Iterated addition

Oprițoiu Flavius
flavius.opritoiu@cs.upt.ro

November 22, 2024

Introduction

Objectives:

- Construct structures for multi-operand addition

Iterated algorithms' hardware implementation implies two distinct phases:

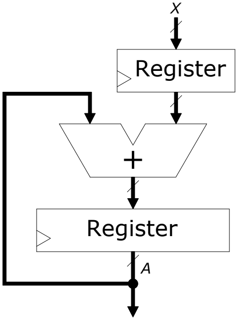
- constructing the state-related part, which stores the algorithm's state from the current to the next iteration
- implement the data processing part, which updates the algorithm's state from one iteration to the other

Sequential multi-operand addition

Each clock cycle a new operand is delivered on input register, X. The following algorithm calculates the result of a multi-operand addition and stores it in accumulator A:

$A \leftarrow 0$
loop
    $A \leftarrow A + X$
end loop

having the following hardware implementation:



Solved problem

Datapath of a cryptographic application

Exercise: Construct the datapath of an architecture for the 256-bit Secure Hash Algorithm 2 (SHA-2) algorithm (see [FIPS15], section 5.1.1).

Solution: The unit receives at input 512-bit blocks which it process sequentially in order to determine the hash associated with the received message. The hash result is delivered at the output as a 256-bit binary vector.

The block processing implies the foolowing operations:

- *Message schedule*: extends the 16 words of a received block to 64 words
- *Compression function*: takes one word from the *message schedule* and updates, in 64 iteration, variables *a*, *b*, *c*, *d*, *e*, *f*, *g* and *h*
- *Hash update*: adds to the current hash value, split in 8 words, the values of the *a* to *h* variables

Solved problem (contd.)

Message schedule

The 512-bit block is split in 16 32-bit words: M_0, M_1, ... , M_{15}, with , M_0 representing the most significant 32-bit of the block and, M_{15} representing the least significant 32-bits.

For 64 iterations, in each clock cycle, the *message schedule* constructs a new word in the least significant position (the first constructed word comes after M_{15}, and this first generated word will be followed by the next one etc.). In each iteration, the most significant word, M_0, is delivered at the output.

At any given moment only 16 words are required for construction of the next word. As a consequence, the new word will occupy the least significant position (M_{15}) all the other more significant words being shifted on the immediate, more significant, position (M_{15} will move to M_{14}, M_{14} to M_{13}, ..., M_1 to M_0).

Solved problem (contd.)

Message schedule

Message schedule is formally described by the algorithm bellow:

Input: Block BLK ▷ BLK is split into 16 32-bit words
Output: Word M_0 on 32-bit ▷ Delivers M_0 in each iteration

```
procedure MessageSchedule(BLK)
    M_0 ← BLK[511 : 480]                    ▷ Initialize the 16 words, M_i
    M_1 ← BLK[479 : 448]
    ...
    M_14 ← BLK[63 : 32]
    M_15 ← BLK[31 : 0]
    for i = 0 to 63 do      ▷ Construct a new word and update the 16 stored words
        NEW_WORD ← σ_1(M_14) + M_9 + σ_0(M_1) + M_0
        M_0 ← M_1
        M_1 ← M_2
        ...
        M_14 ← M_15
        M_15 ← NEW_WORD
    end for
end procedure
```

The addition operator, $+$, in this slide and the next to come is performed $\pmod{2^{32}}$.

Solved problem (contd.)

Message schedule

Functions $\sigma_0(\alpha)$ and $\sigma_1(\beta)$ are defined bellow:

$$\sigma_0(\alpha) = RtRotate(\alpha, 7) \oplus RtRotate(\alpha, 18) \oplus RtShift(\alpha, 3)$$
$$\sigma_1(\beta) = RtRotate(\beta, 17) \oplus RtRotate(\beta, 19) \oplus RtShift(\beta, 10)$$

where: $RtRotate(x, p)$ rotates word x to the right by p positions; $RtShift(x, p)$ shifts word x to the right by p bits (adding 0s to msb) and $\oplus$ denotes the EXCLUSIV-OR operator

For implementing these operators, one can use Verilog functions:

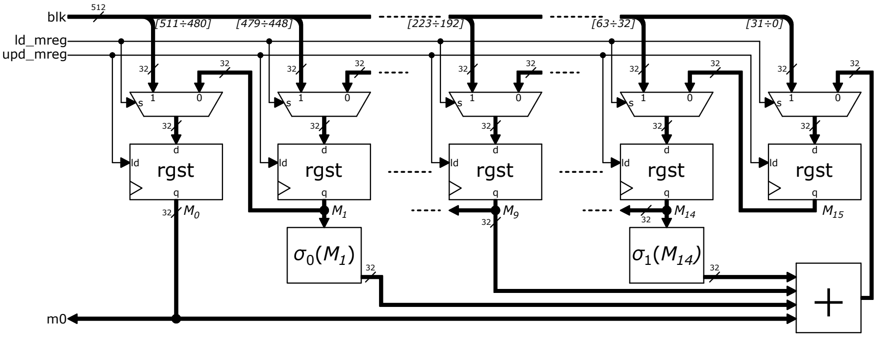
```
function [31:0] RtRotate (input [31:0] x, input [4:0] p);
  reg [63:0] tmp;
  begin
    tmp = {x, x} >> p;
    RtRotate = tmp[31:0];
  end
endfunction
```

This function is called by: `RtRotate(alpha,7)`

Solved problem (contd.)

Message schedule

The datapath component that implements the *message schedule* is depicted in the figure bellow:



Note: module *rgst* is available ▸ here

Solved problem (contd.)

Compression function and hash update

The hash result, on 256-bit, is formed of 8 32-bit words: H_0, H_1, H_2, H_3, H_4, H_5, H_6 and H_7, with H_0 being the most significant and H_7 the least significant.

Compression function uses 8 32-bit variables: a, b, c, d, e, f, g and h. The 8 variables are initialized to the values of words H_0, ..., H_7 of the current hash result. Afterwards, along 64 iterations, variables a to h are updated based on their current value, the value of word M_0 delivered by the *message schedule* and the value of a round constant, $K(i)$.

At the end of the 64 iterations, the hash result is *updated* by adding to each of the 8 words H_0 to H_7 the values of the variables a to h.

The compression function followed by the hash update is performed for each new received block.

Solved problem (contd.)

Compression function and hash update

Input: Message blocks are received

Output: Hash result words H_0, H_1, H_2, H_3, H_4, H_5, H_6, H_7

```
procedure SHA256
    InitializeHashResultWords()
    do
        a ← H0
        b ← H1
        ...
        h ← H7
        for i = 0 to 63 do
            T1 ← h + Σ1(e) + Ch(e, f, g) + K(i) + M0     ▷ M0 from expression
            T2 ← Σ0(a) + Maj(a, b, c)                     ▷ of T1 is delivered by the
            h ← g; g ← f; f ← e                           ▷ message scheduler which,
            e ← d + T1                                    ▷ since performing the same
            d ← c; c ← b; b ← a                           ▷ number of iterations (64), can
            a ← T1 + T2                                   ▷ operate in parallel with this loop
        end for
        H0 ← H0 + a
        H1 ← H1 + b
        ...
        H7 ← H7 + h
    while not last block
end procedure
```

Solved problem (contd.)

Compression function and hash update

The SHA-256 algorithm uses the following functions:

$$\begin{aligned}
\Sigma_0(x) &= RtRotate(x,2) \oplus RtRotate(x,13) \oplus RtRotate(x,22) \\
\Sigma_1(x) &= RtRotate(x,6) \oplus RtRotate(x,11) \oplus RtRotate(x,25) \\
Ch(x,y,z) &= (x \ \underline{and} \ y) \oplus ((\underline{not} \ x) \ \underline{and} \ z) \\
Maj(x,y,z) &= (x \ \underline{and} \ y) \oplus (x \ \underline{and} \ z) \oplus (y \ \underline{and} \ z)
\end{aligned}$$

The above *and* and *not* operators are bit-wise (operate on vectors, at the individual bit level). Constants $K(i)$, indexed by current iteration, i, are specified by the standard ([FIPS15], section 4.2.2):

i	$K(i)$
0	32'h428a2f98
1	32'h71374491
2	32'hb5c0fbcf
...	....
63	32'hc67178f2

Solved problem (contd.)

Compression function and hash update

The 8 words of the hash result, H_0, H_1, H_2, H_3, H_4, H_5, H_6, H_7, are initialized to values specified by the standard ([FIPS15], section 5.3.3):

Output: Initialize the hash result words H_0, H_1, H_2, H_3, H_4, H_5, H_6, H_7

procedure InitializeHashResultWords
- $H_0 \leftarrow 32'h6a09e667$
- $H_1 \leftarrow 32'hbb67ae85$
- $H_2 \leftarrow 32'h3c6ef372$
- $H_3 \leftarrow 32'ha54ff53a$
- $H_4 \leftarrow 32'h510e527f$
- $H_5 \leftarrow 32'h9b05688c$
- $H_6 \leftarrow 32'h1f83d9ab$
- $H_7 \leftarrow 32'h5be0cd19$

end procedure

References

[FIPS15] National Institute of Standards and Technology, "FIPS PUB 180-4: Secure Hash Standard," Gaithersburg, MD 20899-8900, USA, Tech. Rep., Aug. 2015. [Online]. Available: http://dx.doi.org/10.6028/NIST.FIPS.180-4 (Last accessed 06/04/2016).