
**Information technology — Coding of
audio-visual objects —**

**Part 9:
Reference hardware description**

*Technologies de l'information — Codage des objets audiovisuels —
Partie 9: Description de matériel de référence*



Reference number
ISO/IEC TR 14496-9:2009(E)

AVA Documents
Document Sharing Hub
© ISO/IEC 2009

PDF disclaimer

This PDF file may contain embedded typefaces. In accordance with Adobe's licensing policy, this file may be printed or viewed but shall not be edited unless the typefaces which are embedded are licensed to and installed on the computer performing the editing. In downloading this file, parties accept therein the responsibility of not infringing Adobe's licensing policy. The ISO Central Secretariat accepts no liability in this area.

Adobe is a trademark of Adobe Systems Incorporated.

Details of the software products used to create this PDF file can be found in the General Info relative to the file; the PDF-creation parameters were optimized for printing. Every care has been taken to ensure that the file is suitable for use by ISO member bodies. In the unlikely event that a problem relating to it is found, please inform the Central Secretariat at the address given below.



COPYRIGHT PROTECTED DOCUMENT

© ISO/IEC 2009

All rights reserved. Unless otherwise specified, no part of this publication may be reproduced or utilized in any form or by any means, electronic or mechanical, including photocopying and microfilm, without permission in writing from either ISO at the address below or ISO's member body in the country of the requester.

ISO copyright office
Case postale 56 • CH-1211 Geneva 20
Tel. + 41 22 749 01 11
Fax + 41 22 749 09 47
E-mail copyright@iso.org
Web www.iso.org

Published in Switzerland

Contents

Page

Foreword.....	v
Introduction	vii
1 Scope	1
2 Copyright disclaimer for HDL software modules	1
3 Abbreviated terms	2
4 HDL software availability	2
5 HDL coding format and standards	2
5.1 HDL standards and libraries	2
5.2 Conditions and tools for the synthesis of HDL modules	3
5.3 Conformance with the reference software	3
6 Integrated Framework supporting the “Virtual Socket” between HDL modules described in Part 9 and the MPEG Reference Software (Implementation 1).	3
6.1 Introduction	3
6.2 Addressing	4
6.3 Memory Map	4
6.4 Hardware Accelerator Interface	6
6.5 User Hardware Accelerator Sockets	12
7 Integrated Framework supporting the “Virtual Socket” between HDL modules described in Part 9 and the MPEG Reference Software (Implementation 2).	14
7.1 Introduction	14
7.2 Development Example of a Typical Module : Calc_Sum_Product Module	14
7.3 Second Example of a Typical Module : fifo_transfer module	18
7.4 Integrating the Multi-Modules within the Framework	23
7.5 Calc_Sum_Product Module Controller (memory data transfer)	30
7.6 Simulation of the whole system	44
7.7 Debug Menu	46
8 Integrated Framework supporting the “Virtual Socket” between HDL modules described in Part 9 and the MPEG Reference Software (Implementation 3).	47
8.1 An Integrated Virtual Socket Hardware-Accelerated Co-design Platform for MPEG-4	47
8.2 Reference for Virtual Socket API Function Calls	72
8.3 Tutorial on the Integrated Virtual Socket Hardware-Accelerated Co-design Platform for MPEG-4 Part 9 Implementation 3	95
8.4 An Integration of the MPEG-4 Part 10/AVC DCT/Q Hardware Module into the Virtual Socket Co-design Platform	140
8.5 Migrating Virtual Socket Hardware-Accelerated Co-design Platform From WildCard-II to WildCard-4	153
9 Integrated Framework supporting the “Virtual Socket” between HDL modules described in Part 9 and the MPEG Reference Software: Implementation 4 - Virtual Memory Extension	167
9.1 Introduction	167
9.2 Overview of the “Virtual Socket Platform” implementation 4	167
9.3 Development information	171
9.4 Technical details	172
9.5 How to build the platform	196
9.6 Simulation of the platform	203
9.7 Synthesis of the platform	209
9.8 Building the platform system software	212
9.9 How to use the platform	214

9.10	Understanding VHDL code.....	216
9.11	Appendix	220
9.12	Glossary	222
10	HDL MODULES	225
10.1	INVERSE QUANTIZER HARDWARE IP BLOCK FOR MPEG-4 PART 2	225
10.2	2-D IDCT HARDWARE IP BLOCK FOR MPEG-4 PART 2	232
10.3	VLD+IQ+IDCT for MPEG-4 Part 2	242
10.4	A SYSTEM C MODEL FOR 2X2 HADAMARD TRANSFORM AND QUANTIZATION FOR MPEG-4 PART 10	247
10.5	A VHDL HARDWARE BLOCK FOR 2X2 HADAMARD TRANSFORM AND QUANTIZATION WITH APPLICATION TO MPEG-4 PART 10 AVC	256
10.6	A SYSTEMC MODEL FOR 4X4 HADAMARD TRANSFORM AND QUANTIZATION FOR MPEG-4 PART 10	262
10.7	A VHDL HARDWARE IP BLOCK FOR 4X4 HADAMARD TRANSFORM AND QUANTIZATION FOR MPEG-4 PART 10 AVC	269
10.8	A HARDWARE BLOCK FOR THE MPEG-4 PART 10 4X4 DCT-LIKE TRANSFORMATION AND QUANTIZATION	275
10.9	A SYSTEMC MODEL FOR THE MPEG-4 PART 10 4X4 DCT-LIKE TRANSFORMATION AND QUANTIZATION	281
10.10	A 8X8 INTEGER APPROXIMATION DCT TRANSFORMATION AND QUANTIZATION SYSTEMC IP BLOCK FOR MPEG-4 PART 10 AVC	289
10.11	INTEGER APPROXIMATION OF 8X8 DCT TRANSFORMATION AND QUANTIZATION, A HARDWARE IP BLOCK FOR MPEG-4 PART 10 AVC	299
10.12	A VHDL CONTEXT-BASED ADAPTIVE VARIABLE LENGTH CODING (CAVLC) IP BLOCK FOR MPEG-4 PART 10 AVC	306
10.13	A VERILOG HARDWARE IP BLOCK FOR SA-DCT FOR MPEG-4	311
10.14	A VERILOG HARDWARE IP BLOCK FOR SA-IDCT FOR MPEG-4	322
10.15	A VERILOG HARDWARE IP BLOCK FOR 2D-DCT (8X8)	335
10.16	SHAPE CODING BINARY MOTION ESTIMATION HARDWARE ACCELERATION MODULE	344
10.17	A SIMD ARCHITECTURE FOR FULL SEARCH BLOCK MATCHING ALGORITHM	358
10.18	HARDWARE MODULE FOR MOTION ESTIMATION (4xPE)	367
10.19	A IP BLOCK FOR H.264/AVC QUARTER PEL FULL SEARCH VARIABLE BLOCK MOTION ESTIMATION	381
10.20	AN IP BLOCK FOR VARIABLE BLOCK SIZE MOTION ESTIMATION IN H.264/MPEG-4 AVC ...	389
10.21	An IP Block for MPEG-4 Part 10 AVC Deblocking Filter	400
10.22	A HW BLOCK FOR MPEG-4 PART 10 AVC CONTEXT ADAPTIVE VARIABLE LENGTH CODING (CAVLC)	407
10.23	HARDWARE IMPLEMENTATION OF FULL SEARCH MPEG-4 PART 10 AVC MOTION ESTIMATION	421
Annex A	(Informative) Specification of directory structure for reference SW, HDL and documentation files of MPEG-4 Part 9 Reference HW Description	433
A.1	Introduction	433
A.2	Directory Structure of TR SW Modules	433
	CVS Module Name	434
	Integration Framework Version	435
	Reference Software Version and Modifications	435
Annex B	(Informative) Tutorial on Part 9 CVS Client Installation & Operation	436
Annex C	(Informative) Additional utility software	446
Annex D	(Informative) Providers of reference hardware code	447
Bibliography		448

Foreword

ISO (the International Organization for Standardization) and IEC (the International Electrotechnical Commission) form the specialized system for worldwide standardization. National bodies that are members of ISO or IEC participate in the development of International Standards through technical committees established by the respective organization to deal with particular fields of technical activity. ISO and IEC technical committees collaborate in fields of mutual interest. Other international organizations, governmental and non-governmental, in liaison with ISO and IEC, also take part in the work. In the field of information technology, ISO and IEC have established a joint technical committee, ISO/IEC JTC 1.

International Standards are drafted in accordance with the rules given in the ISO/IEC Directives, Part 2.

The main task of the joint technical committee is to prepare International Standards. Draft International Standards adopted by the joint technical committee are circulated to national bodies for voting. Publication as an International Standard requires approval by at least 75 % of the national bodies casting a vote.

In exceptional circumstances, the joint technical committee may propose the publication of a Technical Report of one of the following types:

- type 1, when the required support cannot be obtained for the publication of an International Standard, despite repeated efforts;
- type 2, when the subject is still under technical development or where for any other reason there is the future but not immediate possibility of an agreement on an International Standard;
- type 3, when the joint technical committee has collected data of a different kind from that which is normally published as an International Standard ("state of the art", for example).

Technical Reports of types 1 and 2 are subject to review within three years of publication, to decide whether they can be transformed into International Standards. Technical Reports of type 3 do not necessarily have to be reviewed until the data they provide are considered to be no longer valid or useful.

Attention is drawn to the possibility that some of the elements of this document may be the subject of patent rights. ISO and IEC shall not be held responsible for identifying any or all such patent rights.

ISO/IEC TR 14496-9, which is a Technical Report of type 3, was prepared by Joint Technical Committee ISO/IEC JTC 1, *Information technology*, Subcommittee SC 29, *Coding of audio, picture, multimedia and hypermedia information*.

This third edition cancels and replaces the second edition (ISO/IEC TR 14496-9:2008) which has been technically revised.

ISO/IEC 14496 consists of the following parts, under the general title *Information technology — Coding of audio-visual objects*:

- *Part 1: Systems*
- *Part 2: Visual*
- *Part 3: Audio*
- *Part 4: Conformance testing*
- *Part 5: Reference software*

- *Part 6: Delivery Multimedia Integration Framework (DMIF)*
- *Part 7: Optimized reference software for coding of audio-visual objects* [Technical Report]
- *Part 8: Carriage of ISO/IEC 14496 contents over IP networks*
- *Part 9: Reference hardware description* [Technical Report]
- *Part 10: Advanced Video Coding*
- *Part 11: Scene description and application engine*
- *Part 12: ISO base media file format*
- *Part 13: Intellectual Property Management and Protection (IPMP) extensions*
- *Part 14: MP4 file format*
- *Part 15: Advanced Video Coding (AVC) file format*
- *Part 16: Animation Framework eXtension (AFX)*
- *Part 17: Streaming text format*
- *Part 18: Font compression and streaming*
- *Part 19: Synthesized texture streaming*
- *Part 20: Lightweight Application Scene Representation (LASeR) and Simple Aggregation Format (SAF)*
- *Part 21: MPEG-J Graphical Framework eXtension (GFX)*
- *Part 22: Open Font Format*
- *Part 23: Symbolic Music Representation*
- *Part 24: Audio and systems interaction*
- *Part 25: 3D Graphics Compression Model*

Introduction

The main goal of this Technical Report is to facilitate a more widespread use of the MPEG-4 standard.

Design methodologies of the EDA industry have evolved from schematics to Hardware Description Languages (HDLs) to address the needs of the vast number of gates available on a single device. The increased number of gates allowed more elaborate algorithms to be deployed but also required a shift in design paradigm to handle the complexity created. Through HDLs, more complicated systems could be designed faster through the enabling technology of synthesis of the HDL code towards different silicon technologies where trade offs could be explored. Now the EDA industry again faces challenges where HDLs may not provide the level of abstraction needed for system designers to evaluate system level parameters and complexity issues. There have been a number of tool investigations under way to address this problem. Profiling tools aid in exposing bottlenecks in an abstract way so that early design decisions can be made. C to gates tools allow a C based simulation environment while also enabling direct synthesis to gates for hardware acceleration.

In conclusion, it is the aim of this Technical Report to enable more widespread use of the MPEG-4 standard through reference hardware descriptions and close integration with MPEG-4 Part 7 Optimized Reference Software. Additionally, it is aimed that exposure to such a platform will enable a more systematic way to investigate the complexity of new codecs and open up the algorithm search space with an order of magnitude more compute cycles.

Information technology — Coding of audio-visual objects —

Part 9: Reference hardware description

1 Scope

This part of ISO/IEC 14496 specifies descriptions of the main video coding tools in hardware description language (HDL) form. Such alternative descriptions to the ones that are reported in ISO/IEC 14496-2, ISO/IEC 14496-5 and ISO/IEC TR 14496-7 correspond to the need of providing the public with conformant standard descriptions that are closer to the starting point of the development of codec implementations than textual descriptions or pure software descriptions. This part of ISO/IEC 14496 contains conformant descriptions of video tools that have been validated within the recommendation ISO/IEC TR 14496-7.

2 Copyright disclaimer for HDL software modules

Each HDL module shall be accompanied by the following copyright disclaimer, to be included in each HDL module and all derivative modules:

```
/*****
```

```
This software module was originally developed by
```

```
<Family Name>, <Name>, <email address>, <Company Name>
```

```
(date: <month>,<year>)
```

```
and edited by: <Family Name>, <Name>,<email address>
```

```
This HDL module is an implementation of a part of one or more MPEG-4  
tools(ISO/IEC 14496).
```

```
ISO/IEC gives users of the MPEG-4 free license to this HDL module or  
modifications thereof for use in hardware or software products claiming  
conformance to the MPEG-4 Standard.
```

```
Those intending to use this HDL module in hardware or software products are  
advised that its use may infringe existing patents.
```

```
The original developer of this HDL module and his/her company, the subsequent  
editors and their companies, and ISO/IEC have no liability for use of this HDL  
module or modifications thereof in an implementation.
```

```
Copyright is not released for non MPEG-4 Video conforming products.
```

```
<Company Name> retains full right to use the code for his/her own purpose, assign  
or donate the code to a third party and to inhibit third parties from using the  
code for non MPEG standard conforming products.
```

This copyright notice must be included in all copies or derivative works.

Copyright (c) <year>.

Module Name: <module_name>.vhd

Abstract:

Revision History:

*****/

3 Abbreviated terms

For the purposes of this document, the following abbreviated terms apply:

AV	Audio-Visual
DCT	Discrete Cosine Transform
IDCT	Inverse Discrete Cosine Transform
HDL	Hardware Description language
ISO	International Organization for Standardization
MPEG	Moving Picture Experts Group
Verilog	A Hardware Description Language
VHDL	VHSIC high speed Hardware Description Language
SAD	Sum of Absolute Differences
MAC	Multiply ACcumulate
MAD	Minimum Absolute Difference
SIMD	Single Instruction Multiple Data
DA	Distributive Arithmetic
EDA	Electronic Design and Automation
IEEE	Institute of Electrical and Electronic Engineers
IMEC	Interuniversity Micro Electronic Center
EPFL	École Polytechnique Fédérale de Lausanne

4 HDL software availability

The HDL and System C software modules described in this part of ISO/IEC 14496 are available within the zip file containing this Technical Report. Each module contains a separate directory structure for the source code with a readme.txt file explaining the top level and all files to be included for simulation and synthesis.

5 HDL coding format and standards

5.1 HDL standards and libraries

As the IEEE has several HDL coding standards that are commonly used in hardware reference code (i.e. VHDL1076-1987, VHDL 1164-1993, Verilog 1364-2000, Verilog 1364-1995), the modules constituting this part

of ISO/IEC 14496 are made of the latest IEEE standard possible at the time of coding for all reference HDL code. As the IEEE has provided libraries to assist in the use of HDL, only IEEE standard libraries are needed to use the HDL code.

Custom libraries which are specific to the vendor's (Silicon) base library elements are used only if they are freely available for synthesis and simulation and are provided in an accompanying module version of the submitted HDL code using the standard libraries mentioned above.

5.2 Conditions and tools for the synthesis of HDL modules

As there are many choices commercially for HDL synthesis and HDL simulation software tools, specific synthesis or simulation libraries that are used for reference HDL code are properly documented. The same code that is used to synthesize towards an implementation is also used to perform HDL behavioral simulation of the MPEG-4 tool. The code is properly documented with respect to the synthesis and simulation tool (and version) that has been used to perform the work. HDL module codes with multiple synthesis and simulation tools are also possible. In the event a source code modification must be made to support an additional synthesis or simulation tool, an additional source code is provided with proper documentation.

5.3 Conformance with the reference software

HDL reference code provides sufficient test bench code and documentation on how it is conformant with respect to the reference software. To the extent possible, bit and cycle true models are provided which can be used directly in the reference software code for verification. In the case that the reference HDL code is derived from other languages such as: C, C++, System C, Java, it is recommended that this code and information on the methodology used to generate HDL should be provided to improve verification of conformance of the HDL code.

6 Integrated Framework supporting the “Virtual Socket” between HDL modules described in Part 9 and the MPEG Reference Software (Implementation 1).

6.1 Introduction

The aim of this clause is to document the framework developed by Xilinx Research Labs for the integration of HW modules with the MPEG-4 reference software. The purpose of this virtual socket framework is to create an abstraction between the specific physical layer and specific software driver library to facilitate a reusable hardware/software co-design environment. By acting as an intermediary between specific physical layer bus protocols, the hardware accelerator designer can focus on the acceleration algorithm rather than lower level interface protocols.

The framework of the Virtual Socket allows for 31 addressable hardware accelerators to be present in a single device (see Figure 1). Each specific hardware accelerator will be assigned a bit of the 32-bit hardware identification register and these bit locations shall be assigned to particular MPEG development teams (see Figure 2 for an example containing two accelerators at slots 1 and 6). If an accelerator socket is not present then its bit in the identification register will be de-asserted. Unassigned sockets will also be de-asserted indicating no accelerator is present. In the event that hardware accelerator designers wish to put further identification of their socket they may do so by allocating further identification registers within their socket's assigned register space.

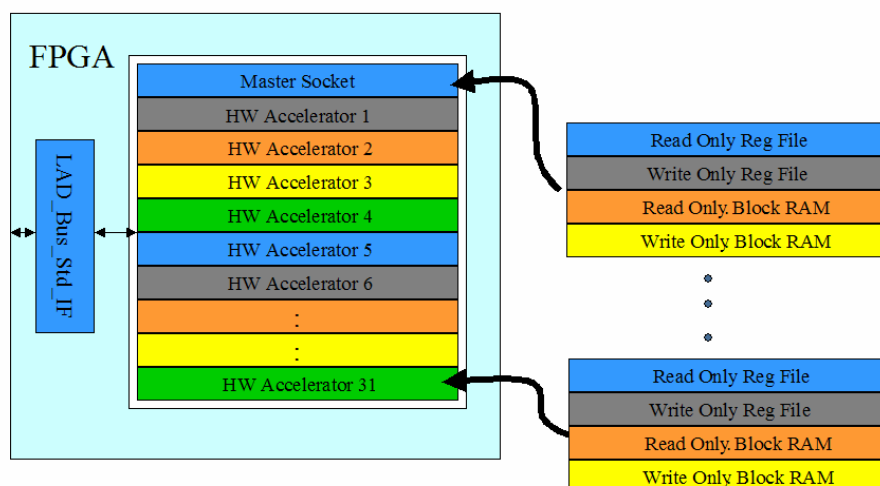


Figure 1 — Block Diagram of Virtual Socket Platform.

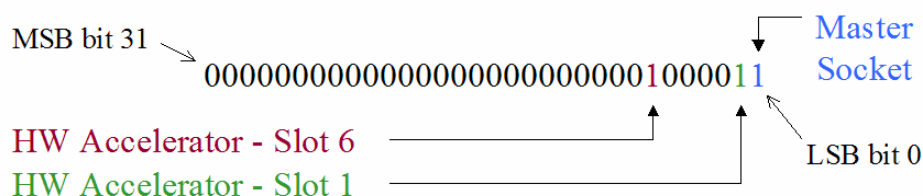


Figure 2 — Example 32-Bit Hardware Identification Register.

6.2 Addressing

The virtual socket provides four strobes that indicate what region of the memory space, register or memory, has been accessed as well as the type of operation, write or read. Although a 16-bit is provided to each socket, the least significant nine bits are only necessary to address within the 512 word assigned memory region. The Virtual Socket API uses macros that assist the software designer in transferring data to and from memory locations.

6.3 Memory Map

Table 1 — Memory Mapping for Register File Allocation.

Socket #	Register Read-Only		Register Write-Only	
	Begin	End	Begin	End
Master	0000	01FF	4000	41FF
1	0200	03FF	4200	43FF
2	0400	05FF	4400	45FF
3	0600	07FF	4600	47FF
4	0800	09FF	4800	49FF
5	0A00	0BFF	4A00	4BFF
6	0C00	0DFF	4C00	4DFF
7	0E00	0FFF	4E00	4FFF

8	1000	11FF		5000	51FF
9	1200	13FF		5200	53FF
10	1400	15FF		5400	55FF
11	1600	17FF		5600	57FF
12	1800	19FF		5800	59FF
13	1A00	1BFF		5A00	5BFF
14	1C00	1DFF		5C00	5DFF
15	1E00	1FFF		5E00	5FFF
16	2000	21FF		6000	61FF
17	2200	23FF		6200	63FF
18	2400	25FF		6400	65FF
19	2600	27FF		6600	67FF
20	2800	29FF		6800	69FF
21	2A00	2BFF		6A00	6BFF
22	2C00	2DFF		6C00	6DFF
23	2E00	2FFF		6E00	6FFF
24	3000	31FF		7000	71FF
25	3200	33FF		7200	73FF
26	3400	35FF		7400	75FF
27	3600	37FF		7600	77FF
28	3800	39FF		7800	79FF
29	3A00	3BFF		7A00	7BFF
30	3C00	3DFF		7C00	7DFF
31	3E00	3FFF		7E00	7FFF

Table 2 — Memory Mapping for the Block RAM Allocation.

	Memory Read-Only			Memory Write-Only	
Socket #	Begin	End		Begin	End
Master	8000	81FF		C000	C1FF
1	8200	83FF		C200	C3FF
2	8400	85FF		C400	C5FF
3	8600	87FF		C600	C7FF
4	8800	89FF		C800	C9FF
5	8A00	8BFF		CA00	CBFF
6	8C00	8DFF		CC00	CDFF
7	8E00	8FFF		CE00	CFFF
8	9000	91FF		D000	D1FF
9	9200	93FF		D200	D3FF
10	9400	95FF		D400	D5FF
11	9600	97FF		D600	D7FF
12	9800	99FF		D800	D9FF
13	9A00	9BFF		DA00	DBFF
14	9C00	9DFF		DC00	DDFF
15	9E00	9FFF		DE00	DFFF
16	A000	A1FF		E000	E1FF
17	A200	A3FF		E200	E3FF
18	A400	A5FF		E400	E5FF
19	A600	A7FF		E600	E7FF
20	A800	A9FF		E800	E9FF
21	AA00	ABFF		EA00	EBFF

22	AC00	ADFF		EC00	EDFF
23	AE00	AFFF		EE00	EFFF
24	B000	B1FF		F000	F1FF
25	B200	B3FF		F200	F3FF
26	B400	B5FF		F400	F5FF
27	B600	B7FF		F600	F7FF
28	B800	B9FF		F800	F9FF
29	BA00	BBFF		FA00	FBFF
30	BC00	BDFF		FC00	FDFF
31	BE00	BFFF		FE00	FFFF

Table 1 and Table 2 show the memory mapping for the 31 hardware sockets in the virtual socket platform. Note in Figure 1 that the memory is allocated into four distinct sections: 1) read-only register file; 2) write-only register file; 3) read-only block RAM; and 4) write-only block RAM. The allocation size for each type of memory for every HW socket is 512 bytes.

6.4 Hardware Accelerator Interface

Figure 3 shows a typical block diagram of a hardware accelerator socket. Note that input and output block RAMs are provided for input and output data while important flags are mapped to the register file sections, such as start and finish flags.

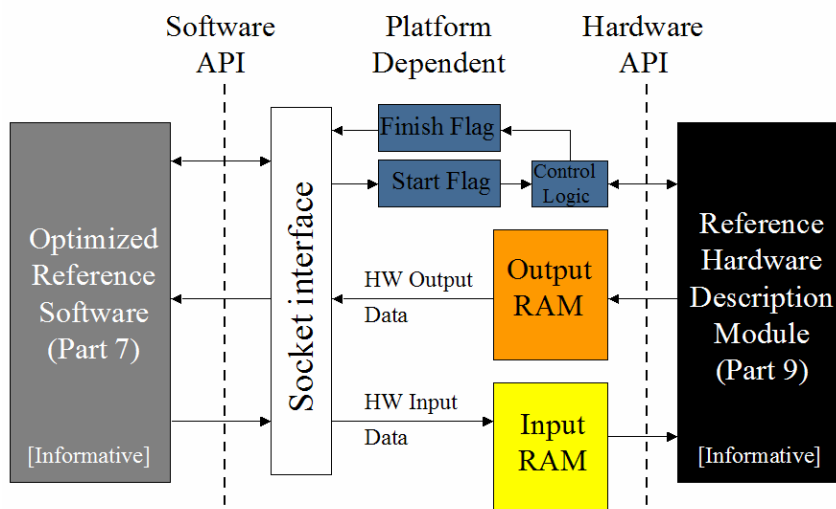


Figure 3 — Block Diagram of Typical Hardware Accelerator.

When a hardware socket is selected for a particular transaction, one of its strobes will be asserted. It is up to the user's particular socket designs whether register or memory regions will be treated differently, however in most cases their behaviour may be identical. The necessary signals to interface to the virtual socket with respect to the hardware accelerator socket are shown in Table 3 below.

Table 3 — Hardware Accelerator Socket Interface.

Signal	Length	Direction*	Polarity	Description
Globals	<2>			
Clk	1	Input	R	hardware accelerator socket clock
global_reset	1	Input	H	global reset
Strobes	<4>			
strobe_reg_read	1	Input	H	read-only register space selected
strobe_reg_write	1	Input	H	write-only register space selected
strobe_ram_read	1	Input	H	read-only memory space selected
strobe_ram_write	1	Input	H	write-only memory space selected
Write Signals	<50>			
write_addr	16	Input		write address
data_in	32	Input		data to write into socket
write_valid	1	Input	H	data_in is valid
write_rdy	1	Output	H	socket available to take more write data
Read Signals	<49>			
read_addr	16	Input		read address
data_out	32	Output		data for read operation
strobe_out	1	Output	H	data_out has requested data
External Memory Manager	<92>			
ZBT_ReadEmpty	1	Input	H	read fifo is empty
ZBT_WriteFull	1	Input	H	write fifo is full
ZBT_ack_job	1	Input	H	job to memory manager accepted
ZBT_wf_grant	1	Input	H	write fifo access granted
ZBT_rf_grant	1	Input	H	read fifo access granted
ZBT_ReadData	32	Input		data read from external memory
ZBT_issue_job	1	Output	H	issue job to memory manager
ZBT_rwb	1	Output	H	job is read = '1' or write = '0'
ZBT_popfifo	1	Output	H	retrieve word of data from read fifo
ZBT_pushfifo	1	Output	H	place data onto write fifo
ZBT_addr	19	Output		address to access data to in external memory
ZBT_dpsh	32	Output		data to send to external memory

The user may optionally connect their device to the external memory manager that allows access to either the ZBT SRAM or DDR DRAM (see subclause 6.4.2). The *block move* and *external block move* example VHDL modules demonstrate the basic interface to the virtual socket (see subclause 6.5). Hardware socket designers are strongly encouraged to read this section and use them as building blocks for their own sockets.

6.4.1 Transferring Data To/From a Socket

When a socket detects a write access to it, it should check to see if the write_valid signal is asserted. This signal will indicate that the data present on Data_In is valid and ready to be processed by the socket. Whenever the user is capable of taking data from the virtual socket interface it should drive its write_rdy signal high. This will bring new data to it from the interface. Register or Memory writes to a socket may be multiple words. The write_rdy signal provides a flow control mechanism back to the virtual socket interface. Below are two waveforms demonstrating example writes to register and memory space.

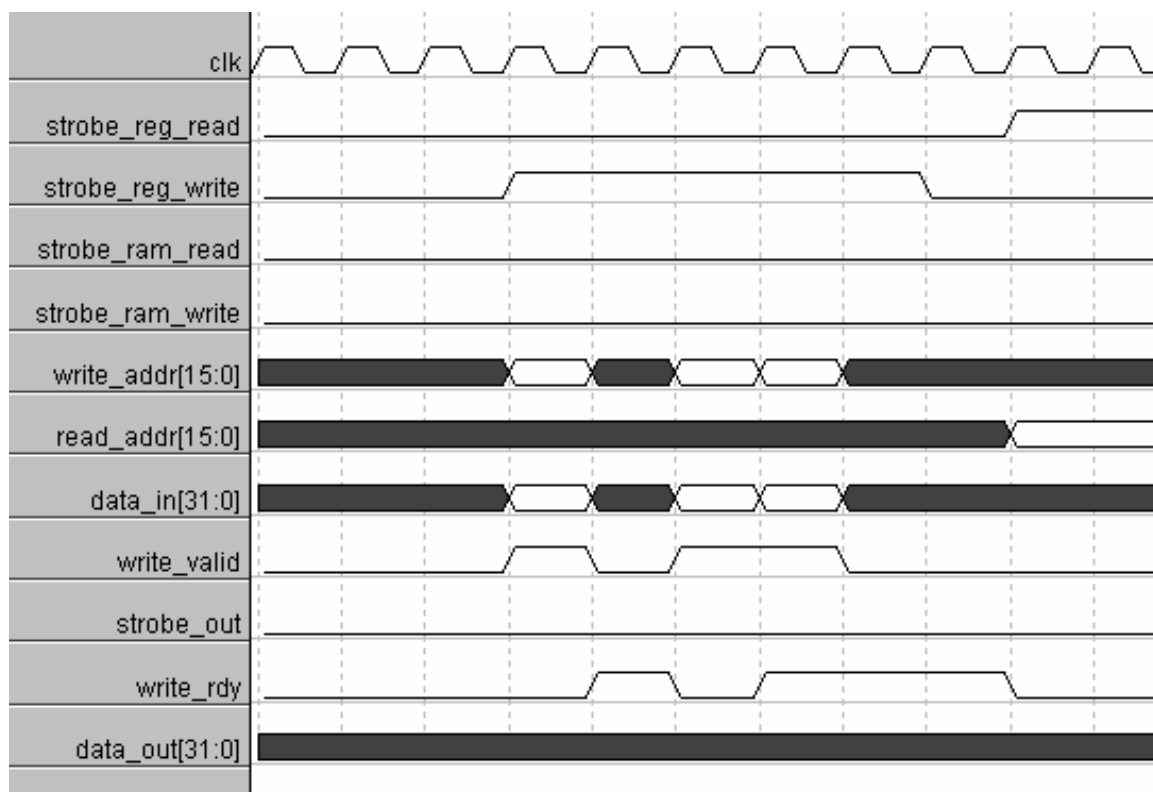


Figure 4 — Timing Diagram of Register Write Operation.

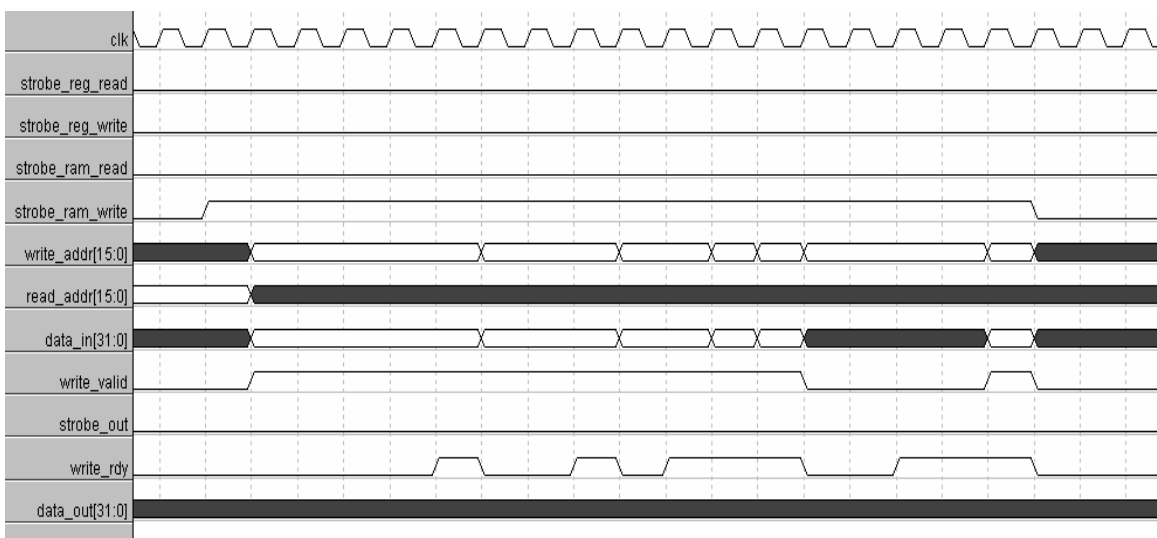


Figure 5 — Timing Diagram of Memory Write Operation.

To perform a read transaction the user should observe a register or memory read strobe asserted. The user has up to 16 clocks to respond to the read transaction by providing the data requested at the read address on data_out and asserting the strobe_out signal. Read operations only return a single word. Samples waveforms are provided below in Figure 6 and Figure 7:

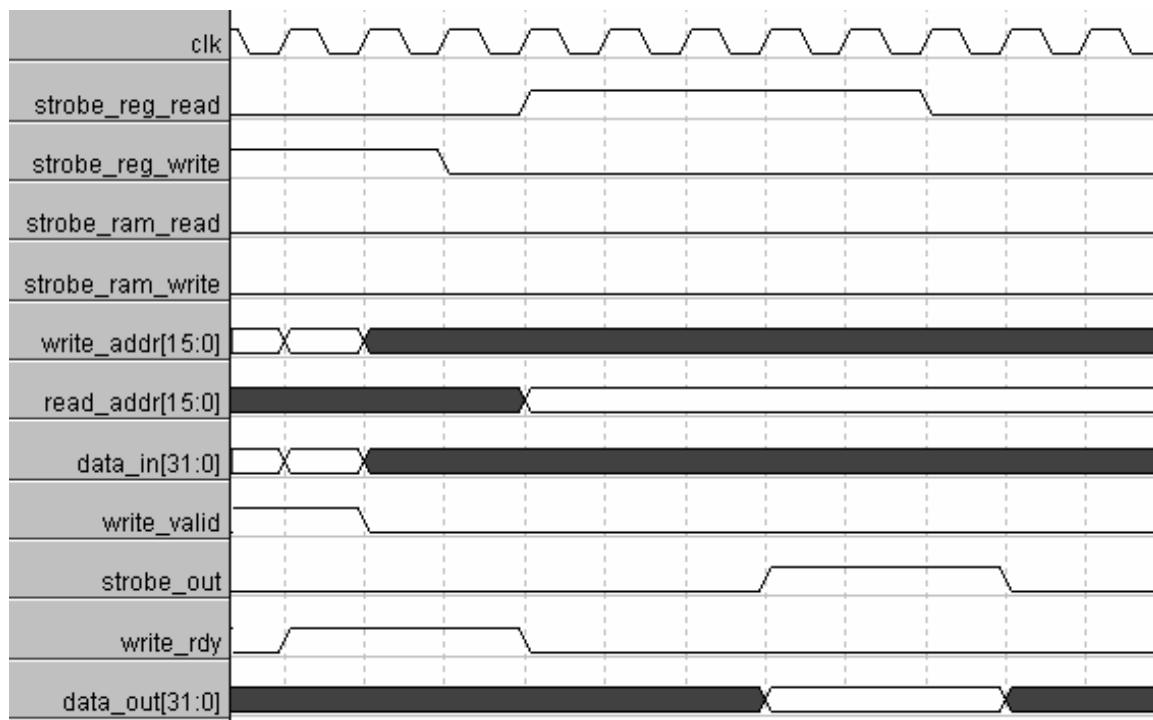


Figure 6 — Timing Diagram of Register Read Operation.

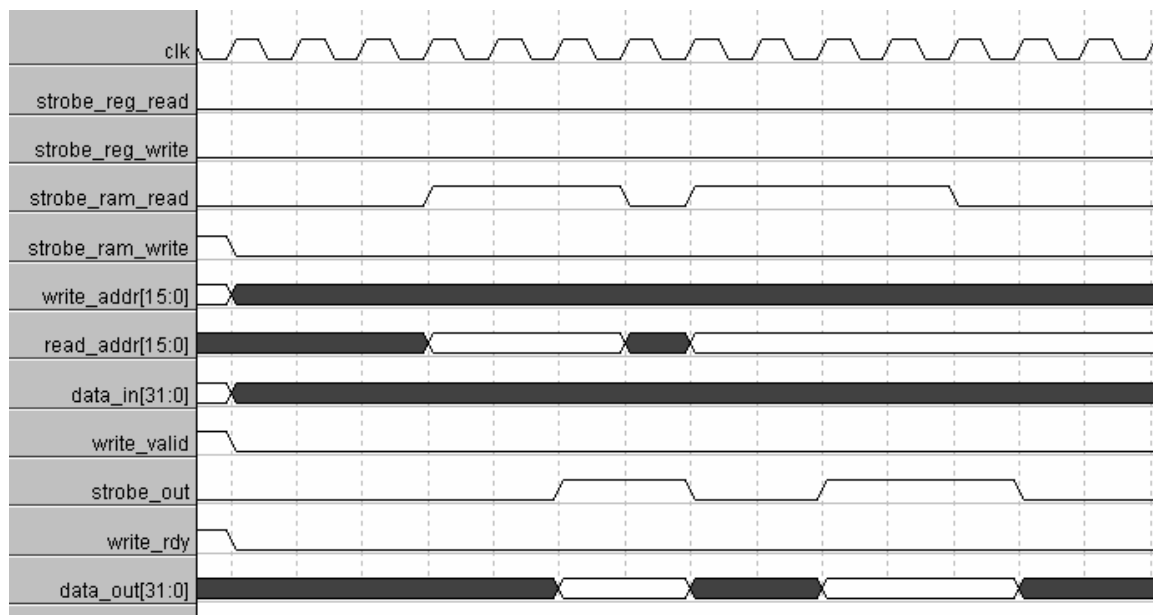


Figure 7 — Timing Diagram of Memory Read Operation.

The hardware socket should implement a simple state machine as shown below in Figure 8.

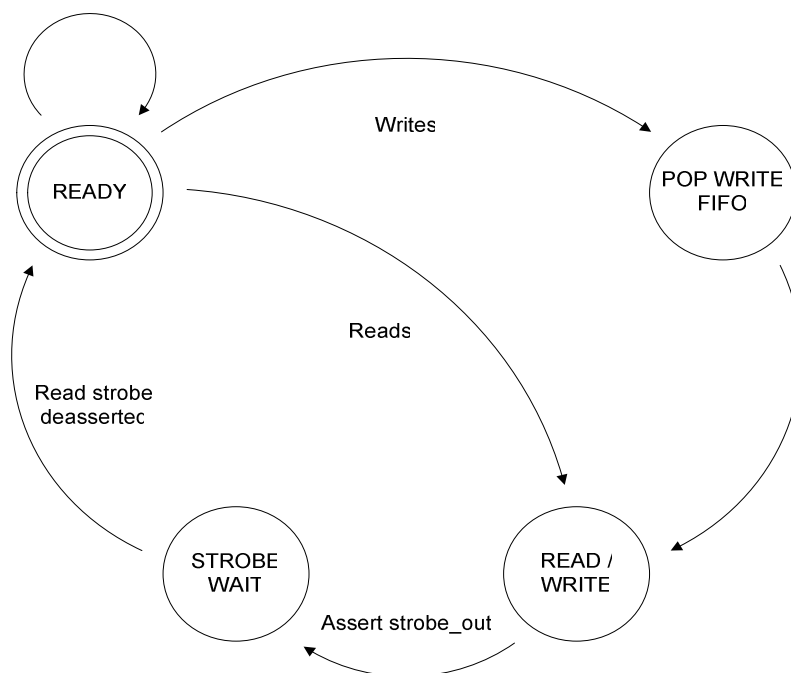


Figure 8 — State Machine of Hardware Accelerator.

6.4.2 External Memory Interface

The external memory manager allows for three hardware accelerator sockets to access the two external memories contained on the WildCard-II, ZBT SRAM and DDR DRAM. The manager arbitrates access to the memory using a round-robin decision method. Sockets requesting access to static or dynamic RAM must first issue a job request to the manager and the manager will respond with an acknowledgement. The socket should then wait until the manager returns either a write or read grant depending on the requested operation. Once a read or write grant is asserted, the socket should either withdraw or deposit (read or write respectively) from the manager. The accesses are for single words only so if a socket requires multiple words, it must request multiple jobs with the manager. An example state machine is provided in Figure 9 demonstrating how the user's socket should interface with the memory manager.

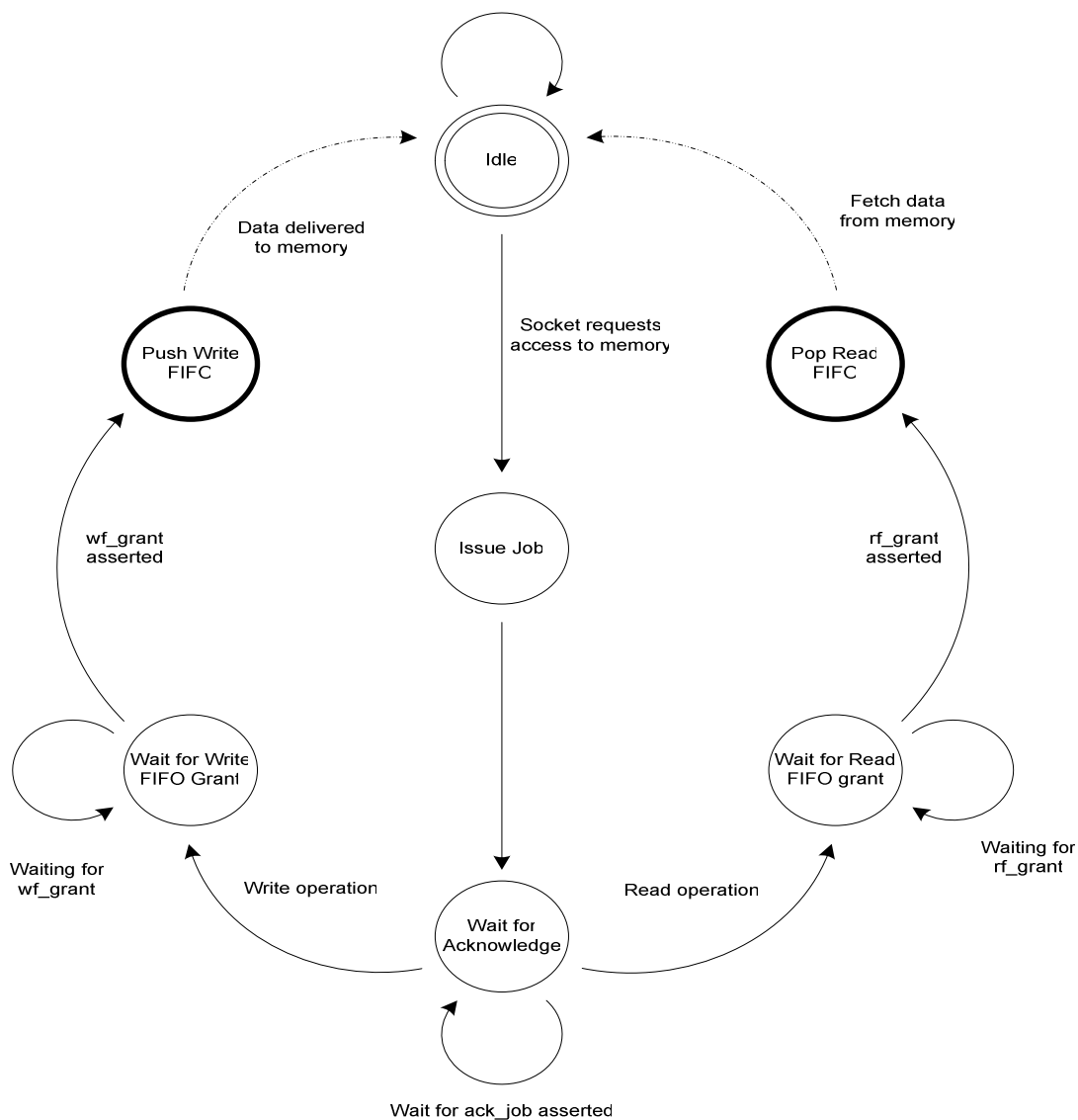


Figure 9 — State Machine of External Memory Interface.

The addressing to the external memories is shown in Figure 10. Note that the user writes a start address value into register location 1 of the master socket. The least-significant nine bits of the memory write value sent to the master socket is then added to the start address register to obtain the final address sent to the external memory.

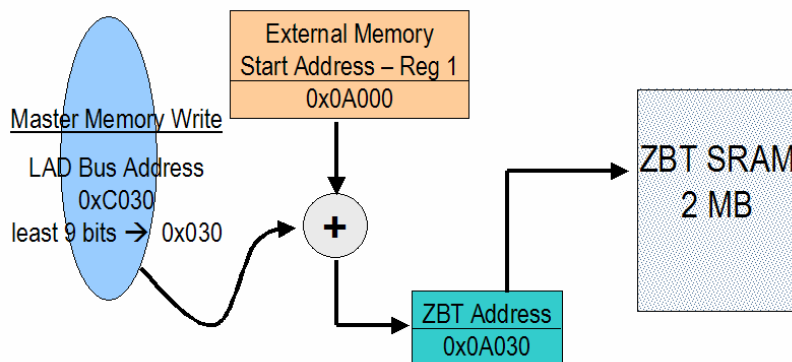


Figure 10 — Addressing Technique Used to Access External Memories.

6.5 User Hardware Accelerator Sockets

Developers of their own hardware accelerator socket should follow the examples listed below to instantiate and activate their own sockets. The VHDL source code for these two examples is provided with the platform and is already pre-connected to the platform. Users are encouraged to copy these examples to use as a template for their own accelerator design. The virtual socket interface is comprised of two major modules that cannot be altered by the user – the master socket and memory manager. These modules must be present to allow software to access external memory as well as provide status information back to control software.

6.5.1 Block Move

The *block move* example connects to the virtual socket interface connections and performs a very simple task - copying the contents of one internal RAM to another. Specifically the block move example copies a region of the Write-Only memory to the Read-Only memory. The example watches for a write to its Start register and begins a loop reading from one memory and writing to the other. Once the block move module has finished this task, it sets its Valid register high. Figure 11 and Figure 12 show the interface and internal block diagram of the block move example, respectively. Note that the interface matches the basic interface listed in Table 3.

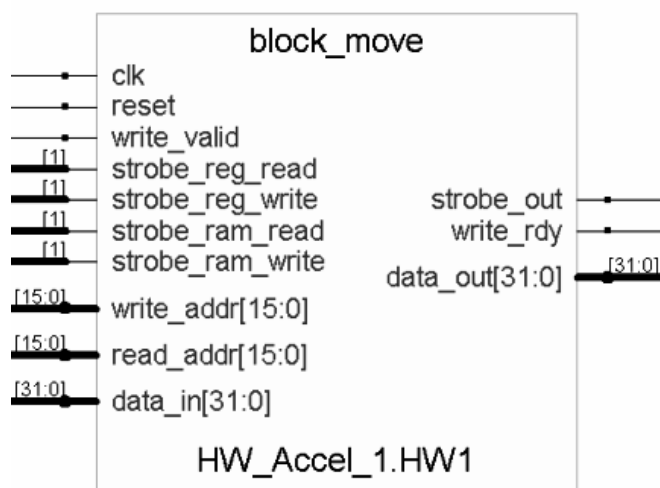


Figure 11 — Interface of Block Move Example.

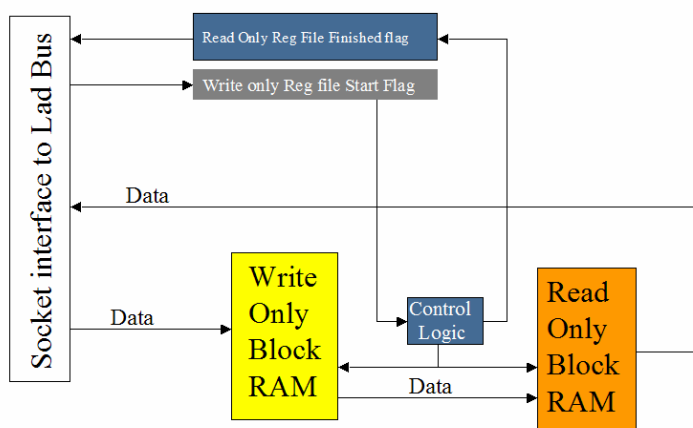


Figure 12 — Block Diagram of Block Move Example.

6.5.2 External Memory Block Move

The *external block move* example VHDL module performs a copy from one region of ZBT SRAM memory to another. Figure 13 shows the interface of this block. It demonstrates the additional connectivity to the external memory manager. This socket provides additional registers to configure the source and destination addresses of the copy and the length of the transfer. A write to the Start register begins the copy and the status of the transfer is provided in the Status register. The number of words moved is present in the high bits of the Status register and the current state of the socket in the lower three bits. When the transfer has completed, the number of words should match the requested length to copy and the state should be all zeroes indicating it has finished.

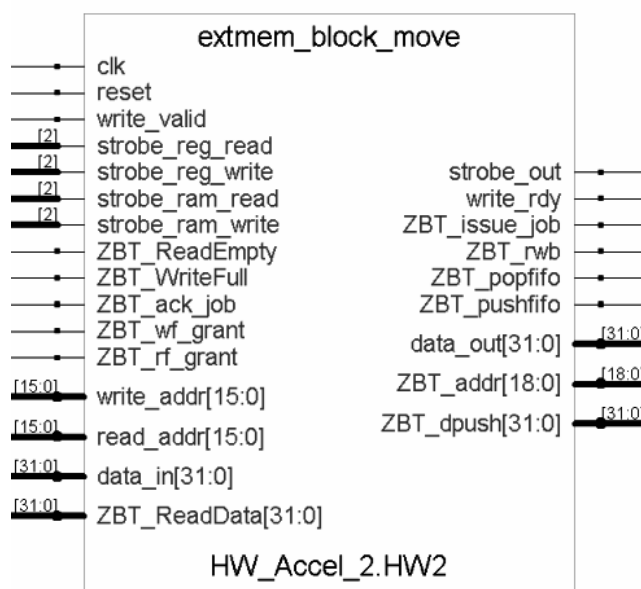


Figure 13 — Interface of External Block Move Example.

7 Integrated Framework supporting the “Virtual Socket” between HDL modules described in Part 9 and the MPEG Reference Software (Implementation 2).

7.1 Introduction

The aim of this clause is to document the framework developed at the University of Calgary for the integration of HW modules with the MPEG-4 reference software.

The clause is formatted in the form of a tutorial that takes the reader step by step through a typical HDL module development case. After the introduction, two examples are described. The first one permit to transfer from host data into FPGA internal memory block, then to do a simple process and then the data are transferred back to host. The second example is similar, but instead of data processing the data are stored into a internal fifo (designed with FPGA memory blocks) and straight away read back by the host. For both examples, a testbench is described to test the two IPs and to help user to understand how host's command words could be interpreted and the processing performed. The following shows the integration of these two examples into the platform and how host controls these IPs. Then it is shown how to use API function and how to use the debug platform.

A typical data follow for an IP accelerator is: The data is transferred from the host computer to FPGA memory block (BRAM) or SRAM memory, (using either direct transfer or DMA protocol) the processing is done, then the results are stored respectively into BRAM or SRAM or (DRAM which is not presented in this document) depending on the source of the original data. The computer host initiates and controls (through software calls) the IP with an IP controller (designed by the user). Different types of controller are detailed into the next sections. The system includes two parts (layers): A software part in C/C++ and a Part in HDL that simulate the IP block. Both share the operation to perform the functionality required. In all examples of this document, it was chosen that the S/W layer receives and responds to the all interrupts (act as Interrupt management). Also the S/W controls the start and acknowledges the operation of the H/W module.

7.2 Development Example of a Typical Module : Calc_Sum_Product Module

In this example, The data are transferred from the host computer to FPGA memory block (BRAM) memory with no DMA, nor SRAM the processing is done, then the results are stored respectively into BRAM. The host initiates and controls the IP with an IP controller. The controller is into the VHDL file calc_sum_product.vhd.

```
entity calc_sum_product is
    generic(no_of_inputs: positive:=64);
    port(
        reset:          in std_logic;
        clock:          in std_logic;
        -----
        module_ready:    out std_logic;
        input_available: in std_logic;
        datain:          in std_logic_vector(7 downto 0);
        -----
        output_available: out std_logic;
        dataout:          out std_logic_vector(15 downto 0);
        -----async transfers--
        async_in:        in std_logic;
        async_out:       out std_logic
    );
end calc_sum_product;
```

Figure 14 — Interface of a Typical Module.

The specifications of this calculator:

1. Takes in 64 words-8 bits each, one word at a time (asynchronous transfer), each word is stored into FPGA memory block.
2. It calculates the sum and the product of each two consecutive words. The sum and the product are separately stored in 16-bit words.

3. When the calculation for all the input is done, it begins to output the 64 results, one word at a time, then waits for new input.

4. The interface signals are:

- a. Reset. (Input)
- b. Clock. (Input)
- c. module_ready. (Output) (the module is ready for receiving new input stream)
- d. Input_ready (Input) (signals start of the input stream)
- e. Output_ready. (Output) (signals start of the output stream)
- f. Data_in. (Input)
- g. Data_out. (Output)
- h. Async_in (Input), Async_out (Output) asynchronous data transfer handshake signals.

They are used in inverse sense when data source and data destination switch roles.

Assumptions:

- a. Reset is active high.
- b. System is positive edge triggered.
- c. One clock delay between data and handshake signals.

The second step is to have this module tested by preferably a testbench file similar to the following one shown in Figure 15.

```

-----
-- example testbench for the example calculator module
-- Tamer Mohamed and Wael Badawy (badawy@ucalgary.ca)
-- University of Calgary
-----

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;

entity tb_calc_sum_product is
end tb_calc_sum_product;

architecture stimulus of tb_calc_sum_product is

    type memory8bits is array (natural range <>) of std_logic_vector(7 downto 0);
    type memory16bits is array (natural range <>) of std_logic_vector(15 downto 0);
    type system_modes is (idle, receive_up, receive_dn, calculating, transmit_up, transmit_dn);

    constant m_period: time :=10 ns; -- suggests operation at 100 MHz
    constant tb_period: time :=7 ns;
    constant no_of_in: positive:=16; -- suggests 16 inputs only

    signal input_array: memory8bits(0 to no_of_in-1);
    signal output_array: memory16bits(0 to no_of_in-1);
    signal current_state: system_modes;

    signal reset, m_clk: std_logic;
    signal tb_clk: std_logic;
    signal module_ready: std_logic;
    signal input_available, output_available: std_logic;
    signal async_in, async_out: std_logic;
    signal datain: std_logic_vector(7 downto 0);
    signal dataout: std_logic_vector(15 downto 0);

    signal initiate_processing: std_logic; -- external trigger signal

    component calc_sum_product

```

```

generic(no_of_inputs: positive:=64);
port(
    reset:          in std_logic;
    clock:          in std_logic;
    module_ready:    out std_logic;
    input_available: in std_logic;
    datain:         in std_logic_vector(7 downto 0);
    output_available: out std_logic;
    dataout:        out std_logic_vector(15 downto 0);
    async_in:       in std_logic;
    async_out:      out std_logic
);
end component;

begin

    UUT: calc_sum_product
        generic map (no_of_inputs => no_of_in)
        port map (
            reset => reset,
            clock => m_clk,
            module_ready => module_ready,
            input_available => input_available,
            datain => datain,
            output_available => output_available,
            dataout => dataout,
            async_in => async_in,
            async_out => async_out
        );

        ----- begin external signals

        input_array
        <=(X"00",X"01",X"02",X"03",X"04",X"05",X"06",X"07",X"08",X"09",X"0A",X"0B",X"0C",X"0D",X"0E",X"0F");
        reset <= '1', '0' after 12 ns;
        initiate_processing <= '1', '0' after 50 ns, '1' after 1000 ns, '0' after 1050 ns;

        module_clock: process
        begin
            m_clk <= '0';          wait for m_period/2;
            m_clk <= '1';          wait for m_period/2;
        end process module_clock;

        testbench_clock: process
        begin
            tb_clk <= '0';          wait for tb_period/2;
            tb_clk <= '1';          wait for tb_period/2;
        end process testbench_clock;

        ----- end external signals

        stimuli: process(reset, tb_clk)
            variable counter: integer range 0 to 63;
        begin
            if reset='1' then
                counter :=0;
                current_state <=idle;
                input_available <='0';
                datain <=(others=>'0');
            elsif rising_edge(tb_clk) then
                case current_state is
                    when idle =>

```

```

        if initiate_processing='1' then
            if module_ready='1' then
                input_available <='1';
                current_state <= receive_up;
            end if;
        end if;
    when receive_up =>
        if async_out='0' then
            async_in <='1';      -- latch
            datain <=input_array(counter);
            current_state <= receive_dn;
        end if;
    when receive_dn =>
        if async_out='1' then
            async_in <='0';
            counter :=((counter+1) mod no_of_in);
            if counter=0 then
                input_available <='0';
                current_state <= calculating;
            else
                current_state <= receive_up;
            end if;
        end if;
    when calculating =>
        if output_available='1' then
            current_state <= transmit_up;
        end if;
    when transmit_up =>
        if async_out='1' then -- latch
            output_array(counter) <= dataout;
            async_in <='1'; --acknowledged
            current_state <= transmit_dn;
        end if;
    when transmit_dn =>
        if async_out='0' then
            async_in <='0';
            counter :=((counter+1) mod no_of_in);
            if counter=0 then
                current_state <=idle;
            else
                current_state <=transmit_up;
            end if;
        end if;
    end case;
end if;

end process stimuli;

end stimulus;

```

Figure 15 — Testbench of the Typical Module.

The following points should be noted about the structure of the test-bench file:

1. The testbench file is divided into external control part and the “stimuli” process which is responsible for feeding the data to the module. This process; as a state machine, has the same states as the calculator module.
2. The process; “*stimuli*”, is made sensitive to a clock and a reset signals. This provides for maximizing code reuse because this process can be taken directly and integrated as part of the module controller discussed in the following sections.

3. The sequence of the operations is described into the testbench, therefore in the implementation a component (HW_controller) will be in charge of it. The 16 data words (constant no_of_in := 16;) are stored into a input array, memory (will see for implementation), after the processing the stimuli process is in charged to store the results into the output array memory. The sequence can be split into 3 phases:

- a. reading data (received_up and received_dn)
- b. processing data (calculating)
- c. writing resulting data (transmit_up and transmit_dn)

4. There are two generated clocks in the test-bench: *m_clk* which is the module clock and *tb_clk* which is the clock used for the test-bench process. There is no relation between the two clocks as *m_period* is 10ns and *tb_period* is 7ns. To account for other cases, the two periods should be interchanged.

5. Data transfers are asynchronous using the handshake signals *async_in* and *async_out*.

6. The external triggering signal called “*initiate_processing*” is asserted two times with a delay between the two times it is asserted. The goal of this is to make sure that both the module and its controlling process; (process “stimuli”) can run again after going into the idle state. This makes sure that the block after an initial run will not be left in a state that may cause errors during a successive run.

7. To make reviewing the results in the simulator easier, the module is instantiated in the test-bench with only 16 inputs. This is the main use of the generic value here.

Figure 16 shows the simulation of the testbench.

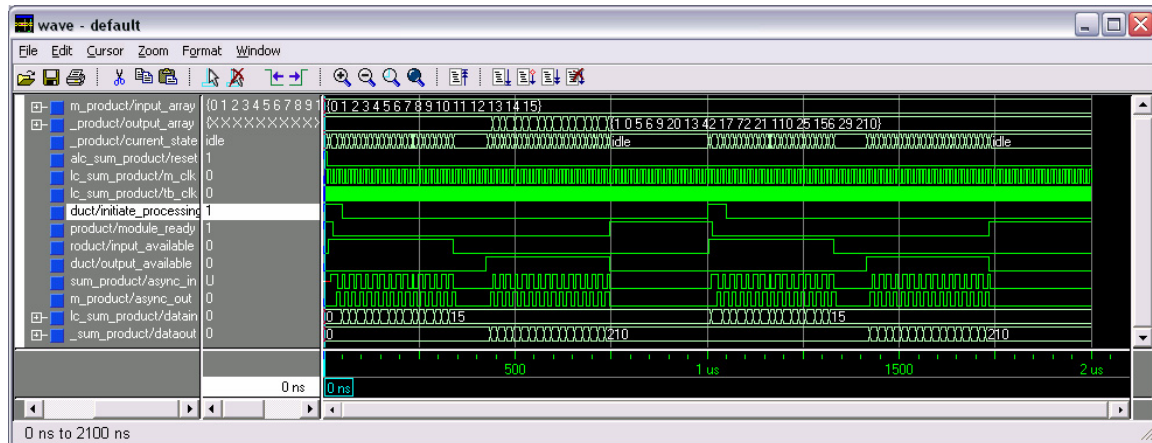


Figure 16 — Simulation of the Testbench for the *calc_sum_product* Module.

The compilation in Modelsim can done by a macro file that can bet similar to the following:

```
-----
-- compilation macro file for target module
-- Tamer Mohamed and Wael Badawy (badawy@ucalgary.ca)
-----

set MODULE_BASE D:/MPEG4/UoC_framework3/vhdl/sim/calc

vlib $MODULE_BASE/calc_lib
vmap calc_lib $MODULE_BASE/calc_lib

vcom -93 -explicit -work calc_lib \
    $MODULE_BASE/calc_sum_product.vhd \
    $MODULE_BASE/tb_calc_sum_product.vhd
```

Figure 17 — “*compilation.do*” macro file for the module and its testbench.

7.3 Second Example of a Typical Module : *fifo_transfer* module

In this subclause we will develop a second example and in the following sections we will integrate both developed modules in our framework. Our second module is a FIFO register file.

The specifications of this FIFO register:

1. Its depth is 64 words-16 bits each, accepts one word at a time (asynchronous transfer).
2. The interface signals are:
 - a. Reset. (Input)
 - b. Clock. (Input)
 - c. fifo_empty. (Output) (stored word counter is at 0)
 - d. fifo_full. (Output) (stored word counter is at maximum)
 - e. Write_fifo (Input) (signals a word to be loaded into the FIFO)
 - f. Write_ack (Output) (asynchronous handshake)
 - g. read_fifo. (Input) (signals topmost word is to be emptied)
 - h. read_ack (Output) (asynchronous handshake)
 - i. Datain. (Input)
 - j. Dataout. (Output)

Assumptions:

- a. Reset is active high.
- b. System is positive edge triggered.
- c. Data is latched with the write_fifo signal.
- d. Data writing and data reading are two independent operations, each has its own asynchronous acknowledge signal. This was not the case in the first example.

The following is a description of the module and its test-bench in VHDL reported in Figure 5 and Figure 6. The simulation result for two runs is shown in Figure 20.

Note that the outline of the testbench code is almost identical to the one in the first example. This helps in code reuse and emphasizes the abstraction layer discussed in the next subclause. Also note the use of two clocks.

```

-----
-- example FIFO hw module
-- Tamer Mohamed and Wael Badawy (badawy@ucalgary.ca)
-- University of Calgary
-----
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;

entity fifo_reg is
  generic(
    fifo_depth: positive:=64;
    fifo_width: positive:=8
  );
  port(
    reset:   in std_logic;
    clock:   in std_logic;
    fifo_empty: out std_logic;
    fifo_full: out std_logic;
    -----
    write_fifo: in std_logic;
    write_ack:  out std_logic;
    datain: in std_logic_vector(fifo_width-1 downto 0);
    -----
    read_fifo:  in std_logic;
    read_ack:   out std_logic;
    dataout:   out std_logic_vector(fifo_width-1 downto 0)
  );
end fifo_reg;

```

Figure 18 — FIFO Module Example (described in fifo_reg.vhd).

```

-----
-- example testbench for the example FIFO module
-- Tamer Mohamed and Wael Badawy (badawy@ucalgary.ca)
-- University of Calgary
-----

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;

entity tb_fiforeg is
end tb_fiforeg;

architecture stimulus of tb_fiforeg is

    constant m_period: time :=7 ns;
    constant tb_period: time :=10 ns; -- suggests operation at 100 MHz
    constant no_of_in: positive:=16;    -- suggests 16 inputs only
    constant d_wid: positive:=8;

    type memorybits is array (natural range <>) of std_logic_vector(d_wid-1 downto 0);
    type system_modes is (idle, receive_ack, transmit_ack);
    type system_requests is (idle, push_fifo, pop_fifo);

    signal current_state: system_modes;
    signal current_request: system_requests;

    signal reset, m_clk, tb_clk: std_logic;
    signal fifo_empty, fifo_full: std_logic;
    signal read_fifo, read_ack: std_logic;
    signal write_fifo, write_ack: std_logic;
    signal datain: std_logic_vector(d_wid-1 downto 0);
    signal dataout: std_logic_vector(d_wid-1 downto 0);
    signal fifo_data: std_logic_vector(d_wid-1 downto 0);
    signal request_done: std_logic;

    signal initiate_processing: std_logic;    -- external trigger signal
    signal counter1, counter2: integer range 0 to no_of_in-1;
    signal input_array: memorybits(0 to no_of_in-1);
    signal output_array: memorybits(0 to no_of_in-1);

    component fifo_reg is
        generic(
            fifo_depth: positive:=64;
            fifo_width: positive:=8
        );
        port(
            reset:         in std_logic;
            clock:          in std_logic;
            fifo_empty:    out std_logic;
            fifo_full:     out std_logic;

            -----
            write_fifo:    in std_logic;
            write_ack:     out std_logic;
            datain:        in std_logic_vector(fifo_width-1 downto 0);

            -----
            read_fifo:     in std_logic;
            read_ack:      out std_logic;
            dataout:       out std_logic_vector(fifo_width-1 downto 0)
        );

```

```

        end component;

begin

    UUT: fifo_reg
        generic map (
            fifo_depth => no_of_in,
            fifo_width => d_wid
        )
        port map (
            reset => reset,
            clock => m_clk,
            fifo_empty => fifo_empty,
            fifo_full => fifo_full,
            write_fifo => write_fifo,
            write_ack => write_ack,
            datain => datain,
            read_fifo => read_fifo,
            read_ack => read_ack,
            dataout => dataout
        );

        ---- begin external signals

        input_array
        <= (X"00",X"01",X"02",X"03",X"04",X"05",X"06",X"07",X"08",X"09",X"0A",X"0B",X"0C",X"0D",X"0E",X"0F");
        reset <= '1', '0' after 12 ns;
        initiate_processing <= '1', '0' after 50 ns, '1' after 750 ns, '0' after 800 ns;

        current_request <= push_fifo, idle after 320 ns, pop_fifo after 350 ns, idle after 700 ns,
        push_fifo after 750 ns;

        counter1_p: process
        begin
            wait until write_ack='1';
            counter1 <= (counter1+1) mod no_of_in;
        end process counter1_p;
        counter2_p: process
        begin
            wait until read_ack='1';
            counter2 <= (counter2+1) mod no_of_in;
        end process counter2_p;

        fifo_data <= input_array(counter1);
        output_array(counter2) <= dataout;

        module_clock: process
        begin
            m_clk <= '0';          wait for m_period/2;
            m_clk <= '1';          wait for m_period/2;
        end process module_clock;

        testbench_clock: process
        begin
            tb_clk <= '0';          wait for tb_period/2;
            tb_clk <= '1';          wait for tb_period/2;
        end process testbench_clock;

        ---- end external signals

        stimuli: process(reset, tb_clk)
        begin

```

```

if reset='1' then
    current_state <=idle;
    write_fifo <='0';
    read_fifo <='0';
    datain <=(others=>'0');
    request_done<='1';
elsif rising_edge(tb_clk) then
    case current_state is
        when idle =>
            if current_request=push_fifo then
                if fifo_full='0' then
                    datain <= fifo_data;
                    write_fifo <='1';
                    current_state <= receive_ack;
                    request_done <='0';
                end if;
            elsif current_request=pop_fifo then
                if fifo_empty='0' then
                    read_fifo <='1';
                    current_state <= transmit_ack;
                    request_done <='0';
                end if;
            else
                write_fifo <='0';
                read_fifo <='0';
                request_done <='1';
            end if;

            when receive_ack =>
                if write_ack='1' then
                    write_fifo <='0';
                    current_state <= idle;
                end if;

            when transmit_ack =>
                if read_ack='1' then
                    read_fifo <='0';
                    current_state <= idle;
                end if;
            end case;

        end if;

    end process stimuli;

end stimulus;

```

Figure 19 — Testbench for the FIFO Module (described in .vhd).

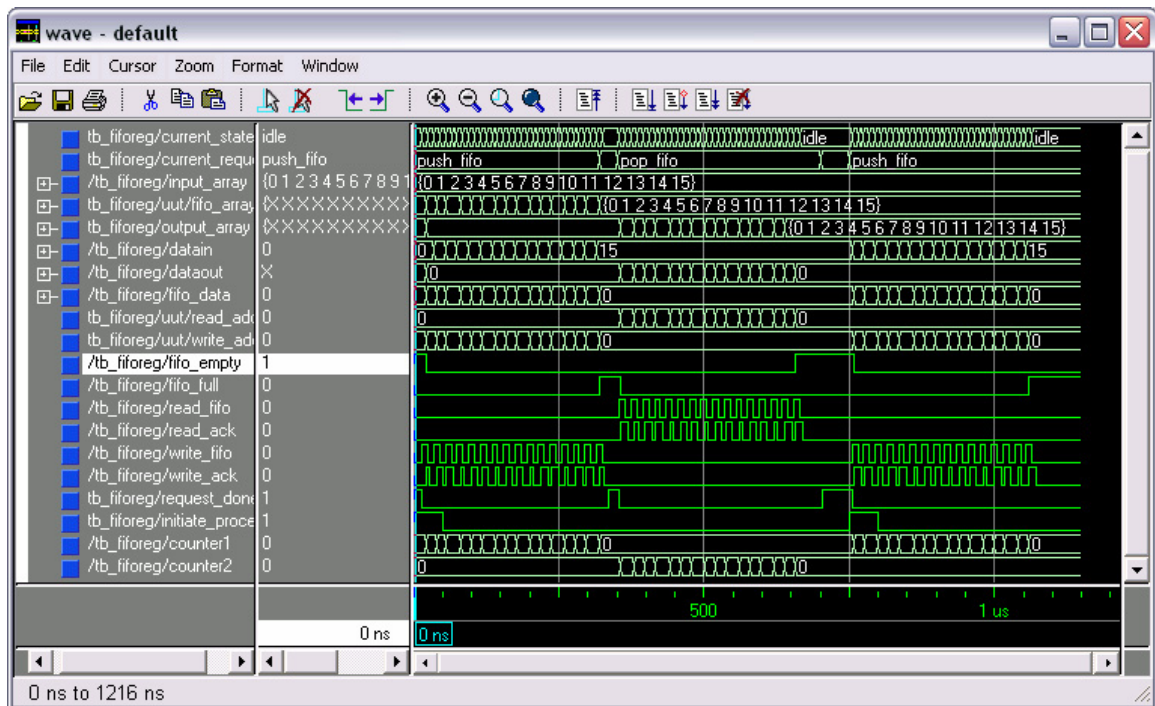


Figure 20 — Simulation of Two Consecutive Runs of the FIFO.

7.4 Integrating the Multi-Modules within the Framework

In this subclause we demonstrate the interaction of 2 modules but the system can be scaled up to 16 modules. In the previous part, the connection is established between the HW integration and the API function calls. To obtain information about these functions, please refer to the software platform code (Visual UoCfv3 project) or for simulation to function used with the test vectors (in host_arch.vhd). The results of test vectors can be observed with ModelSim using the script simulation named project_vsim.do. In the simulation, four examples are described, and two of them are the fifo and cal examples.

USER should be warned that:

1. A new processing IP is ALWAYS associated with a controller module which should be designed by the user!!! In this controller, there is at least a LAD interface. Two controllers are provides into the following examples.
2. The second example permits to user IP to access to SRAM.

The HW part of the framework consists of architecture that is mapped to the FPGA. The architecture can be described by Figure 21.

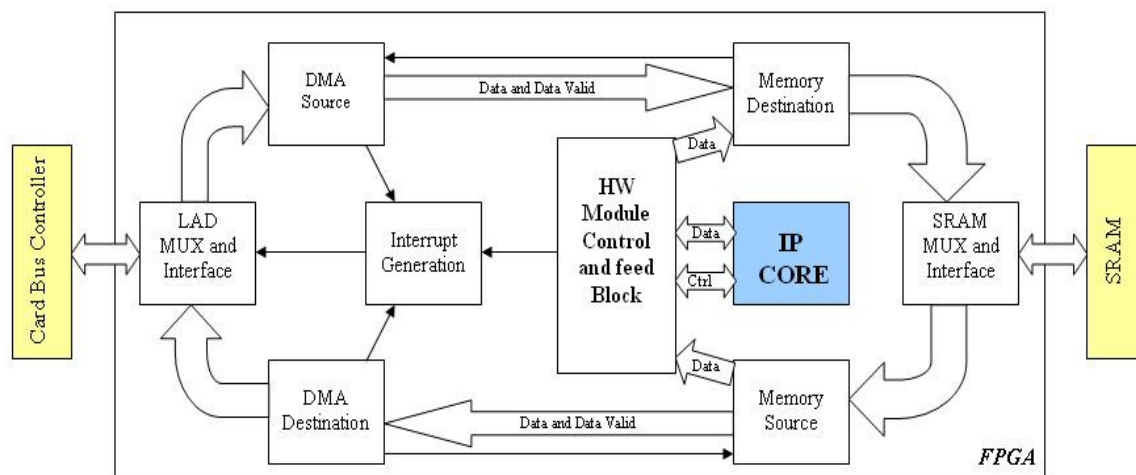


Figure 21 — System architecture mapped to the FPGA.

The “HW module Control and Feed Block” is an abstraction layer whose purpose is to shield the designer of the IP core from the intricate details of the other interfacing blocks required in the HW/SW environment. The other blocks are responsible for interfacing with the PCI bus of the host system and for interfacing with the on board memory chips (SRAM for instance).

Our example system will function properly with the two IP blocks discussed in the previous two sections.

We may choose that the FIFO block be implemented with direct register transfers via the bus and that the calculator takes benefit of the DMA capability. This will illustrate how two different mechanisms of data transfer are implemented through the framework.

We will begin by the simple case of data transfer in the form of one word at a time. The FIFO example suits this type of transfer.

7.4.1 FIFO Module Controller (basic data transfer)

The test-bench file is used as a starting point and is integrated in a simple VHDL template that addresses the following:

1. Interfacing with the local address bus (LAD) through which the host system communicates with the processing element (PE) or FPGA.
2. Passing external control arguments to the “stimuli” process described in the test-bench file and letting it handle the main module.
3. Accessing the card memory if required. (Not the case in this example).
4. Generating an interrupt signal to indicate that the main module finished processing.
5. Assigns values to a status register to indicate the current state of the system. This status register can be queried periodically by the host system if the interrupt signal is masked (which is the case in this example as will be illustrated in the full system).

The description of the controller interface is as follows:

```

entity fifo_hw_module_controller is
  generic(
    BASE_address: std_logic_vector(15 downto 0);
    address_MASK: std_logic_vector(15 downto 0):=X"FFF0"; -- 16 registers
    fifo_size: positive:=16;
    fifo_width: positive:=16
  );
  port(
    reset: in std_logic;
    m_clk: in std_logic;      -- module clock
    b_clk: in std_logic;      -- bus clock
    module_done: out std_logic;
    ----- memory access arbiter
    write_address: out std_logic_vector(20 downto 0);
    read_address: out std_logic_vector(20 downto 0);
    enable_write: out std_logic;
    enable_read : out std_logic;
    access_request: out std_logic;
    access_grant: in std_logic;
    Memory_Source_Data_Valid: in std_logic;
    mem_datain : in std_logic_vector(31 downto 0);
    mem_dataout : out std_logic_vector(31 downto 0);
    ----- interface with LAD bus
    LAD_instrobe: in std_logic;
    LAD_address: in std_logic_vector(15 downto 0);
    LAD_write: in std_logic;
    LAD_datain: in std_logic_vector(31 downto 0);
    LAD_dataout: out std_logic_vector(31 downto 0);
    LAD_strobe_out: out std_logic
  );
end fifo_hw_module_controller;

```

Figure 22 — Interface of the FIFO Controller (declared in fifo_ctrl2.vhd).

The following points should be noticed:

1. Generics: BASE_address and address_MASK provide the parameters for the address bus comparator. Fifo_size and fifo_width are parameters for the main ip module.
2. Signals: The signals are divided into 3 categories: global control (reset and the two clocks), card memory access signals, LAD interface.

The controller file template is divided into 3 main *processes*:

1. LAD interface process.
2. Memory interface process.
3. IP-module interface process. (This is almost the exact form used in the test-bench "*stimuli*" process which is the point in code reuse)

The example here omits the memory access process because it is not used. It will be shown in the second example for the calculator.

The VHDL controller architecture (declared in fifo_ctrl2.vhd) is shown in Figure 23.

```

architecture behav of fifo_hw_module_controller is

  constant zeropad: std_logic_vector(31-fifo_width downto 0):=(others=>'0');
  type system_modes is (idle, receive_ack, transmit_ack);
  type system_requests is (idle, push_fifo, pop_fifo);

  signal current_state: system_modes;
  signal current_request: system_requests;
  signal request_done: std_logic;

```

```

signal status_register: std_logic_vector(7 downto 0);

--signal reset, m_clk, tb_clk: std_logic;
signal fifo_full, fifo_empty: std_logic;
signal read_fifo, read_ack: std_logic;
signal write_fifo, write_ack: std_logic;
signal fifo_data: std_logic_vector(fifo_width-1 downto 0);
signal datain: std_logic_vector(fifo_width-1 downto 0);
signal dataout: std_logic_vector(fifo_width-1 downto 0);

component fifo_reg is
  generic(
    fifo_depth: positive:=64;
    fifo_width: positive:=8
  );
  port(
    reset:          in std_logic;
    clock:          in std_logic;
    fifo_empty:     out std_logic;
    fifo_full:      out std_logic;
    -----
    write_fifo:     in std_logic;
    write_ack:      out std_logic;
    datain:         in std_logic_vector(fifo_width-1 downto 0);
    -----
    read_fifo:      in std_logic;
    read_ack:       out std_logic;
    dataout:        out std_logic_vector(fifo_width-1 downto 0)
  );
end component;

begin
  U_fiforeg: fifo_reg
    generic map (
      fifo_depth => fifo_size,
      fifo_width => fifo_width
    )
    port map (
      reset => reset,
      clock => m_clk,
      fifo_empty => fifo_empty,
      fifo_full => fifo_full,
      write_fifo => write_fifo,
      write_ack => write_ack,
      datain => datain,
      read_fifo => read_fifo,
      read_ack => read_ack,
      dataout => dataout
    );

  module_done <= request_done;
  status_register <= ( 0=>fifo_empty, 1=>fifo_full, others=>'0');

  -- This module will not access memory (optimized during synthesis)
  write_address <=(others=>'0');
  read_address <=(others=>'0');
  enable_write <='0';
  enable_read <='0';
  access_request <='0';

```

```

mem_dataout <=(others=>'0');
-----

LAD_interface: process(reset, b_clk)
begin
    if reset='1' then
        LAD_dataout <=(others=>'0');
        LAD_strobe_out <='0';
        current_request <=idle;
        fifo_data <= (others=>'0');

    elsif rising_edge(b_clk) then
        LAD_strobe_out <='0';
        if (LAD_inStrobe = '1') then
            if ((LAD_Address and address_MASK) = BASE_address) then

                if LAD_Address(2 downto 0)="000" then -- 000 is data write/read
                    if LAD_Write = '1' then
                        fifo_data <= LAD_datain(fifo_width-1 downto 0);
                        current_request <= push_fifo;
                    else
                        current_request <= pop_fifo;
                        LAD_dataout <= zeropad & dataout;
                        LAD_strobe_out <='1';
                    end if;
                else
                    current_request <= idle;
                end if;

                if LAD_Address(2 downto 0)="001" then -- 001 is for control/status
                    if LAD_write='0' then
                        LAD_dataout <= X"000000" & status_register;
                        LAD_strobe_out <='1';
                    end if;
                end if;

            end if; -- ends check addressing
        else
            current_request <= idle;
        end if; -- ends check input strobe
    end if; -- ends reset or bus clock
end process LAD_interface;

stimuli: process(reset, b_clk)
begin
    if reset='1' then
        current_state <=idle;
        write_fifo <='0';
        read_fifo <='0';
        datain <=(others=>'0');
        request_done <='1';
    elsif rising_edge(b_clk) then
        case current_state is
            when idle =>
                if current_request=push_fifo then
                    if fifo_full='0' then
                        datain <= fifo_data;
                        write_fifo <='1';
                        current_state <= receive_ack;
                        request_done <='0';

```

```

        end if;
    elsif current_request=pop_fifo then
        if fifo_empty='0' then
            read_fifo <='1';
            current_state <= transmit_ack;
            request_done <='0';
        end if;
    else
        write_fifo <='0';
        read_fifo <='0';
        request_done <='1';
    end if;

    when receive_ack =>
        if write_ack='1' then
            write_fifo <='0';
            current_state <= idle;
        end if;

        when transmit_ack =>
            if read_ack='1' then
                read_fifo <='0';
                current_state <= idle;
            end if;
        end case;
    end if;

end process stimuli;

end behav;

```

Figure 23 — Architecture of the Controller Designed for Integration with Framework

In order to link the host control and IP processing, the exchange process are going to be detailed for this example. Exchanges can be processed between the host and a HW module, only if the associated HW controller address is recognized, then control signals are LAD_strobe_out, LAD_instrobe, or signal LAD_write permit data and command transfers. The LAD interface is in charge of these operations, moreover it controls the stimuli process which describes the user operation scheduling, in this example the fifo memory processing. Please note that the fifo memory is designed with FPGA memory blocks.

In this example, the design of LAD interface is connected to API function calls. If the LAD_Address finished by "000" then a read or write could be performed on fifo component depending on signal LAD_Write. If the LAD_Address finished by "001" then status register is then read back, it can be used to schedule data transfer, because the status register represents the fifo status (bit 0 = fifo_empty, bit 1 = fifo_full). This structure is then easy to control with API functions or simulation functions (Figure 24 and Figure 25). In both case, functions are the command word [to transfer data](#) or [access status register](#). In this case, the fifo_ctrl_offset is equal to 0x240 (cf UoCfv3.c). Using fifo_ctrl_offset enables [to transfer data](#) to fifo memory (because LAD_Address(2 downto 0)="000"), using fifo_ctrl_offset+1 enables [to access to status register](#).

```

void test_fifo() {
    int k;
    DWORD      dDatain[16];
    DWORD      dDataout[16];
    DWORD      control[1];
    WC_RetCode rc = WC_SUCCESS;
    char operat;
    printf("\n\nPushing 16 word random data to fifo\n");
    for (k=0; k<16; k++) {
        dDatain[k]=rand(); dDatain[k]=(dDatain[k]%100);
        WC_PeRegWrite(TestInfo.DeviceNum,fifo_ctrl_offset,1, dDatain[k]);
        //Function prototype described Annexel
        WC_PeRegRead(TestInfo.DeviceNum, fifo_ctrl_offset+1, 1, control); // not used
        //Function prototype described Annexel
        printf("(%x)",control[0]);
    }
    printf("\n\nPopping 16 word data from fifo\n");
    for (k=0; k<16; k++) {
        dDataout[k]=0;
        WC_PeRegRead(TestInfo.DeviceNum,fifo_ctrl_offset,1,&dDataout[k]);
        WC_PeRegRead(TestInfo.DeviceNum, fifo_ctrl_offset+1, 1, control); // not used
    }
    for(k=0; k<8; k++) {
        printf("{Tx(%2d):%-5d, Rx:%-5d}\t{Tx(%2d):%-5d, Rx:%-5d}\n",
            2*k,dDatain[2*k],dDataout[2*k],2*k+1,dDatain[2*k+1],dDataout[2*k+1]);
    }
}
}

```

Figure 24 — API function call (From UoCfv3.c).

```

-----testfifo
report "Testing FIFO move";
--
-- push then pop
--
for i in 1 to 16 loop
    dData(0) :=i;
    WC_PeRegWrite( Cmd_Req, Cmd_Ack, device, fifo_ctrl_offset, 1, dData);
    report "Push ( " & integer'image(i) & " ) = " & Hex_To_String(dData(0));
end loop;

WC_PeRegRead( Cmd_Req, Cmd_Ack, device, fifo_ctrl_offset+1, 1, dData);
report "Fifo status: " & Hex_To_String(dData(0));

for i in 1 to 16 loop
    dData(0) := 0;
    WC_PeRegRead( Cmd_Req, Cmd_Ack, device, fifo_ctrl_offset, 1, dData);
    report "Pop ( " & integer'image(i) & " ) = " & Hex_To_String(dData(0));
end loop;

WC_PeRegRead( Cmd_Req, Cmd_Ack, device, fifo_ctrl_offset+1, 1, dData);
report "Fifo status: " & Hex_To_String(dData(0));
WC_PeRegWrite( Cmd_Req, Cmd_Ack, device, fifo_ctrl_offset, 1, dData);
report "further Push" & " = " & Hex_To_String(dData(0));
WC_PeRegRead( Cmd_Req, Cmd_Ack, device, fifo_ctrl_offset, 1, dData);
report "further Pop" & " = " & Hex_To_String(dData(0));

```

Figure 25 — Vector test for full system simulation (described in host_arch.vhd).

7.5 Calc_Sum_Product Module Controller (memory data transfer)

The test-bench file is used as a starting point and is integrated in a simple VHDL template that addresses the following:

1. Interfacing with the local address bus (LAD) through which the host system communicates with the processing element (PE) or FPGA.
2. Passing external control arguments to the “stimuli” process described in the test-bench file and letting it handle the main module.
3. Accessing the card memory if required. (Which is one of two possible cases in this example).
4. Generating an interrupt signal to indicate that the main module finished processing.
5. Assigns values to a status register to indicate the current state of the system.

The description of the controller interface (defined into calc_ctrlr.vhd) is as follows (Figure 26):

```
entity calc_hw_module_controller is
  generic (
    BASE_address: std_logic_vector(15 downto 0);
    address_MASK: std_logic_vector(15 downto 0):=X"FFE0"; -- 32 registers
    block_width: positive:=4
  );
  port (
    reset: in std_logic;
    m_clk: in std_logic;      -- module clock
    b_clk: in std_logic;      -- bus clock
    module_done: out std_logic;
    ----- memory access arbiter
    write_address: out std_logic_vector(20 downto 0);
    read_address: out std_logic_vector(20 downto 0);
    enable_write: out std_logic;
    enable_read : out std_logic;
    access_request: out std_logic;
    access_grant: in std_logic;
    Memory_Source_Data_Valid: in std_logic;
    mem_datain : in std_logic_vector(31 downto 0);
    mem_dataout : out std_logic_vector(31 downto 0);
    ----- interface with LAD bus
    LAD_instrobe: in std_logic;
    LAD_address: in std_logic_vector(15 downto 0);
    LAD_write: in std_logic;
    LAD_datain: in std_logic_vector(31 downto 0);
    LAD_dataout: out std_logic_vector(31 downto 0);
    LAD_strobe_out: out std_logic
  );
end calc_hw_module_controller;
```

Figure 26 — Controller VHDL Interface for SRAM and BRAM.

The following points should be noticed:

1. Generics: BASE_address and address_MASK provide the parameters for the address bus comparator. Fifo_size and fifo_width are parameters for the main ip module.
2. Signals: The signals are divided into 3 categories: global control (reset and the two clocks), card memory access signals, LAD interface.
3. The interface is identical to the one used for the fifo controller which is intentional to enforce design consistency and code reuse across control modules. This illustrates the authors' point that most of the effort is concentrated in the development of the ip-module not in writing code for the abstraction layer necessary to interface it with the whole system.

The controller file template is divided into 3 main *processes*:

4. LAD interface process.
5. Memory interface process.
6. IP-module interface process. (This is almost the exact form used in the test-bench "*stimuli*" process which is the point in code reuse)

The design of LAD_interface should be designed in a similar way than in 7.4 (i.e. Figure 23, 27, 29). When IP is addressed (in example address between 0x180 and 0x200), as 7.4 depending of control signals, data can be transferred or status register scanned (structure of status register are presented in _Annexe 1). The structure of the LAD_interface is presented in Figure 27. It receives informations (with address and control signals) to control the stimuli process (described in calc_ctrl.vhd) in charge of operations scheduling (data transfer, processing, results transfer). The user can control operations from the host, in this example (Figure 28), the operations realized in API function as follow:

1. Checking of the status register,
2. Data transfer from host into on-board memory,
3. After the Data transfer an interrupt is generated to the S/W layer running on the computer host. This module then control the operation of the HDL IP. Checking of end of processing (test on dDataout[0] which received bit 0 of status register),
4. Results transfer from internal memory to host.

NB: Address_mask2 is used to permit to access to internal memory (for data transfer), moreover if (LAD_Address and address_MASK) = BASE_address but ((LAD_Address and address_mask2) /= BASE_address) then you access to status register (for instance in this case when LAD_address = calc_ctrl_offset+16 as described in Figure 28).

```
LAD_interface: process(reset, b_clk)
    variable counter: integer range 0 to block_width**2-1;
begin
    if reset='1' then
        LAD_dataout <=(others=>'0');
        LAD_strobe_out <='0';
        counter:=0;
        start_read_address <= (others=>'0');
        start_write_address <= (others=>'0');
        frame_width <=0;
        frame_height <=0;
        no_frames <=0;
        module_programmed<='0';

    elsif rising_edge(b_clk) then
        LAD_strobe_out <='0';
        if (LAD_inStrobe = '1') then
            if ((LAD_Address and address_MASK) = BASE_address) then

                if ((LAD_Address and address_mask2)=BASE_address) then
                    if LAD_Write = '1' then          -- data block
                        input_array(counter) <= LAD_datain(7 downto 0);
                        counter:=(counter+1) mod (block_width**2);
                        if counter=0 then
                            no_frames <= 0;          -- 0 means block mode
                            module_programmed <='1';
                        end if;
                    else
                        LAD_dataout <= X"0000" & output_array(counter);
                        LAD_strobe_out <='1';
                        counter:=(counter+1) mod (block_width**2);
                    end if;
                end if;
            end if;
        end if;
```

```

        else
            if LAD_write='1' then      -- control block
                case LAD_Address(1 downto 0) is
                    when "00" =>
                        frame_width <= conv_integer(unsigned(LAD_datain(9 downto 2))); --
divide width by 4

                        frame_height <= conv_integer(unsigned(LAD_datain(19 downto 10)));
                        no_frames <= conv_integer(unsigned(LAD_datain(23 downto 20)));
                    when "01" =>
                        start_read_address <= LAD_datain(20 downto 0);
                    when "10" =>
                        start_write_address <= LAD_datain(20 downto 0);
                    when "11" =>
                        module_programmed <='1';
                    when others => null;
                end case;
            else
                LAD_dataout <= X"000000" & status_register;
                LAD_strobe_out <='1';
            end if;
        end if;

        end if; -- ends check addressing
    else
        module_programmed<='0';
    end if; -- ends check input strobe
end if; -- ends reset or bus clock
end process LAD_interface;

```

Figure 27 — LAD interface described in calc_ctrl.vhd.

```

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// function used to call processing (P1#) declared in UoCfv3.c
void test_calc() {
    int k;
    DWORD      dDatain[16];
    DWORD      dDataout[16];
    WC_RetCode  rc = WC_SUCCESS;
    for (k=0; k<16; k++) {
        dDatain[k]=k+1; dDataout[k]=0;
    }
    WC_PeRegRead(TestInfo.DeviceNum, calc_ctrl_offset+16, 1, dDataout);    // Test of the status
register
    printf("\nCalc status %x ",dDataout[0]);
    printf("\n\nMoving 16 word test vector\n");
    WC_PeRegWrite(TestInfo.DeviceNum, calc_ctrl_offset, 16, dDatain); // 16 data words written
    dDataout[0]=0;
    while(dDataout[0]<1) {
        WC_PeRegRead(TestInfo.DeviceNum, calc_ctrl_offset+16, 1, dDataout);    // Test of the status
register
        //printf("Calc status %x ",dDataout[0]);
    }
    WC_PeRegRead(TestInfo.DeviceNum, calc_ctrl_offset, 16, dDataout);
    for(k=0; k<16; k++) {
        printf("Tx(%2d):%-5d, Rx:%-5d\n",
            k,dDatain[k],dDataout[k]);
    }
}
}

```

Figure 28 — Example to transfer data and receive data with computer host (described in UoCfv.c).

The full code for the controller with SRAM connection is represented in Figure 29. The controller is made to work in two modes: (1) block data mode and (2) memory addressing mode. This is dependent on the programmed number of frames. If it is zero then the *mem_generate_address* process assumes that a data block has been written directly via the LAD bus and just signals the *stimuli* process. On the other hand, if the number of frames is at least one then it is assumed that at least one whole YUV frame has been written to the SRAM and the *mem_generate_address* process begins to divide each frame in the memory into macro blocks of number of pixels per side equal to *block_width*.

In the block data mode, the host would select data block mode simply by writing the correct amount of data to the first (block_width^2) addresses in the allocated address space for this module. The block size is expected to be much smaller than the actual size.

The memory addressing mode is selected by programming the SRAM read_start, write_start addresses and the number of frames passed to memory by DMA.

The code for this controller might seem complicated, however, the structure is consistent with the idea of code reuse and all the complexity is actually concentrated in the part for calculating how to partition a raster stored YUV image into macro blocks.

```

-----
-- Memory access and module controller
-----

library IEEE;
  use IEEE.std_logic_1164.all;
  use IEEE.std_logic_unsigned.all;
  use IEEE.std_logic_arith.all;

entity calc_hw_module_controller is
  generic(
    BASE_address: std_logic_vector(15 downto 0);
    address_MASK: std_logic_vector(15 downto 0) := X"FFE0"; -- 32 registers
    block_width: positive:=4
  );
  port(
    reset: in std_logic;
    m_clk: in std_logic;      -- module clock
    b_clk: in std_logic;      -- bus clock
    module_done: out std_logic;
    ----- memory access arbiter
    write_address: out std_logic_vector(20 downto 0);
    read_address: out std_logic_vector(20 downto 0);
    enable_write: out std_logic;
    enable_read : out std_logic;
    access_request: out std_logic;
    access_grant: in std_logic;
    Memory_Source_Data_Valid: in std_logic;
    mem_datain : in std_logic_vector(31 downto 0);
    mem_dataout : out std_logic_vector(31 downto 0);
    ----- interface with LAD bus
    LAD_instroke: in std_logic;
    LAD_address: in std_logic_vector(15 downto 0);
    LAD_write: in std_logic;
    LAD_datain: in std_logic_vector(31 downto 0);
    LAD_dataout: out std_logic_vector(31 downto 0);
    LAD_strobe_out: out std_logic
  );
end calc_hw_module_controller;

```

```

architecture behav of calc_hw_module_controller is
    type memory8bits is array (natural range <>) of std_logic_vector(7 downto 0);
    type memory16bits is array (natural range <>) of std_logic_vector(15 downto 0);
    type system_modes is (idle, receive_up, receive_dn, calculating, transmit_up, transmit_dn);

    constant address_mask2:
std_logic_vector:=conv_std_logic_vector(unsigned(address_MASK)+block_width**2,16);

    signal input_array: memory8bits(0 to block_width**2-1);
    signal output_array: memory16bits(0 to block_width**2-1);
    signal current_state: system_modes;

    signal status_register: std_logic_vector(7 downto 0);
    signal control_register: std_logic_vector(7 downto 0);
    signal initiate_processing: std_logic;

    --signal reset, m_clk, tb_clk: std_logic;
    signal module_ready: std_logic;
    signal input_available, output_available: std_logic;
    signal async_in, async_out: std_logic;
    signal datain: std_logic_vector(7 downto 0);
    signal dataout: std_logic_vector(15 downto 0);
    ---- memory access
    signal start_read_address : std_logic_vector(20 downto 0);
    signal start_write_address : std_logic_vector(20 downto 0);
    signal frame_width: integer range 0 to 255;
    signal frame_height: integer range 0 to 511;
    signal no_frames: integer range 0 to 15;
    ----
    signal module_programmed: std_logic;

    signal mem_gen_state: integer range 0 to 15; -- state machine of up to 16 states

    signal i: integer range 0 to 1023;          --10 bits
    signal j: integer range 0 to 255;          --4 bytes at a time
    signal ii: integer range 0 to block_width-1;
    signal jj: integer range 0 to (block_width/4-1); -- block bytes
    signal voffset: integer range 0 to 2**12-1;
    signal fn: integer range 0 to 22;
    signal YUV: std_logic;
    signal read_addr, write_addr : integer range 0 to 2**20-1;

    component calc_sum_product
    generic(no_of_inputs: positive:=64);
    port(
        reset:          in std_logic;
        clock:           in std_logic;
        module_ready:    out std_logic;
        input_available: in std_logic;
        datain:          in std_logic_vector(7 downto 0);
        output_available: out std_logic;
        dataout:         out std_logic_vector(15 downto 0);
        async_in:        in std_logic;
        async_out:       out std_logic
    );
    end component;

begin

    U_calc_sum_product: calc_sum_product

```

```

generic map (no_of_inputs => block_width**2)
port map (
    reset => reset,
    clock => m_clk,
    module_ready => module_ready,
    input_available => input_available,
    datain => datain,
    output_available => output_available,
    dataout => dataout,
    async_in => async_in,
    async_out => async_out
);

status_register <=(0=> module_ready, 1=> output_available, others=>'0');

LAD_interface: process(reset, b_clk)
    variable counter: integer range 0 to block_width**2-1;
begin
    if reset='1' then
        LAD_dataout <=(others=>'0');
        LAD_strobe_out <='0';
        counter:=0;
        start_read_address <= (others=>'0');
        start_write_address <= (others=>'0');
        frame_width <=0;
        frame_height <=0;
        no_frames <=0;
        module_programmed<='0';

    elsif rising_edge(b_clk) then
        LAD_strobe_out <='0';
        if (LAD_inStrobe = '1') then
            if ((LAD_Address and address_MASK) = BASE_address) then

                if ((LAD_Address and address_mask2)=BASE_address) then
                    if LAD_Write = '1' then -- data block
                        input_array(counter) <= LAD_datain(7 downto 0);
                        counter:=(counter+1) mod (block_width**2);
                        if counter=0 then
                            no_frames <= 0; -- 0 means block mode
                            module_programmed <='1';
                        end if;
                    else
                        LAD_dataout <= X"0000" & output_array(counter);
                        LAD_strobe_out <='1';
                        counter:=(counter+1) mod (block_width**2);
                    end if;
                else
                    if LAD_write='1' then -- control block
                        case LAD_Address(1 downto 0) is
                            when "00" =>
                                frame_width <= conv_integer(unsigned(LAD_datain(9 downto 2)));
                                frame_height <= conv_integer(unsigned(LAD_datain(19 downto 10)));
                                no_frames <= conv_integer(unsigned(LAD_datain(23 downto 20)));
                                when "01" =>
                                    start_read_address <= LAD_datain(20 downto 0);
                                when "10" =>
                                    start_write_address <= LAD_datain(20 downto 0);
                                when "11" =>
                                    module_programmed <='1';
                        end case;
                    end if;
                end if;
            end if;
        end if;
    end if;
end process;

```

```

        when others => null;
    end case;
    else
        LAD_dataout <= X"000000" & status_register;
        LAD_strobe_out <='1';
    end if;
end if;

    end if; -- ends check addressing
else
    module_programmed<='0';
    end if; -- ends check input strobe
end if; -- ends reset or bus clock
end process LAD_interface;

stimuli: process(reset, b_clk)
    variable counter: integer range 0 to block_width**2-1;
begin
    if reset='1' then
        counter :=0;
        current_state <=idle;
        input_available <='0';
        datain <=(others=>'0');
    elsif rising_edge(b_clk) then
        case current_state is
            when idle =>
                if initiate_processing='1' then
                    if module_ready='1' then
                        input_available <='1';
                        current_state <= receive_up;
                    end if;
                end if;
            when receive_up =>
                if async_out='0' then
                    async_in <='1'; -- latch
                    datain <=input_array(counter);
                    current_state <= receive_dn;
                end if;
            when receive_dn =>
                if async_out='1' then
                    async_in <='0';
                    counter :=((counter+1) mod (block_width**2));
                    if counter=0 then
                        input_available <='0';
                        current_state <= calculating;
                    else
                        current_state <= receive_up;
                    end if;
                end if;
            when calculating =>
                if output_available='1' then
                    current_state <= transmit_up;
                end if;
            when transmit_up =>
                if async_out='1' then -- latch
                    output_array(counter) <= dataout;
                    async_in <='1'; --acknowledged
                    current_state <= transmit_dn;

```

```

        end if;
    when transmit_dn =>
        if async_out='0' then
            async_in <='0';
            counter :=((counter+1) mod (block_width**2));
            if counter=0 then
                current_state <=idle;
            else
                current_state <=transmit_up;
            end if;
        end if;
    end case;
end if;

end process stimuli;
-----
memory_generate_address: process(reset,b_clk)
    variable buffercounter: integer range 0 to (block_width**2-1);
    variable base_ad1 : integer range 0 to 2**20-1;
    variable bufferfilled, more_addressing: std_logic;
begin
    if (reset='1') then
        -- This module will access memory
        enable_write <='0';
        enable_read <='0';
        access_request <='0';

        initiate_processing <='0';

        mem_gen_state<=0;
        module_done <='1';

        i<=0; j<=0; ii<=0; jj<=0;
        voffset<=0;
        YUV<='0'; fn<=0;
        read_addr <=0; write_addr <=0; base_ad1:=0;
        buffercounter:=0;
        bufferfilled:='0'; more_addressing:='1';

    elsif rising_edge(b_clk) then
        case mem_gen_state is
            when 0 => --initialization
                if module_programmed='1' then
                    if no_frames=0 then -- block mode
                        initiate_processing <='1';
                    else
                        module_done<='0';
                        mem_gen_state<=1; -- generate address
                        i<=0; j<=0; ii<=0; jj<=0;
                        voffset <=0;
                        YUV<='0'; fn<=0;
                        base_ad1:=conv_integer(unsigned(start_read_address));
                        read_addr<=base_ad1;
                        write_addr<=conv_integer(unsigned(start_write_address));
                        buffercounter:=0;
                        bufferfilled:='0'; more_addressing:='1';
                        enable_read<='1';
                        enable_write<='0';
                        access_request<='1';
                    end if;
                end if;
            end if;
        end case;
    end if;
end process;

```

```

        else
            enable_read<='0'; enable_write<='0';
            initiate_processing <='0';
            module_done<='1';
            access_request<='0';
        end if;
    when 1 =>    -- remains stuck till gets memory access grant
        if access_grant='1' then
            mem_gen_state<=2;
        end if;
    when 2 =>    --states 2, 3, 4 for filling an input buffer
        mem_gen_state <= 3;
        if (Memory_Source_Data_Valid='1') then
            input_array(buffercounter*4+3) <= mem_datain(31 downto 24);
            input_array(buffercounter*4+2) <= mem_datain(23 downto 16);
            input_array(buffercounter*4+1) <= mem_datain(15 downto 8);
            input_array(buffercounter*4) <= mem_datain(7 downto 0);
            if (buffercounter < (block_width**2/4-1)) then
                buffercounter:= (buffercounter+1) ;
            else
                buffercounter:=0;
                bufferfilled:='1';
            end if;
        end if;
        if more_addressing='1' then
            if(jj<(block_width/4-1)) then
                jj<=(jj+1);
            else
                jj<=0;
                if YUV='0' then
                    voffset <= voffset+frame_width;
                else
                    voffset <= voffset+frame_width/2;
                end if;
                if (ii<block_width-1) then
                    ii<=ii+1;
                else
                    ii<=0;
                    more_addressing:='0';
                end if;
            end if;
        end if;
    when 3 =>
        read_addr <= (base_adl+voffset+jj+j);
        mem_gen_state<=4;
    when 4 =>
        if (bufferfilled='1') then
            mem_gen_state<=5; -- enough reading, process
            initiate_processing <='1';
            enable_read <= '0';
        else
            mem_gen_state<=2;    -- end cycle waste
        end if;

    when 5 =>

        mem_gen_state<=6;
        if ( ((j+block_width/4)<frame_width and YUV='0') or
            ((j+block_width/4)<(frame_width/2))) then
            j <= j+block_width/4;

```

```

        else
            j<=0;
            base_ad1:=base_ad1+voffset;
            if (i<frame_height-block_width-1) then
                i<=i+block_width; --assumption
            else
                i<=0; fn<=fn+1;
                YUV <= not YUV;
            end if;
        end if;

when 6 => --go process
    initiate_processing<='0';
    if current_state=idle then
        mem_gen_state<=7;
    end if;

when 7 => --writing results to memory
    enable_write<='1';
    mem_dataout <= output_array(buffercounter*2) & output_array(buffercounter*2+1);
    write_addr <= write_addr+1;
    if buffercounter<(block_width**2/2-1) then
        buffercounter:=(buffercounter+1);
    else
        buffercounter:=0;
        voffset<=0;
        mem_gen_state<=8;
    end if;

when 8 =>
    enable_write<='0';
    jj<=0;
    if (fn=(no_frames*2)) then
        mem_gen_state<=10;
    else
        mem_gen_state<=9;
        bufferfilled:='0'; more_addressing:='1';
    end if;
when 9 =>
    enable_read<='1';
    read_addr <= base_ad1+j;
    mem_gen_state<=3;
when 10 =>
    if (Memory_Source_Data_Valid='0') then
        mem_gen_state<=0;
    end if;
when others => null;
end case;
end if;

end process memory_generate_address;
read_address<=conv_std_logic_vector(read_addr,21);
write_address<=conv_std_logic_vector(write_addr,21);

end behav;

```

Figure 29 — Architecture of Second Controller Designed for Integration with Framework.

In Figure 29 the orange lines shows the parameter that user needs to transfer to enable user IP to access to data stored into SRAM. As described into `calc_ctrlr.vhd`, the input data are loaded from the SRAM, then processed, and finally the results transfers into SRAM, a new processing can then start.

For example described into host_arch.vhd:

```
dData(0) := 16#101820#;      -- Mode = 00, frame_width = 0000 1000,
                               -- frame_height = 00 0000 1000,

-- no_frames = 0001

dData(1) := 16#0000#;      -- Mode = 01,

-- start_read_address = 0000 0000 0000 1000 0000

dData(2) := 16#0081#;      -- Mode = 10,

-- start_write_address = 0000 0000 0000 1000 0001

dData(3) := 16#0003#;      -- Mode = 11,

-- module_programmed = '1';
```

The code for this controller might seem complicated, however, the structure is consistent with the idea of code reuse and all the complexity is actually concentrated in the part for calculating how to partition a raster stored YUV image into macro blocks.

7.5.1 Adding a wrapper for a Verilog module

For instantiating a Verilog ip-core, the wrapper would be exactly the same as in the above two examples. To write the component part we may use the utility “vgencomp” which translates the interface of the Verilog component to VHDL so that it can be instantiated from within a higher level VHDL system. This can also be done manually.

7.5.2 Integrating module controllers within the PE system

The next step is to add the previously described blocks within the PE (processing element) architecture. This involves instantiation of each of the two controllers and connecting them within the system interrupt-chain.

This involves adding/modifying the following code fragments to the “PE system” file:

1. Library declarations.
2. Constants for generics and interrupt signal.
3. Component declaration.
4. VHDL configuration.
5. Component instantiation.
6. Connecting interrupt signal.
7. Updating simulation and synthesis project files.

The details of these steps are as follows

7.5.3 Library declarations

This part is for adding the library declaration for the components and the component controllers defined in the previous sections. The next fragment is added to the library section in the “PE system” VHDL file. It is for 3 blocks: block-move, fifo and calc.

```
////////////////////////////////////
// The processing element pe is constituted of tree element
// (instanted in the pe_arch.vhd) first example (no_of_hwas =3):
// 1) calc_hw_module_controller,
```

```
// 2) fifo_hw_module_controller
// 3) bm_hw_module_controller
```

```
-----
-- hardware accelerator libraries
library calc_lib;
    use calc_lib.all;
library fifo_lib;
    use fifo_lib.all;
library bm_lib;
    use bm_lib.all;
```

Figure 30 — Example of library declarations.

7.5.4 Constants for generics and interrupt signals

For every block we add generics for the base address and the address mask and any other parameters. The optional interrupt signal is hw?_done. The choice of the base address is arbitrary but address spaces should not overlap. The constant “no_of_hwas “ is used to define the memory and LAD bus multiplexers so it must be set correctly.

```
constant no_of_hwas: positive:=3; -- update

constant hw1_BASE_ad    : std_logic_vector(15 downto 0) := x"0180";
constant hw1_MASK      : std_logic_vector(15 downto 0) := x"FFE0"; -- 32 registers
signal hw1_done: std_logic;

constant hw2_BASE_ad    : std_logic_vector(15 downto 0) := x"0100";
constant hw2_MASK      : std_logic_vector(15 downto 0) := x"FFE0"; -- 32 registers
signal hw2_done: std_logic;

constant hw3_BASE_ad    : std_logic_vector(15 downto 0) := x"0140";
constant hw3_MASK      : std_logic_vector(15 downto 0) := x"FF40"; -- 64 registers
signal hw3_done: std_logic;
```

Figure 31 — Constant declarations. (described in pe_arch.vhd).

7.5.5 Component declaration

This is a step before component instantiation. It is added to the architecture section before the “begin” keyword.

For example the component instantiation for “blockmove” is as follows.

```

component bm_hw_module_controller
  generic(
    BASE_address: std_logic_vector(15 downto 0);
    address_MASK: std_logic_vector(15 downto 0):=X"FF40"; -- 64 registers
    words_per_block: positive:=64
  );
  port(
    reset: in std_logic;
    m_clk: in std_logic;      -- module clock
    b_clk: in std_logic;      -- bus clock
    module_done: out std_logic;
    ----- memory access arbiter
    write_address: out std_logic_vector(20 downto 0);
    read_address: out std_logic_vector(20 downto 0);
    enable_write: out std_logic;
    enable_read : out std_logic;
    access_request: out std_logic;
    access_grant: in std_logic;
    Memory_Source_Data_Valid: in std_logic;
    mem_datain : in std_logic_vector(31 downto 0);
    mem_dataout : out std_logic_vector(31 downto 0);
    ----- interface with LAD bus
    LAD_instrobe: in std_logic;
    LAD_address: in std_logic_vector(15 downto 0);
    LAD_write: in std_logic;
    LAD_datain: in std_logic_vector(31 downto 0);
    LAD_dataout: out std_logic_vector(31 downto 0);
    LAD_strobe_out: out std_logic
  );
end component;

```

Figure 32 — Component instantiations of “blockmove”.

7.5.6 VHDL configuration statements

The following simple VHDL configuration statements should be added before component instantiation. More elaborate types of VHDL configuration can be used.

The code for the three modules is as follows:

```

-----VHDL configuration
for u_hw1: calc_hw_module_controller use entity calc_lib.calc_hw_module_controller;
for u_hw2: fifo_hw_module_controller use entity fifo_lib.fifo_hw_module_controller;
for u_hw3: bm_hw_module_controller use entity bm_lib.bm_hw_module_controller;
-----

```

Figure 33 — Example of VHDL configuration statements.

7.5.7 Component instantiation

This involves a generic map and a port map. Note that mapping to the LAD multiplexer starts with “4” not zero because there are other 4 bus clients in the system that use the numbers zero to three. Thus, the first “hwa” is bus client number four. However, for memory access, the numbering starts from zero because the memory multiplexer is connected only to the user modules.

Component instantiation for the modules is as follows: