Evolutionary Fault-Tolerant Systems

Garrison W. Greenwood, Ph.D., P.E.
Dept. of Electrical & Computer Engineering
Portland State University
Portland, OR, USA

All systems eventually fail. If the system is critical, it must be either repaired or replaced.

Unfortunately, repair and/or replacement is not always easy...

e.g., consider the mars rovers

In such cases, one solution is to make the system *fault-tolerant*—i.e., able to autonomously repair itself to restore functionality.

These fault-tolant systems must (a) detect the failure, and (b) do something to fix the problem.

We are going to use *evolvable hardware* to do the fixing.

Outline

I. Introduction to fault-tolerant systems

II. What is "evolvable hardware"?

III. Using evolvable hardware to recover from faults

IV. Overview of real-time systems

V. Recovery under real-time constraints

So, what does fault-tolerance deal with?

Suppose a system suddenly fails. Two questions must be answered:

1. Can you find the problem?
   (fault detection)

2. Can you fix the problem? (fault recovery)

Oh, by the way, do both detection and recovery without human intervention.

Notes:

- detection of faults is not always easy
- redundancy is the most common recovery method
- redundance is not always possible; then re-configuration might be a good choice
- in some cases detection/recovery have real-time requirements.

Before discussing failure recovery, we need to identify what the failures are.

But first, “faults” and “failures” are not the same thing.

def. (failure)

inability to accomplish an assigned task

def. (fault)

a defect that can lead to a failure.

e.g., a clamping diode shorts (fault) which causes an input line to be permanently grounded (failure) def. (failure mode)

A specified way in which a system or component can fail

e.g. the failures modes of a diode are "short" and "open"

How can we find these failure modes?

1. historical operational data

2. results from testing or experiments

3. technical literature (journals, reports, etc.)

Once the component failure modes are known, the system failure modes are found using

Fault Tree Analysis (FTA)

> A top-down approach where a system failure is assumed to have happened and one tries to find the fault that caused it.

Failure Modes & Effects Analysis (FMEA)

> A bottom-up approach where the effect of every failure mode of every component is determined.

So what causes faults???

1. components have limited lifespans (e.g., connectors corrode, transistors "burn out")

2. operational environment changes

An observation...

Environmental conditions include humidity, temperature, shock, vibration and radiation.

Unanticipated environmental conditions are of most concern because it makes the system operate in an unpredictable manner.

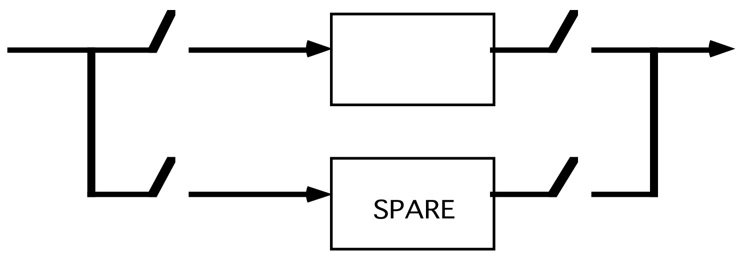
def. (fault-tolerant system (FTS))

A system that can continue to operate in the presence of failures (albeit with degraded performance)

FT = fault detection + fault recovery

FT can be achieved by

- fault masking (doesn't fix the problem, just hides it)
- fault recovery via reconfiguration (the EHW approach)
- fault recovery via redundancy (most common method)



IMPORTANT NOTES:

1. In some cases only degraded performance may be achieved after fault recovery takes place.

2. When a system fails because of non-environmental reasons, redundancy is the <u>only</u> recovery method guaranteed to restore full functionality.

3. If redundancy not possible, reconfiguration may be the only viable recovery method

EHW = EA + reconfigurable circuitry

- FPGA, FPAA and FPTA are reconfigurable devices

- "EA" means ES, GA, etc.

$$P(t+1) = \mathcal{S}(\mathcal{E}(\mathcal{V}(P(t))))$$

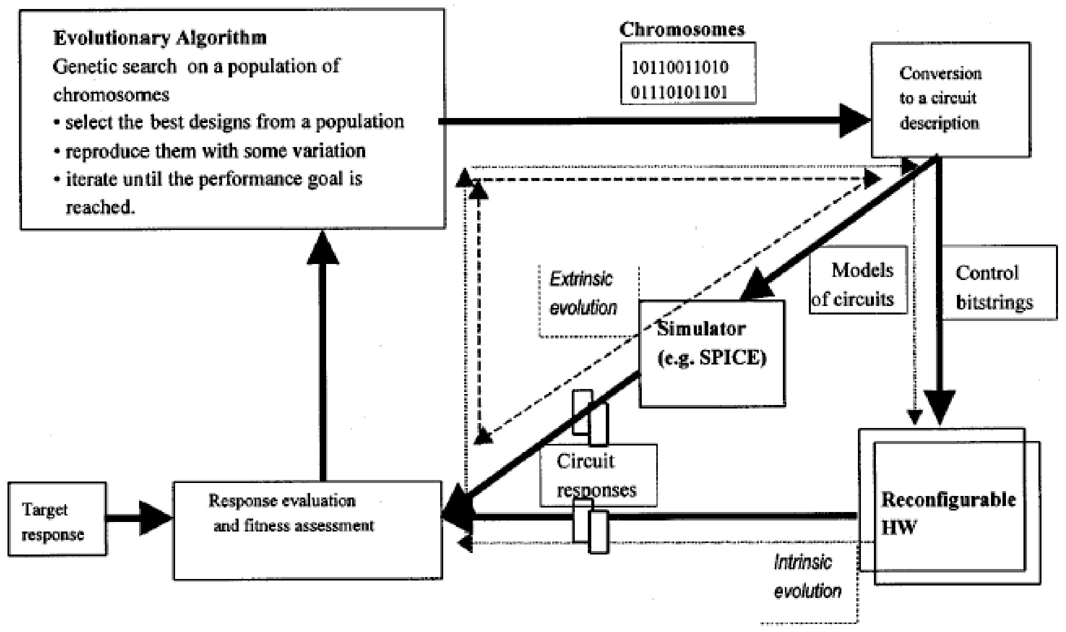
$P(t)$	population of solutions at time t
$\mathcal{V}(\cdot)$	random variation operator
$\mathcal{E}(\cdot)$	evaluation operator
$\mathcal{S}(\cdot)$	selection operator

- each "solution" is a hardware configuration

- evaluation can be done in two ways:

  1. in hardware (intrinsic)

  2. in software (extrinsic)


Evolutionary Algorithm
Genetic search on a population of chromosomes
• select the best designs from a population
• reproduce them with some variation
• iterate until the performance goal is reached.
Chromosomes
10110011010
01110101101
Conversion to a circuit description
Models of circuits
Control bitstrings
Extrinsic evolution
Simulator (e.g. SPICE)
Circuit responses
Reconfigurable HW
Intrinsic evolution
Target response
Response evaluation and fitness assessment

Main steps for the evolutionary synthesis of electronic circuits.

From A. Stoica et al., “Reconfigurable VLSI Architectures for Evolvable Hardware: From Experimental Field Programmable Transistor Arrays to Evolution-Oriented Chips”, *IEEE Trans. on VLSI Sys.*, Vol 9(1), 2001

Field Programmable Gate Array (FPGA)

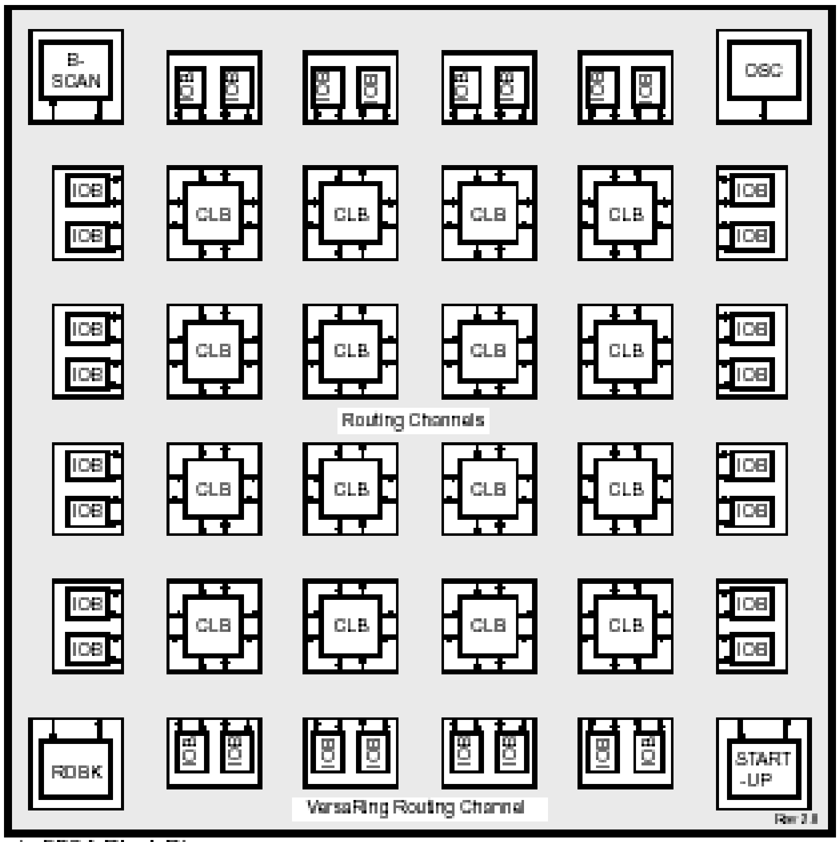


Figure 1: Basic FPGA Block Diagram

from Xilinx Spartan family datasheet, Version 1.4, Jan 1999

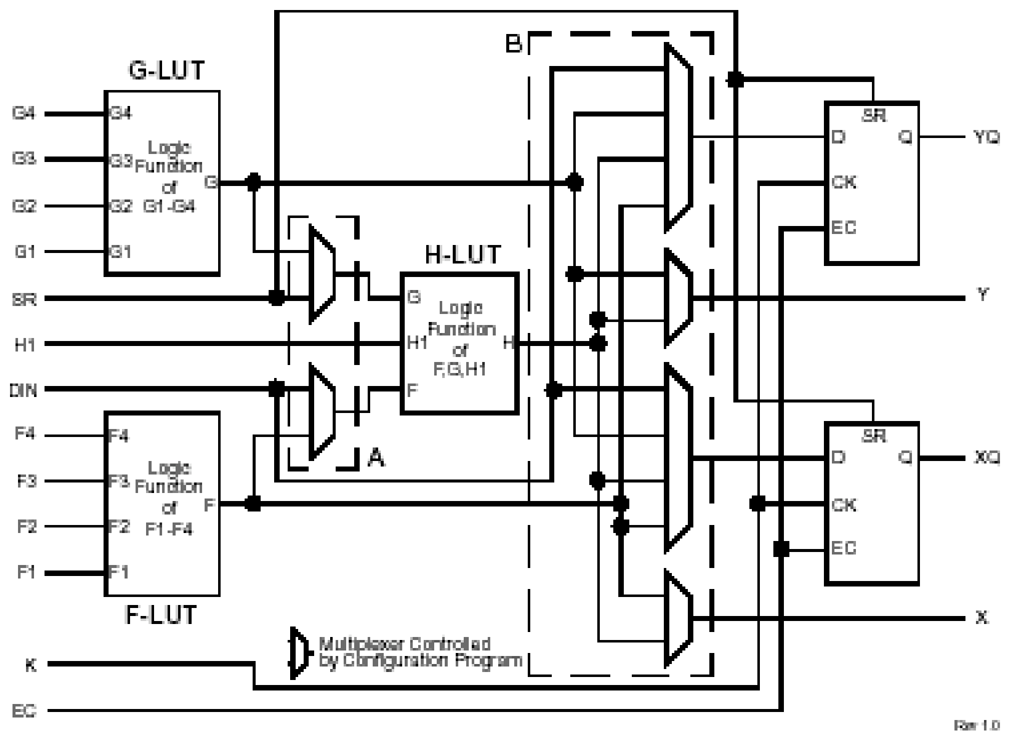
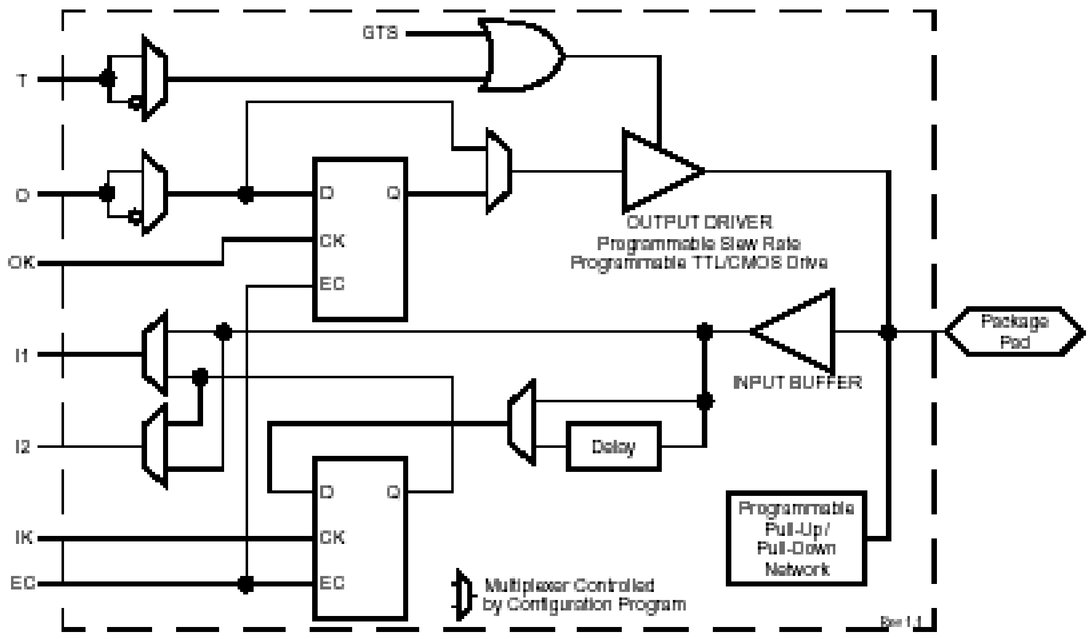


Figure 2: Spartan Simplified CLB Logic Diagram (some features not shown)



GTS
T
O
OK
I1
I2
IK
EC
D
Q
CK
EC
OUTPUT DRIVER
Programmable Slew Rate
Programmable TTL/CMOS Drive
INPUT BUFFER
Package Pad
Delay
Programmable Pull-Up/ Pull-Down Network
Multiplexer Controlled by Configuration Program
Rev 1.1

Figure 5: Simplified Spartan IOB Block Diagram

figures from Xilinx Spartan family datasheet, Version 1.4, Jan 1999

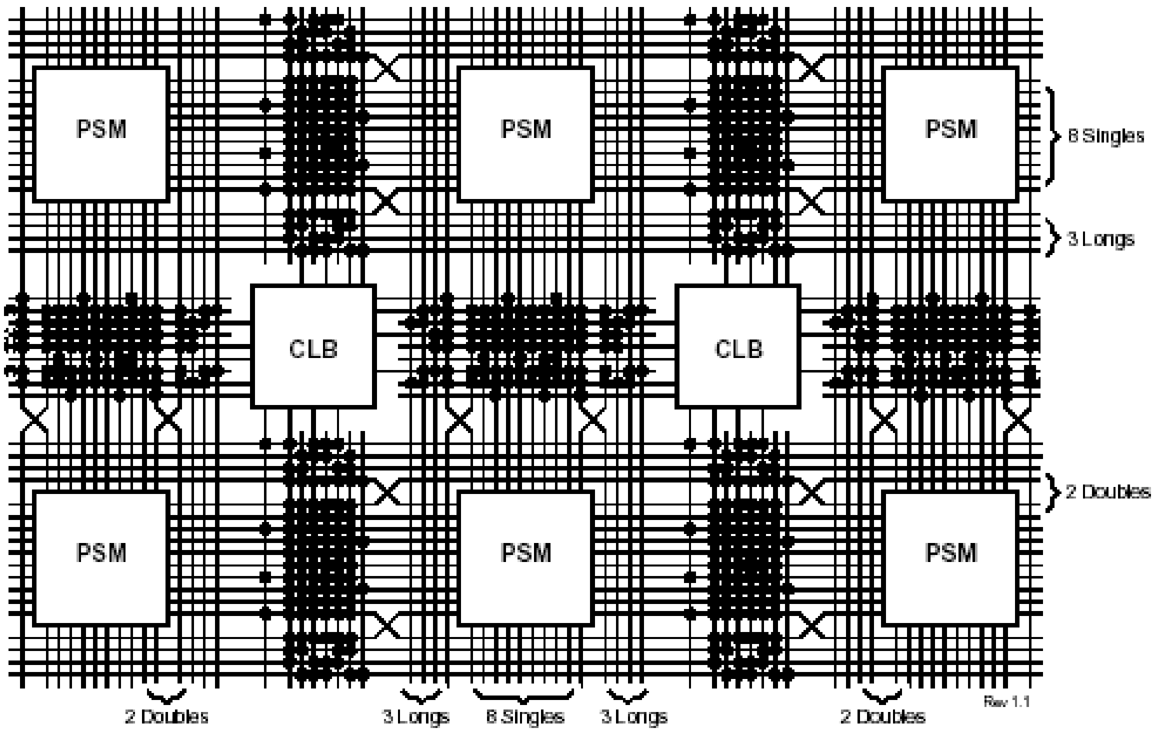


Figure 8: Spartan Series CLB Routing Channels and Interface Block Diagram

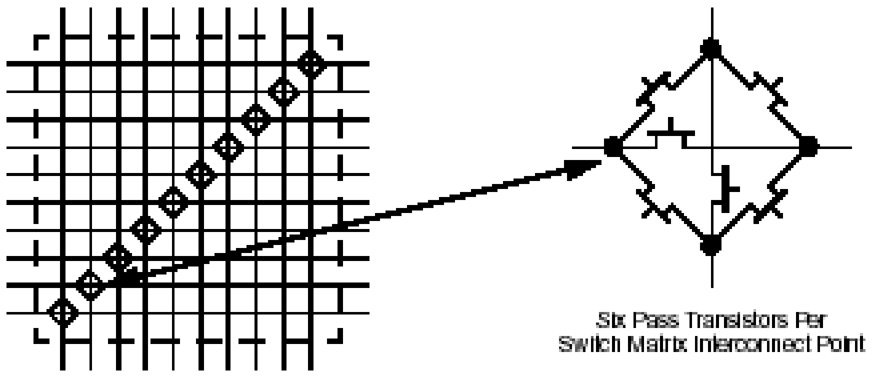


Figure 10: Programmable Switch Matrix

figures from Xilinx Spartan family datasheet, Version 1.4, Jan 1999

So, how do you program these FPGAs with an EA?

A good example is the use of "Jbits" used to program the Xilinx Virtex FPGA family.

Jbits are a set of Java classes that provide an interface into the Virtex FPGA configuration bitstream. (The bitstream can come from a Xilinx design tool or readback from the actual device.)

Each configurable logic block (CLB) is the FPGA has coordinates (i, j) and each CLB configuration is located in a specific region of the datastream.

The EA creates a new configuration and then uses the Jbit interface to modify the bitstream accordingly. The modified bitstream is reloaded into the Virtex FPGA.

G3	G2	G1	G0	OR	AND
0	0	0	0	0	0
0	0	0	1	1	0
0	0	1	0	1	0
0	0	1	1	1	0
0	1	0	0	1	0
0	1	0	1	1	0
0	1	1	0	1	0
0	1	1	1	1	0
1	0	0	0	1	0
1	0	0	1	1	0
1	0	1	0	1	0
1	0	1	1	1	0
1	1	0	0	1	0
1	1	0	1	1	0
1	1	1	0	1	0
1	1	1	1	1	1

To change a LUT implementation of an OR gate to an AND gate, the bitstream is uploaded, the bits of the LUT are located, and

OR 1111 1111 1111 1110   0xfffe

is changed to

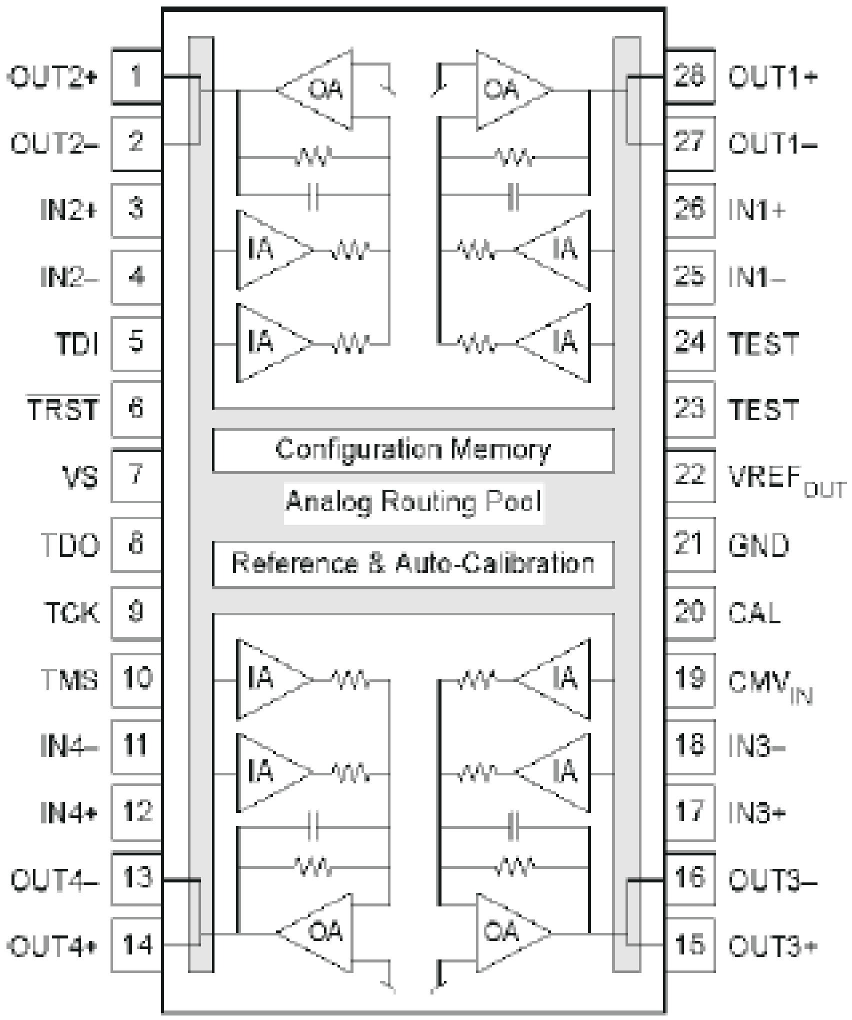
AND 1000 0000 0000 0000   0x8000

```
/* Attempt to connect to the hardware */
result = board.connect(remoteHostName,
                       port);

/* Get the type of devices used by the
** first FPGA on the board */
deviceType = board.getDeviceType(0);

/* Read in the bit file */
jBits.read(infileName);

/* Set the Top left CLB (1,1) to an AND gate
** i.e. 1000 0000 0000 0000 */
jBits.set(1, 1, LUT.SLICE0_G, 0x8000);

/* Get the now updated Bitstream */
bs = jBits.getAllPackets();

/* Finally send the new bitstream to the
** Virtex Chip */
result = board.setConfiguration(0,bs);

/* And cleanup... */
board.disconnect();
```

Figure 2. Source code for JBits example

from G. Hollingworth et al., “Safe Intrinsic Evolution of Virtex Devices”, *Proc. 2000 NASA/DOD Evolvable Hardware Conf., 2000*

The analog counterpart to the FPGA is the *field programmable analog array* (FPAA)



Lattice Semiconductor ispPAC10

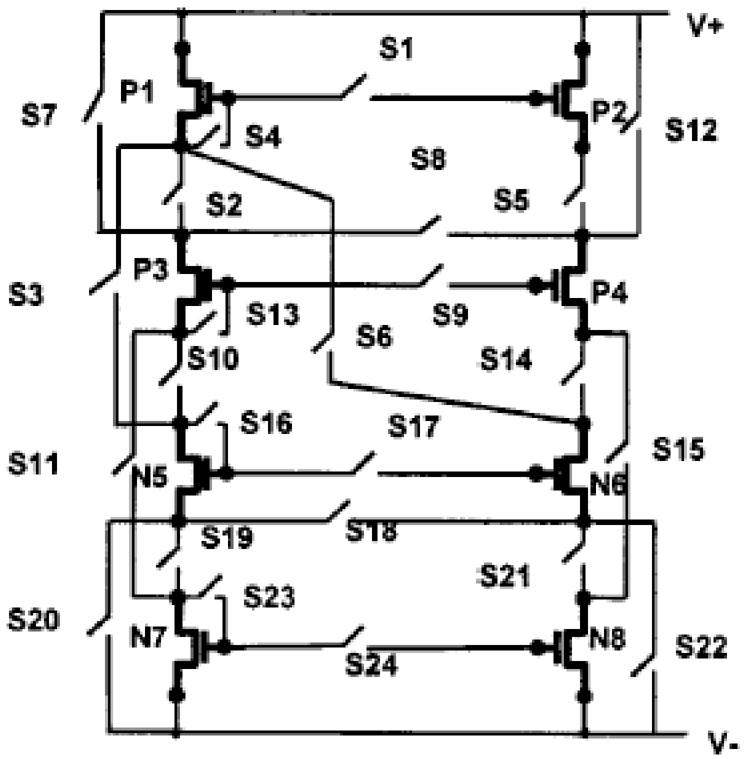
The user can program

- input amplifiers gain (by choosing one of 8 resistor values plus polarity)
- output amplifier bandwidth (by choosing 1 of 128 capacitor values)
- intrablock and interblock interconnections

The datastream for the ispPAC10 is $\approx$ 320 bits long. The EA manipulates the binary string directly.

The lowest granularity device is the 0.5μm CMOS *field programmable transistor array* (FPTA). This is a prototype chip developed for NASA's JPL.

The FPTA is organized as a 2D array of cells



. Schematic of an FPTA cell consisting of 8 transistors and 24 switches.

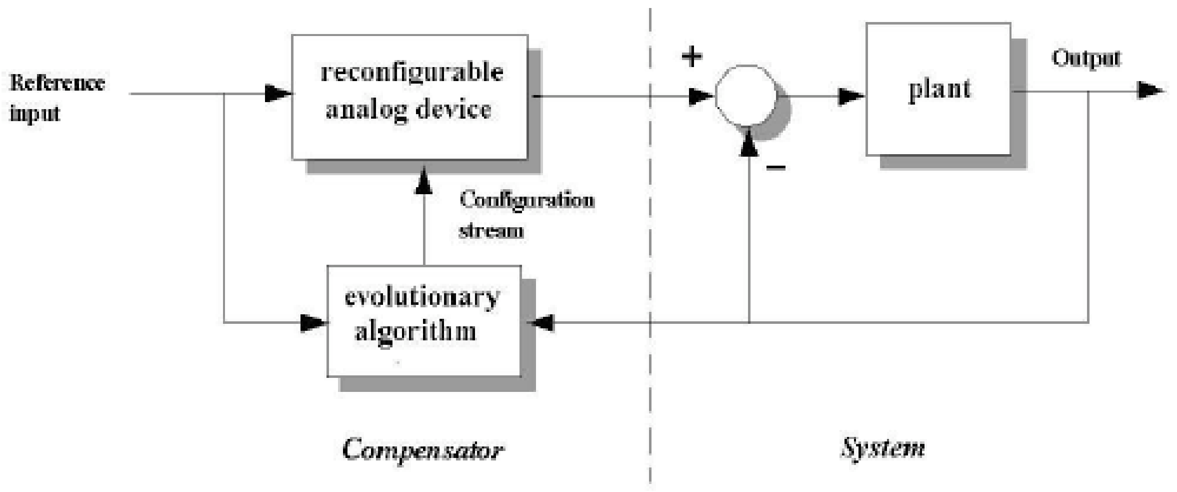
Cells can be combined to form current mirrors, logic gates and op amps.

From A. Stoica et al., "Reconfigurable VLSI Architectures for Evolvable Hardware: From Experimental Field Programmable Transistor Arrays to Evolution-Oriented Chips", *IEEE Trans. on VLSI Sys.*, Vol 9(1), 2001

To illustrate how EHW can be used for fault recovery, we will consider an analog FTS. We assume

1. the system is linear with dynamics governed by constant coefficient ordinary differential equations
2. the system is a "black box"—i.e., no access to inside to repair or replace failed components
3. evolution is done intrinsically

Note: black box $\Rightarrow$ compensation is the only means of handling faults

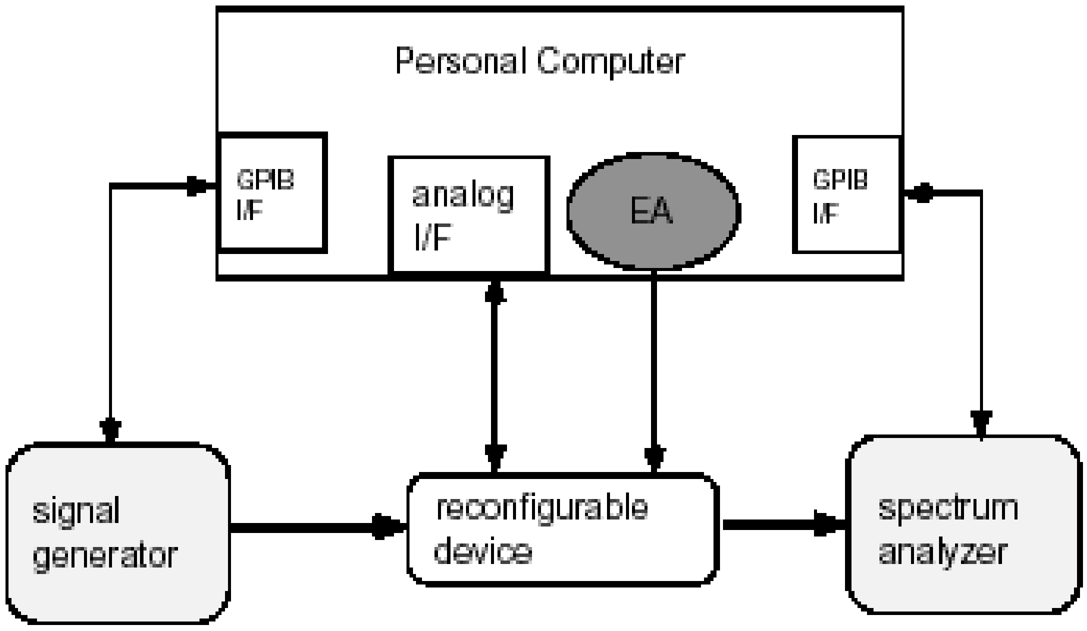


Fault-tolerant analog system

The plant is 3rd order. The fault is manifested by a change in bandwidth; fault recovery should restore the bandwidth.

The reconfigurable device is the ispPAC10, which implements a lead or lag compensator.

The EA evolved a population of 20 for 200 generations using recombination and mutation.



The intrinsic EHW testbench

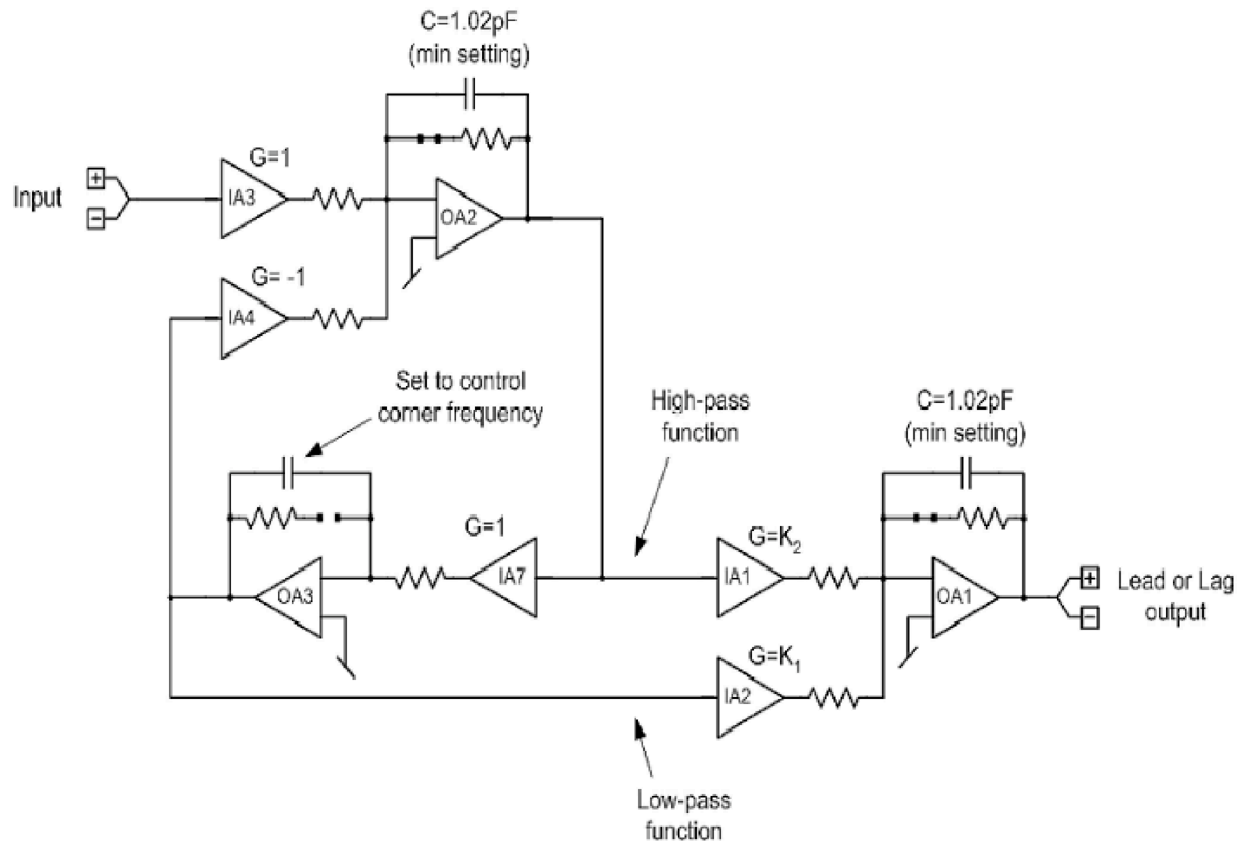
The fitness of configuration C is given by

$$\text{fitness}(C) = \sum_{i=1}^{5} [M(i) - M^*(i)]^2$$

where $M(i)$ is the compensated system's magnitude and $M^*(i)$ the desired magnitude at frequency i.

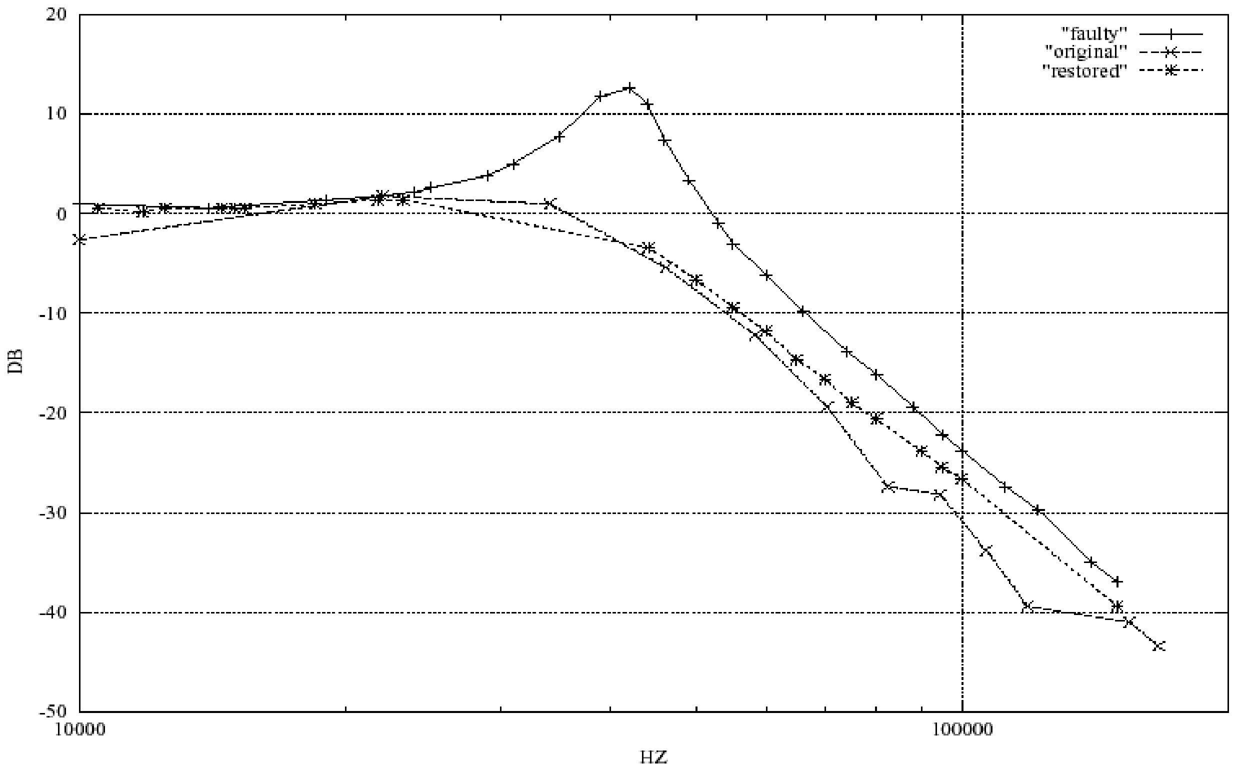
The 5 test frequences where chosen such that some were in the passband, one at the -3db point, and some in the stopband.

Note: just using the -3db point would produce the trivial solution of increasing or decreasing the open-loop gain.



The EA had to evolve the capacitor value and the two amplifier gain values (K_1 and K_2).

We also had to evolve two switch positions because the only way to get K_1 or K_2 equal to 0 was to remove the amplifiers IA_1 and IA_2.


"faulty"
"original"
"restored"
DB
HZ
20
10
0
-10
-20
-30
-40
-50
10000
100000

Overview of Real-Time Systems

def. (real-time systems)

A real-time system (RTS) is any system that is both logically and temporally correct.

def. (logically correct)

satisfies all functions specifications

def. (temporally correct)

completes all tasks within specified timeframes

Fast does not mean real-time

and

Real-time does not mean fast

For example, consider two real-time delivery systems: one is a courier who guarantees delivery in less than 3 days and an email system that guarantees delivery in 10 minutes.

Notice both systems guarantee delivery (logically correct), but one is orders of magnitude faster than the other.

Scenario #1: need delivery in 5 days
(both are RTS)

Scenario #2: need delivery in 5 minutes
(neither are RTS)

So, why is RT an issue in fault-tolerance?

ANS: because faults cannot be left uncorrected indefinitely.

This impacts intrinsic EHW used for fault recovery because reconfiguration takes time.

In fact, in some cases intrinsic reconfiguration may not be practical!!!

With intrinsic reconfiguration every solution must be downloaded into the reconfigurable time, which takes time t_p.

Device	**Type**	**Size**	t_p (ms)
ispPAC10	FPAA	4	100
AN220E04	FPAA	4	3.8
XC3020A	FPGA	64	1.5
Virtex XCV50	FPGA	1728	7
XC4085XL	FPGA	3136	192
APEX II EP2A70	FPGA	6720	12.5
JPL's FPTA2	FPTA	64	0.008

Let λ be the number of new configurations created each generation

Let t_f the time to conduct a fitness test.

Then an EA running for k generations has an intrinsic reconfiguration time of

$$T_r(k,\lambda) = k\lambda(t_p + t_f)$$

But the real problem is t_f and especially for analog systems.

Example:

An AN220E04 FPAA is used to compensate for aging effects in a control system responsible for positioning a satellite's communications antenna.

The reconfiguration search is done by a generational GA run for 500 generations with a population size of 100.

The system's step response is measured to determine if the compensation is correct. This step response test takes $t_f = 625$ milliseconds to conduct.

Hence, $\lambda = 100$, $k = 500$ and $t_p = 3.8$ ms, which makes $T_r(500, 100) \approx 8.7$ hours.

Is 8.7 hours too long?

Ans: maybe

Reconfiguration times are meaningless unless they are put into context.

For instance, suppose the above control system is in a communications sattelite. It only operates for a few minutes every 20 hours. However, failure to operate can lead to the loss of the sattelite.

If a control system fault occurs just after one of these operational periods, no problem.

If the operational period starts in 10 minutes, big problem.

This means you can only determine if a reconfiguration time is too long by comparing it against the fault recovery deadline.

Since fault recovery has a deadline, it meets the definition of a RTS.

This has <u>strong</u> implications for the EHW community.

It is no longer sufficient to just talk about how an EA was able to restore a circuit's functionality. (only shows logical correctness)

Just reporting an algorithm's running time doesn't say anything about temporal correctness either.

The key point is expressed by the following first principle:

> *No EHW-based recovery method can legitimately proclaim efficacy until it is proven to be both logically and temporally correct.*

Logical correctness is easy to prove. (Try it out and see if it works!)

Temporal correctness is proven by comparing the EA running time against the fault recovery deadline (defined when FTA or FMEA was conducted).

If both are satisfied, your EHW recovery method is viable.

Final comments:

- Extrinsic reconfiguration is often impractical as a fault recovery method because

  1. The precise nature of the failure may not be known. Hence, any simulation is likely to be inaccurate.

  2. It may take too long.

     For instance, consider trying to extrinsically reconfigure hardware in a deep space probe operating near the plant Neptune. Communication with the probe will take hours.

- Genetic programming should be avoided for any fault recovery methods. It simply requires far too much computational effort and is unlikely to meet any realistic fault recovery deadline.