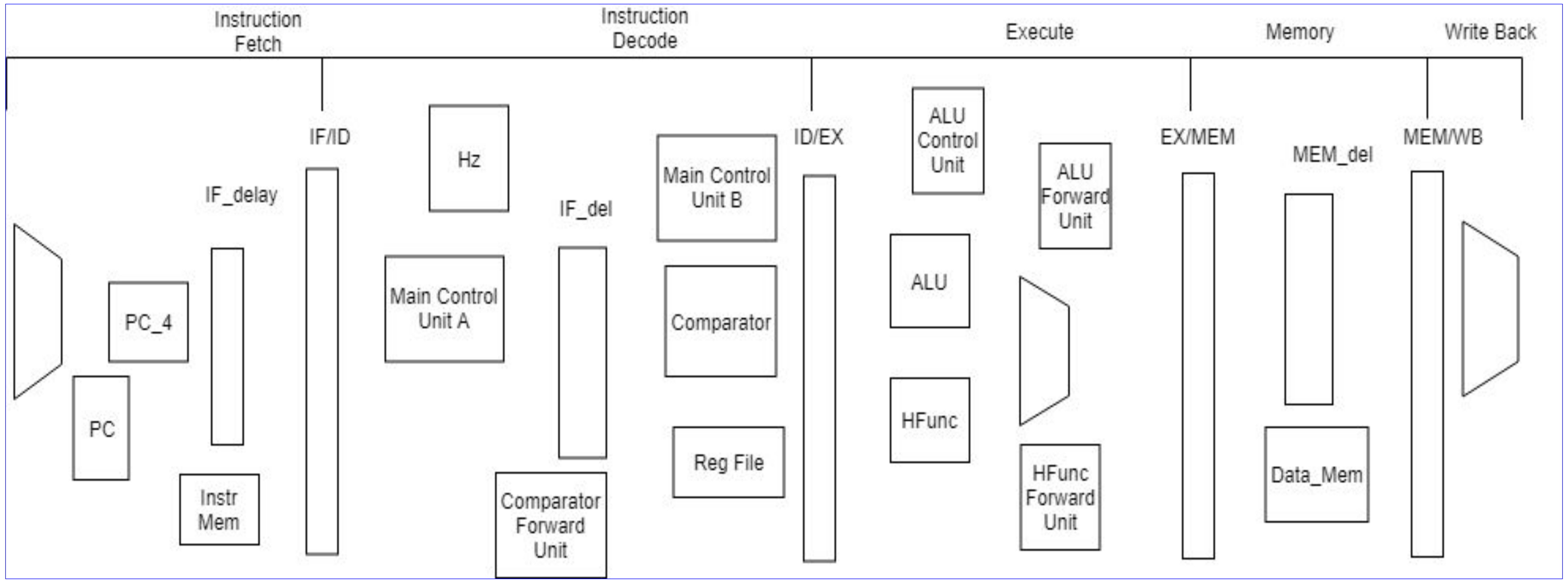
Custom MIPS-Based Processor for Calculating SHA-256

Pasha Beglar, Electrical and Computer Engineering
Adriel Dela Paz, Computer Engineering
Renrong Huang, Computer Engineering
Gordon Keil, Computer Engineering
Kristi Rounsefell, Computer Engineering
Mentor: Dr. Mohamed El-Hadedy Aly

Git-Hub page:

Abstract:
A hashing algorithm is a one-way encryption mathematical function that takes in a set of data and condenses it to an output of a fixed size called the message digest (*hash*). It has many applications in computing systems, from checking data integrity in digital communications to speeding up performances of databases. Our project involves creating a processor based on the MIPS Instruction Set that is specialized to perform theSHA-256 hashing algorithm. Custom hardware is incorporated in the processor for specific oper ations that are needed in calculating the hash. This simplifies the instruction calls for those specific operations, thereby condensing the required assembly code for implementing the SHA-256 hashing algorithm.



Sub-Stage Explanation:

Stage1: Instruction Fetch
In this stage, instructions are fetched from the instruction memory (Instr Mem). The Program Counter (PC) is incremented by 4 to make PC_4. PC and PC_4 are passed to delay registers (IF_delay). PC is also passed to Instr Mem.

Stage2: Extracting Instruction
In this stage, a 32 bit instruction is extracted from Instr Mem.

Stage3: Instruction Decoding
In this stage, data hazards that cannot be solved by forwarding is detected with the use of the Hazard Detection Unit (Hz). The data path is set up based on the instruction type provided by the opcode with the use of Main Control Unit A. Also, the register address is passed to the Register File.

Stage4: Register File Extraction
In this stage, the value of addressed registers is extracted from the Register File. The values in the pipeline registers (IF_delay and IF/ID) are either kept or discarded. The PC is updated if a branching instruction is being decoded. This is made possible with Main Control Unit B.

Stage5: Instruction Execution
In this stage, data is operated on using the Arithmetic Logic Unit (ALU) or HFunc based on the instruction.

Stage6: Memory Operations
In this stage, the control signals, address, and data are passed to Data Memory (Data_Mem). The pipeline values are passed to delay registers (Mem_delay)

Stage7: Memory Extraction
In this stage , a value from memory is extracted based on previously given address.

Stage8: Write Back
In this stage, the results of the previous stage are committed to a register in the Register File (Reg File) if necessary.

Implementation result of the custom processor:

FPGA	Nexys4 DDR2_xc7a100tcsg324-1

Resource	Estimation	Available
LUT	1051	63400
FF	72	19000
BRAM	547	126800
IO	0.5	135
BUFG	4	210
LUT	1	32

Implementation result from a similar device on the paper *A compact FPGA-based processor for the Secure Hash Algorithm SHA-256*.

Implementation results.

FPGA	xc5vlx50t-3ff1136
Frequency (MHz)	64.45
Slices	139
LUTs	527
Latency	280
Throughput (Mbps)	117.85
Efficiency (Mbps/Slice)	0.84

Custom Hardware Explanation:

Hardware separate from the standard MIPS instruction set architecture was added to perform operations specific to calculating SHA-256. New instructions that utilize this hardware instead of the standard MIPS ALU are labeled as H-type instructions (as oppose to R-type, I-type, and J-type instructions of standard MIPS ISA).

Hash Function Assembly Code:

```
lui  v0 0
ori  v0 v0 0x0000         #get value of h0_0
lw   h0 0 v0              #load value of h0_0
lui  v0 0
ori  v0 v0 0x0004         #get value of h0_1
lw   h1 0 v0              #load value of h0_1
lui  v0 0
ori  v0 v0 0x0008         #get value of h0_2
lw   h2 0 v0              #load value of h0_2
lui  v0 0
ori  v0 v0 0x000C         #get addr of h0_3
lw   h3 0 v0              #load value of h0_3
lui  v0 0
ori  v0 v0 0x0010         #get addr of h0_4
lw   h4 0 v0              #load value of h0_4
lui  v0 0
ori  v0 v0 0x0014         #get addr of h0_5
lw   h5 0 v0              #load value of h0_5
lui  v0 0
ori  v0 v0 0x0018         #get addr of h0_6
lw   h6 0 v0              #load value of h0_6
lui  v0 0
ori  v0 v0 0x001C         #get addr of h0_7
lw   h7 0 v0              #load value of h0_7
lui  v0 0
ori  v0 v0 0x0020         #get addr of nlim,
address right after h0_7
lw   nlim 0 v0            #load value of nlim
lui  v0 0
ori  v0 v0 0x0100
or   kb z v0              #initializing $kb;
addr of k[0]
lui  v0 0
ori  v0 v0 0x0200
or   wb z v0              #initializing $wb;
addr of w[0]
lui  v0 0
ori  v0 v0 0x300
or   mb z v0              #initializing $mb;
addr of m[0]
addi n z 0                #initialize $n ->
i=0
```

```
#BEGIN: prepare the message schedule W[t]
#PROCESS1:
lui  v0    0xFFFF
ori  v0 v0 0xFFFF        #initialize t1 --> t
= -1; adds 1 before first op
addi v3 z  0x000F        #v3 contains the
limit of the loop which is 15

#W1:                     #<increment, then
operate, then check>
addi v0 v0 0x0001        #increment t
sll  v1 v0 0x0002        #get t*4
add  v2 mb v1            #get address of M[t]
lw   v8 0  v2            #get value of M[t]
add  v2 wb v1            #get address of W[t]
sw   v8 0 v2             #save value of M[t]
to address of W[t]
bne  v0 v3 0xFFF9        #loop condition
checkback to W1

addi v0 v0 0x0001        #increment t; t=16
sll  v1 v0 0x0002        #get t*4
add  v4 wb v1            #get address of Wt
for instr below
addi v4 v4 0xFFF8        #get address of Wt-2;
sub 8 from v4
add  v5 wb v1            #get address of Wt
for instr below
addi v5 v5 0xFFE4        #get address of Wt-7
add  v6 wb v1            #get address of Wt
for instr below
addi v6 v6 0xFFC4        #get address of Wt-15
add  v7 wb v1            #get address of Wt
for instr below
addi v7 v7 0xFFC0        #get address of Wt-16

addi v3 z  0x0040        #reset loop limit to
64;
#W2:
lw    v1 0  v4           #get value of W[t-2]
```

```
hf_o1 v9 v1               #v9 = hf_o1(W[t-2])
lw    v1 0  v5            #get value of W[t-7]
add   v9 v9 v1            #v9 = hf_s1(W[t-2]))
+ W[t-7]
lw    v1 0  v6            #get value of W[t-15]
hf_o0 v2 v1               #hf_o0(W[t-15])
add   v9 v9 v2            #v9 = hf_o1(W[t-2]) +
W[t-7] + hf_o0(W[t-15])
lw    v1 0  v7            #get value of W[t-16]
add   v9 v9 v1            #v9 = hf_o1(W[t-2]) +
W[t-7] + hf_o0(W[t-15]) + W[t-16])

sll   v1 v0 0x0002        #calculating addr of
W[t]
add   v1 wb v1            #add t*4 to address
of W[0]
sw    v9 0  v1            #store value of W[t]
to addrs to W[t]

addi  v0 v0 0x0001        #increment t by 1
addi  v4 v4 0x0004        #increment address of
W[t-2]
addi  v5 v5 0x0004        #increment address of
W[t-7]
addi  v6 v6 0x0004        #increment address of
W[t-15]
addi  v7 v7 0x0004        #increment address of
W[t-16]
bne   v0 v3 0xFFEE        #if the value of t !=
64, go back to W2

#PROCESS2: copying the previous H value
or a0 h0 z                #a = t1 = h0
or a1 h1 z                #b = t2 = h1
or a2 h2 z                #c = t3 = h2
or a3 h3 z                #d = t4 = h3
or a4 h4 z                #e = t5 = h4
or a5 h5 z                #f = t6 = h5
or a6 h6 z                #g = t7 = h6
or a7 h7 z                #h = t8 = h7
```

```

and v0 z z                #initialize t = 0
addi v1 z 0x0040          #set loop limit to 64


#PROCESS 3:                <operate, then
increment, then check>
hf_s1  v2 a4               #v2 = hf_s1(e)
hf_ch  v3 a4 a5 a6         #v3 = hf_ch(e,f,g)
add    v4 a7 v2            #v4 = h + hf_s1(e)
add    v4 v4 v3            #v4 = h + hf_s1(e) +
hf_ch(e,f,g)
sll    v5 v0 0x0002        #get v5 = t*4
add    v6 kb v5            #get address of K[t]
lw     v6 0  v6            #get value of K[t]
add    v4 v4 v6            #v4 = h + s1(e) +
ch(e,g,f) + K[t]
add    v7 wb v5            #get address of W[t]
lw     v7 0  v7            #get value of W[t]
add    v4 v4 v7            #TEMP1; v4 =
h+s1(e)+ch(e,g,f)+K[t]+W[t]

hf_s0  v2 a0               #v2 = hf_s0(a)
hf_maj v3 a0 a1 a2         #v3 = hf_maj(a,b,c)
add    v5 v2 v3            #TEMP2; v5 = s0(a) +
maj(a,b,c)

or     a7 a6 z             #h = g
or     a6 a5 z             #g = f
or     a5 a4 z             #f = e
add    a4 a3 v4            #e = d + T1
or     a3 a2 z             #d = c
or     a2 a1 z             #c = b
or     a1 a0 z             #b = a
add    a0 v4 v5            #a = T1 + T2

addi   v0 v0 0x0001        #add 1 to t
bne    v0 v1 0xFFE8        #if t!= 64 go back
in loop Process3
```

```
#PROCESS4:
add h0 a0 h0              #h0 = a + H0^i-1
add h1 a1 h1              #h1 = b + H1^i-1
add h2 a2 h2              #h2 = c + H2^i-1
add h3 a3 h3              #h3 = d + H3^i-1
add h4 a4 h4              #h4 = e + H4^i-1
add h5 a5 h5              #h5 = f + H5^i-1
add h6 a6 h6              #h6 = g + H6^i-1
add h7 a7 h7              #h7 = h + H7^i-1

addi mb mb 0x0040
addi n n 1                #i = i+1
bne n nlim 0xFF87         #if i!= n, loop back
to BEGIN
                          #ENDPROCESS: showing
resulting message
hf_swcr h0 0
hf_swcr h1 1
hf_swcr h2 2
hf_swcr h3 3
hf_swcr h4 4
hf_swcr h5 5
hf_swcr h6 6
hf_swcr h7 7

nops
nops
nops
nops
nops
nops
nops
hf_done
```

References:

Safaa S. Omran and Laith F. Jumma, "Design Of SHA-1 & SHA-2 MIPS Processor Using FPGA," *A Mutual Conference on New Trends in Information & Communication Technology Applications*, 7-9 March 2017. Accessed on: 26 November 2018. [Online] Available: https://ieeexplore-ieee-org
SHA2_algorithm_NIST.FIPS.180-4
https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.180-4.pdf
Computer Organization and Design_ The Hardware_Software Interface 5th Edition - David A. Patterson, John L. Hennessy - With all appendices and advanced material 5(2013, Elsevier)
García, Rommel et al. "A compact FPGA-based processor for the Secure Hash Algorithm SHA-256." Computers & Electrical Engineering 40 (2014): 194-202.