FP01x8 Accelerator on TE0723-03M

Jiři Kadlec, Zdeněk Pohl, Lukáš Kohout
kadlec@utia.cas.cz *xpohl@utia.cas.cz* *kohoutl@utia.cas.cz*

Revision history

Rev.	Date	Author	Description
0	1.12.2019	J. Kadlec	Initial publication
1	31.5.2020	J. Kadlec	Update
2			

WAKeMeUP has been accepted for funding within the Electronic Components and Systems For European Leadership Joint Undertaking in collaboration with the European Union's H2020 Framework Programme (H2020/2014-2020) and National/Local Authorities, under grant agreement n° 783176

Table of Contents

Table of Figures

Acknowledgement

This work has been partially supported from project WakeMeUp, project number ECSEL 783176 and the corresponding Czech NFA (MSMT) institutional support project 8A18001.

1 Evaluation version of 8xSIMD FP01x8 accelerator for ArduZynq shield

This application note describes released UTIA support for the evaluation version of 8xSIMD FP01x8 floating-point, run-time-reconfigurable accelerator for the ArduZynq shield.

ArduZynq shield TE0723-03M [1] works with Xilinx XC07010-1C device with the dual core Arm A9 32 bit and relatively small programmable logic area on single 28 nm chip.

The ArduZynq shield PCB has Arduino compatible form factor with connectors physically compatible with the STM32 Nucleo 144 boards.

The ArduZynq shield is designed and manufactured by the company Trenz Electronic [1].

SW developer can program application without SDSoC 2018.2 compiler license.

The standard g++ compiler and „make“ can be used.

The accelerator and HW data communication is represented for the SW developer as shared C++ library with simple SW API, identical for several alternative HW data movers.

See next chapters of this application note for overview of internal details of the FP01x8 accelerator and for overview of SW API details and descriptions.

1.1 Confidence test

This is basic confidence test of the evaluation package.

INSTALLATION OF PC TOOLS and DEBIAN OS for ARM A9

- Install Xilinx SDK 2018.2 on Win 10 PC.
- Install Xilinx Lab Tools 2018.2 on Win 10 PC.
- Install Win32DiskImager (for writing of image to 8 GByte Micro SD card)
- Install Putty (for USB based serial console and Ethernet based serial console)
- Unzip disk image (cca 8 GByte)
- Write disk image to micro SD card

HW SETUP

- Insert micro SD card to the ArduZynq shield mode
- Connect micro USB cable to ArduZynq shield and to PC

TEST

- ArduZynq shield will start to boot OS
- Open Putty terminal (115200 bps, 8 data bits, stop bit 1, parity none, flow control off)
- Use Putty terminal to login as user: **`root`** password: **`root`**
- Change directory to **`/boot`**
- Connect to the shared library: **`export LD_DATA_PATH=/boot`**
- Start application code by typing: **`fp01x8_v26x1_dma_sw.elf`**

RESULT

- The application will compute floating point single precision matrix multiplication on ARM A9 and on the 8xSIMD FP01x8 accelerator.
- The results are compared to be identical and the performance acceleration is measured. The accelerator is in the range 50x to 60x comparing to the ARM SW. The

acceleration range is high as the ARM is compiled for Debug (SW is compiled with –O0). In case of compilation with maximal SW acceleration (–O3), the acceleration is in the range 5x to 6x.

1.2 Compilation and debug of projects from source code

The evaluation package includes four SW projects for Xilinx SDK 2018.2. These projects can be modified and recompiled for ARM and executed on ArduZynq shield with or without debugging support.

In Xilinx SDK 2018.2, select one of the projects present in the package:

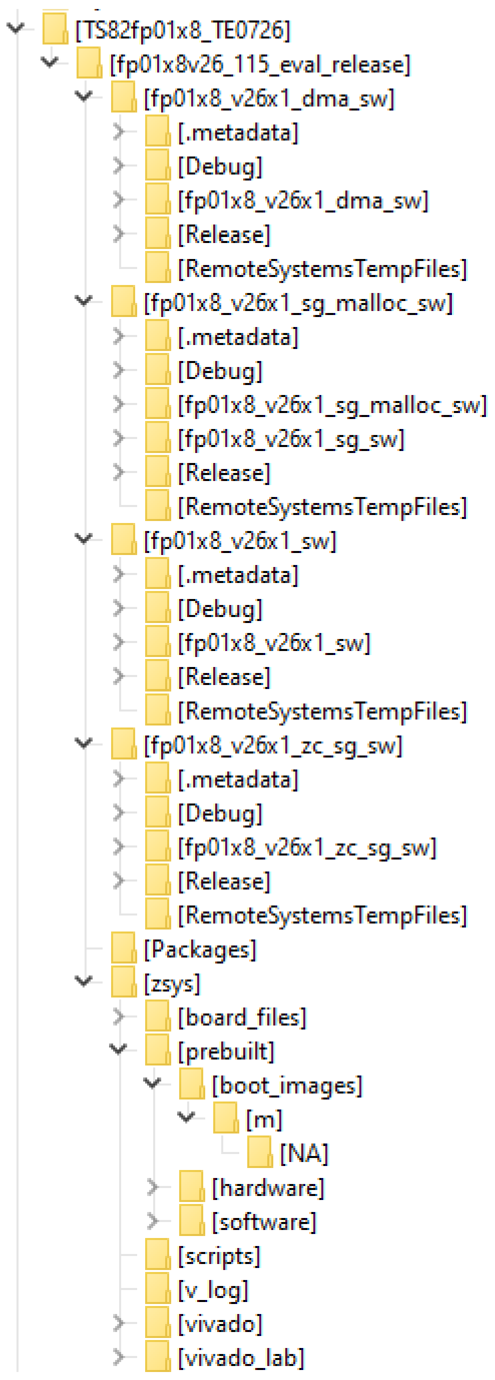
- fp01x8v26x1_dma_sw
- fp01x8v26x1_sg_malloc_sw
- fp01x8v26x1_dma_sw
- fp01x8v26x1_zc_sg_sw

Each project has two configurations:

- Debug for debugging with –O0 flag
- Release for maximal performance with-O3 flag and without debug symbols

You can modify and compile the SW code.
Each project comes with (.so) library precompiled for debug (-O0) or for Release (-O3).
These libraries contain SW representation of the evaluated accelerator with given type of data mover HW IPs instantiated in the PL logic of the device.
Before test on the ArduZynq shield, you have to write to the on-board FLASH the correct **`BOOT.BIN`** file which includes inside the correct bitstream with evaluated accelerator and data mover HW IPs.

It is done by performing two steps. First, copy of the associated **`BOOT.BIN`** from (for example) **`fp01x8v26x1_dma_sw/Debug/sd_card/BOOT.BIN`** to **`zynq/prebuilt/boot_images/m/NA/BOOT.BIN`**
Second, change directory to **`zynq/prebuilt/boot_images/m/NA`**
and execute in PC terminal the script:
`program_flash_binfile.cmd`
This script will write the selected **`BOOT.BIN`** to the ArduZynq shield and the board will start to boot Petalinux kernel and Debian from the micro SD card with correct bitstream in the PL logic.



```
[TS82fp01x8_TE0726]
  [fp01x8v26_115_eval_release]
    [fp01x8_v26x1_dma_sw]
      [.metadata]
      [Debug]
      [fp01x8_v26x1_dma_sw]
      [Release]
      [RemoteSystemsTempFiles]
    [fp01x8_v26x1_sg_malloc_sw]
      [.metadata]
      [Debug]
      [fp01x8_v26x1_sg_malloc_sw]
      [fp01x8_v26x1_sg_sw]
      [Release]
      [RemoteSystemsTempFiles]
    [fp01x8_v26x1_sw]
      [.metadata]
      [Debug]
      [fp01x8_v26x1_sw]
      [Release]
      [RemoteSystemsTempFiles]
    [fp01x8_v26x1_zc_sg_sw]
      [.metadata]
      [Debug]
      [fp01x8_v26x1_zc_sg_sw]
      [Release]
      [RemoteSystemsTempFiles]
    [Packages]
    [zsys]
      [board_files]
      [prebuilt]
        [boot_images]
          [m]
            [NA]
        [hardware]
        [software]
      [scripts]
      [v_log]
      [vivado]
      [vivado_lab]
```

1.3 Compile SW application directly on the ArduZynq shield

Xilinx SDK creates make which can be used for compilation of SW application directly on the board with use of the g++ compiler of the Debian OS support for FP01x8 Accelerator on ArduZynq shield TE0723

1.4 Precompiled shared libraries with different data mover HW IPs

The PL part of the ArduZynq shield contains one evaluation version of the 8xSIMD run-time reprogrammable single precision HW accelerator FP01x8. The 8xSIMD FP01x8 accelerator is run-time reprogrammable. Its functionality, performance and re-programmability is demonstrated by four SW demos. Each demo performs single precision floating point matrix-by-matrix multiplication

C[64,64] = A[64,64] * B[64,64]

Table 1: Shared Libraries represent FP01x8 accelerator HW with different HW data movers

Zynq 7000 (ArduZynq shield) TE0723
Single 8xSIMD FP01 Accelerator, no div op
libfp01x8_v26x1_dma_hw.so
libfp01x8_v26x1_sg_malloc_hw.so
libfp01x8_v26x1_hw.so
libfp01x8_v26x1_zc_sg_hw.so

The released design time support for the FP01x8 accelerator provides for the SW designer set of precompiled shared libraries representing the HW platform. See *Table 1*:

The libraries represent different data movers used for connection of the 8xSIMD run-time reprogrammable single precision HW accelerator FP01x8 in these HW configurations:

- **libfp01x8_v26x1_hw.so** is using Zero Copy HW data movers. It is not using the DMA IP cores. The data movers are realized as C++ function compiled to HW by the SDSoC 2018.2 compiler. The HW supported data transfer requires data to be present in “sd_alloc” memory (continuous physical section reserved in the DDR3). Start of the data transfer is no blocking. The end of data transfer is tested by pooling. The SW overhead needed to start this data transfer is minimal.

- **libfp01x8_v26x1_dma_hw.so** 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 DDR3). Start of the data transfer is no blocking. 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.

- **libfp01x8_v26x1_sg_malloc_hw.so** is using combination of Zero Copy HW data mover and DMA SG HW data mover with interrupt. The HW supported data allocated by “sd_alloc” memory (continuous physical section reserved in the DDR3). Start of the data transfer is no blocking. The end of data transfer is tested by interrupts. The SW overhead needed to start this data transfer is larger in comparison to the DMA data mover. Data can be allocated in the standard Linux user-space memory, allocated by the standard Linux “malloc” function. This is the only HW implementation capable to work directly with standard “malloc” allocated linux data.
    - If “malloc” data allocation is used, the overhead of this SG DMA is really large.
    - If “sd_alloc” data allocation is used (continuous physical section reserved in the DDR3), the overhead of this SG DMA is larger in comparison to DMA based HW support, but much shorter in comparison to the case of data allocation based on standard “malloc”.

- **libfp01x8_v26x1_zc_sg_hw.so** is using DMA SG data mover with “sd_alloc” allocated data and interrupts. Start of the data transfer is no blocking. The end of data transfer is based on interrupt. The SG FMA is using the advanced coherent port of the Zynq device. There is no need to flush the Zynq cache before accessing of data.

Table 2: HW resources used by the FP01x8 Accelerator with different HW data movers

Device: 7z010clg225-1	**lut**	**reg**	**bram**	**dsp**
Available (100%)	12462	15376	60	80
One FP01X8_v26_40 Accelerator				
fp03x8_v26x2_dma_hw	74,06%	46,42%	69,17%	40,00%
fp03x8_v26x2_hw	69,43%	41,80%	63,33%	40,00%
fp03x8_v26x2_sg_malloc_hw	87,36%	58,42%	77,50%	40,00%
fp03x8_v26x2_zc_sg_hw	85,72%	56,63%	75,00%	40,00%

.

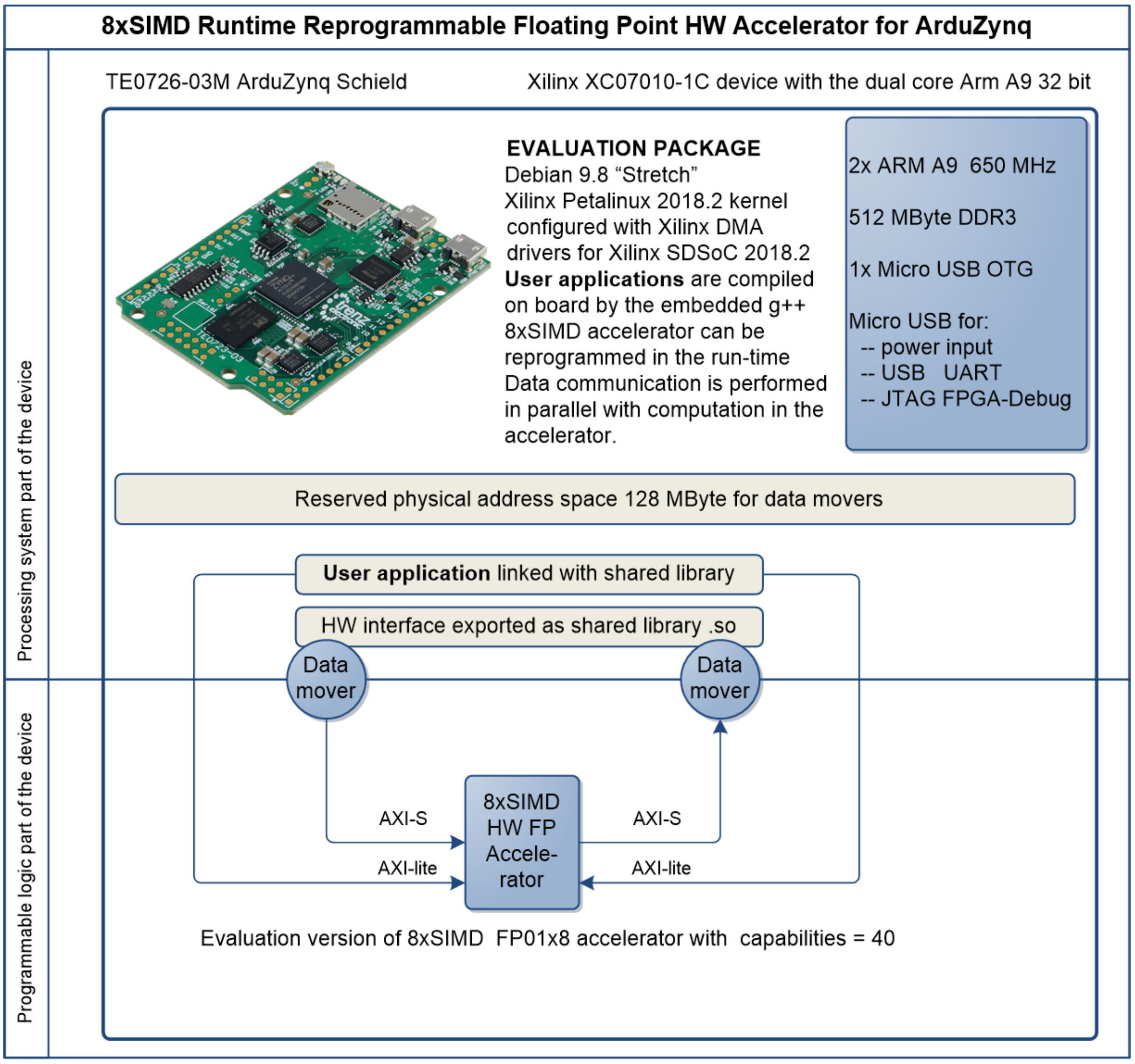

8xSIMD Runtime Reprogrammable Floating Point HW Accelerator for ArduZynq
TE0726-03M ArduZynq Schield
Xilinx XC07010-1C device with the dual core Arm A9 32 bit
EVALUATION PACKAGE
Debian 9.8 "Stretch"
Xilinx Petalinux 2018.2 kernel configured with Xilinx DMA drivers for Xilinx SDSoC 2018.2
User applications are compiled on board by the embedded g++
8xSIMD accelerator can be reprogrammed in the run-time
Data communication is performed in parallel with computation in the accelerator.
2x ARM A9 650 MHz
512 MByte DDR3
1x Micro USB OTG
Micro USB for:
-- power input
-- USB UART
-- JTAG FPGA-Debug
Reserved physical address space 128 MByte for data movers
User application linked with shared library
HW interface exported as shared library .so
Data mover
Data mover
AXI-S
AXI-S
AXI-lite
AXI-lite
8xSIMD HW FP Accele-rator
Evaluation version of 8xSIMD FP01x8 accelerator with capabilities = 40
Processing system part of the device
Programmable logic part of the device

Figure 1: ArduZynq shield with evaluation version of FP01x8 accelerator

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:

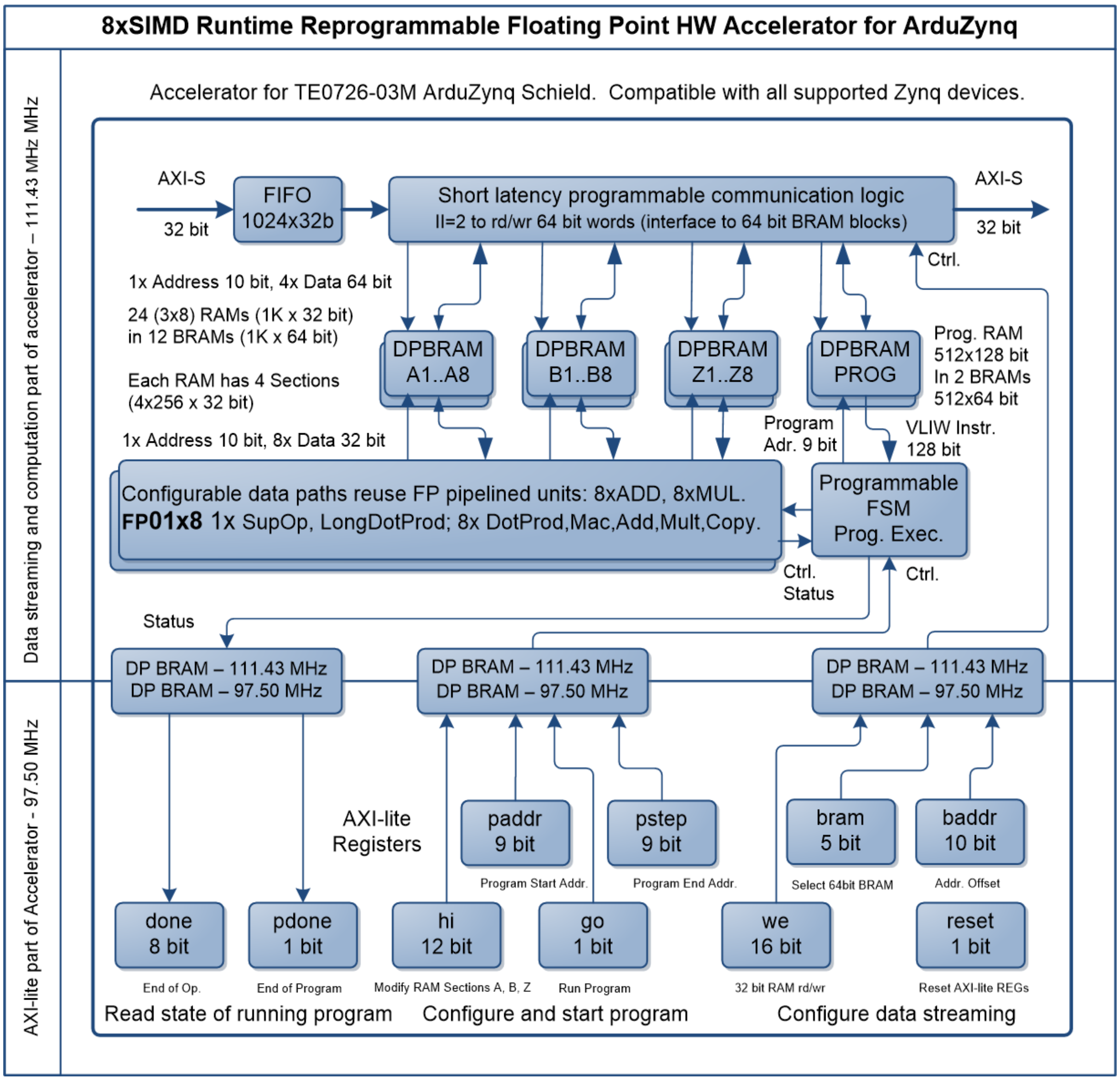
- Registers indicating end of program accessible to Arm processor via AXI-lite.
- Floating point single precision result data accessible via AXI stream interface for the Arm processor.

Connectivity:

- AXI stream input with input FIFO 2048x32 and support for the AXI stream side channel indicating the last transferred word sent to the component via the DMA transaction from Arm processor.
- AXI stream output with output FIFO 2048x32 bit with support for the output side channel indicating the last transferred word sent from the component via the DMA to Arm processor.
- AXI-lite input/output configuration registers.

Target:

- Zynq devices with PL part and processor on single chip.



8xSIMD Runtime Reprogrammable Floating Point HW Accelerator for ArduZynq
Accelerator for TE0726-03M ArduZynq Schield. Compatible with all supported Zynq devices.
Data streaming and computation part of accelerator – 111.43 MHz MHz
AXI-lite part of Accelerator - 97.50 MHz
AXI-S
32 bit
FIFO 1024x32b
Short latency programmable communication logic
II=2 to rd/wr 64 bit words (interface to 64 bit BRAM blocks)
AXI-S
32 bit
Ctrl.
1x Address 10 bit, 4x Data 64 bit
24 (3x8) RAMs (1K x 32 bit) in 12 BRAMs (1K x 64 bit)
Each RAM has 4 Sections (4x256 x 32 bit)
1x Address 10 bit, 8x Data 32 bit
DPBRAM A1..A8
DPBRAM B1..B8
DPBRAM Z1..Z8
DPBRAM PROG
Prog. RAM 512x128 bit In 2 BRAMs 512x64 bit
Program Adr. 9 bit
VLIW Instr. 128 bit
Configurable data paths reuse FP pipelined units: 8xADD, 8xMUL.
FP01x8 1x SupOp, LongDotProd; 8x DotProd,Mac,Add,Mult,Copy.
Programmable FSM Prog. Exec.
Ctrl. Status
Ctrl.
Status
DP BRAM – 111.43 MHz DP BRAM – 97.50 MHz
DP BRAM – 111.43 MHz DP BRAM – 97.50 MHz
DP BRAM – 111.43 MHz DP BRAM – 97.50 MHz
AXI-lite Registers
paddr 9 bit
pstep 9 bit
bram 5 bit
baddr 10 bit
Program Start Addr.
Program End Addr.
Select 64bit BRAM
Addr. Offset
done 8 bit
pdone 1 bit
hi 12 bit
go 1 bit
we 16 bit
reset 1 bit
End of Op.
End of Program
Modify RAM Sections A, B, Z
Run Program
32 bit RAM rd/wr
Reset AXI-lite REGs
Read state of running program
Configure and start program
Configure data streaming

Figure 2: Architecture 8xSIMD FP01x8 accelerator

1.5 Commercial Positioning

The commercial version of accelerators is available starting from 20.4.2020. UTIA will offer this license on commercial base. Contract has to be signed with UTIA.

For information about details of the commercial license write to Jiri Kadlec kadlec@utia.cas.cz.

2 8xSIMD FP01x8 floating point accelerator

Component serves for run-time reprogrammable 8xSIMD single precision floating point computation.

Versions of accelerators:

- fp01x8_capabilities capabilities= 10, 20, 30 or 40
- Zynq 7000 family of devices

Table 3: Floating point functions present in all accelerators {10 or 20 or 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;}*

Accelerators fp01x8 with all capabilities do not support 8xSIMD floating point division.
Accelerators fp03x8 with all capabilities support 8xSIMD floating point division.

Table 4: Floating point functions in accelerators with the capabilities {10, 20, 30, 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;}*

Table 5: Structure of the 128 bit wide VLIW program instruction.

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

Accelerator Interfaces

Type of interface	Device	Clock
• Data streaming I/O: AXI-S 32 bit	Zynq	111.43 MHz
• Computation: 8xSIMD FP32	Zynq	111.43 MHz
• Firmware program VLIW 128 bit	Zynq	111.43 MHz
• Configuration I/O: AXI-lite 32 bit	Zynq	97.50 MHz

Design-time support

The design time support (WP3) provides design flow for automated generation of data streaming HW (data movers):

• Zero Copy	HW data mover without DMA unit
• DMA	HW data mover with DMA unit
• SG DMA	HW data mover with interrupts

This design time support is based on the Xilinx SDSoC 2018.2 system level compiler.

Run-time support

- Firmware is re-programable in run-time by data streaming.
- Computation & data streaming can be performed in parallel. See WP4 run-time support.

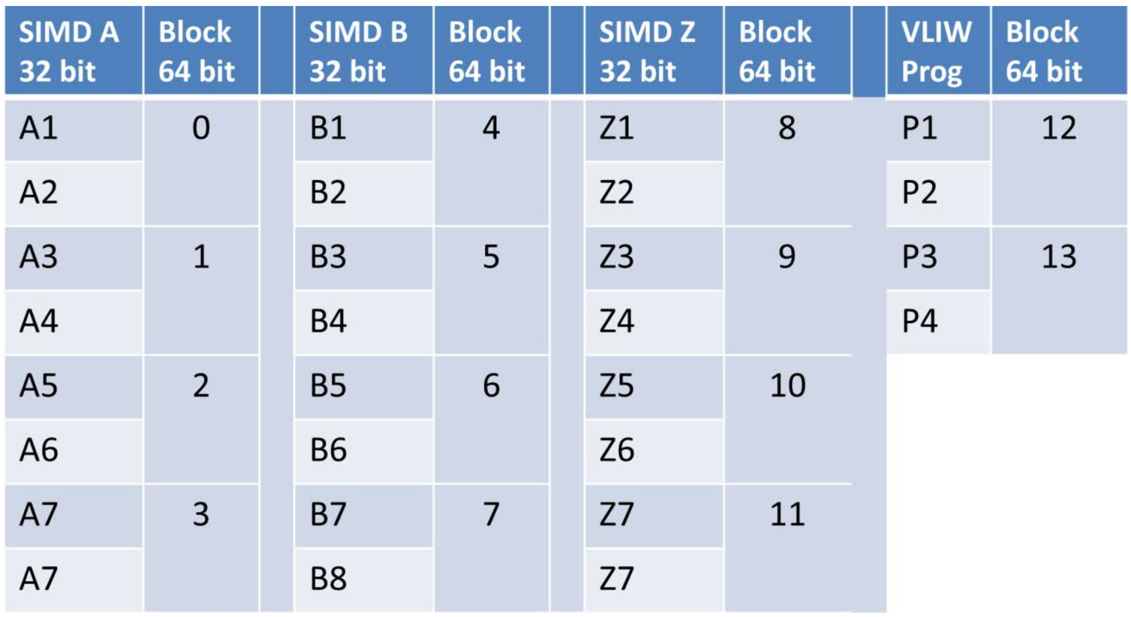
AXI-lite Registers Controlled by Arm app.

reset	1 bit:	"1" Reset AXI lite Registers; "0" NOP
we	16 bit:	Write from stream to blocks 0 .. 13
baddr	10 bit:	Stream will rd/wr from addr=baddr
bram	5 bit:	Read from Block 0 .. 13 to stream
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

Memory of the Accelerator

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

Table 6: Internal block rams of accelerators.



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		
A7		B8		Z7			

Stream Data from/to ARM DDR memory

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

3 Arm SW API for Streaming of Data

len = number of 32 bit words in the Stream
Calls are unblocking, trigger HW threads.
HW threads synchronize with SW in blocking `sds_wait()` Calls are similar to the pthread `barrier()`.

Serial Streaming SW API

API for single accelerator and for single serial chain of multiple chained accelerators
void data2hw_wrapper(unsigned *src, unsigned len); //1
void capture_wrapper(unsigned *storage, unsigned len); //2

Example:
```
data2hw_wrapper((unsigned*)src_P1_P2, len);      //1
capture_wrapper((unsigned*)dest_P1_P2, len);     //2
…
sds_wait(1);
sds_wait(2);
```

4 Performance

Acceleration of single precision floating point Matrix by Matrix multiplication has been prepared as an application example to evaluate the performance of released evaluation versions of accelerators. It performs: C[64,64] = A[64,64] * B[64,64].

A single instance of FP01x8 accelerator on ArduZynq shield accelerates A9 processor (with 650 MHz clock) computation of this single precision floating point Matrix by Matrix multiplication by factor of **5x**.

5 Intellectual Property information

This evaluation package includes one **evaluation versions** of accelerators:

- fp01x8_capabilities capabilities = 40
- Zynq 7000 family of devices with evaluation package for ArduZynq shield.

6 License:

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.

Starting from 20.4.2020, the evaluation versions of accelerators can be publicly downloaded for free from UTIA www page: http://sp.utia.cz/index.php?ids=projects/fitoptivis

The commercial version of accelerators is available starting from 1.4.2020. UTIA will offer this license on commercial base. Contract has to be signed with UTIA.
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 Zynq platforms 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 board, using the C++ compiler (g++) present in the Debian OS and Xilinx data-mover support drivers.
2. The entire HW platform with one evaluation version of accelerator, is provided in form of a shared library. The provided library API is compatible with C++ development practice and standard "make" can be used to build the user application.
3. The hardware of the floating point accelerators is fixed. Reconfiguration is performed by reprogramming the firmware code which defines the function of the programmable finite state machine (FSM) inside the accelerator and the function of the communication logic.
4. Data communication is implemented as an AXI-stream and supports accelerator chaining.
5. The data communication support HW is determined at design time and cannot be changed at runtime. The following variants can be generated:
    a. Zero copy (ZC) HW data movers consuming minimal HW resources,
    b. DMA data HW data movers,
    c. Scatter gather (SG) DMA data movers with interrupts,

   d. Combination of ZC HW (DDR to Accelerator) and SG DMA HW (Accelerator to DDR)
6. All communication alternatives have to work with identical SW API. It means that the user SW code remains identical and does not need modifications at run-time.
7. Software must be able to query the list of SIMD FP operations supported by the accelerator. Based on this information, the software can be reconfigured to take advantage of supported operations.
8. The accelerator must be able to provide information on whether the HW license coming with the accelerator is valid.
9. The accelerator firmware is a simple sequence of VLIW vector instructions which support *for-loops*, *if-else*, and similar constructs. However, there is no support for checking overflow/underflow in floating point operations. Such constructs have to be implemented in the host code (executing on ARM core).
10. Computation performed in the accelerator can overlap with stream-based data communication. This is controlled by the user-space host software running on the ARM core.
11. Data are stored in 64bit-wide dual-ported blocks. This arrangement enables to use the Ultra RAM blocks (4096x64b) present in some larger Zynq UltraScale+ devices without affecting the accelerator library API or user code.

Reconfiguration by change of firmware

The accelerator executes sequences of VLIW vector instructions (firmware) stored in accelerator program memory. This firmware can be first defined in the 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 host software running on the ARM core 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 subsequent stages.

Reconfiguration by temporary change of firmware

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

1. Save data and firmware from accelerator to DDR,
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 data and firmware from DDR.

After performing the above steps, the accelerator data and firmware is back in its original state and 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.

Consider a scenario in which the application software needs to find out which SIMD operations are actually supported by the accelerator. This information is required to determine, e.g., which firmware version can be used with the accelerator. If the **DotProd** instruction is supported by the accelerator, the accelerated computation of 64x64 matrix multiplication (Z[64,64] = A[64,64] × B[64,64]) will use the instruction to improve efficiency.

Alternatively, if the **DotProd** instruction is unsupported, the application software running on the ARM host can implement an accelerated matrix multiplication using sequences of **Mac** (multiply and accumulate) instructions.

If the **Mac** instruction is also unsupported, the matrix multiplication can be implemented using **Add** and **Mult** instructions. The performance of the matrix multiplication will be reduced by approximately 50%, but the accelerator will require less HW resources to implement. This might be necessary for some platform configurations where the programmable logic area is used by pre-defined HW accelerated video processing.

References

[1] Trenz Electronic, "TE0723 TRM," [Online].
ArduZynq - Arduino compatible Xilinx Zynq-7010 FPGA module
https://shop.trenz-electronic.de/en/TE0723-03M-ArduZynq-Arduino-compatible-Xilinx-Zynq-7010-FPGA-module?c=349

[2] WAKeMeUP Wafers for Automotive and other Key applications using Memories, embedded in Ulsi Processors, UTIA www page.
http://sp.utia.cz/index.php?ids=projects/wakemeup