Computer Architecture
ECE 586
Application Specific Multicore Single Cycle Processor for Cryptographic block cipher algorithms
Winter 2016
Sukrut Kelkar
938621692

Introduction:
Implementation of a Multicore Single cycle processor which is designed for specific application of running Cryptographic block cipher algorithms is presented. Block Ciphers are methods used for encrypting data to produce cipher data. Block ciphers use a set of cryptographic keys with a specific algorithm which together is used on a block of data and encrypted data is obtained. This functional simulator is designed to read a memory image which contains set of opcodes converted from an assembly code which is written for a specific block cipher algorithm. Types of block cipher algorithms with symmetric keys are AES, DES, IDEA, Blowfish, and RC5 etc. The main motivation behind this project is encryption. To study how passwords or secret text is sent in encrypted formats. IDEA is commonly used for security purposes and so it was chosen for diving more into symmetric key encryption. This design is thoroughly tested for all basic instructions and IDEA algorithm is used for testing the design and for presenting the working and simulation of the design.

IDEA Algorithm:
[1][11]IDEA algorithm is one of the block cipher algorithms. It is a **symmetric key block cipher** design. This algorithm avoids use of any look-up tables or S-boxes and is widely used in many applications including security. Input to this algorithm is **128 bits of cryptographic key** and **64 bits of data**. This algorithm breaks down this keys and using a specific algorithm, 52 sub keys each 16 bits long are generated out of the key set available. It consists of total of 8 rounds and a half round at the final stage.

Key Generation:
This algorithm takes 128 bit key as the input. This key is taken and first **eight sub keys** are taken directly from the total bits. The next set of eight keys is generated after **circular shifting** the input by **25bits**. This process is repeated and all 52 sub keys are generated.

In this implementation a **python script** is written to execute this key generation and 52 sub keys are generated and stored in a text file.

Each round of IDEA needs 6 keys. First for keys at the start of the round and two keys in the intermediate stages. This algorithm needs 64 bits of data at the input. This data will be divided into 4 chunks of 16 bits. Three operations are used in this algorithm. Addition, multiplication and XOR are used where addition is 2^16 modulo addition and multiplication is 2^16+1 modulo multiplication. Special cases to be considered for multiplication modulo are that none of the input operands can be zero. **If any of the input is zero** it is taken as 65536 and if the result is zero it is considered as 65536.

One round of IDEA is computed using the following steps shown in the diagram:

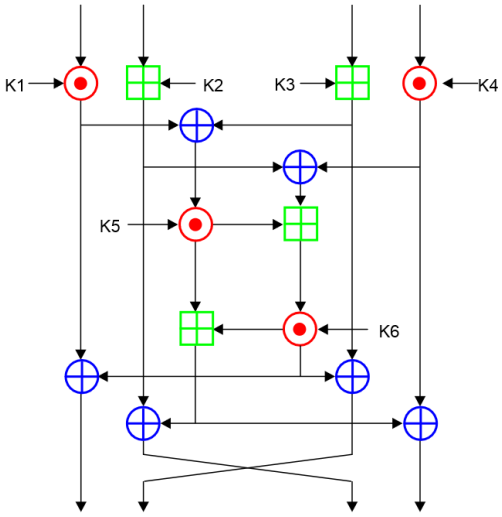
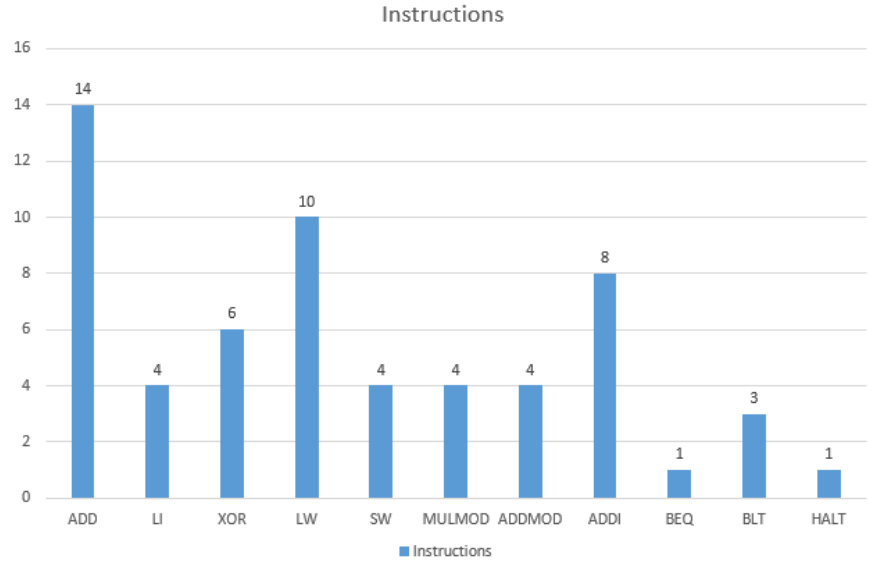


Fig 1: IDEA Algorithm[11]

K1-K6 are the keys. First four keys are operated with the four chunk of input data.
Red symbol ➔ **2^16+1 Modulo Multiplication**.
Green symbol ➔ **2^16 Modulo Addition**.
Blue symbol ➔ **XOR**.
The important thing to remember is that after each round you swap the middle two outputs as shown in the diagram and then provide the outputs as inputs to the next round to take place.
This swapping will take place for 8 rounds. Important thing to note is for the **last 0.5 rounds the swapping should not be done**.
It is called 0.5 round because the last computation is only till the first stage where we use the last four keys.
The output of this one IDEA round is a 64 bit Encrypted data.

The code for IDEA algorithm was written using a specific **ISA (Assembly Code)** described in sections to come.



Number of Occurrence of each of the instructions in the IDEA assembly code is projected by this diagram.

Fig 2

Constraints:

Input:

- Simulator must load a text file that contains **memory image** as an input.
- Data memory image should be **32 bits** wide.
- Each line in the text file represents **one word (4 bytes)** of memory shown in hexadecimal format.
- The addresses start at “0” from the first line in the image.
- The program must use **1024 bits** of input data and a **128-bit key**.

Output:

- Program must terminate when it encounters **HALT** instruction.
- Simulator output needs to be memory image that shows the encrypted data.
- Output data should be 1024 in encrypted format.

Instruction Set:

- Arithmetic
  - ADD, SUB, MUL.
- Logical (bitwise)
  - OR, AND, XOR.
- Memory access
  - LOAD, STORE.
- Control
  - BZ, BEQ, BP, BN, JR, HALT.

Program Analysis:

Comparison between design options:

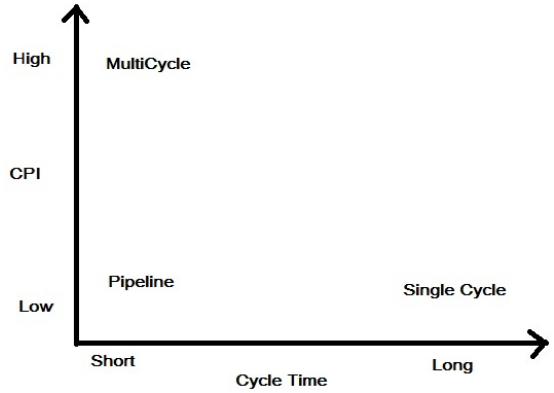


Fig 3: CPI vs Cycle time

	Single Cycle	**Multi Cycle**	**Pipeline**
CPI	1	Variable	Fixed #
Clock Cycle time	Long(Critical path instruction time)	Short (depends on particular instruction)	Short (long enough for the slowest pipeline stage)
Parallel Instructions	1	1	Depends on # of pipeline stages
Extra Registers	No	Yes	Yes (Intermediate to store states)
Performance	Moderate	Faster than single cycle for particular instructions	Faster than single and multi-cycle if Pipeline stages are well balanced.
Basic Cores	1	1	1
Cost	Cheap	Costly than single cycle	Costly than single and multicycle
Logic Complexity	Moderate	Complex	Complex

Table 1

Comparing the three models, **Single cycle model should be well suited** for block cipher implementation because Pipelined models are more complex and expensive and looking at the graph **fig 2**, there are branches in the code. IDEA implementation or any block cipher implementation has **recursive branch** operations and so handling hazards in pipeline will be a problem. Multicycle models have **High CPI** even if the cycle time is less. As IDEA algorithm or any block cipher algorithms have many Load, store and modulo operations as seen in the graph fig 2, choosing multicycle data path model will not increase the throughput to a great extent considering its complexity. Single cycle data path model on the other hand can take either the load or modulo operation as the critical path and compute its clock time. **Extra registers** and **High CPI** can be avoided by using single cycle data path model. As it is comparatively cheap

in terms of cost, many single cycle chips can be combined to form a **multicore processor** which will increase the throughput by a huge amount. Because of all these reasons, Single cycle model is chosen for this project.

Processor Specifications:

- High Performance Functional Simulator.
- Architecture: **Harvard Architecture**.
- Data Path: **Single Cycle Data path**.
- Memories:
    - **Instruction Memory** Size 2Kbits (Standard available in the market)
    - **Data Memory** Size 2Kbits (Standard available in the market)
- Registers
    - **32 16 bit GPR's R0-R31**
    - R0 contains zero. Cannot be used for any other operations.
- **16 bit** Arithmetic Logic Unit
    - Adder
    - Fast Multiplier
- Provision for multiplier to be clocked at a faster clock.
- Addressing mode: Displacement type addressing modes.

Architecture/Block Diagram:

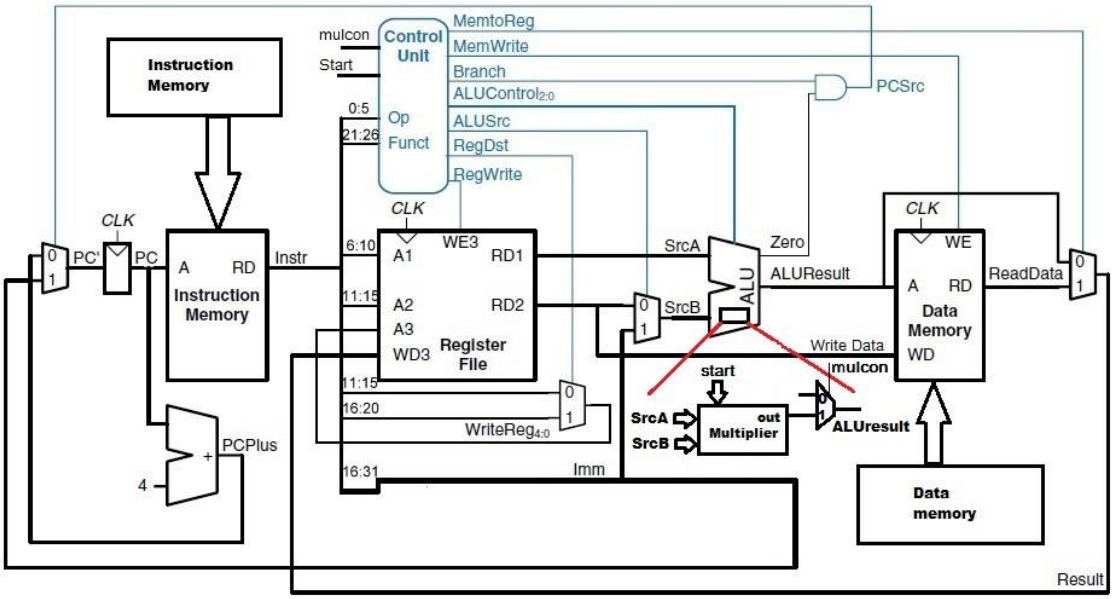


Fig 4: Single Cycle Data path block diagram

This Architecture is based on the **Single Cycle Datapath MIPS Processor Architecture**. It consists of an Instruction Memory and a Data Memory. The instruction memory will contain set of opcodes in hex format copied from an external memory. The data memory will contain keys and data that is to be operated on. The instruction memory can store as much as 2kb of data. The data memory is also 2 kb in size.

The architecture consists of the following components:

- Instruction Memory

- Program Counter
- Decode Stage
- Control unit
- Register Files
- Arithmetic Logic Unit
- Data Memory
- Mux
- Adder
- Fast Multiplier

The instruction opcodes are fetched one by one depending on the value of the program counter. They are decoded in the **decode stage** and depending on the opcode the control unit generates necessary signals.
There are total 32 general purpose registers available but only 31 are available for user as the register R0 is hard coded to have a value zero in it. The **register R0** can directly be used as an operand having value zero.
The operand from these registers are passed on to the ALU where operations are performed on the operands and result is pushed out. One of the operands of the ALU can be an immediate value which will be decided by a mux at the input of the second operand. There is a fast multiplier inside the ALU for performing multiplication at a faster rate than normal repetitive addition multipliers. The output of the ALU is given to the data memory in case of store instructions or bypassed and directly written back into the register files. The architecture has a specific instruction set with its respective opcodes. Depending on these opcodes the decoding of the instructions will take place and corresponding control signals will be generated. The following table gives details about the instruction set, its respective opcode and functions with a detail description of what control signals will generated for what opcode.

There are specific **instruction formats** that are used in this ISA. These formats specify how the processor decodes the instructions. There are three formats available which the user can use to write his assembly code instructions.

Register Type Instruction set format:

6 bits	5 bits	5 bits	5 bits	6 bits	5 bits
Opcode	Operand 1(rs)	Operand 2(rt)	Dest(rd)	Function	Reserved

Immediate Type Instruction set format:

6 bits	5 bits	5 bits	16 bits
Opcode	Operand (rs)	Dest(rt)	Address/Immediate

Jump Type Instruction set format:

6 bits	10 bits	16 bits
Opcode	Reserved	Target address

Opcode: Basic Operation of the Instruction
Operand 2: Second Source operand register (Destination for Immediate type instructions)
Function: for specific opcode select from variety of functions
Address/immediate: Address for load store instructions or immediate (constant value) for immediate instructions

Instruction list with Opcodes and respective control unit signals:

Instruction	Opcode	Funct	RegW	RegDst	ALUSrc	Branch	MemW	MemRead	ALUCon
HALT	000000	000000	0	0	0	0	0	0	0
R-TYPE									
ADD	000000	000000	1	1	0	0	0	0	0
SUB	000000	000001	1	1	0	0	0	0	1
MUL	000000	000010	1	1	0	0	0	0	2
OR	000000	000011	1	1	0	0	0	0	3
AND	000000	000100	1	1	0	0	0	0	4
XOR	000000	000101	1	1	0	0	0	0	5
LOAD/STORE									
Li	000001	000000	1	0	1	0	0	1	0
LW	000010	000000	1	0	1	0	0	1	0
SW	000011	000000	0	0	1	0	1	0	0
BRANCH									
BZ	000100	000000	0	0	0	1	0	0	12
BEQ	000101	000000	0	0	0	1	0	0	1
BP(BGT)	000110	000000	0	0	0	1	0	0	6
BN(BLT)	000111	000000	0	0	0	1	0	0	7
JUMP	001000	000000	0	0	0	1	0	0	8
ADDI	001001	000000	1	0	1	0	0	0	9
R-type New									
MULMOD	001010	000000	1	0	1	0	0	0	10
ADDMOD	001011	000000	1	0	1	0	0	0	11

Table 2

This table gives clear idea about the instructions set which can used on this processor. From the table we can see that the arithmetic instructions, all have the same opcode but with different **function fields**. There are two instructions which operate directly on **immediate values**, addi, li. Four branch instructions are available with one direct jump instruction. Branch and jump set the ALU result to zero making the zero flag high. Two special instructions are provided by this ISA, **mulmod** and **addmod**. These are register type instructions and will trigger a **modulo multiplier** placed inside the fast multiplier unit to generate 2^16+1 modulo operations for mulmod and the addmod will perform 2^16 modulo operation. A HALT instruction is provided which the design detects as end of the code and terminates the simulation. An examples is provided to demonstrate the use of these instructions with their formats.

Sample code:

```
        ADDI    R4,R0,1 //moving 1 to register R4
L1:     ADD     R5,R4,R7        //R5=R4+R7
        ADDI    R6,R0,8 //moving 8 to R6
        BEQ     R6,R5,L1        //comparing R6 contents with R5 contents if equal PC[L1]
        HALT                    //Terminate the code

        BLT     R6,R8,L1        //If R6 is less than R8 then PC=[L1]
        Format for mulmod and addmod is also similar: MULMOD R1,R2,R3/ADDMOD R4,R5,R14
```

Data Path:

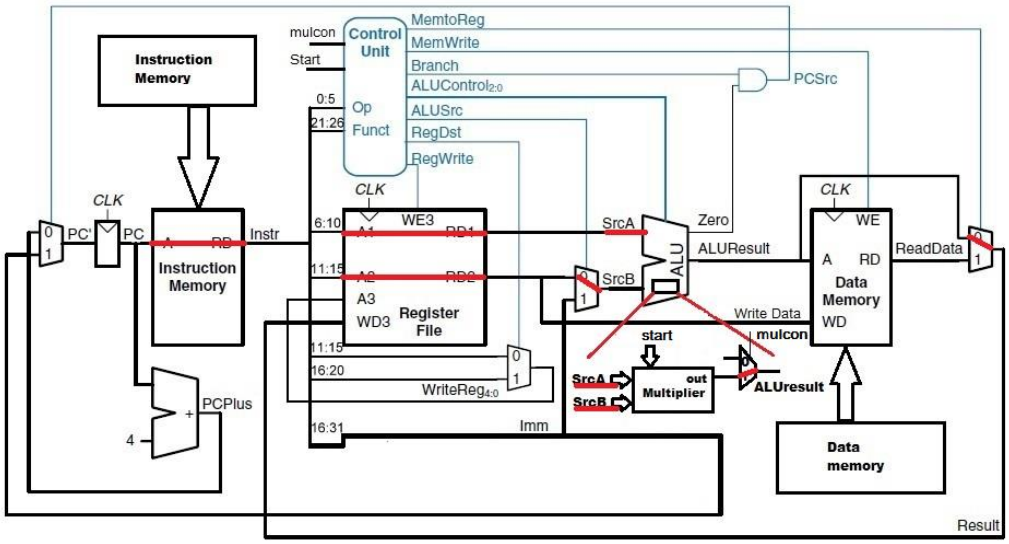


Fig 5: mulmod data path

The diagram above demonstrates single cycle data path for modulo multiplication instruction (**mulmod**).
The opcode for this will be taken from the location pointed by the **program counter** in the instruction memory. This opcode will be **decoded** and respective registers will be selected depending on the opcode. The **mux** will pass the value from the register files as this is **R-type command**. Fast multiplication will be performed inside the ALU block in the **multiplier chip** and output will be given out by selecting the mux which will pass the **multiplier result**. This output will bypass the **data memory** and will be **written back** to the register files. The main issue in this is the multiplication part. This design includes a fast multiplier which can perform both **multiplication** and **modulo multiplication** operations. The design for multiplier used here was proposed in 2007 by **Shamim Akhter**. The ancient **Vedic mathematics** is used to calculate the result of a **16 bit multiplication**. Using this algorithm and 8 bit multiplier a method is designed which will perform the multiplication at a very fast and efficient rate. Look up tables with the combination of Vedic mathematics speeds up the multiplication. The technique is proposed in the [4][10]"*VHDL IMPLEMENTATION OF FAST NXN MULTIPLIER BASED ON VEDIC MATHEMATIC*" paper which was published in *2007 IEEE*. This algorithm was simulated and tested on XC2S100-5tq144 family FPGA. The conclusion of the paper is that using the given conditions the time required for a 16 bit multiplier is **2 ns**. The diagram below shows the internals of the ALU block. There are **dedicated signals** for the multiplier block. The ALU control signal will be given both to the adder as well as the multiplier. Whenever the control unit comes across a mul or mulmod opcode, it generates a "**mulcon**" and a "**start signal**". Whenever the multiplier block receives a "start" and an ALU control signal, it shows that it is a multiplication or a mulmod operation. The "Start" triggers the multiplier. Whenever it is a multiplication operation "mulcon" is high which will select which output to pass from the two available outputs. Addmod is performed in the ALU itself.

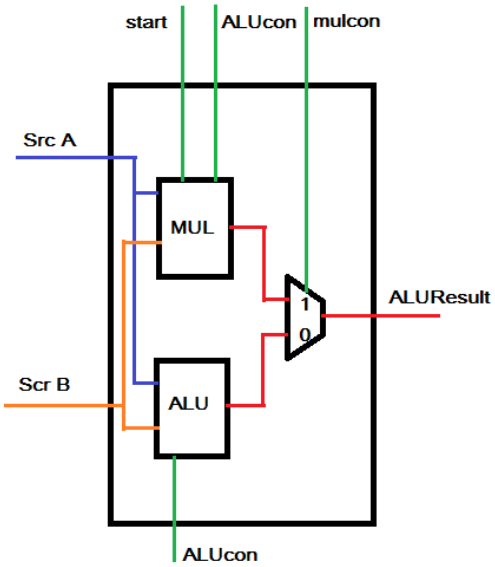


Fig 6: internal block diagram of ALU

Modulo multiplication operation is performed using a modification of a special algorithm by knuth's [8]. The algorithm is as follows:

1) Multiply the two Operands.
2) Subtract Higher 16 bits from lower 16 bits.
3) If the difference is positive, Answer is the subtraction value.
4) If the difference is negative, Answer is subtraction added with 65537.

Delay Path (Critical Path) calculations:

Table 7.6 Delays of circuit elements

Element	Parameter	Delay (ps)
register clk-to-Q	t_{pcq}	30
register setup	t_{setup}	20
multiplexer	t_{mux}	25
ALU	t_{ALU}	200
memory read	t_{mem}	250
register file read	t_{RFread}	150
register file setup	$t_{RFsetup}$	20

The following table is given in ***Digital Design and Computer Architecture by David Money Harris & Sarah L. Harris***. Using these delays as the basis of our calculations the critical path delay is found out. The Cycle time will be decided by the time taken by the **slowest instruction**. **Cycle time** is given by adding all the delays that each block in the architecture takes to perform its function.
tMUL is for multiplier. tMUL = 2ns.
Considering LW as critical instruction as initially given in the text, the Tc for LW will be:
Tc= t_pcq_PC + tmem + t_RFread + tmux + tALU + tmem + tmux + tRFsetup = 30 + 250 + 150 + 25 + 200 + 250 + 25 + 20 = 950 ps.
Now considering for **MUL operation**
Tc= t_pcq_PC + tmem + t_RFread + tmux + tMUL + tmux (inside ALU) + tmux + tRFsetup = 30 + 250 + 150 + 25 + 2 ns + 25 + 25 + 20 = 0.525 ns + 2 ns = 2.525 ns
Incase this is a modulo multiplication, according to the algorithm used for Modulo multiplication , **2 ALU delays** will get added to the MUL operation delay as one extra subtraction and addition is needed for Modulo Multiplication according to the algorithm used. Therefore, Tc= 2.525ns + 0.400 ns = **2.925 ns**
Therefore looking at the figures it is clear that **Modulo Multiplication** will incur **maximum cycle time** and is considered as the **critical path** for this design. To achieve this Tc we need a clock of 342Mhz.

Transistor

SRAM is used for data memory and instruction memory as well as for registers. Therefore, transistor count can be calculated as:

1) 2048b (Instruction Memory) + 2048b (Data Memory) + 512b (32 16 bits GPRs) = 4608 bits.
   Transistors required for 4068 bits with a 6T SRAM = 24408 T
2) ALU is considered to be built using Gate Diffusion Technique and the transistor comes to **938 T**

3) Multiplier has **421 slices** and uses look up tables as well, assuming multiplier need 500000 transistors. therefore the transistor count is T (The exact transistor count was not found out but looking at many other multipliers and considering the presence of look up tables the transistor count is assumed)
4) 5 **multiplexers** are used which sums up to **10T**
5) Assuming **temporary buffers** occupy 128 bits, therefore transistor count will be 128*12T(1 flip flop) = **1536T**

Therefore **total transistors required are** = 26892 + 500000 T = 526892 transistors.

Point 2 and 4 [3] are referred from "*32 – BIT ARITHMETIC AND LOGIC UNIT DESIGN WITH OPTIMIZED AREA AND LESS POWER CONSUMPTION BY USING GDI TECHNIQUE" by G. Sree Reddy and K. V. Koteswara Rao.* **Point 5** [2] is referred from "transistor count" *Wikipedia*.

Testing and Design:

The initial step in designing this architecture was building each and every component of the design separately and testing its **functionality at the unit level** itself. Then all these design blocks were combined together to form a working architecture. The **ISA** for this design was designed keeping in mind the application that we need to run on this architecture i.e. **the block cipher algorithms**. This is the basic reason why **modulo instructions** are included in this architecture. Depending on the algorithm used to convert the instructions into opcodes (according to the instruction format given before), the decoding logic was designed. During the testing each step was monitored like, is the decoding stage producing correct outputs, is the program counter pointing to the right location, generation of control signals, ALU working, Branching etc all was tested individually. Each instruction was pushed into the core and the core was checked for right functionality. Combination of instructions was pushed into the core to check for its outputs and behavior. After confirming all individual tests, IDEA algorithm for 64 bits was run successfully on this core.

Multi Core Implementation

According to the problem statement, IDEA algorithm is used to demonstrate the working of the Processor simulation. The constraint to this project is that the **input should be 1024 bits**. Running 1024 bits on a single cycle data path will incur a lot of latency and therefore a Multi Core approach has been implemented. The input data stored in the memory image or main memory will be divided into 512 bits chunk of data other than the keys and provided to two different cores which run the same processor logic. Each core has its own dedicated memory space in the main memory from which it will take its data. The instructions provided to both the cores are common.

The specialty of this approach is that **processing of 1024 bits** using this approach will be executed in **half the time** in comparison with what single cycle data path will take. Each core will take same instructions and keys but the data going to both the cores will be different. The data memory divides inside the processor in such a way that keys and 512 bits of data will be given to one core and another data with same keys will be given to the other core. Due to Multicore approach number of clock cycles will decrease, the overall **throughput** and **bandwidth** will **increase**. 8 rounds of 64 bits each are performed on each core processing total of 1024 bits of data.

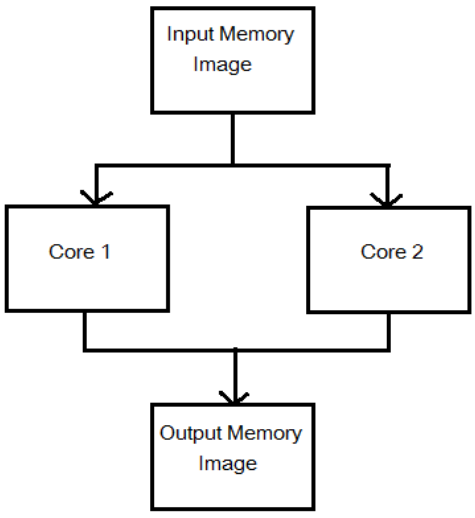


Fig 7: MultiCore Block Diagram

Result:

Number of lines in **assembly code**: 59
Number of clock cycles for 64 bits as in IDEA round: 304
Number of clock cycles for 1024 bits as in 16 IDEA round: 2411
Cycle Time for this architecture is 2.925ns
CPI for single cycle is ideally considered 1.
Total Execution time = # instructions x (1 cycle/instruction) x (clock time/cycle)

Total Execution time for 64 bits = 304 x 1 x 2.925ns = 889.2ns
Therefore **bandwidth is 72 Mbps.**

Total Execution time for 1024 bits = 2411 x 1 x 2.925ns = 7052.2ns
Therefore **bandwidth is 145 Mbps.**

Adding more cores to the architecture can increase the throughput more for 1024 bits.
The Performance cost ratio can be calculated 72/526892T.

Implementation:

The simulation of this processor is done in **Python 3.5**. Each component of the hardware is one definition in the program. Each core has its own components and **separate threads** are made for each core. Using the properties of multithreading both the cores are run in parallel to process 512 b of data per core and the out bits of each core is fused to form **1024 bits of encrypted data** which is stored in the data memory and pushed to a separate memory image. Evidence of parallel execution is provided in the trace files at the end. time.clock() is used to track time of each thread as well as the total time required for execution of the total program. All the time values come out to be approximately the same. Attached with this report: 1) Fully commented Python Code 2) Trace files 3) Instruction memory image, Data Image and output memory image. 4) Fully commented Assembly code written in the above mentioned ISA 5) Readme file to guide the user run the code.

Fig 8

Summary:

A Single cycle data path processor implementation for running block ciphers is implemented. Using special algorithms and special instructions specific to block ciphers, assembly code of IDEA algorithm is run with 1024 bits input and 128 bit key. Bandwidth of 72 Mbps was found out for 64bits implementation and 145 Mbps for 1024 bit input data. The diagram above gives the evidence of the output. For checking the output, data was considered from the homework example and repeated 16 times to obtain 1024 bits of input data.

Literature survey:

M.P. Leong, O.Y.H. Cheung, K.H. Tsoi and P.H.W. Leong. *A Bit-Serial Implementation of the International Data Encryption Algorithm IDEA*. N.T. Hong Kong: Department of Computer science and Engineering, © 2000 IEEE.[6]

M.P. Leong, O.Y.H. Cheung, K.H. Tsoi and P.H.W. Leong's paper presents a high performance implementation of IDEA algorithm. The authors believe that cryptography actually is an ideal application for field programmable custom computing machines (FCCMs) and the advantages of FCCMs over VLSI implementations are mentioned. The paper states that hardware implementation offer significant speed improvements over software implementation as it exploits parallelism among operators. The proposed algorithm is implemented on Xilinx Virtex XCV300-6 device and a throughput of 500Mb/sec is obtained. The paper states that a bit serial architecture is used which offers advantages like high degree of fine-grain parallelism, scalability, high clock rate and compact implementation. Bit serial architecture has an advantage over the bit-parallel architecture as it facilitate high system clock rates. It is clear from the paper that modulo operation is the critical path in the algorithm. After researching lot of potential Multiplication modulo techniques, the paper explains the architecture proposed by Meier and Zimmer with a modification. As this is IDEA specific implementation, one subtraction can be eliminated from the proposed modulo method. The implementation also uses a modified version of Lyon's Serial Parallel multiplier in their multiplication modulo method. Using this the required speed up in modulo operations is achieved. To conclude with, this paper mentions that this bit-serial implementation of IDEA can be efficiently scaled up to produce higher encryption rate.

Antti Hamalainen, Matti Tommiska, and Jorma Skytta. 6.78 Gigabits per Second Implementation of the IDEA Cryptographic Algorithm. Finland: Helsinki University of Technology, © 2002 Springer-Verlag. [7]

Antti Hamalainen, Matti Tommiska, and Jorma Skytta's paper presents 6.78 giga bits per second implementation of the IDEA algorithm. The author explains that this algorithm uses 16 bit sub blocks and can be fully pipelined. This pipelined implementation presented by this paper achieves a clock rate of 105.9 Mhz on Xilinx' XCV1000E-6BG560 FPGA of the Virtex device family. This particular implementation uses 18105 logic cells and this design achieves a throughput of 6.78 Gbps as the title states, with a latency of 132 cycles. The paper states that FPGAs are ideal components for algorithms which use fast cryptography. It is mentioned that by using million gate FPGAs, it is possible to have implementation of fully un-rolled cryptographic algorithms. The author mentions that modulo multiplication is the critical path and can be improved by using better and improved design methods. For modulo multiplication the author has used a *partial product generation* technique proposed by Ma for 16 bits. The inputs to this partial product generator logic are 16 bit integers that are represented in diminished-one format. The authors explain that to achieve a clock rate of 100Mhz the carry save adder structure used in the implementation is replaced by a three-stage adder tree which helps in reducing area as well as increase the clock rate. The paper concludes by projecting values in a table which shows performance of the proposed method on different devices with a comparison between them and that FPGA based cryptography module is a strong candidate when high performance cryptography is considered.

Comparison with other models proposed in the Papers Mentioned below:

Parameters	Paper 1	Paper 2	Proposed design
Throughput	500 Mbps	6.78 Gbps	72 Mbps
# transistors	2801 slices	11204 slices	421 slices
Latency		1.25 us	889.2 ns
Clock rate	125.202 Mhz	105.9 Mhz	342 Mhz
Core / Impementation	XCV300E-6 FPGA	XCV1000E-6 FPGA	Single Cycle Processor
Modulo technique used	Meir & Zimmer algorithm and modified version of Lyon's serial parallel multiplier	Partial Product generation proposed by Ma.	Multiplier using Vedic Multiplier and Knuth's algorithm.

Table 3

References:

[1]http://www.quadibloc.com/crypto/co040302.htm

[2]https://en.wikipedia.org/wiki/Transistor_count#FPGA

[3]http://www.ijrcar.com/Volume_3_Issue_4/v3i418.pdf

[4]http://ieeexplore.ieee.org/xpl/login.jsp?tp=&arnumber=4529635&url=http%3A%2F%2Fieeexplore.ieee.org%2Fstamp%2Fstamp.jsp%3Ftp%3D%26arnumber%3D4529635

[5] Digital Design and Computer Architecture by David Money Harris & Sarah L. Harris

[6] *A Bit-Serial Implementation of the International Data Encryption Algorithm IDEA* by M.P. Leong, O.Y.H. Cheung, K.H. Tsoi and P.H.W. Leong. N.T. Hong Kong: Department of Computer science and Engineering, © 2000 IEEE.

[7] 6.78 Gigabits per Second Implementation of the IDEA Cryptographic Algorithm. By Antti Hamalainen, Matti Tommiska, and Jorma Skytta. Finland: Helsinki, University of Technology, © 2002 Springer-Verlag.

[8] http://www.ciphersbyritter.com/NEWS2/93022501.HTM

[9] https://en.wikipedia.org/wiki/Multiplication_algorithm

[10] VHDL IMPLEMENTATION OF FAST NXN MULTIPLIER BASED ON VEDIC MATHEMATIC by Shamim Akhter

[11] https://www.wikipedia.org/

[12] Digital Design and Computer Architecture by David Money Harris & Sarah L. Harris (Single Cycle data Path block architecture)