MOS6502 Final Report

1st Kelly Xu
Department of EECS
Massachusetts Institute of Technology
Cambridge, MA, USA
krxu@mit.edu

2nd Jan Strzeszynski
Department of EECS
Massachusetts Institute of Technology
Cambridge, MA, USA
jstrz@mit.edu

***Abstract*—We aim to implement the MOS6502 Microprocessor using Field Programmable Gate Arrays. We implement the control logic with a complex FSM with many stages handling the large instruction set and supporting all addressing modes. We use two 8 bit addressing registers and implement memory using dual port xilinx block RAMs to allow for reads and writes. Processor functionality is extended with a custom Secure Hash Algorithm 2 (SHA-2) module. We interface with the processor through a specialized manta module. Finally, we evaluate our current system using modularized testbenches.**

***Index Terms*—Digital Systems, Field Programmable Gate Arrays, MOS6502, Complex Instruction Set Processor, SHA-2, SHA-256, Manta**

I. Introduction

We aim to implement the MOS6502 processor, a complex instruction set (CISC) processor on FPGA. Implementing a processor on FPGAs allows for extending functionality through customizable modules. The final design is a functional MOS6502 processor with an additional security module that allows users to be authenticated prior to running software.

The challenges of implementing this processor on FPGA are the complex instruction set needed and additional connections needed for testing. Adding an additional security module requires determining timing constraints due to operations on large inputs, and implementing separate states for data accesses and calculations.

The design is a functional MOS6502 processor that follows the specifications of the original with added security in the form of a specialized security module. Additionally, a specialized module allows for easy interfacing with the processor.

II. System Overview

The goal of a processor is to execute instructions. The processor must be able to parse an instruction set, execute those instructions, and modify memory accordingly. We introduce security into the MOS 6502 by adding a customized authentication module requiring a correct password before a process can be run. We also allow for easier interfacing with the processor with the addition of a manta module [6]. The manta module allows for straightforward interfacing with the processor through immediate interaction. With the manta module, users are able to load programs to memory and the processor is able to control when it's accepting new inputs.

A. Overview Diagram

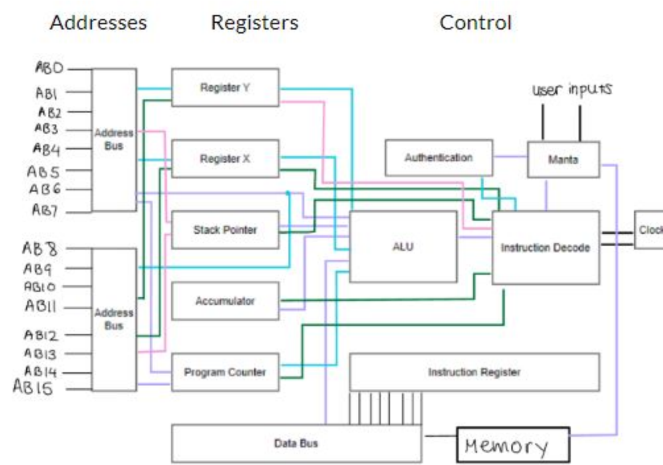

Fig. 1. High level overview of key components

The processor is divided into three main components: address, registers, and control logic. The addresses of the processor are used to determine the location in memory that the processor should use. Addresses are 16 bits, divided into two buses - address bus low and address bus high.

The registers are used for fast access to values needed for operations. This includes registers such as the program counter that stores the memory address of the next instruction to be executed. Finally, the control logic determines the action that the processor should take through decoding instructions, making calculations, etc.

B. Processor FSM

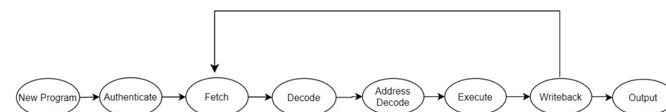

Fig. 2. Simplified overall FSM

The processor is able to take in a new program and authenticate the user loading the program. If the user is authenticated, the processor fetches the instructions from memory, decodes

the instruction, executes the instruction, and writes the result to memory. Fetching, decoding, execution, and writeback are all part of the processors control logic.

III. Registers

The processor works with the following set of external (i.e. available to the user) registers:

- ACC - accumulator, used for performing ALU operations
- X, Y - index registers, used for indexing in some of the addressing modes
- SP - stack pointer, used by instructions accessing the stack (e.g. JSR, RTS)
- PC - program counter
- FLAGS - processor status flags, the one byte register is interpreted as NV1BDIZC
  - NFLAG - indicates whether the result of last operation was negative (i.e. whether the most significant bit was 1)
  - VFLAG - indicates whether the last operation resulted in an overflow (decided by ALU)
  - bit 5 of the FLAGS register is always 1
  - BFLAG - set on interrupts generated by the BRK instruction
  - DFLAG - indicates whether the processor is currently operating in decimal mode
  - IFLAG - when set, the processor will ignore any external (i.e. from devices) interrupts
  - ZFLAG - indicates whether the result of last operation was zero
  - CFLAG - carry produced by the last operation (decided by ALU)

IV. ALU

The ALU is the part of the processor that performs all the logical and mathematical operations necessary for the processor to function. The ALU is connected to the pointers, registers, accumulators, data bus, and address bus as these all require ALU operations.

A. Inputs and Outputs

Inputs

- Operand 1: An 8 bit operand to the ALU.
- Operand 2: The second 8 bit operand to the ALU.
- Opcode: An enum determining which operation should be taken.
- Carry in: One bit for indicating if there's a carry from a previous operation that should be accounted for. Only applicable to the ADD and SUB operations.
- Decimal: One bit for indicating if the ALU should operate in decimal mode or normal binary mode. Only applicable to the ADD and SUB operations.

Outputs

- Result: 8 bit result of the operation.
- Carry out: One bit indicating whether there was a carry from the result of the operation.
- Zero: One bit indicating if the result is equal to 0.
- Overflow: One bit indicating if the result of the operation is too large to be accurately represented. Applicable to ADD and SUB operations on signed numbers.

B. Operations

The MOS6502 ALU has six main operations: ADD, SUB, AND, OR, XOR, and Shift Right.

- ADD: Operates in decimal mode or in binary mode. Adds the two operands and the carry input in either two's complement or binary coded decimal.
- SUB: Operates in decimal mode or in binary mode. Subtracts operand 2 from operand 1 and subtracts the inverse of the carry in bit to allow for chained operations.
- AND: Bitwise AND function
- OR: Bitwise OR function
- XOR: Bitwise XOR function
- Shift Right: Shifts the first operand and shifts in the second operand. Sets the shifted-out bit as the carry out bit.

C. Decimal Mode

Decimal mode is implemented using binary coded decimal (BCD), which uses binary 0000-1001 to represent digits 0-9. The basic module to implement decimal mode summation is a four bit BCD adder that operates on the logic that if the immediate output of the summation of the two operands and the carry is greater than 9, then add 6 to make the BCD representation equal to the binary representation. If the immediate output is less than 9, than the BCD representation is already equal to the binary representation. For subtraction, the second operand input to the BCD adder is inverted and one is added to the inverted input to allow for re-use of the adder.

D. ALU Logic

We chose to implement the ALU logic combinationally as a series of case statements based on the opcode. While the rest of the processor will be clock driven, we decided it is more optimal for the ALU outputs to be available as soon as possible instead of requiring one extra clock cycle. Given the original MOS6502 had a relatively slow clock compared to what is typically used today, we did not want to add any intentional delay to the ALU.

The input opcode is an enum of the six operations to indicate which operation the ALU should perform. This allows for simplicity and ease of adding operations if necessary. It also allows for easy readability and use of the ALU. Additionally, the default is that every output is set to 0.

E. Testing

We tested our ALU by creating a thorough testbench that runs through each possible valid pair of input operands -128 to 127. The testbench also covered cases when certain inputs would have carry in set to one or decimal mode set to one. Additionally, the testbench covered cases of overflow. The testbench did not cover cases of illegal inputs such as an input

of -129 because the system has unspecified behavior at that point.

V. Memory

Every module that uses memory was implemented with BRAMs. The amount of memory needed was implementable with BRAMs. Additionally, BRAM memory allows for fast access compared to other types of memory, thus adhering to our goal of an efficient processor.

The overall processor uses a dual port BRAM with read and write access. Dual port allows for easier interfacing between the manta module and with the processor. The processor memory has a RAM width of 8 bits for an 8 bit word size and a RAM depth of 2^{16} bits to represent all possible addresses given by addressing with two 8-bit registers. This uses approximately 0.52 Mbits.

For the processor BRAM, the xilinx dual port ram was used for the memory module with default high performance to allow for an output register. The memory module has a two clock cycle latency. If write enabled, the memory module writes the input data to the address specified. The memory is first initialized to all zeroes prior to any writes. We use the clogb2 function to calculate the address width based on input depth. As we specified the default to 2^{16}, as long as the default is not overridden, the address width should be 16.

The authentication module also uses memory in the form of BRAMs. One read-only single port BRAM is used to store 64 32-bit constants and is initialized with the constants. One read-and-write single-port BRAM is used to store the 64 32-bit results of the calculations during authentication. Both BRAMs have a RAM width of 32 bits and a RAM depth of 64 bits.

VI. Control logic

This section introduces the instruction set of the processor (Appendix A) along with the supported addressing modes. Additionally, the way each instruction is broken down to over multiple clock cycles is described, which provides a general idea how the FSM of the control logic looks.

A. Addressing modes

- **Implicit:**
  Used for instructions which do not need an argument (e.g. CLC, PHA).
- **Accumulator:**
  Use the accumulator instead of a value from memory (e.g. ASL, ROR).
- **Immediate:**
  Use the value directly following the instruction - the argument (e.g. ADC, LDA).
- **Zero Page:**
  Absolute addressing to an address in the range 0x0000-0x00FF, requires only one byte of address.
- **Zero Page X / Zero Page Y:**
  The instruction provides one byte of address. Register X or Y is added to this value (ignoring carry) to get the actual address.
- **Relative:**
  Used by branch instructions. The argument is a one byte of signed offset, which is added to PC if the branch is taken.
- **Absolute:**
  Takes two bytes of the address as an argument.
- **Absolute X / Absolute Y:**
  Takes two bytes of the address as an argument. Register X or Y is added to this 16 bit value to get the actual address.
- **Indirect:**
  Used only by JMP. Takes two bytes of the address as an argument, and fetches a new address (also two bytes) from the location the argument pointed to. Sets PC to this second address.
- **Indexed indirect:**
  Takes one byte of argument. Register X is added to this argument (ignoring carry) and a new 2 byte address is fetched from the location this sum points to. The instruction acts on the value in memory at this new address.
- **Indirect indexed:**
  Takes one byte of argument. Fetches a 2 byte value from where that argument points to and adds register Y to this value to calculate the actual address. The instruction acts on the value in memory at this new address.

For the most part, any sensible combination of an instruction and addressing mode is recognized (some examples of instructions understandably not recognized are ADC with implicit addressing or STA with immediate addressing).

B. Execution cycles

The basic principle for determining how many clock cycles an instruction takes is one cycle per memory access. For example an instruction with Zero Page addressing needs to fetch the instruction, fetch the one byte of address and fetch the value from that address, which all takes 3 clock cycles. Performing the actual operation on the data (e.g. an ALU operation) is in most cases pipelined i.e. executed during the fetch stage of next instruction.

There are, however, exceptions from this rule. Some of these are easily understandable, for example instructions with implicit addressing still take 2 clock cycles, because they need to go through both fetch and decode stages. One less obvious reason an instruction might take more cycles is page crossing. Page crossing might happen when a instruction acts on a 16 bit address calculated at runtime (e.g. Absolute X addressing). Because of the 8 bit word size, the added value is always a single byte, so such calculations are first carried out for just the lower byte of the address. In many cases this will be enough to calculate the address (i.e. there is no overflow from the addition), so the processor predicts that and starts fetching from the address. In the case of overflow, that fetched value is discarded, the higher half of the address is incremented and fetched in the next cycle. This prediction cannot, however, be

done for store instructions, because the processor cannot store a wrong value in memory.

C. Control logic

The control logic is implemented in Verilog as a finite state machine with states corresponding to sets of actions that the processor might do in a clock cycle. Most of the states focus primarily on fetching a value from memory and deciding the next state to move to. There is also a number of states which are often more general (i.e. used by more different instructions) and perform more actions like using the ALU or storing a value in memory.

Many of the states are reused for multiple instructions to reduce the size of the processor and avoid large amounts of repeated code. The tradeoff for this decision is the more complicated logic relative to a design using a separate set of states for each instruction-addressing mode combination (e.g. deciding what state to move to requires careful consideration in the first, but is very straightforward in the second approach).

D. Bus

One major (but necessary on an FPGA) design difference from the original is the lack of buses. The FPGA does not allow implementation of tristate logic, which is necessary for a bus. Instead, our implementation simply assigns values to registers based on current state and instruction, which is equivalent to using muxes wherever input would normally be taken from the bus.

VII. Authentication

The goal of the authentication module is to address the threat model of unauthorized users who may try to run unauthorized code on the processor. We aim to authenticate a user prior to the user being able to run software. With the authorization module, we are able to address the issue of processor security as well as adding more modern functionality to an older processor. The authorization module uses SHA-2 with standard 256 bit-length output. SHA-2 is also commonly known as SHA-256.

A. Overview

Hash functions are functions that map data to an output, typically a fixed sized output. Hash functions need to be collision resistant, meaning that two inputs should only have a very small probability of hashing to the same value. A one-way hash is a hash function that is easy to compute in the forward direction, but very hard to compute in the reverse direction. This means it's very easy given an input to calculate the hash of the input, but given a hash, it's very difficult to compute the input that resulted in the hash. While hash functions have many applications, in this context they are used for security purposes.

Passwords are crucial to maintaining security. We aim to keep our processor secure by only allowing software to be run after the correct password has been entered. However, the issue of password storage still remains. If we store a password in its original state on the processor, it's possible for an adversary to determine the password by looking at the source code. To solve this issue, we store the hash of the password. We compare the hash of any input password to the correct hash to determine authorization.

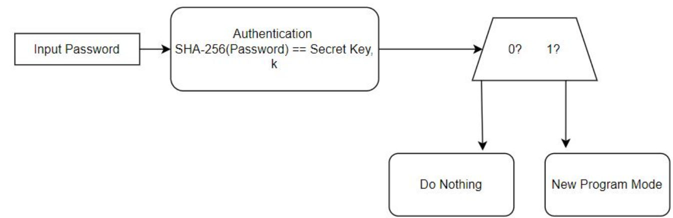

Fig. 3. Connecting authentication to the rest of the processor. If the password is correct, the processor is allowed to run.

The authentication module follows the SHA-256 algorithm, a cryptographic hash function. A cryptographic hash function is one-way, collision resistant, and a small change in the input results in a very different output hash. These properties make it difficult to determine the password through brute forcing random inputs.

B. Design

We followed the standard SHA-256 implementation with one key modification. Typically, SHA-256 is able to hash an input of any size to the 256-bit length output. This is regulated by the preprocessing stage where the number of zeros appended ensure that the output is a multiple of 512. Then, it divides the entire preprocessed input into chunks of 512 and performs hashing on each chunk. However, we opted to limit our implementation to one iteration on a preprocessed input of length 512 to prevent increasing latency and BRAM usage. This limited input password size to 446 bits, which still provides 2^{446} bits of variability, and is enough to be secure.

SHA-256 was chosen due to the constraint on size and iterations. While there are newer variants of cryptographic hash functions that are more secure, they require more iterations and/or more memory usage. Other implementations may require memory access which would add a minimum of two clock cycles per iteration. This implementation balanced security, memory usage, and latency.

C. Inputs and Outputs

Inputs

- Clock: The clock driving the module. Operates on the same clock as the rest of the processor.
- Reset: Input to reset the module. Used when receiving a new password to reset the hash.
- Password: An input of length determined by the parameter password length. The input to hash

Outputs

- Valid: 1 if the input password hashes to the correct hash, 0 otherwise. Checks if the input password is correct.
- Hashed Password: Hashed input password of standard size 256 bits.

Parameters

- Password Length: Length of the input password. Cannot be more than 446 bits due to the constraints of the module.
- Correct Hash: The hash of the correct password, stored as a parameter

D. Authentication Finite State Machine

The authentication module is a state machine with several main stages: Preprocessing, Initialization, Calculation, Compression, and Reporting.

- Overall FSM

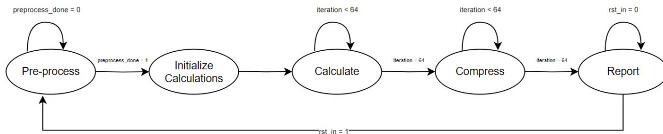

Fig. 4. Module FSM with five main states

  - Preprocessing: The input password must be processed so that the rest of the module is able to operate on a standard size input. Preprocessing is done by appending a 1 to the original input, then enough 0s to make this intermediate 448 bits. Then the length of the input password is appended as a 64 bit integer such that the final preprocessed input is 512 bits. In the module, preprocessing takes one clock cycle to append the necessary information to the original input and report the output to the rest of the module.

  - Initialization: SHA-256 uses a message schedule of 64 32-bit messages to operate on. The first 16 messages are obtained from the preprocessed input. The preprocessed input is divided into 16 32-bit messages and placed into the message schedule. As the message schedule is stored in BRAMs, each individual message takes two clock cycles to be written into memory, resulting in 32 clock cycles for the entire preprocessed input to be placed into memory.

  - Calculation: This is the main calculation portion of the hash function. For the remaining messages on the message schedule, each message is calculated as follows:
    s0 = (w[i-15] rightrotate 7) xor (w[i-15] rightrotate 18) xor (w[i-15] shift right 3)
    s1 = (w[i-2] rightrotate 17) xor (w[i-2] rightrotate 19) xor (w[i-2] shift right 10)
    w[i] = w[i-16] + w[i-7] + s0 + s1
    where w[i] is the current message and w[i-x] is the xth previous message. For example, to update the 17th message, calculate s0 = (2nd message rightrotate 7) xor (2nd message rightrotate 18) xor (2nd message shift right 3) and so on. The indexing of w is used to determine which address to access the BRAM at, such that w[3] indicates access BRAM at address 3.
    Each message retrieval takes two cycles to access from BRAM and must be written to a register to store the values to be used in calculation, resulting in each retrieval state taking three cycles. This is done four times per iteration. The calculation state has substates to manage the retrieval of each message within an iteration. The calculation state additionally divides the message retrieval stage from the calculation stage.

  - Compression: The compression part takes in the messages calculated previously and produces 8 32-bit messages. The new messages are a function of constants and outputs from the previous calculation stage. Compression iterates 64 times, with each iteration updating the 8 32-bit messages. Similar to the calculation state, the compression state has substates to manage data retrieval, as the constants are stored in a BRAM as well. For each iteration, data retrieval and writing to a different register takes three cycles, and three more cycles are needed to finish the compression iteration.

  - Reporting: When the hash has been computed, this state reports whether the final computed hash matches the parameter correct hash and reports the final computed hash as well. The final hash is computed as a concatenation of the 8 32-bit messages from the compression state.

- FSM - Calculation

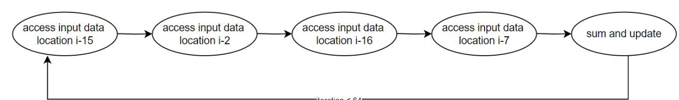

Fig. 5. The first four states within calculation are responsible for obtaining 32-bit messages from the calculation memory while the last one calculates the resulting 32-bit message

The calculation portion of the authentication module requires accessing the calculation memory for four values. As we opted to use a single port BRAM, this required four separate states to obtain each value from memory and store the value in a register. For further optimizations, we calculate s0 and s1 when w[i-15] and w[i-2] are ready. The last state is where w[i] is calculated as s0 + s1 and then written to the calculation memory at address i. The calculation stages are iterated through 48 times.

- FSM - Compression

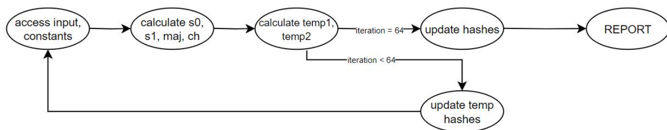

Fig. 6. The compression state accesses memory then updates the initial hashes iteratively to effectively mix the results from the calculation stage

The compression portion of the module relies on predetermined constants. One set of predetermined constants are the initial hashes, h0-h7. Another set of predetermined constants are the round constants used in each round of compression. These are stored in memory as 64 32-bit constants. Both the hash constants and round constants are standard to the SHA-256 algorithm.

The calculations involved in the compression stage are divided into two separate stages because the calculation of temp1 and temp2 depend on the results from the previous stage. In each iteration, the temporary hashes are updated with the results of the previous iteration and the calculated results of temp1 and temp2. After 64 iterations, the 8 hash values are updated with the temporary hashes.

E. Memory Usage

The calculation state results in 64 32-bit messages and the compression state requires 64 32-bit constants. We opted to use BRAMs to store this information to avoid using too many logics and registers and causing timing issues. The constants are stored in a single-port read only BRAM and are initalized with a file containing the constants. Read-only ensures that no accidental writes will occur to protect data integrity.

The results from the calculation state are stored in a single-port read and write BRAM. We limit write access to only certain states. One design consideration is that increasing the number of BRAMs to two and making each BRAM dual port would reduce the number of cycles needed during the calculation state by allowing for four simultaneous reads. However, the priority in the authentication module was accuracy and by extension, data integrity. To avoid any chance of race conditions or mismatched data across BRAMs, only one BRAM was used.

The authentication module is run very infrequently and optimizations made on latency would be unnoticeable to the user. The current implementation results in a hash in 15200 nanoseconds. Consequently, the constraints on BRAM usage are greater than the constraint on module latency.

F. Verification

The module itself uses big endian, which is compatible with the rest of the processor. However, the module was converted to little endian for testing purposes. Testing the module relied on using a previously developed C program [8] that implemented SHA-256 but only runs in little endian. We used the C program to verify the intermediates of the authentication module. While the implementation still has slight inconsistencies, the outputs of the module are deterministic and incorporate the avalache effect, as explained below.

One key property of cryptographic hashes is that a small change in the input results in a very different hash. This is known as the avalanche effect. This is demonstrated below with a small change in the input by changing the first bit from a 0 to a 1 resulting in a very different output hash.

```
input password: 6162636462636465636465666465666765666768666768696768696a68696a6b696a6b6c6a6b6c6d6b6c6d6e6c6d6e6f6d6e6f70
valid 1
output hash: e371971a2d1bcbe83efb33b2cbde91fc0fa43300cec0a90ab6f2a5d9917627bd

input password: e162636462636465636465666465666765666768666768696768696a68696a6b696a6b6c6a6b6c6d6b6c6d6e6c6d6e6f6d6e6f70
valid 0
output hash: b36bcfc47feb3e0aa3f4c16771db688d98287832da3a2504d73fda710080f7a6
```

Fig. 7. Changing one bit in the input results in a very different and incorrect hash

VIII. Manta

In order to interact with our system (run programs, pass inputs, receive outputs) we use a manta module, which communicates with the processor using MMIO. The module controls the reset and hold signals for the processor, the password and the inputs to the second memory port. It takes in the output of the memory port and the result of authentication (accepted or rejected). Some additional inputs and outputs were included for debugging purposes, but are not necessary and can safely be removed.

A typical interaction with the processor starts with halting and resetting the system using appropriate signals. Then, the password is passed to the authentication module. After the password is accepted, the program and its inputs are written to memory byte by byte using the second way. Finally, the halt signal is set to 0 and the processor is allowed to run the program. After it is finished, any output written to memory can also be retrieved using manta.

IX. Testing

The testing of our system was done in the following stages:

1) Separately testing all of the modules:
   This includes the ALU and the authorization module. The testing for these is described in their respective sections.
2) Unit testing in simulation:
   We developed a set of small programs aiming to test specific instructions and addressing modes. These programs where then assembled using xa (an open-source 6502 assembler) and used as the initial values of the memory. After running the program, the result is placed in the accumulator register and verified against the expected output.
3) Unit testing on hardware:
   After finishing the entire design, the same set of unit tests was used on hardware to check for any errors not caught by the simulation.
4) Testing complex programs:
   Lastly, some longer and more complex programs were run on the processor to check if it behaves correctly in a

more realistic setting. These programs were taken from the internet and were written for the original 6502.

Throughout the testing process we developed a set of scripts in python and bash to make it more efficient and easily replicable. These include:

- assemble.sh - takes a 6502 assembly program as an argument and uses the xa assembler to produce the binary in a text format which can be used as initial memory state
- test_single_instruction.sh - runs through all unit tests either in simulation or on hardware
- libtalk.py - initializes manta and implements functions for easy interfacing with the processor (e.g. read_mem(), write_mem())

X. Discussion

We evaluate our system based on latency, efficiency, and correctness. We aim for a processor that is fast and correct, and does not exceed resource capabilities within the FPGA. As discussed previously, we have tested the correctness of our processor with test suites and benchmarks.

In terms of memory usage, the design is very optimal. One key design choice with regards to memory is storing constants in BRAMs in the authentication module. This is better than having 64 parameters and/or logics of 32 bits each. Storing the intermediate calculation values from the authentication module incurred extra BRAM usage as well but is more optimal in terms of resource constraints. In total, we use one dual port BRAM with an 8-bit width and a 2^{16}-bit depth for the main processor, and two single port BRAMs with a 32-bit width and a 64-bit depth for the authentication module. The total usage is approximately 0.528 Mbits which is well within the 2.7 Mbits provided on the FPGA.

The main processor adheres to the original MOS6502 cycles per instruction and the original timing specifications as well. While the FPGA is run on a 100 MHz clock, the original processor uses a 1 MHz clock. To achieve this, we implement an internal signal that activates every 100 cycles to change state.

We evaluate the main processor overall by examining basic requirements such as worst negative slack (WNS) and total negative slack (TNS). Every WNS was positive with zero total negative slack when run with a 100 MHz clock.

Additional custom modules result in increased latency. One key source of added latency in the system is the security module. The module itself takes approximately 15200 clock cycles to complete, both calculated and verified in simulation. While this is a very large overhead, the authentication module is only run once each time a new program is loaded to the processor. However, despite the overhead, the module provides the added benefit of security and provides a safeguard against malicious users.

Overall, the processor covers the minimal goal of having a working MOS 6502 processor. The processor is able to run programs in MOS 6502 assembly. Additionally, the security module addresses the threat model of unauthorized users and covers scenarios that require a secure processor. An example would be the processor memory storing confidential data with an unauthorized user that runs a program to retrieve the data. This implements our ideal goal of adding the specialized security module. We modified our original stretch goal from pipelining to producing a visual component. Although the processor is not fully equipped with visuals, such as HDMI, the processor is capable of displaying outputs from programs through the use of manta.

By implementing a processor on FPGA, the processor is easily customizable and additional modules that extend use cases could be added. One example is cached memory to allow for optimizing programs that require a lot of repeated memory access. Another example is including a data compression module that allows for storing large quantities of data. A data compression module could be added by taking in a large input and writing it to a specialized BRAM. This could be attached to the processor in a straightforward manner.

Implementation-wise, a key change would be to implement more separate sub-modules for the SHA-256 module. Preprocessing was done separately in case there was a need to add more stall cycles due to the large input. However the other states were not implemented in separate modules. This would have made it easier to debug, rather than trying to debug a very large system with many iterations within.

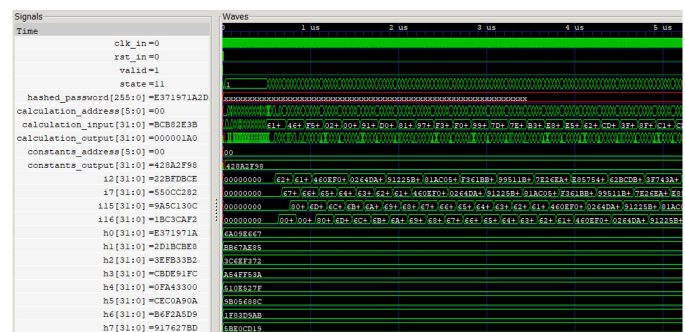

Fig. 8. This is approximately a third of the uut variables involved, and there are two more BRAMs with 64 addresses not shown here. Additionally, approximately 15 microseconds are required for the hash to complete.

Additionally, it's important to consider endianness while interfacing and testing outside of the immediate system. Other systems that the processor may be interfacing with may not have the same endianness.

XI. Contributions

While this was a group effort, we each contributed to the processor implementation in different ways. Kelly implemented the ALU module, security module and provided evaluation and overviews of the system. Jan implemented the control logic, integrated all the modules, and created the interface for testing and running programs on the processor through manta.

XII. Source Code

Source code is here: https://github.mit.edu/jstrz/6502-fpga

XIII. Acknowledgements

We want to thank the 6.2050 staff members for their support during the class. Thank you to our professor Joe Steinmeyer for providing us with the information to build our own processor on FPGAs and providing feedback on our designs. We also want to thank our TA Darren for helping us get our processor working and providing guidance on our design.

References

[1] http://6502.org
[2] http://6502.org/tutorials/decimal_mode.html
[3] http://6502.org/tutorials/vflag.html
[4] http://6502.org/users/obelisk/6502/reference.html
[5] https://archive.org/details/mos_microcomputers_programming_manual
[6] https://fischermoseley.github.io/manta/
[7] https://www.geeksforgeeks.org/bcd-adder-in-digital-logic/
[8] https://github.com/amosnier/sha-2
[9] https://en.wikipedia.org/wiki/SHA-2
[10] https://llx.com/Neil/a2/opcodes.html

XIV. Appendix

A. Instructions

Instructions implemented in the MOS6502 processor

TABLE I: Instructions

Instruction	Description
ADC	$ACC <= ACC + MEM + CFLAG$
SBC	$ACC <= ACC - MEM - (1 - CFLAG)$
AND	$ACC <= ACC \& MEM$
ORA	$ACC <= ACC \| MEM$
EOR	$ACC <= ACC \oplus MEM$
ASL	$MEM <= MEM << 1$ OR $ACC <= MEM << 1$
LSR	$MEM <= MEM >> 1$ OR $ACC <= ACC >> 1$
ROL	$\{CFLAG, ACC\} <= \{ACC, CFLAG\}$ OR $\{CFLAG, MEM\} <= \{MEM, CFLAG\}$
ROR	$\{ACC, CFLAG\} <= \{CFLAG, ACC\}$ OR $\{MEM, CFLAG\} <= \{CFLAG, MEM\}$
DEC	$MEM <= MEM - 1$
DEX	$X <= X - 1$
DEY	$Y <= Y - 1$
INC	$MEM <= MEM + 1$

Continued on next page

TABLE I: Instructions (Continued)

Instruction	Description
CMP	$FLAGS <= ACC - MEM$
CPX	$FLAGS <= X - MEM$
CPY	$FLAGS <= Y - MEM$
BIT	$FLAGS <= ACC \& MEM$
LDA	$ACC <= MEM$
LDX	$X <= MEM$
LDY	$Y <= MEM$
STA	$MEM <= ACC$
STX	$MEM <= X$
STY	$MEM <= Y$
JMP	$PC <= MEM$
JSR	$STACK <= PC$ $PC <= MEM$
RTS	$PC <= STACK$
BCC	if $CFLAG = 0$ then $PC <= PC + MEM$
BCS	if $CFLAG = 1$ then $PC <= PC + MEM$
BNE	if $ZFLAG = 0$ then $PC <= PC + MEM$
BEQ	if $ZFLAG = 1$ then $PC <= PC + MEM$
BPL	if $NFLAG = 0$ then $PC <= PC + MEM$
BMI	if $NFLAG = 1$ then $PC <= PC + MEM$
BVC	if $VFLAG = 0$ then $PC <= PC + MEM$
BVS	if $VFLAG = 1$ then $PC <= PC + MEM$
BRK	trigger interrupt
RTI	retrieve values from stack after interrupt
PHA	$STACK <= ACC$
PHP	$STACK <= FLAGS$
PLA	$ACC <= STACK$
PLP	$FLAGS <= STACK$
TAY	$Y <= ACC$
TYA	$ACC <= Y$
TAX	$X <= ACC$
TXA	$ACC <= X$
TSX	$X <= SP$
TXS	$SP <= X$
CLC	$CFLAG <= 0$
CLD	$DFLAG <= 0$
CLI	$IFLAG <= 0$
CLV	$VFLAG <= 0$

Continued on next page

TABLE I: Instructions (Continued)

Instruction	**Description**
SEC	$CFLAG <= 1$
SED	$DFLAG <= 1$
SEI	$IFLAG <= 1$
NOP	do nothing