Instruction Set Architecture (ISA) for MAHA

1 Overview

1.1 General Features

MAHA (Memory-Array centric Hardware Accelerator) is a fabric to utilize the abundant, high-speed cache memory already available on modern processors as a reconfigurable fabric for hardware acceleration of common algorithmic tasks (e.g. security and signal processing applications). The memory is partitioned into several MLBs (Memory Logic Blocks) each one acting as a small temporal-spatial computing element. Unlike many traditional hardware accelerators (e.g. FPGA), MBC (Memory Based Computing) is an instruction based framework that reutilizes hardware resources over multiple clock cycles. MLBs utilize a combination of table lookups (using the cache memory) and a custom datapath optimized for energy efficiency.

1.2 Instruction Overview

The MLB ISA uses a 32-bit instruction encoding and has a datapath granularity of 8-bits (most operations are 16-bits wide). Op codes are represented using the leading 4 bits of the instruction and an additional control field is utilized in specific operations for added flexibility. Each MLB supports 32 8-bit registers that are pair aligned. The upper 6 registers are connected to the local bus and will latch bus data if a read is performed while a data is available on the bus. If no bus transmission is occurring, the stored register value will be read. Additionally, there are 16 "scalar" single-bit registers for very fine grained operations. These registers are used only in scalar mode operations and cannot be accessed directly in vector operations. The ISA supports the following types of instructions:

- 16-bit unsigned addition and subtraction
- 32-bit and 16-bit logical shift and rotate operations
- 64-bit and 8-bit memory Load and Store
- Reconfigurable datapath for fused logical operations of up to 3x16-bit variables
- 2-way and 3-way select operations (only 2-way in initial release)
- 2-way and 3-way branch operations (only 2-way in initial release)
- Unconditional Jump
- 8/12-bit input – 8/16/32-bit output table lookup operations
- 5/8-bit input – 1-bit output table lookup operations (currently not supported)
- 3-bit input – 1-bit output table lookup operations with the table directly encoded in the instruction
- MOV instruction to transfer data between the register files and buses
- Support for select on-demand SIMD datapath operations

Additionally, a Processor Status Register (PSR) will be utilized to record metadata about the last operation processed used in determining branch and select conditions. This will include whether the last operation produced:

- PSR[3] = User Selected
- PSR[2] = Carry Out

- PSR[1] = Overflow
- PSR[0] = Zero

1.3 Microarchitecture

Each MLB will contain 16KB of byte-addressable memory divided into 8 blocks (4KB/block). Block 0 will be reserved for LUTs and additional instructions and will be optimized for read accesses since this block will only be written on time at configuration. An asymmetric memory design can be applied to this block to reduce the read access energy at the expense of write energy. The remaining 3 blocks will be utilized for data and will not be optimized for read or write. Each MLB will also support VLIW of 2 with parallel memory access and/or datapath operations in each clock cycle. Instructions will be primarily stored in a schedule table of 128 entries of 64-bits each (to support VLIW 2). SIMD operations utilize both entries in a schedule table row to perform 4 simultaneous operations. Figure 1 below illustrates a block diagram of a sample MLB implementation.

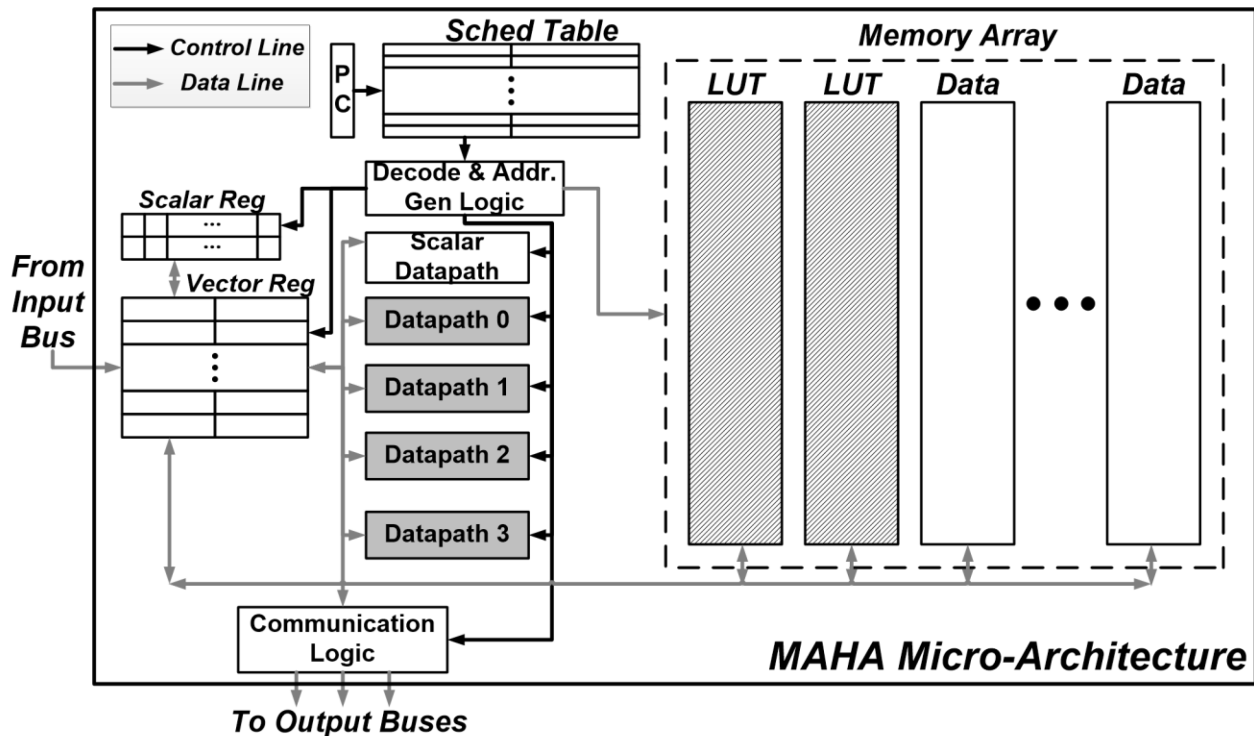

Figure 1: Block diagram of MLB

1.4 Bus Structure

In Multi-MLB systems, a sparse interconnect network is utilized to facilitate inter-MLB messaging. The MBC is divided hierarchically into *Clusters* and *Tiles*. Each Cluster contains 4 MLBs and has an intra-Cluster bus network enabling point to point communication between all MLBs. MLBs may access these buses in the same cycle as another operation through the use of virtual register ports. Reading from one of these buses will latch the value into one of the upper 6 registers as follows (index given is modulo 4):

- R26 – lower 8 bits from MLB_{i+1}

- R27 – upper 8 bits from MLB_{i+1}
- R28 – lower 8 bits from MLB_{i+2}
- R29 – upper 8 bits from MLB_{i+2}
- R30 – lower 8 bits from MLB_{i+3}
- R31 – upper 8 bits from MLB_{i+3}

Sets of 4 MLB Clusters are organized into Tiles with a similar interconnection scheme. The MLBs in each cluster share a single 16-bit bus that connects to the other three Clusters in a given tile. The inter-Tile communications take place over a mesh connection to allow for greater scalability of the platform. The overall connection scheme is shown in Figure 2.

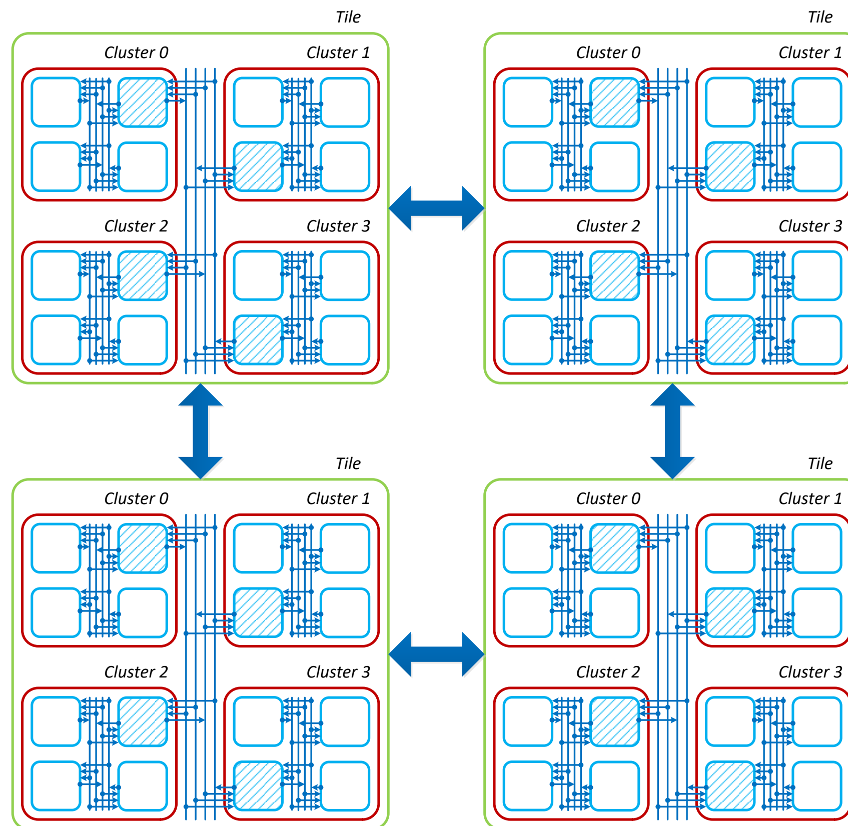

Figure 2: MAHA bus architecture

When writing to the buses, either an 8-bit or 16-bit value can be sent. If a 16-bit value is being transmitted, only 1 instruction may utilize the bus in a given cycle. If only 8 bits are being written, both instructions can write to the bus in the same cycle. This is accomplished by reserving the lowest 8 bits of the bus for "instruction 1" (the instruction contained in the lower 32 bits of the schedule table) and the upper 8 bits for instruction "2". Care must be taken when compiling an application that 8-bit reads are performed by the correct instruction on the receiving MLBs to capture the correct 8-bit values.

The following diagram illustrates the input/output structure of each MLB to the buses.

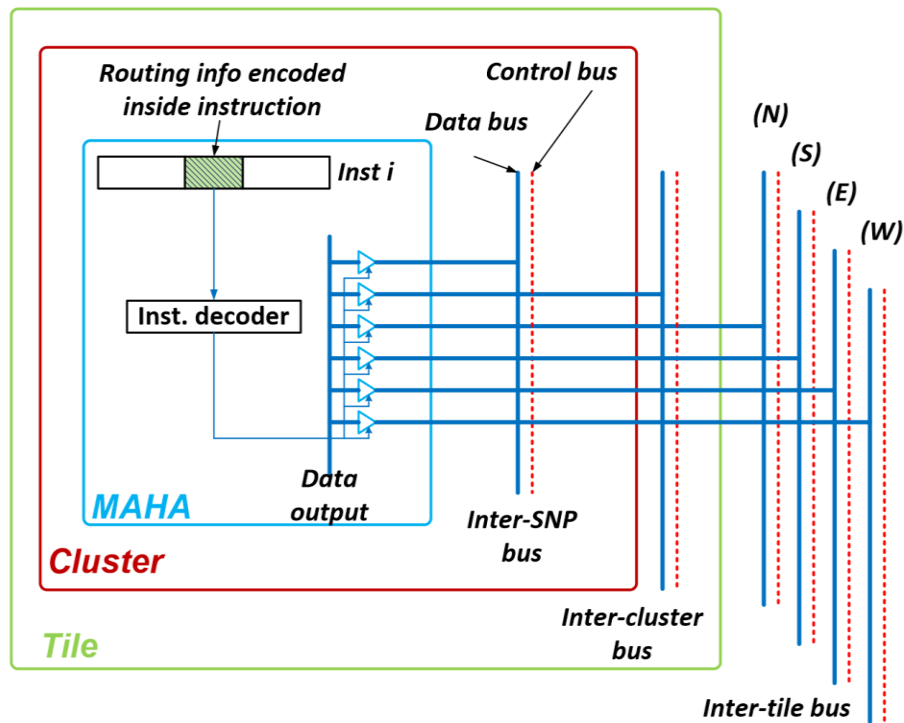

Figure 3: Bus I/O structure

A sample communication between distant MLBs is illustrated below in Figure 4.

Inter-Tile Communication

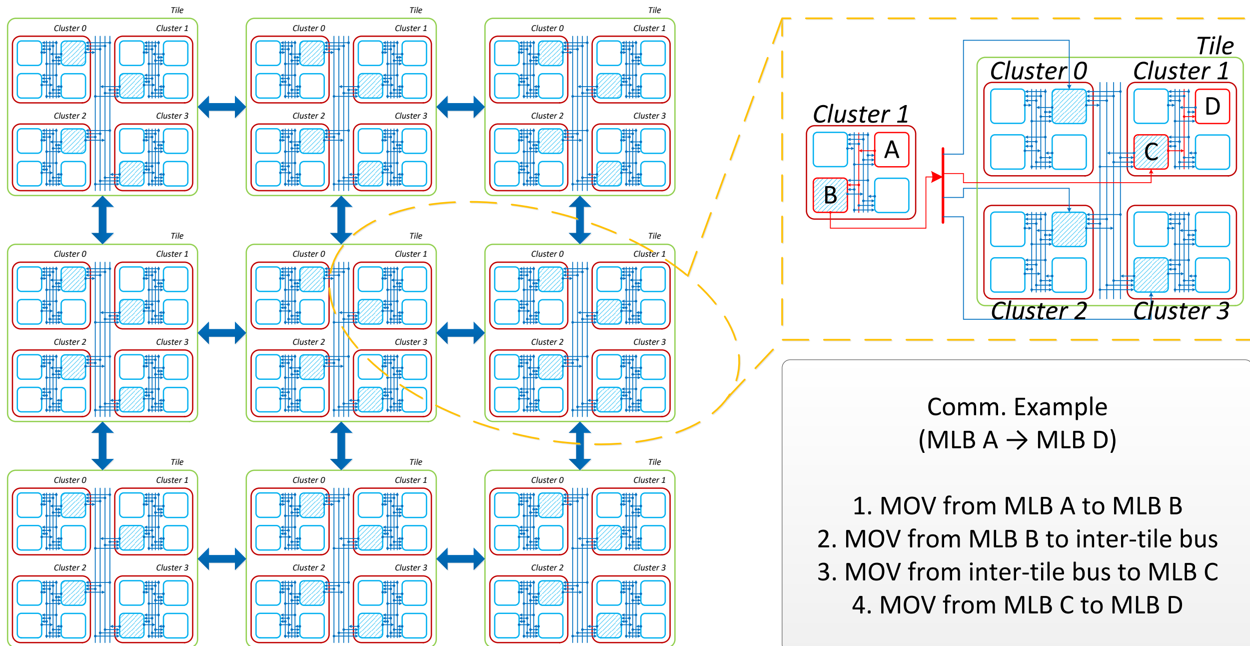

Figure 4: Sample Communication

2 Distinction from Existing Frameworks

MAHA has many differences from both traditional reconfigurable computing platforms and RISC processor architectures. Compared to a FPGA, some distinctions include:

- Follows a temporal-spatial computing model enabling both reuse of hardware resources over multiple clock cycles and partitioning of a large task over multiple units.
- Sparse interconnect framework that exploits locality of mapped tasks to both improve routing delays and overall energy efficiency
- Utilizes dense 2-D memory arrays very close to the computation engine to achieve both fast access times and to better handle data-intensive tasks

And compared to a RISC processor some distinctions include:

- Mimics common hardware structures such as select and fused logic as atomic operations
- Instructions are not typically fetched from memory, but are instead preloaded into a large schedule table within the MLB controller
- Hardware support for lookup table operations of varying sizes
- Message routing and memory allocation is all handled statically at compile time, eliminating the need for hardware to dynamically perform all these tasks
- Target applications are highly algorithmic and are therefore highly amenable to VLIW style architecture
- Support for manipulation of single bits

3 Instruction Encoding

3.1 Instruction Formats

3.1.1 Register Format (R)

<table>
<tr><td>31</td><td>30</td><td>29</td><td>28</td><td>27</td><td>26</td><td>25</td><td>24</td><td>23</td><td>22</td><td>21</td><td>20</td><td>19</td><td>18</td><td>17</td><td>16</td></tr>
<tr><td colspan="4">Op Code</td><td colspan="3">Func</td><td colspan="5">Rd</td><td colspan="4">Ra</td></tr>
<tr><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td>Ra</td><td colspan="5">Rb</td><td colspan="5">Rc</td><td colspan="5">Cond</td></tr>
</table>

3.1.2 Immediate Format (I)

<table>
<tr><td>31</td><td>30</td><td>29</td><td>28</td><td>27</td><td>26</td><td>25</td><td>24</td><td>23</td><td>22</td><td>21</td><td>20</td><td>19</td><td>18</td><td>17</td><td>16</td></tr>
<tr><td colspan="4">Op Code</td><td colspan="3">Func</td><td colspan="5">Rd</td><td colspan="4">Ra</td></tr>
<tr><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td>Ra</td><td colspan="15">Imm</td></tr>
</table>

3.1.3 Branch Format (B)

<table>
<tr><td>31</td><td>30</td><td>29</td><td>28</td><td>27</td><td>26</td><td>25</td><td>24</td><td>23</td><td>22</td><td>21</td><td>20</td><td>19</td><td>18</td><td>17</td><td>16</td></tr>
<tr><td colspan="4">Op Code</td><td colspan="3">Func</td><td colspan="9">Ta</td></tr>
<tr><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td>Ta</td><td colspan="10">Tb</td><td colspan="5">Cond</td></tr>
</table>

3.1.4 Scalar Format (S)

<table>
<tr><td>31</td><td>30</td><td>29</td><td>28</td><td>27</td><td>26</td><td>25</td><td>24</td><td>23</td><td>22</td><td>21</td><td>20</td><td>19</td><td>18</td><td>17</td><td>16</td></tr>
<tr><td colspan="4">Op Code</td><td colspan="3">Func</td><td colspan="4">Sd</td><td colspan="4">Sa</td><td>Sb</td></tr>
<tr><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td colspan="3">Sb</td><td colspan="4">Sc</td><td colspan="9">Imm</td></tr>
</table>

3.1.5 SIMD Format (Q)

<table>
<tr><td>63</td><td>62</td><td>61</td><td>60</td><td>59</td><td>58</td><td>57</td><td>56</td><td>55</td><td>54</td><td>53</td><td>52</td><td>51</td><td>50</td><td>49</td><td>48</td></tr>
<tr><td colspan="4">Op Code</td><td colspan="5">Rd1</td><td colspan="5">Ra1</td><td colspan="2">Rb1</td></tr>
<tr><td>47</td><td>46</td><td>45</td><td>44</td><td>43</td><td>42</td><td>41</td><td>40</td><td>39</td><td>38</td><td>37</td><td>36</td><td>35</td><td>34</td><td>33</td><td>32</td></tr>
<tr><td colspan="3">Rb1</td><td colspan="5">Rd2</td><td colspan="5">Ra2</td><td colspan="3">Rb2</td></tr>
<tr><td>31</td><td>30</td><td>29</td><td>28</td><td>27</td><td>26</td><td>25</td><td>24</td><td>23</td><td>22</td><td>21</td><td>20</td><td>19</td><td>18</td><td>17</td><td>16</td></tr>
<tr><td colspan="2">Rb2</td><td colspan="5">Rd3</td><td colspan="5">Ra3</td><td colspan="4">Rb3</td></tr>
<tr><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td>Rb3</td><td colspan="5">Rd4</td><td colspan="5">Ra4</td><td colspan="5">Rb4</td></tr>
</table>

3.2 Field Definitions

Op Code: first 4 bits of all instructions specify the operation and encoding of the rest of the instruction

Func: op code extension field used in select operations

Ra/Rc/Rd: byte-wise address in register file

Rb: byte-wise address in register file or amount of shift/rotate

Sa/Sb/Sc/Sd: bit-wise address in the scalar register file

Imm: 9-bit or 15-bit constant hardcoded into the instruction.

Cond: Sets the condition for select and branch operations according to the table below. To compare two numbers A and B, the branch/select operation will need to be preceded by a subtraction operation. Assuming that the subtraction performed is A-B, the branch/select will check the comparison listed below.

Cond Value[2:0]	000	001	010	011	100	101	110	111
Comparison	A == B	A <> B	A > B	A < B	A >= B	A <= B	N/A	N/A
Cond1	P[0]	!P[0]	P[2]	!P[2]	P[0] || P[2]	P[0] || !P[2]	P[3]	!P[3]
Cond2	0	0	0	0	0	0	0	0

To accomplish comparison operations, the branch and select instructions must be preceded with an arithmetic operation to achieve the comparison.

Ta/Tb: immediate values representing target destinations in memory

Op1/Op2/Op3: Set the logical operations in the reconfigurable datapath (see appendix A for more details on the functional decomposition)

3.3 Operation Notation

R_B[A] : The contents of the register at address A concatenated with the next *n* registers to produce data of width B

M_B[A]: The contents of memory at address A. If B is 8, the operation is a byte access and only the byte at address A is returned. If B is 64, A must be a word-aligned address (multiple of 4), and the entire 64-bit row will be returned

S_B[A]: If B=1, this represents the contents of the scalar register at bit-wise address A, if B=8, this represents the contents of the byte at byte-wise address A.

X[B:A]: selects bits A to B from variable X

B[X]: The contents of the inter-Cluster and inter-Tile buses.

>>: Shift Left

<<: Shift Right

>>^: Rotate Left

^<<: Rotate Right
PC: Program counter register
ST[A]: Entry at the ath position in the schedule table

3.4 Instruction Functionality

3.4.1 Wait (WAIT)

3.4.1.1 Field Usage

Format: I
Op Code: 0000
Func[2:1]: Mode select

11: end of execution
10: wait for event
01: delay (register value)
00: delay (immediate value)

Imm: Specifies the length of delay (func 00), or the event to wait on (func 10) per Table 1

3.4.1.2 Assembly Syntax

WAIT Func, Rd, Ra, Imm

3.4.1.3 Operation Performed

If (!waiting) begin
    waiting = 1
    if(func[2:1] == 2'b01)
        R_8[Rd] = R_8[Ra]
    else if(func[2:1] == 2'b00)
        R_8[Rd] = Imm[7:0]
end
case(Func[2:1])
    10: if (event) begin
        PC = PC + 1
        waiting = 0
    0X: begin
        If (R_8[Rd] == 8'b0)
            PC = PC + 1
            waiting = 0
        end else
            R_8[Rd] = R_8[Rd] - 1
    end
endcase

3.4.1.4 Notes

- WAIT operations do not affect the contents of the PSR at all
- Doing WAIT 000, 0, 0, 0 is equivalent to the NOP/NOOP instruction in most ISAs
- If 2 WAIT commands are issued at once, whichever resolves first will resume execution
- Use of the End of Execution function code (11) will cause the MLB to terminate operation until some external controller wakes it up again.
- See Appendix B for additional comments on use of dynamic scheduling in the MAHA fabric and for a complete listing of possible events

3.4.2 Add/Subtract (ADD)

3.4.2.1 Field Usage

Format: R

Op Code: 0001

Func[2]: If 1, carry in comes from carry out bit stored in PSR, otherwise, carry in is 0 for addition and 1 for subtraction

Func[1]: If 1 operation is subtract, if 0 operation is add

Func[0]: If 1, outputs to local bus as well as writing to register

Cond[2]: if 1 write carry out back to register file to prevent overflow

Cond[1]: if 1 Rb is 16 bits wide, if 0 is 8 bits wide

Cond[0]: if 1 Ra is 16 bits wide, if 0 is 8 bits wide

3.4.2.2 Assembly Syntax

ADD {Func, Cond[1:0]} Rd, Ra, Rb

3.4.2.3 Operation Performed

$R_{8/16/24}[Rd] = R_{8/16}[Ra] \pm R_{8/16}[Rb]$

PC = PC + 1

3.4.2.4 Notes

The width of Rd is selected as the max{size(Ra), size(Rb)} and is given an extra 8 bits if cond[2] is asserted to prevent overflow

3.4.3 Add/Subtract Immediate (ADDI)

3.4.3.1 Field Usage

Format: I

Op Code: 0010

Func[2]: If 1, carry in comes from carry out bit stored in PSR, otherwise, carry in is 0 for addition and 1 for subtraction

Func[1]: If 1 operation is subtract, if 0 operation is add

Func[0]: If 1, outputs to local bus as well as writing to register

3.4.3.2 Assembly Syntax

ADDI Func, Rd, Ra, Imm

3.4.3.3 Operation Performed

$R_{16}[Rd] = R_{16}[Ra] \pm Imm$

PC = PC + 1

3.4.4 Shift/Rotate (SHF)

3.4.4.1 Field Usage

Format: R

Op Code: 0011

Func[2]: 1 for shift/rotate of 32 bits, 0 for 16 bits

Func[1]: 1 for rotate, 0 for shift

Func[0]: 1 for shift/rotate right, 0 for shift/rotate left

3.4.4.2 Assembly Syntax

SHF Func, Rd, Ra, Rb

3.4.4.3 Function Performed

$R_{16/32}[Rd] = R_{16/32}[Ra]$ (>>/<</^>>/^<<) Rb

PC = PC + 1

3.4.5 Logical Operation (LOG)

3.4.5.1 Field Usage

Format: R

Op Code: 0100

Func[2:1]: 11: bitwise and
10: bitwise or
01: bitwise xor
00: bitwise not

Func[0]: If 1, outputs to local bus as well as writing to register

Cond[0]: 1 for 16-bit operation, 0 for 8-bit operation

3.4.5.2 Assembly Syntax

LOG {Func, Cond[0]}, Rd, Ra, Rb

3.4.5.3 Function Performed

case(Func[2:1])

11: $R_{8/16}$[Rd] = $R_{8/16}$[Ra] & $R_{8/16}$[Rb]

10: $R_{8/16}$[Rd] = $R_{8/16}$[Ra] | $R_{8/16}$[Rb]

01: $R_{8/16}$[Rd] = $R_{8/16}$[Ra] ^ $R_{8/16}$[Rb]

00: $R_{8/16}$[Rd] = ~$R_{8/16}$[Ra]

endcase

PC = PC + 1

3.4.6 Lookup Table (LUT)

3.4.6.1 Field Usage

Format: I

Op Code: 0101

Func[2:1]: 11: 32-bit output
10: 16-bit output
01: 8-bit output
00: not used

Func[0]: If 1, outputs to local bus as well as writing to register

Imm[14]: 1 for 12-bit input, 0 for 8-bit inputFunction Performed

3.4.6.2 Assembly Syntax

LUT {Func, Imm[14]}, Rd, Imm (Ra)

3.4.6.3 Function Performed

3.4.6.4

$R_{8/16/32}$[Rd] = M[$R_{8/12}$[Ra] + Imm[11:0]]

PC = PC + 1

3.4.7 Load/Store (LS)

3.4.7.1 Field Usage

Format: I

Op Code: 0110

Func[2]: 1 for write, 0 for read

Func[1]: 1 for 64-bit output, 0 for 8-bit output

Func[0]: Reserved for possible load instruction command

3.4.7.2 Assembly Syntax

LS Func, Rd, Imm (Ra)

3.4.7.3 Function Performed

If (func[2])

M[R_8[Ra] + Imm] = $R_{8/64}$[Rd]

else

$R_{8/64}$[Rd] = M[R_8[Ra] + Imm]

PC = PC + 1

3.4.8 Move (MOV)

3.4.8.1 Field Usage

Format: I

Op Code: 0111

Func[2]: If 1, operation is vector mode, if 1 operation is scalar mode

Vector Mode

Func[1]: If 1, operation is register-register, if 0, is register-bus

Func[0]: If 1, transfer is 16 bits, 0 is transfer of 8 bits

Scalar Mode

Func[1]: If 1, operations reads from scalar register file, 0 writes to scalar register file

Func[0]: If 1, other operand is the PSR, 0 other operand is vector register file

3.4.8.2 Assembly Syntax

MOV Func, Rd, Ra, Imm

3.4.8.3 Function Performed

```
If (func[2])
        If (func[1])
                R8/16[Rd] = R8/16[Ra]
        Else
                B[Imm[12:0]] ⇔ R8/16[Ra]        // as outlined in 3.4.8.4
else
        Case (func[1:0])
                00: S8[Rd] = R8[Ra]
                01: R8[Rd] = S8[Ra]
                10: S1[Rd] = PSR[Ra]
                11: PSR[3] = S1[Ra]
        endcase
PC = PC + 1
```

3.4.8.4 Notes

For 16-bit transfer, Ra and Rd must be pair-aligned

Bus addressing as follows:

0_0000_0000_0010:

0_0000_0000_0100: output to inter-Cluster bus

0_0000_0000_1001: read from inter-Cluster bus 1

0_0000_0001_0001: read from inter-Cluster bus 2

0_0000_0010_0001: read from inter-Cluster bus 3

0_0000_0100_0000: output to North inter-Tile bus (gMLB only)

0_0000_1000_0000: output to East inter-Tile bus (gMLB only)

0_0001_0000_0000: output to South inter-Tile bus (gMLB only)

0_0010_0000_0000: output to West inter-Tile bus (gMLB only)

0_0000_0100_0001: read from North inter-Tile bus (gMLB only)

0_0000_1000_0001: read from East inter-Tile bus (gMLB only)

0_0001_0000_0001: read from South inter-Tile bus (gMLB only)

0_0010_0000_0001: read from West inter-Tile bus (gMLB only)

3.4.9 Select (SEL)

3.4.9.1 Field Usage

Format: R
Op Code: 1000
Func[2]: 1 for 3-way select, 0 for 2-way (cond2 fixed to 0 and Rb ignored)
Func[1]: 1 if operands are 16 bit, 0 if 8 bit
Func[0]: 1 to also output selected value to the local bus

3.4.9.2 Assembly Syntax

SEL Func, Rd, Ra, Rb, Rc, Cond

3.4.9.3 Function Performed

if (cond1)
    $R_{8/16}$[Rd] = $R_{8/16}$[Ra]
else if (cond2)
    $R_{8/16}$[Rd] = $R_{8/16}$[Rb]
else
    $R_{8/16}$[Rd] = $R_{8/16}$[Rc]
PC = PC + 1

3.4.10 Branch (BR)

3.4.10.1 Field Usage

Format: B
Op Code: 1001
Func[2]: 1 for 3-way branch, 0 for 2-way (cond2 fixed to 0 and Tb ignored)
Func[1]: 1 for unconditional jump, 0 for branch
Func[0]: Unused

3.4.10.2 Assembly Syntax

BR Func, Ta, Tb, Cond

3.4.10.3 Function Performed

If (func[1]) then
    PC = {Ta, Tb}
Else
    If (cond1) then
        PC = Ta
    Else if (cond2) then
        PC = Tb
    Else
        PC = PC + 1

3.4.11 Fused Logic (FUSE)

3.4.11.1 Field Usage

Format: R

Op Code: 1010 (16-bit operation), 1011 (8-bit operation)

{Func, Cond}: These two fields concatenated control the operation performed on the three operands per a Reed-Muller expansion (as described in Appendix A)

3.4.11.2 Assembly Syntax

FUSE8 Rd, Ra, Rb, Rc, {Func, Cond}

FUSE16 Rd, Ra, Rb, Rc, {Func, Cond}

3.4.11.3 Function Performed

$R_{8/16}[Rd] = f(R_{8/16}[Ra], R_{8/16}[Rb], R_{8/16}[Rc])$

PC = PC + 1

3.4.12 Scalar Lookup (LUTS)

3.4.12.1 Field Usage

Format: S

Op Code: 1100

Imm: Encodes a 3-input, 1-output lookup table indexed into by the values in the three source registers

3.4.12.2 Assembly Syntax

LUTS Sd, Sa, Sb, Sc, Imm

3.4.12.3 Function Performed

S[Sd] = Imm[{S[Sa], S[Sb], S[Sc]}]
PC = PC + 1

3.4.13 SIMD Exclusive-Or (QXOR)

3.4.13.1 Field Usage

Format: Q
Op Code: 1101

3.4.13.2 Assembly Syntax

QXOR Rd1, Ra1, Rb1, Rd2, Ra2, Rb2, Rd3, Ra3, Rb3, Rd4, Ra4, Rb4

3.4.13.3 Function Performed

R_8[Rd1] = R_8[Ra1] ^ R_8[Rb1]
R_8[Rd2] = R_8[Ra2] ^ R_8[Rb2]
R_8[Rd3] = R_8[Ra3] ^ R_8[Rb3]
R_8[Rd4] = R_8[Ra4] ^ R_8[Rb4]
PC = PC + 1

3.4.14 SIMD Add (QADD)

3.4.14.1 Field Usage

Format: Q
Op Code: 1110

3.4.14.2 Assembly Syntax

QADD Rd1, Ra1, Rb1, Rd2, Ra2, Rb2, Rd3, Ra3, Rb3, Rd4, Ra4, Rb4

3.4.14.3 Function Performed

R_{16}[Rd1] = R_{16}[Ra1] + R_{16}[Rb1]
R_{16}[Rd2] = R_{16}[Ra2] + R_{16}[Rb2]
R_{16}[Rd3] = R_{16}[Ra3] + R_{16}[Rb3]
R_{16}[Rd4] = R_{16}[Ra4] + R_{16}[Rb4]
PC = PC + 1

3.4.15 SIMD Lookup Table (QLUT)

3.4.15.1 Field Usage

Format: Q
Op Code: 1111

3.4.15.2 Assembly Syntax

QLUT Rd1, Ra1, Rb1, Rd2, Ra2, Rb2, Rd3, Ra3, Rb3, Rd4, Ra4, Rb4

3.4.15.3 Function Performed

$R_8[Rd1] = M[R_8[Ra1] + (Rb1 << 8)]$
$R_8[Rd2] = M[R_8[Ra2] + (Rb2 << 8)]$
$R_8[Rd3] = M[R_8[Ra3] + (Rb3 << 8)]$
$R_8[Rd4] = M[R_8[Ra4] + (Rb4 << 8)]$
PC = PC + 1

3.4.16 Multiply (MULT)

3.4.16.1 Field Usage

Format: R
Op Code: TBD
Func[2]: If 1, operation is carried out over a Galois Field, if 0 is an arithmetic multiplication
Func[1]: If 1, operands are 16 bits each with at 32-bit output, if 0 operands are 8 bits with 16-bit
Func[0]: If 1, outputs to local bus as well as writing to register
Cond: Only used in GF mode, specifies the primitive reducing polynomial for the field from a table (shown below)

3.4.16.2 Assembly Syntax

TBD

3.4.16.3 Function Performed

$R_{8/16/32}[Rd] = R_{8/16}[Ra] * R_{8/16}[Rb]$
PC = PC + 1

Appendix A Reed-Muller Expansions

Boolean functions are typically denoted by a Sum-of-Products (SOP) canonical form. While simple to interpret, this form may not be ideal for logic synthesis because it requires inverted inputs. Another canonical representation for Boolean functions is known as the Reed-Muller (RM) expansion. The RM expansion uses the exclusive or of product terms to represent

an arbitrary Boolean function. Additionally, the RM representation of any function can be achieved without the use of inverted inputs.

To compute the RM representation of a function of n variables, $f(x_1, x_2, \ldots, x_n)$, one of the variables, x_i, is selected. The two cofactors of f with respect to x_i are then computed as follows:

$$f_{x_i}(x) = f(x_1, x_2, \ldots x_{i-1}, 1, x_{i+1,} \ldots, x_n)$$
$$f_{\overline{x_i}}(x) = f(x_1, x_2, \ldots x_{i-1}, 0, x_{i+1,} \ldots, x_n)$$

The original function f, can then be represented as $f = f_{\overline{x_i}} \oplus x_i \frac{\partial f}{\partial x_i}$ where $\frac{\partial f}{\partial x_i} = f_{\overline{x_i}} \oplus f_{x_i}$. This process can be repeated for every variable until the only operations remaining are Exclusive Or and And. Note that this form, unlike a SOP representation, does not require any inverted inputs, and can also perform output inversion.

Using this expansion, all functions of two inputs can be reduced to the following form: $f_{00} \oplus f_{01}x_1 \oplus f_{10}x_2 \oplus f_{11}x_1x_2$, where $f_{00}, f_{01}, f_{10}, f_{11}$are coefficients resulting from the expansion. The Op fields contain these coefficients in as follows $\{f_{11}, f_{10}, f_{01}, f_{00}\}$. A comparison of the hardware required to implement a RM canonical form compared to a SOP canonical form for arbitrary functions of 2 inputs is given below.

Gate Type	SOP Required	RM Required
2-input XOR	0	3
2-input AND	0	2
3-input AND	4	1
2-input OR	3	0
Inverter	2	0
Transistor Count	48	38

As an example, consider the function $f = \overline{AB}$. First, $f$is evaluated with respect to A resulting in:

$$f_{\bar{A}} = \bar{B}$$
$$f_A = 0$$

Substituting into the expansion, it can be shown that:

$$f = \bar{B} \oplus A(0 \oplus \bar{B}) = \bar{B} \oplus A\bar{B}$$

Since f_A evaluates to a constant, so it can be ignored from here on out. $f_{\bar{A}}$ still needs to be evaluated in terms of B.

$$f_{\bar{A}\bar{B}} = 1$$
$$f_{\bar{A}B} = 0$$

Plugging in to expression for f, $f = (1 \oplus B(1 \oplus 0)) \oplus A(1 \oplus B(1 \oplus 0))$. Reducing the expression:

$$f = 1 \oplus B \oplus A \oplus AB$$

For this example then, the original function can be coded as $\{f_{11}, f_{10}, f_{01}, f_{00}\}$ = {1, 1, 1, 1}

Some Common Reed-Muller Expansions are as follows for a three input system:

Operation	**$\{f_{ABC}, f_{BC}, f_{AC}, f_{AB}, f_C, f_B, f_A, f_0\}$**
A & B	0001_0000 (0x10)
A | B	0001_0110 (0x16)
A ⊕ B	0000_0110 (0x06)
~ A	0000_0011 (0x03)
A & B & C	1000_0000 (0x80)
A | B | C	1111_1110 (0xFE)
A ⊕ B ⊕ C	0000_1110 (0x0E)

Appendix B Dynamic Scheduling in MAHA

There exist some applications (give examples) where it may be desirable to pause execution for an extended (and potentially unknown) period due to a non-deterministic workload. This may not be applicable for all domains of applications, but it is considered here in this example implementation of MAHA. This is not nearly sophisticated as the dynamic scheduling mechanisms seen in modern superscalar, out-of-order CPUs. The objective of MAHA dynamic scheduling is to allow two or more MLBs to resynchronize with each other after some form of conditional operation removes them from a standard lock-step mode of operation. Some examples where this can be used:

1. MLB A takes a conditional branch which shortens an execution loop relative to MLB B, and MLB A and B need to communicate later in the program
2. MLB A is part of a processing pipeline which includes MLB B, but MLB A's pipeline stage requires fewer steps than that of MLB B
3. MLB A acts as a "helper" MLB to perform a specific complex operation shared by some number of other MLBs, and only needs to execute when this particular task is required

To achieve this functionality, MAHA supports a WAIT instruction as outlined in Section 3.4.1. When a MLB executes a wait instruction, it enters a WAIT state indicated by setting a dedicated waiting memory element. While this bit is set, the program counter will not increment, effectively stalling execution until the wait condition is satisfied. Whenever the condition is resolved, the waiting bit is unset and the program counter is allowed to continue incrementing.

There are two modes of operation in a WAIT state: delay and wait for event. In delay mode, an instruction specified register is loaded with a count of the number of cycles to pause execution for (can be either a value from another register or an immediate operand). Each cycle, this value is decremented by 1. When the counter hits 0, the waiting bit is cleared. This can be useful in situations where there is a known offset between the execution of two (or more) MLBs. This can also be used to perform a "no operation" operation.

In wait for event mode, execution stalls for an arbitrary amount of time until the specified condition is satisfied. This is most applicable in the type of situation where one MLB may branch causing other MLBs (which may need to communicate) to be unaware exactly when a message is expected. A complete listing of available events is shown below:

Code	Event	Code	Event
0x0000	New data loaded to local memory	0x0040	Data on cluster bus 3
0x0001	Data on local bus 1	0x0080	reserved
0x0002	Data on local bus 2	0x0100	Data on N tile bus (gMLB only)
0x0004	Data on local bus 3	0x0200	Data on E tile bus (gMLB only)
0x0008	reserved	0x0400	Data on S tile bus (gMLB only)
0x0010	Data on cluster bus 1	0x0800	Data on W tile bus (gMLB only)
0x0020	Data on cluster bus 2	0x10YY	Data matching YY on any local bus

Table 1: Listing of possible events for the WAIT instruction

Appendix C Future Improvements Under Consideration

- Support for a binary encoded case statements
- Support for NxN mux style select operation
- Support for 5x1 and 8x1 lookup operations from memory
- Support for mixed vector and scalar operations
- Reading from L2 bus as virtual register ports
- Investigate support of moving single bits around in scalar register file
- Investigate need of higher inter-cluster bandwidth
- Add support for dynamic instruction scheduling
- Add support for dynamic reuse of lut memory for data and vice-versa

Appendix D Sample Programs

D.1 C499 Verilog Benchmark

D.1.1 Introduction

This implements a single error detection-single error correction circuit. This implementation considers a single MLB assuming memory is initialized as specified and all registers are 0. The program executes as follows:

1. Load values of ID (R0-R3), IC (R4), and R (R5) from memory to registers
2. Compute partial sums of the syndromes using lookup tables for each byte of ID
3. Use the fused datapath to XOR the partial sums together and XOR with IC and R
4. Use lookup tables to compute the partial sums of OD based on the syndromes
5. Use the fused datapath to XOR the partial OD sums with the ID values to get the result
6. Store the resulting OD value to memory

The high level Verilog model that this implements can be found for free at:
http://web.eecs.umich.edu/~jhayes/iscas.restore/c499b.v

D.1.2 Instruction Overview

Cycle	Instruction 1	Instruction 2
1	LS 010, R0, 4096 (R25)	NOP
2	LUT 0100, R6, 0 (R0)	LUT 0100, R7, 256 (R1)
3	LUT 0100, R8, 512 (R2)	LUT 0100, R9, 768 (R3)
4	FUSE8 R10, R6, R7, R8, 0x0E	FUSE8 R11, R9, R4, R5, 0x82
5	FUSE8 R12, R10, R11, R25, 0x06	NOP
6	LUT 1000, R14, 1024 (R12)	LUT 1000, R16, 1536 (R12)
7	FUSE16 R18, R14, R0, R25, 0x06	FUSE16 R20, R16, R2, R25, 0x06
8	LS 011, R18, 4104 (R25)	NOP

D.1.3 Schedule Table Entries

Address	Instruction 1	Instruction 2
0	0x540E_9000	0x0000_0000
1	0x4060_0000	0x4070_8100
2	0x4081_0200	0x4091_8300
3	0xC0A3_1D0E	0xC8B4_90A2
4	0xC0C5_2FA6	0x0000_0000
5	0x44E6_0400	0x4506_0600
6	0xB127_03A6	0xB148_0BA6
7	0x57D9_1008	0x0000_0000

D.1.4 Initial Memory Contents

0x0000-0x00FF: Lookup table of partial sums of S for ID[31:24]

0x0100-0x01FF: Lookup table of partial sums of S for ID[23:16]

0x0200-0x02FF: Lookup table of partial sums of S for ID[15:8]

0x0300-0x03FF: Lookup table of partial sums of S for ID[7:0]

0x0400-0x05FF: Lookup table of partial sums of ID[31:16] for S

0x0600-0x07FF: Lookup table of partial sums of ID[15:0] for S

0x1000-0x1003: ID

0x1004-0x1004: IC

0x1005-0x1005: {8{R}}

D.1.5 SIMD Optimization

The above algorithm could be further optimized by making used of the SIMD style operations as follows to save 1 cycle.

Cycle	Instruction 1	Instruction 2
1	LS 010, R0, 4096 (R25)	NOP
2	QLUT R6, R0, 0, R7, R1, 1, R8, R2, 2, R9, R3, 3	
3	FUSE8 R10, R6, R7, R8, 0x0E	FUSE8 R11, R9, R4, R5, 0x82
4	FUSE8 R12, R10, R11, R25, 0x06	NOP
5	LUT 1000, R14, 1024 (R12)	LUT 1000, R16, 1536 (R12)
6	FUSE16 R18, R14, R0, R25, 0x06	FUSE16 R20, R16, R2, R25, 0x06
7	LS 011, R18, 4104 (R25)	NOP

D.2 AES (Advanced Encryption Standard)

D.2.1 Introduction

AES (or Rijndael as the algorithm was originally known) is commonly used encryption algorithm that works on 128-bit blocks of data (referred to as the “state”) and can have key sizes of 128-bit, 192-bit, and 256-bit. The state can be thought of as a 4x4 matrix of bytes. The algorithm follows the following procedure:

1. Key Expansion – compute the round keys from the cipher key using Rijndael’s key schedule
2. Initial Round
    a. Add Round Key – perform bitwise XOR of each state byte with the corresponding round key byte
3. Main Rounds – repeated 10 times for 128-bit keys, 12 times for 192-bit, and 14 times for 256 bit
    a. Substitute Bytes – replace each byte of the state using the Rijndael S-box
    b. Shift Rows – Rotate each row cyclically
    c. Mix Columns – multiply each column by a fixed matrix to produce a new, diffused column of the state
    d. Add Round Key
4. Final Round
    a. Substitute Bytes
    b. Shift Rows
    c. Add Round Key

Some good high level overviews of the algorithm can be found at the following two links:

http://en.wikipedia.org/wiki/Advanced_Encryption_Standard

http://www.cs.bc.edu/~straubin/cs381-05/blockciphers/rijndael_ingles2004.swf

It is worth noting that the algorithm is not carried out in a traditional arithmetic sense, but instead operates in a finite field (specifically $GF(2^8)$). The two primary differences between finite field arithmetic and standard arithmetic is that addition is performed as a bitwise XOR (no carry), and multiplication is performed modulo a reducing polynomial (Rijndael selects $x^8 + x^4 + x^3 + x + 1$ as the reducing polynomial).

To implement the algorithm in the MLB architecture, the following steps will be taken. We assume that the cipher key has already been expanded and the round keys are stored in memory as specified per section 0.

1. Load plaintext and the first round key into MLBs
    a. Each MLB will get one column of the state and the corresponding columns of the round keys. Since there are only 4 columns of the state, the entire algorithm will fit in 1 cluster, and thus local bus transfers are all that is required.
2. Perform bitwise XOR of initial round key and state

3. Resulting bytes from the XOR will be sent to the local bus and read in by the corresponding MLB to perform the shift rows step
4. Use a lookup table to perform the S-box transformation and multiplications by 2 and 3
5. Use SIMD XOR to perform the additions required by the matrix multiplication
6. Perform bitwise XOR of initial round key and state
7. Repeat steps 3-6 the number of times specified for the key size used
8. Repeat steps 3-4 and then 6 once more for the final round
9. Write cipher text back to memory

D.2.2 Instruction Overview

This example is written assuming a 128-bit key size, but can easily be extended to any key size by increasing the number of rounds and adding the corresponding round keys to memory

Cycle	Instruction 1	Instruction 2
1	LS 010, R0, 4096 (R20)	NOP
2	LOG 0100, R0, R0, R4	LOG 0110, R1, R1, R5
3	LOG 0110, R2, R1, R6	LUT 1100, R12, 0 (R0)
4	LUT 1100, R16, 0 (R0)	LOG 0110, R3, R3, R7
5	LUT 1100, R8, 0 (R0)	LUT 1100, R20, 0 (R0)
6	QXOR R0, R10, R13, R1, R11, R14, R2, R11, R15, R3, R9, R15	
7	QXOR R0, R0, R19, R1, R1, R17, R2, R2, R18, R3, R3, R19	
8	QXOR R0, R0, R23, R1, R1, R23, R2, R2, R21, R3, R3, R22	
9	LS 010, R8, 4104 (R20)	NOP
10	LOG 0100, R0, R0, R8	LOG 0110, R1, R1, R9
11	LOG 0110, R2, R2, R10	LUT 1100, R12, 0 (R0)
12	LUT 1100, R16, 0 (R0)	LOG 0110, R3, R3, R11
13	LUT 1100, R8, 0 (R0)	LUT 1100, R20, 0 (R0)
14	QXOR R0, R10, R13, R1, R11, R14, R2, R11, R15, R3, R9, R15	
15	QXOR R0, R0, R19, R1, R1, R17, R2, R2, R18, R3, R3, R19	
16	QXOR R0, R0, R23, R1, R1, R23, R2, R2, R21, R3, R3, R22	
Repeat 2-16 to perform additional rounds as required by the key size, incrementing the load address in (9) to get the appropriate key (4 more times for 128-bit).		
77	LOG 0100, R0, R0, R4	LOG 0110, R1, R1, R5
78	LOG 0110, R2, R1, R6	LUT 0100, R12, 0 (R0)
79	LUT 0100, R16, 0 (R0)	LOG 0110, R3, R3, R7
80	LUT 0100, R8, 0 (R0)	LUT 0100, R20, 0 (R0)
81	LS 010, R8, 4144 (R20)	NOP
82	LOG 0101, R0, R0, R8	LOG 0101, R2, R2, R10
83	LS 110, R20, 4152	NOP

Thus encrypting one block of data will take 83 clock cycles using this approach. 192-bit encryption would need an additional 15 cycles (98 in total) to perform 2 additional main rounds, and 256-bit encryption would need an additional 15 beyond that for a total of 113 cycles.

D.2.3 Initial Memory Contents

0x0000-0x03FF: S-box lookup table including 2x and 3x multiples

0x0100-0x02FF: S-box lookup table

0x1000-0x1003: Plaintext column for this MLB

0x1004-0x1007: Round Key 1

0x1008-0x100F: Round Keys 10 and 11

0x1010-0x1017: Round Keys 8 and 9

0x1018-0x101F: Round Keys 6 and 7

0x1020-0x1027: Round Keys 4 and 5

0x1028-0x102F: Round Keys 2 and 3

0x1030-0x1033: Round Key 12

D.2.4 No SIMD Version

If an implementation does not desire to use the SIMD operations for any reason, the algorithm can be mapped by replacing steps 6-8 and 14-16 (and all following instances of this block) with the following equivalent code:

1	FUSE8 R0, R10, R13, R19, 0x0E	FUSE8 R1, R11, R14, R17, 0x0E
2	LOG 0100, R0, R0, R23	LOG 0100, R1, R1, R23
3	FUSE8 R2, R11, R15, R18, 0x0E	FUSE8 R3, R9, R15, R19, 0x0E
4	LOG 0100, R2, R2, R21	LOG 0100, R3, R3, R22