Application Note

Akademie věd České republiky
Ústav teorie informace a automatizace AV ČR, v.v.i.

Eight FP03x8 accelerators for TE0808-09-EG-ES1 module on TEBF0808 carrier board

Jiři Kadlec,
kadlec@utia.cas.cz

Revision history

Rev.	Date	Author	Description
0	6.02.2021	J. Kadlec	Initial draft
1			
2			

Table of Contents

Table of Figures

Acknowledgement

This work has been partially supported from project FitOptiVis, project number ECSEL 783162 and the corresponding Czech NFA (MSMT) institutional support project 8A18013.

1 Evaluation version of FP03x8 accelerators for ZU09-EG-ES1

This application note describes an evaluation package with eight 8xSIMD, FP03x8, floating-point, run-time-reconfigurable accelerators for Zynq Ultrascale+ TE0808-09EG-ES1 module [1] on TEBF0808 carrier board [2]. The TE0808-09EG-ES1 module and TEBF0808 carrier board are designed and manufactured by the company Trenz Electronic [1]. Xilinx device ZU09-EG-ES1 device requires in the design phase Xilinx Vivado tools version 2017.4. These tools must have enabled support for the Xilinx ZU09-EG-ES1 device. The Xilinx Vivado 2017.4 is currently the last Xilinx toolchain supporting the ZU09-EG-ES1 device.

The evaluation package provides several pre-compiled HW designs (see Figure 1) represented in form of SD-cards containing the designs and API interface for SW developer in form of shared Debian Linux libraries. The SW developer can program ARM host application in standard gcc or g++ compiler and „make“ can be used for compilation of host applications directly on the embedded Zynq Ultrascale+ ZU09-EG-ES1 based system.

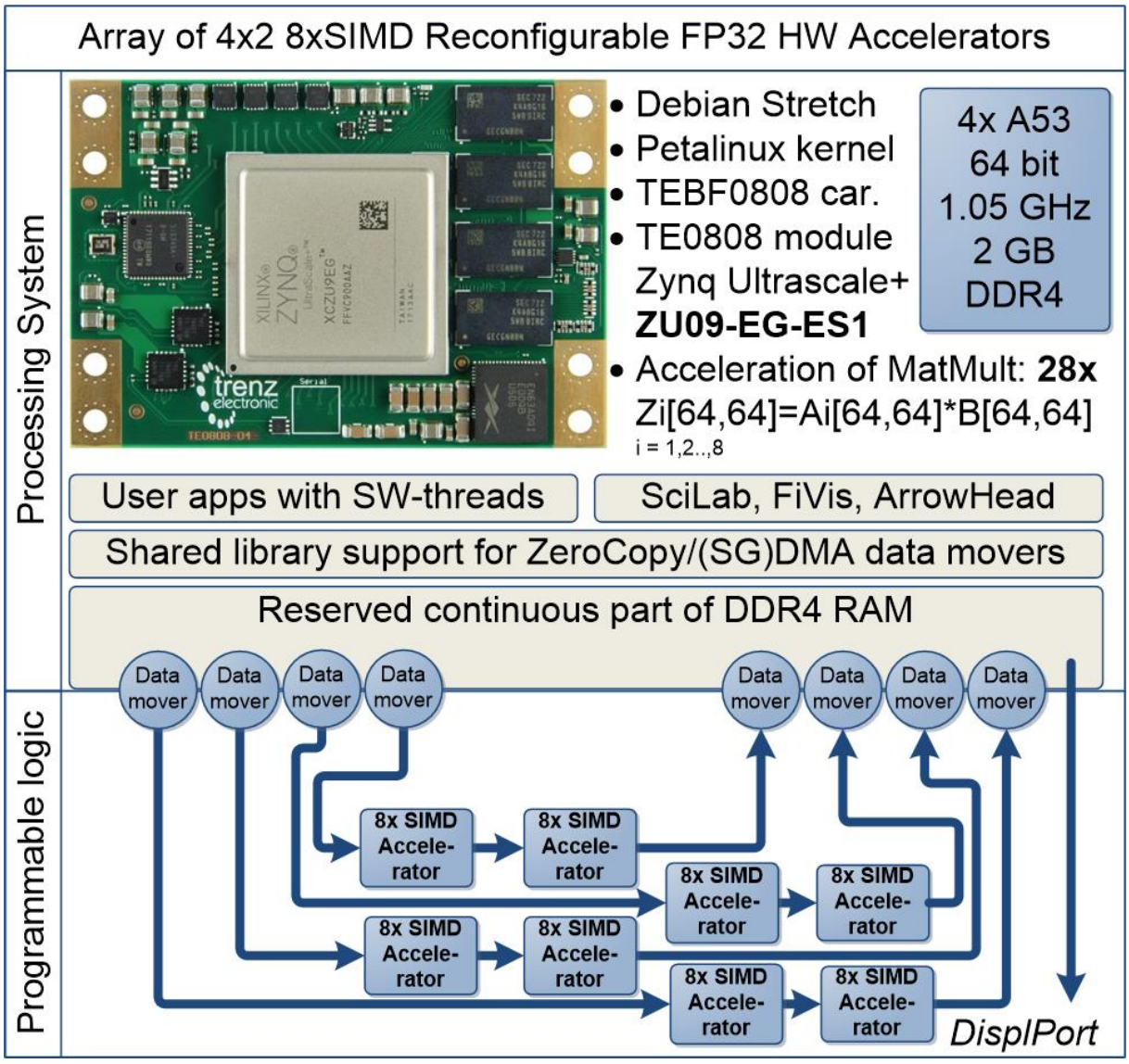

Figure 1: Eight connected FP03x8 accelerators in ZU09-EG-ES1 device

2 8xSIMD FP03x8 floating point accelerators for ZU09-EG-ES1

The FP03x8 HW accelerators serve for run-time reprogrammable 8xSIMD single precision floating point computation. The internal structure of FP03x8 accelerators is described in Figure 2.

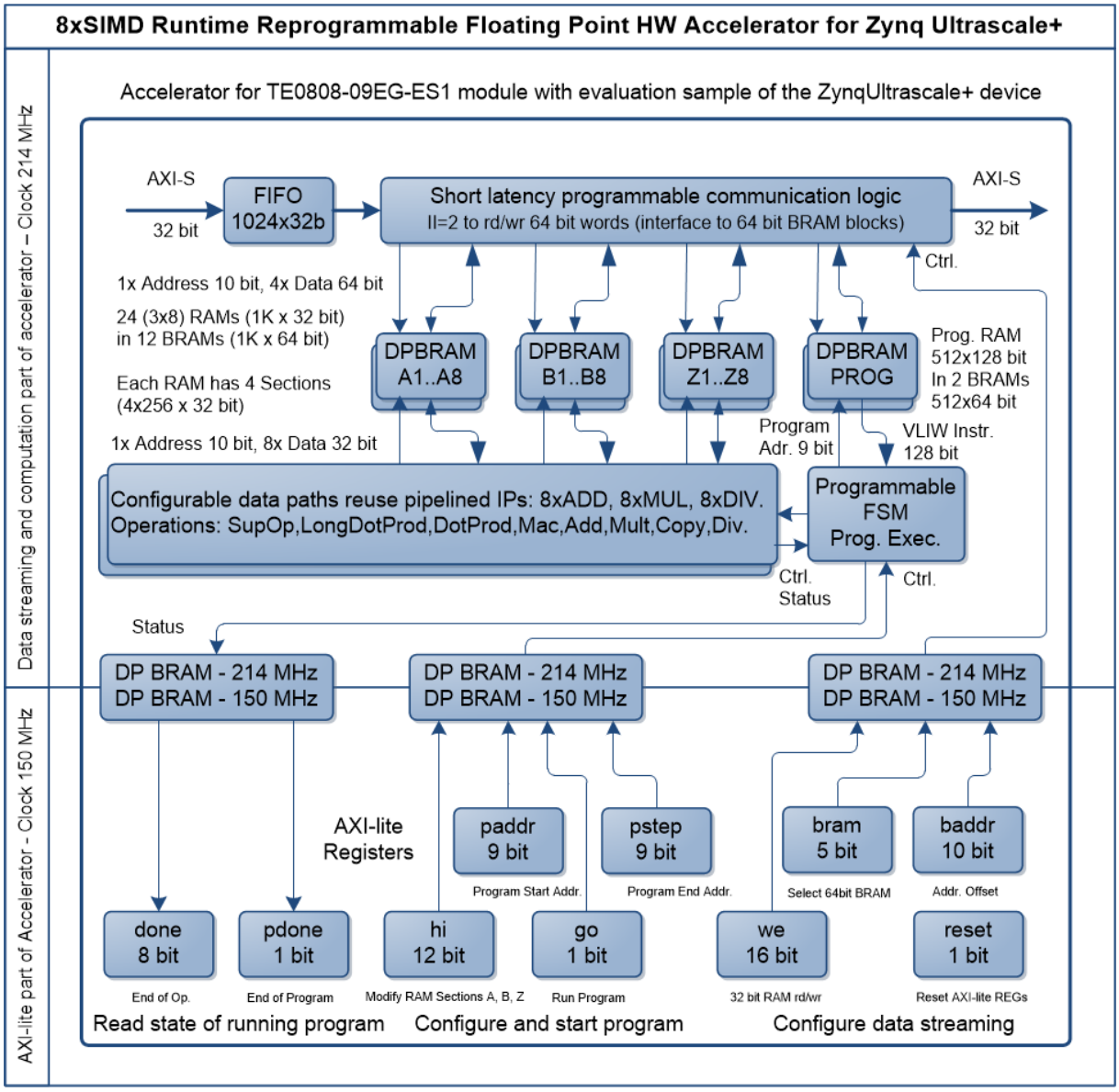

Figure 2: FP03x8 accelerator for the ZU09-EG-ES1 device

Input:

- Program firmware data received via AXI stream interface from Arm processor.
- Configuration Write registers for scalar control received via AXI-lite interface from Arm processor.
- Floating point single precision data received via AXI stream interface from Arm processor.

Output:

- Registers indicating end of program accessible to Arm processor via AXI-lite.
- Floating point single precision result data accessible via AXI stream interface for the Arm processor.

Connectivity:

- AXI stream data/program input from ARM to HW accelerator with input FIFO 1024x32. The side channel indicates the last transferred word sent to the component via the DMA transaction from ARM processor.
- AXI stream data/program output from HW accelerator to ARM. The output side channel indicates the last transferred word sent from the component to Arm processor.
- AXI-lite input/output configuration registers.

All designs present in this evaluation package contain four independent twins of serial connected FP03x8 accelerators in the programmable logic part of the device.

The HW data movers supporting the data communication are represented for the SW developer as shared c/C++ library with simple SW API. The API is identical for several alternatives of HW data movers.

The evaluation package includes 8xSIMD FP32 accelerators with HW license enabling only restricted number of operations. If these licensed operations are all used, user has to reset complete system. This will enable to use the licensed count of operations again.

Please contact UTIA (kadlec@utia.cas.cz) if you are interested in license for 8xSIMD accelerators without this restriction.

Accelerator Interfaces

Type of interface:	Device:	Clock:
• Data streaming I/O: AXI-S 32 bit	ZU09-EG-ES1	214 MHz
• Firmware program VLIW 128 bit	ZU09-EG-ES1	214 MHz
• Configuration I/O: AXI-lite 32 bit	ZU09-EG-ES1	150 MHz

Memory of the Accelerator in the programmable logic part of the device

- 12 dual-ported 1024x64 bit BRAMs Blocks (0 .. 11) are used as:
  - 24 Data RAMs organised as 1024x32 bit blocks: A1..A8, B1..B8 and Z1..Z8.
- 2 dual-ported 512x64 bit BRAMs Blocks (12, 13) are used as
  - 4 Program RAMs organised as 512x32 bit blocks: P1..P3

SIMD A 32 bit	Block 64 bit		SIMD B 32 bit	Block 64 bit		SIMD Z 32 bit	Block 64 bit		VLIW Prog	Block 64 bit
A1	0		B1	4		Z1	8		P1	12
A2			B2			Z2			P2	
A3	1		B3	5		Z3	9		P3	13
A4			B4			Z4			P4	
A5	2		B5	6		Z5	10			
A6			B6			Z6				
A7	3		B7	7		Z7	11			
A8			B8			Z8				

Figure 3: Internal block rams of accelerators.

AXI-lite Registers

Name:	Data:	Description:
reset	1 bit:	“1” Reset AXI lite Registers; “0” NOP
we	16 bit:	Write from stream to block(s) (bit 0 .. 13)
baddr	10 bit:	Stream will rd/wr from addr=baddr
bram	5 bit:	Read from Block 0 .. 13 to Stream; 16 for: Move-data-through
paddr	9 bit:	Program start address
pstep	9 bit:	Program stop address
go	1 bit:	“1” go from paddr to pstep; “0” NOP
hi	12 bit:	SubBank prog. mod: 00zz00bb00aa (bits)
done	8 bit:	Read only. “0” => Instruction runs
pdone	1 bit:	Read only. “0” => Program runs

Parameters of stream data interfaces from/to ARM DDR memory

- Maximal supported stream data size is 2048 x 32 bit
- Data streaming can have variable size:
  - Min: 2 x 32 bit
  - Max 2048 x 32 bit
- Mode of operation (same for Data and for Program):
  - **Write to a block**: It is defined by **we** (from address defined in **baddr)**
  - **Broadcast Write**: It is defined by setting more bits in **we** (from address defined in **baddr**)
  - **Read from block**: It is defined by setting **bram** (from address defined in **baddr)**
  - **Write or Broadcast Write and Read in parallel**: It is defined by setting more bits in **we** and by setting **bram** (from address defined in **baddr**)
  - **Send data through the Accelerator**: It is defined by setting **we** = 0 and by setting **bram** =16;

Design-time support

These data streaming HW data movers are supported:

- Zero Copy HW data mover without DMA
- DMA HW data mover with DMA
- SG DMA HW data mover with SG DMA and interrupts

The design time support is based on the Xilinx SDSoC 2017.4 system level compiler.

Run-time support

- Data can be written to and/or read from the accelerator by user Arm app.
- Firmware can be written to and/or read from the accelerator user Arm app.
- Computation & data streaming can be performed in parallel.

Versions of accelerators:

- **FP03x8**_capabilities capabilities = 10, 20, 30 or 40

SIMD OP	code (dec)	8xSIMD Floating Point Operation Description
VVER	0	Return capabilities of the accelerator and status of license
VZ2A	1	8xSIMD vector copy $a_m[i] <= z_m[j]$; m=1..8
VB2A	2	8xSIMD vector copy $a_m[i] <= b_m[j]$; m=1..8
VZ2B	3	8xSIMD vector copy $b_m[i] <= z_m[j]$; m=1..8
VA2B	4	8xSIMD vector copy $b_m[i] <= a_m[j]$; m=1..8
Auto-increments:		*Example: for (n=0;n<=CNT;n++){i=i+B_INC; j=j+A_INC;}*
VADD	5	8xSIMD vector add $z_m[i] <= a_m[j] + b_m[k]$; m=1..8
VADD_BZ2A	6	8xSIMD vector add $a_m[i] <= b_m[j] + z_m[k]$; m=1..8
VADD_AZ2B	7	8xSIMD vector add $b_m[i] <= a_m[j] + z_m[k]$; m=1..8
Auto-increments:		*Example: for (n=0;n<=CNT;n++){i=i+B_INC; j=j+A_INC; k=k+Z_INC;}*
VSUB	8	8xSIMD vector sub $z_m[i] <= a_m[j] - b_m[k]$; m=1..8
VSUB_BZ2A	9	8xSIMD vector sub $a_m[i] <= b_m[j] - z_m[k]$; m=1..8
VSUB_AZ2B	10	8xSIMD vector sub $b_m[i] <= a_m[j] - z_m[k]$; m=1..8
Auto-increments:		*Example: for (n=0;n<=CNT;n++){i=i+B_INC; j=j+A_INC; k=k+Z_INC;}*
VMULT	11	8xSIMD vector mult $z_m[i] <= a_m[j] * b_m[k]$; m=1..8
VMULT_BZ2A	12	8xSIMD vector mult $a_m[i] <= b_m[j] * z_m[k]$; m=1..8
VMULT_AZ2B	13	8xSIMD vector mult $b_m[i] <= a_m[j] * z_m[k]$; m=1..8
Auto-increments:		*Example: for (n=0;n<=CNT;n++){i=i+B_INC; j=j+A_INC; k=k+Z_INC;}*

Figure 4: Floating point functions present in all accelerators {10 or 20 or 30 or 40}.

SIMD OP	code (dec)	8xSIMD Floating Point Operation Description
VPROD FP01, FP03: 30,40	14	8xSIMD vector products. $z_m[i] <= a_m'[j..j+nn]*b_m[k..k+nn]$; m=1..8; nn range 0..255
VMAC FP01, FP03: 20,30,40	15	8xSIMD vector MACs. $z_m[i..i+nn] <= z_m[i..i+nn] + a_m[j..j+nn] * b_m[k..k+nn]$; m=1..8; nn range 0..10
VMSUBAC FP01, FP03: 20,30,40	16	8xSIMD vector MSUBACs. $z_m[i..i+nn] <= z_m[i..i+nn] - a_m[j..j+nn] * b_m[k..k+nn]$; m=1..8; nn range 0..10
LONG_VPROD FP01, FP03: 40	17	Single long vector product . $z_m[i] <= ((a_1'[j..j+nn]*b_1[k..k+nn]+a_2'[j..j+nn]*b_2[k..k+nn]) + (a_3'[j..j+nn]*b_3[k..k+nn]+a_4'[j..j+nn]*b_4[k..k+nn])) + ((a_5'[j..j+nn]*b_5[k..k+nn]+a_6'[j..j+nn]*b_6[k..k+nn]) + (a_7'[j..j+nn]*b_7[k..k+nn]+a_8'[j..j+nn]*b_8[k..k+nn]));$ m=1..8; nn range 0..255
VDIV FP03: 10,20,30,40 FP01: not supported	20	8xSIMD vector Division. $z_m[i] <= a_m[j] / b_m[k]$; m=1..8
Auto-increments:		*Example: for(n=0;n<=CNT;n++){i=i+Z_INC; j=j+A_INC; k=k+B_INC;}*

Figure 5: Specific functions present only in some versions accelerators.

3 Programming of 8xSIMD FP03x8 floating point accelerators

Host arm application can form the VLIW program instructions in DDR4 memory as two 64bit words. Components of the low 64 bit word are marked by light green background. Components of the high 64 bit word components are marked by light blue. See Figure 6.

Host arm application can form a sequence of such VLIW program instructions in DDR4 memory and write them to one of two accelerator program memories.

FP01, FP03	Size	VLIW: hi lo	Description
[not_used]	*[8bit]*	*8 bit [63..56]*	*Not used by FP01 or FP03*
[not_used]	*[8bit]*	*8 bit [55..48]*	*Not used by FP01 or FP03*
[0,Z_MEM_SECTION]	[0,2bit]	8 bit [47..40]	Z_MEM SECTION (0..3)
[CNT]	[8bit]	8 bit [39..32]	Number of 8xSIMD steps (0 .. 255)
[Z_INC]	[8bit]	8 bit [31..24]	Auto increment of Z address (0 .. 255)
[Z_MEM_SADDR]	[8bit]	8 bit [23..16]	Set Z address after auto-increment overflow
[Z_MEM_ADDR]	[8bit]	8 bit [15..08]	Initial Z address
[B_INC]	[8bit]	8 bit [07..00]	Auto increment of B address (0 .. 255)
[OP]	[8bit]	8 bit [63..56]	8xSIMD vector operation
[0, B_MEM_SECTION]	[0,2bit]	8 bit [55..48]	B_MEM SECTION (0..3)
[0, A_MEM_SECTION]	[0,2bit]	8 bit [47..40]	A_MEM SECTION (0..3)
[B_MEM_SADDR]	[8bit]	8 bit [39..32]	Set B address after auto-increment overflow
[B_MEM_ADDR]	[8bit]	8 bit [31..24]	Initial B address
[A_INC]	[8bit]	8 bit [23..16]	Auto increment of A address (0 .. 255)
[A_MEM_SADDR]	[8bit]	8 bit [15..08]	Set A address after auto-increment overflow
[A_MEM_ADDR]	[8bit]	8 bit [07..00]	Initial A address

Figure 6: Structure of the 128 bit wide VLIW program instruction.

Sequences of VLIW instructions present in the accelerator program memory can be autonomously executed by the accelerator (see Figure 2).

User defines start address in **paddr** AXI-lite register and end address in **pstep** AXI-lite register.

User requests execution of the sequence of VLIW operations by setting the single bit AXI-lite register **go** = 1. The accelerator executes the VLIV sequence from **paddr** to **pstep**.

State of the execution can be tested by the host application by reading of the AXI Read only register **pdone.** If **pdone**==0, the sequence of VLIW instructions is being executed.
If **pdone**==1, the sequence of VLIW instructions is completed.

Finally the host application has to set the single bit AXI-lite register back to **go** = 0.

The host application can also copy data/program to/from the accelerator while the sequence of VLIW instructions is being executed on current internal data and internal program of the accelerator.

This parallel copy data/program to/from the accelerator (while accelerators executes its sequence of VLIW instructions) requires to avoid race-condition caused by parallel writing to the same memory address both by accelerator and by parallel copy of data defined by the user in the same time instance. This has to be avoided by the user application, by writing only to accelerator data which are not used for writing by the currently executed sequence of VLIW instructions.

The sequence of VLIW instructions can be also reduced to a single VLIW instruction. The **paddr** and **pstep** registers are set to an identical program address in such case.

This technique can be used by the developer of host Arm program for stepping through the sequence of VLIW instructions one by one. User can modify the host application for reading partial results of each VLIW instruction. Data can be uploaded from the accelerator to host app for inspection and stepping through/debug in the Xilinx SDK 2017.4 gdb debugger GUI.

```
// Connections of eight 8xSIMD floating point Accelerators fp03x8_v26_4x2
// X={0,1,2,3,4,5,6,7}

     ___     ___
-->|___|-->|___|-->
    0       1

     ___     ___
-->|___|-->|___|-->
    2       3

     ___     ___
-->|___|-->|___|-->
    4       5

     ___     ___
-->|___|-->|___|-->
    6       7
// The fp03x8 driver instance data is defined for every fp03x8 accelerator
// A pointer to a variable of this type is then passed to the driver
// API functions.
typedef struct {
    UINTPTR fp03x8_BaseAddress;
    u32 IsReady;
} fp03x8;

// Declaration of driver instances
fp03x8 fp03x8_0_inst;
fp03x8 fp03x8_1_inst;
fp03x8 fp03x8_2_inst;
fp03x8 fp03x8_3_inst;
fp03x8 fp03x8_4_inst;
fp03x8 fp03x8_5_inst;
fp03x8 fp03x8_6_inst;
fp03x8 fp03x8_7_inst;
// Declaration of volatile variables
volatile u32 we,dest_we,reset,dest_reset,baddr,dest_baddr,bram,dest_bram;
volatile u32 go,dest_go,paddr,dest_paddr,pstep,dest_pstep,hi,dest_hi;
volatile u32 dest_done, dest_pdone;
```

```
//API functions for WR to accelerator instance AXI-lite registers
fp03x8_we_write(&fp03x8_X_inst, we);
fp03x8_reset_write(&fp03x8_X_inst, reset);
fp03x8_baddr_write(&fp03x8_X_inst, baddr);
fp03x8_bram_write(&fp03x8_X_inst, bram);
fp03x8_paddr_write(&fp03x8_X_inst, paddr);
fp03x8_pstep_write(&fp03x8_X_inst, pstep);
fp03x8_go_write(&fp03x8_X_inst, go);
fp03x8_hi_write(&fp03x8_X_inst, hi);
```

Figure 7: API functions for write to accelerator X={0,1,2,3,4,5,6,7} AXI-lite registers.

```
//Connections of eight 8xSIMD floating point Accelerators fp03x8_v26_4x2
//X={0,1,2,3,4,5,6,7}

     ___     ___
-->|___|-->|___|-->
    0       1

     ___     ___
-->|___|-->|___|-->
    2       3

     ___     ___
-->|___|-->|___|-->
    4       5

     ___     ___
-->|___|-->|___|-->
    6       7
// The fp03x8 driver instance data is defined for every fp03x8 accelerator
// A pointer to a variable of this type is then passed to the driver
// API functions.
typedef struct {
    UINTPTR fp03x8_BaseAddress;
    u32 IsReady;
} fp03x8;

// Declaration of driver instances
fp03x8 fp03x8_0_inst;
fp03x8 fp03x8_1_inst;
fp03x8 fp03x8_2_inst;
fp03x8 fp03x8_3_inst;
fp03x8 fp03x8_4_inst;
fp03x8 fp03x8_5_inst;
fp03x8 fp03x8_6_inst;
fp03x8 fp03x8_7_inst;

// Declaration of volatile variables
volatile u32 we,dest_we,reset,dest_reset,baddr,dest_baddr,bram,dest_bram;
volatile u32 go,dest_go,paddr,dest_paddr,pstep,dest_pstep,hi,dest_hi;
volatile u32 dest_done, dest_pdone;
// …
```

//APIfunctions for RD from accelerator instance AXI-lite registers
`dest_we = fp03x8_we_read(&fp03x8_X_inst);`
`dest_reset = fp03x8_reset_read(&fp03x8_X_inst);`
`dest_baddr = fp03x8_baddr_read(&fp03x8_X_inst);`
`dest_bram = fp03x8_bram_read(&fp03x8_X_inst);`
`dest_paddr = fp03x8_paddr_read(&fp03x8_X_inst);`
`dest_pstep = fp03x8_pstep_read(&fp03x8_X_inst);`
`dest_go = fp03x8_go_read(&fp03x8_X_inst);`
`dest_hi = fp03x8_hi_read(&fp03x8_X_inst);`
`dest_done = fp03x8_done_read(&fp03x8_X_inst);`
`dest_pdone = fp03x8_pdone_read(&fp03x8_X_inst);`

Figure 8: API functions for read from accelerator X={0,1,2,3,4,5,6,7} AXI-lite registers.

SW API for data or program streaming to/from accelerator

API for memory allocation and data or program transfer to and from chains of accelerators {0,1}, {2,3}, {4,5} and {6,7} is introduced in Figure 9.

```
// Data transfer by data movers is perfomed in 64bit in these two
// 64 bit wide data structures for data and for program.
#define NUM_ELEMENTS_DATA   2 // To make structure size 64bit
#define NUM_ELEMENTS_PROG   8 // To make structure size 64bit

typedef struct wide_dt_struct{
   float d[NUM_ELEMENTS_DATA];
} wide_dt;
typedef struct wide_dt_struct_prog{
   unsigned char p[NUM_ELEMENTS_PROG];
} wide_dt_prog;

// Memory allocation as continuous, non-cacheable block of data
// in Xilinx SDSoC 2017.4 runtime API
wide_dt *src_A1_A2 =
        (wide_dt *)sds_alloc_non_cacheable(block_data*sizeof(wide_dt));
// Example how to free continuous, non-cacheable block of data
sds_free(src_A1_A2);

// Memory allocation as standard linux data
// in Xilinx SDK 2017.4 runtime API
float *A = (float *) malloc(sizeof(float) * columns * rows);
// Example how to free allocated standard linux data
free(A);
```

//API functions for data/program transfer to/from accelerators

```
// data transfer by data movers to/from accelerators 0 and/or 1

     ___     ___
-->|___|-->|___|-->
     0       1
void data2hw_wrapper(unsigned *src, unsigned len);
void capture_wrapper(unsigned *storage, unsigned len);
```

```
// data or program transfer by data movers to/from accelerators 2 and/or 3

     ___     ___
-->|___|-->|___|-->
     2       3
void data2hw_wrapper1(unsigned *src, unsigned len);
void capture_wrapper1(unsigned *storage, unsigned len);
```

```
// data or program transfer by data movers to/from accelerators 4 and/or 5

     ___     ___
-->|___|-->|___|-->
     4       5
void data2hw_wrapper2(unsigned *src, unsigned len);
void capture_wrapper2(unsigned *storage, unsigned len);
```

```
// data or program transfer by data movers to/from accelerators 6 and/or 7

     ___     ___
-->|___|-->|___|-->
     6       7
void data2hw_wrapper3(unsigned *src, unsigned len);
void capture_wrapper3(unsigned *storage, unsigned len);
```

Figure 9: API functions for data/program transfer to/from accelerators.

HW version of matrix multiplication with 4 threads

The four chains of accelerators {0,1}, {2,3}, {4,5} and {6,7} are independent and can be configured and executed in threads started from the user host application. Threads can run in parallel under control of the Arm Linux kernel and on four cores of the Arm A53 processor. See Figure 10.

```
// Configuration and execution of HW accelerators controled in 4 threads

     ___     ___
-->|___|-->|___|-->
     0       1

     ___     ___
-->|___|-->|___|-->
     2       3

     ___     ___
-->|___|-->|___|-->
     4       5

     ___     ___
-->|___|-->|___|-->
     6       7
// Data structures for threads
struct thread_args data =  {
   fp03x8_0_inst, fp03x8_1_inst,
   src_B_01,dest_B_01,src_B_02,dest_B_02,
   src_B_03,dest_B_03,src_B_04,dest_B_04,
   src_B_05, dest_B_05, src_B_06, dest_B_06,
   src_B_07, dest_B_07, src_B_08, dest_B_08};

struct thread_args_1 data_1 =  {
   fp03x8_2_inst, fp03x8_3_inst,
   src_B_01_1, dest_B_01_1, src_B_02_1, dest_B_02_1,
   src_B_03_1, dest_B_03_1, src_B_04_1, dest_B_04_1,
   src_B_05_1, dest_B_05_1, src_B_06_1, dest_B_06_1,
   src_B_07_1, dest_B_07_1, src_B_08_1, dest_B_08_1};

struct thread_args_2 data_2 =  {
   fp03x8_4_inst, fp03x8_5_inst,
   src_B_01_2, dest_B_01_2, src_B_02_2, dest_B_02_2,
   src_B_03_2, dest_B_03_2, src_B_04_2, dest_B_04_2,
   src_B_05_2, dest_B_05_2, src_B_06_2, dest_B_06_2,
   src_B_07_2, dest_B_07_2, src_B_08_2, dest_B_08_2};

struct thread_args_3 data_3 =  {
   fp03x8_6_inst, fp03x8_7_inst,
   src_B_01_3, dest_B_01_3, src_B_02_3, dest_B_02_3,
   src_B_03_3, dest_B_03_3, src_B_04_3, dest_B_04_3,
   src_B_05_3, dest_B_05_3, src_B_06_3, dest_B_06_3,
   src_B_07_3, dest_B_07_3, src_B_08_3, dest_B_08_3};

// 8 matrix [64,64] floating point multiplications controlled from
// 4 threads t, t_1, t_2 and t_3 executed potencially in parallel
// on 4 Arm cores controling 8 hw accelerators connected in 4 AXI
// data and program streams with instantiated HW data movers.

pthread_create(&t,   NULL, hw_thrd,   &data);
pthread_create(&t_1, NULL, hw_thrd_1, &data_1);
pthread_create(&t_2, NULL, hw_thrd_2, &data_2);
pthread_create(&t_3, NULL, hw_thrd_3, &data_3);
```

```
pthread_join(t, NULL);
pthread_join(t_1, NULL);
pthread_join(t_2, NULL);
pthread_join(t_3, NULL);
```

Figure 10: Configuration and execution of HW accelerators controlled in 4 threads

SW version of matrix multiplication with 4 threads

The SW version of matrix multiplication can be executed in threads started from the user host application. Threads can run in parallel under control of the Arm Linux kernel and on four cores of the Arm A53 processor. See Figure 11.

```
// Matrix multiplication in 4 SW threads

void matmul(float *C, float *A, float *B, int M) {
   for (int k = 0; k < M; k++) {
      for (int j = 0; j < M; j++) {
         for (int i = 0; i < M; i++) {
            C[k + j * M] += A[k + i * M] * B[i + j * M];
         }
      }
   }
}

struct thread_sw_args{
      float * A; float * A1; float * B; float * gold; float * gold1;
      int columns; int rows;
};

struct thread_sw_args_1{
      float * A2; float * A3;float * B1;float * gold2;float * gold3;
      int columns; int rows;
};

struct thread_sw_args_2{
      float * A4;float * A5;float * B2;float * gold4;float * gold5;
      int columns;int rows;
};

struct thread_sw_args_3{
      float * A6;float * A7;float * B3;float * gold6;float * gold7;
      int columns;int rows;
};

void *sw_thrd(void *ptr){
  int i, j;
  struct thread_sw_args *data = (struct thread_sw_args *)ptr;
  for (j = 0; j < 100; j++){
      for(i = 0; i < data->columns * data->rows; i++) {
        data->gold[i] = (float) 0.0;
        data->gold1[i] = (float) 0.0;
      }
      matmul(data->gold, data->A, data->B, data->columns);
      matmul(data->gold1, data->A1, data->B, data->columns);
  }
  return NULL;
```

```
}
void *sw_thrd_1(void *ptr){
  int i, j;
  struct thread_sw_args_1 *data = (struct thread_sw_args_1 *)ptr;
  for (j = 0; j < 100; j++){
       for(i = 0; i < data->columns * data->rows; i++) {
          data->gold2[i] = (float) 0.0;
          data->gold3[i] = (float) 0.0;
       }
       matmul(data->gold2, data->A2, data->B1, data->columns);
       matmul(data->gold3, data->A3, data->B1, data->columns);
  }
  return NULL;
}
void *sw_thrd_2(void *ptr){
  int i, j;
  struct thread_sw_args_2 *data = (struct thread_sw_args_2 *)ptr;
  for (j = 0; j < 100; j++){
       for(i = 0; i < data->columns * data->rows; i++) {
          data->gold4[i] = (float) 0.0;
          data->gold5[i] = (float) 0.0;
       }
       matmul(data->gold4, data->A4, data->B2, data->columns);
       matmul(data->gold5, data->A5, data->B2, data->columns);
   }
   return NULL;
}
void *sw_thrd_3(void *ptr){
   int i, j;
   struct thread_sw_args_3 *data = (struct thread_sw_args_3 *)ptr;
   for (j = 0; j < 100; j++){
       for(i = 0; i < data->columns * data->rows; i++) {
          data->gold6[i] = (float) 0.0;
          data->gold7[i] = (float) 0.0;
       }
       matmul(data->gold6, data->A6, data->B3, data->columns);
       matmul(data->gold7, data->A7, data->B3, data->columns);
   }
   return NULL;
}

// User application host code for Aem A53. Four SW threads.
struct thread_sw_args   data_sw   = {A, A1,B, gold, gold1,columns,rows};
struct thread_sw_args_1 data_sw_1 = {A2,A3,B1,gold2,gold3,columns,rows};
struct thread_sw_args_2 data_sw_2 = {A4,A5,B2,gold4,gold5,columns,rows};
struct thread_sw_args_3 data_sw_3 = {A6,A7,B3,gold6,gold7,columns,rows};

// Eight matrix multiplications as four SW threads on arm A53 (4 cores):
pthread_create(&t_sw,   NULL, sw_thrd,   &data_sw);
pthread_create(&t_sw_1, NULL, sw_thrd_1, &data_sw_1);
pthread_create(&t_sw_2, NULL, sw_thrd_2, &data_sw_2);
pthread_create(&t_sw_3, NULL, sw_thrd_3, &data_sw_3);
pthread_join(t_sw, NULL);
pthread_join(t_sw_1, NULL);
pthread_join(t_sw_2, NULL);
pthread_join(t_sw_3, NULL);
```

Figure 11: Eight matrix multiplications, four Arm A53 processor SW threads.

SciLab reference script, 8 matrix multiplications, 4 mex calls

Figure 12 is listing SciLab script serving as golden reference model. Matrix multiplications are implemented by four sequential calls to C mex function which performes in floating point two matrix multiplications. Execution tome is measured and data are stored to header files. Header files are used in the SW and HW implementation on ZU09-EG-ES1 device for definition of input data and also for verification of results of computation.

Script can be executed in Arm A53 version of Debian version of SciLab on ZU09-EG-ES1 device. Script can be also executed on X86 PC Ubuntu version of SciLab (Intel I7 laptop). The single threaded performance can be evaluated and compared.

```
// Display mode
mode(0);
// Display warning for floating point exception
ieee(1);
clear
exec("multf_1x2_loader.sce");

L=100;
N=64;
A= ones(N,N);
A1=A; A2=A; A3=A; A4=A; A5=A; A6=A; A7=A;
B=A;
B1=A; B2=A; B3=A;
Z=A;
Z1=A; Z2=A; Z3=A; Z4=A; Z5=A; Z6=A; Z7=A;

for i=1:N
  for j=1:N
    A(i,j)  = 1.0 + i * 0.01  + j * 0.02;
    A1(i,j) = 1.0 + i * 0.001 + j * 0.0001;
    A2(i,j) = 1.0 + i * 0.002 + j * 0.0002;
    A3(i,j) = 1.0 + i * 0.003 + j * 0.0003;
    A4(i,j) = 1.0 + i * 0.004 + j * 0.0004;
    A5(i,j) = 1.0 + i * 0.005 + j * 0.0005;
    A6(i,j) = 1.0 + i * 0.006 + j * 0.0006;
    A7(i,j) = 1.0 + i * 0.007 + j * 0.0007;
    B(i,j)  = 1.0 + i * 0.001 + j * 0.002;
    B1(i,j) = 1.0 + i * 0.002 + j * 0.003;
    B2(i,j) = 1.0 + i * 0.003 + j * 0.004;
    B3(i,j) = 1.0 + i * 0.004 + j * 0.005;
 end
end

tic;
  [A,  B, Z,  A1, Z1, L, N]  = multf_1x2(A,  B, Z,  A1, Z1, L, N);
  [A2, B1, Z2, A3, Z3, L, N] = multf_1x2(A2, B, Z2, A3, Z3, L, N);
  [A4, B2, Z4, A5, Z5, L, N] = multf_1x2(A4, B, Z4, A5, Z5, L, N);
  [A6, B3, Z6, A7, Z7, L, N] = multf_1x2(A6, B, Z6, A7, Z7, L, N);
comp_input1 = toc();

comp_input1s = string(comp_input1);
disp("The 4x2 matrix multiplications computed in "+comp_input1s+"
seconds");

fd = mopen('.\A.h','w'); M_out = A;
```

```
MN = size(M_out); mfprintf(fd,'%s%d%s\n','static float
mat_a[',MN(1)*MN(2),'] = {');
for j=1:MN(2) for i=1:MN(1)
if (i*j == MN(1)*MN(2)) mfprintf(fd,'%18.12e\n',M_out(i,j)); else
mfprintf(fd,'%18.12e%s\n',M_out(i,j),','); end
end end mfprintf(fd,'%s\n','};'); mclose(fd);

fd = mopen('.\A1.h','w'); M_out = A1;
MN = size(M_out); mfprintf(fd,'%s%d%s\n','static float
mat_a1[',MN(1)*MN(2),'] = {');
for j=1:MN(2) for i=1:MN(1)
if (i*j == MN(1)*MN(2)) mfprintf(fd,'%18.12e\n',M_out(i,j)); else
mfprintf(fd,'%18.12e%s\n',M_out(i,j),','); end
end end mfprintf(fd,'%s\n','};'); mclose(fd);

fd = mopen('.\A2.h','w'); M_out = A2;
MN = size(M_out); mfprintf(fd,'%s%d%s\n','static float
mat_a2[',MN(1)*MN(2),'] = {');
for j=1:MN(2) for i=1:MN(1)
if (i*j == MN(1)*MN(2)) mfprintf(fd,'%18.12e\n',M_out(i,j)); else
mfprintf(fd,'%18.12e%s\n',M_out(i,j),','); end
end end mfprintf(fd,'%s\n','};'); mclose(fd);

fd = mopen('.\A3.h','w'); M_out = A3;
MN = size(M_out); mfprintf(fd,'%s%d%s\n','static float
mat_a3[',MN(1)*MN(2),'] = {');
for j=1:MN(2) for i=1:MN(1)
if (i*j == MN(1)*MN(2)) mfprintf(fd,'%18.12e\n',M_out(i,j)); else
mfprintf(fd,'%18.12e%s\n',M_out(i,j),','); end
end end mfprintf(fd,'%s\n','};'); mclose(fd);

fd = mopen('.\A4.h','w'); M_out = A4;
MN = size(M_out); mfprintf(fd,'%s%d%s\n','static float
mat_a4[',MN(1)*MN(2),'] = {');
for j=1:MN(2) for i=1:MN(1)
if (i*j == MN(1)*MN(2)) mfprintf(fd,'%18.12e\n',M_out(i,j)); else
mfprintf(fd,'%18.12e%s\n',M_out(i,j),','); end
end end mfprintf(fd,'%s\n','};'); mclose(fd);

fd = mopen('.\A5.h','w'); M_out = A5;
MN = size(M_out); mfprintf(fd,'%s%d%s\n','static float
mat_a5[',MN(1)*MN(2),'] = {');
for j=1:MN(2) for i=1:MN(1)
if (i*j == MN(1)*MN(2)) mfprintf(fd,'%18.12e\n',M_out(i,j)); else
mfprintf(fd,'%18.12e%s\n',M_out(i,j),','); end
end end mfprintf(fd,'%s\n','};'); mclose(fd);

fd = mopen('.\A6.h','w'); M_out = A6;
MN = size(M_out); mfprintf(fd,'%s%d%s\n','static float
mat_a6[',MN(1)*MN(2),'] = {');
for j=1:MN(2) for i=1:MN(1)
if (i*j == MN(1)*MN(2)) mfprintf(fd,'%18.12e\n',M_out(i,j)); else
mfprintf(fd,'%18.12e%s\n',M_out(i,j),','); end
end end mfprintf(fd,'%s\n','};'); mclose(fd);

fd = mopen('.\A7.h','w'); M_out = A7;
MN = size(M_out); mfprintf(fd,'%s%d%s\n','static float
mat_a7[',MN(1)*MN(2),'] = {');
```

```
for j=1:MN(2) for i=1:MN(1)
if (i*j == MN(1)*MN(2)) mfprintf(fd,'%18.12e\n',M_out(i,j)); else
mfprintf(fd,'%18.12e%s\n',M_out(i,j),','); end
end end mfprintf(fd,'%s\n','};'); mclose(fd);

fd = mopen('.\B.h','w'); M_out = B;
MN = size(M_out); mfprintf(fd,'%s%d%s\n','static float
mat_b[',MN(1)*MN(2),'] = {');
for j=1:MN(2) for i=1:MN(1)
if (i*j == MN(1)*MN(2)) mfprintf(fd,'%18.12e\n',M_out(i,j)); else
mfprintf(fd,'%18.12e%s\n',M_out(i,j),','); end
end end mfprintf(fd,'%s\n','};'); mclose(fd);

fd = mopen('.\B1.h','w'); M_out = B1;
MN = size(M_out); mfprintf(fd,'%s%d%s\n','static float
mat_b1[',MN(1)*MN(2),'] = {');
for j=1:MN(2) for i=1:MN(1)
if (i*j == MN(1)*MN(2)) mfprintf(fd,'%18.12e\n',M_out(i,j)); else
mfprintf(fd,'%18.12e%s\n',M_out(i,j),','); end
end end mfprintf(fd,'%s\n','};'); mclose(fd);

fd = mopen('.\B2.h','w'); M_out = B2;
MN = size(M_out); mfprintf(fd,'%s%d%s\n','static float
mat_b2[',MN(1)*MN(2),'] = {');
for j=1:MN(2) for i=1:MN(1)
if (i*j == MN(1)*MN(2)) mfprintf(fd,'%18.12e\n',M_out(i,j)); else
mfprintf(fd,'%18.12e%s\n',M_out(i,j),','); end
end end mfprintf(fd,'%s\n','};'); mclose(fd);

fd = mopen('.\B3.h','w'); M_out = B3;
MN = size(M_out); mfprintf(fd,'%s%d%s\n','static float
mat_b3[',MN(1)*MN(2),'] = {');
for j=1:MN(2) for i=1:MN(1)
if (i*j == MN(1)*MN(2)) mfprintf(fd,'%18.12e\n',M_out(i,j)); else
mfprintf(fd,'%18.12e%s\n',M_out(i,j),','); end
end end mfprintf(fd,'%s\n','};'); mclose(fd);

fd = mopen('.\Z.h','w'); M_out = Z;
MN = size(M_out); mfprintf(fd,'%s%d%s\n','static float
mat_z[',MN(1)*MN(2),'] = {');
for j=1:MN(2) for i=1:MN(1)
if (i*j == MN(1)*MN(2)) mfprintf(fd,'%18.12e\n',M_out(i,j)); else
mfprintf(fd,'%18.12e%s\n',M_out(i,j),','); end
end end mfprintf(fd,'%s\n','};'); mclose(fd);

fd = mopen('.\Z1.h','w'); M_out = Z1;
MN = size(M_out); mfprintf(fd,'%s%d%s\n','static float
mat_z1[',MN(1)*MN(2),'] = {');
for j=1:MN(2) for i=1:MN(1)
if (i*j == MN(1)*MN(2)) mfprintf(fd,'%18.12e\n',M_out(i,j)); else
mfprintf(fd,'%18.12e%s\n',M_out(i,j),','); end
end end mfprintf(fd,'%s\n','};'); mclose(fd);

fd = mopen('.\Z2.h','w'); M_out = Z2;
MN = size(M_out); mfprintf(fd,'%s%d%s\n','static float
mat_z2[',MN(1)*MN(2),'] = {');
for j=1:MN(2) for i=1:MN(1)
if (i*j == MN(1)*MN(2)) mfprintf(fd,'%18.12e\n',M_out(i,j)); else
```

```
mfprintf(fd,'%18.12e%s\n',M_out(i,j),','); end
end end mfprintf(fd,'%s\n','};'); mclose(fd);

fd = mopen('.\Z3.h','w'); M_out = Z3;
MN = size(M_out); mfprintf(fd,'%s%d%s\n','static float
mat_z3[',MN(1)*MN(2),'] = {');
for j=1:MN(2) for i=1:MN(1)
if (i*j == MN(1)*MN(2)) mfprintf(fd,'%18.12e\n',M_out(i,j)); else
mfprintf(fd,'%18.12e%s\n',M_out(i,j),','); end
end end mfprintf(fd,'%s\n','};'); mclose(fd);

fd = mopen('.\Z4.h','w'); M_out = Z4;
MN = size(M_out); mfprintf(fd,'%s%d%s\n','static float
mat_z4[',MN(1)*MN(2),'] = {');
for j=1:MN(2) for i=1:MN(1)
if (i*j == MN(1)*MN(2)) mfprintf(fd,'%18.12e\n',M_out(i,j)); else
mfprintf(fd,'%18.12e%s\n',M_out(i,j),','); end
end end mfprintf(fd,'%s\n','};'); mclose(fd);

fd = mopen('.\Z5.h','w'); M_out = Z5;
MN = size(M_out); mfprintf(fd,'%s%d%s\n','static float
mat_z5[',MN(1)*MN(2),'] = {');
for j=1:MN(2) for i=1:MN(1)
if (i*j == MN(1)*MN(2)) mfprintf(fd,'%18.12e\n',M_out(i,j)); else
mfprintf(fd,'%18.12e%s\n',M_out(i,j),','); end
end end mfprintf(fd,'%s\n','};'); mclose(fd);

fd = mopen('.\Z6.h','w'); M_out = Z6;
MN = size(M_out); mfprintf(fd,'%s%d%s\n','static float
mat_z6[',MN(1)*MN(2),'] = {');
for j=1:MN(2) for i=1:MN(1)
if (i*j == MN(1)*MN(2)) mfprintf(fd,'%18.12e\n',M_out(i,j)); else
mfprintf(fd,'%18.12e%s\n',M_out(i,j),','); end
end end mfprintf(fd,'%s\n','};'); mclose(fd);

fd = mopen('.\Z7.h','w'); M_out = Z7;
MN = size(M_out); mfprintf(fd,'%s%d%s\n','static float
mat_z7[',MN(1)*MN(2),'] = {');
for j=1:MN(2) for i=1:MN(1)
if (i*j == MN(1)*MN(2)) mfprintf(fd,'%18.12e\n',M_out(i,j)); else
mfprintf(fd,'%18.12e%s\n',M_out(i,j),','); end
end end mfprintf(fd,'%s\n','};'); mclose(fd);
```

Figure 12: SciLab script, eight matrix multiplications; write data to header files.

SciLab C mex function performing two matrix multiplications

Figure 13 provides listing of C mex function performing 2 matrix multiplication on floating point single precision.

Mex function can be compiled with -O3 optimisation and executed in Arm A53 Debian version of SciLab on ZU09-EG-ES1 device and also on X86 PC Ubuntu version of SciLab (Intel I7 laptop). The single threaded performance can be evaluated and compared.

```
#include <math.h>

void multf_1x2(a, b, z, a1, z1, nn)
float *a, *b, *z, *a1, *z1;
int nn;
{
  int i,j,k;
  int a_dim1, a_offset, b_dim1, b_offset, z_dim1, z_offset;
  int a1_dim1, a1_offset, z1_dim1, z1_offset;
  a_dim1 = nn;  a_offset = a_dim1 + 1;   a -= a_offset;
  b_dim1 = nn;  b_offset = b_dim1 + 1;   b -= b_offset;
  z_dim1 = nn;  z_offset = z_dim1 + 1;   z -= z_offset;
  a1_dim1 = nn; a1_offset = a1_dim1 + 1; a1 -= a1_offset
  z1_dim1 = nn; z1_offset = z1_dim1 + 1; z1 -= z1_offset;
  for (k = 1; k <= z_dim1; k++) {
    for (j = 1; j <= a_dim1; j++) {
      z[k + j * z_dim1] = (float) 0.0;
      for (i = 1; i <= b_dim1; i++) {
        z[k + j * z_dim1] += a[k + i*a_dim1]*b[i + j*b_dim1];
      }
    }
  }
  for (k = 1; k <= z1_dim1; k++) {
    for (j = 1; j <= a1_dim1; j++) {
      z1[k + j * z1_dim1] = (float) 0.0;
      for (i = 1; i <= b_dim1; i++) {
        z1[k + j*z1_dim1] += a1[k + i*a1_dim1]*b[i + j*b_dim1];
      }
    }
  }
}

void multf_1x2_(a, b, z, a1, z1, l, n)
float *a, *b, *z, *a1, *z1, *l, *n;
{
  int nn, ll, ii;
  int a_dim1, a_offset, b_dim1, b_offset, z_dim1, z_offset;
  int a1_dim1, a1_offset, z1_dim1, z1_offset;
  ll = (int) *l;
  nn = (int) *n;
  a_dim1 = nn;  a_offset = a_dim1 + 1;   a -= a_offset;
  b_dim1 = nn;  b_offset = b_dim1 + 1;   b -= b_offset;
  z_dim1 = nn;  z_offset = z_dim1 + 1;   z -= z_offset;
  a1_dim1 = nn; a1_offset = a1_dim1 + 1; a1 -= a1_offset;
  z1_dim1 = nn; z1_offset = z1_dim1 + 1; z1 -= z1_offset;
  for (ii = 1; ii <= ll; ii++){
    multf_1x2(&a[a_offset], &b[b_offset], &z[z_offset],
              &a1[a1_offset], &z1[z1_offset], nn);
  }
  return;
}
#define  A      plhs[0]
#define  B      plhs[1]
#define  Z      plhs[2]
#define  A1     plhs[3]
#define  Z1     plhs[4]
#define  L      plhs[5]
#define  N      plhs[6]
```

```
#include "mex.h"
void mexFunction(int nlhs, mxArray *plhs[], int nrhs, const mxArray
*prhs[]) {
  int i, j, m, n;
  double *s, *d;
  float  *sf, *df;
  if (nlhs != 7)
    mexErrMsgTxt("Define correct number of input arg.");
  if (nrhs != nlhs)
    mexErrMsgTxt("Define same output arguments as input arg.");
  for (i = 0; i < nrhs; i++) {
    m = mxGetM(prhs[i]);
    n = mxGetN(prhs[i]);
    plhs[i] = mxCreateDoubleMatrix(m, n, mxREAL);
    s = mxGetPr(prhs[i]);
    d = mxGetPr(plhs[i]);
    for (j = 0; j < m * n; j++) {
      *d = *s;
      d++;
      s++;
    }
  }
  for (i = 0; i < nrhs; i++) {
    m = mxGetM(plhs[i]);
    n = mxGetN(plhs[i]);
    s = mxGetPr(plhs[i]);
    df= (float *) mxGetPr(plhs[i]);
    for (j = 0; j < m * n; j++) {
      *df = (float) *s;
      df++;
      s++;
    }
  }
  multf_1x2_((float *) mxGetPr(A),(float *) mxGetPr(B),
             (float *) mxGetPr(Z),(float *) mxGetPr(A1),
             (float *) mxGetPr(Z1),(float *) mxGetPr(L),
             (float *) mxGetPr(N));
  for (i = 0; i < nrhs; i++) {
    m = mxGetM(plhs[i]);
    n = mxGetN(plhs[i]);
    sf= (float *) mxGetPr(plhs[i]);
    d = mxGetPr(plhs[i]);
    sf= sf + (m * n) -1;
    d = d + (m * n) -1;
    for (j = 0; j < m * n; j++) {
       *d = (double) *sf;
       d--;
       sf--;
    }
  }
}
```

Figure 13: SciLab C mex-function serves as golden model of two matrix multiplications.

4 Performance comparison

Acceleration of eight single precision floating point matrix by matrix multiplications has been prepared as an application example to evaluate performance of the released ZU09-EG-ES1 system with eight HW accelerators. See Figure 14.

The eight instances of the FP03x8 accelerators on ZU09-EG-ES1 device are controlled by 4 SW threads (see Figure 10). HW accelerators accelerate the SW optimized (-O3) 4-thread implementation (see Figure 11) on A53 processor executed on 4 A53 cores (with 1.05 GHz clock) **28x** on Zynq system with Zero Copy data mover HW IPs running at 214 MHz. We provide also comparison to SciLab C Mex implementations on PC and Arm. See Figure 14.

ZU09-EG-ES1 Floating point HW 4x2 FP03x8, 214MHz, 4 threads**:**	18 355 MFLOPs
I7 PC 3.0 GHz Floating point SW Ubuntu, SciLab C mex, 1 thread:	1 933 MFLOPs
ZU09-EG-ES1 Floating point SW Debian, C code, 4 threads:	650 MFLOPs
ZU09-EG-ES1 Floating point SW Debian, SciLab C mex, 1 thread:	166 MFLOPs

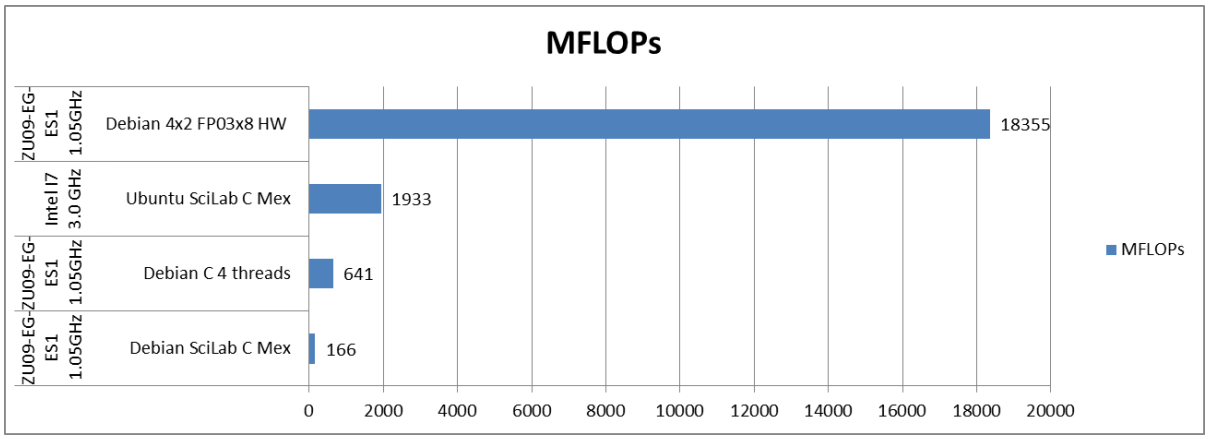

Figure 14: Performance comparison for floating point matrix multiplications (ops: +,-,*).

We have performed also initial comparison for QRD Lattice Filters with system order 256 and 3x256. Filters are internally systolic arrays. This allows for implementation in 4 SW threads, with system order 256. Filters require in total for each time step 34 850 floating point operations (ops: +,-,*,/). From this count, 8200 operations are floating point division.

We have not managed to implement QRD Lattice Filters on the 4x2 FP03x8 HW accelerators, yet. This is still work in progress and the real target for the implemented 4x2 FP03x8 array of HW accelerators. See Figure 15.

ZU09-EG-ES1 Floating point HW 4x2 FP03x8, 214 MHz, 4 threads**:**	*not implemented*
I7 PC 3.0 GHz Floating point SW Ubuntu, SciLab C mex, 1 thread:	1 096 MFLOPs
ZU09-EG-ES1 Floating point SW Debian, C code, 4 threads:	341 MFLOPs
ZU09-EG-ES1 Floating point SW Debian, SciLab C mex, 1 thread:	96 MFLOPs

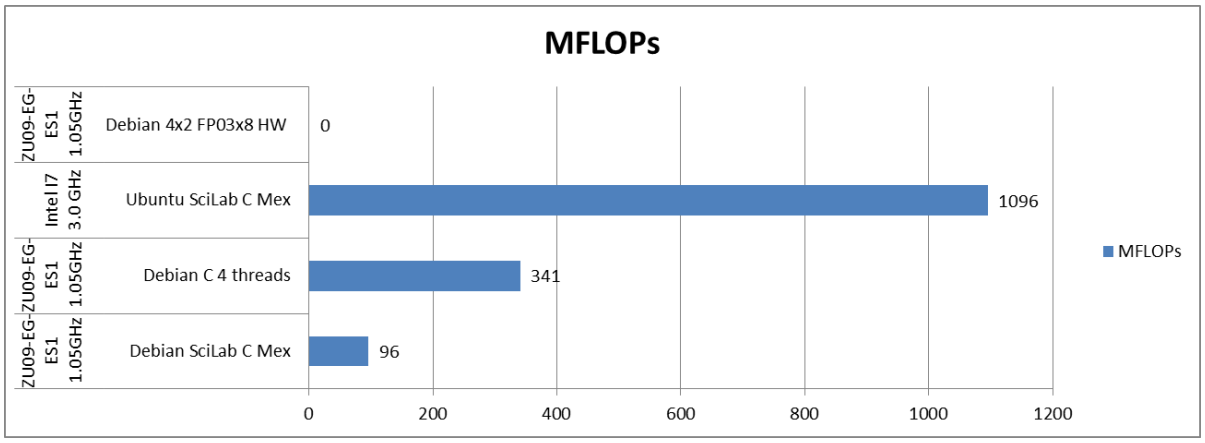

Figure 15: Performance comparison for floating point QRD Lattice Filters (ops: +,-,*,/).

5 Evaluation versions of shared libraries

The PL part of the ZU09-EG-ES1 device contains eight evaluation versions of the 8xSIMD run-time-reprogrammable single-precision-floating-point HW accelerator FP03x8 organized as 4x2 accelerators. Released four evaluation HW platforms exported for linking with the arm A53 host programs have these properties:

- C project: **fp03x8_v26_4x2_md45_c_sw** (4x SW Lattice, double precision)
  Shared library: **./Debug/libfp03x8_v26_4x2_md45_c_hw.so**
  Shared library: **./Release/libfp03x8_v26_4x2_md45_c_hw.so**
  C project: **fp03x8_v26_4x2_md45f_c_sw** (4x SW Lattice, single precision)
  Shared library: **./Debug/libfp03x8_v26_4x2_md45f_c_hw.so**
  Shared library: **./Release/libfp03x8_v26_4x2_md45f_c_hw.so**

  Debug and Release version of shared libraries Debian Stretch 9.8 OS for the ZU09EG-ES1 device for the SDK 2017.4 C SW flow (gcc compiler).
  PL is working with 4x2 evaluation versions of FP03x8 HW accelerators. It is using Zero Copy HW data movers. The data movers are realised as C functions compiled to HW by the SDSoC 2017.4 compiler. The HW supported data transfers require data to be present in “sd_alloc” memory (continuous physical section reserved in the DDR4). The end of data transfers are tested by pooling. The SW overhead needed to start this type of data transfer is relatively low.

- C++ project: **fp03x8_v26_4x2_md45_sw** (4x SW Lattice, double precision)
  Shared library: **./Debug/libfp03x8_v26_4x2_md45_hw.so**
  Shared library: **./Release/libfp03x8_v26_4x2_md45_hw.so**
  C++ project: **fp03x8_v26_4x2_md45f_sw** (4x SW Lattice, single precision)
  Shared library: **./Debug/libfp03x8_v26_4x2_md45f_dma_hw.so**
  Shared library: **./Release/libfp03x8_v26_4x2_md45f_dma_hw.so**

  Debug and Release version of shared libraries Debian Stretch 9.8 OS for the ZU09EG-ES1 device for the SDK 2017.4 C++ SW flow (g++ compiler).PL is working with 4x2 evaluation versions of FP03x8 HW accelerators. It is using Zero Copy HW data movers. The data movers are realized as C++ functions compiled to HW by the SDSoC 2017.4 compiler. The HW supported data transfers require data to be present in “sd_alloc” memory (continuous physical section reserved in the DDR4). The end of data transfers

are tested by pooling. The SW overhead needed to start this type of data transfer is relatively low.

- C++ project: **fp03x8_v26_4x2_md45_dma_sw** (4x SW Lattice, double precision)
  Shared library: **./Debug/libfp03x8_v26_4x2_md45_dma_hw.so**
  Shared library: **./Release/libfp03x8_v26_4x2_md45_dma_hw.so**
  C++ project: **fp03x8_v26_4x2_md45f_dma_sw** (4x SW Lattice, single precision)
  Shared library: **./Debug/libfp03x8_v26_4x2_md45f_dma_hw.so**
  Shared library: **./Release/libfp03x8_v26_4x2_md45f_dma_hw.so**

  Debug and Release version of shared libraries Debian Stretch 9.8 OS for the ZU09EG-ES1 device for the SDK 2017.4 C++ SW flow (g++ compiler). PL is working with 4x2 evaluation versions of FP03x8 HW accelerators. It is using DMA HW data movers. The HW supported data transfers require data to be present in “sd_alloc” memory (continuous physical section reserved in the DDR4). The end of data transfer is tested by pooling. The SW overhead needed to start this data transfer is larger in comparison to the Zero Copy data mover.

- C++ project: **fp03x8_v26_4x2_md45_zc_sg_sw** (4x SW Lattice, double precision)
  Shared library: **./Debug/libfp03x8_v26_4x2_md45_zc_sg_hw.so**
  Shared library: **./Release/libfp03x8_v26_4x2_md45_zc_sg_hw.so**
  C++ project: **fp03x8_v26_4x2_md45f_zc_sg_sw** (4x SW Lattice, single precision)
  Shared library: **./Debug/libfp03x8_v26_4x2_md45f_zc_sg_hw.so**
  Shared library: **./Release/libfp03x8_v26_4x2_md45f_zc_sg_hw.so**

  Debug and Release version of shared libraries Debian Stretch 9.8 OS for the ZU09EG-ES1 device for the SDK 2017.4 C++ SW flow (g++ compiler). PL is working with 4x2 evaluation versions of FP03x8 HW accelerators. It is using Zero copy data movers for write to HW and SG data movers for read from HW with interrupts. The HW supported data transfer requires data to be present in “sd_alloc” memory (continuous physical section reserved in the DDR4). The end of data transfer is indicated by HW interrupt created by the SG data mover HW. The SW overhead needed to start this data transfer is larger in comparison to the DMA data mover due to larger complexity of the SG DMA HW data mover, but the Arm A53 load is reduced due to the interrupt based end of operation which does not require SW pooling.

6 License

This evaluation package includes **evaluation versions** of accelerator:

- **FP03x8** with **capabilities = 40** described in Figure 4 and Figure 5**.**

The license for the evaluation versions of accelerators enables execution of certain large number of floating point operations before it expires. If this happens, the board has to be switched off and switched on again to restart the evaluation license again.

The evaluation versions of accelerators can be publicly downloaded for free from UTIA www page [2]: http://sp.utia.cz/index.php?ids=projects/fitoptivis

The commercial version of accelerators is available in UTIA. UTIA offers this license on commercial base. Contract with UTIA is required. For information about details of the commercial license write to Jiri Kadlec kadlec@utia.cas.cz.

7 Conclusion

The run-time reconfigurable floating point accelerators for the ZU09-EG-ES1device have been designed and realized with respect to the following considerations and requirements:

1. Software utilizing the accelerator can be developed also directly on the embedded system, using the C compiler (gcc) or C++ compiler (g++) present in the Debian Stretch 9.8 operating system running on the Arm A53 device.
2. The entire HW platform with 4x2 FP32x8 SIMD HW accelerators is provided in form of a shared library. The provided shared library API is compatible with the standard gcc and g++ based compilation flows. Scripts are auto generated for the standard Debian OS “make” of the embedded system.
3. The 4x2 FP32x8 SIMD HW hardware of floating point accelerators is fixed. Reconfiguration is performed by reprogramming the firmware code. The firmware defines what sequences of operations will do the programmable finite state machine (FSM) inside the accelerator.
4. Data communication is implemented as an AXI-stream and supports accelerator chaining. The 4x2 FP32x8 SIMD HW hardware configurations are provided.
5. The data communication support HW data movers are defined in design time and cannot be changed during the run time. The following variants are prepared:
    a. Zero copy (ZC) HW data movers with C interface, (minimal HW resources)
    b. Zero copy (ZC) HW data movers with C++ interface (minimal HW resources)
    c. DMA data HW data movers with C++ interface
    d. Combination of ZC HW (DDR to Accelerator) and SG DMA HW (Accelerator to DDR) with interrupts and C++ interface.

   All communication alternatives work with identical SW API. It means that the user host SW code for ARM A53 remains identical and does not need modifications for all four versions of HW data movers.
6. Software must be able to query and identify which SIMD FP operations are supported by each HW accelerator. Based on this information, the software can be reconfigured to take the advantage of supported operations.
7. The accelerator must be able to query and identify information about the actual status of the HW license defined in with each HW accelerator.
8. The HW accelerator scheduler executing the sequence of VLIW operation is very simple. It can execute only a linear sequence of VLIW vector instructions. It does not support *for-loops*, *if-else*, and similar constructs. There is also no support for checking for the overflow/underflow or NaN in performed floating point operations. All these program control constructs have to be implemented in the host code running on the ARM A53 processor.
9. Computations performed in HW accelerators can overlap with stream-based data communications. This is controlled by the user host software running on the ARM A53 processor, usually in several parallel executed threads.
10. Data are stored as 64 bit words. This arrangement enables potential use the Ultra RAM blocks (4096x64b) present in some larger Zynq UltraScale+ devices without affecting the accelerator library API or user code.

Reconfiguration of accelerator by change of firmware

The FP32x8 HW accelerator executes sequences of VLIW vector instructions (firmware) stored in accelerator program memory. This firmware can be first defined in the Arm host

software and then downloaded via the streaming interface to the accelerator. The program memory will usually contain multiple different sequences of VLIW instructions.

Computation performed in the accelerator can overlap with stream-based data communication. This is controlled by the Arm host software and it can be used for run-time reconfiguration by loading a new VLIW instruction sequence to the accelerator program memory while computation is in progress.

For example, consider an application which needs to perform accelerated multiplication of 64x64 matrices (Z[64,64] = A[64,64] × B[64,64]). The application running on the host will split the matrix operation into shorter sequences of VLIW instructions and loaded instruction sequences into the accelerator program memory schedule scheduled by the application software running on the ARM host by adjusting pointers to instruction sequences to be loaded into the accelerator program memory while streaming parts of matrix B[64,64] from host DDR memory to the accelerator. Rows of the matrix are propagated as identical to all 8xSIMD memories in 8 stages.

Reconfiguration of accelerator by temporary change of firmware

Application software can temporarily reconfigure the accelerator in the following steps:

1. Save pars of data and firmware from accelerator to DDR4,
2. Change firmware and upload it to the accelerator,
3. Execute the firmware (for example the **SupOp** instruction)
4. Read the results from accelerator data memory into ARM host memory,
5. Restore saved data and firmware back from DDR to accelerator.

After performing the above steps, the accelerator data and firmware are back in its original state and can continue. The application software running on the ARM host has information about the supported SIMD operations as well as about the status of the HW license.

This temporary replacement of firmware and data can be re-used and work independently on the actual content of the accelerator.

Consider a scenario in which the host application software needs to find out if needed VLIW instructions and the corresponding SIMD operation is actually supported by the HW accelerator.

This information is required by the host software to decide, which firmware version can be used for programming of the HW accelerator:

- If the **DotProd** instruction is supported by the HW accelerator, the computation of 64x64 matrix multiplication (Z[64,64] = A[64,64] × B[64,64]) will use the instruction to improve efficiency.
- If the **DotProd** instruction is unsupported by the HW accelerator, the host software running on the ARM processor can implement the accelerated matrix multiplication using different sequences of **Mac** (multiply and accumulate) VLIW instructions.
- If the **Mac** instruction is also unsupported, the matrix multiplication can be implemented by again different sequence of **Add** and **Mult** VLIW instructions.

The performance of the matrix multiplication might be reduced in the last case, but such HW accelerator requires less HW resources in the programmable logic of the device.

Use of such compact HW accelerators with reduced set of VLIW instructions might be necessary if the programmable logic area is limited. See Figure 8 for available HW accelerator versions.

8 References

[1] The module supported by this application note (TE0808-ES1 with Xilinx device xczu9eg-ffvc900-1-i-es1 and 2GB DDR4) is not in production at present.
This module was available as one of the first widely used Zynq Ultrascale+ industrial module, mainly in the time period from 2016 and 2017.

Similar, currently available products:
Trenz Electronic, UltraSOM+ MPSoC Module with Zynq UltraScale+ ZU9EG-1FFVC900E, 4 GByte DDR4, module with order number: TE0808-04-9BE21-A

https://shop.trenz-electronic.de/en/TE0808-04-9BE21-A-UltraSOM-MPSoC-Module-with-Zynq-UltraScale-ZU9EG-1FFVC900E-4-GByte-DDR4

or module with order number: TE0808-05-9BE21-A

https://shop.trenz-electronic.de/en/TE0808-05-9BE21-A-UltraSOM-MPSoC-Module-with-Zynq-UltraScale-XCZU9EG-1FFVC900E-4-GByte-DDR4

On request, UTIA can recompile HW systems for TE0808-ES1 device presented in this application note also for these currently supported modules.
Designs for these two modules would require Xilinx Vivado and SDK 2018.2 tools.

[2] Trenz Electronic, UltraITX+ Baseboard for Trenz Electronic TE080X UltraSOM+
https://shop.trenz-electronic.de/en/TEBF0808-04A-UltraITX-Baseboard-for-Trenz-Electronic-TE080X-UltraSOM?c=261

Disclaimer

This disclaimer is not a license and does not grant any rights to the materials distributed herewith. Except as otherwise provided in a valid license issued to you by UTIA AV CR v.v.i., and to the maximum extent permitted by applicable law:

(1) THIS APPLICATION NOTE AND RELATED MATERIALS LISTED IN THIS PACKAGE CONTENT ARE MADE AVAILABLE "AS IS" AND WITH ALL FAULTS, AND UTIA AV CR V.V.I. HEREBY DISCLAIMS ALL WARRANTIES AND CONDITIONS, EXPRESS, IMPLIED, OR STATUTORY, INCLUDING BUT NOT LIMITED TO WARRANTIES OF MERCHANTABILITY, NON-INFRINGEMENT, OR FITNESS FOR ANY PARTICULAR PURPOSE; and

(2) UTIA AV CR v.v.i. shall not be liable (whether in contract or tort, including negligence, or under any other theory of liability) for any loss or damage of any kind or nature related to, arising under or in connection with these materials, including for any direct, or any indirect, special, incidental, or consequential loss or damage (including loss of data, profits, goodwill, or any type of loss or damage suffered as a result of any action brought by a third party) even if such damage or loss was reasonably foreseeable or UTIA AV CR v.v.i. had been advised of the possibility of the same.

Critical Applications:
UTIA AV CR v.v.i. products are not designed or intended to be fail-safe, or for use in any application requiring fail-safe performance, such as life-support or safety devices or systems, Class III medical devices, nuclear facilities, applications related to the deployment of airbags, or any other applications that could lead to death, personal injury, or severe property or environmental damage (individually and collectively, "Critical Applications"). Customer assumes the sole risk and liability of any use of UTIA AV CR v.v.i. products in Critical Applications, subject only to applicable laws and regulations governing limitations on product liability.