UNIVERSITY OF
CAMBRIDGE

FPGA programming for the masses with CASPER: A community of Radio Astronomy Instrument designers.

Jack Hickish

Cavendish Laboratory, University of Cambridge
CASPER Group, Berkeley
jack.hickish@mrao.cam.ac.uk

August 13, 2014

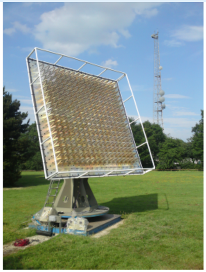
Outline

The Radio Astronomy Mantra: More is more

- More bandwidth
- More field-of-view
- More sensitivity
- More resolution

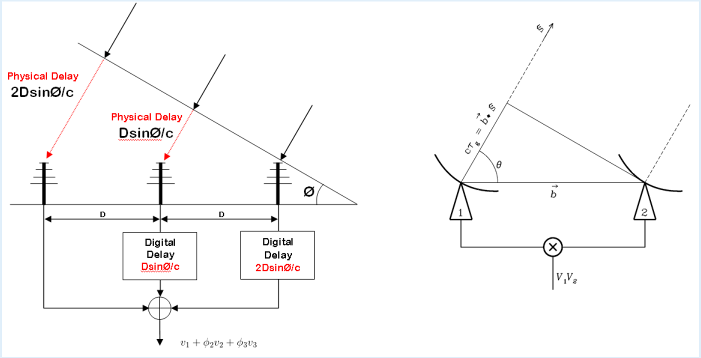
Large Dishes

Arrays of small telescopes


Phased Arrays

EXPENSIVE ELEMENTS → **EXPENSIVE COMPUTATION**

Beamforming & Correlation



Beam power:

$$\sum_{\{i,j\}} \phi_i \phi_j^* v_i v_j^*$$

Building blocks of Radio-Astronomy DSP

- Digitize Antenna signals
- Channelize
- Combine signals from N different antennas
  - Summation – $O(N)$
  - Cross-multiplication – $O(N^2)$
- Average
- Record to disk

Radio-Astronomy DSP Parameters

- Number of antennas ($1 \rightarrow \sim 1000$)
- Bandwidth (100 MHz $\rightarrow$ 10 GHz)
- Frequency channels ($\sim 2^{10} \rightarrow \sim 2^{30+}$)
- Averaging period (None $\rightarrow$ 10 s)

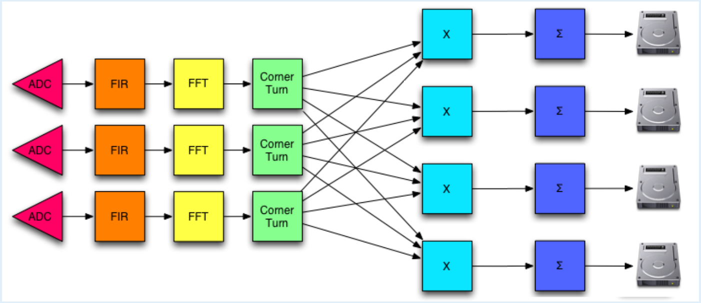
- Many Tb/s interconnect, many complex Tops/s
- Filtering is parallel by antenna
- Correlation/Beamforming is parallel by frequency channel

Radio Astronomy DSP Parameters

A large variety of Radio Astronomy instruments can be build from a small number of parameterised

- Filters (F-Engines)
- Correlation Engines (X-Engines)
- Beamforming Engines (B-Engines)
- Interconnect

The generic Radio Telescope



CASPER

Center for Astronomy Signal Processing and Electronics Research

CASPER

~~Center~~ for Astronomy Signal Processing and Electronics Research Collaboration

CASPER

~~Center~~ for Astronomy Signal Processing and Electronics Research

~~Collaboration~~

Community?

CASPER

"The primary goal of CASPER is to streamline and simplify the design flow of radio astronomy instrumentation by promoting design reuse through the development of platform-independent, open-source hardware and software."

CASPER

- Simplify
  - Leverage industry standards (eg, ethernet for interconnect)
  - Small number of custom [FPGA] platforms
  - Optimize for ease of use (not ops/watt, ops/rack unit)
  - Low knowledge-barrier for users
- Re-use
  - General purpose hardware
  - Modular, upgradable piecemeal
  - Flexible, scalable architectures

CASPER

- Simplify
  - Leverage industry standards (eg, ethernet for interconnect)
  - Small number of custom [FPGA] platforms
  - Optimize for ease of use (not ops/watt, ops/rack unit)
  - Low knowledge-barrier for users
- Re-use
  - General purpose hardware
  - Modular, upgradable piecemeal
  - Flexible, scalable architectures

CASPER

- Simplify
  - Leverage industry standards (eg, ethernet for interconnect)
  - Small number of custom [FPGA] platforms
  - Optimize for ease of use (not ops/watt, ops/rack unit)
  - Low knowledge-barrier for users
- Re-use
  - General purpose hardware
  - Modular, upgradable piecemeal
  - Flexible, scalable architectures

CASPER

- Simplify
  - Leverage industry standards (eg, ethernet for interconnect)
  - Small number of custom [FPGA] platforms
  - Optimize for ease of use (not ops/watt, ops/rack unit)
  - Low knowledge-barrier for users
- Re-use
  - General purpose hardware
  - Modular, upgradable piecemeal
  - Flexible, scalable architectures

CASPER

- Simplify
  - Leverage industry standards (eg, ethernet for interconnect)
  - Small number of custom [FPGA] platforms
  - Optimize for ease of use (not ops/watt, ops/rack unit)
  - Low knowledge-barrier for users
- Re-use
  - General purpose hardware
  - Modular, upgradable piecemeal
  - Flexible, scalable architectures

CASPER

- Simplify
    - Leverage industry standards (eg, ethernet for interconnect)
    - Small number of custom [FPGA] platforms
    - Optimize for ease of use (not ops/watt, ops/rack unit)
    - Low knowledge-barrier for users
- Re-use
    - General purpose hardware
    - Modular, upgradable piecemeal
    - Flexible, scalable architectures

CASPER

- Simplify
  - Leverage industry standards (eg, ethernet for interconnect)
  - Small number of custom [FPGA] platforms
  - Optimize for ease of use (not ops/watt, ops/rack unit)
  - Low knowledge-barrier for users
- Re-use
  - General purpose hardware
  - Modular, upgradable piecemeal
  - Flexible, scalable architectures

CASPER

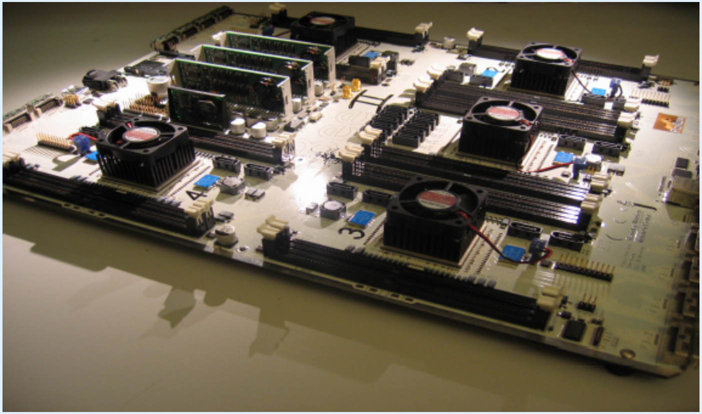
- Simplify
  - Leverage industry standards (eg, ethernet for interconnect)
  - Small number of custom [FPGA] platforms
  - Optimize for ease of use (not ops/watt, ops/rack unit)
  - Low knowledge-barrier for users
- Re-use
  - General purpose hardware
  - Modular, upgradable piecemeal
  - Flexible, scalable architectures

CASPER

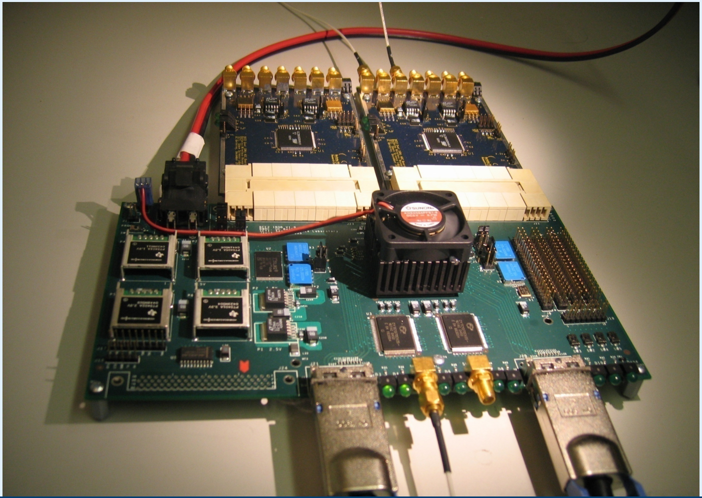
- Simplify
  - Leverage industry standards (eg, ethernet for interconnect)
  - Small number of custom [FPGA] platforms
  - Optimize for ease of use (not ops/watt, ops/rack unit)
  - Low knowledge-barrier for users
- Re-use
  - General purpose hardware
  - Modular, upgradable piecemeal
  - Flexible, scalable architectures

Outline

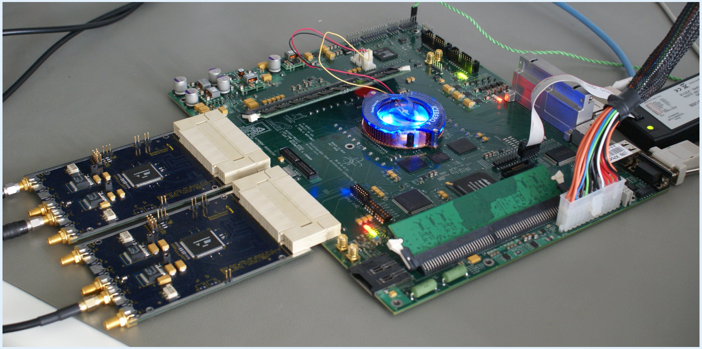
BEE2 (Virtex 2 Pro) 2005—



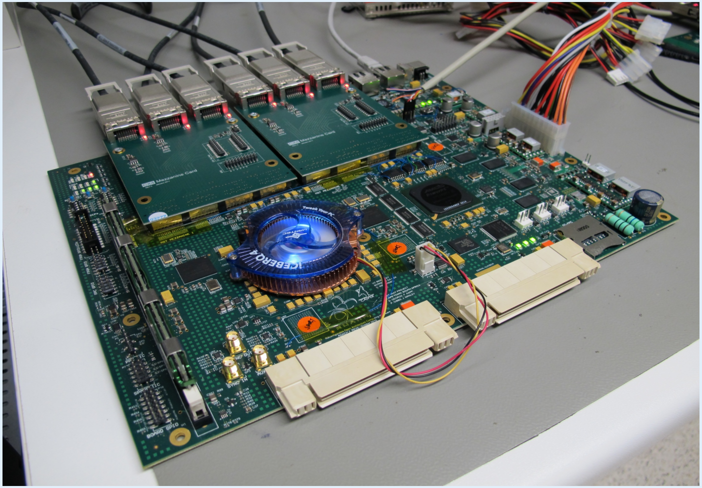
iBOB (Virtex 2 Pro) 2005—



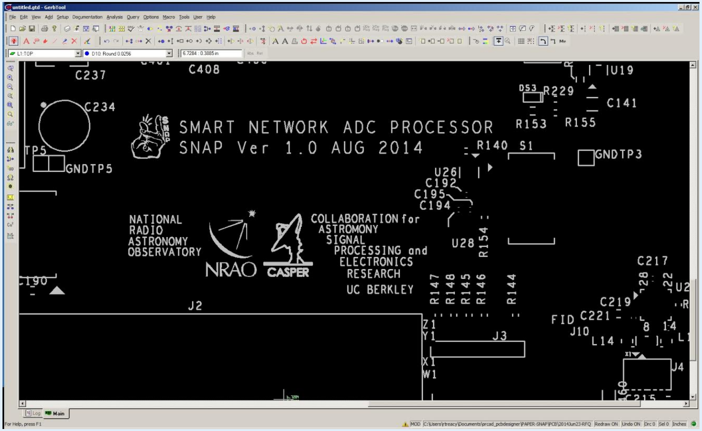
ROACH (Virtex 5 SX95T) 2009—



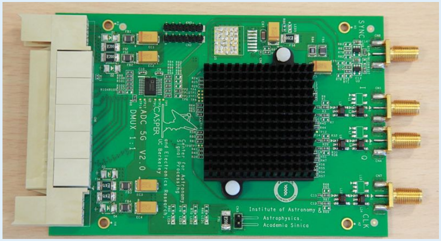
ROACH2 (Virtex 6 SX475T) 2010—



SNAP (Kintex 7 160T/325T/410T) 2014—



Lots of ADCs!



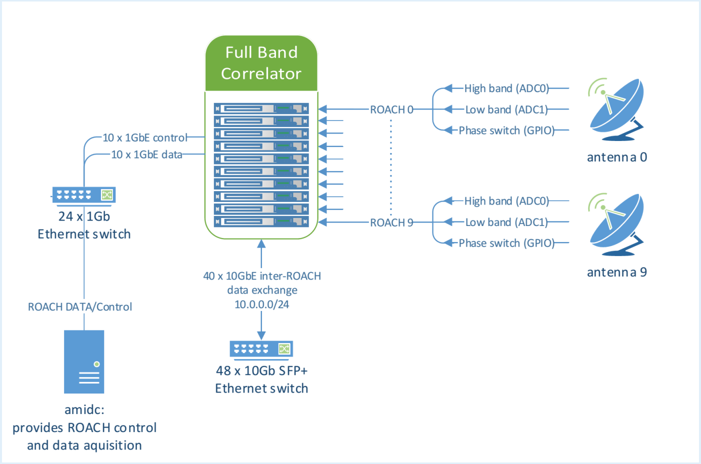
- 1 input, 8 bit, 5 GS/s
- 2 input, 8 bit, 1.5 GS/s
- 16 inputs, 8 bit, 250 MS/s
- 64 inputs, 12 bit, 65 MS/s
- DAC: 2 outputs, 16 bit, 1000 MS/s
- (and others, currently 11 boards listed on CASPER wiki)

Outline

The Arcminute Microkelvin Imager

- 10 antennas (single polarization)
- 2 $\times$ 2.5 GHz bands (12 – 18 GHz RF)
- 4096 frequency channels
- Integration length $\sim$ second
- Implemented on 10 ROACH-2 boards + ethernet switch

AMI architecture

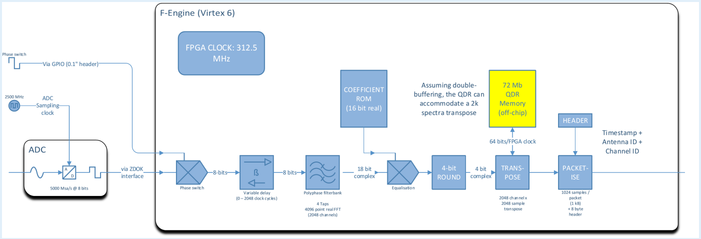


UNIVERSITY OF CAMBRIDGE

AMI architecture

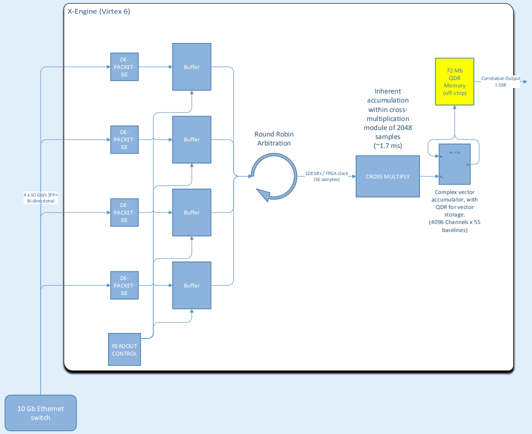
Mellanox SX1012 (12 x QSFP, 48 x SFP+)

AMI firmware



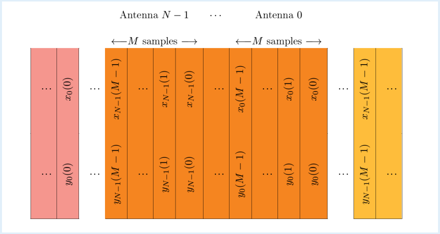
Two F-engines per ROACH-2 (one ADC per ZDOK interface)

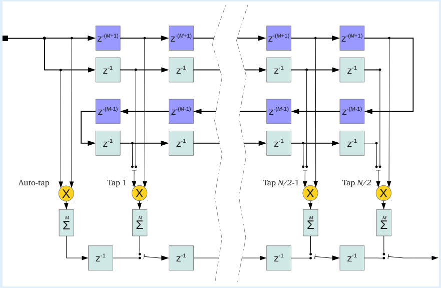
AMI firmware


X-Engine (Virtex 6)
DE-PACKET-ISE
DE-PACKET-ISE
DE-PACKET-ISE
DE-PACKET-ISE
Buffer
Buffer
Buffer
Buffer
READOUT CONTROL
4 x 10 Gb/s SFP+ Bi-direcitonal
10 Gb Ethernet switch
Round Robin Arbitration
128 bits / FPGA clock (16 samples)
CROSS MULTIPLY
Inherent accumulation within cross-multiplication module of 2048 samples (~1.7 ms)
$x_1 + x_2$
x_1
x_2
u_1
Complex vector accumulator, with QDR for vector storage. (4096 Channels x 55 baselines)
72 Mb QDR Memory (off-chip)
Correlation Output 1 GbE

UNIVERSITY OF CAMBRIDGE

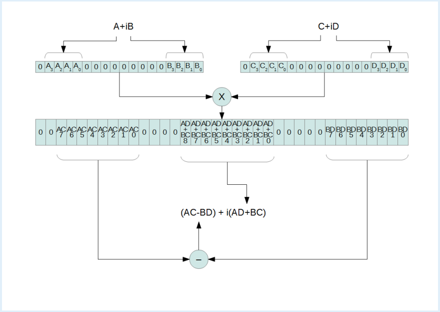
X-Engine firmware





X-Engine firmware

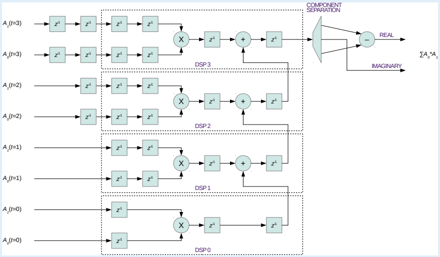
$$\begin{aligned} A &= a + ib \\ B &= c + id \\ A' &= a' + ib' = (a+1) + i(b+1) \\ B' &= c' + id' = (c+1) + i(d+1) \end{aligned}$$

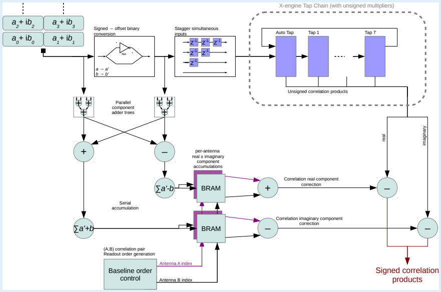


A+iB
C+iD
0 A3 A2 A1 A0 0 0 0 0 0 0 0 0 0 B3 B2 B1 B0
0 C3 C2 C1 C0 0 0 0 0 0 0 0 0 0 D3 D2 D1 D0
X
0 0 AC7 AC6 AC5 AC4 AC3 AC2 AC1 AC0 0 0 0 0 AD+BC8 AD+BC7 AD+BC6 AD+BC5 AD+BC4 AD+BC3 AD+BC2 AD+BC1 AD+BC0 0 0 0 0 0 BD7 BD6 BD5 BD4 BD3 BD2 BD1 BD0
(AC-BD) + i(AD+BC)
–

$$\begin{aligned} \langle A'^{*} B' \rangle &= \langle (a' - ib')(c' + id') \rangle \\ &= \langle [(a+1) - i(b+1)][(c+1) + i(d+1)] \rangle \\ &= \langle A^{*} B \rangle + \langle (a+b+c+d) - i(a-b-c+d) + 1 \rangle \\ &= \langle A^{*} B \rangle + \langle (a' + b + c' + d) - i(a' - b - c' + d) \rangle \end{aligned}$$

X-Engine firmware




X-engine Tap Chain (with unsigned multipliers)
$a_2 + ib_2$
$a_3 + ib_3$
$a_0 + ib_0$
$a_1 + ib_1$
Signed → offset binary conversion
Stagger simultaneous inputs
Auto Tap
Tap 1
Tap T
Unsigned correlation products
Parallel component adder trees
per-antenna real ± imaginary component accumulations
$\sum a'-b$
$\sum a'+b$
BRAM
BRAM
Serial accumulation
Correlation real component correction
Correlation imaginary component correction
real
imaginary
(A,B) correlation pair Readout order generation
Baseline order control
Antenna A index
Antenna B index
Signed correlation products

Outline

Goals of the CASPER toolflow

FPGAs as commodity signal processing hardware – PetaOp/Second FPGA Signal Processing for SETI and Radio Astronomy, Parsons et. al., 2006

Make FPGA design easy for CASPER users who are:

- NOT FPGA experts
- physics / astronomy undergraduates
- experimental physicists

The CASPER toolflow (attempts to):

- Hide implementation details users don't care about (a.k.a. almost EVERYTHING)
- Simplify DSP design for the bits users do care about
- Simplify loading design
- Simplify interacting with the design

Goals of the CASPER toolflow

FPGAs as commodity signal processing hardware – PetaOp/Second FPGA Signal Processing for SETI and Radio Astronomy, Parsons et. al., 2006
Make FPGA design easy for CASPER users who are:

- NOT FPGA experts
- physics / astronomy undergraduates
- experimental physicists

The CASPER toolflow (attempts to):

- Hide implementation details users don't care about (a.k.a. almost EVERYTHING)
- Simplify DSP design for the bits users do care about
- Simplify loading design
- Simplify interacting with the design

UNIVERSITY OF CAMBRIDGE

Goals of the CASPER toolflow

FPGAs as commodity signal processing hardware – PetaOp/Second FPGA Signal Processing for SETI and Radio Astronomy, Parsons et. al., 2006
Make FPGA design easy for CASPER users who are:

- NOT FPGA experts
- physics / astronomy undergraduates
- experimental physicists

The CASPER toolflow (attempts to):

- Hide implementation details users don't care about (a.k.a. almost EVERYTHING)
- Simplify DSP design for the bits users do care about
- Simplify loading design
- Simplify interacting with the design

UNIVERSITY OF CAMBRIDGE

Goals of the CASPER toolflow

FPGAs as commodity signal processing hardware – PetaOp/Second FPGA Signal Processing for SETI and Radio Astronomy, Parsons et. al., 2006
Make FPGA design easy for CASPER users who are:

- NOT FPGA experts
- physics / astronomy undergraduates
- experimental physicists

The CASPER toolflow (attempts to):

- Hide implementation details users don't care about (a.k.a. almost EVERYTHING)
- Simplify DSP design for the bits users do care about
- Simplify loading design
- Simplify interacting with the design

Goals of the CASPER toolflow

FPGAs as commodity signal processing hardware – PetaOp/Second FPGA Signal Processing for SETI and Radio Astronomy, Parsons et. al., 2006

Make FPGA design easy for CASPER users who are:

- NOT FPGA experts
- physics / astronomy undergraduates
- experimental physicists

The CASPER toolflow (attempts to):

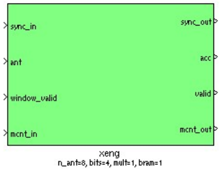
- Hide implementation details users don't care about (a.k.a. almost EVERYTHING)
- Simplify DSP design for the bits users do care about
- Simplify loading design
- Simplify interacting with the design

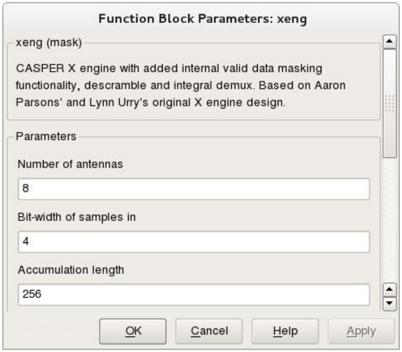
What is the CASPER toolflow?

- Simulink frontend – User creation
- Matlab middleware – peripheral management
- EDK project generation
- ISE backend
- Software Generation

Outline

Simulink Frontend




Function Block Parameters: xeng

xeng (mask)

CASPER X engine with added internal valid data masking functionality, descramble and integral demux. Based on Aaron Parsons' and Lynn Urry's original X engine design.

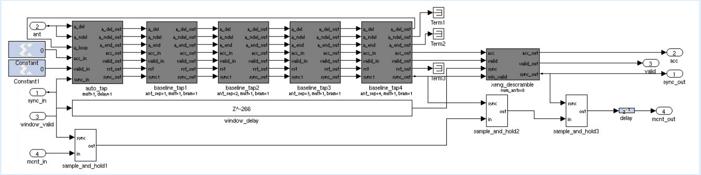
Parameters

Number of antennas: 8

Bit-width of samples in: 4

Accumulation length: 256

Simulink Frontend



Simulink Frontend

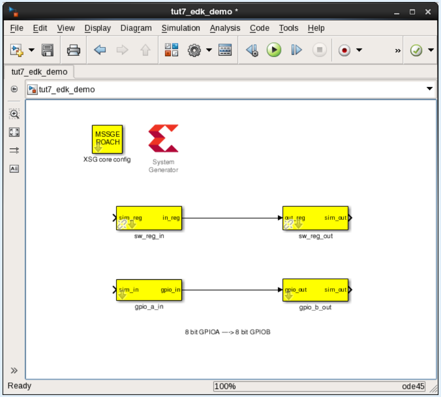
- Very intuitive
- Helps users (who probably have at least a little programming experience) appreciate they are designing circuits
- Automatic data type propagation
- Large parametrized simulink library available

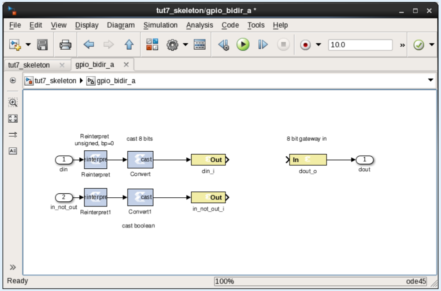
Simulink Frontend

Yellow blocks – interfaces to peripherals or other devices hidden by the toolflow

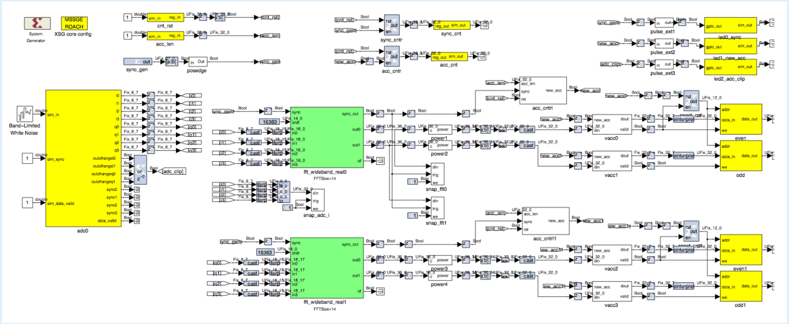
- ADCs
- DACs
- GPIO
- Memory – QDR / DRAM
- XAUI / Ethernet interfaces
- Software accessible memories / registers
- Platform configuration

Simulink Frontend





Simulink Frontend



Outline

EDK Backend

- Each platform in the CASPER suite has a complete EDK template project in the toolflow.
- The goal of the toolflow is to instantiate code into this project corresponding to the user's design (i.e., netlist generated by System Generator) and yellow block interfaces.
- After modification, the complete EDK project is compiled to bitstream

EDK Backend

Why EDK?

- Bus abstractions and address management
- Very tolerant specification format (.mhs)
  - Port width inference
  - Don't have to declare signals prior to use

EDK Backend

Why EDK?

- Bus abstractions and address management
- Very tolerant specification format (.mhs)
    - Port width inference
    - Don't have to declare signals prior to use

EDK Backend

Why EDK?

- Bus abstractions and address management
- Very tolerant specification format (.mhs)
  - Port width inference
  - Don't have to declare signals prior to use

EDK Backend

Why EDK?

- Bus abstractions and address management
- Very tolerant specification format (.mhs)
    - Port width inference
    - Don't have to declare signals prior to use

EDK Backend

```
# EPB Ports
PORT epb_clk_in  = epb_clk_in,  DIR = I
PORT epb_data    = epb_data,    DIR = IO, VEC = [15:0]
PORT epb_addr    = epb_addr,    DIR = I,  VEC = [22:0]
PORT epb_addr_gp = epb_addr_gp, DIR = I,  VEC =  [5:0]
...
BEGIN epb_opb_bridge
  PARAMETER INSTANCE = epb_opb_bridge_inst
  PARAMETER HW_VER   = 1.00.a
  BUS_INTERFACE MOPB = opb0
  PORT OPB_Clk       = epb_clk
  PORT epb_data_oe_n = epb_data_oe_n
  PORT epb_cs_n      = epb_cs_n_int
...
END
```

Outline

Matlab Control

- MATLAB controls how EDK is modified based on the contents of the users Simulink design.
- Each yellow block triggers construction of an xps_class (i.e. YellowBlock class) object whose attributes and methods determine how the EDK project is modified.

An Example Software Register

```
function b = xps_sw_register(blk_obj)
...
switch get_param(blk_name,'io_dir')
  case 'From Processor'
    b = set(b,'ip_name','opb_register_ppc2simulink');
  case 'To Processor'
    b = set(b,'ip_name','opb_register_simulink2ppc');
end
...
 misc ports
misc_ports.user_clk     = {1 'in'  get(xsg_obj,'clk_src')};
b = set(b,'misc_ports',misc_ports);
...
borph_info.size = 4;
b = set(b,'borph_info',borph_info);
```

An Example Software Register

Some key attributes:

- ip_name – the pcore (module) associated with this yellow block
- misc_ports – ports connected internally in EDK
  - Need only know name
- ext_ports – ports connected to FPGA IO
  - Must know name, width, direction, iostandard, loc
  - Refer to locs by platform independent names – e.g. zdok_0[39:0]
- borph_info – software bus connection information

An Example Software Register

Run class methods to modify EDK

- gen_mhs_ip – instantiate pcores, based on instance attributes
- gen_ucf_ip – add constraints to .ucf file, based on ext_ports

Compile bitstream

Software Generation

- Merge borph_info (register names, sizes, etc) into the bitstream to form a .bof file
- .bof file format is understood by software servers which ship as part of CASPER board software.
    - `fpga = FpgaClient(<ipaddress>)`
    - `fpga.progdev(<boffile_name>)`
    - `fpga.write_int(<register_name>, <value>)`

Software Generation

- Merge borph_info (register names, sizes, etc) into the bitstream to form a .bof file
- .bof file format is understood by software servers which ship as part of CASPER board software.
  - `fpga = FpgaClient(<ipaddress>)`
  - `fpga.progdev(<boffile_name>)`
  - `fpga.write_int(<register_name>, <value>)`

Outline

Frontend

- Simulink is intuitive, but it is slooooow (and enraging).
- Severe version compatiblity headaches.
- Poor version control support.
- Solution ????

Middleware

- MATLAB is not free (in beer, freedom or any other sense)
- No-one understands MATLAB OOP.
- Users will happily override methods and subvert the toolflow's intentions at every opportunity
- All code generation capabilities completely assume Xilinx EDK backend
- System Generator reliance baked in at every turn.
- Some strange and arbitrary limitations

EDK

- Time to use Vivado (or Altera!)
- Adds a layer of wrapping on top of simple verilog/vhdl modules. Another thing to learn.
- Some users want to play with CASPER hardware in ISE. A base package in EDK irritates them.
- Each hardware platform base package includes a complete set of pcores, even the platform-independent ones.

Outline

Solutions

- We can't solve all the problems yet.
- As a start, we can disentangle the Frontend, Middleware and Backend.
- A "Plan"
  - Keep Simulink Frontend (for now)
  - Standard (plaintext) format for describing yellowblocks (which any frontend could generate)
  - Pure Python middleware
  - Generate Verilog
  - Encapsulate platform-independent compile information (source files, software addresses, port constraints) in python objects
  - Pass all this information to a user-specified backend (Quartus, ISE, Vivado)

Solutions

- We can't solve all the problems yet.
- As a start, we can disentangle the Frontend, Middleware and Backend.
- A "Plan"
  - Keep Simulink Frontend (for now)
  - Standard (plaintext) format for describing yellowblocks (which any frontend could generate)
  - Pure Python middleware
  - Generate Verilog
  - Encapsulate platform-independent compile information (source files, software addresses, port constraints) in python objects
  - Pass all this information to a user-specified backend (Quartus, ISE, Vivado)

Solutions

- We can't solve all the problems yet.
- As a start, we can disentangle the Frontend, Middleware and Backend.
- A "Plan"
  - Keep Simulink Frontend (for now)
  - Standard (plaintext) format for describing yellowblocks (which any frontend could generate)
  - Pure Python middleware
  - Generate Verilog
  - Encapsulate platform-independent compile information (source files, software addresses, port constraints) in python objects
  - Pass all this information to a user-specified backend (Quartus, ISE, Vivado)

Solutions

- We can't solve all the problems yet.
- As a start, we can disentangle the Frontend, Middleware and Backend.
- A "Plan"
  - Keep Simulink Frontend (for now)
  - Standard (plaintext) format for describing yellowblocks (which any frontend could generate)
  - Pure Python middleware
  - Generate Verilog
  - Encapsulate platform-independent compile information (source files, software addresses, port constraints) in python objects
  - Pass all this information to a user-specified backend (Quartus, ISE, Vivado)

Solutions

- We can’t solve all the problems yet.
- As a start, we can disentangle the Frontend, Middleware and Backend.
- A “Plan”
  - Keep Simulink Frontend (for now)
  - Standard (plaintext) format for describing yellowblocks (which any frontend could generate)
  - Pure Python middleware
  - Generate Verilog
  - Encapsulate platform-independent compile information (source files, software addresses, port constraints) in python objects
  - Pass all this information to a user-specified backend (Quartus, ISE, Vivado)

An Example

A Simulink generated peripherals file (YAML), specifying everything you need to know to put yellow blocks and the simulink netlist into a complete design.

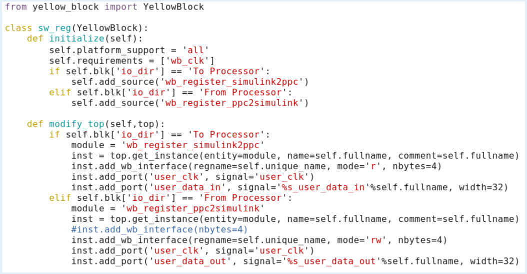
```
my_model/sw:
  name: sw
  fullpath: my_model/sw
  tag: xps:sw_reg
  mode: one value
  io_dir: To Processor
  ...
my_model/xsg:
  name: xsg
  fullpath: my_model/xsg
  tag: xps:xsg
  hw_sys: 'KC705:xc7k325t'
  clk_src: sys_clk
  ...
```

```
...
user_modules:
  my_model:
    clock: clk
    ports:
      - my_model_sw_user_data_out
      - my_model_gpio_gateway
    sources:
      - test_models/my_model/sysgen
```

UNIVERSITY OF CAMBRIDGE

An Example

- A sample class for software registers.
- Commands are called against the top verilog module class, so it's (hopefully) obvious what they do.
- No more arbitrary misc_ports, ext_ports, inferred ports distinctions
- Assumes nothing about the number of modules / wishbone interfaces associated with a yellow block
- get_instance will return an existing module with the same name



```python
from yellow_block import YellowBlock

class sw_reg(YellowBlock):
    def initialize(self):
        self.platform_support = 'all'
        self.requirements = ['wb_clk']
        if self.blk['io_dir'] == 'To Processor':
            self.add_source('wb_register_simulink2ppc')
        elif self.blk['io_dir'] == 'From Processor':
            self.add_source('wb_register_ppc2simulink')

    def modify_top(self,top):
        if self.blk['io_dir'] == 'To Processor':
            module = 'wb_register_simulink2ppc'
            inst = top.get_instance(entity=module, name=self.fullname, comment=self.fullname)
            inst.add_wb_interface(regname=self.unique_name, mode='r', nbytes=4)
            inst.add_port('user_clk', signal='user_clk')
            inst.add_port('user_data_in', signal='%s_user_data_in'%self.fullname, width=32)
        elif self.blk['io_dir'] == 'From Processor':
            module = 'wb_register_ppc2simulink'
            inst = top.get_instance(entity=module, name=self.fullname, comment=self.fullname)
            #inst.add_wb_interface(nbytes=4)
            inst.add_wb_interface(regname=self.unique_name, mode='rw', nbytes=4)
            inst.add_port('user_clk', signal='user_clk')
            inst.add_port('user_data_out', signal='%s_user_data_out'%self.fullname, width=32)
```

An Example

```
localparam N_WB_SLAVES    = 5;

localparam SLAVE_BASE = {
  32'h0001000c, // sw1
  32'h00010008, // sw2
  32'h00010004, // sw3
  32'h00010000, // sw
  32'h00000000
};

localparam SLAVE_HIGH = {
  32'h0001000f, // sw1
  32'h0001000b, // sw2
  32'h00010007, // sw3
  32'h00010003, // sw
  32'h0000ffff
};
```

```
wire    [N_WB_SLAVES - 1:0] wbs_cyc_o;
wire    [N_WB_SLAVES - 1:0] wbs_stb_o;
wire                        wbs_we_o;
wire                  [3:0] wbs_sel_o;
wire                 [31:0] wbs_adr_o;
wire                 [31:0] wbs_dat_o;
wire [32*N_WB_SLAVES - 1:0] wbs_dat_i;
wire    [N_WB_SLAVES - 1:0] wbs_ack_i;
wire    [N_WB_SLAVES - 1:0] wbs_err_i;

wbs_arbiter #(
  .NUM_SLAVES (N_WB_SLAVES),
  .SLAVE_ADDR (SLAVE_BASE),
  .SLAVE_HIGH (SLAVE_HIGH),
  .TIMEOUT    (1024)
) wbs_arbiter_inst (
  .wb_clk_i  (wb_clk_i),
  .wb_rst_i  (wb_rst_i),
  ...
);
```

An Example

- Constraints added as python objects
- Toolflow can map these into LOCs / IOSTANDARDs
- Up to the Backend to decide how to format the constraint files
- (A lot like migen)

```
def gen_constraints(self):
        return [
        PortConstraint('mgt_ref_clk_p', 'eth_clk_p'),
        PortConstraint('mgt_ref_clk_n', 'eth_clk_n'),
        PortConstraint('mgt_tx_p%d'%self.port, 'mgt_tx_p%d'%s
        PortConstraint('mgt_tx_n%d'%self.port, 'mgt_tx_n%d'%s
        PortConstraint('mgt_rx_p%d'%self.port, 'mgt_rx_p%d'%s
        PortConstraint('mgt_rx_n%d'%self.port, 'mgt_rx_n%d'%s
        ClockConstraint('mgt_reg_clk_p', name='ethclk', freq=
        ]
```

Final compilation steps

- Software infrastructure for register mappings kept
- boffiles can be loaded over ethernet as usual

Current State of Affairs

- 10 GbE, software registers, brams, GPIO and some ADCs have been ported
- Two backends working
  - PlanAhead (for Virtex 6 & ROACH boards)
  - Vivado (for upcoming Kintex 7 CASPER boards, tested with KC705 dev board)

Outline

Lessons

- General CASPER musings...
  - Just getting everyone using the same hardware / software is half the battle.
  - Even a community of new users can be very effective by just sharing what worked for them.
  - Ease of use needs to be extremely good to get (and keep) the attention of new users.
  - A toolflow can't just end at bitstream generation
- The CASPER toolflow
  - Clever toolflows are not clever if no-one can use them
  - Those who can make contributions to the toolflow's hardware support are EE/HDL pros, not necessarily MATLAB OOP gurus.
  - No-one in CASPER will be happy until there is a viable alternative to Simulink supported by CASPER. Suggestions?

Lessons

- General CASPER musings...
  - Just getting everyone using the same hardware / software is half the battle.
  - Even a community of new users can be very effective by just sharing what worked for them.
  - Ease of use needs to be extremely good to get (and keep) the attention of new users.
  - A toolflow can't just end at bitstream generation
- The CASPER toolflow
  - Clever toolflows are not clever if no-one can use them
  - Those who can make contributions to the toolflow's hardware support are EE/HDL pros, not necessarily MATLAB OOP gurus.
  - No-one in CASPER will be happy until there is a viable alternative to Simulink supported by CASPER. Suggestions?

Thanks

Thanks!