DD589F196DC9C1E7BC4C3D;

    initial begin
        clock = 0;
```

```
        #10;

        if (sum != OUTPUT) begin
            $display("mod_sub failed: sum = %x, expected %x", sum, OUTPUT);
            $stop;
        end

        a_input = 256'hD32E1674426BB9251DF6E79F80D4518A2EBA853F9E0009656F2A2A56964903E4;

        #10;

        if (sum != OUTPUT2) begin
            $display("mod_sub failed: sum = %x, expected %x", sum, OUTPUT2);
            $stop;
        end

        a_input = 256'hFFFFFFFFFFFFFFFFFFFFFFaFFFFFFFFFFFFFFFFF0FFFFFFFFFFFFFFFFFFFFFFFF;
        b_input = 256'hFF04ed02aef57789f1312d6817b6e9e214fade46622a760e692363e1843b3c2;

        #10;

        if (sum != OUTPUT3) begin
            $display("mod_sub failed: sum = %x, expected %x", sum, OUTPUT3);
            $stop;
        end

        $display("mod_sub passed");
    end

endmodule
```

C C library

C.1 ec.c

```
#include "ec.h"

#include "bn.h"

void my_ec_point_free(struct my_ec_point* p) {
    if(p != NULL) {
        BN_free(p->X);
```

```
        BN_free(p->Y);
        free(p);
    }
}

struct my_ec_ctx* my_ec_ctx_new() {
    struct my_ec_ctx* returning = malloc(sizeof(struct my_ec_ctx));

    returning->bn_ctx = BN_CTX_new();
    if(returning->bn_ctx == NULL) {
        return NULL;
    }

    const char* p_hex
    = "FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFEFFFFFC2F";
    int ret = BN_hex2bn(&returning->p, p_hex);
    if(ret == 0) {
        return NULL;
    }

    returning->G = malloc(sizeof(struct my_ec_point));
    const char* g_x_hex
    = "79BE667EF9DCBBAC55A06295CE870B07029BFCDB2DCE28D959F2815B16F81798";
    const char* g_y_hex
    = "483ADA7726A3C4655DA4FBFC0E1108A8FD17B448A68554199C47D08FFB10D4B8";

    ret = BN_hex2bn(&returning->G->X, g_x_hex);
    if(ret == 0) {
        return NULL;
    }

    ret = BN_hex2bn(&returning->G->X, g_y_hex);
    if(ret == 1) {
        return NULL;
    }

    return returning;
}

void my_ec_ctx_free(struct my_ec_ctx* ctx) {
    if(ctx != NULL) {
        BN_CTX_free(ctx->bn_ctx);
        BN_free(ctx->p);
        my_ec_point_free(ctx->G);
        free(ctx);
```

```
    }
}

void my_ec_point_print(struct my_ec_point* a) {
    char* x = BN_bn2hex(a->X);
    char* y = BN_bn2hex(a->Y);

    printf("(%s, %s)\n", x, y);

    free(x);
    free(y);
}

int my_ec_point_eq(struct my_ec_point* p, struct my_ec_point* q) {
    const int x_cmp = BN_cmp(p->X, q->X);
    const int y_cmp = BN_cmp(p->Y, q->Y);

    return y_cmp == 0 && x_cmp == 0;
}

BIGNUM* my_ec_point_get_lambda(struct my_ec_ctx* ctx,
                    struct my_ec_point* p, struct my_ec_point* q) {
    BIGNUM* lambda = BN_new();

    int res;

    if(my_ec_point_eq(p, q)) {
        BIGNUM* xp_sq = BN_new();

        res = BN_mod_sqr(xp_sq, p->X, ctx->p, ctx->bn_ctx);
        if(res != 1) {
            printf("my_ec_point_get_lambda");
            exit(-1);
        }

        res = BN_mul_word(xp_sq, 3);
        if(res != 1) {
            printf("my_ec_point_get_lambda");
            exit(-1);
        }

        BIGNUM* xp_sq_mod = BN_new();

        res = BN_nnmod(xp_sq_mod, xp_sq, ctx->p, ctx->bn_ctx);
        if(res != 1) {
```

```
        printf("my_ec_point_get_lambda");
        exit(-1);
    }

    BIGNUM* yp = BN_dup(p->Y);

    res = BN_mul_word(yp, 2);
    if(res != 1) {
        printf("my_ec_point_get_lambda");
        exit(-1);
    }

    BIGNUM* yp_mod = BN_new();

    res = BN_nnmod(yp_mod, yp, ctx->p, ctx->bn_ctx);
    if(res != 1) {
        printf("my_ec_point_get_lambda");
        exit(-1);
    }

    BN_mod_inv_mul(ctx->bn_ctx, lambda, xp_sq_mod, yp_mod, ctx->p);

    BN_free(xp_sq);
    BN_free(yp);
    BN_free(xp_sq_mod);
    BN_free(yp_mod);
} else {
    BIGNUM* y_sub = BN_new();
    BIGNUM* x_sub = BN_new();

    res = BN_mod_sub(y_sub, q->Y, p->Y, ctx->p, ctx->bn_ctx);
    if(res != 1) {
        printf("my_ec_point_get_lambda");
        exit(-1);
    }

    res = BN_mod_sub(x_sub, q->X, p->X, ctx->p, ctx->bn_ctx);
    if(res != 1) {
        printf("my_ec_point_get_lambda");
        exit(-1);
    }

    res = BN_mod_inv_mul(ctx->bn_ctx, lambda, y_sub, x_sub, ctx->p);
    if(res != 1) {
        printf("my_ec_point_get_lambda");
```

```
            exit(-1);
        }

        BN_free(y_sub);
        BN_free(x_sub);
    }

    return lambda;
}

struct my_ec_point* my_ec_point_add(struct my_ec_ctx* ctx,
           struct my_ec_point* p, struct my_ec_point* q) {
    BIGNUM* lambda = my_ec_point_get_lambda(ctx, p, q);

    BIGNUM* lambda_squared = BN_new();

    int res = BN_mod_sqr(lambda_squared, lambda, ctx->p, ctx->bn_ctx);
    if(res != 1) {
        printf("BN_mod_sqr");
        exit(-1);
    }

    struct my_ec_point* ret = malloc(sizeof(struct my_ec_point));
    ret->X = BN_new();
    ret->Y = BN_new();

    BIGNUM* lambda_squared_minus_xp = BN_new();

    /* lambda^2 - x_p */
    res = BN_mod_sub(lambda_squared_minus_xp, lambda_squared,
                     p->X, ctx->p, ctx->bn_ctx);
    if(res != 1) {
        printf("BN_mod_sub");
        exit(-1);
    }

    /* (lambda^2 - x_p) - x_q */
    res = BN_mod_sub(ret->X, lambda_squared_minus_xp,
                     q->X, ctx->p, ctx->bn_ctx);
    if(res != 1) {
        printf("BN_mod_sub");
        exit(-1);
    }

    /* x_p - x_r */
```

```
    BN_mod_sub(ret->Y, p->X, ret->X, ctx->p, ctx->bn_ctx);

    /* lambda(x_p - x_r) */
    BN_mod_mul(ret->Y, ret->Y, lambda, ctx->p, ctx->bn_ctx);

    /* lambda(x_p - x_r) - y_p */
    BN_mod_sub(ret->Y, ret->Y, p->Y, ctx->p, ctx->bn_ctx);

    BN_free(lambda);
    BN_free(lambda_squared);
    BN_free(lambda_squared_minus_xp);

    return ret;
}

struct my_ec_point* my_ec_point_mul(struct my_ec_ctx* ctx,
                        BIGNUM* d, struct my_ec_point* P) {
    struct my_ec_point* N = malloc(sizeof(struct my_ec_point));
    struct my_ec_point* Q = malloc(sizeof(struct my_ec_point));

    N->X = BN_dup(P->X);
    N->Y = BN_dup(P->Y);

    Q->X = NULL;
    Q->Y = NULL;

    unsigned char d_bytes[32];
    BN_bn2binpad(d, &d_bytes[0], 32);

    int i;
    for(i = 0; i < 256; i++) {
        if((d_bytes[31 - (i / 8)] & (0x01 << (i % 8))) != 0) {
            if(Q->X == NULL) {
                Q->X = BN_dup(N->X);
                Q->Y = BN_dup(N->Y);
            } else {
                struct my_ec_point* old_Q = Q;
                Q = my_ec_point_add(ctx, Q, N);
                my_ec_point_free(old_Q);
            }
        }
        struct my_ec_point* old_N = N;
        N = my_ec_point_add(ctx, N, N);
        my_ec_point_free(old_N);
    }
```

```
    my_ec_point_free(N);

    return Q;
}
```

C.2 ec.h

```
    #include <openssl/bn.h>

struct my_ec_point {
    BIGNUM* X;
    BIGNUM* Y;
};

struct my_ec_ctx {
    BN_CTX* bn_ctx;
    BIGNUM* p;
    struct my_ec_point* G;
};

void my_ec_point_free(struct my_ec_point* p);
struct my_ec_ctx* my_ec_ctx_new();
void my_ec_ctx_free(struct my_ec_ctx* ctx);
void my_ec_point_print(struct my_ec_point* a);
int my_ec_point_eq(struct my_ec_point* p, struct my_ec_point* q);
struct my_ec_point* my_ec_point_add(struct my_ec_ctx* ctx,
                    struct my_ec_point* p, struct my_ec_point* q);
struct my_ec_point* my_ec_point_mul(struct my_ec_ctx* ctx,
                    BIGNUM* d, struct my_ec_point* P);
```

C.3 bn.c

```
#include <openssl/bn.h>

/*
 * Calculates (a/b) % order
 *
 * This is *not* divide. It's a modular inverse followed by a
 * modular multiply. i.e. (b^(-1) * a) % order
 */
int BN_mod_inv_mul(BN_CTX* ctx, BIGNUM* ret, BIGNUM* a, BIGNUM* b, BIGNUM* order) {
    BIGNUM* binv = BN_new();
```

```
    BN_mod_inverse(binv, b, order, ctx);

    BN_mod_mul(ret, a, binv, order, ctx);

    BN_free(binv);

    return 1;
}
```

C.4 bn.h

```
#include <openssl/bn.h>

int BN_mod_inv_mul(BN_CTX* ctx, BIGNUM* ret, BIGNUM* a, BIGNUM* b, BIGNUM* order);
```

C.5 premake5.lua

```
workspace "cpu_validation"
configurations {"Debug", "Release"}
platforms {"Static", "Shared"}
language "C"
cdialect "C89"
symbols "On"
warnings "Extra"

filter {"configurations:Debug"}
    optimize "Off"

filter {"configurations:Release"}
    optimize "Full"

project "ecc_ops"

kind "ConsoleApp"
files {"*.c", "*.h"}
links {"crypto"}
```