Hacking USRP gateware

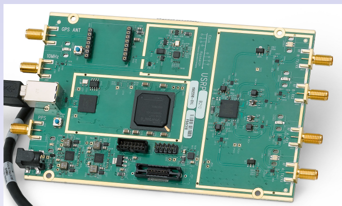
Gwenhaël GOAVEC-MEROU, Jean-Michel FRIEDT
gwenhael.goavec@femto-st.fr

slides at `www.trabucayre.com/gnuradioDays2023.pdf`

March 30, 2023

Target platforms

B210:



source: EttusResearch

- Xilinx Spartan6 LX150
- ADi AD936x RF frontend
- USB3 interface
- one monolitic gateware → a text editor is required to modify it

X310:

source: EttusResearch

- Xilinx Kintex7 410T
- ADC + DAC interface (requires daughterboards)
- Ethernet 1G/10G
- RFNoC support → OOT block / tool box

Motivations

- adding new module into the gateware
  - in stream processing
  - out of stream (independent task)

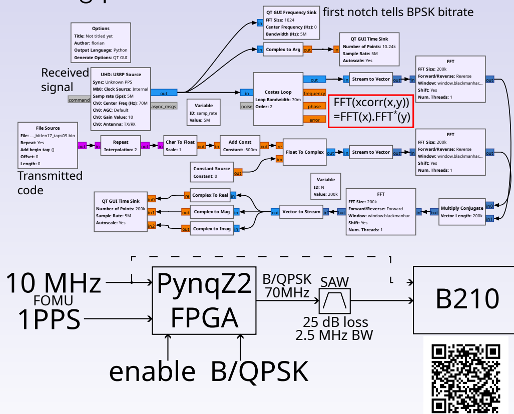
Examples:

- tagging/corrupting a sample on a PPS's rising edge (internal or external) (in stream): allows to know relationship between event and sample (**warning** keep in mind latency introduce by the frontend and processing chain)
- Two Way Time Transfer: PRN generation aligned to and external PPS (out of stream task)

USRPSs `B210` & `X310` have free resources → avoid to add an additional FPGA (for out stream), required when stream must be modified.

TWSTFT: out stream example

The big picture:



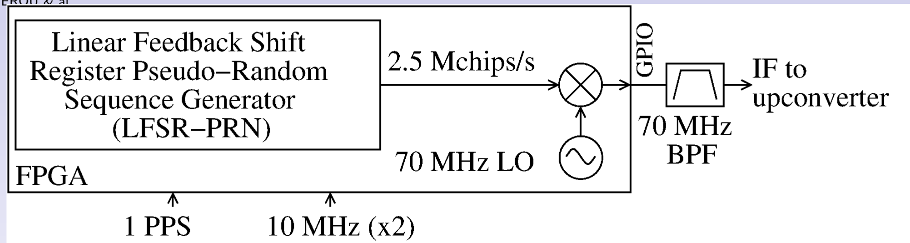
See demo

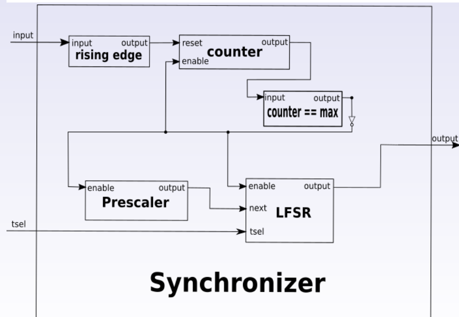
- PynqZ2 Xilinx xc7z020
- 10MHz & PPS: FOMU Lattice iCE40 UP5k
- SAW filter
- acquisition: B210 + GNU Radio

Principle:

- PRN generator @2.5MS/s + 70MHz BPSK or QPSK
- a GPIO is used as output

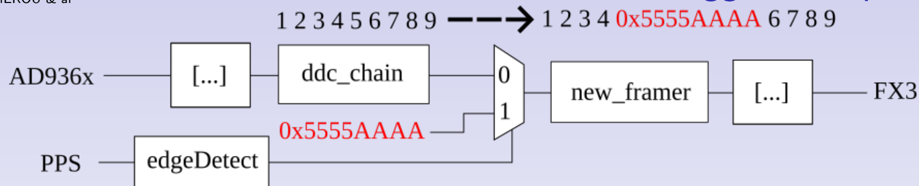
TWSTFT: out stream example





- up: full scheme placed in parallel to the stream (can be used on a standalone FPGA)
- left: internal schematic details

GNSS-SDR 1PPS: stream tagger example (B210)



- needs an input GPIO
  - may be a front panel GPIO by disabling GPIO controler
  - or the 1PPS SMA input
- logic is introduced by cutting direct link between `ddc_chain` module and `new_framer` module.
- replaces one sample when PPS rise edge is detected

Warning: after `new_framer` data may be converted: a search is required to find corresponding value.

```
#define TAG_RESC16 0x53AB2A3F
#define TAG_IMSC16 0x53AD2ABF
[...]
myconversion.myint=TAG_RESC16;
_tag_pps_int.real(myconversion.→
    ↪myfloat);
if (input[i] == _tag_pps_int)
    // do something
```

Modifying gateware instead of creating a new one

Pros:

- base is available → adding new module is straightforward
- writing a new gateware may be an heavy task:
  - RF Frontend is complex to configure
  - requires the software part too
  - (B210 only) a firmware for the cypress interface is also required
- long term works → whitout a community may be hard to maintain
- overkill for a simple task (adding a subtree)

Cons:

- no control to the gateware (patches to refresh, repository to resync/rebase to do, ... for each new release)
- sometime hard/impossible to mainline (partially or totally) modifications
- sustainability?

Is is relevant to modify USRP's gateware?

→ For in stream processing: mandatory / no way to access samples otherwise
→ independant tasks: it depends:

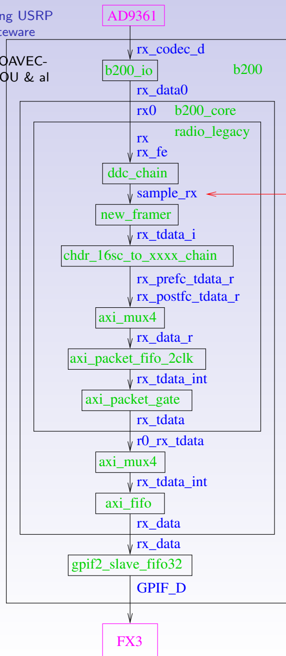
- this avoid to have too many hardware, it's more easy to move, less "fragile"
- bitstream generation (Synthesis, Place and Routage) time is longer for B210 and X310 compared to a standalone FPGA (original gateware takes lot of resources)

TWSTFT project works on a standalone FPGA, B210 and X310

- ≈ one/two minute to generate + loading the bitstream on a Zynq based board;
- ≈ 45minutes to generate the bitstream for B210
- ≈ 1h30 for X310

⇒ it is relevant to avoid use of an USRP in test / development phase to increase productivity (when it is possible).

G. GOAVEC-MEROU & al



add here
data stream
modification
[15: 0] –> I
[31:16] –> Q

X: Verilog module
X: signal name
X: physical device

B210 gateware modifications: in stream

Requires to find where to place code (cut stream)

- too early: breaks AD936x calibration
- too late: packets contains more than just samples. More difficult to modify stream

Examples: replacing next sample by a tag on an external or internal signal rise detection. See:

https://github.com/oscimp/gnss-sdr-1pps

```
+    wire [31:0] sample_rx_ggm;
     /* add something here with sample_rx as input
        and sample_rx_ggm as output */
     new_rx_framer #(.BASE(SR_RX_CTRL+4),.→
         ↪SAMPLE_FIFO_SIZE(SAMPLE_FIFO_SIZE)) →
         ↪new_rx_framer
     [...]
-       .strobe(strobe_rx), .sample(sample_rx), .run(→
     ↪run_rx), .eob(eob_rx), .full(full),
+       .strobe(strobe_rx), .sample(sample_rx_ggm), .run→
     ↪(run_rx), .eob(eob_rx), .full(full),
     [....]
```

independent tasks / unrelated to the stream

Example TWSTFT see.
`github.com/oscimp/amaranth_twstft/blob/main/0001-b200-add-pps_gen-stream-tag-and-twstft-amaranth_uhd-v4.1.0.5.patch`

Top verilog module (`b200.v`) is the best location to insert code

Again Makefile must be appended:

```
B210:  ##Build USRP B210 design.
    [...]
    python -m amaranth_twstft.flashZedBoard --no-load --no-build --build-dir amaranth
    [...]
```

`Makefile.b200.inc` must also be updated (inject files to integrate to the project)

```
diff --git a/fpga/usrp3/top/b200/Makefile.b200.inc b/fpga/usrp3/top/b200/Makefile.b200.inc
index 788280c6a..e7beb5923 100644
--- a/fpga/usrp3/top/b200/Makefile.b200.inc
+++ b/fpga/usrp3/top/b200/Makefile.b200.inc
@@ -69,6 +69,10 @@ b200.v \
 b200_core.v \
 b200_io.v \
 b200.ucf \
+my_pps_gen.v +ggm_b210_hack.v amaranth/top.v pps_atr.v \
 timing.ucf \
 [...]
```

adding an out of stream task for X310

Similar to B210 (mainly Makefile to update and top verilog module to modify)

- Makefile update (adding verilog files / rules when generated using high level language)
- disable gpio controller to take control to Front Panel GPIOs
- additional verilog code instantiation

Makefile update

```
X310_HG:
+	python -m amaranth_twstft.flashZedBoard --no-load \
+		--no-build --build-dir amaranth $(AMARANTH_OPTS)
	$(call vivado_build,X310,$(HG_DEFS) X310=1)
	$(call post_build,X310,HG)
```

instantiating a module

```
mixer mixer (
	.clk(amaranth_clk280),
	.global_enable(FrontPanelGpio9),
	.output_carrier(FrontPanelGpio7),
	.mod_out(FrontPanelGpio10),
	.pps_in(EXT_PPS_IN),
	.pps_out(FrontPanelGpio8),
	.the_pps_we_love(FrontPanelGpio11),
	.rst(!amaranth_rst),
	.invert_prn_o(FrontPanelGpio6),
	.switch_mode(1'b0/*FrontPanelGpio11*/)
;
```

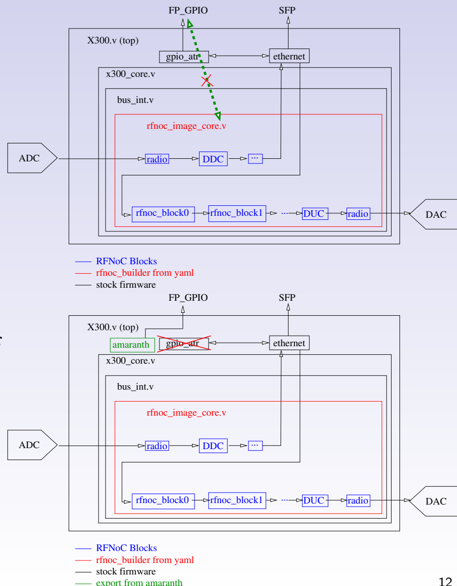
RFNoC on X310 **WIP**

Default gateware: access to GPIOs and PPS → no way.
RFNoC blocks are too deep in gateware tree

- modifying top level verilog module remain possible (as B210)
- possible to add your OOT block with these input/output
- middle verilog file is autogenerated → don't try to modify: will be overwritten

Solution: Modifying (sic) yml files used to describes interface and platform
⇒ relevant to open a PR (TODO) but not sure if (un)acceptable
For recent (3.10) GNU Radio release: not clear

- docs/workshop refers to 3.7 (UHD 3.5) or 3.8 (UHD 4) and to gr-ettus / rfnocmodtools
- gr-ettus will no more be updated for 3.10
- sometime questions about this topic on mailing list



RFNoC structure modification: adding new signal

Requires to update UHD files:

- `io_signature.yml`: interface, signals, ... definition
- `x310_bsp.yml`: platform definition
- yml file used by `rfnoc_builder` to build `rfnoc_image_core.v`

your OOT block yml

```
io_ports:
  pps:
    type: pps_if
    drive: listener
```

`io_signature.yml`

```
[...]
pps_if:
  type: slave
  ports:
  - name: pps_i
    width: 1
[...]
```

`x310_bsp.yml`

```
[...]
io_ports:
  pps:
    type: pps_if
    drive: broadcaster
[...]
```

And the dedicated OOT block core definition/example (`rfnoc/icores/xxx_core.yml`)

```
[...]
connections:
  [...]
  # BSP connections
  [...]
  - { srcblk: _device_, srcport: pps, dstblk: instanceOfTheOOTBlock, dstport: pps}
[...]
```

Coffee time: long build time...

how to create new module

FPGA not a CPU $\rightarrow$ needs to use a specific HDL (High level Description Language).
CPU: say what it should do.
FPGA: say what it should be.

- VHDL: verbose, strong typed. Based on ADA structure;
- verilog: more permissive, less typed (sometime error prone).
- systemverilog: an evolution of verilog language

But not limited to these language:

- Chisel: scala based approach
- spinalHDL: a Chisel fork/evolution
- Migen + LiteX: python based language
- Amaranth (previously: nmigen): another python based language (a reboot of Migen)
- flexC, Silice, ...

$\Rightarrow$ Many new language with a more expressivity $\rightarrow$ ease/speed up creating module/gateware

Simple square wave: Verilog/VHDL (aka ASM for FPGA)

Simple square wave ⇒ blink led!

```
module blink(
    input i_clk,
    input i_rst,
    output led
);
reg [31:0] cnt;
// async operation
assign led = cnt < 5_000_000-1;

// sync operations
always @(posedge i_clk) begin
    cnt <= (cnt < 10_000_000-1)cnt + 1'b1 : →
        ↪32'b0;
    if (i_rst)
        cnt <= 32'b0;
end
endmodule
```

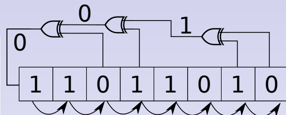
```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.numeric_std.all;
Entity blink is
    port (
        i_clk : in std_logic;
        i_rst : in std_logic;
        led   : out std_logic
    );
end entity;
Architecture bhv of blink is
    signal cnt : unsigned(31 downto 0);
begin
    led <= '1' when cnt < 5000000-1 else '0';
    process(clk_i) begin
        if rising_edge(clk_i) then
            cnt <= cnt + 1;
            if rst_i = '1' or cnt == →
                ↪10000000-1 then
                cnt <= (others => '0');
            end if;
        end if;
    end process;
end Architecture bhv;
```

Same task, two languages:

- above: verilog: concise
- right: VHDL: more verbose.

Not enough → needs to create a constrains file (pin mapping) and manage FPGA tools.

Amaranth: a PRN generator




```
class PrnGenerator(Elaboratable):
    def __init__(self,bit_len , taps = 0,seed = 1):
        self.output = Signal(name="prn_output")
        self._taps  = Signal(bit_len , reset = taps , name="taps")
        self.next   = Signal(name="prn_next")
        self.enable = Signal(reset = 0,name="prn_enable")
        self.reg    = Signal(bit_len , reset = seed ,
                              name="register")
    def elaborate(self,platform):
        m = Module()

        insert = Signal(name="register_input")
        m.d.comb += [
            insert.eq((self._taps & self.reg).xor()),
            self.output.eq(self.reg[0]),
        ]

        with m.If(self.enable):
            m.d.sync += self.reg.eq(Cat(self.reg[1:],insert))
        with m.Else():
            m.d.sync += self.reg.eq(self.reg.reset)

        return m
```

python produces verilog code:

- may be directly used ⇒
- or instanciate into existing project: see: `github.com/oscimp/amaranth_twstft/`

Dealing with pin mapping / FPGA tools:

```
class TWSTFT_top(Elaboratable):
    def __init__(self, bitlen, taps):
        self._taps   = taps
        self._bitlen = bitlen
    def elaborate(self, platform):
        m = Module()
        m.submodules.prn = prn = PRNGenerator(self._bitlen, self→
            ↪._taps)
        output = platform.request("one_pin", 0)
        m.d.comb += output.eq(prn.output)
        return m
# build gateware and load
ThePlatform().build(TWSTFT_top(bitlen, taps=taps),
                    do_program=True, do_build=True)
```

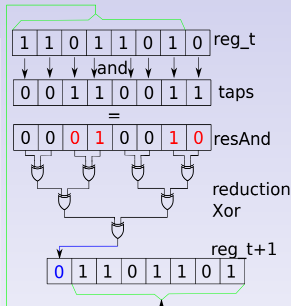
Amaranth: a PRN generator

```python
class PrnGenerator(Elaboratable):
    def __init__(self, bit_len, taps = 0, seed = 1):
        self.output = Signal(name="prn_output")
        self._taps  = Signal(bit_len, reset = taps, name="taps")
        self.next   = Signal(name="prn_next")
        self.enable = Signal(reset = 0, name="prn_enable")
        self.reg    = Signal(bit_len, reset = seed,
                             name="register")
    def elaborate(self, platform):
        m = Module()

        insert = Signal(name="register_input")
        m.d.comb += [
            insert.eq((self._taps & self.reg).xor()),
            self.output.eq(self.reg[0]),
        ]

        with m.If(self.enable):
            m.d.sync += self.reg.eq(Cat(self.reg[1:], insert))
        with m.Else():
            m.d.sync += self.reg.eq(self.reg.reset)

        return m
```



python produces verilog code:

- may be directly used ⇒
- or instanciate into existing project: see: github.com/oscimp/amaranth_twstft/

Dealing with pin mapping / FPGA tools:

```python
class TWSTFT_top(Elaboratable):
    def __init__(self, bitlen, taps):
        self._taps   = taps
        self._bitlen = bitlen
    def elaborate(self, platform):
        m = Module()
        m.submodules.prn = prn = PRNGenerator(self._bitlen, self→
            ↪._taps)
        output = platform.request("one_pin", 0)
        m.d.comb += output.eq(prn.output)
        return m
# build gateware and load
ThePlatform().build(TWSTFT_top(bitlen, taps=taps),
                    do_program=True, do_build=True)
```

1Hz square wave + 10MHz : LiteX/Migen

See demonstration: it's the clock and PPS generator (connected to PynQ USB port)

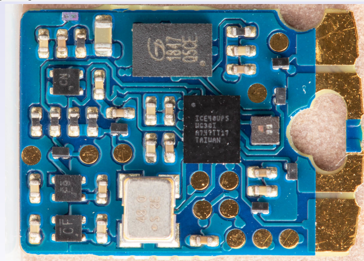
```
class BaseSoC(SoCCore):
    def __init__(self, sys_clk_freq = int(20e6), **kwargs):
        platform = kosagi_fomu_pvt.Platform()# platform def (pin/clk/...)
        self.crg = _CRG(platform, sys_clk_freq)
        SoCCore.__init__(self, platform, sys_clk_freq,
            ident="10M 1PPS", **kwargs) # soc (but totaly unused)
        clk10M_out1 = platform.request("user_touch_n", 0) # touch_pins0
        clk10M_out2 = platform.request("user_touch_n", 1) # touch_pins1
        pps_out     = platform.request("user_touch_n", 3) # touch_pins3

        # pps generation
        pps_cnt      = Signal(max=int(sys_clk_freq)) # counter
        pps_cnt_next = pps_cnt + 1
        self.comb += clk10M_out2.eq(clk10M_out1) # async operation
        self.sync += [ # synchrounous operations
            clk10M_out1.eq(~clk10M_out1), # 20 -> 10MHz
            pps_out.eq(pps_cnt_next < int((sys_clk_freq * 200e-3))),
            If(pps_cnt == int(sys_clk_freq)-1,
                pps_cnt.eq(0)
            ).Else(
                pps_cnt.eq(pps_cnt_next)
            )
        ]
```

Bitstream + flash

```
[...]
soc = BaseSoC(args.sys_clk_freq, **parser.soc_argdict)
builder = Builder(soc, **parser.builder_argdict)
if args.build:
    builder.build(**parser.toolchain_argdict)
if args.load:
    flash(builder.output_dir, soc.build_name, args.timeout)
```

```
# CRG (Clock And Reset) snip
[...]
pll_lock = Signal()
self.pll = pll = iCE40PLL()
pll.clko_freq_range = ( 12e6,  275e9)
pll.register_clkin(clk48, 48e6) # in clock
# sys_clk_freq == 20MHz (16MHz <= output <= 275→
    ↪ MHz
pll.create_clkout(self.cd_sys, sys_clk_freq,
    with_reset=False)
[...]
```



Don't panic: it's a (more or less) full gateware

Conclusion

Modifying `gateware` for USRP is straightforward and limited to:

- updating `Makefilexxx` to:
  - reference new files
  - add new rule / modifying existing
- code instanciation and potential tree modifications

But no control to the repository:

- a `UHD` fork is required or at least patch generate $\rightarrow$ save work
- an hybrid approach by copying Makefile and refering to `UHD` repository
- in both case: for each new release $\Rightarrow$ full update (modifications, bitstream, ...)

Modifying a gateware vs creating a new one is dependent of multifactors

- people interested, able to help
- task to add (for 10 lines of code a big dev is not a good idea)

But

- writing a custom gateware allows to be free/independent to official one