Implementation of Hardware Accelerated SHA-256 Hashing Algorithm

EE2003: Computer Organization

J Antonson EE19B025

Arun Krishna AMS EE19B001

C S Hariharan EE19B016

Abstract:

A hash function takes an arbitrary-length message input to produce a fixed-length output. This project aims at implementing an hardware accelerator peripheral for SHA256 hashing algorithm with AXI4 interfacing with PicoRV32 CPU. The project focuses on multiple implementations of the accelerator with gradual improvements through spatial pre-computation techniques and pipelining. The SHA256 accelerators are implemented using Verilog and synthesized using Yosys Open Synthesis Suite. The optimized designs are then compared with a base-line C implementation in software.

SHA 256 Algorithm

The SHA family of hash function algorithms are used during data transmission to produce an indecipherable message digest. A hash function is a one-way deterministic function which is practically infeasible to invert a hash value to its message input. Therefore, it becomes an essential tool for embedded security in e-mail, internet banking, and other applications. The strong motivation behind this project is that these applications especially on the server-side require high-throughput low-latent encryption devices.

The general architecture of the SHA256 hashing algorithm consists of following modules:

- **Padding and Parsing** that ensures that the input message has length of multiples of 512 bits.
- **Message Expansion** that decomposes the input message into 16 blocks of 32 bits each and further expands it into 64 blocks {W}
- **Block Compression** that iterates 64 times on 8 - 32 bit variables {a,b,c,d,e,f,g,h} using {W} values obtained from previous stage and constants {K}
- **Hash Value Generation** that utilizes the 8 variables to generate a concatenated 256 bit output

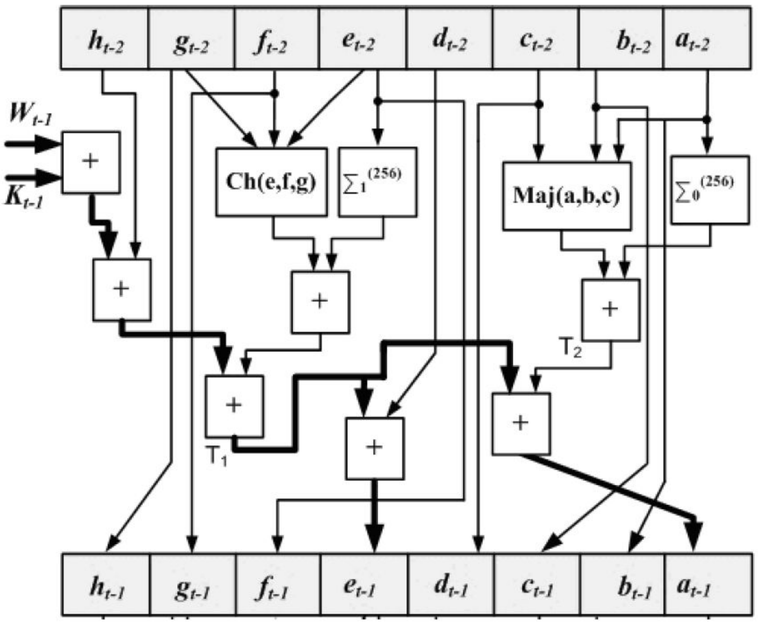

Figure 1: Block compression in Base implementation

Base Implementation - 4 - Stage Pipelined Architecture

A pipelined architecture consisting of four stages is implemented with each stage comprising of the module mentioned above. Such implementations are available in github ([Reference]) - This has been considered as our base architecture. It has been estimated by hand that the critical path for such an implementation is the calculation of {a,e} variables consisting of 4 adder- stages in the "Block Compression" stage of the pipeline (Figure 1). It takes 66 clock cycles for complete computation.

Two-Unrolling of Block Compression Module

An attempt at reducing the critical path time delay is made through spatial pre-computation techniques. Through computing a_{i+2} directly from a_i (The Base implementation computed a_{i+2} from a_{i+1} from a_i) it has been observed that certain parts of the algorithm can be parallelized. We estimate that the resulted architecture will have a critical path associated with 6 adders + a combinational block. This **results in an increase in critical path but reduces the number of clock cycles by almost half** (from 66 to 34).

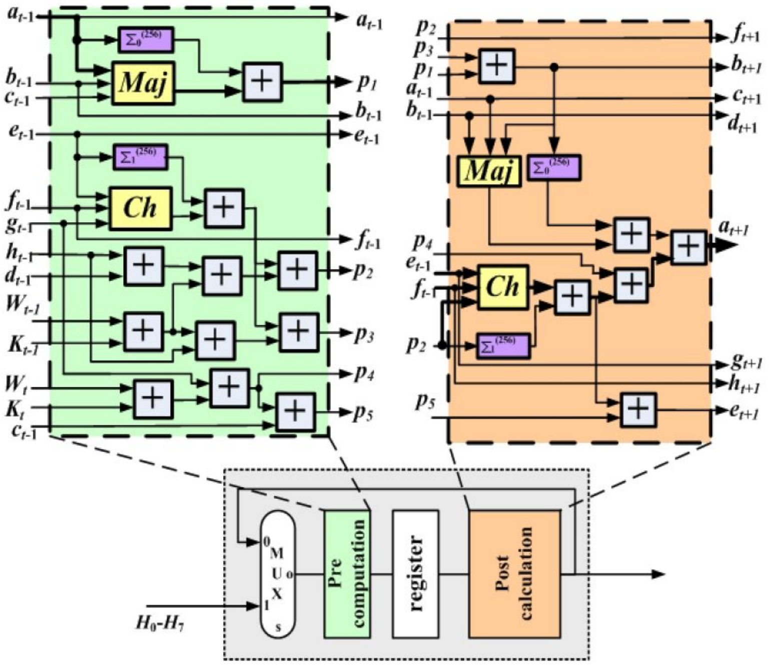

Figure 2: Block compression in Unrolled and Pipelined implementation

Two-Unrolling and 5-Stage Pipelined Architecture

The two-unrolled architecture of the block compression module is divided into two pipelined stages (Figure 2). Such an implementation will have a critical path consisting of a multiplexer + 3 adders + a combination block; But takes only 35 clock cycles to complete. Thus **this architecture is almost 90% faster than the generic base implementation**.

Observations:

We estimate that the highly single-operation & serialized nature of Padding and Parsing module will not provide much acceleration in hardware (Verilog code of the padding module was also designed but failed in compilation stage). Message Expansion, Block Compression and Hash Value generation Modules has been implemented.

The three architectures (Basic, Unrolled, Unrolled and Pipelined) mentioned above has been designed and synthesized in Verilog & Yosys and tested. Following results were observed

	Base Implementation	Two-Unrolling	Two-Unrolling and Pipelined
Clock Cycles (Verilog test bench)	66	34	35
Number of Logic Cells	7255	12606	13107
Clock Cycles (with AXI4 Interface)	561	531	531

After interfacing the peripheral with the pico-CPU using AXI4 interface, **we observed a difference of 30 clock cycles between Base Implementation and the other two architectures - as expected**. However the total number of cycles taken in all the implementations have been observed to be in similar order. This is because of **slow transfer of inputs & outputs from CPU to peripheral** and vice versa taking up significant number of cycles thereby masking any advantage the fastest implementation possess.

The two-unrolled and pipelined architecture is compared with a traditional software implementation of SHA256 using C. Following results were obtained:

```
Software Calculation Completed in 26657 cycles.
Instruction counter ..5275
CPI: 5.05

Hardware Calculation Completed in 531 cycles.
Instruction counter ..127
CPI: 4.18
```

We can observe that **the peripheral is about 50x faster compared to its software counterpart** (Based on number of cycles taken). Hence this could provide a significant boost for systems that require encryption at low latency and high throughput rates

We also observed that the PicoRV32 CPU takes huge number of clock cycles while executing "print" statements

References:

1. ULTRA HIGH SPEED SHA-256 HASHING CRYPTOGRAPHIC MODULE FOR IPSEC HARDWARE/SOFTWARE CODESIGN
2. GITHUB: Secworks
3. GITHUB: B-Con
4. WIKIPEDIA-SHA2

Work Split:

- **Arun Krishna A M S**
  - Design and Implementation of Basic (Improvements upon Antonson's code), Unrolled, Unrolled and Pipelined Architectures.
  - Interfacing PicoRV32 CPU with the peripheral.
  - C - Code implementation of SHA256 (Improvements upon CS Hariharan's code) and execution on PicoRV32 CPU.
  - Created Report Documentation.

- **Antonson J**
  - First Implementation of SHA256.
  - Design of Unrolled, Unrolled and Pipelined Architecture.
  - Implemented makefile generation, run.sh file, synthesis of verilog code.
  - Tried interfacing with MicroBlaze and Synthesis using Vivado MLx.
  - Created README file for details about execution of the code.
  - Maintained Gitlab repository with periodic updates.

- **CS Hariharan**
  - First Implementation of C-code and tried execution on PicoRV32 CPU.
  - Tried designing Padding and Parsing block.
  - Designed Message Expansion block.
  - Created Report Documentation.