Erlang for UniBoard: *f*unctional monitoring and control

SEVENTH FRAMEWORK PROGRAMME
Contract nr. 227290

Er·lang /ˈɜːlæŋ/

Is a functional programming language and runtime system specifically built for distributed monitoring-and-control of large soft-realtime (hardware)systems.

Erlang IRL Factsheet

In active development since 1988 by Ericsson of Sweden
Open-sourced in 1998
Battle tested on Ericsson's carrier grade ATM switch AXD301
Facebook Chat is powered (amongst others) by Erlang
Runs on Linux, *BSD, OSX and Windows in 32 and/or 64 bit

Functional Language: eliminate classes of bugs

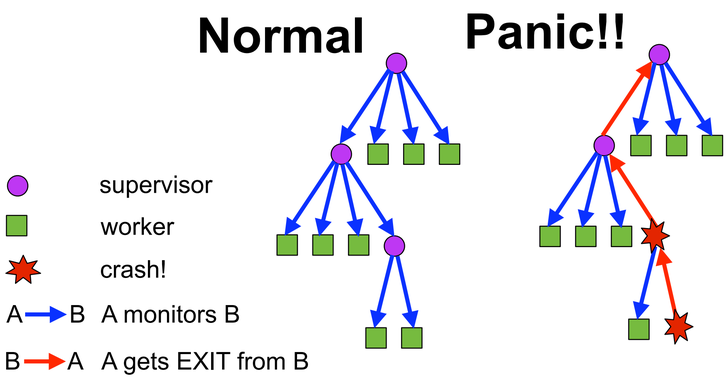
*f*unctions can be used like variables
Being side-effect free eliminates **~20%** of bugs
No global state ("variables") yields safety *and* lock-free multiprocessing
Variables aren't. Single assignment only eliminates **~60%** of bugs
Pattern match in stead of direct function-call
Conditional matching based on properties of the *f*unction call arguments

Erlang Goodies

Dealing with binary data is built in to the language
Pattern matching on binary patterns
Extremely lightweight concurrency (10^6 processes easily)
Distributes transparently across machines and systems
Hot code-swapping makes for uninterrupted services
Assume hard- and software *will* fail
Infrastructure and utilities to monitor and handle are built in to the system

Monitoring: supervision trees

The key to Erlang's fault tolerance. Processes can monitor each other and get notified if a 'linked' process crashes. Processes are idiomatically divided into supervisors and workers. Workers perform tasks and if they crash their supervisor takes appropriate action or pass the error on up the supervision tree.



Control: message handling

Pattern match on an what is received. `receive` is a keyword in the Erlang language. The function below handles messages and (tail)recurses to handle the next message.

```
handle_message() ->
    receive
        % Did we receive the string literal "hello"?
        "hello" ->
            io:format(" world!~n"),
            handle_message();
        % Did we receive a number? Then compute+print its reciprocal
        N when is_number(N) ->
            io:format("1/~p = ~p~n", [N, 1/N]),
            handle_message();
        % Maybe it was XDR opaque data:
        % 32bit unsigned <Length> in network byte order (big
        % endian) followed by exactly <Length> unsigned 8-bit bytes.
        % The pattern is written such that it will only match if
        % the message is exactly received like this.
        << Length:32/unsigned-big, Data:Length/unsigned-unit:8 >> ->
            io:format("Received ~p bytes of data ~w~n",
                      [Length, Data]),
            handle_message()
    end.
```

Control: fun with processes

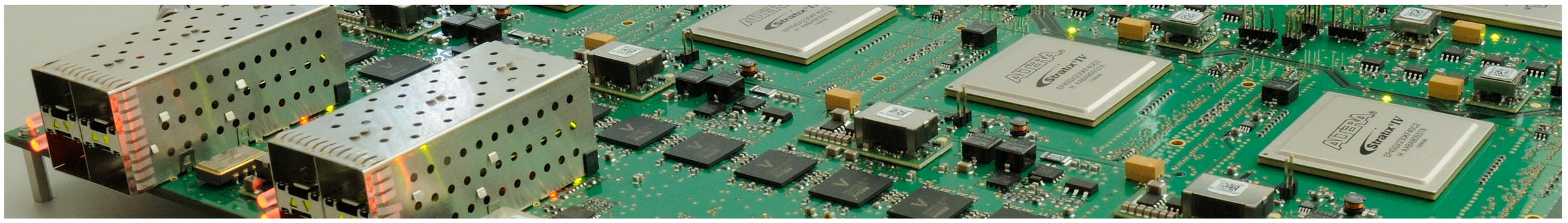
"`spawn`" spawns a new Erlang process and returns an *Erlang* process-id. "`!`" is the 'send' operator: `process-id ! <term>` sends the Erlang term `<term>` to the indicated process.

```
% Start the handle_message function from the previous block in
% a separate process
% "fun() -> ... end" is Erlang syntax for an anonymous function/lambda
1> ProcessID = spawn( fun () -> handle_message() end ).

% Send the literal string "hello" - we know how this should be handled
2> ProcessID ! "hello".
world!

% Make it compute + print the reciprocal of 3
3> ProcessID ! 3.
1/3 = 0.3333333333333333

% Trigger a fatal error in the message handler.
% All processes monitoring the handler process will be informed
% (currently only the Erlang interactive shell)
4> ProcessID ! 0.
=ERROR REPORT==== 1-Nov-2011::17:26:01 ===
Error in process <0.96.0> with exit value: {badarith,[{erlang,'/',
[1.000000e+00,0]},{erl_eval,do_apply,5},{erl_eval,expr,5},
{erl_eval,expr,5},{erl_eval,expr_list,6},{erl_eval,expr,5},
{erl_eval,exprs,5}]}
```



UniBoard

A multi-purpose, FPGA based, scalable, high-performance computing platform for radio-astronomical applications such as correlators or beamformers. The board is developed by an international consortium of radio-astronomical institutes.

UniBoard specifications

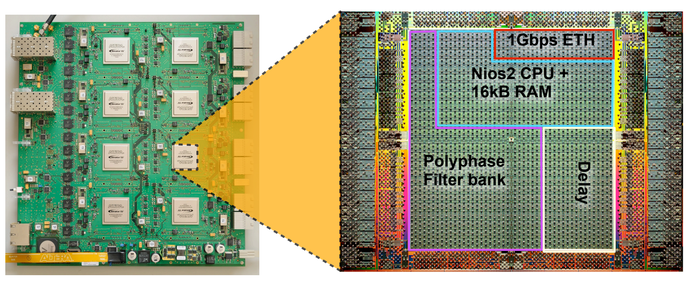
8 Altera Stratix IV FPGA (=0.4TOPS total)
2 x 16 x 10Gbps I/O
64GB RAM
240Gbps inter-FPGA IO
separate 1Gbps control network

Monitoring and control

Its eight FPGAs will be individually controlled using a binary protocol over UDP/IPv4. Some applications use » one UniBoard. The APERTIF beam former and correlator will require over a hundred, necessitating control of > 800 FPGAs. This is exactly what Erlang was designed and built for.

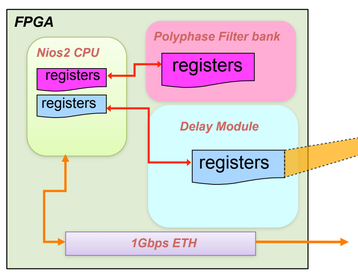
UniBoard at the Joint Institute for VLBI in Europe (JIVE)

JIVE will implement the European VLBI Network's (EVN) next generation correlator using UniBoard. Illustrated is how the correlator is implemented and controlled. The trail reads from the hardware on the left via progressive abstractions/zooms to software control on the right.

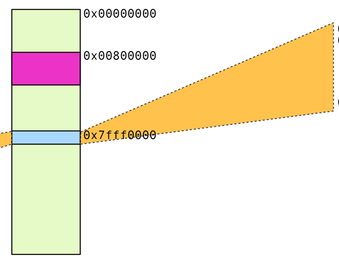


1. *A UniBoard, 8 FPGAs*

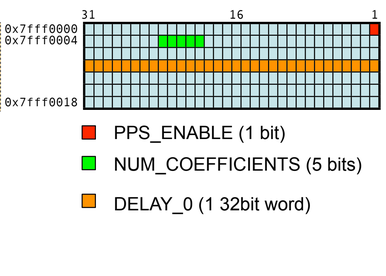
2. *A single FPGA with some VHDL on it*



3. *Schematic overview of* 2.



4. *Memorymap of the Nios2 CPU*



5. *Register layout of the* Delay Module

File: delay_module.erl

```
-module(delay_module).

registers() ->
  [ fpga:bit(pps_enable,1, 16#7fff0000),
    fpga:bitrange(num_coefficients, 5, 19, 16#7fff0004),
    fpga:word(delay_0, 16#7fff000c) ].

set_delay(FPGA, Delay) ->
  OldPPS = fpga:read(pps_enable),
  fpga:execute(FPGA, [fpga:write(pps_enable, 0),
                      fpga:write(num_coefficients, 1),
                      fpga:write(delay_0, Delay),
                      fpga:write(pps_enable, OldPPS)]).
```

6. *High-level Erlang code to drive the* Delay Module

http://www.erlang.org

Harro Verkouter verkouter@jive.nl ADASS XXI - Paris