4th FPGAworld CONFERENCE

SEPTEMBER 11 AND 13
LUND AND STOCKHOLM, SWEDEN

EDITORS

Lennart Lindh and Vincent J. Mooney III

PROCEEDINGS 2007

The FPGAworld Conference addresses all aspects of digital and hardware/software system engineering on FPGA technology. It is a discussion and network forum for researchers and engineers working on industrial and research projects, state-of-the-art investigations, development and applications. The proceedings contain the academic presentations and some of the industrial presentations (from 2007); for more information see (www.fpgaworld.com/conference).

ISSN 1404-3041 ISRN MDH-MRTC-215/2007-1-SE

SPONSORS

Additional copies of 2007 or prior Proceedings may be found at www.FPGAworld.com or at Malardalens University library (www.mdh.se), ISSN 1404-3041 ISRN MDH-MRTC-215/2007-1-SE

2007 PROGRAM COMMITTEE

General Chairman's Message

The FPGAworld program committee welcomes you to the 4th FPGAworld conference. This year's conference is held in Electrum-Kista, Stockholm, and Lund, Ideon, Sweden. We hope that the conferences provide you with much more then you expected. This year it is the third time we have academic reviewed papers; this is an important step to incorporate the academic community into the conference program. Due to the high quality, 6 out of the 10 papers submitted this year were accepted.

We will try to balance academic and industrial presentations (Stockholm), exhibits and tutorials to provide a unique chance for our attendants to obtain knowledge from different views. This year we have the strongest program in FPGAworld´s history. Also, industrial papers were submitted then accepted.

The FPGAworld 2007 conference is bigger than the FPGAworld 2006 conference.

All are welcome to submit industrial/academic papers, exhibits and tutorials to the conference, both from academic and industrial backgrounds. Together we can make the FPGAworld conference exceed even above our best expectations!

Please check out the website (http://fpgaworld.com/conference/) for more information about FPGAworld 2007. In addition, you may contact David Källberg (david@fpgaworld.com) for more information.

Lennart Lindh

General Program Chair

Preliminary programme for FPGAworld 2007 Stockholm

8:30 – 12:30

08:30 - 09:00	Registration
09:00 - 09:15	Conference Opening Lennart Lindh, FPGAworld
09:15 - 10:00	**Keynote Session** Research Trends in Hardware/Software Codesign of Embedded Operating Systems for FPGAs Dr. Vincent J. Mooney III Georgia Institute of Technology, USA
10:00 - 10:30	Coffee Break
10:30 - 11:30	Exhibitors Presentations
11:30 - 12:30	Lunch Break, Sponsored by Mentor Graphics

Preliminary programme for FPGAworld 2007 Stockholm

<u>12:30 – 19:00</u>

	Chair Fredrik Lång EBV Elektronik	Chair Vincent J Mooney, Georgia Institute of Technology	Chair Kim Petersén, HDC	
12:30 - 14:30	Session A1 VHDL is still the best language for FPGA verification Session A2 Development of complex FPGA applications require new design technologies Session A3 Digital Data Processor for a Synthetic Aperture Radar Session A4 Thinking Ouitside The Flow: Creating Customized Backend Tools For Xilinx Based Designs *	Session B1 OBSAI RP3-01 6.144 Gbps Interface Implementation * Session B2 A Dynamically Reprogrammable CSA-Generic Platform Architecture * Session B3 Application of ASM++ methodology on the design of a DSP processor * Session B4 The Effect of Dependence Graphs' Size and Complexity, in the Implementation of Processor Arrays on FPGA Devices *	Session C1 Product Presentation The Dini Group Session C2 Tools: Keeping up with the FPGA Challenge Synplicity Session C3 Product Presentation Atmel Session C4 Product Presentation Digitas AS, Data Respons	Session D2 Altera Innovate Nordic Contest
14:30 15:00	Coffee Break			
	Chair Ola Wall Synplicity	Chair Lena Engdahl, Altera	Chair Lars Sageryd EBV Elektronik	
15:00 - 16:30	Session A5 Atmel CAP: ARM processors with a dedicated interface to Altera/Xilinx FPGAs Session A6 Next generation ARM Industrial Standard processor for FPGA designs Session A7 Leveraging spreadsheets for integrating FPGA Integration in a Board design flow	Atmel Product Demonstration Workshop	Session C5 SystemVerilog for Design and Verification Session C6 PowerExploration made possible by C/C++ Synthesis Session C7 Product Presentation Xilinx	Session D3 Altera Innovate Nordic Contest
16:30	Exhibition & Snacks Sponsored by Altera			

`*Paper part of Academic Program`

Presentation; "Busy Generation in a large Trigger Based Data Acquisition System" will be schedule in the final programme *

Programme FPGAworld 2007 Lund

08:30 09:00	Lund, Registration	
09:00 09:15	Conference opening, Lennart Lindh and David Källberg, FPGAworld	
09:15 - 10:00	Keynote Session Redefining the FPGA for the Next Generation Paul Evans, European Marketing Manager, Xilinx	
10:00 10:30	Coffee Break	
10:30 - 11:30	Exhibitors Presentation	
11:30 - 13:00	Lunch Break	
13:00 - 14:30	Session Chair Tryggve Mathiesen, BitSim	Session Chair Lars Sageryd, EBV Elektronik
	Session A1 The GRLIB VHDL IP library and its usage in LEON3 based developments Session A2 Leveraging spreadsheets for integrating FPGA Integration in a Board design flow Session A3 Development of complex FPGA applications require new design technologies	Session C1 Product Presentation The Dini Group Session C2 Tools: Keeping up with the FPGA Challenge, Synplicity Session C3 Product Presentation Atmel
14:30 15:00	Coffee Break	
15:00 - 16:00	Session Chair Per Henricsson, Elektroniktidningen	Session Chair Ola Wall, Synplicity
	Session A4 Atmel CAP: ARM processors with a dedicated interface to Altera/Xilinx FPGAs Session A5 Next generation ARM Industrial Standard processor for FPGA designs	Session C4 SystemVerilog for Design and Verification Session C5 PowerExploration made possible by C/C++ Synthesis
16:00	Exhibition & Snacks	

Exhibition 2007

Stockholm - Kista

1. BitSim
2. Acal
3. Arrow
4. Altera - Innovate Nordic
5. Synplicity
6. EBV Elektronik
7. Avnet Silica, XILINX
8. EWE AB
9. The Dini Group
10. Actel
11. The Mathworks
12. Digitas AS
13. Atmel

Lund - Ideon

1. BitSim
2. Acal
3. Arrow
4. Synplicity
5. EBV Elektronik
6. Avnet Silica, XILINX
7. EWE AB
8. The Dini Group
9. Gaisler Research
10. Actel
11. The Mathworks
12. Atmel

TABLE OF ACADEMIC PROCEEDINGS CONTENTS

OBSAI RP3-01 6.144 Gbps Interface Implementation

Christian F. Lanzani[*]

RADIOCOMP ApS

and

Technical University of Denmark, NET•COM•DTU

Abstract

A cost-efficient digital hardware implementation for high speed RP3-01 serial interface at 6.144 Gbps is presented for OBSAI compliant BTS systems. Such data rate represents a 8x increment of the lowest RP3-01 rate and it might enable transmission of multiple wide-band carriers across multi-node RRH network infrastructures for use in WiMAX 802.16e-2005 and 3GPP LTE wireless applications. The implementation is based on Altera EP2SGX90FF1508 FPGA device, which transceivers handles the electrical physical layer. The optical physical layer is implemented by Finisar SFP+ FTLX8571D3BCL devices. The upper layers of the RP3-01 protocol stack are implemented using Radiocomp's OBSAI RP3-01 IP core. The implementation is backward compatible with existing RP3-01 line rates and the design methodology of the IP core makes it usable also on lower cost FPGA families. The FPGA design flow is based on Altera Quartus II programming environment for simulations, synthesis and mapping onto the target device. The system's performance is measured with internal BER counters and eye diagram evaluation using Agilent 86105 DCA.

Keywords: OBSAI RP3-01, 6.144 Gbps, Remote Radio Head, High Speed.

INTRODUCTION

New approaches are recently being introduced in wireless infrastructure networks for distributing and de-centralizing Base Transceiver Station (BTS) nodes. Such approaches aims at reducing the relative capital (CAPEX), operating (OPEX) expenditures and the development efforts while increasing system performances and flexibility by defining a modular and standardized internal BTS architecture and interfaces. The BTS is an integral part of the radio access network and is the bridge between the handset and the wireless infrastructure core network. In a distributed BTS network architecture, the radio module is remote relative to the channel card (base-band processing) and communicates with the channel card via a standardized digital optical interface. Distance ranges over the fiber vary from indoor coverage up to a few kilometers. This is done to improve site performances and reduce site footprint as well as enable high efficiency sector coverage with multiple remote radio nodes.

The Open Base Station Architecture Initiative (OBSAI) [1] defines a modular architecture with standardized functions split and inter-module interfaces into a modern wireless BTS. OBSAI defines the Remote Radio Head (RRH) concept as a radio module connected to the base-band through the Reference Point 3-01 (RP3-01) interface, as defined in [2]. The RP3-01 interface realizes a high speed optical communication link between the Local Converter (LC) module and a RRH. This interface is used to provide bi-directional transfer of digitized base-band radio data together with control and air-interface synchronization information [2].

Emerging wireless standards like WiMAX 802.16e-2005 [4] and 3GPP Long Term Evolution (LTE) [5] enhance throughput and radio signal quality performance also by defining wide-band radio channels and advanced modulation schemes for uplink and downlink channels. Currently, RP3-01 interface definitions set the line rates up to 3.072 Gbps [1], which is a bottle-neck to provide wide-band carrier support in multi-node RRH setups. The impact that a 6.144 Gbps data rate increment will have on the existing RP3 standard specifications is as today under consideration as a number of challenges exist in defining a cost-efficient solution. These challenges are the identification of suitable technologies for the physical layer, the protocol design choices, the definition of electrical specifications for compliance and the evaluation of the system performances and limitations for such interfaces.

This work describes a suitable physical layer technology and the design choices required to demonstrate an optical 6.144 Gbps RP3-01 interface which requires minimal changes to the existing OBSAI standard. The design is targeting an FPGA-based implementation

[*]email: cla@com.dtu.dk

usable for both BTS and RRH applications. The test setup consists of a full-duplex point-to-point optical communication at 6.144 Gbps between two Altera Stratix II GX Audio/Video (SIIGXAV) evaluation boards using Enhanced Small Form-Factor pluggable (SFP+) transceiver modules and a RP3-01 engine. The RP3 bus clock and frame synchronization (SYNC) signals [3] are provided externally and the signal's quality is measured via internal Bit Error Rate (BER) counters in the design blocks and the signal eye diagram using Agilent 86105 equipment.

This paper is organized as follows: *Section II* outlines the bandwidth increment requirements for multi-node wide-band carriers RRH networks. *Section III* describes RP3-01 functional architecture and blocks. *Section IV* briefly describes the SFP+ transceiver technology benefits and shows measurements of the optical signal performances. *Section V* describes briefly the GX transceivers features shows measurements of their electrical performances. *Section VI* illustrates the RP3-01 design considerations for 6.144 Gbps and for low cost FPGA-based implementations. *Section VII* illustrates the hardware test setup and BER measurement results. The conclusions are given in *Section VIII.*

II - BANDWIDTH REQUIREMENTS

In [4] a number of Orthogonal Frequency Division Multiple Access (OFDMA) profiles and radio channel bandwidths up to 28 MHz are defined in WiMAX [5]. 3GPP LTE [5] also supports a number of profiles and radio channels with bandwidths up to 20 MHz. Each channel bandwidth is associated with its base-band digital sample rate, where the samples are given in In-phase (I) and Quadrature (Q) format of 16 bits each at the RP3-01 stage [2]. For such wireless standards, performance optimization on the radio link can be achieved by exploiting advanced Multiple-Input Multiple-Output (MIMO) antenna techniques[1], which is increasing the number of antenna carriers required into a single radio node. Site coverage optimization can be obtained by exploiting multiple RRH in a number of possible topologies, like daisy-chaining, ring or tree-and-branch.
RP3-01 link has a limited support in terms of bandwidth available to transport multiple wide-bad radio carriers signals across multiple RRH nodes using multiple virtual RP3 links [2]. Table 1 shows an example of how many (X) wide-band carriers at 20 MHz can be transported over a 3.072 Gbps virtual RP3 link [2]. In this case the whole link is allocated for the same standard data and we assume that the RP3 virtual channel is specified by parameters (index,module) with value (0,1) [2].

[1]MIMO technology configurations offers significant increases in data throughput and link range without requiring additional channel bandwidth or transmit power, giving thus higher spectral efficiency and link reliability reduced fading.

	WiMAX	**LTE**
Line Rate	3.072 Gbps	3.072 Gbps
Channel BW	20 MHz	20 MHz
Sample Rate	22.4 Msps	30.72 Msps
Carriers (X)	2	2

Table 1: Amount of 20 MHz carriers to fit into a virtual RP3 link at 3.072 Gbps.

In case of modern radio setups using MIMO techniques, the amount of carriers (X) [2] required per RRH node can be 2 (2x2 MIMO) or 4 (4x4 MIMO) for common configurations. Thus current OBSAI RP3-01 line rate definitions[2] are not sufficient to provide bandwidth enough[3] to support multi wide-band carriers across multiple RRH nodes as shown in Table 1. A 6.144 Gbps line rate would allow the bandwidth increment necessary for supporting multi node RRH network architectures.

III - IMPLEMENTATION ARCHITECTURE

The functional architecture of the RP3-01 interface design is represented in Fig.1, showing the split between the physical and the higher layers (Application, Transport, Data Link).

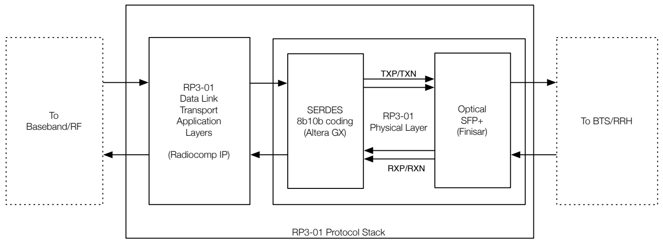


Figure 1: OBSAI RP3-01 6.144 Gbps architecture and data-path.

In this implemenation the RP3-01 physical layer consists of high speed Stratix II GX transceivers and of optical SFP+ transceivers, while the logical layers are part of the Radiocomp's IP. The higher layer (Application) can be interfaced with Base-band or RF cards, while the lower layer (Data Link) is interfaced with the

[2]Existing OBSAI RP3-01 rates are 768 Mbps, 1536 Gbps and 3.072 Gbps.
[3]Mapping of WiMAX and LTE digitized radio samples into the OBSAI RP3-01 link is done using (index,module) and dual bit maps algorithm as defined in [2].

physical layer. The whole high speed design is hosted from the SIIGXAV evaluation board.

IV - SFP+ OPTICAL TRANSCEIVER TECHNOLOGY

The SFP (Small Form-Factor Pluggable) compact optical transceivers are commonly used in optical communications for both telecommunication and data communication applications and they are Commercial-Off-Ther-Shelf (COTS) available devices with capability for data rates up to 4.25 Gbps. The latest generation of such transceivers, called Enhanced Small Form-Factor Pluggable (SFP+), has been designed within the same form-factor for higher data rates up to 10 Gbps, for lower power consumption, less complexity, and as a lower cost alternative to the 10-Gbps XFP form factor[4]. The measurements have been done using FTLX8571D3BCL 10Gbps 850nm Multimode Datacom SFP+ Transceiver. Optimized results may be achieved in the near future by using the FTLF8528P2BNV 8.5 Gbps Short-Wavelength SFP+ transceivers. In Fig.4 and Fig.5 are shown the eye diagram measurements of RP3-01 optical signals at 3.072 Gbps and 6.144 Gbps rates respectively on the SIIGXAV. The signal measured consist of valid RP3-01 frame structure[5] with data in every RP3-01 message slot.

These measurements are taken using Agilent 89105 DSO equipment over 1 m distance with multimode 850 nm fiber. The instrument has been configured with a 153.6 MHz trigger reference locked to the transmitted data, which consist of valid RP3-01 data messages. At 3.072 Gbps rate a 3.125 Gbps rate filter is applied. At 6.144 Gbps a 9.125 Gbps rate filter is applied, since the 6.250 Gbps filter option was not currently installed in the instrument.

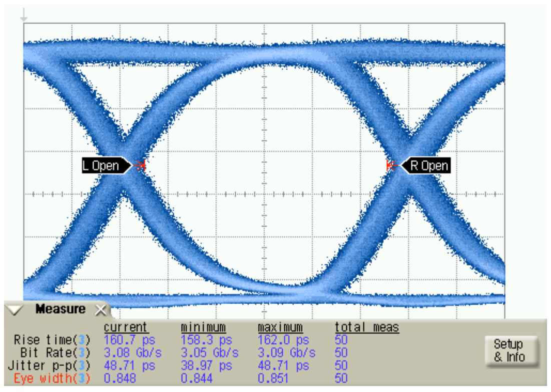
In [2] the indicative minimum values for eye mask compliance relative to the eye width for transmitter and receiver are 0.656 UI and 0.45 UI respectively. The eye diagram measurement at 3.072 Gbps rate shows a peak-to-peak jitter value at 48.71 ps with a eye width value at 0.848 Unit Interval (UI).

The eye diagram measurement at 6.144 Gbps rate shows a peak-to-peak jitter value at 60.16 ps with a eye width value at 0.685 UI.

V - HIGH SPEED SERDES TECHNOLOGY

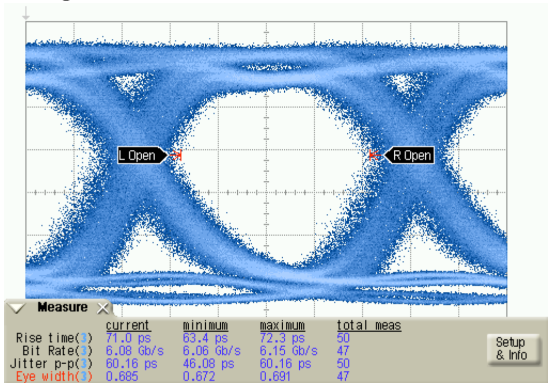
The physical electrical layer is implemented by the Altera Stratix II GX device family, which combines up to 20 duplex channels capable of operating between 600 Mbps and 6.375 Gbps into a single FPGA. The low power transceivers offer optimal signal integrity and provide a number of features such as Dynamic Pre-emphasis, Equalization and Adaptive Equalization to simplify board design. The transceivers also provide optimal jitter performance, meaning they comply electrically with the majority of serial standards being used today, including many of the telecom standards. For OBSAI RP3/RP3-01 applications, they offer compliance to the XAUI electrical interface specified in Clause 47 of IEEE 802.3ae-2002 [10] up to 3.072 Gbps and to the Common Electrical I/O (CEI) for both the Short Reach and Long Reach 6.25 Gbps standards (CEI-6G-SR and CIE-6G-LR) [6], which is a candidate standard recommendations for applications above 3.072 Gbps.

The GX transceiver includes dedicated digital building blocks to support the PCS-sublayer of many key protocols, this means many of the physical layers of a protocol can be built inside the transceiver. In the case of OBSAI RP3/RP3-01, the 8b10b encoding and word alignment blocks are embedded in the transceiver block and do not need to use dedicated FPGA logic.



```
                 current     minimum     maximum     total meas
   Rise time(3)  160.7 ps    158.3 ps    162.0 ps    50
    Bit Rate(3)  3.08 Gb/s   3.05 Gb/s   3.09 Gb/s   50
  Jitter p-p(3)  48.71 ps    38.97 ps    48.71 ps    50
   Eye width(3)  0.848       0.844       0.851       50
```

Figure 2: 3.072 Gbps RP3-01 optical signal.



```
                 current     minimum     maximum     total meas
   Rise time(3)  71.0 ps     63.4 ps     72.3 ps     50
    Bit Rate(3)  6.08 Gb/s   6.06 Gb/s   6.15 Gb/s   47
  Jitter p-p(3)  60.16 ps    46.08 ps    60.16 ps    50
   Eye width(3)  0.685       0.672       0.691       47
```

Figure 3: 6.144 Gbps RP3-01 optical.

[4]10 Gigabit Small Form Factor Pluggable - Vendors in the cost-sensitive 10-Gigabit Ethernet (10 GbE) market are making a strong push to standardize SFP+ technology for use in 10 GbE applications and similar as an alternative to the XFP form factor.

[5]Which includes frame boundary marking characters (K28.7) and Message Group boundary marking characters (K28.5).

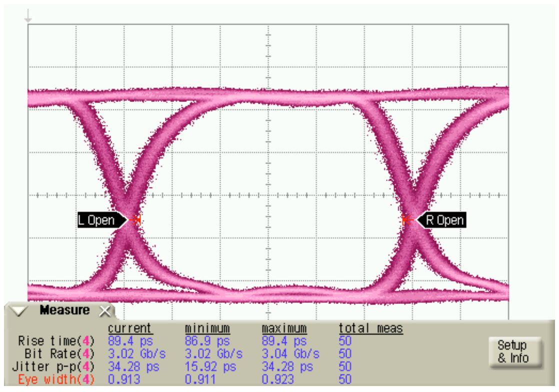
The relevant GX transceiver configuration used is as it follows:

Parameter (tx/rx)	**Value**
double data mode	true
data rate	6144
protocol	6G basic
equalizer	0
preemphasis	0
8b10b enc/dec	cascaded
ref clk	153.6 MHz
rx cru pll	tx clk

Table 2: GX Transceiver configuration

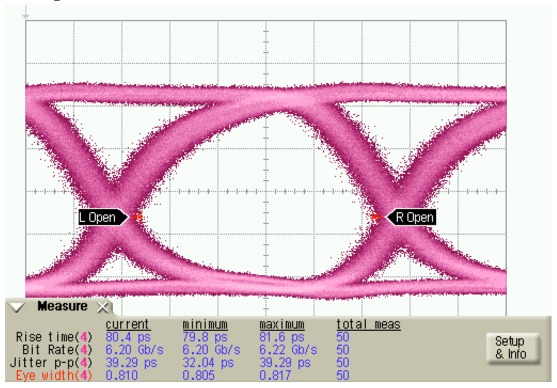
Also dynamical reconfiguration of each transceiver from one operating mode to another is supported. This mode reconfiguration involves reconfiguring of the data rate, data path, or both [7]. For this implementation a fixed double-width data-path of 32-bits is chosen and only data rate settings are set being dynamically reconfigurable from the user[6].
Fig.2 and Fig.3 shows the eye diagram measurements of 3.072 Gbps and 6.144 Gbps electrical signals respectively on the SIIGXAV[7]. The signal measured consist of valid RP3-01 frame structure[8] with data in every RP3-01 message slot.

These measurements are taken using Agilent 89105 DSO equipment that has been configured with a 153.6 MHz trigger reference locked to the transmitted data, which consist of valid RP3-01 data messages.
In [2] the indicative minimum values for eye mask compliance relative to the eye width for transmitter and receiver are 0.656 UI and 0.45 UI respectively. The eye diagram measurement at electrical 3.072 Gbps rate shows a peak-to-peak jitter value at 34.28 ps with a eye width value of 0.913 Unit Interval (UI).
The eye diagram measurement at electrical 6.144 Gbps rate shows a peak-to-peak jitter value at 39.29 ps with a eye width value at 0.810 UI.



```
                current      minimum      maximum      total meas
  Rise time(4)  89.4 ps      86.9 ps      89.4 ps      50
   Bit Rate(4)  3.02 Gb/s    3.02 Gb/s    3.04 Gb/s    50
 Jitter p-p(4)  34.28 ps     15.92 ps     34.28 ps     50
  Eye width(4)  0.913        0.911        0.923        50
```

Figure 4: 3.072 Gbps RP3-01 electrical signal.



```
                current      minimum      maximum      total meas
  Rise time(4)  80.4 ps      79.8 ps      81.6 ps      50
   Bit Rate(4)  6.20 Gb/s    6.20 Gb/s    6.22 Gb/s    50
 Jitter p-p(4)  39.29 ps     32.04 ps     39.29 ps     50
  Eye width(4)  0.810        0.805        0.817        50
```

Figure 5: 6.144 Gbps RP3-01 electrical signal.

VI - RP3-01 TIMING AND CONFIGURATION

The OBSAI BTS has a reference system clock (SCLK) of 30.72 MHz [3]. This is used as a convenient frequency for operations at a value eight times multiple of the WCDMA chip rate[9]. The RP3-01 interface reference frequency is different from SCLK, since an overhead of 3 bytes per RP3 message and control bandwidth are defined in [2], and the next convenient way if getting this extra bandwidth is a higher frequency reference multiple of 12.5 ($\times 10/8$) times the SCLK, thus 38.4 MHz. The RP3-01 byte clock frequency is a multiple of 38.4 MHz and it is defined being a factor of 10 the line rate used due to the 8b10b coding and phase locked to SCLK [2]. Table 3 illustrates the core clock frequencies according to the data path chosen.

Rate (Mbps)	**8DP clk (MHz)**	**32DP clk (MHz)**
6144	614.4	153.6
3072	307.2	76.8
1536	153.6	38.4
768	76.8	19.2

Table 3: OBSAI RP3-01 core clock frequencies for 8-bits and 32-bits data paths reespectively.

In order to run design into lower cost FPGA technology while maintaining the same serial throughput, higher data-path parallelization is chosen to lower the operating core clock frequency. In this design 6.144

[6]In case of dynamic reconfiguration enabled in double-width mode, only the 768 Mbps line rate is not supported from the transceivers since only line rates between 1 Gbps and 6.25 Gbps are allowed.

[7]These measurements are performed with the standard analog pre-emphasis and equalization settings on the ALT2GXB Megawizard.

[8]Which includes frame boundary marking characters (K28.7) and Message Group boundary marking characters (K28.5).

[9]The WCDMA chip rate is 3.84 Mcps.

Gbps and 32 bits data-path are chosen, giving a 153.6 MHz frequency.
This operating frequency enable usage of the RP3-01 IP block also in low-cost Altera Cyclone II and Cyclone III families, which internal logic supports maximum operating frequencies of around 167 MHz [8] and around 180 MHz [9] respectively. In this case an external physical layer is required.
The RP3-01 frame structure parameters are defined in [2] and reported in Table 4, and they are invariant for WCDMA, WiMAX and LTE configurations. i defines the frame structure according to the line rate used, and it takes integer values according to Table 5. IIn this implementation value eighth (i=8) is chosen. Thus only reconfiguration of i and of the core clock frequency is required to enable dynamic line rate re-configuration from the RP3-01 IP core.

M_MG	**N_MG**	**K_MG**	**i**
21	1920	1	variable

Table 4: RP3-01 Frame structure for WCDMA, 802.16 and LTE.

Line Rate (Mbps)	**i**
768	1
1536	2
3072	4
6144	8

Table 5: Line rate and "i" values definition.

VII - TEST SETUP AND RESULTS

The test setup block architecture is illustrated in Fig.6 where two SIIGXAV boards are connected to implement a full duplex optical communication at 6.144 Gbps of valid RP3-01 traffic. The measurements was also performed at 3.072 Gbps for comparing the 6.144 Gbps line rate results to the existing RP3 specifications Setup options for the RP3-01 IP and GX transceivers are done via DIP switches. External clock generator is used to generate 153.6 MHz for both the boards and the trigger signal to the 89105 DSO.

The Pseudo Random Bit Sequence (PRBS) generator blocks implements a simple 3-bytes counter for the RP3-01 message header and a 16-bytes counter for the RP3-01 message payload for each message slot[10] in a Message Group. A PRBS validator checks the received messages counters values and the BER counter measures the amount of bit errors received. A picture of the setup while it is running is given in Fig. 7, where the 7-segments display on each board shows the BER counter values and the LED bank shows that the system is operating correctly according to the mapping in Table 6.

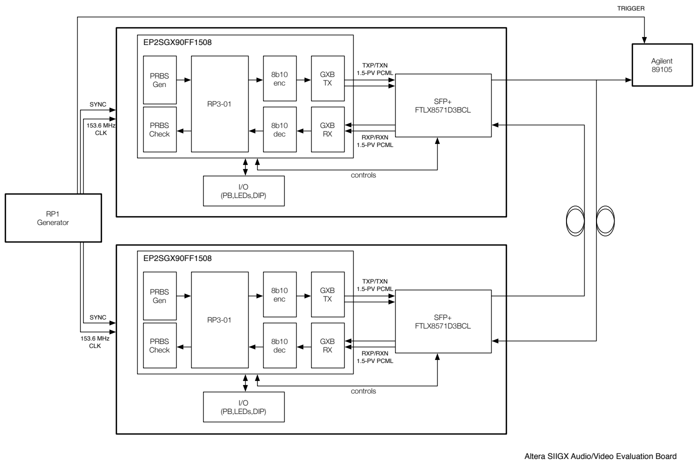


Figure 6: Block diagram of the test setup.

LED	**Signal**	**High**	**Low**
1	TX PLL	unlocked	locked
2	RX PLL	unlocked	locked
3	RP3-01 RX IDLE	false	true
4	RP3-01 RX SYNC	false	true
5	PRBS Errors	present	not present
6	LCV Errors	present	not present
7	RE-SYNC	on	off
8	N.A.	-	-

Table 6: Status LED signals mapping.

The results indicates that the transmission at 6.144 Gbps over each link is error free (zero value is constant on both the receivers end) measured over a time window of a few hours. It is possible to verify the correct operational status of the interface through the LED bank status indication that are mapped as indicated in Table 6 and they all shows "low" logic values as expected.

[10]RP3/RP3-01 message slot size is defined as 19 bytes [2].

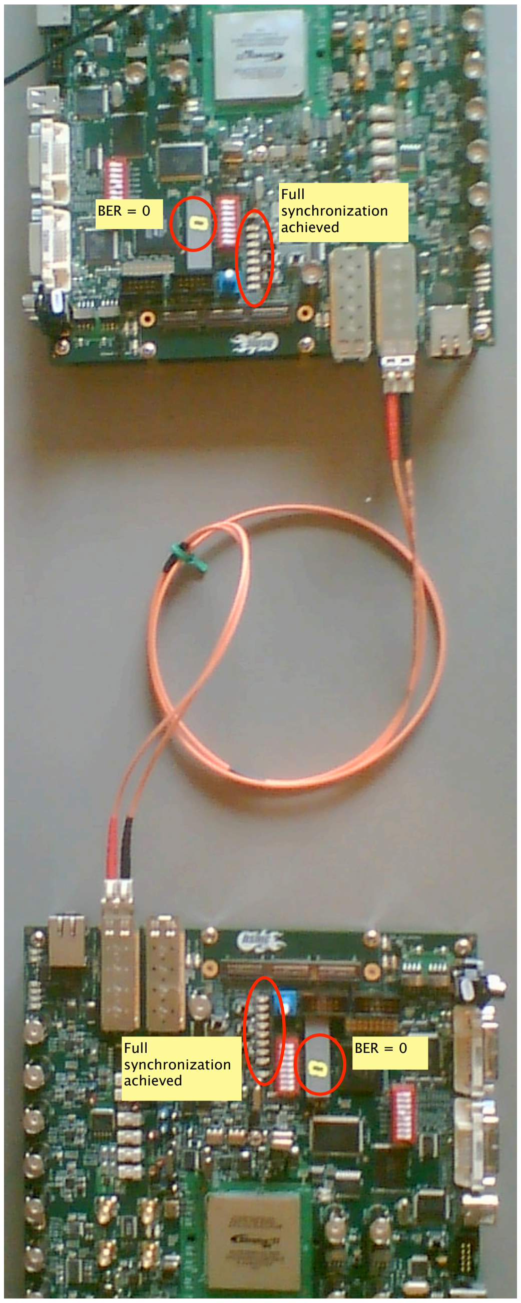


Figure 7: Hardware setup.

VIII - CONCLUSIONS

A 6.144 Gbps OBSAI RP3-01 point-to-point full-duplex transmission test setup was built running at 153.6 MHz. The OBSAI RP3-01 IP is supporting all OBSAI RP3-01 line rates, including the 6.144 Gbps one via register interface requiring reconfiguration only of the core clock frequency and frame structure parameter *i*. Stratix II GX dynamic channel reconfiguration also is supporting multiple rate configurations and backward compatibility with 3.072 Gbps link have been demonstrated using SFP+ FTLX8571D3BCL optical transceivers as RP3-01 physical layer. The measurements of the signal integrity compare the 3.072 Gbps with the 6.144 Gbps eye diagram with acceptable eye quality, proving an error-free communication with internal BER measurements.

ACKNOWLEDGMENTS

Thanks to Radiocomp ApS for permission of using part of their RP3-01 IP core. Thanks to Altera for providing evaluation boards and software tools and discussions. Thanks to Finisar and Laser2000 for providing SFP+ FTLX8571D3BCL samples. Thanks to Agilent for providing the 86100 measurement equipment. Thanks to Bob Blake and Akshaya Trivedi from Altera, Laurent Drozin from Finisar, Henrik Wessing and Lara Scolari from COM for valuable discussions and reviews.

TRADEMARKS

Finisar^{TM}, Altera^{TM}, Radiocomp^{TM}and Agilent^{TM}are registered trademarks.

REFERENCES

[1] [www.obsai.com]

[2] [www.obsai.com], "RP3 Specification v4.0"

[3] [www.obsai.com], "RP1 Specification v2.0"

[4] [www.ieee802.org], "802.16e-2005 WirelessMAN"

[5] [www.3gpp.org/Highlights/LTE/LTE.htm]

[6] [www.t10.org/ftp/t10/document.05/05-210r0.pdf]

[7] [http://www.altera.com/literature/hb/stx2gx/stxiigx_sii5v2_01.pdf] "Stratix II GX Transceiver User Guide"

[8] [http://www.altera.com/literature/hb/cyc2/cyc2_cii5v1.pdf], "Cyclone II Device Handbook, Volume 1"

[9] [http://www.altera.com/literature/hb/cyc3/cyc3_ciii5v1.pdf], "Cyclone III Device Handbook, Volume 1"

[10] "IEEE 802.3ae Standard for Information Technology Local &Metropolitan Area Networks Part 3: Carrier sense multiple access with collision detection (CSMA/CD) access method and physical layer specifications – Media Access Control (MAC) Parameters, Physical Layer, and Management Parameters for 10 Gb/s Operation"

A Dynamically Reprogrammable CSA-Generic Platform Architecture

Bruno Pimentel
Department of Electronics and Telecommunications and Informatics / IEETA
University of Aveiro, Portugal
brunopimentel@ua.pt

Abstract

This paper suggests a set of reusable hardware functional blocks and a platform architecture composed of such, for implementing a wide range of combinatorial search algorithms with a relevant development speed-up. Their conception was based on careful analysis of various classical algorithms of that kind, amongst which important similarities and differences have been identified. An overview of the relevant grounding is presented.
Making use of a control unit reprogramming strategy and a recently developed prototyping platform, the proposed hardware architecture is easily combined with a software application running in any general-purpose computer, allowing the user to load different combinatorial search algorithms, submit new problem instances and get the correspondent results, all in run-time.

1. Introduction

Combinatorial search algorithms (CSAs) continue to evolve in response to important combinatorial optimization problems (COPs) that arise within different areas, namely synthesis, optimization and testing of digital circuits [1, 2], mapping, placing and routing for integrated circuits design components [3], topology and cartography [4], artificial intelligence [5], etc. Some of the CSAs gain big relevance because they provide solution for classical COPs, such as determining a shortest or longest path within graphs, graph coloring, Boolean function optimization and minimal coverage problems, which have wide application scopes.
Hardware accelerators have been developed, but each COP is commonly addressed independently. In order to get the speed-up that hardware can provide, designing effort is generally targeted to CSA-specific accelerators, despite the significant similarities one can find amongst them.
The approach carried out in this research began with the detailed identification of those similarities as well as the differences. By taking advantage of the former and finding the best way to cope with the latter, the objective was to achieve:

– A thorough set of hardware component specifications which are reusable for a wide range of CSAs;
– A reprogrammable platform architecture able to enforce different CSA specifications provided in run-time.

2. Overview on CSAs

Before pointing out important similarities and differences amongst CSA implementations, let us consider some CSA examples, each addressing a different classical COP.
The Matrix Covering (MC) problem consists of finding the smallest row set of a given binary matrix that includes at least one value 1 in each column [6]. The approximate algorithm proposed in [7] to solve this problem is depicted in Fig. 1. When the algorithm has finished, the solution is the set of rows that have been removed.

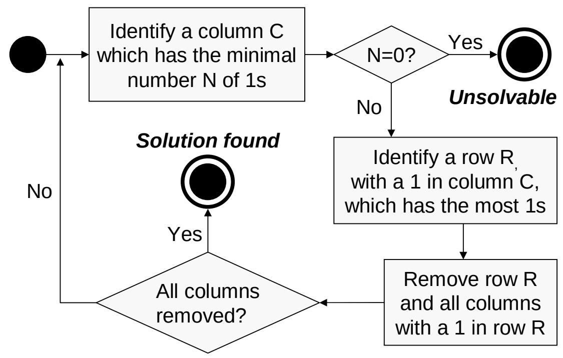
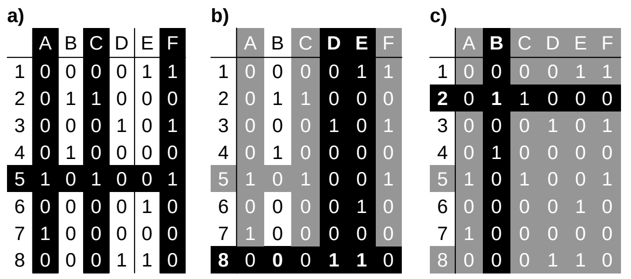

Identify a column C which has the minimal number N of 1s
N=0?
Yes
Unsolvable
No
Identify a row R with a 1 in column C, which has the most 1s
Remove row R and all columns with a 1 in row R
All columns removed?
Yes
Solution found
No

Fig. 1 - Approximate algorithm for the MC problem

Fig. 2 demonstrates the resulting steps of this algorithm with a practical example, depicting the 3 iterations of the algorithm cycle which lead to the solution. The row and the columns which are removed at each iteration are presented with a black background while a grey background indicates previously removed matrix parts. In the first iteration, row 5 gets to be removed because no other row contains more values 1 and then columns A, C and F are removed, as these contain a value 1 in that row. With 2 more iterations the algorithm reveals the solution composed of the removed rows 2, 5 and 8.



a)

	A	B	C	D	E	F
1	0	0	0	0	1	1
2	0	1	1	0	0	0
3	0	0	0	1	0	1
4	0	1	0	0	0	0
5	1	0	1	0	0	1
6	0	0	0	0	1	0
7	1	0	0	0	0	0
8	0	0	0	1	1	0

b)

	A	B	C	D	E	F
1	0	0	0	0	1	1
2	0	1	1	0	0	0
3	0	0	0	1	0	1
4	0	1	0	0	0	0
5	1	0	1	0	0	1
6	0	0	0	0	1	0
7	1	0	0	0	0	0
8	0	0	0	1	1	0

c)

	A	B	C	D	E	F
1	0	0	0	0	1	1
2	0	1	1	0	0	0
3	0	0	0	1	0	1
4	0	1	0	0	0	0
5	1	0	1	0	0	1
6	0	0	0	0	1	0
7	1	0	0	0	0	0
8	0	0	0	1	1	0

Fig. 2 - Solving a MC problem instance example

The Boolean Satisfiability (SAT) problem consists of determining whether the variables of a given Boolean

formula can be assigned in such a way as to make the formula evaluate to true. Conversion between Boolean formulas expressing SAT problem instances and their equivalent ternary matrices can be found in [8]. SAT-solving CSAs are typically based on operations over such matrices. In this approach, solving the problem corresponds to finding a ternary vector which is orthogonal to every row in the matrix that expresses the problem instance at hand. Note that i) a ternary value is either 0, 1 or "don't care" and that ii) two ternary vectors are considered orthogonal if at least one of their pairs of homologous elements is composed of a 0 and a 1.
Throughout the solving process, it is some times possible (and advantageous) to simplify the matrix obtaining an equivalent. This simplification is called reduction and it does not change the search route. However, when no further reduction operations can be performed, the solver might have to try alternative paths in order to check whether there is one which leads to a solution. The set of operations that determine which path to follow is called selection. When a chosen path fails, it is necessary to backtrack and select another, if available. Such a strategy requires a CSA like the one shown in Fig. 3.

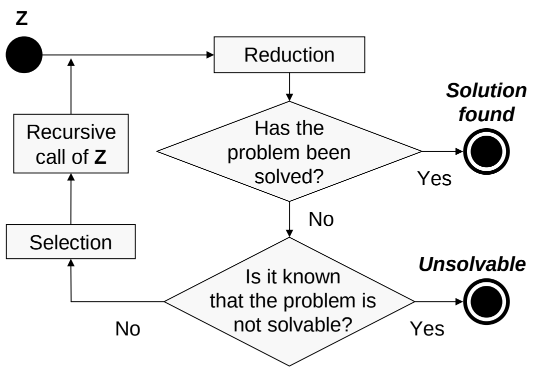

Z
Reduction
Has the problem been solved?
Yes
Solution found
No
Is it known that the problem is not solvable?
Yes
Unsolvable
No
Selection
Recursive call of Z

Fig. 3 - Basic combinatorial search algorithm

This generic algorithm can be used to solve many COPs, and it was used for the SAT solver presented in [8], in which the "Has the problem been solved?" condition test is verified if all matrix rows have been deleted and the "Is it known that the problem is not solvable?" condition test is verified if all matrix columns have been deleted. The reduction rules used in that example are the following:

1. If a column contains just "don't care" values, it must be deleted from the matrix;
2. All rows that are orthogonal to an intermediate vector w (that incrementally forms a solution) must be removed from the matrix. All columns that correspond to the components of the vector w with values 1 and 0 must be deleted from the matrix;
3. If the matrix contains a row with just one component 0 (1) with an index i, then the element i of the vector w must be assigned the value 1 (0), i.e. the inverted value;
4. If there is a column j in the matrix without values 1 (0) then the element j of w must be assigned the value 1 (0).

Finally, the selection rule used in this example was the following: A column C_{max} is selected that contains the maximum number of values 1 (let us designate this N_1) and 0 (let us designate this N_0); i.e. C_{max} has a minimum number of "don't care"s. If $N_1 \geq N_0$, then the value 0 for the column C_{max} is included in w. If $N_0 > N_1$, then the value 1 for the column C_{max} is included in w. This creates a sub-matrix that will be examined in the next step. If this path fails, the solver backtracks and repeats the attempt including the alternative value for the column C_{max} in vector w.
Fig. 4 illustrates the resulting search steps when applying these rules to a practical example matrix, representing "don't care" values with the character "-". Once again, the rows and columns which are removed at each iteration are presented with a black background while a grey background indicates previously removed matrix parts. When an element of vector w is assigned a value, its cell is also highlighted with a black background.

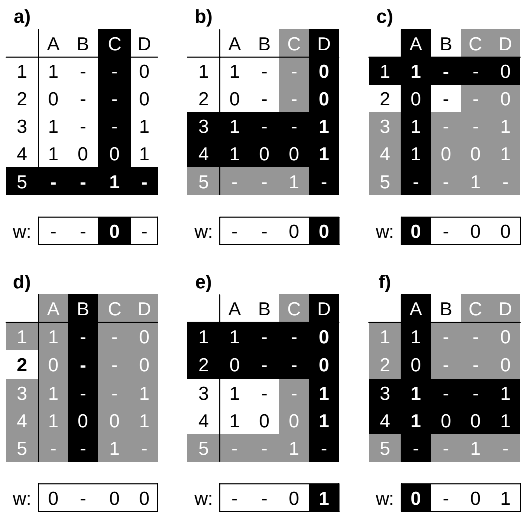


Fig. 4 - Solving a SAT problem instance example

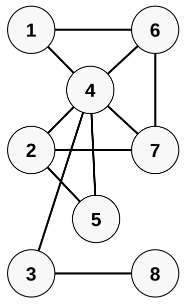
Let us focus on the selection rules only. After Fig. 5-a, no more reduction can take place and there are still rows and columns left, so in Fig. 5-b the value 0 for column D is included in vector w. Then, some reduction takes place and after Fig. 5-d there is still one row left (namely row 2), which means a solution was not yet found, but all columns have been removed, meaning this search path cannot provide a solution. Thus, the algorithm backtracks in order to try the search path alternative to the one chosen in Fig. 5-b. This time (Fig. 5-e), the value 1 for column D is included in vector w and then reduction rules are applied again. Finally (Fig. 5-f), all rows have been removed, meaning a solution has been found. Vector w ("0-01", at the end) has been constructed throughout this process and is now orthogonal to all given matrix rows.
The Graph Coloring (GC) problem consists of assigning one color to each vertex of a given graph, using the

minimum number of colors and taking into account that connected vertices must be assigned different colors. Solvers for the GC problem are also commonly based on algorithms that execute condition tests and operations over matrices that express the problem instances. In this third example, let us focus on how a graph that has to be colored can be converted to a ternary matrix in such a way that solving the problem over the matrix is equivalent to solving the problem over the graph. As shown in previous research [9], this conversion can be done using the following steps:

1. Have a matrix with N rows and N columns, N being the number of vertices in the graph;
2. Fill the main diagonal up with 0s;
3. Within the lower triangle (as filling the upper one would be redundant and unnecessary), insert a 1 in every cell with coordinates corresponding to connected edges in the graph;
4. Exclude every column that has no cells with a 1 (keeping track of which vertex each column corresponds to);
5. Fill all empty cells up with "don't care" values.

Fig. 5 presents a practical example of this conversion method. The graph that has to be colored (Fig. 5-a) is composed of 8 vertices and 11 edges. Fig. 5-b shows the results of this conversion right until the initial 8x8 matrix gets reduced. At step 4, columns E, G and H (highlighted with a black background) get excluded and the final 8x5 ternary matrix is presented in Fig. 5-c.

a) Graph to be converted



b) Conversion at step 4

	A	B	C	D	E	F	G	H
A	0							
B		0						
C			0					
D	1	1	1	0				
E		1		1	0			
F	1			1		0		
G		1		1		1	0	
H			1					0

c) Resulting matrix

	A	B	C	D	F
A	0	-	-	-	-
B	-	0	-	-	-
C	-	-	0	-	-
D	1	1	1	0	-
E	-	1	-	1	-
F	1	-	-	1	0
G	-	1	-	1	1
H	-	-	1	-	-

d) An optimal solution

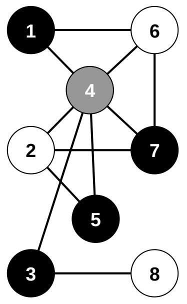


Fig. 5 - Converting a GC problem instance example

As mentioned above, two ternary vectors are considered orthogonal if at least one of their pairs of homologous elements is composed of a 0 and a 1. The method used to convert graphs to matrices guarantees that connected vertices correspond to orthogonal rows. Therefore, solving the GC problem corresponds to discovering a set with the minimum number of row subsets, each one having no orthogonal pair of rows and together including all rows. The number of compiled row subsets expresses the minimum number of colors the given graph requires, while rows grouped in a same subset correspond to vertices assigned the same color.

The exact algorithm presented in [9] is based on four steps that are common for combinatorial search problems and that are repeated sequentially until the solution is found:

1. *Reduction*: the matrix is reduced as much as possible applying some pre-established rules;
2. *Splitting*: the problem is decomposed in less complicated sub-problems;
3. *Termination*: The current step is terminated as soon as a new incomplete solution is of the same order (i.e. contains the same number of colors) as any previous solution that has already been found.
4. *Search for the optimal result*: steps 1, 2 and 3 are repeated until all possible solutions have been implicitly examined.

Because what was addressed so far already constitutes a good CSA overview (for our goal), let us skip other details on the GC algorithm and move on.

3. Similarities and Differences amongst CSA Implementations

Matrices are clearly the data structure most used as cores of COP solvers and the reason for this is the combination of 2 facts:

– Usually, any COP can be expressed in several equivalent mathematical formulations based on different standard data structures [10], such as graphs, matrices and Boolean functions;
– Matrices are the standard data structure which has proven itself more advantageous for storing and processing in digital systems [10].

Binary and ternary matrices are the most used in CSAs, e.g. in those considered in the previous section. If we think of digital circuits (with which designers can achieve best time performances), while a binary value requires a single bit that explicitly states its value, a ternary value can be coded using 2 bits. Thus, 2 binary matrices can be used to compose a ternary matrix.

A second criterion regarding the matrix CSA core divides COP solvers in 2 groups: one with simple access to the matrices (by rows or by columns) and the other one needing dual access (by rows and by columns). All 3 CSAs considered in the previous section make use of ternary matrices.

Combining these 2 criteria, 4 CSA classes emerge with direct correspondence to 4 kinds of matrix: Single Access Binary Matrix (SABM), Single Access Ternary Matrix (SATM), Dual Access Binary Matrix (DABM) and Dual Access Ternary Matrix (DATM). Again in the context of digital circuits, aiming for good time performance provided by RAM-based logic-vector arrays, dual access to a matrix calls for a replication of its data. One copy is organized as an array of rows and the second as one of columns. As a result, the number of RAM-based logic-vector arrays used to implement COP solvers in function of matrix and matrix access types is presented in Table 1.

Table 1 - Number of logic-vector arrays in function of matrix and matrix access types

	Binary Matrix	**Ternary Matrix**
Simple Access	(SABM Class) 1 array required	(SATM Class) 2 arrays required
Dual Access	(DABM Class) 2 arrays required	(DATM Class) 4 arrays required

As mentioned before, a CSA usually starts with a matrix that expresses the problem instance to solve (the search tree root) and then rows and columns are removed as the algorithm runs forward in some search path. Eventually it can backtrack, which implies recovering rows and columns that were previously removed. This requirement is commonly satisfied with the use of masks, which are easily implemented with logic-vectors. A row mask contains 1 bit per matrix row, indicating whether that row has been removed or not. The same approach is applied for the column mask. The use of masks has nevertheless an implication: the operations over rows and columns must be designed in such a way as to correctly cope with partial vectors.
Regarding the problem instances, there are 2 kinds of COPs: those for which any instance is solvable (e.g. the GC problem), and those for which there are instances with no solution (e.g. the SAT problem). Depending on the kind of COP, it becomes possible to conceive solving CSAs from 3 algorithmic flow (AF) categories:

- Single path;
- Preemptive search tree;
- Exhaustive search tree.

Single path algorithms do not use backtracking, resulting in somewhat simple algorithms, like the approximate one considered for the MC problem (Fig.1). Search tree AF solvers require a more complex algorithm to support testing alternative paths. With preemptive search tree algorithms, such as the one considered for the SAT problem (Fig. 3), the search ends as soon as the first solution is found. With exhaustive search tree algorithms, such as the one considered for the GC problem, all branches of the search tree which can provide a solution are tested as to ensure that an optimal one is found. Note that both search tree AFs may include pruning techniques to shorten the search.
CSAs can be quite different from one another and their constituting steps can have completely different meanings (within the context of how they approach the problem). Still, because the data structures they manipulate are, as mentioned above, basically the same, the operations used as basic blocks to implement those CSAs are in fact very much the same. Examples of generally used micro-operations (the most basic ones) are the following:

- Remove a row/column;
- Read a row/column;
- Count 1s/0s in a binary/ternary vector;
- Find the address of the first 0/1 in a binary/ternary vector;
- Check whether 2 binary/ternary vectors are orthogonal;
- Combine 2 binary/ternary vectors.

There are also composed operations (compilations of micro-operations) which are still very commonly used, such as:

- Find the row/column with the most/least 0s/1s in a matrix;
- Count the number of rows/columns which have no 0s/1s in a matrix;
- Check whether there are any matrix rows/columns orthogonal to some binary/ternary vector;
- Combine all rows/columns of a matrix which are combinable with some binary/ternary vector.

4. Reusable Functional Blocks

Taking into account the similarities and differences amongst CSA implementations, the following functional blocks were prepared:

1. Memory permitting to store both binary and ternary matrices and to provide access addressing either lines or columns;
2. Mask registers making it possible to use the same storage for handling initial matrices and their sub-matrices, which result from removing rows and/or columns;
3. Stacks for managing forward and backward propagation steps, which permit to sequentially construct sub-matrices and to return back to any intermediate sub-matrix if required;
4. General-purpose registers over which operations can be executed when required (namely by a control unit enforcing some algorithm);
5. Operational Unit (OU) implementing a variety of generic basic operations over binary and ternary vectors with and without mask as a parameter;
6. Reprogrammable Control Unit (RCU) to enforce CSAs by activating the operations on an OU. The algorithm enforced can be dynamically replaced through reconfiguration of the control circuit, which is modeled by a Reconfigurable Hierarchical Finite State Machine [11]. An example of such machine for implementing operations over Boolean and ternary vectors was considered in detail in [12];
7. User Agent (UA) circuit to allow testing, debugging and interacting with the desired system through a general-purpose computer.

Parameterization on the considered FBs was provided so as to make them fit any required scale, e.g. depending on the available hardware resources and on the targeted matrix maximum dimensions.

There are many varieties of the operations required for CSAs, for example: use or not the contents of a mask register; store (in a general-purpose register) or not store the result; use just one vector of a binary matrix or two vectors of a ternary matrix. However the number of such operations is limited and thus the proper reusable interface can be described.

Because the use of masks has different implications in each operation implemented by the OU, they are provided as a parameter and handled internally.

5. CSA-Generic Platform Architecture

Fig. 6 depicts the CSA-generic platform architecture developed. For the reasons previously presented, 4 binary vector arrays permit to store the problem instance matrices (left hand size of Fig. 6) for any of the 4 COP solver classes addressed:

- SABM, using Rows ones;
- SATM, using Rows ones and Rows zeros;
- DABM, using Rows ones and Columns ones;
- DATM, using all 4 binary arrays.

The binary matrices stored in Columns ones and Columns zeros must be the exact transposes of those stored in Rows ones and Rows zeros, respectively, and they are used only with DABM or DATM solvers, as SABM and SATM ones access the matrix only by rows.

With SABM solvers, Rows ones are used to explicitly map values 1 and values 0. With SATM solvers, Rows ones and Rows zeros map with 1s values 1 and values 0, respectively, while "don't care" values are implicitly mapped where neither 1s nor 0s are. With dual access matrix solvers, this approach is also applied for Columns ones and Columns zeros.

Row/Column addresses are used for reading or writing a whole row/column. In a binary context, these operations require one binary register to read from or to write to, while in a ternary context, 2 binary vectors are required.

Row/Column masks indicate with 1s the rows/columns that the algorithm has set as removed from the matrix. In fact, the whole initial matrix remains stored until the algorithm finishes and stores a new one; only the FBs that implement the operations over its rows and columns take the correspondent masks into account in order to produce the correct result.

An OU is used to operate over stacks, masks, addresses, rows, columns and general-purpose registers, as required by the algorithms that the RCU carries out.

A UA interacts with a general-purpose computer (e.g. using USB or Bluetooth), allowing the user to:

- Reprogram the RCU with a new CSA as explained in [11];
- Send a new problem instance (in the form of a matrix) which is stored in the logic-vector arrays;
- Get back the solver results.

Various stacks are used to store and restore context and HFSM support variables as the RCU calls and returns from different hierarchical level modules.

The architecture is adjustable in respect to the following parameters:

- Supported CSA classes (implicating a different number of logic-vector arrays and different available operations on the OU);
- Matrix dimensions;
- Stacks depth;
- Number of general-purpose registers;

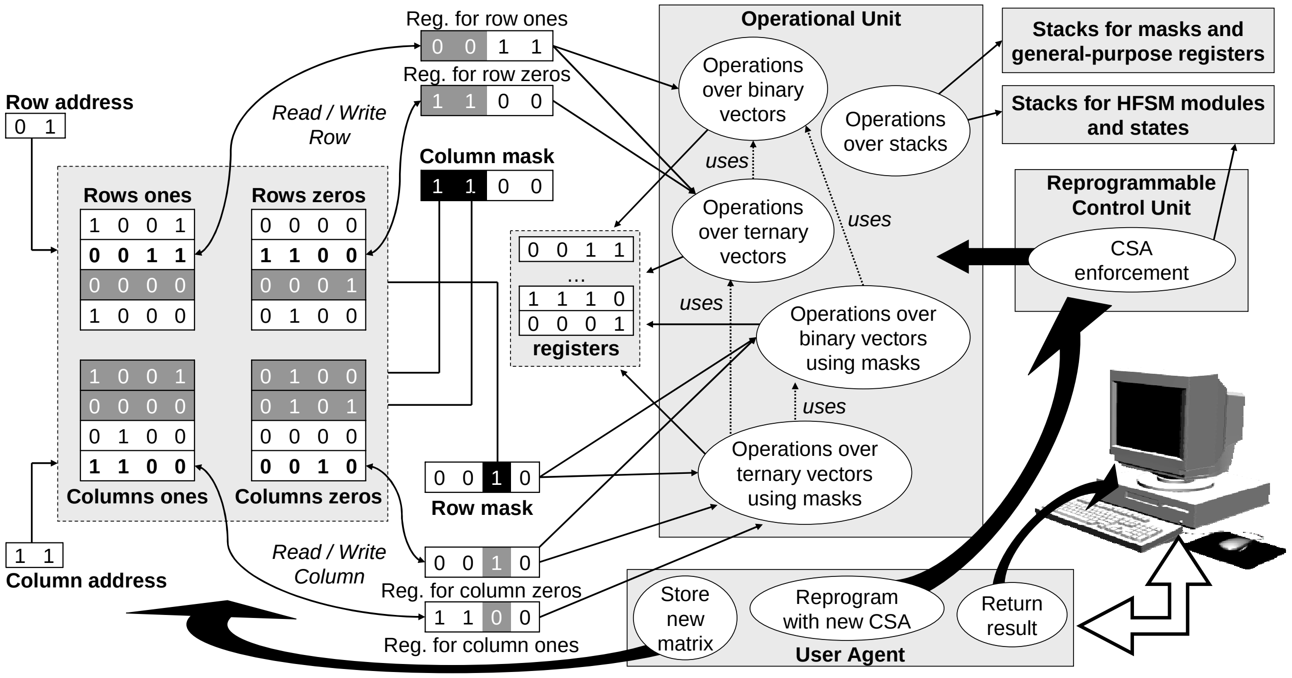


Fig. 6 - The dynamically programmable CSA-generic platform architecture proposed

– Some other parameters regarding the RCU which can be found and explained in [11].

6. Validation, Implementation, Test and Future Work

The proposed architecture is composed of 2 main components:

– The control component, composed of the RCU and the UA together with the software application;
– The operational component, which is composed of all the other FBs and implements the whole set of operations over the data structures considered.

The operational component and all interaction between its FBs have been validated in a software application programmed in C# in which each FB was described by a class that emulates the behavior expected from its hardware implementation. Using the same objects, the application ran all 3 CSA-solvers considered in section 2: for MC, SAT and GC. A special class emulated the RCU behavior to validate enforcing those different algorithms.

After validation, the architecture's operational component was implemented and successfully tested using Handel-C [13] system-level specification language and the recently developed DEITUA-S3 prototyping board [14], which incorporates a Xilinx Spartan-3 FPGA (namely a XC3S400). The 4 binary vector arrays for storing the matrix were implemented using the FPGA's embedded block RAM. A USB interface was used for data exchange between the hardware platform and the software.

Hardware RCU and the UA modules were designed using VHDL, whilst a software application to interact with the UA was developed in C#. The expected run-time RCU reprogramming was successfully achieved.

In future work, a set of good compromises regarding the assignment of the architecture's parameters, taking the available FPGA resources and subtle CSA classes specialization into account, will be determined. For each resulting platform, various tests will be carried out, with the control and the operational components integrated, and compare the results with other solutions. The Handel-C code produced to implement the FBs and the platforms will be made available online [15].

7. Conclusions

The significant applicability of combinatorial search algorithms in many different areas stimulates the implementations of hardware accelerators to run them. Despite the strong and frequent similarities that can be found amongst those algorithms, solver implementations are usually problem-specific.

On the groundings of a thorough analysis of combinatorial search algorithms, a set of reusable functional blocks and a dynamically reprogrammable platform architecture supporting a wide range of those algorithms were developed. Such tools allow for significant reduction of solvers design time, as the design process can be realized at a high level of abstraction without losing sight of the details of a particular problem or reducing performance.

Acknowledgments

The author would like to acknowledge Valery Sklyarov, Iouliia Skliarova and Manuel Almeida for their valuable collaboration.

References

[1] G. De Micheli, "Synthesis and Optimization of Digital Circuits", McGraw-Hill, Inc., 1994.
[2] M.A. Breuer, M. Sarrafzadeh, F. Somenzi, "Fundamental CAD Algorithms", IEEE Trans. on Computer Aided Design of Integrated Circuits and Systems, vol. 19, no. 12, Dec. 2000, pp. 1449-1475.
[3] N. Togawa, M. Yanagisawa, T. Ohtsuki, "Maple-opt: a performance-oriented simultaneous technology mapping, placement, and global routing algorithm for FPGAs", IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems, vol. 17, no. 9, Sep. 1998, pp. 803-818.
[4] T. Tambouratzis, "A consensus-function artificial neural network for map-coloring", IEEE Trans. on Systems, Man and Cybernetics, Part B, vol. 28, no. 5, Oct. 1998, pp. 721-728.
[5] A.D. Zakrevski, "Combinatorial Problems over Logical Matrices in Logic Design and Artificial Intelligence", Electrónica e Telecomunicações, vol. 2, no. 2, 1998, pp. 261-268.
[6] A.D. Zakrevski, "Algorithms of Synthesis of Discrete Automata", Moscow: Science, 1971, (in Russian).
[7] A.D. Zakrevskij, "Logical Synthesis of Cascade Networks", Moscow: Science, 1981.
[8] I. Skliarova, A.B. Ferrari, "The Design and Implementation of a Reconfigurable Processor for Problems of Combinatorial Computation", Journal of Systems Architecture, Special Issue on Reconfigurable Systems, vol. 49, 2003, pp. 211-226.
[9] V. Sklyarov, I. Skliarova, B. Pimentel, "Modeling and FPGA-based implementation of graph coloring algorithms", Proc. of the 3rd International Conference on Autonomous Robots and Agents - ICARA'2006, Palmerston North, New Zealand, Dec. 2006, pp. 443-448.
[10] I. Skliarova, "Reconfigurable Architectures for Problems of Combinatorial Optimization", Ph.D. Thesis, University of Aveiro, Portugal, January 2004.
[11] V. Sklyarov, I. Skliarova, "Reconfigurable Hierarchical Finite State Machines", Proc. of the 3rd International Conference on Autonomous Robots and Agents - ICARA'2006, Palmerston North, New Zealand, Dec. 2006, pp. 599-604.
[12] V. Sklyarov, I. Skliarova, A. Oliveira, A.B. Ferrari, "A Dynamically Reconfigurable Accelerator for Operations over Boolean and Ternary Vectors", Proc. of the EUROMICRO Symposium on Digital System Design - DSD'2003, Antalya, Turkey, Sep. 2003, pp. 222-229.
[13] Celoxica, "Handel-C for hardware design", available at www.celoxica.com.
[14] M. Almeida, B. Pimentel, V. Sklyarov, I. Skliarova, "Design Tools for Rapid Prototyping of Embedded Controllers", Proc. of the 3rd International Conference on Autonomous Robots and Agents - ICARA'2006, Palmerston North, New Zealand, Dec. 2006, pp. 683-688.
[15] IEETA homepage, available at www.ieeta.pt.

Application of ASM++ methodology on the design of a DSP processor

S. de Pablo, S. Cáceres, J.A. Cebrián
University of Valladolid
E.T.S.I.I., Paseo del Cauce, s/n
47011 Valladolid (Spain)
sanpab@eis.uva.es

M. Berrocal
eZono AG
Winzerlaerstrasse 2
07745 Jena (Germany)
manuel@ezono.com

Abstract

This article presents the application of a graphical methodology used to develop a Digital Signal Processor designed for FPGA. The instruction set and main features of this processor are introduced. Then, a modified Algorithmic State Machine methodology, named ASM++, is applied to fully describe the processor implementation. This processor has been simulated and physically tested on Xilinx Spartan-3 devices, achieving 37.5~75 MIPS and up to 150 MOPS running at 75 MHz.

1. Introduction

Most intellectual property (IP) modules are designed as synchronous digital circuits using a standard hardware description language (HDL), usually VHDL or Verilog. Designers usually prefer a text-based tool to describe their circuits because editing and managing texts is easier than dealing with the arrangement of schematics. Compared to schematic entry, productivity is increased, mostly when parametrical modules are required.

To assist designers in their daily job, several visual tools have been developed to facilitate the circuit behavior description and understanding, namely Finite State Machines (FSM) and Algorithmic State Machine (ASM) [1], [3]. However, these tools are limited in their scope, so they are applied only on small state machines and circuits.

This paper presents several modifications of standard ASM diagrams with the aim of applying this methodology to design real-life circuits, document them and ease their supervision [8]. As an example, this methodology has been successfully applied in the design of an FPGA based DSP processor.

2. The DSPuva18 processor

The DSPuva18 processor is based on the former DSPuva16 [6], a Digital Signal Processor developed for Power Electronic applications [2]. These are the main features of this new processor and their improvements:

- Its computational instructions are executed using two clock cycles, rather than four [6], thanks to the use of an FPGA hardwired multiplier.
- Its control instructions (call, ret, jp, …) are usually executed in one clock cycle.
- It has adaptive conditional jumps and returns: it introduces one or two wait states to leave previous operations to finish.
- The program length can be up to 64K instructions.
- It can execute from 16 to 128 nested subroutines.
- The data memory is up to 64K words, with fast direct and indirect access (two clock cycles).
- It has direct access to 256 ports/devices.
- The instruction set, as shown in table 1, has been designed around 17 basic instructions, but most of these instructions lead to more possibilities.
- It has access to immediate constants in program code to ease filter implementation.
- An implicit access to last port used, with write back capability, has been introduced to speed up filters. It allows up to four operations per instruction.
- The range of fixed-point registers and values can be selected at instantiation time between ±1, ±2, ±4 and ±8. This feature eases in-circuit debugging.

As can be seen, some of these features are common with other processors, but other ones are new. The basic instruction set of this processor is shown below.

Table 1. DSPuva18 basic instruction set.

OpCode	Mnemonic	Function
0000 dddd dddd dddd	call <destination-address>	Jump to a subroutine.
0001 dddd dddd dddd	goto <destination-address>	Unconditional jump.
0010 0fff dddd dddd	jp*FLAG* <relative-jump>	Conditional jump.
0010 1fff •••• ••••	ret*FLAG*	Conditional return.
0011 kkkk kkkk kkkk	imm K_{12}	Prepare a constant.
0100 kkkk kkkk nnnn	rN = port(K_8)	Read from a direct port.
0101 kkkk kkkk nnnn	port(K_8) = rN	Write to a direct port.
0110 •••• bbbb nnnn	rN = mem({rB,K_{16}})	Read from memory.

0111 •••• bbbb nnnn	mem({rB,K_{16}}) = rN	Write to memory.
1000 sfff bbbb nnnn	if*FLAG* rN = [–]{rB,K_{16}}	Conditional assignment.
1001 xxxx bbbb nnnn	rN = *fx*({rN,*LP},{rB,K_{16}})	Extra functions.
1010 nnnn bbbb aaaa	rN = {rA,*LP} + {rB,K_{16}}	Addition.
1011 nnnn bbbb aaaa	rN = {rA,*LP} – {rB,K_{16}}	Subtraction.
1100 nnnn bbbb aaaa	rN = {rA,*LP} * {rB,K_{16}}	Multiply two values.
1101 nnnn bbbb aaaa	rN = – {rA,*LP} * {rB,K_{16}}	Multiply and change sign.
1110 nnnn bbbb aaaa	rN += {rA,*LP} * {rB,K_{16}}	Positive accumulation.
1111 nnnn bbbb aaaa	rN –= {rA,*LP} * {rB,K_{16}}	Negative accumulation.

This basic instruction set is extended as seen on tables 2 and 3. Additionally, most instructions allow the use of a register ('rB') or a 16-bit constant ('K_{16}'), easing constant coefficient filter implementation. This constant is built using four bits of the current instruction and twelve bits of the previously executed 'imm' instruction.

At the same time, a completely new feature has been added: when 'r0' is addressed as register 'rA', the last port used ('*LP') is read, the read value is used instead of r0's value, and then it is written back to the same port. As seen later, this feature speed up the implementation of large filters, requiring just one instruction per tap.

The control instructions of this processor are easy to understand. First of all, 'call' and 'goto' execute an absolute jump to a 4K to 64K address in one clock cycle. As long as only twelve bits are available to give the destination address, its value is multiplied by 1, 2, 4, 8 or 16, depending on the processor model, thus allowing larger programs. Consequently, all subroutines must be aligned to a reachable address, but the assembler can do it easily using the '#align' directive.

Conditional jumps and returns are a bit different (see the eight available conditions on table 2, that shows conditional assignments): they execute their task, but they wait one clock cycle for arithmetic and logic operations to finish, and two clock cycles for multiplications. This way, the use of interleaving 'nop' instructions is avoided. When unconditional 'jp' or 'ret' is used, it is executed in one clock cycle.

The access to external data is fast and flexible. The processor can address up to 256 direct ports, usually related to physical devices or small memories, maybe shared with other FPGA processors. When large amounts of data must be used, the processor implements a dedicated interface enabling the use of synchronous FPGA memories like Xilinx BlockRAM or Altera M4K and M-RAM. It can address up to 64K words per page, and different pages may be selected using a page-register controlled through a port. All these accesses are executed using two clock cycles.

This processor can conditionally load a register with a constant or the value of another register (see table 2), and it also implements more functions as shown in table 3. Right and left shifts are a bit different than expected because most used shifts are the shortest ones, thus using shifts by 7, 3, 2 and 1 rather than 8, 4, 2 and 1 it is on average better. The 'max' and 'min' instructions are also useful, particularly "rN = abs(rN)" is recognized by the assembler and replaced by "rN = max(rN,–rN)". All these instructions use two clock cycles for their execution, like additions and subtractions; their results are immediately available in the following instruction.

The four multiplying instructions, with optional positive or negative accumulation, are executed using only two clock cycles, but the result cannot be used as an operand, except for accumulation, at the following instruction. If required, a one clock 'nop' (an assembler macro replaced by "jp <next-address>") must be added.

Table 2. Conditional assignments of DSPuva18.

OpCode	Mnemonic	Function
1000 0000 bbbb nnnn	rN = {rB,K_{16}}	Load a register.
1000 0001 bbbb nnnn	ifV rN = {rB,K_{16}}	Load if oVerflow.
1000 0010 bbbb nnnn	ifEQ rN = {rB,K_{16}}	Load if EQual to 0.
1000 0011 bbbb nnnn	ifNE rN = {rB,K_{16}}	Load if Not Equal to 0.
1000 0100 bbbb nnnn	ifGT rN = {rB,K_{16}}	Load if Greater Than 0.
1000 0101 bbbb nnnn	ifGE rN = {rB,K_{16}}	Load if Greater or Equal.
1000 0110 bbbb nnnn	ifLE rN = {rB,K_{16}}	Load if Less or Equal.
1000 0111 bbbb nnnn	ifLT rN = {rB,K_{16}}	Load if Less Than 0.
1000 1000 bbbb nnnn	rN = –{rB,K_{16}}	Load changing sign.
1000 1001 bbbb nnnn	ifV rN = –{rB,K_{16}}	Load if oVerflow.
1000 1010 bbbb nnnn	ifEQ rN = –{rB,K_{16}}	Load if EQual to 0.
1000 1011 bbbb nnnn	ifNE rN = –{rB,K_{16}}	Load if Not Equal to 0.
1000 1100 bbbb nnnn	ifGT rN = –{rB,K_{16}}	Load if Greater Than 0.
1000 1101 bbbb nnnn	ifGE rN = –{rB,K_{16}}	Load if Greater or Equal.
1000 1110 bbbb nnnn	ifLE rN = –{rB,K_{16}}	Load if Less or Equal.
1000 1111 bbbb nnnn	ifLT rN = –{rB,K_{16}}	Load if Less Than 0.

Table 3. Extra instructions of DSPuva18.

OpCode	Mnemonic	Function
1001 0000 bbbb nnnn	rN = rB >> 7	Right shift seven bits.
1001 0100 bbbb nnnn	rN = rB >> 3	Right shift three bits.
1001 1000 bbbb nnnn	rN = rB >> 2	Right shift two bits.
1001 1100 bbbb nnnn	rN = rB >> 1	Right shift one bit.
1001 0001 bbbb nnnn	rN = rB << 7	Left shift seven bits.
1001 0101 bbbb nnnn	rN = rB << 3	Left shift three bits.
1001 1001 bbbb nnnn	rN = rB << 2	Left shift two bits.
1001 1101 bbbb nnnn	rN = reverse rB	Reverse all bits.
1001 0010 bbbb nnnn	rN = {rN,*LP} and {rB,K_{16}}	Logic AND.
1001 0110 bbbb nnnn	rN = {rN,*LP} or {rB,K_{16}}	Logic OR.
1001 1010 bbbb nnnn	rN = {rN,*LP} xor {rB,K_{16}}	Logic XOR.

1001 1110 bbbb nnnn	rN = not rB	Logic NOT.
1001 0011 bbbb nnnn	rN = min ({rN,*LP},{rB,K_{16}})	Minimum of two values.
1001 0111 bbbb nnnn	rN = max({rN,*LP},{rB,K_{16}})	Maximum of two values.
1001 1011 bbbb nnnn	rN = min({rN,*LP},–{rB,K_{16}})	Minimum changing sign.
1001 1111 bbbb nnnn	rN = max({rN,*LP},–{rB,K_{16}})	Maximum changing sign.

A program example that implements an infinite impulse response filter (IIR) is shown below. Most instructions of this filter execute up to four operations: a read from last used port (through '*LP'), a write back of the read value to the same port (so it reads an old sample or output from a FIFO and returns it to the same FIFO for the next filter update), a fixed-point 18x18 product and a positive 32-bit accumulation. This means 37.5 MIPS and 150 MOPS running at 75 MHz.

```
/*
   Demonstration program of DSPuva18 for FPGAworld'2007
   2007/08/27    Santiago de Pablo (sanpab@eis.uva.es)
*/

#model     E               // Programs up to 64K instructions
#range     8               // DSP values between +-8.0
#include   "uva18std.h"    // Several definitions

// IIR filter implementation:
//     Input X values are available at port 200.
//     Output Y values are written at port 201.
//     Old X values are stored in a small FIFO at port 202.
//     Old Y values are stored in a small FIFO at port 203.

   #define IN_X         200
   #define OUT_Y        201
   #define FIFO_X       202
   #define FIFO_Y       203

   #define YC1          0.9345
   // Define also YC2...YC4 and XC0...XC5 constants.

0x0000:                    // Programs begins here after reset
        call InitFilter    // Prepare the filter
Loop:   call UpdateFilter  // 14 + 2x(NX + NY) clks
        jp Loop            // Infinite loop (2 MSPS at 70 MHz)

#align
InitFilter:
   // First reset FIFO_X and FIFO_Y (not done here)
   // Then load dummy values as old samples
   r1 = 0.0
   port(FIFO_Y) = r1   // Load four values on FIFO_Y:
   port(FIFO_Y) = r1   //   they are y4, y3, y2 & y1.
   port(FIFO_Y) = r1
   port(FIFO_Y) = r1
   port(FIFO_X) = r1   // Load five values on FIFO_X:
   port(FIFO_X) = r1   //   they are x5, x4, x3, x2 & x1.
   port(FIFO_X) = r1
   port(FIFO_X) = r1
   port(FIFO_X) = r1
   ret
```

```
#align
UpdateFilter:
   r2 = port(FIFO_Y)   // Read y4 value (and loose it later)
   r1 =          r2 * YC4 // … and multiply y4 by its coefficient
   r1 = r1 + *LP * YC3 // Get y3 and multiply it by its coefficient
   r1 = r1 + *LP * YC2 // Get y2 and multiply it by its coefficient
   r1 = r1 + *LP * YC1 // Get y1 and multiply it by its coefficient
   r2 = port(FIFO_X)   // Read x5 value (and loose it later)
   r1 = r1 +   r2 * XC5 // … and multiply x5 by its coefficient
   r1 = r1 + *LP * XC4 // Get x4 and multiply it by its coefficient
   r1 = r1 + *LP * XC3 // Get x3 and multiply it by its coefficient
   r1 = r1 + *LP * XC2 // Get x2 and multiply it by its coefficient
   r1 = r1 + *LP * XC1 // Get x1 and multiply it by its coefficient
   r2 = port(IN_X)     // Get a new x0 value (from an A/D?)
   r1 = r1 +   r2 * XC0 // … and multiply x0 by its coefficient
   port(FIFO_X) = r2   // Put x0 value on its FIFO for later use
   port(FIFO_Y) = r1   // Put y0 value on its FIFO for later use
   port(OUT_Y) = r1    // Output of the IIR filter (to a D/A?)
   ret                 // Finish
```

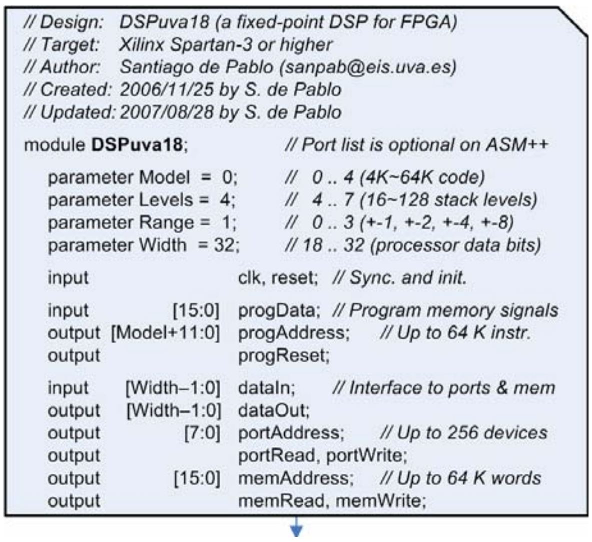
3. ASM++ diagram of DSPuva18

The design of this processor has been entirely done using ASM++ diagrams. These diagrams, proposed at [8] and described further here, are an extension of Algorithmic State Machines [1], [3], a methodology used forty years ago for the development of microprocessors. As can be seen with this example, the ASM++ diagrams are now fully capable of describing whole IP modules.

This diagram and the manually generated equivalent code use Verilog 2001, but VHDL may be used instead. An ASM++ compiler that accept standard Verilog and VHDL languages for input and output is in progress.

The first ASM++ box of this design, as seen below on Fig. 1, is a "code box", able to introduce Verilog or VHDL code. It is used in this case to describe the processor interface.

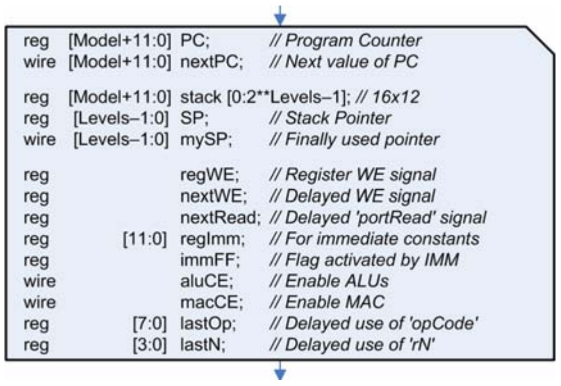
Figure 1. Design header using Verilog.



```
// Design:   DSPuva18 (a fixed-point DSP for FPGA)
// Target:   Xilinx Spartan-3 or higher
// Author:   Santiago de Pablo (sanpab@eis.uva.es)
// Created:  2006/11/25 by S. de Pablo
// Updated:  2007/08/28 by S. de Pablo

module DSPuva18;                 // Port list is optional on ASM++

   parameter Model  =  0;        //   0 .. 4 (4K~64K code)
   parameter Levels =  4;        //   4 .. 7 (16~128 stack levels)
   parameter Range  =  1;        //   0 .. 3 (+-1, +-2, +-4, +-8)
   parameter Width  = 32;        // 18 .. 32 (processor data bits)

   input                  clk, reset;  // Sync. and init.

   input           [15:0] progData;  // Program memory signals
   output   [Model+11:0]  progAddress;     // Up to 64 K instr.
   output                 progReset;

   input    [Width–1:0]   dataIn;       // Interface to ports & mem
   output   [Width–1:0]   dataOut;
   output          [7:0]  portAddress;     // Up to 256 devices
   output                 portRead, portWrite;
   output         [15:0]  memAddress;      // Up to 64 K words
   output                 memRead, memWrite;
```

Afterwards, a second code box specifies several internal signals. As long as this box has global meaning, other signals would be and will be declared later.

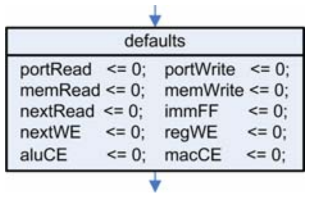
Figure 2. Declaration of several signals.



```
reg   [Model+11:0] PC;         // Program Counter
wire  [Model+11:0] nextPC;     // Next value of PC

reg   [Model+11:0] stack [0:2**Levels–1]; // 16x12
reg    [Levels–1:0] SP;        // Stack Pointer
wire   [Levels–1:0] mySP;      // Finally used pointer

reg                 regWE;     // Register WE signal
reg                 nextWE;    // Delayed WE signal
reg                 nextRead;  // Delayed 'portRead' signal
reg          [11:0] regImm;    // For immediate constants
reg                 immFF;     // Flag activated by IMM
wire                aluCE;     // Enable ALUs
wire                macCE;     // Enable MAC
reg           [7:0] lastOp;    // Delayed use of 'opCode'
reg           [3:0] lastN;     // Delayed use of 'rN'
```

The third box introduces a first difference between ASM++ and the pure code. It specifies global defaults for synchronous and asynchronous internal signals and outputs. If the user does not assign anything to a synchronous signal in a state the default behavior is to *keep* its last value; for an asynchronous signal the compiler must implement a *don't care* logic value. Designer can easily change this default behavior using this box.

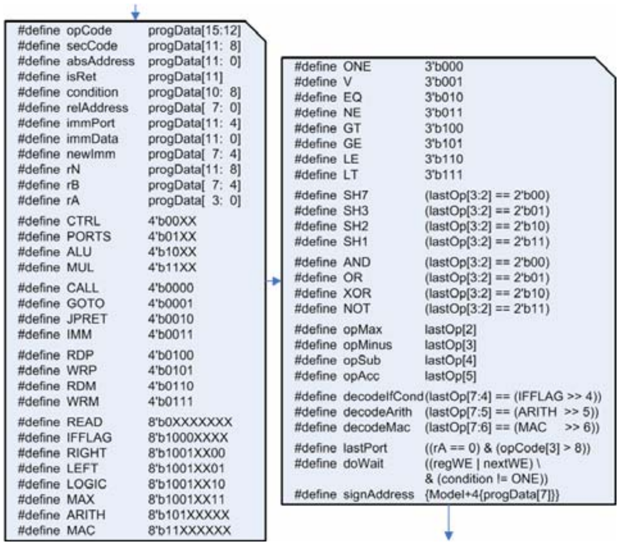
Figure 3. Default values of signals and outputs.



```
defaults
portRead  <= 0;   portWrite  <= 0;
memRead   <= 0;   memWrite   <= 0;
nextRead  <= 0;   immFF      <= 0;
nextWE    <= 0;   regWE      <= 0;
aluCE     <= 0;   macCE      <= 0;
```

The following two code boxes are a combinational instruction decoder implemented using a C-like "#define" compiler directive. Other directives are also available to include files and other purposes.

Figure 4. Instruction decoder.

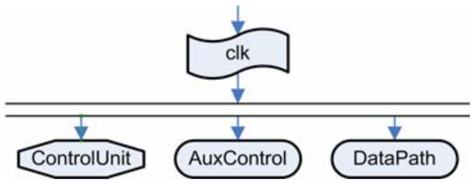


```
#define opCode       progData[15:12]
#define secCode      progData[11: 8]
#define absAddress   progData[11: 0]
#define isRet        progData[11]
#define condition    progData[10: 8]
#define relAddress   progData[ 7: 0]
#define immPort      progData[11: 4]
#define immData      progData[11: 0]
#define newImm       progData[ 7: 4]
#define rN           progData[11: 8]
#define rB           progData[ 7: 4]
#define rA           progData[ 3: 0]

#define CTRL         4'b00XX
#define PORTS        4'b01XX
#define ALU          4'b10XX
#define MUL          4'b11XX

#define CALL         4'b0000
#define GOTO         4'b0001
#define JPRET        4'b0010
#define IMM          4'b0011

#define RDP          4'b0100
#define WRP          4'b0101
#define RDM          4'b0110
#define WRM          4'b0111

#define READ         8'b0XXXXXXX
#define IFFLAG       8'b1000XXXX
#define RIGHT        8'b1001XX00
#define LEFT         8'b1001XX01
#define LOGIC        8'b1001XX10
#define MAX          8'b1001XX11
#define ARITH        8'b101XXXXX
#define MAC          8'b11XXXXXX
```

```
#define ONE          3'b000
#define V            3'b001
#define EQ           3'b010
#define NE           3'b011
#define GT           3'b100
#define GE           3'b101
#define LE           3'b110
#define LT           3'b111

#define SH7          (lastOp[3:2] == 2'b00)
#define SH3          (lastOp[3:2] == 2'b01)
#define SH2          (lastOp[3:2] == 2'b10)
#define SH1          (lastOp[3:2] == 2'b11)

#define AND          (lastOp[3:2] == 2'b00)
#define OR           (lastOp[3:2] == 2'b01)
#define XOR          (lastOp[3:2] == 2'b10)
#define NOT          (lastOp[3:2] == 2'b11)

#define opMax        lastOp[2]
#define opMinus      lastOp[3]
#define opSub        lastOp[4]
#define opAcc        lastOp[5]

#define decodeIfCond (lastOp[7:4] == (IFFLAG >> 4))
#define decodeArith  (lastOp[7:5] == (ARITH  >> 5))
#define decodeMac    (lastOp[7:6] == (MAC    >> 6))

#define lastPort     ((rA == 0) & (opCode[3] > 8))
#define doWait       ((regWE | nextWE) \
                     & (condition != ONE))
#define signAddress  {Model+4{progData[7]}}
```

After all these definitions, a box is used to specify the synchronism of this circuit. In this case there is a unique clock signal, named 'clk', but several clocks may be used instead. Then, three branches are initiated: the first one is a state machine named "ControlUnit"; the second one contains several synchronous and asynchronous components that assist at any time to the previous state machine; the last one is the data path of this processor, also described as an independent thread. Any dependence between branches may be implemented using the name of the state of each thread. This example shows how easily ASM++ diagrams may describe multi-clocked or multi-threaded circuits.

Figure 5. Parallel circuits description.



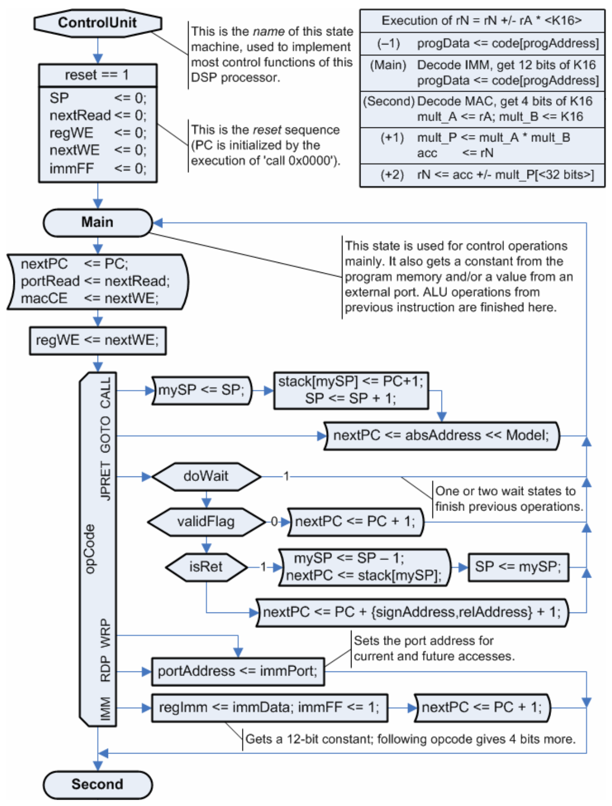
The first branch, which state variable is named 'ControlUnit' as seen on Fig. 6, begins with an asynchronous reset sequence controlled by the active high 'reset' signal. This box increases the ASM possibilities: standard diagrams cannot describe properly reset sequences.

Then, a first state named 'Main', which begins with an oval "state box", executes several overlapped operations from the previous instruction and decodes the current instruction. For 'call', 'goto', 'jp' and 'ret' instructions only one clock is needed, so the next state is 'Main' again; other instructions require a 'Second' state.

Figure 6 shows more ASM++ features:

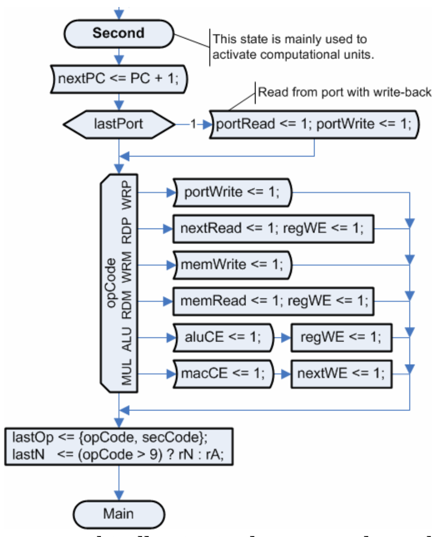
- Synchronous operations, those that are executed when the current clock cycle finishes, like "SP <= SP + 1", are described using a rectangular box anywhere. This is a difference with traditional ASM diagrams, where only unconditional operations use these boxes at the beginning of any clock cycle.
- Asynchronous operations, executed all through the current clock cycle, like "nextPC <= PC + 1", use a box with bent sides. This is a nice feature, that shows the difference in the behavior between synchronous and asynchronous signals. When Verilog language is used, the equal operator ('=') may also be used for asynchronous assertions.
- Conditions are expressed in the same way than standard ASM diagrams, but also multiple output decisions are included.
- The use of VHDL/Verilog expressions allows an easy implementation of complex functions, like a register file or a returning address stack, that need vector notation.

Figure 6. Processor control unit (I).


ControlUnit
This is the name of this state machine, used to implement most control functions of this DSP processor.
reset == 1
SP <= 0;
nextRead <= 0;
regWE <= 0;
nextWE <= 0;
immFF <= 0;
This is the reset sequence (PC is initialized by the execution of 'call 0x0000').
Execution of rN = rN +/- rA * <K16>
(–1) progData <= code[progAddress]
(Main) Decode IMM, get 12 bits of K16 progData <= code[progAddress]
(Second) Decode MAC, get 4 bits of K16 mult_A <= rA; mult_B <= K16
(+1) mult_P <= mult_A * mult_B acc <= rN
(+2) rN <= acc +/- mult_P[<32 bits>]
Main
nextPC <= PC;
portRead <= nextRead;
macCE <= nextWE;
This state is used for control operations mainly. It also gets a constant from the program memory and/or a value from an external port. ALU operations from previous instruction are finished here.
regWE <= nextWE;
opCode
CALL
GOTO
JPRET
mySP <= SP;
stack[mySP] <= PC+1; SP <= SP + 1;
nextPC <= absAddress << Model;
doWait
One or two wait states to finish previous operations.
validFlag
nextPC <= PC + 1;
isRet
mySP <= SP – 1; nextPC <= stack[mySP];
SP <= mySP;
nextPC <= PC + {signAddress,relAddress} + 1;
WRP
RDP
IMM
Sets the port address for current and future accesses.
portAddress <= immPort;
regImm <= immData; immFF <= 1;
nextPC <= PC + 1;
Gets a 12-bit constant; following opcode gives 4 bits more.
Second

The following state named 'Second', seen at Fig. 7, executes all computational instructions after receiving operands from the previous clock cycle. Actually, this state just activates all the required control signals, because data path and external devices do the real job.

Figure 7. Processor control unit (II).


Second
This state is mainly used to activate computational units.
nextPC <= PC + 1;
Read from port with write-back
lastPort
portRead <= 1; portWrite <= 1;
opCode
WRP
RDP
WRM
RDM
ALU
MUL
portWrite <= 1;
nextRead <= 1; regWE <= 1;
memWrite <= 1;
memRead <= 1; regWE <= 1;
aluCE <= 1;
regWE <= 1;
macCE <= 1;
nextWE <= 1;
lastOp <= {opCode, secCode};
lastN <= (opCode > 9) ? rN : rA;
Main

Readers are kindly invited to translate this state machine to HDL code[1], either using VHDL or Verilog.

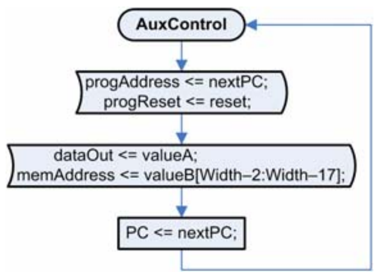
[1] During the translation process, at least two processes or always blocks are needed, one of them for all *clk-dependent* synchronous operations and the other one, unconnected from the former, for the asynchronous operations. ASM++ diagrams join both worlds.

Then, the relationship between ASM++ and HDL arises, and the advantages of using a graphical tool to design and/or document complex circuits also becomes clear.

To complete control tasks a second thread is more than convenient (see Fig. 8). Several operations must be done during or at the end of *all* clock cycles. Writing these operations in the previous thread is at least uncomfortable and prone to mistakes. Real life circuits require the possibility of writing parallel threads, but standard ASM diagrams cannot do it.

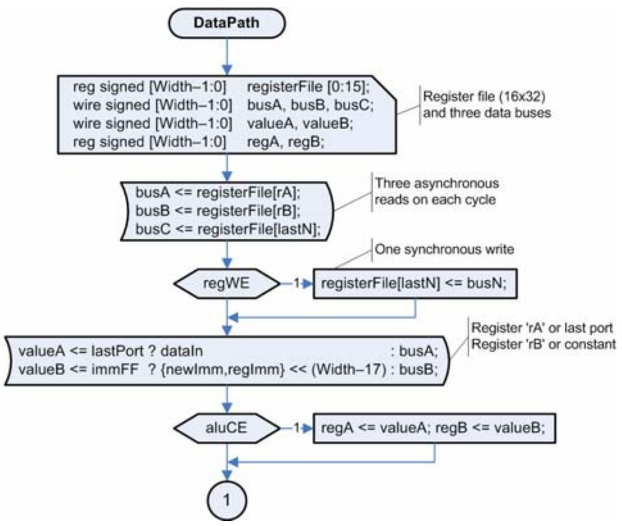
A second detail of Fig. 8 is that, from the point of view of the 'PC' signal, this is a *state-less* state machine: it needs no state at all because it has just one state. Additionally, the only reference to a clock here is the rectangular box used for 'PC'; in absence of it, this could be a *clock-less* thread, a pure-combinational circuit properly described using ASM++ diagrams.

Figure 8. Processor control unit (III).



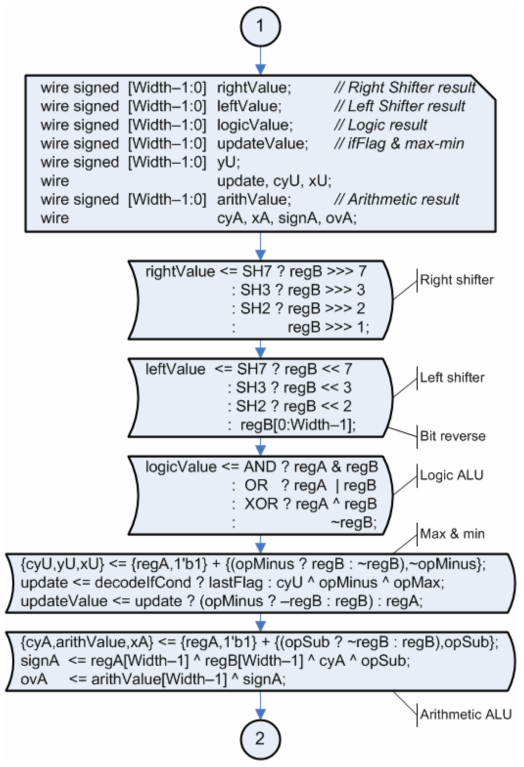
Following figures, from 9 to 13, implement the data path of this processor. First of all, a register file keeps the 32-bit values of r0 to r15 registers. Its design is based on two dual-ported distributed memories, allowing up to four asynchronous reads and one synchronous write on every clock cycle; only three reads are actually needed. During the state 'Second', if 'aluCE' signal is asserted, two operands are stored at register 'regA' and 'regB' for their operation during the following 'Main' state.

Figure 9. Processor data path (I).


DataPath
reg signed [Width–1:0] registerFile [0:15];
wire signed [Width–1:0] busA, busB, busC;
wire signed [Width–1:0] valueA, valueB;
reg signed [Width–1:0] regA, regB;
Register file (16x32) and three data buses
busA <= registerFile[rA];
busB <= registerFile[rB];
busC <= registerFile[lastN];
Three asynchronous reads on each cycle
One synchronous write
regWE
registerFile[lastN] <= busN;
Register 'rA' or last port
Register 'rB' or constant
valueA <= lastPort ? dataIn : busA;
valueB <= immFF ? {newImm,regImm} << (Width–17) : busB;
aluCE
regA <= valueA; regB <= valueB;
1

After operand selection, several computational units calculate different results throughout the clock period: a right or left shifted value, a logic or arithmetic result [4], [7], and an update value used for conditional assignments and maximum and minimum evaluation.

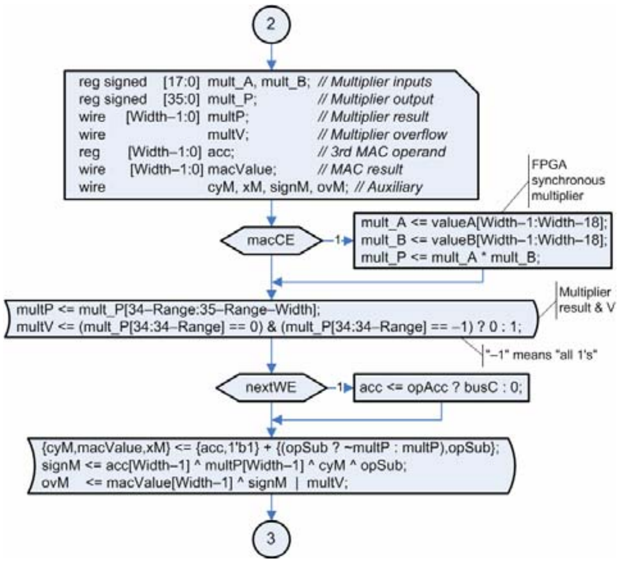
Figure 10. Processor data path (II).



```
wire signed [Width–1:0] rightValue;   // Right Shifter result
wire signed [Width–1:0] leftValue;    // Left Shifter result
wire signed [Width–1:0] logicValue;   // Logic result
wire signed [Width–1:0] updateValue;  // ifFlag & max-min
wire signed [Width–1:0] yU;
wire                    update, cyU, xU;
wire signed [Width–1:0] arithValue;   // Arithmetic result
wire                    cyA, xA, signA, ovA;

rightValue <= SH7 ? regB >>> 7
            : SH3 ? regB >>> 3
            : SH2 ? regB >>> 2
            :       regB >>> 1;

leftValue  <= SH7 ? regB << 7
            : SH3 ? regB << 3
            : SH2 ? regB << 2
            :  regB[0:Width–1];

logicValue <= AND ? regA & regB
            : OR  ? regA | regB
            : XOR ? regA ^ regB
            :             ~regB;

{cyU,yU,xU} <= {regA,1'b1} + {(opMinus ? regB : ~regB),~opMinus};
update <= decodeIfCond ? lastFlag : cyU ^ opMinus ^ opMax;
updateValue <= update ? (opMinus ? –regB : regB) : regA;

{cyA,arithValue,xA} <= {regA,1'b1} + {(opSub ? ~regB : regB),opSub};
signA  <= regA[Width–1] ^ regB[Width–1] ^ cyA ^ opSub;
ovA    <= arithValue[Width–1] ^ signA;
```

The core of this processor, a fixed-point 18x18 multiplier with 32-bit result, is described below in such a way that most synthesis tools infer a wired synchronous multiplier: it registers two operands during one clock cycle and gives the product of them at the end of the following cycle. This segmentation stage introduces a one clock latency, so a 'nop' or any dummy instruction must be used before retrieving the product result.

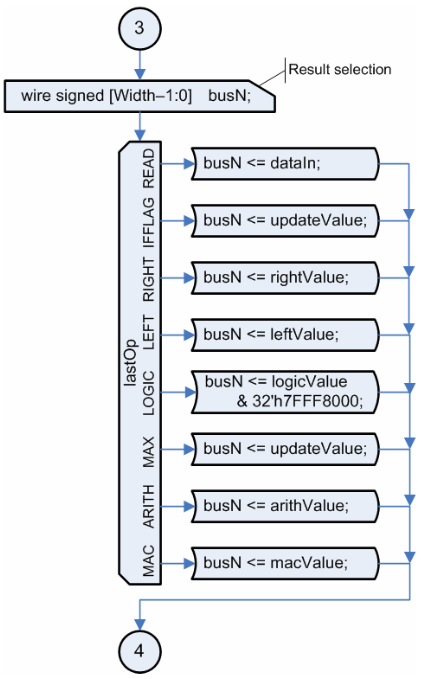
Figure 11. Processor data path (III).



```
reg signed  [17:0] mult_A, mult_B; // Multiplier inputs
reg signed  [35:0] mult_P;         // Multiplier output
wire    [Width–1:0] multP;         // Multiplier result
wire                multV;         // Multiplier overflow
reg     [Width–1:0] acc;           // 3rd MAC operand
wire    [Width–1:0] macValue;      // MAC result
wire                cyM, xM, signM, ovM; // Auxiliary

macCE –1→
mult_A <= valueA[Width–1:Width–18];
mult_B <= valueB[Width–1:Width–18];
mult_P <= mult_A * mult_B;

multP <= mult_P[34–Range:35–Range–Width];
multV <= (mult_P[34:34–Range] == 0) & (mult_P[34:34–Range] == –1) ? 0 : 1;

nextWE –1→ acc <= opAcc ? busC : 0;

{cyM,macValue,xM} <= {acc,1'b1} + {(opSub ? ~multP : multP),opSub};
signM <= acc[Width–1] ^ multP[Width–1] ^ cyM ^ opSub;
ovM   <= macValue[Width–1] ^ signM | multV;
```

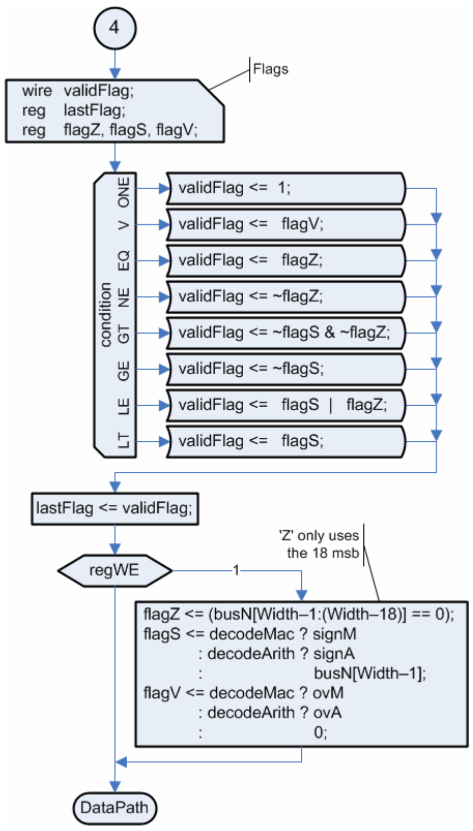
When all partial results are available, they are multiplexed in order to store the final value in the register file and to update flags. In these diagrams, it is not important if a signal like 'busN' has been used *before* its declaration (see Figs. 9 and 12).

Figure 12. Processor data path (IV).



```
wire signed [Width–1:0] busN;

lastOp
READ   busN <= dataIn;
IFFLAG busN <= updateValue;
RIGHT  busN <= rightValue;
LEFT   busN <= leftValue;
LOGIC  busN <= logicValue
              & 32'h7FFF8000;
MAX    busN <= updateValue;
ARITH  busN <= arithValue;
MAC    busN <= macValue;
```

Figure 13. Processor data path (V).



```
wire  validFlag;
reg   lastFlag;
reg   flagZ, flagS, flagV;

condition
ONE  validFlag <=  1;
V    validFlag <=  flagV;
EQ   validFlag <=  flagZ;
NE   validFlag <= ~flagZ;
GT   validFlag <= ~flagS & ~flagZ;
GE   validFlag <= ~flagS;
LE   validFlag <=  flagS | flagZ;
LT   validFlag <=  flagS;

lastFlag <= validFlag;

regWE –1→
flagZ <= (busN[Width–1:(Width–18)] == 0);
flagS <= decodeMac ? signM
       : decodeArith ? signA
       :               busN[Width–1];
flagV <= decodeMac ? ovM
       : decodeArith ? ovA
       :               0;
```

4. Conclusions

This article has presented a small and easy to understand digital signal processor developed using Verilog and ASM++ diagrams for FPGA. Throughout this paper, the capabilities of ASM++ for the development and documentation of IP modules has arisen. Additionally, supervision of complex designs would be ease when using this methodology. Compared with classic HDL description, the learning curve of ASM++ is shorter and the possibility of mixing synchronous and asynchronous signals is also a great advantage.

The proposed DSP processor executes all its instructions in one or two clock cycles, achieving up to 150 MOPS at 75 MHz on Xilinx Spartan3 devices. It introduces several new features: a variable code length between 4K and 64K, a variable range at implementation time between ±1 and ±8 for numerical values, a transparent access to constants and a built-in read with write back capability to speed up filter implementation. This processor is currently been used in power electronics applications.

5. Acknowledgments

The authors would like to acknowledge the partial financial support of *eZono AG* at Jena, Germany, *ISEND SA* at Boecillo, Valladolid, Spain, and the regional government, *Junta de Castilla y León*, under grants VA004B06 and VA021B06.

References

[1] C.R. Clare, *Designing Logic Using State Machines*, McGraw-Hill, 1973. Referenced by [5].

[2] epYme workgroup, online at *http://www.dte.eis.uva.es/epYme*, last updated on August 2007.

[3] D.D. Gajski, *Principles of Digital Design*, Prentice Hall, Upper Saddle River, NJ, 1997.

[4] J. Gray, “Designing a Simple FPGA-Optimized RISC CPU and System-on-a-Chip”, *DesignCon’2001*, online at *http://www.fpgacpu.org/gr/index.html*, 2001.

[5] S. Leibson, “The NMOS II Hybrid Microprocessor: Fusing silicon, ceramic, and aluminium with rubber baby buggy bumpers”, online at *http://www.hp9825.com/html/hybrid_microprocessor.html*, revised on August 2007.

[6] S. de Pablo et al., “A soft fixed-point Digital Signal Processor applied in Power Electronics”, *FPGAworld Conference 2005*, Stockholm, Sweden, 2005.

[7] S. de Pablo et al., “A very simple 8-bit RISC processor for FPGA”, *FPGAworld Conference 2006*, Stockholm, Sweden, 2006.

[8] S. de Pablo et al., “A proposal for ASM++ diagrams”, *10th Workshop on Design and Diagnostics of Electronic Circuits and Systems (DDECS 2007)*, Kraków, Poland, 2007.

The Effect of Dependence Graphs' Size and Complexity, in the Implementation of Processor Arrays on FPGA Devices.

Stavros Dokouzyannis
Department of Electrical and
Computer Engineering
Aristotle University of Thessaloniki,
Greece
Email: dok@auth.gr

Argiris Mokios
Department of Electrical and
Computer Engineering
Aristotle University of Thessaloniki,
Greece
Email: amok@ee.auth.gr

***Abstract*—Dependence graphs (DGs) constitute the initial step of an algorithm to a systolic array (SA) transformation. The derivation of the intermediate signal flow graph representation from the DG using proper scheduling and projection vectors, is crucial for the final form of the generated SA. In this paper, a set of DG to SA transformations and its further implementation on FPGAs are presented. Examining the generated results the implemented architectures are evaluated with respect to their constituting logic elements and their timing performance.**

I. Introduction

Since the appearance of the first computer in the early 50s, it was clear that parallel computations is an attractive alternative to sequential computations. While sequential computations are dominated by a single calculation model, i.e., von Neumman's model, that incorporates the basic principles of Turing's study in a practical design, in parallel computations there is a plethora of different processor arrays models. They can be categorized in four major types, i.e., systolic arrays that are regular arrays with synchronous data flow, wavefront arrays that have asynchronous data flow, simple instruction multiple data (SIMD) arrays, and multiple instruction multiple data (MIMD) arrays. Processor arrays are very important for a wide area of applications (e.g., digital signal processing (DSP), image processing, image compression etc.), that require high performance intensive computations, because of their regular structure and ease of hardware realization. Numerous algorithms and architectures have been developed the past 3 decades targeting these applications, some examples are given below.

In the region of DSP, Kortke [1] presented affine recurrence equations mappings onto local memory processor array systems, consisting of TMS320C40 and TMS320C44 processing elements (PEs). Lange [2] explored the use of CORDIC processors as a design element for processor arrays, implementing real time DSP applications algorithms.

In image processing, Johannesson [3], [4] developed two processor arrays architectures called radar video image processor (RVIP) and infra-red VIP (IVIP), targeted in radar image processing and autonomous vehicle navigation applications respectively. Lin [5] presented proper parallel algorithms for contour extraction and its approximation with line segments. The algorithms were implemented to MasPar MP-2 processor array, which comprises from $p \times p$ PEs and two level memory, corresponding to local memory and I/O memory. Tang [6] presented a horizontal-vertical regional integration algorithm implementation, on a processor array architecture. Finally Frimou [7] implemented a pel-recursive motion estimation algorithm in a processor array, where every PE consists off an initialization, a routing, and an updating part.

In video compression, Mayer [8] presented a video decoding architecture, which uses regular hardware and software. Baglietto [9] proposed a motion estimation block matching algorithm, implemented onto a parallel processor array, used to calculate motion estimation in compression algorithms like H.261 and MPEG-1 & 2.

Although numerous implementations were presented, most of them were targeted on custom VLSI processor arrays and only a small fraction was realized on FPGA devices. In this paper, a complete platform for the implementation of regular iterative algorithms (RIAs) onto FPGA devices is utilized in order to study the interrelation between the embedded architectures and the capacity of the targeted devices. As shown is Section III, the presented matrix by matrix multiplication algorithm implementations, reach and in a lot of cases exceed the FPGA devices' resource limits. In Section IV, the timing characteristics of the generated architectures are examined. The conclusions of this work are summarized in Section V.

II. Scope of the Work.

In this paper the effect of the DG's size and complexity, on the basic implementation logic elements and the input/output resources and on the time performance of the implemented architectures is studied. In order to examine this behavior a platform that consists of four implementation stages is utilized.

As shown in Fig. 1, at first the RIA, in DG form, is written in a text file using the graph description interchange format

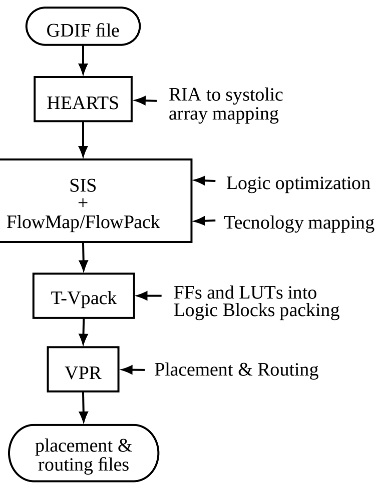


Fig. 1. Platform flow

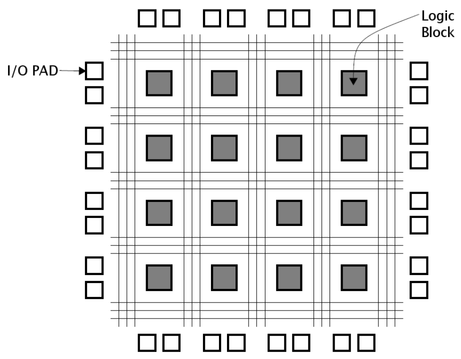
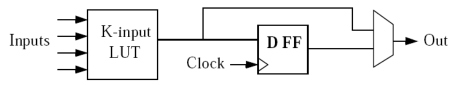
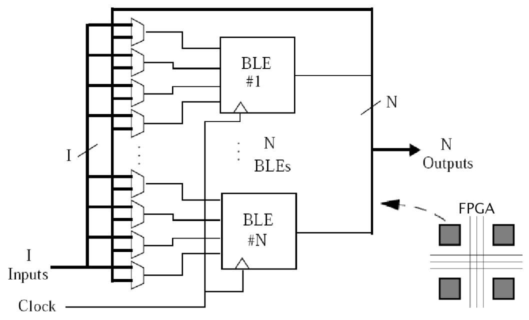


Fig. 2. Island type FPGA



(a) Basic Logic Element



(b) Cluster Based Logic Block

Fig. 3. FPGA model

TABLE I
HEARTS LUTs Results for $[0\ 0\ 1]^T - [1\ 1\ 1]^T$ vectors.

DG Size	Element bit Size				
	8	12	16	24	32
4x4x4	3504	7568	13168	28976	50928
5x5x5	5495	11855	20165	45335	79655
6x6x6	7932	17100	29724	65340	114780
7x7x7	10185	23303	40495	88991	156303
8x8x8	14144	30464	52928	116288	204224

(GDIF) [10]. HEARTS [11] reads the file, converts the DG in a systolic array and outputs a BLIF file format. SIS [12] reads the BLIF file and performs technology mapping through FlowMap and FlowPack [13]. A new BLIF file consisting of LUTs and flip flops is produced. The file is read by T-Vpack [14] and is converted into logic blocks. Next, it will be used by VPR [15], along with an FPGA architecture definition file, for the generation of placement and routing files.

An island type FPGA architecture is used, where an array of logic blocks is surrounded from interconnection lines, as shown in Fig. 2. The I/O pads are uniformly allocated in the perimeter of the device. The structure of the logic block (LB), used in the presented test-cases is of the clustered based logic block (CLB) type and is presented in Fig. 3b. Each CLB consists of four basic logic elements (BLEs), which are connected to the 16 input of the cluster. The BLE shown in Fig. 3a, consists of a 4-LUT and a register, that feeds a two input multiplexer.

In order to create the test-cases, DGs that compute the product of two matrices were generated, with sizes ranging from 4×4 to 8×8, constituting of elements with bit sizes: 8, 12, 16, 24, and 32. The DGs that perform these computations have dimensions $4 \times 4 \times 4$ - $8 \times 8 \times 8$, respectively. For every DG the implementation process was performed for 3 different couples of projection and scheduling vectors, namely: $[0\ 0\ 0]^T - [1\ 1\ 1]^T$, $[0\ 1\ 0]^T - [1\ 1\ 1]^T$, and $[1\ 0\ 0]^T - [1\ 1\ 1]^T$.

III. The Effect of the DG's size and its input bit size in the implementation modules.

In this section we examine the evolution of the basic implementation modules, i.e., the number of LUTs, and the communication resources, i.e., the number of input and output. The required data are collected, during the execution of the platform, through the usage of specifically created scripts. Tables I, II, and III, present the results derived from the application of the first pair of vectors. Each table corresponds to the results collected from the execution of the previously introduced tools HEARTS , FlowMap and FlowPack. Tab. I contains in each column the trade-off between the number of logic gates and the DG size, and in each row the trade-off between the number of logic gates and the elements' bit size of the input matrices. Tab. II and III contain in each column the trade-off between the number of LUTs and the DG size, and in each row the trade-off between the number of LUTs and the elements' bit size of the input matrices. Similar tables are formed for the other two pairs of vectors.

Closely examining these tables, it is noted that the number of used LUTs decreases, during the implementation process (starting from HEARTS and ending in FlowPack) for every projection and scheduling vector pair. On the contrary the number of utilized LUTs is increasing during the implementation of larger DGs or the increment of the number of bits

TABLE II

FLOWMAP LUTS RESULTS FOR $[0\ 0\ 1]^T - [1\ 1\ 1]^T$ VECTORS.

DG Size	Element bit Size				
	8	12	16	24	32
4x4x4	3056	6896	12272	27632	49136
5x5x5	4775	10775	19175	43175	76775
6x6x6	6876	15516	27612	62172	110556
7x7x7	9359	21119	37583	84623	150479
8x8x8	12224	27584	49088	110528	196544

TABLE III

FLOWPACK LUTS RESULTS FOR $[0\ 0\ 1]^T - [1\ 1\ 1]^T$ VECTORS.

DG Size	Element bit Size				
	8	12	16	24	32
4x4x4	2352	5104	8880	19536	34224
5x5x5	3675	7975	13875	30525	53475
6x6x6	5292	11484	19980	43956	77004
7x7x7	7203	15631	27195	59829	104811
8x8x8	9408	20416	35520	78144	136896

TABLE IV

LUTS RESOURCES % GAIN IN TECHNOLOGY MAPPING STAGE FOR $[0\ 0\ 1]^T - [1\ 1\ 1]^T$ VECTORS.

DG Size	Element bit Size				
	8	12	16	24	32
4x4x4	32.88%	32.55%	32.56%	32.58%	32.80%
5x5x5	33.12%	32.72%	31.19%	32.67%	32.87%
6x6x6	33.28%	32.84%	32.78%	32.73%	32.91%
7x7x7	29.28%	32.92%	32.84%	32.77%	32.94%
8x8x8	33.48%	32.98%	32.89%	32.80%	32.97%

TABLE V

LUTS RESOURCES % GAIN IN TECHNOLOGY MAPPING STAGE FOR $[0\ 1\ 0]^T - [1\ 1\ 1]^T$ AND $[1\ 0\ 0]^T - [1\ 1\ 1]^T$ VECTORS.

DG Size	Element bit Size				
	8	12	16	24	32
4x4x4	36.65%	35.29%	34.70%	34.07%	33.94%
5x5x5	36.13%	30.72%	34.40%	33.86%	33.78%
6x6x6	35.79%	34.66%	34.21%	33.72%	37.15%
7x7x7	35.54%	34.48%	34.06%	33.62%	33.59%
8x8x8	35.36%	34.35%	33.96%	33.54%	33.54%

that represent the elements of the input matrices.

Analyzing furthermore the contents of the tables, the percentage value of the optimization level succeeded during the technology mapping stage, is calculated. The percentage value refers to the decrement of the circuit resources initially considered as logic gates and finally as LUTs. Tab. IV and V display the percentage values for the vector pairs. The implementations derived from the application of $[0\ 0\ 0]^T] - [1\ 1\ 1]^T$ and $[0\ 1\ 0]^T - [1\ 1\ 1]^T$, have the same optimization level, because the systolic arrays that are generated have the same geometry and differ only in the direction of the applied input and derived output.

The required Input/Output resources are stored in Tab. VI and VII, where each column contains the trade-off between the summation of the I/O and the DG size, and each row contains the trade-off between the summation of the I/O and the bits size that corresponds to the utilized input data precision. The equations that calculate the numbers of input and output are

TABLE VI

INPUTS/OUTPUTS RESULTS FOR $[0\ 0\ 1]^T - [1\ 1\ 1]^T$ VECTORS.

DG Size	Element bit Size				
	8	12	16	24	32
4x4x4	321	481	641	961	1281
5x5x5	481	721	961	1441	1921
6x6x6	673	1009	1345	2017	2689
7x7x7	897	1345	1793	2689	3585
8x8x8	1153	1729	2305	3457	4609

TABLE VII

INPUTS/OUTPUTS RESULTS FOR $[0\ 1\ 0]^T - [1\ 1\ 1]^T$ AND $[1\ 0\ 0]^T - [1\ 1\ 1]^T$ VECTORS.

DG Size	Element bit Size				
	8	12	16	24	32
4x4x4	225	337	449	673	897
5x5x5	321	481	641	961	1281
6x6x6	433	649	865	1297	1729
7x7x7	561	841	1121	1681	2241
8x8x8	705	1057	1409	2113	2817

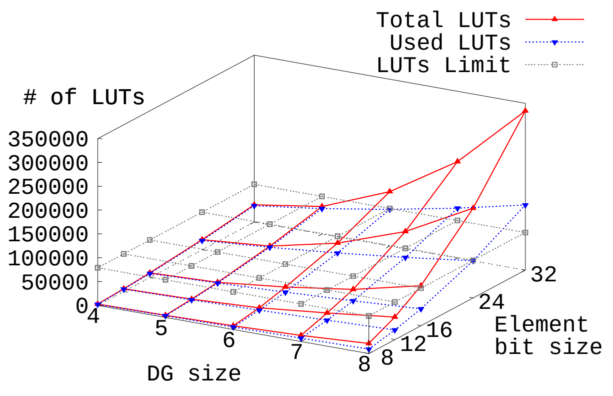
$PI = a \cdot x + 1$ and $PO = 32 \cdot a$ respectively, where a is the number of bits and x is the number input or output variables used in the systolic array. The maximum I/O pin number and the maximum number of LUTs for 4-LUT FPGA architectures, based on the current VLSI technology, is 1203 and 79040 correspondingly for ALTRERA's Startix devices [16] and 960 and 178176 for Xilinx's Virtex-4 devices [17]. According to these numbers, there are limitations imposed to the systolic array architectures that can be implemented. For the models studied in this paper, it is obvious that the if the DG is projected to $[0\ 0\ 1]^T$ direction, then it is impossible to implement products of matrices greater than 5×5 with 16-bit elements and greater that 4×4 with 24-bit elements. It is also noted that there can be no implementation of DGs between $4 \times 4 \times 4$- $8 \times 8 \times 8$ with 32-bit elements for the specified projection vector.

In the case where projection vectors $[0 1 0]^T$ and $[1 0 0]^T$ are used, the limitations are displayed in matrices with size 7×7 with 16-bit elements, 5×5 with 24-bit elements, and 4×4 for 32-bit elements.

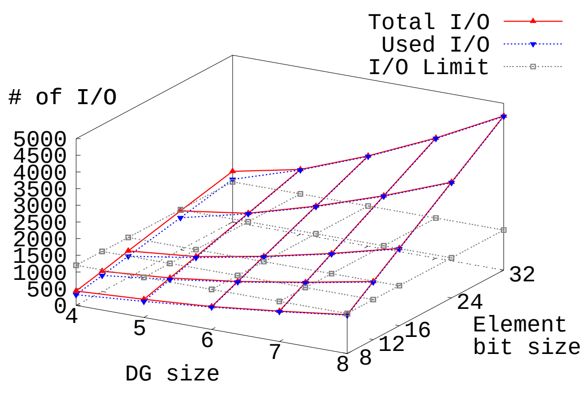
Defining as *absolute* number of LUTs and I/O for each FPGA device, the number of LUTs and I/O that are used for the implementation of the DG for every vector pair and as *used* number of LUTs and I/O, the number of LUTs and I/O that an FPGA device consists of, a utilization study is performed.

Comparing the values of absolute and used LUTs and I/O for every pair of vectors, graphs are generated that depict the evolution of the number of LUTs and the number of I/O, in accordance to the DG size and the input data bit size. The graphs are shown in Fig. 4a, 4b, 5a and 5b, where an extra surface is added to denote the upper bound of LUTs or I/O respectively. From the difference between the two graphs on each figure and the evaluation of the graphs for every pair of vectors, the advantages and disadvantages of each architecture with respect to the implementation elements and the communication resources, are derived.

The architecture that is generated from the projection vector

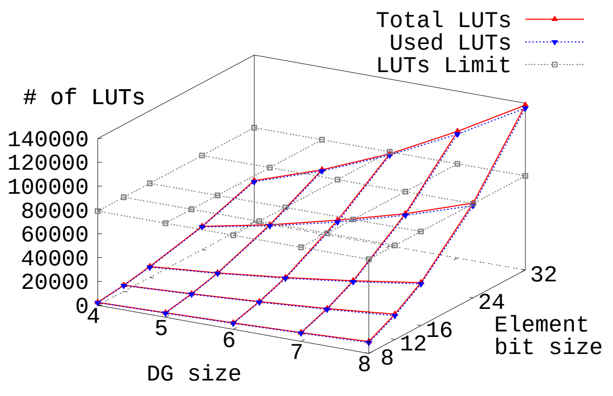


(a) LUTs utilization plot,

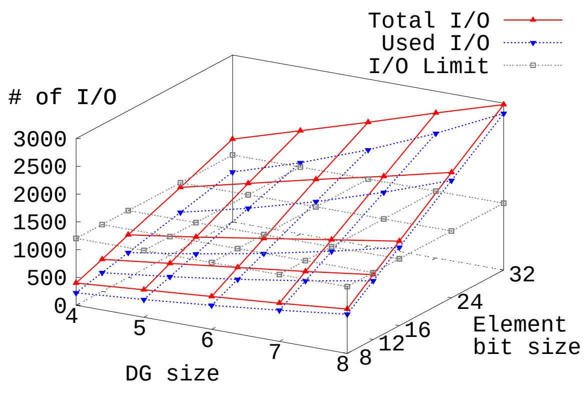


(b) I/O utilization plot,

Fig. 4. FPGA resources utilization plots for $[0\ 0\ 1]^T - [1\ 1\ 1]^T$.



(a) LUTs utilization plot,



(b) I/O utilization plot,

Fig. 5. FPGA resources utilization plots for $[0\ 1\ 0]^T - [1\ 1\ 1]^T$ and $[1\ 0\ 0]^T - [1\ 1\ 1]^T$.

$[0\ 0\ 1]^T$, as shown from Fig. 4a and 4b, uses less LUTs than than those provided by the targeted FPGA device. As the DG size increases, this phenomenon becomes more intense. Thus, this architecture causes a potential loss in the usage of logic elements that overcomes 50% of the total number of LUTs. In the contrary the level of I/Os usage begins from 80% of the total number of I/O and approximates 100%.

Evaluating the architectures that are derived from the projection vectors $[0\,1\,0]^T$ and $[1\,0\,0]^T$ by examining Fig. 5a and 5b, it is noted that they utilize less I/O than those provided by the targeted device. As the DG size decreases, this phenomenon eliminates. Thus, these architectures cause a potential I/O communication loss of 40%. Conversely, the level of used logic elements begins from 89% of total logic and remains in a high value approximating 100%.

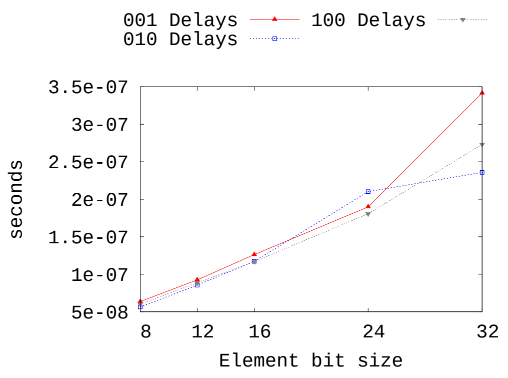
IV. The Effect of the DG's Size and its Input Bit Size in the Timing Characteristics of the Implemented Models.

In this section the timing characteristics of the implemented architectures, in terms off the total logic delay, the total net delay, and the critical path with respect to DG size and input data bit size are examined. During the creation of the test-cases and the derivation of the results, VPR was unable to process the 8×8 systolic arrays that multiply 32 bit elements. In order to have complete and solid graphic representations the values derived from the 7×7 systolic arrays that multiply same size (i.e. 32 bit) elements, were used instead.

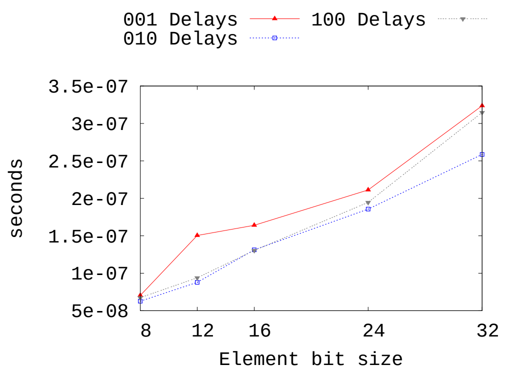
Forming $2 - Dimensional$ graphic representations having a constant DG size and a variable element bit size, the differences that are presented on every form of delay, are examined. Observing the graphical equations on Fig. 6a, 6b and 6c it emerges that the architecture produced by projecting the DG in the $[0\,0\,1]^T$ direction has a faster critical path than the other architectures.

Moreover, it is observed that although the architectural structure, before placement and routing, of the systolic arrays derived from the projection vectors $[0\ 1\ 0]^T$ and $[1\ 0\ 0]$ are almost identical (see section III), the results from placement and routing describe different timing characteristics for each architecture.

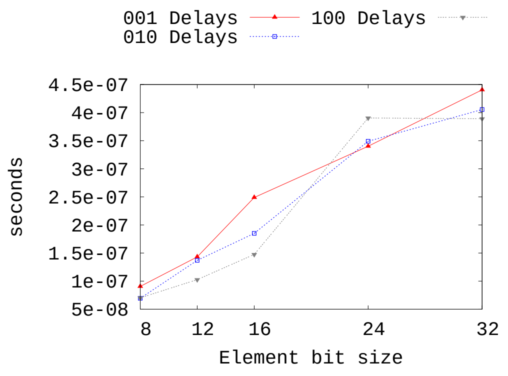
Finally, it is noted that the total net delay and the critical path have incremental tendencies, with respect to the incre-



(a) 4x4x4 DG,

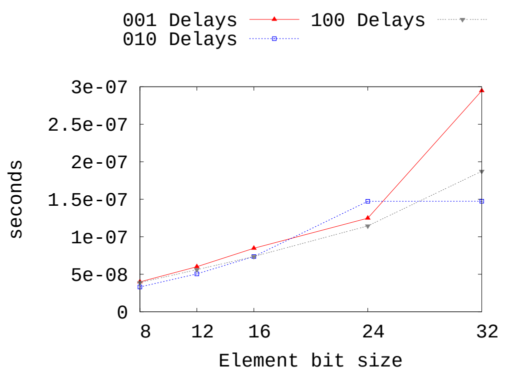


(b) 6x6x6 DG,

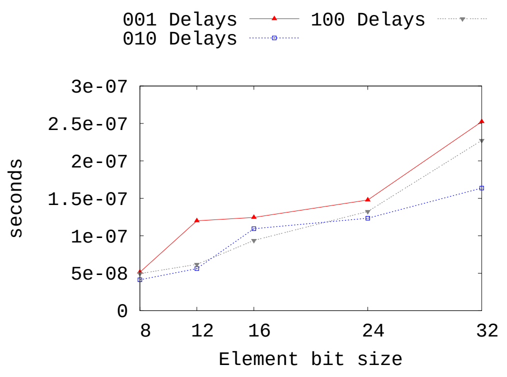


(c) 8x8x8 DG,

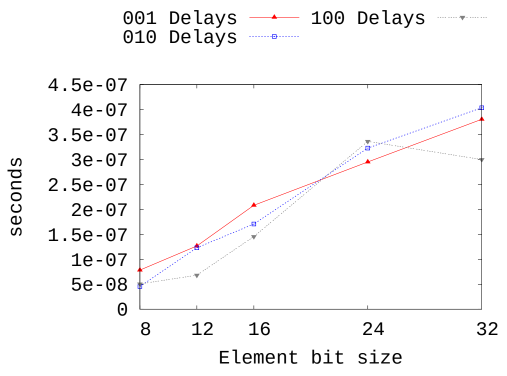
Fig. 6. FPGA critical path plots for $[0\,0\,1]^T - [1\,1\,1]^T$, $[0\,1\,0]^T - [1\,1\,1]^T$ and $[1\,0\,0]^T - [1\,1\,1]^T$ vectors.



(a) 4x4x4 DG,



(b) 6x6x6 DG,



(c) 8x8x8 DG,

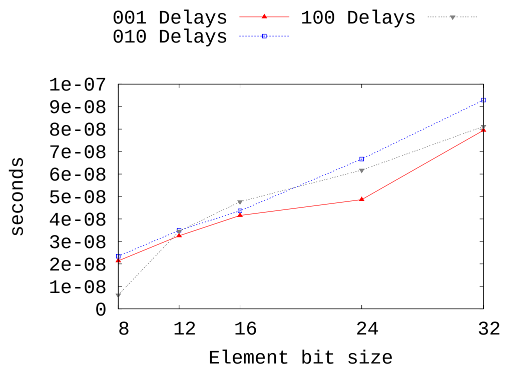
Fig. 7. FPGA net delay plots for $[0\,0\,1]^T - [1\,1\,1]^T$, $[0\,1\,0]^T - [1\,1\,1]^T$ and $[1\,0\,0]^T - [1\,1\,1]^T$ vectors.

ment of the DG size and/or the increment of the elements bit size. Conversely, the total logic delay decreases with respect to the increment of the elements bit size (Fig. 8b and 8d) and increases with respect to the increment of the DG size (Fig. 8a and 8c). This phenomenon might be a result of the direct association that exists between the BLEs and the logic delay. Thus, a potential decrement of the BLE levels, that is faster than the increment of the BLE's delay, could have as a result the decrement of logic delay.

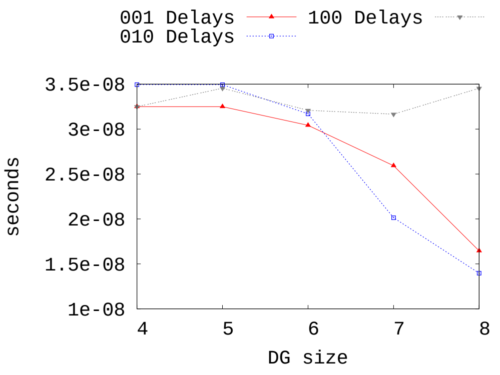
V. Conclusions

The existence and the usage of a complete platform for the transformation of regular algorithms into processor arrays and their implementation onto FPGA devices, has many advantages. During the implementation process numerous results are derived, that with certain processing generate valuable feedback information to the designer. Acquiring this knowledge, important decisions like the type of the architecture that is going to be selected, in order to fulfil the design goals, are easily made. Thus, it is possible to explore all the possible architectures, based on the requirements for constraints in logic and communication resources (LUTs, I/O, etc.), the demand for high speed designs or the case where a trade-off of these conditions is required.

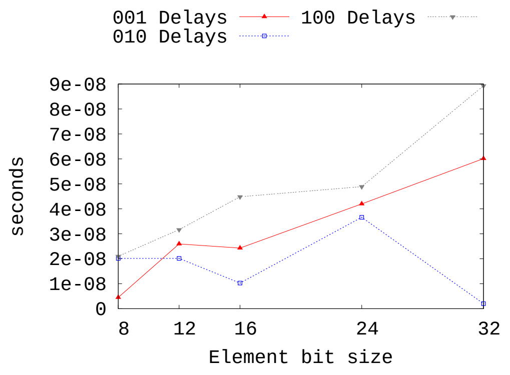
During the experimentation with different models, the need for the incorporation of DG partitioning methods into HEARTS was detected, in order to be able to design large and complex architectures into current FGPA devices. This, along with the



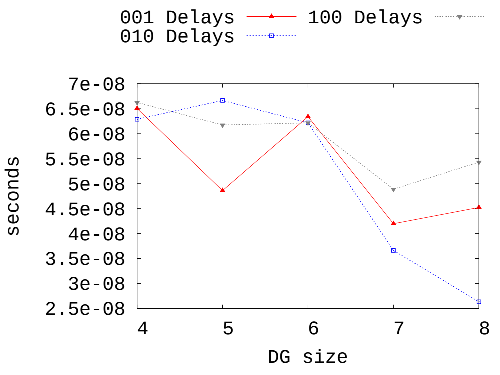
(a) 5x5x5 DG,



(b) 12 bit input data,



(c) 7x7x7 DG,



(d) 24 bit input data,

Fig. 8. FPGA logic delay plots for $[0\ 0\ 1]^T - [1\ 1\ 1]^T$, $[0\ 1\ 0]^T - [1\ 1\ 1]^T$ and $[1\ 0\ 0]^T - [1\ 1\ 1]^T$ vectors.

enhancement of the library with more PEs models are going to be the main future objectives for the development and improvement of the platform.

References

[1] M. Kortke, D. Fimmel, and R. Merker, “Parallelization of algorithms for a system of digital signal processors,” in *Proc. 25th EUROMICRO Conference*, vol. 1, Sep.8-10 1999, pp. 46–50.

[2] A. de Lange, A. van der Hoeven, E. Deprettere, and P. Dewilde, “An application specific IC for digital signal processing: the floating point pipeline CORDIC processor,” in *Euro ASIC ’90*, May29/1Jun. 1990, pp. 62–67.

[3] M. Johannesson and M. Gokstorp, “Video-rate pyramid optical flow computation on the linear SIMD array IVIP,” in *Proc. Computer Architectures for Machine Perception, CAMP ’95*, Sep.18-20 1995, p. 280287.

[4] M. Johannesson, A. Astrom, and P. Ingelhag, “The RIVP image processor array,” in *Proc. Computer Architectures for Machine Perception*, Dec.15-17 1993, pp. 385–392.

[5] C. Lin, V. Prasanna, and A. Khokhar, “Scalable parallel extraction of linear features on MP-2,” in *Proc. Computer Architectures for Machine Perception*, Dec.15-17 1993, pp. 352–361.

[6] Y. Tang, T. Li, and S. Lee, “VLSI implementation for HVRI algorithm in pattern recognition,” in *Proc. of the Second International Conference on Document Analysis and Recognition*, Oct.20-22 1993, pp. 460–463.

[7] E. Frimou, I. Driessen, and E. Deprettere, “Parallel architecture for a pel-recursive motion estimation algorithm,” *IEEE Trans. Circuits Syst. Video Technol.*, vol. 2, no. 2, pp. 159–168, Jun. 1992.

[8] A. Mayer, “The architecture of a processor array for video decompression,” *IEEE Trans. Consum. Electron.*, vol. 39, no. 3, pp. 565–569, Aug. 1993.

[9] P. Baglietto, M. Maresca, A. Migliaro, and M. Migliardi, “Parallel implementation of the full search block matching algorithm for motion estimation,” in *Proc, International Conference on Application Specific Array Processors*, Jul.24-26 1995, pp. 182–192.

[10] A. Mokios, “GDIF: A graph description interchange format,” Aristotle University of Thessaloniki, Tech. Rep., 2006.

[11] ——, “HEARTS: A system for the automatic transformation of dependence graphs to systolic arrays,” Aristotle University of Thessaloniki, Tech. Rep., 2006.

[12] E. Sentovich, K. Singh, L. Lavagno, C. Moon, R. Murgai, A. Saldanha, H. Savoj, P. Stephan, R. Brayton, and A. Sangiovanni-Vincentelli, “SIS: A system for sequential circuit synthesis,” Berkeley Electronics Research Laboratory, Tech. Rep., 1992.

[13] J. Cong and Y. Ding, “Flowmap: An optimal technology mapping algorithm for delay optimization in lookup-table based FPGA designs,” *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 13, pp. 1–13, Jan. 1994.

[14] A. S. Marquardt, V. Betz, and J. Rose, “Using cluster-based logic blocks and timing-driven packing to improve FPGA speed and density,” in *Proc. of the 1999 ACM/SIGDA seventh international symposium on Field programmable gate arrays*, Monterey, California, Feb.21-23 1999, pp. 37–46.

[15] V. Betz and J. Rose, “VPR: A new packing, placement and routing tool for FPGA research,” *Field-Programmable Logic and Applications*, pp. 213–222, 1997.

[16] ALTERA, *Stratix Architecture*, ALTERA Corporation, jul 2005. [Online]. Available: www.altera.com

[17] Xilinx, *Virtex-4 User Guide*, Xilinx, Inc., sep 2005. [Online]. Available: www.xilinx.com

Busy Generation in a large Trigger Based Data Acquisition System

M. Munkejord[*] A. Stangeland[*] J. Alme[*] W. Rauch[†] M. Richter[*]
A. Rossebø[*] D. Röhrich[*] C. Soos[‡] K. Ullaland[*]

Abstract

This paper gives an overview of a specific trigger and data acquisition system used in experimental nuclear physics, and describes one of its many components, which generates the busy signal. It is a FPGA based device that continuously keeps track of the number of issued triggers and computes the number of free buffers in the Front End Electronics.

I Introduction

The Large Hadron Collider at CERN accelerates two separate, circular beams of nuclei. The two beams move in opposite directions and at four points they intersect, allowing for collisions. ALICE (A Large Ion Collider Experiment) [1] is placed at one of these points and comprises several detectors. Recording and transfer of event data will be controlled through trigger signals, which are based on inputs from fast detectors. The Time Projection Chamber (TPC) [2] is one of the main tracking detectors in ALICE. It has approximately 560000 channels and generates data at a rate of up to 25 GB/s. For Lead-Lead collisions the maximum interaction rate will be about 8 kHz. In proton-proton collisions it will be higher, about 200 kHz in ALICE. Not all interactions will be recorded and kept for later analysis, and it is the trigger system that controls which. On average the collision rates will therefore be somewhat higher than the transfer rate to the Data Acquisition System (DAQ). For this reason the detector Front End Electronics (FEE) [3] has some buffer memory. To prevent overflow in the FEE buffers, a mechanism to halt the issuing of new triggers is required. This is what is referred to as busy generation and will be provided by a dedicated device called the Busy Box. The Busy Box will be used in several of the detectors of ALICE and it is the subject of this paper.

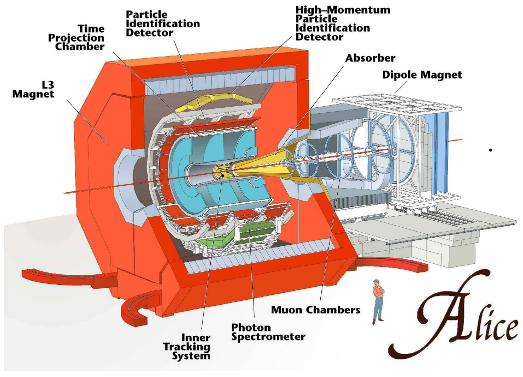


Figure 1: Illustration of ALICE

II The Trigger System

ALICE has one Central Trigger Processor (CTP) [7]. It receives information from all sub detectors and makes decisions on what triggers to issue. All

[*]Department of Physics and Technology, University of Bergen, Norway
[†]University of Applied Sciences, Frankfurt, Germany
[‡]CERN, European Organization for Nuclear Research, Geneva, Switzerland

triggers are forwarded to the Local Trigger Units which distribute them to the FEE over an optical fiber channel. The global system clock will be distributed over the same fiber. This clock signal drives all of the digital electronics in the detector and runs at the nominal bunch crossing rate of 40.08 MHz. The clock is also used as reference when creating Event IDs for collisions. Event IDs will be distributed with the triggers and makes it possible to compare data from different sub detectors when analyzing events. Event IDs also play an important role in the busy handling, as will be explained later.

The hardware trigger system for ALICE has three levels - Level 0, Level 1 and Level 2, and they are issued in sequence. A trigger sequence is started by a Level 0 trigger, which will be issued once a collision has been detected. Some time after that a Level 1 trigger will be issued if the collision satisfies certain conditions. If not the Level 1 trigger is suppressed, the trigger sequence is aborted and any data recorded so far discarded. Provided a Level 1 trigger was issued, a Level 2 trigger will be issued. The Level 2 trigger will indicate whether the event was accepted or not. If the event was accepted the FEE will mark the data in its buffers for transmission to the DAQ system. The DAQ system will receive the event data whenever there is capacity available. If a Level 2 Reject trigger is issued then FEE will overwrite its buffer when new triggers are received.

The TPC is constructed like a barrel filled with gas (see figure 1). When particles from a collision travels through the TPC, they will ionize the gas in their path leaving a trail of ionized atoms. Electric fields will cause the freed electrons to drift towards the ends of the barrel where they can be detected. To fully record an event the TPC requires about 90 μs. This makes the TPC a slow detector and new collisions can occur while there still are drifting electrons from a previous collision. However, during analysis one is able to distinguish up to a certain number of events, the number depending amongst other things on the quality of the reconstruction algorithms. If too many collisions occur after a trigger has been issued, the CTP will issue a Level 2 Reject trigger and the data will be discarded as explained earlier. This feature is called the past-future protection and is meant to discard data from events that can not be analyzed.

III Busy Handling

The task of the Busy Box is to let the trigger system know when the detector is busy and can not handle new trigger sequences. As long as the busy signal is asserted, the CTP will not issue additional trigger sequences. The generation of the busy signal is a logical OR between two separate processes inside the Busy Box. One is a simple timer started whenever a Level 0 trigger is received. In the case of the TPC the timer is set to approximately $90\mu s$, which is the time it takes to record one event. The other process will flag busy when all buffers on the FEE are occupied.

If a Level 2 Accept is issued for an event, the FEE will tag the data with the Event ID and push it over optical fibre links to DAQ computers. These are regular PCs with special data adapters called D-RORCs (DAQ-Read Out Receiver Card) connected to a PCI bus. Instead of communicating directly with the FEE to find the number of buffers in use, the Busy Box queries the D-RORCs. Once a D-RORC has received the data for an event from the FEE, it extracts the Event ID and transmits it upon request to the Busy Box over LVDS lines. The Busy Box also extracts the Event ID from the Level 2 Accept trigger, but stores it in a local queue. Once an Event ID enters the queue, the Busy Box will start polling the D-RORCs and compare the Event ID from the trigger with that from every D-RORC. If all the Event IDs match it can be safely assumed that all the corresponding buffers are freed, and the used buffers counter will be decremented. In this way the number of free FEE buffers can be calculated indirectly.

Traditionally the FEE in the detectors has generated its own busy signals. For the TPC alone, however, there are more than 4000 Front-End Cards but only 216 D-RORC cards. Communicating with the D-RORCs therefore significantly reduces the need for connections. Also, the D-

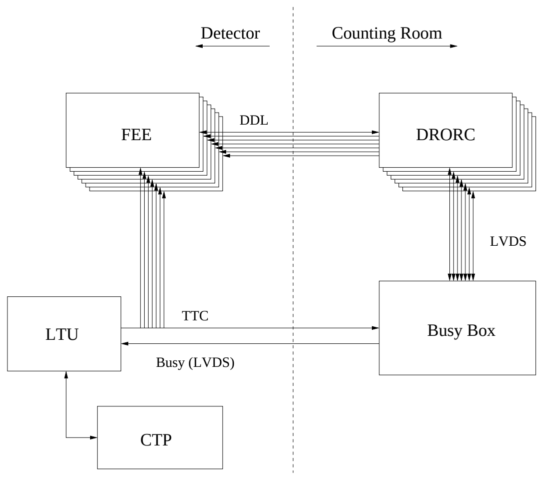


Figure 2: Illustration of Busy Box concept

RORC cards are placed in a counting room, away from the radiation environment close to the detector. By also placing the Busy Box in the same counting room easy access is assured.

IV Busy Box

In the case of the TPC, the Busy Box needs to communicate with the 216 D-RORCs over 15 meters TP (Twisted Pair) cables with RJ-45 connectors. Many of the other sub detectors have fewer data links and hence, fewer D-RORCs. For this reason the Busy Box is made modular (see figure 3). The motherboard has 40 ports for RJ-45 connectors. If more ports are needed, mezzanine cards with 48 ports can be attached with ribbon cables. The boards/cards are built in standard 19" rack cases up to five units in height. The logic resources are provided by one or two Virtex-4 FPGAs, depending on the number of ports required. The Virtex-4 FPGA in the ff1148 package was chosen because it has many IO pins, supports LVDS and supports programming by SelectMAP [6].

Attached to the motherboard is a DCS card. The DCS card is part of the DCS (Detector Control System) which monitors, configures and controls most of ALICE. The DCS card is mainly composed of an Altera EPXA1 (containing a 32 bit ARM processor), 8 MB Flash ROM, 32 MB SDRAM and an Ethernet transceiver. With these components it is able to run a lightweight version of Linux. Device drivers for Linux have been developed so that programming the FPGA with SelectMAP from a remote location is possible. This feature, although very handy for the Busy Box, was initially developed for the FEE which resides inside the detector and is unreachable once the accelerator has been started. The DCS card also has a 16 bit wide bus interface to both FPGAs, allowing software to access memory mapped registers inside the FPGAs. The DCS board provides connectivity to the trigger system and the Detector Control System.

The main requirements for the firmware are to provide communication with the D-RORCs and an interface to the DCS bus and triggers. In addition it will do most of the work of processing the incoming messages from the D-RORCs. It is essential to implement as many of the low-level functions in firmware as possible since it is faster than the software. There will be two versions of the motherboard, with one or two FPGAs. The first FPGA is connected to the first 120 of the RJ45 ports and the second to the 96 remaining. Since the number of ports will vary for different Busy Boxes, the firmware is designed to be scalable at compile time (by generics) to include any number of ports from 1 to 120. The two FPGAs will operate in parallel, with some simple logic in the first FPGA to coordinate the busy-signal.

Every received message from the D-RORCs will be stored in memory that is available to the

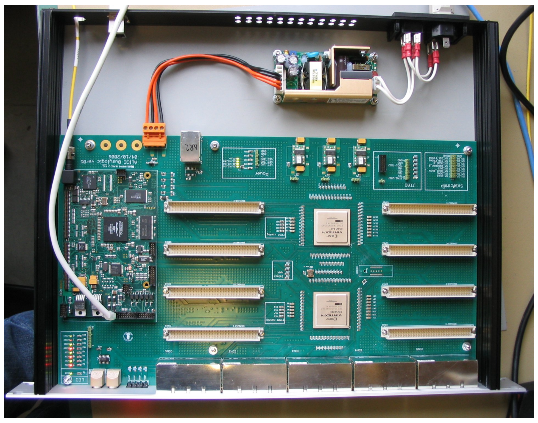


Figure 3: Picture of the inside of the Busy Box

software. The firmware also provides registers for transmitting messages to any or all of the connected D-RORCs. This allows software to communicate with the D-RORCs directly so that higher level error handling can be done in software. It is also very useful for debugging in the development phase.

Messages from the D-RORCs will also be pushed into a FIFO queue for processing by the firmware. Internal status registers for each D-RORC will be updated as messages are processed. The information in these registers will be used to determine when all D-RORCs have received data for the current Event ID or, if not, determine the next appropriate action.

As described earlier, the Busy Box will request the Event IDs from the D-RORCs. It will receive messages from all active D-RORCs containing the requested Event ID or a message saying that the Event ID has not been received yet. The Busy Box will wait until it has received messages from all D-RORCs or until a programmable timeout runs out and then re-request from those that had not received the Event ID. The firmware will retry this procedure a few times before it sets appropriate error registers and allows software to resolve the error or report it to the DCS.

The communication logic on both sides (Busy Box and D-RORC) runs on 200 MHz. Dedicated hardware inside the Virtex-4 called Digital Clock Managers are used to generate this clock in the Busy Box. The D-RORCs are referenced to the clocks of their host computer. This means that the two devices do not share clock source and clock skew and jitter noise is to be expected. A protocol that includes a bit clock in the encoded signal is desirable but due to the large number of receivers that have to be implemented into a single FPGA, Non-Return-to-Zero encoding is used. Currently, the receivers utilizes 5x oversampling which gives a bit rate of 40 Mbps. The receivers will push samples into a shift register long enough to contain samples for a complete word. When the receiver sees valid start and stop bits in the samples, it will use majority gates to determine the value of each bit and store the resulting bits in an output buffer. Parity checks are also implemented to maintain data integrity. A message from a D-RORC to the Busy Box is 48 bits. To make the protocol more tolerant of jitter and keep the receivers small, the 48 bits are transmitted as 3 times 16 bit words (with a very short timeout between the words). This allows the receiver to resynchronize to the bit stream more often. It also reduces the probability that noise from floating inputs produce garbage data by accident because it is less likely that this noise will produce three valid words consecutively.

V VERIFICATION

The design has been tested in simulations with the QuestaSim software. For this purpose testbenches has been written in VHDL that emulates the devices that the Trigger Busy Box firmware will interface with. For some of the emulated devices, a dedicated VHDL entity has been written, others are emulated by VHDL procedures that drives the signals of the interface. A main test sequence process calls procedures that controls the emulators to interact with Busy Box firmware. The main test sequence can easily be modified to simulate specific scenarios. The testbench does not automatically verify the result but gives the opportunity to study the functional operation of the design in operation.

The first priority of the hardware tests was to verify a reliable communication between the Busy Box and the D-RORC. Several test setups have been used in the different stages of development. The first was a *loopback* test where the Busy Box transmitted messages to itself through a TP cable. By using the DCS board to access registers of the FPGA, messages can be sent, and the received messages can be read out and verified by software.

After some modifications to the firmware the Busy Box was brought to CERN for testing with the D-RORC. These tests were concluded with a "proof-of-concept" test where software running on the DCS board controlled the communications of the Busy Box. The test included retrieval of an event ID form the D-RORC and successfully comparing it with the event ID which was sent from the LTU in emulator mode.

Further test of the communication has been performed with another FPGA based device were firmware have been developed specifically to emulate the D-RORC in the absence of the real D-RORC and the remaining components of a real test setup.

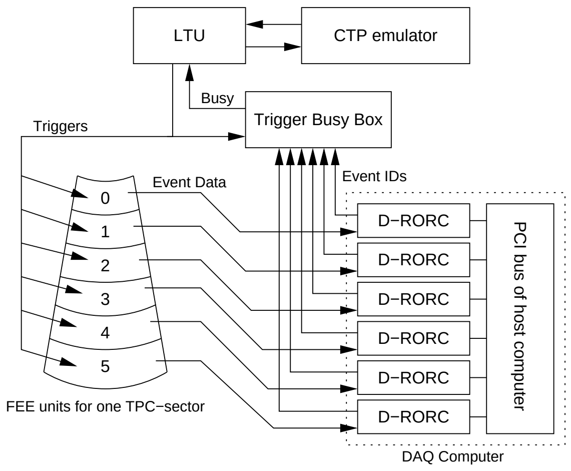


Figure 4: Illustration of test setup.

Integration tests at CERN

Recently tests have been performed with the current Busy Box design at CERN in a real test environment, including real components and several channels. The setup is illustrated in figure 4. On the detector side four complete FEE-units have been used simultaneously (the TPC has a total of 216 FEE-units). The FEE sampled floating inputs instead of real detector signals to simulate data, and on the trigger side a real LTU has been used. The LTU receives the BUSY-output from the Busy Box and passes it on to the CTP. Trigger inputs to the LTU will come from the CTP in the final setup, but so far a CTP emulator has been used instead. The CTP emulator will issue triggers at a variable rate, as is expected in the real system under normal operation. In the test the Busy Box verifies that all event data has been successfully transfered to the D-RORCs by comparing the Event IDs from the D-RORCs with the Event IDs from the trigger system.

VI CONCLUSION

The modular design of the Busy Box and its scalability makes it possible to use it with several ALICE sub-detectors. This also allows independent testing of different functionalities and makes it easy to add new or modify existing ones. Both during development and integration of the Busy Box in a detector system, the combination of software and firmware gives flexibility. So far laboratory tests have been performed to verify basic functionality, and error handling will be added. Further commissioning tests using more channels will be performed in the near future.

REFERENCES

[1] ALICE Collaboration, *Technical Proposal For A Large Ion Collider Experiment at the CERN LHC.* CERN/LHCC 1995-71, 1995.

[2] ALICE Collaboration, *Technical Design Report of the Time Projection Chamber*, CERN/LHCC 2000-001, ALICE TDR 7, 7 January 2000. ISBN 92-9083-155-3 https://edms.cern.ch/file/398930/1/ALICE-DOC-2003-011.pdf

[3] L. Musa et al., *The ALICE TPC Front End Electronics*, in proc. of the IEEE Nuclear Science Symposium, Portland, October 2003.

[4] ALICE Collaboration, *Technical Design Report of the Photon Spectrometer (PHOS)* CERN/LHCC 99-4, ALICE TDR 2, 5 March 1999. ISBN 92-9083-138-3 https://edms.cern.ch/file/398934/1/Cover-Contents.pdf

[5] Rossebø Anders, *BUSY-logikk for ALICE TPC*, Master thesis, University of Bergen, 2006.

[6] Xilinx Inc., *Virtex-4 User Guide v.1.5*, January 2006.

[7] D. Evans, S. Fedor, G. T. Jones, P. Jovanović, A. Jusko, I. Králik, R. Lietava, L. Šándor, J. Urbán and O. Villalobos Baillie for the ALICE collaboration. http://lhc-workshop-2005.web.cern.ch/lhc-workshop-2005/ParallelSessionB/51-OrlandoVillalobosBaillie.pdf

[8] ALICE collaboration, *Technical Design Report of the Trigger, Data Acquisition, High-Level Trigger and Control System*, CERN-LHCC-2003-062, ALICE TDR 010, CERN, 2004. ISBN 92-9083-217-7. https://edms.cern.ch/document/456354/

[9] Wiki-page of the Experimental Nuclear Physics group at the Department of Physics and Technology at the Univerity of Bergen: http://web.ift.uib.no/k̃jeks/wiki/

[10] J. Alme, *TTC receiver requirement specification v1.1*, University of Bergen, 02.03.2007.

THINKING OUTSIDE THE FLOW: CREATING CUSTOMIZED BACKEND TOOLS FOR XILINX BASED DESIGNS

Andreas Ehliar [*]

Department of Electrical Engineering
Linköping University
Sweden
email: ehliar@isy.liu.se

Dake Liu

Department of Electrical Engineering
Linköping University
Sweden
email: dake@isy.liu.se

ABSTRACT

This paper is intended to serve as an introduction to how to build a customized backend tool for a Xilinx based design flow. A Python based library called PyXDL is presented which allows a user to manipulate XDL files which contain a placed and routed design. Three different tools are presented which uses this library, ranging from a simple resource utilization viewer to a tool which will insert a logic analyzer into an already routed design, thus avoiding a costly complete rerun of the place and route tool.

1. INTRODUCTION

Traditionally, users are not very interested in the inner workings of the FPGA tool chain they are using. As long as everything is working correctly there is no perceived need to invest time and effort on learning about obscure implementation details. Although most users have probably looked at a routed design in for example Xilinx' FPGA editor relatively few users have modified such a design.

There are however large opportunities for those who are interested in inspecting and modifying placed and routed designs. For example, a design viewer could be constructed that not only shows the slices of the design, like the floorplanner does, but also figures out the functionality of a slice and shows a symbol for a mux, adder, inverter, and so on. This will allow a user to quickly see if the synthesizer has created reasonable logic without having to load the FPGA editor which usually shows much more detail than necessary.

In terms of modifying a placed and routed design, most users are probably interested in tools that are helpful for debugging a design such as instrumenting a design to improve the visibility of internal signals. The FPGA editor has included functionality to insert probes into a design and route those signals to external pins for a long time and the ChipScope [1] product has improved on this functionality by allowing the user to insert a full logic analyzer into the FPGA.

Finally, when the usage of partial reconfiguration of FPGAs is more widespread it is likely that already placed and routed designs will have to be modified before deployment.

This paper presents a simple way to write useful programs capable of inspecting and modifying placed and routed Xilinx designs. The used method is to use the *xdl* tool to translate Xilinx proprietary NCD (Native Circuit Description) files into XDL (Xilinx Design Language) text files which can easily be processed by an application. A Python library called PyXDL has been developed to analyze and modify XDL files and three different backend tools written in Python has been written to demonstrate the capabilities of this library. The first tool can take a design and report the resource utilization of individual modules in the design. The second tool is a design viewer capable of showing the type of logic in each LUT as described above. The final tool allows a logic analyzer core to be inserted into an already routed design and present a user interface over RS232.

While it might seem esoteric and cumbersome to write your own backend tool the main parts of the Python library and tools described in this paper were actually written over a period of less than two weeks (except for the logic analyzer core which was already written for another project where it had to be manually instantiated in the RTL source code). It is therefore feasible for even smaller developers to write their own customized tools and we hope that this paper might serve as an inspiration for like-minded developers.

2. RELATED WORK

As previously mentioned, the FPGA editor included in ISE can show a design in more detail than most users care for. It is also possible to change the design although this is probably impractical for larger changes. There is also a command line version of the FPGA editor available called *fpga_edline*

[*]Funded by the Stringent research center of the Swedish Foundation for Strategic Research

which is capable of executing scripts created by the FPGA editor.

Unfortunately there is no documented way to control the FPGA editor from a user written program. The included scripting support is just a way to repeat previously defined commands, the script language is not a complete programming language. This makes it unsuitable for an application that needs to read data from a design as opposed to making changes to a design at fixed locations.

A much more interesting alternative is the JBits SDK [2] from Xilinx. This allows Xilinx designs to be manipulated from Java. In fact, it probably contains all the functionality that a user could want in terms of design manipulation. It isn't publicly available and users have to ask for access to it. The main drawback is that JBits has been discontinued and there is no support at all for newer FPGAs in it (newer than Virtex-II) and there seems to be little interest from Xilinx to add such support. In fact, if JBits was publicly available with support for all new FPGAs from Xilinx, there wouldn't have been any need to write this paper.

Finally, abits [3] is a tool similar in spirit to JBits which allows Atmel bit streams to be manipulated.

3. THE XDL FORMAT

The XDL file format is an ASCII based translation of Xilinx' proprietary NCD file format. It will typically contain two types of statements, instances and nets. An instance can be any logic element in the FPGA such as for example a slice, ram block, or DSP block. It may or may not be placed at a certain location. A net statement will describe the name of a certain net and the instances it is connected to. It may also contain routing information. An example of a very simple XDL file is shown in Figure 1.

A drawback of the XDL file format is the scarcity of documentation. Earlier releases of ISE such as 6.3 contained written documentation about the file format [4]. Unfortunately this documentation has been removed in later versions of ISE. Even so, some details of the XDL format wasn't documented in 6.3 either. Luckily some basic information about the format is included in every XDL output file created by the *xdl* tool unless the *-noformat* switch is given.

4. PYXDL - PYTHON BASED XDL MANIPULATION LIBRARY

A Python based library called PyXDL has been developed to simplify development of backend applications. The basic idea behind the library is to convert a placed and routed design into XDL by using the *xdl* tool included in ISE. This file can be modified as required and converted back into Xilinx native NCD format. This allows small changes to be made to a design without requiring a complete and often time consuming synthesize, placement, and routing iteration. This is accomplished by telling *par* (the place and routing tool) to only route un-routed nets and only place unplaced instances. (The guide-file feature of par is used for this purpose.) This flow is illustrated in Figure 2.

```
net "simple_net" ,
  outpin "slice1" XQ ,
  inpin  "slice2" BX ,
;

inst "slice1" "SLICEL",unplaced  ,
 cfg "BXINV::BX CEINV::CE CLKINV::CLK
      DXMUX::BX FFX:slice1_r:#FF
      FFX_INIT_ATTR::INIT0" ;

inst "slice2" "SLICEL",unplaced  ,
 cfg "BXINV::BX CEINV::CE CLKINV::CLK
      DXMUX::BX FFX:slice2_r:#FF
      FFX_INIT_ATTR::INIT0" ;
```

Fig. 1. An example of a simple XDL file which shows two slices each containing one flip flop connected by a wire.

4.1. Constraints

One problem which occurs when merging two designs, which isn't immediately obviously when looking at the XDL files, is the constraints files. The timing constraints in these must also be merged if reliable timing estimates is expected.

4.2. Resource analyzer script

The design resource analyzer is a small tool written for a designer who wants to know the resource utilization of a certain module or modules in larger design. One way to figure this out is to synthesize that particular module separately. This method may or may not work depending on the properties of the larger design. For example, if the synthesizer can determine that only relatively few values can appear on a certain input port of a module included in a larger design, the synthesizer could potentially remove large parts of the module.

As hinted at in the previous section it would be better to be able to analyze a large design directly to find the resource usage of individual components. This is exactly what the resource analyzer script does as shown in Figure 3. The script itself is very simple and the most complex part is actually printing the design usage in a hierarchical and cumulative fashion. This kind of XDL parsing, although easy, can still lead to useful results. A regression test incorporating this script could for example warn about a submodule which has

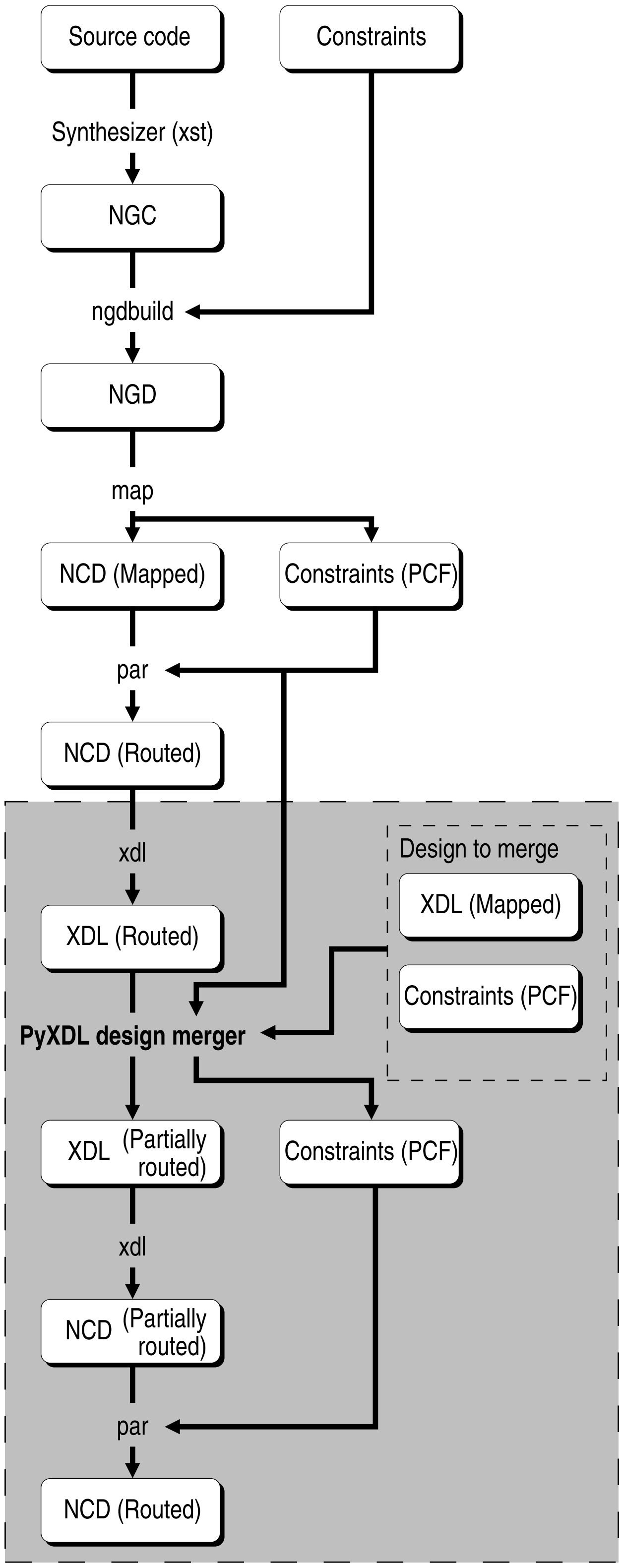


Fig. 2. The typical Xilinx flow augmented with the PyXDL tool to merge a design such as a logic analyzer into a placed and routed design. The new part of the flow is shown in gray.

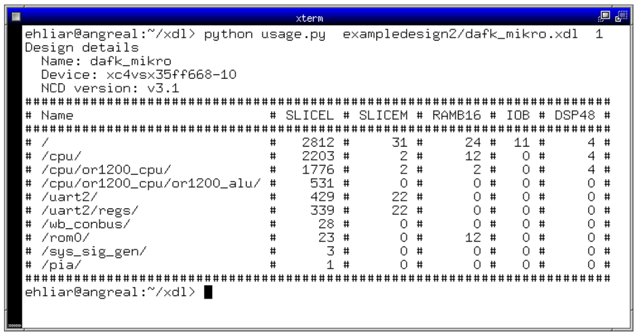

```
ehliar@angreal:~/xdl> python usage.py  exampledesign2/dafk_mikro.xdl  1
Design details
  Name: dafk_mikro
  Device: xc4vsx35ff668-10
  NCD version: v3.1
########################################################################
# Name                        # SLICEL # SLICEM # RAMB16 # IOB # DSP48 #
########################################################################
# /                           #   2812 #     31 #     24 #  11 #     4 #
# /cpu/                       #   2203 #      2 #     12 #   0 #     4 #
# /cpu/or1200_cpu/            #   1776 #      2 #      2 #   0 #     4 #
# /cpu/or1200_cpu/or1200_alu/ #    531 #      0 #      0 #   0 #     0 #
# /uart2/                     #    429 #     22 #      0 #   0 #     0 #
# /uart2/regs/                #    339 #     22 #      0 #   0 #     0 #
# /wb_conbus/                 #     28 #      0 #      0 #   0 #     0 #
# /rom0/                      #     23 #      0 #     12 #   0 #     0 #
# /sys_sig_gen/               #      3 #      0 #      0 #   0 #     0 #
# /pia/                       #      1 #      0 #      0 #   0 #     0 #
########################################################################
ehliar@angreal:~/xdl> █
```

Fig. 3. Using the resource analyzer script to view the resource utilization of various parts of a design.

grown (or shrinked) by a large factor when compared to the previous run.

4.3. Design viewer

The design viewer is capable of viewing a design and showing the configuration of the slices. It is similar in functionality to the floorplanner. In Figure 4 a part of an OpenRisc based design is analyzed by the design viewer.

4.4. Logic analyzer

Putting a logic analyzers into a chip is not a new idea. Both Xilinx and Altera already offers such products (ChipScope and SignalTap). There are also some logic analyzers written by hobbyists available on the net such as Fpgadbg [5].

The main idea behind this section is to show that it is easy for any user to duplicate the main selling point of ChipScope, i.e. the capability to insert a core into an already synthesized and routed design. While it would be easy to create a logic analyzer core which fully mimics ChipScope by connecting to the internal boundary scan primitive we did not intend this tool to be a ChipScope clone. Instead, the intention was that this tool should be useful in systems that might not easily be connected to a PC with a ChipScope client such as remote systems. Therefore the logic analyzer core is operated via a simple serial port interface.

An example of the output of the logic analyzer is shown in Figure 6 and an example of a simple GUI which allows the core to be easily inserted into a design is shown in Figure 7.

4.4.1. Implementation details

The design of the the logic analyzer is shown in Figure 5. It consists of a simple 8 bit microcontroller which is responsible for presenting a text based user interface to a serial port. The MCU is connected to a logic analyzer core via a Wishbone bus. This bus also creates an easy way to extend the

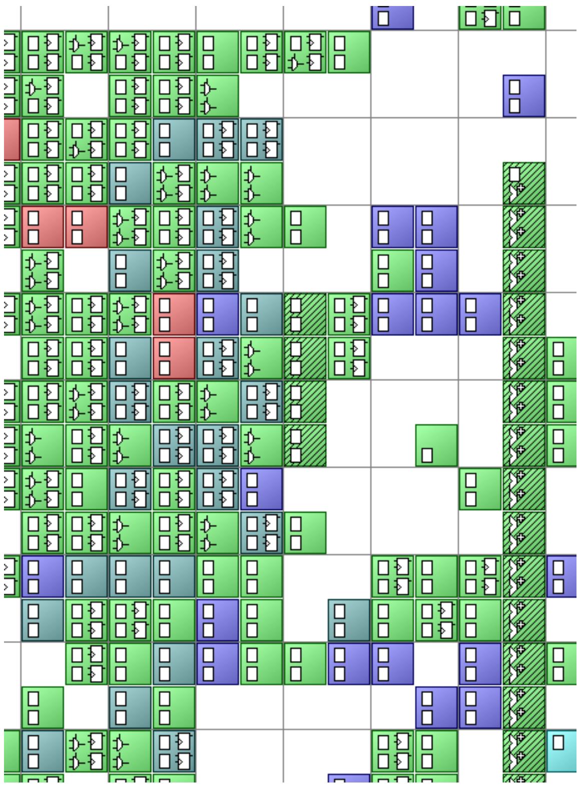


Fig. 4. An example of the output from the design viewer when run on a OpenRisc 1200 based design.

functionality of this core with additional modules. The logic analyzer is currently hardcoded for a maximum of 64 signals which is stored to a 2 kilo-word large buffer.

The Python GUI allows the user to load an XDL design and select which nets to monitor. After the user is satisfied with the selection the program will load the synthesized version of the logic analyzer and remove any elements which will make it hard to merge the logic analyzer into the design (e.g. IOBs and BUFGs). The appropriate flip-flops in the logic analyzer is added as an extra destination of the selected nets. The program memory of the MCU is also modified so that net information such as name and width is available to it. Finally, a user selected clock net is connected to all flip-flops in the logic analyzer core.

The curious reader is also referred to Appendix A which contains an example of how PyXDL can be used to merge a small design into a large design.

4.5. Availability of PyXDL

The PyXDL library will be published under the GPL at `http://www.da.isy.liu.se/~ehliar/pyxdl/` together with the sample applications described in the previous sections. The RTL code of the logic analyzer core will also be made available under the MIT license so that users can use and distribute merged designs without worrying about the stricter terms of the GPL license.

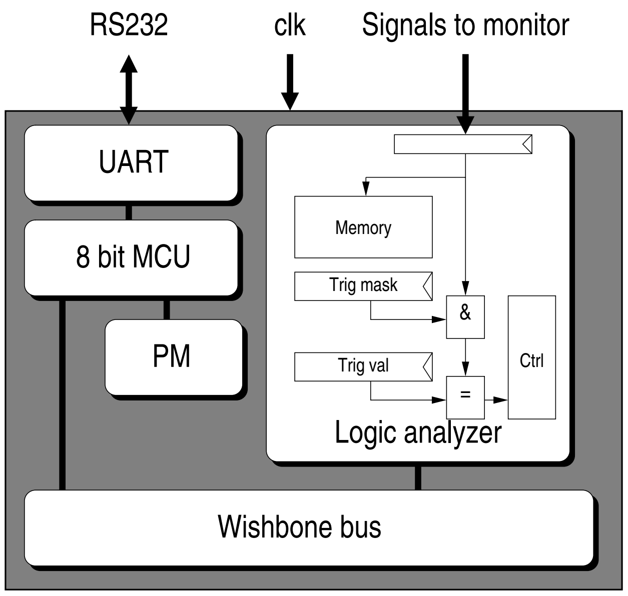


Fig. 5. An overview of the logic analyzer module.

5. DISCUSSION

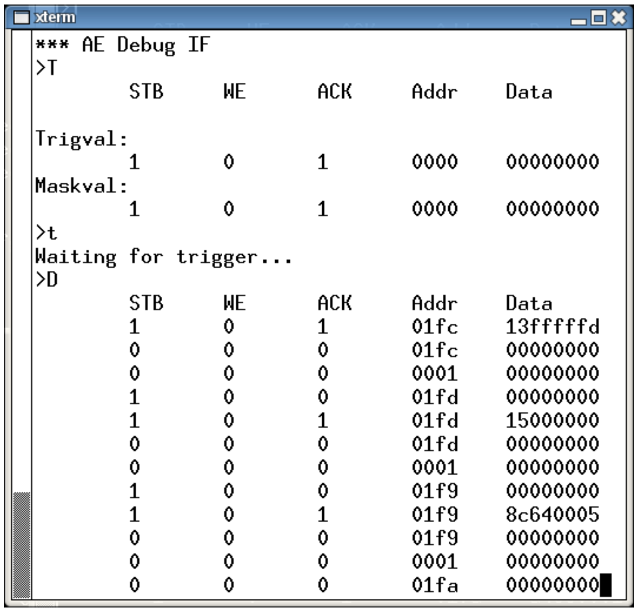
The applications presented in this paper shows only a few of the many possibilities that could be tapped by a creative designer. The applications described earlier could of course be improved by improving them. The design viewer could be improved to show more points of interest to a designer such as clock domain crossings, pipeline depths, and perhaps even show some sort of design complexity metrics for different parts of the design (a long pipeline without feedback is far less complicated and probably easier to test and verify than a state machine with many feedback paths).

The logic analyzer could be improved by adding additional modules to it such as counter modules for statistic gathering. Another interesting addition would be to replace the RS232 interface with another interface such as for example Ethernet or USB.

5.1. Other possible applications

There are many other interesting applications which would be possible to develop. One example would be for those interested in very large FPGA designs that must be mapped onto several FPGAs. A tool could be created that automatically partitioned the XDL file into more than one FPGA.

A similar tool could be made that partitioned a design for a large FPGA into different region of such an FPGA. The advantage of such a design would be that the time consuming



```
*** AE Debug IF
>T
        STB     WE      ACK     Addr    Data

Trigval:
        1       0       1       0000    00000000
Maskval:
        1       0       1       0000    00000000
>t
Waiting for trigger...
>D
        STB     WE      ACK     Addr    Data
        1       0       1       01fc    13fffffd
        0       0       0       01fc    00000000
        0       0       0       0001    00000000
        1       0       0       01fd    00000000
        1       0       1       01fd    15000000
        0       0       0       01fd    00000000
        0       0       0       0001    00000000
        1       0       0       01f9    00000000
        1       0       1       01f9    8c640005
        0       0       0       01f9    00000000
        0       0       0       0001    00000000
        0       0       0       01fa    00000000
```

Fig. 6. The logic analyzer user interface showing instruction fetches on a Wishbone bus. The analyzer has been set to trigger when STB and ACK are both asserted.

placement and routing of the partitioned design could easily be parallelized on a cluster of computers.

5.2. Remaining issues

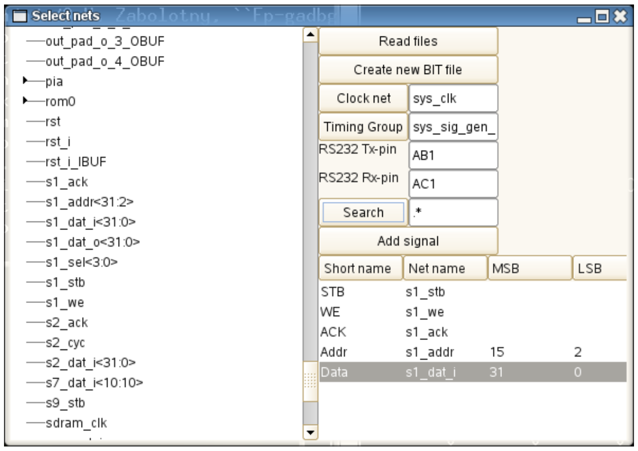
There are unfortunately some issues that are hard to solve in a satisfactory fashion. The main problem is that there is very little information available about routing. Whereas placement is relatively straightforward, reliably routing a design requires detailed timing information about the internals of the FPGA, something which Xilinx hasn't released for modern FPGAs and most likely will not release for the foreseeable future.

Another problem that any tool of this kind will face is that the synthesized design isn't exactly the same as the RTL source code. The various optimizations employed by the synthesizer will remove and rename many nets, making it harder to find the correct signal/bus to inspect. This could be mitigated if more back-annotation information was available to the tools.

Finally, the PyXDL library has only been tested on Virtex-4 based designs.

6. CONCLUSION

We have shown that it is easy to create powerful backend tools for a Xilinx based design flow such as a logic analyzer inserter. By manipulating the design file directly a time consuming full synthesis/placement/routing iteration is avoided



```
──out_pad_o_3_OBUF
──out_pad_o_4_OBUF
▸──pia
▸──rom0
──rst
──rst_i
──rst_i_IBUF
──s1_ack
──s1_addr<31:2>
──s1_dat_i<31:0>
──s1_dat_o<31:0>
──s1_sel<3:0>
──s1_stb
──s1_we
──s2_ack
──s2_cyc
──s2_dat_i<31:0>
──s7_dat_i<10:10>
──s9_stb
──sdram_clk
```

Clock net: sys_clk
Timing Group: sys_sig_gen_
RS232 Tx-pin: AB1
RS232 Rx-pin: AC1
Search: .*

Short name	Net name	MSB	LSB
STB	s1_stb		
WE	s1_we		
ACK	s1_ack		
Addr	s1_addr	15	2
Data	s1_dat_i	31	0

Fig. 7. The GUI used to insert the logic analyzer core into a design.

and therefore increasing productivity. It is our intention that this paper will inspire other designers to explore these possibilities as well.

7. REFERENCES

[1] Xilinx, "Chipscope pro," *http://www.xilinx.com/ise/optional_prod/cspro.htm*.

[2] ——, "Jbits sdk," *http://www.xilinx.com/products/jbits/*.

[3] A. Megacz, "A library and platform for fpga bitstream manipulation," *Proceedings of IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'07)*, 2007.

[4] Xilinx, "Xilinx design language," *help/data/xdl/xdl.html in ISE 6.3*, 2000.

[5] W. Zabolotny, "Fpgadbg - a tool for fpga debugging," *http://www.ise.pw.edu.pl/ wzab/fpgadbg/*, 2006.

Appendix A. PYXDL EXAMPLE

This appendix contains an example of how to use PyXDL to merge a synthesized design into a larger design. The example consists of a design which will monitor a signal and assert an external signal forever if an internal signal has ever been asserted (e.g. an error signal of some sort). In order to shorten the example, the constraints file is not updated with the timing group from the small design. Some values are also hardcoded instead of dynamically getting the values from the XDL files such as the name of the clock networks.

PyXDL source code to merge a synthesized design (test.xdl) into a large design (system.xdl):

```
from xdl import xdl,xdlnet
from pcf import pcf
from xdlutil import par_with_guide

largedes = xdl("system.xdl")
largedespcf = pcf("system.pcf")
# Clock network for the large design
clocknet = largedes.netsbyname["clk_i_BUFGP"]

tinydes = xdl("test.xdl")
# Unplace stuff we don't need
tinydes.unplace_design()
tinydes.remove_unused_dcminsts()
tinydes.remove_inst("clk")
tinydes.remove_net("clk")

# Create a unique prefix for the other design so
# that we don't have to worry about name clashes
tinydes.add_prefix("TEST/")

# Convert flip flop in the IOB to an internal signal
myiob = tinydes.insts["TEST/testin"]
testinpin = tinydes.convert_input_to_internal(myiob)

oldclknet = tinydes.netsbyname["TEST/clk_BUFGP"]

# Remove old clock network
tinydes.remove_net("TEST/clk_BUFGP")
tinydes.remove_inst("TEST/clk_BUFGP/BUFG")

# Merge designs
largedes.mergedesign(tinydes)
# Merge old clock network into new design
for pin in oldclknet.inpins:
    largedes.add_inpin_to_net(clocknet,pin[0],pin[1])

# Select signal to monitor
thenet = largedesign.netsbyname["traceit/state_r_FFd1"]
largedes.add_inpin_to_net(thenet,testinpin[0],
                       testinpin[1])

# Add the IOB to the PCF constraint file and
# select where to place it (at pin AC6)
largedespcf.addiob("TEST/testout","AC6")

# Place and route the design
par_with_guide(largedes,largedespcf,"new.ncd","tmp")
```

Verilog source code for a simple monitor application. testout will be asserted if testin has ever been asserted:

```
   module test(
     input clk,
     input wire testin,
     input wire rst,
     output reg testout);

     reg tmp,sample;
     wire fbloop;

     always @(posedge clk) begin
         sample <= testin;

         tmp <= fbloop;
         testout <= tmp;
     end

     FD monitorfd(.C(clk),.D(fbloop | sample),
                  .Q(fbloop));
endmodule // test
```

TABLE OF SOME INDUSTRIAL PROCEEDINGS CONTENTS

1) Leveraging spreadsheets for integrating FPGA Integration in a Board design flow,

Abha Jain, S.Dharamarajan, Vikrant Khanna, Vikas Kohli, Cadence Design Systems, India.

This paper describes the limitations of integrating an FPGA in schematics driven board design flow and how some of these can be effectively addressed in a spreadsheet based flow. Allegro System Architect© gives a spreadsheet based view of the design. The advantages and ease of integrating an FPGA in a spreadsheet driven board flow are presented. A new methodology for managing large pin-count FPGAs on the board and efficiently handling FPGA driven ECOs that come late in the design cycle, is also discussed.

2) A Digital Data Processor for SYNTHETIC Aperture Radar

Wouter Vlothuizen, Department Transceivers & Real Time Signal Processing, TNO Defence, Security and Safety, The Hague, Netherlands, email: wouter.vlothuizen@tno.nl
Henk Medenblik, Department Transceivers & Real Time Signal Processing, TNO Defence, Security and Safety, The Hague, Netherlands, email: henk.medenblik@tno.nl

This paper presents a Digital Data Processor (DDP) for Synthetic Aperture Radar (SAR). The DDP captures SAR data at a 1 GHz sample rate and processes data at 350 MB/s. Data reduction is performed by a digital down converter, programmable decimating filter and a fully programmable presummer. The total processing power amounts to 12.6 GOPS/s.
Configuration of the DDP on a pulse to pulse basis is achieved by means of a high speed LVDS serial data link capable of transferring up to 500 k messages per second with deterministic timing. The DDP has been implemented on a commercial FPGA digitizer board.

3) AT91 CAP products Configurable Advanced Processors, Ulf Samuelsson, Atmel

Leveraging spreadsheets for integrating FPGA Integration in a Board design flow

Abha Jain, S.Dharamarajan, Vikrant Khanna, Vikas Kohli
Cadence Design Systems (India)
abhaj@cadence.com, rajan@cadence.com, vkhanna@cadence.com, vikas@cadence.com

Abstract

This paper describes the limitations of integrating an FPGA in schematics driven board design flow and how some of these can be effectively addressed in a spreadsheet based flow. Allegro System Architect© gives a spreadsheet based view of the design. The advantages and ease of integrating an FPGA in a spreadsheet driven board flow are presented. A new methodology for managing large pin-count FPGAs on the board and efficiently handling FPGA driven ECOs that come late in the design cycle, is also discussed.

1. Introduction

FPGAs are no longer considered to be just a means for fast prototyping. With the advent of high performance, high density FPGAs, they are increasingly being used in production boards, replacing even ASICs in some cases [1, 2]. These high pin-count FPGAs are advantageous since they are very adaptive to design changes during and after the product development. PCB technologies, like the support for high data rates, high density interconnects with microvias and embedded components also make it easy to use large FPGAs in the board. Aggressive time-to-market dictates that the FPGA and board designs proceed concurrently. This poses the challenge of importing and maintaining such high pin-count FPGAs in the board design and need a tight PCB-FPGA design flow integration. This integration requires effective and efficient transfer of a large amount of data from one design flow to the other [3].

FPGA driven ECO can happen to achieve timing closure and it triggers a board synchronization and verification cycle that is very time consuming [4]. Similarly, there can be a PCB driven ECO because of signal integrity constraints or routing optimizations in the board layout that will require the FPGA designers to again achieve timing closure with the new pin assignments. A broad picture of the PCB and FPGA design flow and their interaction at various stages is shown in Figure 1.

The following section will present the challenges of concurrently designing FPGAs and its board. Section 3 explains the inherent limitations of the FPGA import and PCB-FPGA integration in schematic driven flow. Section 4 discusses the FPGA import and integration with board design in a spreadsheet based environment. Future needs for better PCB-FPGA design integration discussed in Section 5.

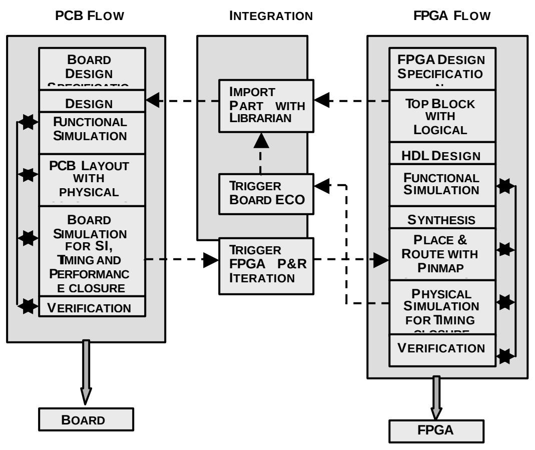


FIGURE 1. Flow of PCB-FPGA integration.

2. Concurrent PCB-FPGA Design Challenge

Concurrent PCB and FPGA design demands certain amount of transparency, large amount of data transfer and proper synchronization of this data between the two design flows.

FPGA design flow needs to take in pin-assignment recommendations and constraints from the board layout. The layout should understand the FPGA design rules, pin directions, pin-bank details, differential pins and simulation models for pins, to facilitate effective pin assignment recommendations with proper signal integrity analysis and routing optimizations. A board may use more than one such large pin-count FPGAs, thereby, increasing the amount of data synchronization involved.

There is also the challenge to achieve a high degree of automation in the ECO process to make it error-free so that precious time is not wasted in synchronizing the board and the FPGA design. Any solution for the PCB-

FPGA integration must have an effective FPGA import in the board flow and fairly automated ECOs to reduce the time-to-market.

3. Limitations of the schematic based FPGA import and PCB-FPGA integration

The problems in a schematic based FPGA import and PCB-FPGA integration [4] are discussed here.

3.1 FPGA as a hierarchical block in schematic

Preliminary pin-assignment of the FPGA is the first step in concurrent PCB and FPGA design process and this step means generating the part along with the symbol of the FPGA and instantiating the symbols in the board design. This step has some problems in schematic flow.

Hierarchical encapsulation: Most PCB designers encapsulate the FPGA symbols within a hierarchical block exposing the pins of the FPGA as ports of the block. This insulates the top design from frequent FPGA symbol changes, but hierarchical symbol does not show the logical to physical pin mapping and this is a serious limitation during debugging. It forces the use of hierarchical design that increases the flow complexity.

Block size and signal short: The hierarchical block symbol size is an issue for a large pin-count FPGA. Most schematic driven board flow does not support splitting hierarchical block symbol. Power pins can be removed from the hierarchical block symbol and declared as globals, to reduce the size but this causes an unintentional short of these power pins in case there are more than one FPGA s in the same design.

3.2 Split Symbols for FPGA in schematic

Front-end engineers can tackle the size of the FPGA symbol inside the hierarchical block by splitting it into multiple small symbols. Splitting the symbol has some limitations.

Connecting through symbols: Split symbols are created by Librarian during part generation. The split symbols need to be individually placed and connected in the schematic. Since the pin count is large and pins are split over many symbols, this step is extremely tedious and error-prone.

Updating connections in schematic: ECOs, whether Board or FPGA driven, place tall demands on the schematics flow. The designers need a design preview showing the current state of the FPGA and the board. They need some comprehensive way to figure out the differences and trigger automatic updations of split symbols and their connections in the board schematic. Providing all this in a schematic based flow is very difficult and hence, ECOs are still largely manual.

4. Spreadsheet based FPGA import methodology

There are a number of advantages offered in a spreadsheet based view of the design. This section talks about the FPGA import methodology as implemented in Allegro System Architect© and explain ways to exploit the spreadsheet based view for easy and highly automated ECOs.

4.1 Introduction to Spreadsheet view

Spreadsheet view presents the design and it's connectivity in the form a table. A table view helps focus on design entry and the Front-end engineer need not worry about schematics at this stage. A snapshot of Cadence Allegro System Architect© presenting the table view of the design is shown in Figure 2. It shows the list of component instances and nets in the design and their connectivity is shown in the connectivity details pane.

The table is fully customizable with the user deciding the visibility of columns, thereby making optimum use of real-estate on the pane to avoid cluttering. There are pattern matching filters for columns with sorting capability. These features go a long way in helping the designer in ma king connections without being overwhelmed by the number of pins. Auto-connection options are also available where the user has the option of creating signals from the pin names (same as pin name or with some prefix/suffix) and making corresponding connections. There are a number of other features provided in the tool that make design entry very easy and automatic as compared to schematics view. This is extremely useful when we have large pin-count devices, especially large FPGAs.

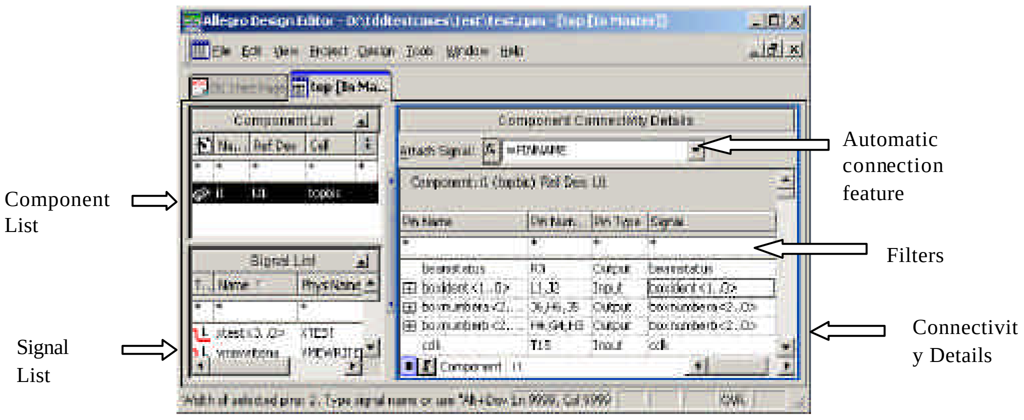


FIGURE 2: **Spreadsheet View in Allegro System Architect.**

4.2 Importing FPGA through wizard

Allegro System Architect© can directly import the FPGA into the board design using an Import FPGA wizard. The wizard supports FPGA families from three major FPGA vendors: Xilinx, Altera (Quartus and MaxplusII) and Actel. It guides the user through a series of steps that take as input the FPGA files supplied by the FPGA design team and automatically generate a part from those files. Some steps in the import process are shown in Figure 3. Once the wizard finishes its job, the user has the option of automatically adding any number of instances of that imported FPGA part in the board design. The use of a wizard provides seamless and direct FPGA import in the board design and makes it very easy as compared to the existing schematics based import methodology.

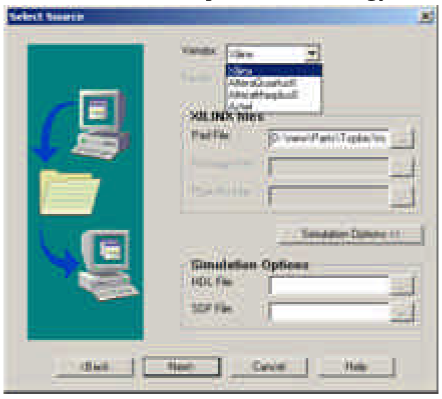


Figure 3(a): FPGA Families and Import file

The user can create two types of parts during FPGA import as shown in Figure 3(b): *Custom* and *Standard*. In custom component, the pin names are generated from the pins used in the logic mapped to the FPGA. In standard component, the pin names are made from the FPGA device pins. For standard component, the wizard generates a logical-pin-name to physical-pin-number mapping in the form of a file. This file is used to overlay the logical pin names over the physical pin names in the spreadsheet view. This greatly simplifies the process of making connections to the pins because the engineer directly sees the logical pin to physical pin number mapping.

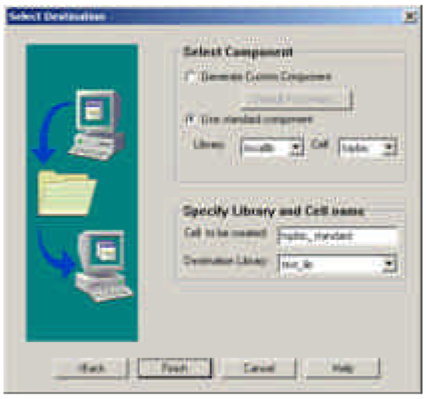


Figure 3(b): Standard and Custom Part

Using this import methodology helped in overcoming the limitations of schematics based FPGA import. There is no need to go outside the tool to create FPGA parts, the logical-pin-name to physical-pin-number mapping is available to the user, there is no need for hierarchy or split symbols, unintentional short of power pins is done away with and use of filters in making connection makes the design entry very fast.

4.3 Automatic Schematic Generation and PCB-FPGA integration

Allegro System Architect© has a schematic generator that takes the component connectivity from the design files and symbols from the part libraries to automatically generate a schematic view of the design. While the board designer imports the FPGA in the design and make connections, the Librarian can generate split parts for the FPGA in parallel. Then the designer can automatically generate the schematics and the utility will automatically place and connect those split parts in the schematic. The schematic can also be updated automatically in preserve mode if there are updations in FPGA symbols or their connectivity because of ECOs. This degree of automation makes the schematic generation and updation step very fast.

4.4 Update ECO through Import Wizard

The Board and the FPGA can go out of sync in two scenarios: the board layout changed the FPGA pin-assignment by swapping pins from the same pin-bank or the FPGA team came up with a new pin assignment or a pin model change.

Whenever the FPGA files undergo a change, the board designer will need to update the FPGA part. In Allegro System Architect©, we can handle FPGA triggered Board ECOs. The import FPGA wizard has an *Update ECO* option for updating the FPGA part. The user is presented with a preview of the part differences in a tabular form with an option of creating a new part. Once this new part is created, the front-end engineer has the option to update the old FPGA part and preserve the connectivity in a Replace Compoent wizard shown in Figure 4. This ECO will be much easier in a spreadsheet view than in a schematic view because symbols are not involved in updation. The schematic view can be automatically updated by running the schematic generation utility.

5. Future Direction

The use of spreadsheet view and its new methodology for importing FPGA addresses some of the PCB-FPGA integration challenges; we have to address some requirements for complete PCB-FPGA design flow integration.

Support for cache: We have to automatically recognize the need for a Board ECO triggered by FPGA changes. This can be done through caching. Automatic polling at load time will tell the tool whether the imported FPGA

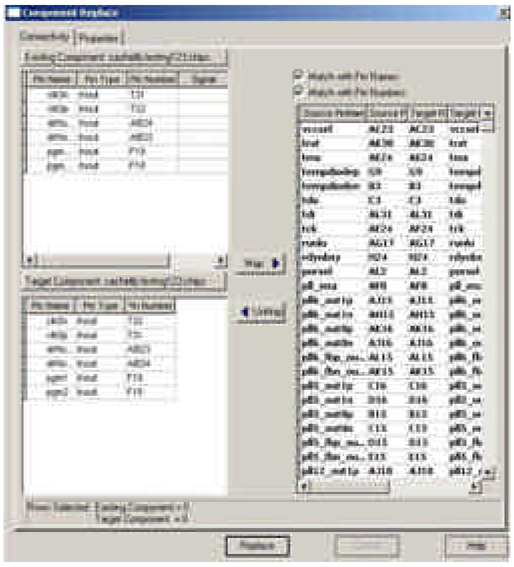


FIGURE 4: Replace Dialog for FPGA ECO

is stale or not by comparing the cached information about the FPGA [1].

Layout Triggered FPGA ECO: In case FPGA pin assignments are modified in the board layout, the user should have the option of generating a vendor specific pin map constraints file which can be taken in by the FPGA vendor's P&R tool to update the FPGA [1,2,3].

Recognizing Pin Banks and LVDS in board design flow: Current FPGA architectures group pins with common characteristics into banks that share same IO standard model. They also have support for LVDS (Low Voltage Differential Signal) pins. The current PCB-FPGA integration at the front-end must have support for pin banks and LVDS pin-pair data so that this can be annotated to the board layout, where this can be understood and handled [1, 2, 3].

6. References

[1] Philippe, G., *Methodologies for Efficient FPGA Integration into PCBs*, White Paper on Xilinx FPGAs, March 2003. http://www.xilinx.com

[2] Irwin, R., *FPGA-PCB co-design means more than just data transfer*, FPGA and Programmable Logic Journal, Altium. http://www.fpgajournal.com/articles/20040824_altium.htm

[3] Morris, K., *Board with FPGAs*, FPGA and Programmable Logic Journal, Altium, http://www.fpgajournal.com/articles/boardwithfpgas.htm

[4] Dharmarajan, S., *Integrate FPGA & System Design using ConceptHDL*, Xcell Journal, Issue 32, Second Quarter 1999 http://www.xilinx.com/xcell/xl32/xl32_19.pd

A DIGITAL DATA PROCESSOR FOR SYNTHETIC APERTURE RADAR

Wouter Vlothuizen

Department Transceivers & Real Time Signal Processing
TNO Defence, Security and Safety
The Hague, Netherlands
email: wouter.vlothuizen@tno.nl

Henk Medenblik

Department Transceivers & Real Time Signal Processing
TNO Defence, Security and Safety
The Hague, Netherlands
email: henk.medenblik@tno.nl

ABSTRACT

This paper presents a Digital Data Processor (DDP) for Synthetic Aperture Radar (SAR). The DDP captures SAR data at a 1 GHz sample rate and processes data at 350 MB/s. Data reduction is performed by a digital down converter, programmable decimating filter and a fully programmable presummer. The total processing power amounts to 12.6 GOPS/s.

Configuration of the DDP on a pulse to pulse basis is achieved by means of a high speed LVDS serial data link capable of transferring up to 500 k messages per second with deterministic timing. The DDP has been implemented on a commercial FPGA digitizer board.

1. INTRODUCTION

Synthetic aperture radar (SAR) provides a capability for all weather, day and night ground observation of static objects. Moving target indication (MTI) adds the capability to detect moving objects. Within the MiniSAR project TNO Defence Security and Safety, located in the Netherlands, is developing a combined SAR/MTI radar system operating on X-band (3 cm wavelength).
This scalable and modular radar system has an active electronically steered antenna array and makes use of commercial off-the-shelf components where relevant. It is designed for use within small airborne platforms, in particular tactical UAV and small civil airplanes. The latter requirement results in a compact (50x50x30 cm) and light weight (50 kg) design which consumes limited power (max. 500 W).

Figure 1 MiniSAR

The MiniSAR radar operates by emitting pulses to the ground and receiving the resulting reflections. These reflections are amplified, down converted and filtered by analogue components, and then fed to the DDP where they are digitized and signal processing is performed.
The next section describes the architecture of the DDP with its different signal processing blocks. The implementation of these blocks inside an Altera Stratix FPGA is described in the following section and finally conclusions are summarized.

2. DDP ARCHITECTURE

An architectural overview of the Digital Data Processor is shown in ***Figure 2***. The DDP data path can be decomposed into a pulse acquisition block, a digital down converter, programmable range filters, programmable presummer and finally a DMA engine block. Components shown in gray are implemented inside the FPGA.

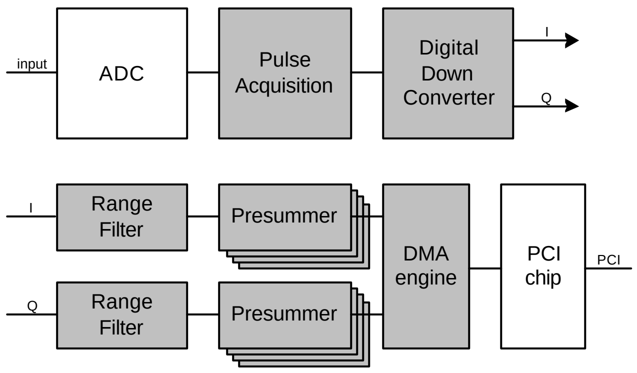


Figure 2 DDP architecture

2.1. Pulse Acquistion

The input of the DDP is an analogue signal with a bandwidth of almost 500 MHz centred round an IF frequency of 250 MHz. This signal is sampled with 8-bit resolution at 1 GHz.

Although this results in a peak data flow of 1GByte/s, the actual required data processing rate is lower due to the pulsed behaviour of the radar system. The maximum acquisition duty cycle is 35%, resulting in an average data flow of 350Mbyte/s.

2.2. Digital Down Converter

After data acquisition the signal needs to be down converted to in-phase (I) and quadrature-phase (Q) baseband signals. In fact the signal which is centred around 250 MHz is shifted to DC. This function is performed by the digital down converter block which performs mixing with a 250 MHz carrier component.

2.3. Programmable Range Filters

The I- and Q-channels from the down converter need additional filtering. These filters improve the selectivity by removing receiver noise outside the transmitted bandwidth. They also remove aliasing products introduced by the Digital Down Converter. For low bandwidth modes the filters can perform decimation, resulting in a lower data rate. Programmable decimating FIR filters are used for this purpose.

2.4. Programmable presummers

Complete radar lines coming out of the range filters are in fact samples of what is called an azimuth spectrum. For a typical radar mode, the sample frequency (i.e. the pulse repetition frequency) might be 5 kHz whereas the band-width of interest is 85 Hz. The presummers are decimating FIR filters which reduce the data rate and improve signal to noise ratio.
The filter architecture differs from a common decimating FIR because these presummer filters process complete radar lines instead of separate radar pulse samples. The filter output vector O is the sum of input vectors I multiplied by coefficients C:

$$\vec{O} = \sum^{N} \vec{I}_n * C_n$$

When an input vector I_n arrives, we immediately multiply it by the proper coefficient C_n and add the result to the previous sum, which is initialized to zero at the start:

$$\vec{O}_{n+1} = \vec{O}_n + \vec{I}_n * C_n$$

Therefore, only a single line of storage is required and the end result is immediately available after the last line has been processed. The formula above can in fact be implemented on a per input sample basis, which nicely matches the preceding stage.
A large reduction in data is performed at this stage because only the result of multiple accumulated radar lines is forwarded to the DMA engine, which we don't describe here.

2.5. Control

Many settings throughout the MiniSAR radar can change on a per pulse basis, i.e. at a rate of 20 kHz. Because the total message rate can exceed 120 k/s and deterministic timing is required, a dedicated communication bus was developed.
This radar bus is based on point to point links using Altera high speed LVDS serializers running at 500 Mbit/s. The bus has a ring topology, where every node either repeats its incoming data, or inserts its response in the data stream. We use a master/slave protocol with a single master issuing messages. The maximum message rate is in excess of 500 k/s, which leaves ample room for growth.

3. IMPLEMENTATION

The DDP has been implemented on a commercially available high speed digitizer card. This 3U CompactPCI card contains a high speed AD converter, FPGA, Double Data rate SDRAM memory, cPCI interface chip and the necessary power supplies.
All signal processing functionality is written in VHDL for implementation on an Altera Stratix EP1S25 FPGA. This FPGA contains 25,660 Logic Elements, 80 dedicated 9x9 DSP multipliers, 224 M512 RAM blocks (32 × 18 bits), 138 M4K RAM blocks (128 × 36 bits) and 2 M-RAM blocks (4K × 144 bits).

3.1. Pulse Acquisition

Digitization of the received radar pulse is performed with a 1 Gsps MAX108 AD converter. This ADC has an internal 8:16 demultiplexer which allows for interfacing to the FPGA by means of a 16 bit wide differential LVPECL bus running at 500 MHz.
In order to better match the lower clock speed of the signal processing blocks inside the FPGA, the 16 bit/500 MHz input bus is further demultiplexed inside the FPGA to 64 bit/125 MHz by means of dedicated high speed deserializers.
The 64 bit wide words enter the line storage buffer which is implemented as a FIFO. This FIFO is capable of storing 2048 x 66 bits words. Two additional memory bits are needed to indicate the start and stop of a radar pulse.
The secondary side of the line storage FIFO runs at a 175 MHz clock rate. This is the clock speed where the actual

signal processing is performed. We use a fully synchronous data driven approach of the data path where the FIFO is automatically read if it is not empty and all subsequent units are required to process data sent to them. The large amount of memory needed for this line storage FIFO is implemented with Trimatrix M4K memory blocks.

3.2. Digital Down Converter

Every fourth clock cycle a 66 bit word representing 8 ADC samples is pulled out of the FIFO. These are regrouped to 2 ADC samples for every single clock cycle. These samples are multiplied by the appropriate sine or cosine terms to obtain the I-path and Q-path.
Due to the fact that the carrier frequency equals a quarter of the sample rate Fs, efficient Fs/4 mixing can be performed. With this well known technique the sine and cosine components are reduced to {0,1,0,-1} and {1,0,-1,0} sequences. Multiplication by -1 (and 1) is of course very simple. Additionally, in real hardware the multiplications with the zeros are omitted and a factor two reduction in processing rate is achieved. In our case, the processing rate at the I- and Q-path output is reduced to 175MByte/s per channel.

3.3. Programmable Range Filters

The following stage consists of two decimating FIR filters, one for the I-path and one for the Q-path. Matlab simulations indicate that the required frequency response for these filters can be achieved using 32 taps for each filter.
However, due to the Fs/4 mixing scheme in the digital down converter which discards the 'zero' samples, each of the filters actually uses only half of its taps whilst the other half would always receive zeros as input. The original 32 tap impulse response is thus reduced to the 16 odd taps only for the I-path and the 16 even taps for the Q-path. Therefore, the Fs/4 mixing scheme also saves 50% in required filter resources.
For lower bandwidths longer impulse responses are needed which can easily be supported when a polyphase implementation of the FIR filters is chosen. With a polyphase filter implementation it is possible to exchange hardware resource count against speed and vice versa. The architecture in Figure 3 shows a minimum resource version of a polyphase decimating filter with an impulse response length of twelve taps. The original impulse response h(0) .. h(11) is separated into three phases containing four taps each. The three different tapsets on the multipliers are changed sequentially at the high input rate. The intermediate multiplication results of the input data with the different coefficients are accumulated and after each new filter output cycle the accumulated results are shifted right into the adder chain.

Note that the transposed FIR filter requires an adder chain instead of an adder tree. The Stratix DSP block contains a fast adder, however this DSP block is optimized for adder trees. For this reason the fast adder block in the Stratix DSP cannot be used and therefore the adder chain needs to be implemented with LEs.

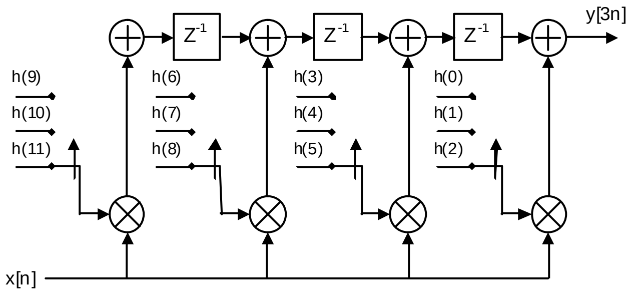


Figure 3 Minimum resource polyphase decimating filter

The two FIR filters need to operate at 175 MHz in order to achieve the required throughput of 350 MByte/sec in the data path. The effective processing rate of the FIR filters is 11.2 GOPS/s.
The combination of high speed and programmable decimation functionality forces the usage of embedded DSP block multipliers and fast M512 Trimatrix memory blocks for a final implementation of these FIR filters.
A total of 32 embedded 18x18 multipliers are needed for implementing both FIR filters. This is equivalent to 64 of the available 80 embedded 9x9 multipliers on a Stratix EP1S25.
Each of the 2 x 16 taps requires a dedicated memory block for coefficient storage. A total of 32 separate M512 memory blocks are used. Each M512 memory block has storage capability for 32 different coefficients which allows a maximum decimation factor of 32 for the polyphase FIR filters.
Finally a programmable scaler is added at the outputs of both filters. This scaler performs output scaling to 18 bit words for both output channels. The coefficient memories, decimation factor and output scale factor are programmed through the high speed radar bus.

3.4. Presummers

The presummer is a FIR filter where each sample is a complete radar line. Every radar line is weighted with a different coefficient and added to the intermediate summed radar line. The basic elements for the presummer consist of a large memory block and a fast multiplier.

Figure 4 shows the architecture of a single presummer. The presummer has a data input, data input valid control line, data output, coefficient input, and three control lines named *first_line* , *last_line* and *start_line*.

The presummer operates on multiple lines and therefore some control mechanism is needed to indicate that a) the first radar line enters the presummer memory or b) that the presumming of multiple lines has ended and the accumulated output result can be forwarded to the final formatter block. This is achieved with the *first_line* and *last_line* control signals.

The *start_line* signal is high during one clock cycle and indicates the start of a new radar line. This signal resets a pixel counter which is normally counting up at the same rate where input data arrives. The counter points to the read location of the presummer memory which is equivalent to the next pixel location within a radar line. The pixel counter pointer is also delayed to achieve the write address on the memory. This delay is equal to the pipeline delay which is introduced by the multiplier and adder. The states of the switches drawn in Figure 4 belong to the situation where their control signals are '0'.

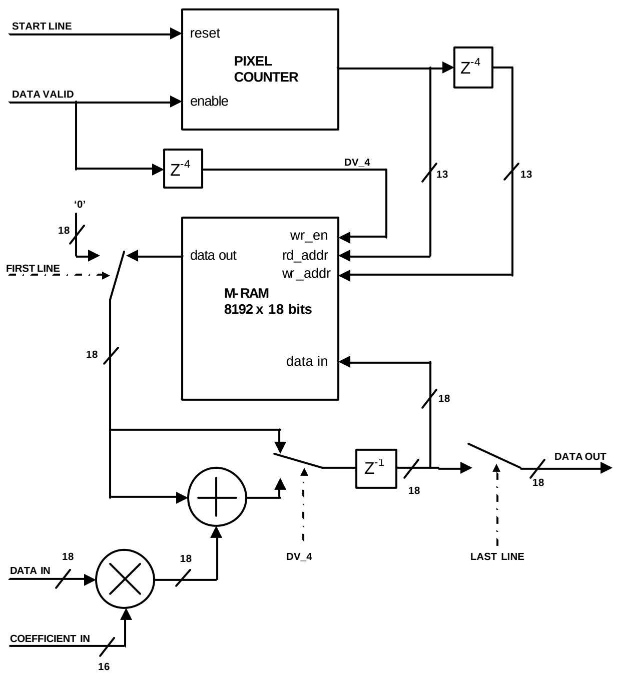


Figure 4 Single Presummer Architecture

During the first radar line which enters the presummer, the *first_line* signal is held active high. Therefore previous stored data from the memory is not fed to the adder. Instead, the output of the adder represents the multi-plication of the input data of the presummer and the coefficient value for that specific radar line. This result is then written into the presummer memory.

During the next line, a new coefficient is applied to the multiplier and new data enters the presummer. The *first_line* signal is held inactive now which means that the adder also accumulates the corresponding radar pixel from the previous weighted line which was stored into memory. This process can continue for several adjacent radar lines.

When *last_line* is active the switch is connected to the output of the adder and the accumulated results of several radar lines are send to the output.

A special situation occurs when both the *first_line* and *last_line* signals are held active; in that case the presummer is effectively bypassed. For the final architecture which contains four presummers this provides a manner to control which of the presummers are active during a radar line.

Both the I-path and the Q-path need 4 presummers which are independently controlled through the radar bus on a pulse to pulse timing scheme. With 8 presummers running at 175 MHz, the total processing rate is 1.4 GMAC/s whilst memory is accessed at a rate of 6.3 GB/s.

A total of 8 memories and 8 dedicated multipliers are needed. Each memory must have enough capability to store a complete radar line which means a maximum size of 8K x 18 bit samples each. The large memory resource for the presummers is implemented in the only two available large M-RAM blocks on the EP1S25, one block for the presummers in the in-phase path and one block for the presummers in the quadrature path. Both M-RAM blocks lie adjacent to each other on the physical die close to the DSP multiplier blocks which gives a logical partitioning between the I-path and the Q-path. The M-RAM blocks are configured as two 8K x 72 bit memories to support 4 presummer memories for each channel.

4. CONCLUSION

Using a moderately sized FPGA we are able to perform high speed signal processing using limited room and power. With the complete data path running at 175 MHz, we achieve a processing power of 12.6 GOPS/s and a memory access rate of 6.3 GB/s.

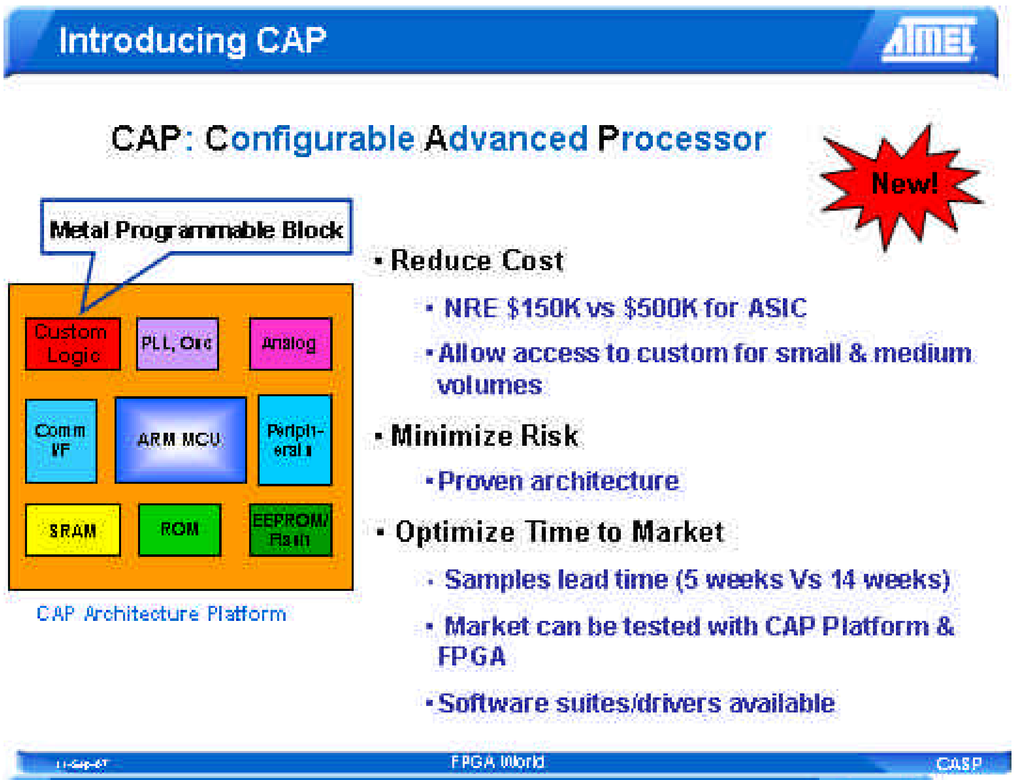
The signal processing functions implemented in hardware allow us to reduce a 350 MB/s input data stream about tenfold, which makes further processing in software manageable.

5. REFERENCES

[1] Fredric J. Harris, "Multirate signal processing," *Prentice Hall*, ISBN 0-13-146511-2, pp. 108–116, Nov. 2004.

[2] Stratix Device Handbook, Volume 1, S5V1-3.2, January 2005.

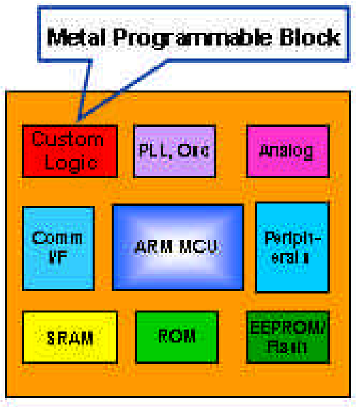
AT91 CAP products Configurable Advanced Processors, Ulf Samuelsson, Atmel



Introducing CAP

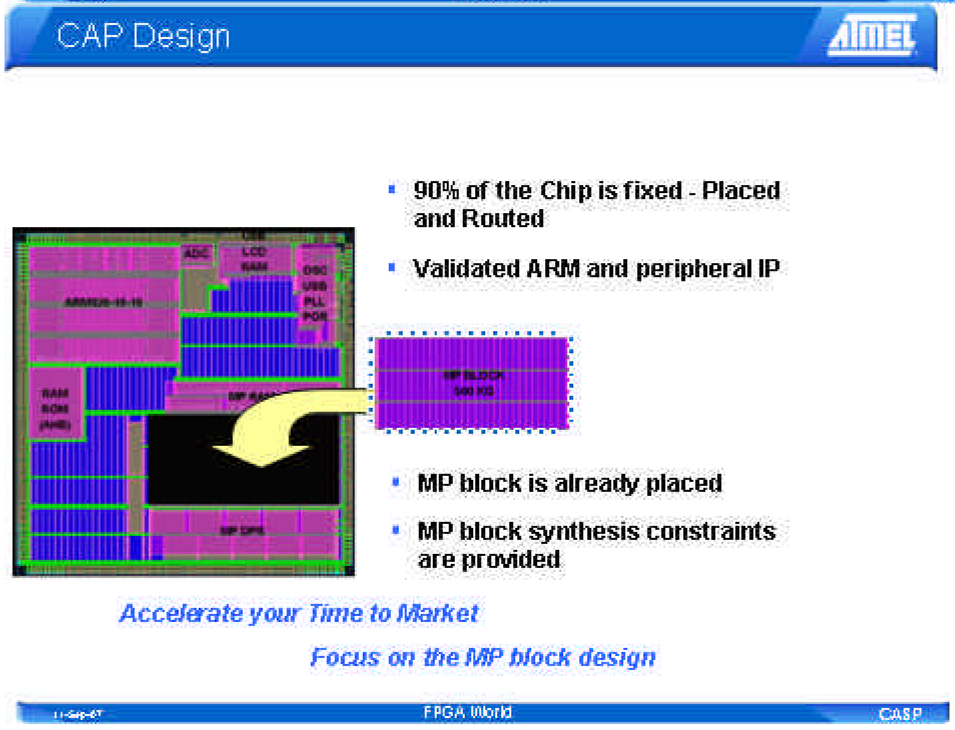
CAP: Configurable Advanced Processor

New!



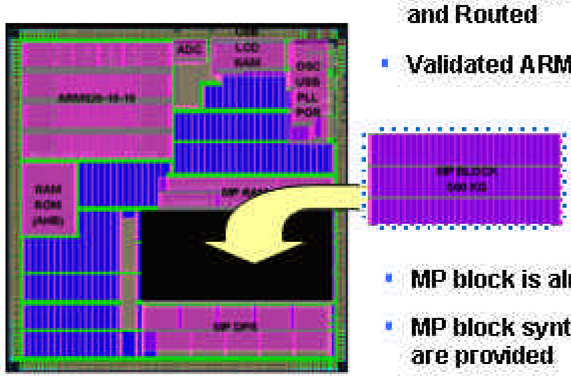
CAP Architecture Platform

- Reduce Cost
  - NRE $150K vs $500K for ASIC
  - Allow access to custom for small & medium volumes
- Minimize Risk
  - Proven architecture
- Optimize Time to Market
  - Samples lead time (5 weeks Vs 14 weeks)
  - Market can be tested with CAP Platform & FPGA
  - Software suites/drivers available



CAP Design

- 90% of the Chip is fixed - Placed and Routed
- Validated ARM and peripheral IP

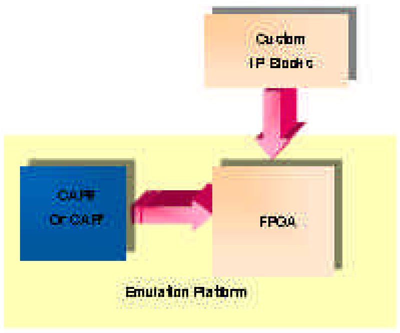


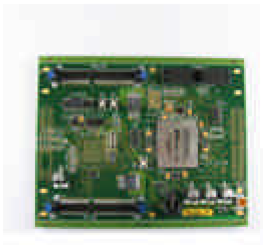
- MP block is already placed
- MP block synthesis constraints are provided

Accelerate your Time to Market

Focus on the MP block design

CAP Emulation Strategy





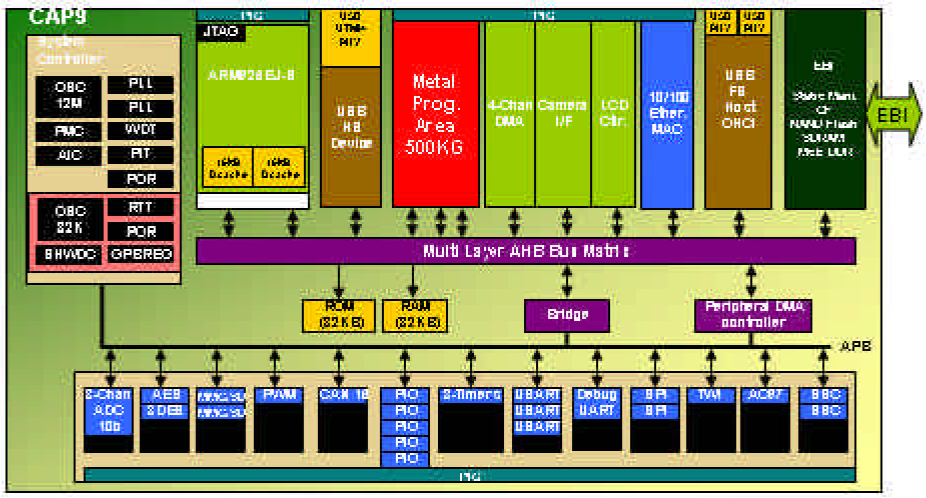
CAP7
or CAP9
Mezzanine
Card

Dedicated Hi-Speed Interface to industry standard FPGAs (Altera, Xilinx)

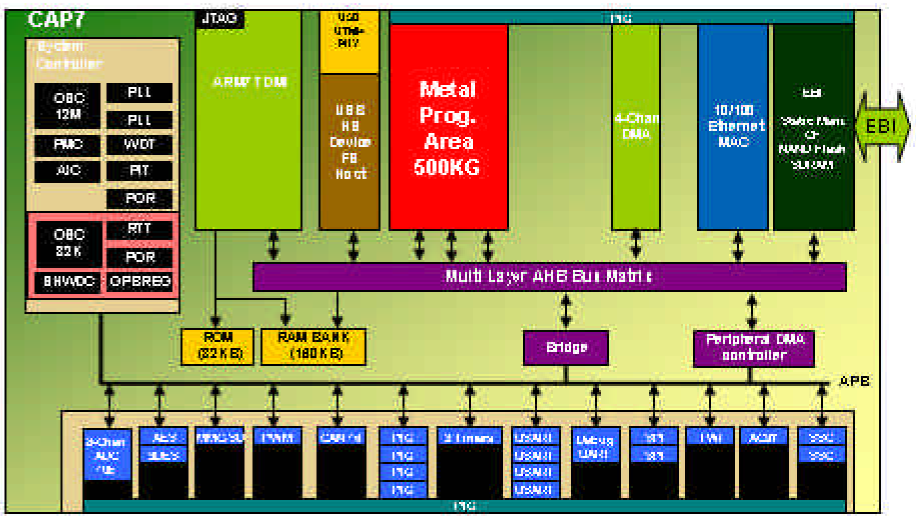
FPGA has bus master access to ARM AMBA bus

Atmel provides FPGA block, implementing interface

CAP9 Architecture



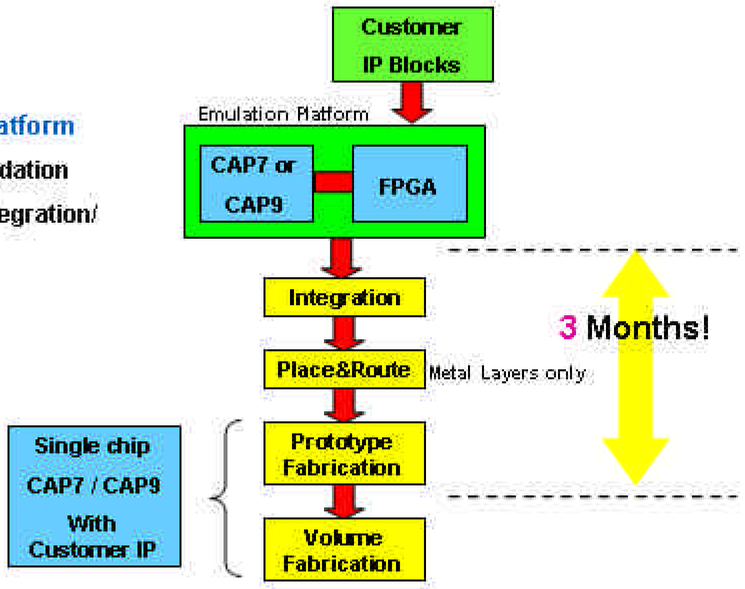
CAP7 Architecture



CAP Design Flow

Emulation Platform

- ➢Software validation
- ➢Hardware integration/ validation



Bridging the Gap

Choosing CAP allows you to:

- Boost your overall system development
- No Risk
- Accelerate your product introduction
- Save 600K$ designing the IC
- Rapidly develop a full roadmap of products around CAP architecture

CAP is the ideal solution:

- Address emerging business with low upfront investment
- Serve any volume business with a totally commercial optimized solution

5th FPGAworld CONFERENCE 2008

Next year´s FPGAworld will be better if you help.

Next year, FPGAworld will probably be in Stockholm (academic and industrial) and Lund (Industrial).

Are you interested in being one of the Presenters?	Please mail lennart.lindh@fpgaworld.com
Are you interested in being one famous Exhibitor?	Please mail david@fpgaworld.com
Can you be one of the impo rtant Sponsors?	Please mail david@fpgaworld.com

Are you interesting to work in the academic program group? Please mail lennart.lindh@fpgaworld.com

Are you interesting to work in the industrial program group? Please mail lennart.lindh@fpgaworld.com

We need publicity chairmen for both industrial and academic!
We need to expand the industrial program group from other countries.
Next year we probably will have two days in Stockholm, so we need one evening sponsor.
………

FPGAworld Thank You!