535 Project Final Report

Shankar Subramanian and Stelios Papoutsakis

Architecture Overview:

Our architecture is a stack based machine with emphasis on crypto math instructions. It is developed as a clean slate architecture. Our inspiration is from the Bitcoin/Ethereum stack machines.

Word size:

32 bits

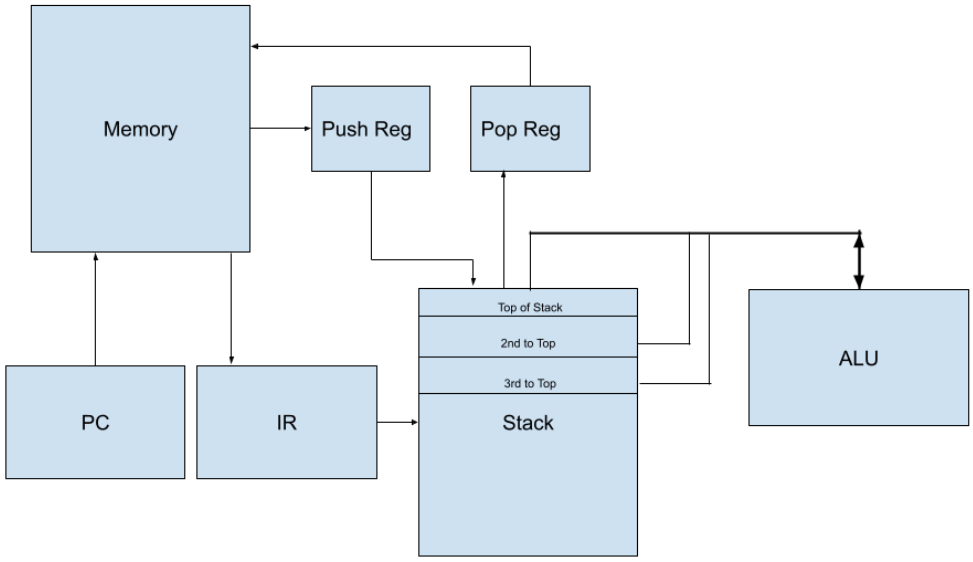
Instruction size:

8 bits

Supported data types:

1. Unsigned Integer (32 bit)
2. Block for AES (128 bit)
3. Block for RSA (64 bit)
4. Group Element (64 bit)

Stack Machine Organization



As shown above, the ALU is able to operate on the top three stack elements. It then writes its computation directly to the top of the stack. The push and pop registers are used to access and write values back to memory. If the stack overflows, the oldest element inserted into the stack (element 32) is overwritten.

Special Registers:

1. Program Counter
2. Offset Register
3. 32x 128 bit stack registers as a register file
4. Pop Register (128 bit)
5. Push Register (128 bit)
6. Link Register

There are no general purpose registers in our design.

The offset register contains the byte offset (high to low) for an instruction in the word pointed to by PC

The execution model is a Stack Based Machine, fetching four instructions per word.
We will be using the Princeton memory organization.
Our address space would be 65532 (2^{16}) words, though not all of it will be used in simulation.
We will be doing word addressing.

We will use indirect addressing for all instructions, with the address being read from the stack.

Pipeline:

Our architecture features a 3 stage pipeline, described below

Fetch: Fetches instruction based on PC and OFFSET. Issues a byte to decode

Decode: Decodes the byte issued from fetch. Pops off operands from the stack if necessary and classifies the instructions as “Alu”, “Branch”, “Memory” or “Crypto”

Execute: Executes the instruction from Decode. Based on the classification of the instruction it feeds it to the appropriate executor module. It uses a lookup table to identify how long certain non memory instructions should block for (memory instructions block based on the status of memory). Pushes any results generated immediately to the stack.

We opted for a 3 stage pipeline because of the high hazards a 5 stage one would provide. This is because most operations always write to the top of the stack, which would result in the top of stack being a pending register. This would result in the decode stage stalling often as the decode stage is responsible for popping operands from the top of stack. Modifying execute to write to the top of stack removed this hazard

Cache:

We used a direct mapped write through no allocate cache. We only have one level of cache. The cache size is 16 lines or in other words 64 bytes.

Instructions:

There are two types of instructions, ones that operate with immediate values and ones that exclusively use the stack

Since a majority of our instructions use the top stack values as input and push a new value on completion, there is no need to define multiple type fields that specify different operand formats. The amount of pops used for operands are defined in the implementation of the instruction itself.

Any value used by an instruction (specified by stack Input:) is removed from the stack as part of its execution. Additionally, instructions will use however many bits they need starting from the least significant bits. For example, unsigned integer arithmetic will only use 32 out of the 128

bits a stack register has as an operand value. Output lengths are specified in the sub sections below. Any output less than 128 bits is extended with 0s.

Instructions With Immediates (Type 0):

Format:

	7	6	5	4	3	2	1	0
Num	TYPE	OPCODE		Stack Position/Immediate				
0	0	PUSH_VAL		INTEGER_LSB				
1	0	SWAP		INTEGER_LSB				
2	0	DUP		INTEGER_LSB				

Opcode: 0
Mnemonic : PUSH_VAL
Stack Input:
Stack Output: Immediate
Operation: Pushes the immediate value on top of the stack. The upper 122 significant bits are extended with 0s

Opcode: 1
Mnemonic : SWAP
Stack Input:
Stack Output:
Operation: Takes the value in the stack index by im mediate, removes it and pushes it to the top of stack

Opcode: 2
Mnemonic : DUP
Stack Input:
Stack Output:
Operation: Takes the value in the stack index by immediate, duplicates it and pushes it on top of the stack

Instructions OPCODE Only (Type 1):

Format:

	7	6	5	4	3	2	1	0
Num	TYPE	OPCODE						
0	1	LDR_32						
1	1	STR_32						
2	1	LDR_64						
3	1	STR_64						
4	1	LDR_128						
5	1	STR_128						
6	1	PUSH						
7	1	POP						

	7	6	5	4	3	2	1	0
Num	TYPE	OPCODE						
8	1	JMP						
9	1	JMP_if_1						
10	1	JMP_if_0						
11	1	SR						
12	1	RET						

	7	6	5	4	3	2	1	0
Num	TYPE	OPCODE						
13	1	ADD						
14	1	MUL						
15	1	SUB						
16	1	DIV						
17	1	MOD						
18	1	AND						
19	1	OR						
20	1	XOR						
21	1	NOT						
22	1	EQ						
23	1	EQ_0						
24	1	GEQ						
25	1	LEQ						
26	1	GT						
27	1	LT						
28	1	R_SHIFT						
29	1	L_SHIFT						

	7	6	5	4	3	2	1	0
Num	TYPE	OPCODE						
47	1	NOOP						
48	1	HALT						
50	1	DUP_TOP						
51	1	SWAP_TOP						

	7	6	5	4	3	2	1	0
Num	TYPE	OPCODE						
30	1	GCD						
31	1	LCM						
32	1	RAND_INT						
33	1	RAND_GRP						
34	1	RABIN						
35	1	MOD_ADD						
36	1	MOD_MUL						
37	1	MOD_INV						
38	1	DH						
39	1	RSA						
40	1	AESE						
41	1	AESD						
42	1	RAND_BLK						
43	1	XOR_BLK						
44	1	SCHNORR_SIG						
45	1	SCHNORR_VER						
46	1	SHA256						

Load/Store instructions:

Opcode: 0
Mnemonic : LDR_32
Stack Input: Address
Stack Output: None
Operation: Loads a word from memory into the PUSH_REGISTER register at the address on the top of the stack.

Opcode: 1
Mnemonic : STR_32
Stack Input: Address
Stack Output: None
Operation : Stores the 32 LSB in the POP_REGISTER at the address on the top of the stack.

Opcode: 2
Mnemonic : LDR_64
Stack Input: Address
Stack Output: None

Operation: Loads two consecutive words from memory into the PUSH_REGISTER register at the address on the top of the stack.

Opcode: 3
Mnemonic : STR_64
Stack Input: Address
Stack Output: None
Operation: Stores the 64 LSB value in the POP_REGISTER at the address on the top of the stack.

Opcode: 4
Mnemonic : LDR_128
Stack Input: Address
Stack Output: None
Operation: Loads four consecutive words from memory into the PUSH_REGISTER register at the address on the top of the stack.

Opcode: 5
Mnemonic : STR_128
Stack Input: Address
Stack Output: None
Operation: Stores the 128 LSB value in the POP_REGISTER at the address on the top of the stack.

Opcode: 6
Mnemonic : PUSH
Stack Input: None
Stack Output: Literal at top of the stack
Operation : Pushes the value located in the PUSH_REGISTER to the top of the stack.

Opcode: 7
Mnemonic : POP
Stack Input: Value at the top of the stack
Stack Output: None
Operation: Pops the value at the top of the stack into the POP_REGISTER

Control instructions/modes:

Opcode: 8
Mnemonic : JMP
Stack Input: Address
Stack Output: None
Operation: An unconditional jump to the address at the top of the stack.

Opcode: 9
Mnemonic : JMP_if_1
Stack Input: condition Address
Stack Output: None
Operation: Tests to see if the top of stack's least significant bit is 1. Jumps to address stored at the second from the top of stack if true

Opcode: 10
Mnemonic : JMP_if_0
Stack Input: condition Address
Stack Output: None
Operation: Tests to see if the top of stack's least significant bit is 0. Jumps to address stored at the second from the top of stack if true

Opcode: 11
Mnemonic : SR
Stack Input: Address
Stack Output: None
Operation : Places current address in link register and sets the PC to address on the top of stack.

Opcode: 12
Mnemonic : RET
Stack Input: None
Stack Output: None
Operation: Replaces the PC with the value located in the link register

Unsigned Integer ALU Instructions:

All operations result in 32 bit integer values. The 33rd bit is used as overflow if applicable. The exception is MULT which returns a 64 bit product.

Opcode: 13
Mnemonic: ADD
Stack Input: num1 num2
Stack Output: sum
Operation: Added the top two values on the stack and push their sum

Opcode: 14
Mnemonic: MUL
Stack Input: num1 num2
Stack Output: mult
Operation: Multiply the top two values on the stack and push a 64 bit product. Note that the stack uses 128 bit registers

Opcode: 15
Mnemonic: SUB
Stack Input: num1 num2
Stack Output: sub
Operation: Subtract the top two values on the stack and push their difference.

Opcode: 16
Mnemonic: DIV
Stack Input: num1 num2
Stack Output: quo
Operation: Divide the top two values on the stack and push the quotient. If the num2 is 0, this operation pushes zero onto the stack

Opcode: 17
Mnemonic: MOD
Stack Input: num mod
Stack Output: rem
Operation: Take the value from the 2nd top of stack to use as a mod value and use the top of stack as divisor. Pushes the result on top of the stack

Opcode: 18
Mnemonic: AND
Stack Input: num1 num2
Stack Output: res
Operation: Bitwise AND the top two values of the stack. Push the result on the top of the stack.

Opcode: 19

Mnemonic: OR
Stack Input: num1 num2
Stack Output: res
Operation: Bitwise OR the top two values of the stack. Push the result on the top of the stack.

Opcode: 20
Mnemonic: XOR
Stack Input: num1 num2
Stack Output: res
Operation: Bitwise XOR the top two values of the stack. Push the result on the top of the stack.

Opcode: 21
Mnemonic: NOT
Stack Input: num
Stack Output: res
Operation: Invert the top value of the stack.. Push the result on the top of the stack.

Opcode: 22
Mnemonic: EQ
Stack Input: num1 Num2
Stack Output: condition
Operation: Pushes 1 onto the stack if the top two values are equal, else pushes 0

Opcode: 23
Mnemonic: EQ_0
Stack Input: num1
Stack Output: condition
Operation: Pushes 1 onto the stack if the top value equals 0, else pushes 0

Opcode: 24
Mnemonic: GEQ
Stack Input: num1 Num2
Stack Output: condition
Operation: Pushes 1 onto the stack if num1 is greater than or equal num2, else pushes 0

Opcode: 25
Mnemonic: LEQ
Stack Input: num1 Num2
Stack Output: condition
Operation: Pushes 1 onto the stack if num1 is less than or equal num 2, else pushes 0

Opcode: 26
Mnemonic: GT
Stack Input: num1 Num2

Stack Output: condition
Operation: Pushes 1 onto the stack if num1 is greater than num2, else pushes 0

Opcode: 27
Mnemonic: LT
Stack Input: num1 Num2
Stack Output: condition
Operation: Pushes 1 onto the stack if num1 is less than num2, else pushes 0

Opcode: 28
Mnemonic: R_SHIFT
Stack Input: pad num
Stack Output: shifted_num
Operation: Right shifts the element of the stack at depth 2 by the value at the top of the stack, appends 0 as pads to the left end.

Opcode: 29
Mnemonic: L_SHIFT
Stack Input: pad num
Stack Output: shifted_num
Operation: Left shifts the element of the stack at depth 2 by the value at the top of the stack, append 0 as pads to the right end.

Opcode: 30
Mnemonic: GCD
Stack Input: a b
Stack Output: gcd
Operation: Computes the GCD of the top 2 elements of the stack

Opcode: 31
Mnemonic: LCM
Stack Input: a b
Stack Output: lcm
Operation: Computes the LCM of the top 2 elements of the stack

Opcode: 47
Mnemonic: NOOP
Stack Input:
Stack Output:
Operation: Sends a NOOP through the pipeline

Opcode: 48
Mnemonic: HALT

Stack Input:
Stack Output:
Operation: Halts the pipeline's execution until run is triggered.

Opcode:50
Mnemonic: DUP_TOP
Stack Input: stack_location
Stack Output: element at stack location
Operation: Computes the LCM of the top 2 elements of the stack

Opcode: 51
Mnemonic: SWAP_TOP
Stack Input: stack_location
Stack Output:
Operation: Takes the value in the stack index by immediate, and swaps it with the element at index 1 in the stack

Crypto instructions:

Opcode: 32
Mnemonic: RAND_INT
Stack Input: None
Stack Output: rand integer
Operation: Adds onto the stack a random 32 bit integer

Opcode: 33
Mnemonic: RAND_GRP
Stack Input: None
Stack Output: rand group integer
Operation: Adds onto the stack a random 64 bit integer

Opcode: 34
Mnemonic: RABIN
Stack Input: a
Stack Output: 1 or 0
Operation: Checks if the number on top of the stack is a prime or not.

Opcode: 35
Mnemonic: MOD_ADD
Stack Input: a b n
Stack Output: sum
Operation: Perform ADD and then modulus with the 3rd element in the stack.

Opcode: 36
Mnemonic: MOD_MUL
Stack Input: a b n
Stack Output: prod
Operation: Performs MUL and then modulus with the 3rd element in the stack.

Opcode: 37
Mnemonic: MOD_INV
Stack Input: a b n
Stack Output: The inverse modulus as an unsigned integer
Operation: Takes the top 3 elements of the stack as (group element, group element, modulo value) and computes the inverse modulus.

Opcode: 38
Mnemonic: DH
Stack Input: a b n
Stack Output:key
Operation: Takes in the top 3 elements of the stack as (group element, group element, modulo value) and computes the diffie-hellman key.

Opcode: 39
Mnemonic: RSA
Stack Input: m e n
Stack Output: The RSA encryption/decryption as an unsigned integer
Operation: Performs m raised to e and modulo in n

Opcode: 40
Mnemonic: AESE
Stack Input: m k
Stack Output: c
Operation: Computes the AES encryption of the top most value and uses the second from the top value as the key.

Opcode: 41
Mnemonic: AESD
Stack Input: c k
Stack Output: m
Operation: Computes the AES decryption on the top most value and uses the second from the top value as the key

Opcode: 42
Mnemonic: RAND_BLK
Stack Input:

Stack Output: rand
Operation: Generates a block (128 bit) random integer

Opcode: 43
Mnemonic: XOR_BLK
Stack Input: a b
Stack Output: m
Operation: Computes the XOR of the top 2 blocks on the stack

Opcode: 44
Mnemonic: SCHNORR_SIG
Stack Input: m
Stack Output: sig
Operation: Computes the Schnorr signature of the topmost value of the stack with the key in the key register.

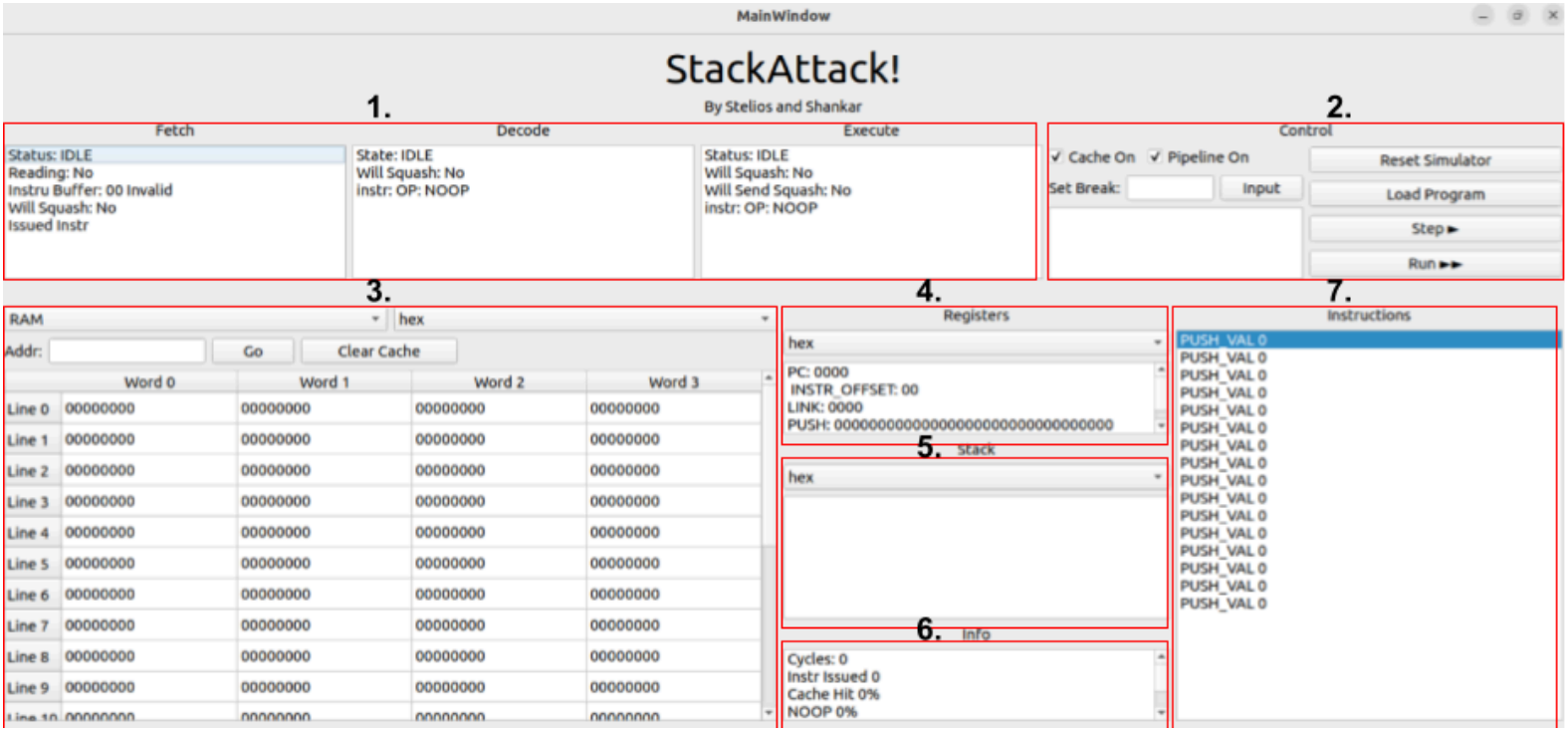
Opcode: 45
Mnemonic: SCHNORR_VER
Stack Input: sig pub
Stack Output: 1 or 0
Operation: Verifies if the data at the top of the stack passes through the schnorr verification test with the public key as the 2nd element in the stack.

Opcode: 46
Mnemonic: SHA256
Stack Input: m
Stack Output: SHA256 hash
Operation: Hashes (SHA256) the topmost element of the stack and puts it onto the stack

Crypto Demo:

We plan on adding an ascii encoded mode in our GUI for the RAM + Cache. This will help us with our AES, RSA, schnorr signatures, and Hashing demo, which will show a simple message being encrypted, decrypted, hashed and signed.

Simulator:



Gui Description

Below is an explanation of the 7 main components

1. Pipeline State: These windows show the internal state of each stage in the pipe. Information displayed includes status, squashing state, and instruction currently worked on
2. Control Panel: This is where the user controls the simulation. This includes the ability to toggle the cache and pipeline, load a binary program, reset the simulator state, step through a program, or run a program. This is also where the user defines breakpoints in the format “ADDR, OFFSET”, where ADDR is the address of the word containing the instruction and OFFSET is the byte within the word (highest to lowest). The user may double click on a displayed breakpoint to remove it
3. Memory Panel: This displayed the memory. A user may type an address (base 10 only), into the search box and 20 lines starting at the requested address will be displayed. The user can switch between ram and cache change with the left combo box and change data representation with the right. The user may also clear the cache at any point
4. Register Panel: Displays the register values. The user may change the data representation with the combo box
5. Stack Panel: Displays the stack. The user may change the data representation with the combo box

6. Info Panel: Displays statistics about the simulation. This includes cycle count, instructions issued, Cache hit rate, and percent of time spent executing each instruction type
7. Disassembled Panel: Displays disassembled binary code. The code that is displayed is taken from the line that includes the address typed into the ADDR search box in 3. If the PC is pointing to a byte that is disassembled, the gui will highlight it. Any byte which is not a valid instruction is displayed as '-'

User Manual

To run a program, a binary executable may be loaded using the "load" button in the control panel. To define breakpoints, the user may enter a value into the textbox formatted as "ADDR, OFFSET", where ADDR is the address of the word containing the instruction and OFFSET is the byte within the word (highest to lowest). The user may then use "Step" to simulate a cycle, or "Run" to simulate the program until a breakpoint or HALT instruction is reached.

SWE Methods:

Tech Stack

The simulator was written exclusively in python. Qt for Python was used for a GUI library. Qt Designer was used as a drag and drop GUI creator. Pyuic5 was used to compile the XML produced by Qt Designer into Python.

Version Control

Git and Github.

Management Plan

Stelios was more focused on the GUI, Execution stage and writing benchmarks (AES, Exchange Sort, Standard Matrix Multiplication).
Shankar worked on designing the Fetch and Decode stages, testing, and a Matrix benchmark that tried to take advantage of the stack.

Communication and Meeting Plan

Communication was done through discord. Communication occurs at least once a week which includes status updates and encountered difficulties. In person meetings were not regularly scheduled and only occurred in initial stages of pipeline design

Division of Work:

Stelios:

- GUI Creation
- Execution Stage
- Memory
- AES, Exchange sort and standard matrix benchmark

Shankar:

- Fetch
- Decode
- Testing
- Stack Matrix Benchmark

Benchmark Results:

Note that the cache delay was set to 3 cycles and main memory delay was set to 100 cycles for the benchmarks below

Matrix	Cache/Pipeline			
	On/On	On/Off	Off/On	Off/Off
Cycles	63,209.00	86,619.00	382,944.00	405,384.00
Inst Issued	12,173.00	11,873.00	12,173.00	11,873.00
Cache Hit	88.61%	88.58%	0.00%	0.00%
NOOP	33.81%	51.41%	80.28%	81.90%
ALU	17.30%	12.63%	2.86%	2.70%
BRANCH	0.51%	0.37%	0.08%	0.08%
CRYPTO	0.00%	0.00%	0.00%	0.00%
MEMORY	47.87%	34.90%	16.45%	15.17%

Matrices were of sizes:

- Matrix 1 - 20x5
- Matrix 2 - 3x20
- Resulting matrix - 3x5

The values were random positive integers generated in the range [0,100]

Insertion Sort	Cache/Pipeline			
	On/On	On/Off	Off/On	Off/Off
Cycles	1,280,274.00	1,648,448.00	7,755,889.00	7,185,444.00
Inst Issued	221,903.00	212,000.00	221,903.00	212,000.00
Cache Hit	93.36%	92.65%	0.00%	0.00%
NOOP	18.49%	39.81%	66.72%	83.70%
ALU	14.44%	11.21%	2.38%	2.57%
BRANCH	1.38%	1.07%	0.23%	0.25%
CRYPTO	0.00%	0.00%	0.00%	0.00%
MEMORY	61.46%	46.70%	17.91%	13.21%

The array used was of size 100 with random positive integers ranging from [0,1000]

AES (AESE + AESD)	Cache/Pipeline			
	On/On	On/Off	Off/On	Off/Off
Cycles	17,419.00	19,416.00	42,638.00	44,639.00
Instr Issued	1,049.00	1,020.00	1,049.00	1,020.00
Cache Hit	84.74%	84.74%	0.00%	0.00%
NOOP	13.94%	22.79%	61.79%	66.54%
ALU	5.05%	4.53%	2.06%	1.97%
BRANCH	0.17%	0.15%	0.07%	0.07%
CRYPTO	2.58%	2.31%	1.05%	1.01%
MEMORY	78.09%	69.92%	34.96%	30.29%

The message used was 1632-bits long and the key was 16-bits long.

The AES was run on CBC mode, which required a random IV (initialization vector) that was used to XOR the message block. This was generated using RAND_INT.

The benchmark covered both encryption and decryption of the message

Summary:

In conclusion, we learned that there are a lot of hazards that can impact the performance of our architecture. This is apparent with the high percentage of CPU idle time (NOOP), for every benchmark ran, even with cache and pipeline on. This figure could have been improved if we had decided to implement a longer pipe with result forwarding. This demonstrates the trade off between design complexity and performance.

We also learned that it is important to have faster memory or cache. In our exchange sort benchmark, our simulator performed worse with the pipeline when cache was turned off. This was due to the fact that fetch must finish reading an instruction once started, even if it's going to get squashed. With slow memory and optimization such as a pipeline, a lot of work is wasted. Maybe some sort of prefetcher or predictor could help solve this, or a memory which allows changing of the address during reading.