Digital synthesis for rad-hard components

Single Event Upsets mitigation techniques with TMRG tool

cern.ch/tmrg

Ecole de Microélectronique IN2P3
14 - 19 May 2017, Bénodet

Szymon KULIS, EP-ESE-ME, CERN

Agenda

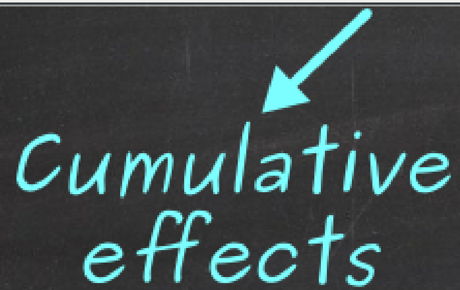
TMRG

- Radiation Effects
  - Single-event effects
  - Mitigation Techniques
- Triple Modular Redundancy Generator
  - Design flow
  - Triplicating the design (tmrg)
  - Physical implementation (plag)
  - Verification (tbg, seeg)
- Tools tips & tricks
- Summary

Many thanks to: Paulo Moreira, Pedro Leitao, Davide Ceresa, Alessandro Caratelli, Krzysztof Świentek, Xavi Llopart Cudie, Tuomas Poikela, Cesar Marin Tobon

Radiation effects

TMRG



Ionization (TID)

Displacement (fluence)

Radiation effects

TMRG

Cumulative effects

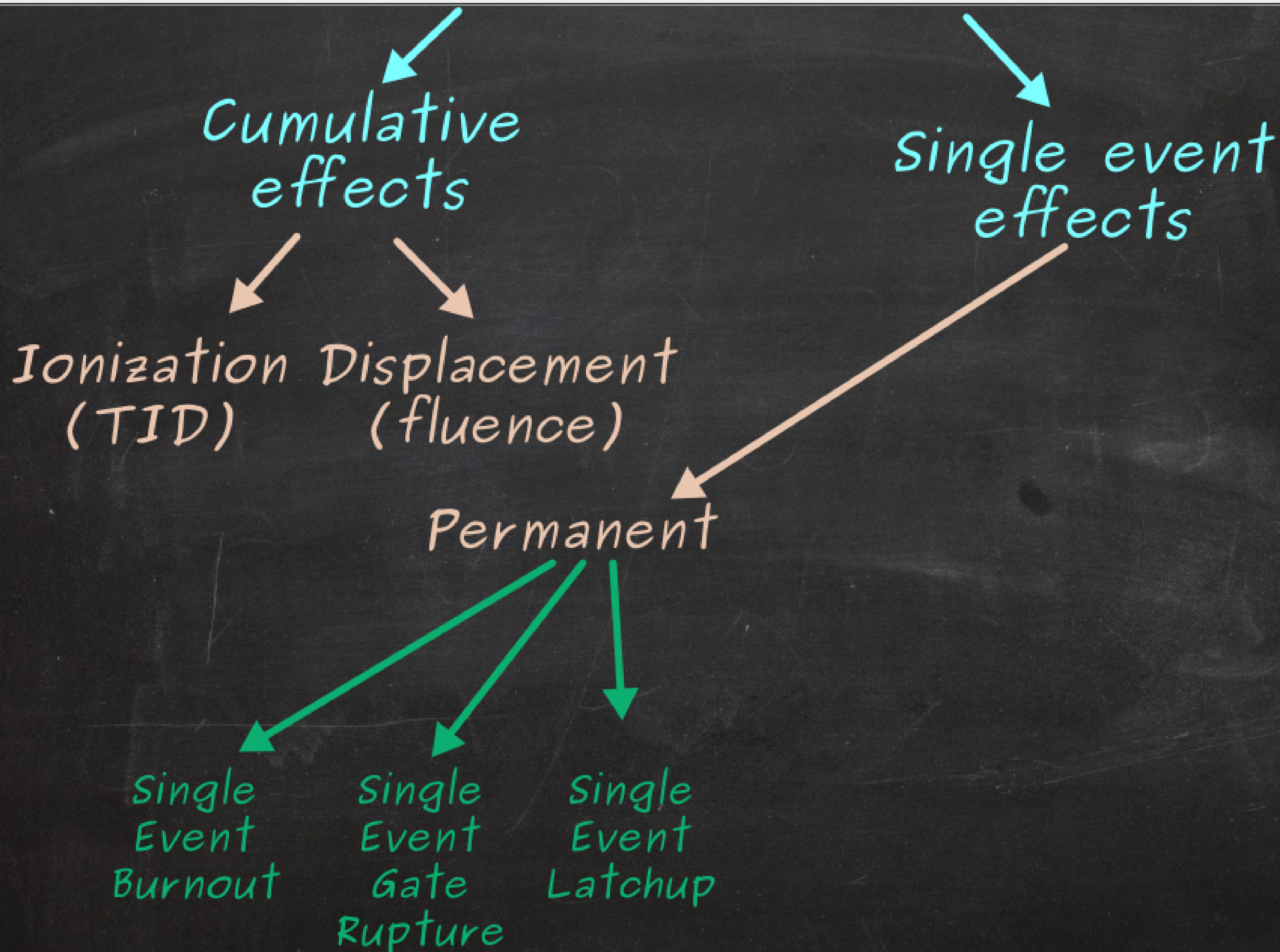
Single event effects

Ionization (TID)

Displacement (fluence)

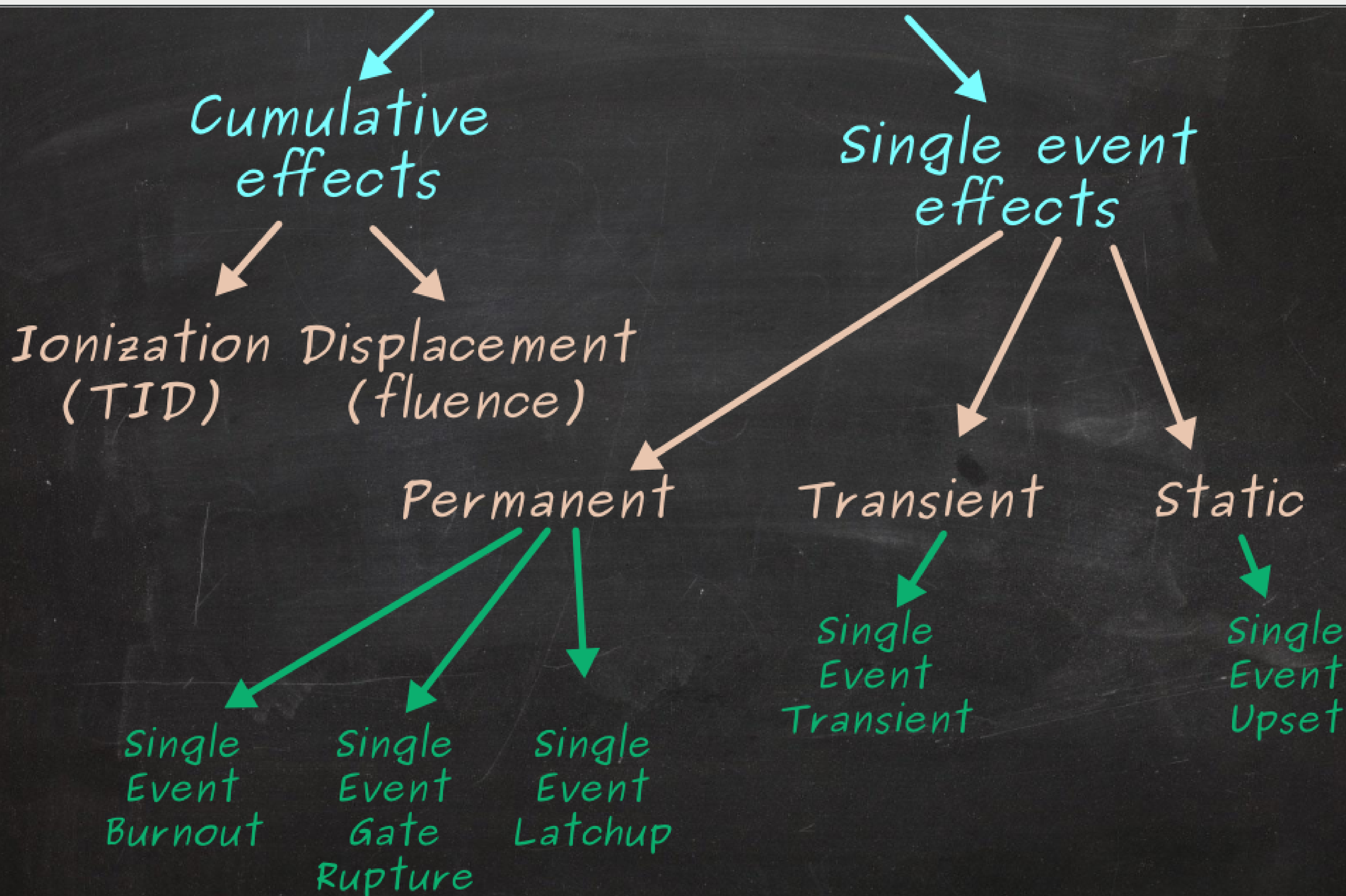
Radiation effects

TMRG

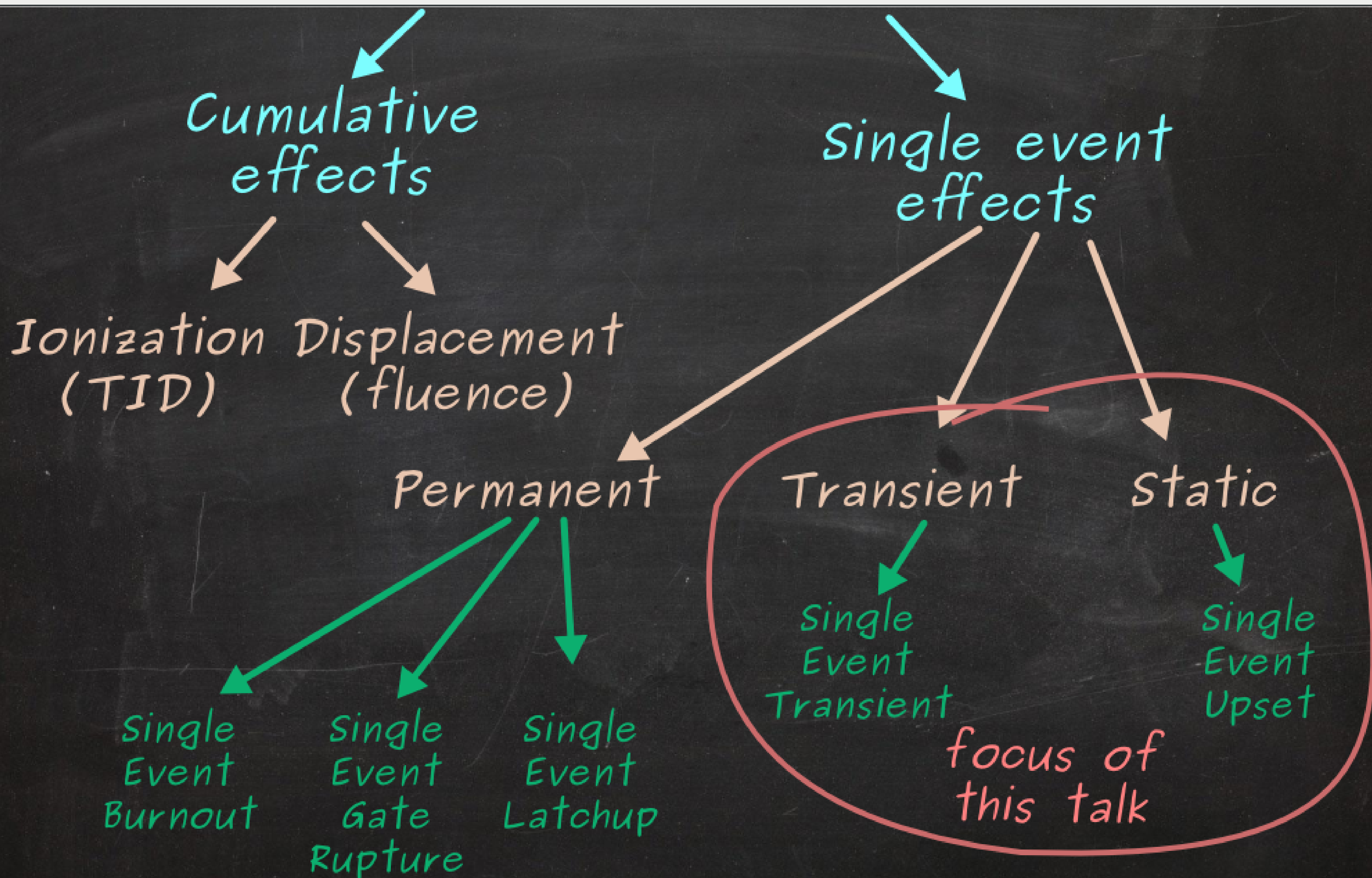


Radiation effects

TMRG

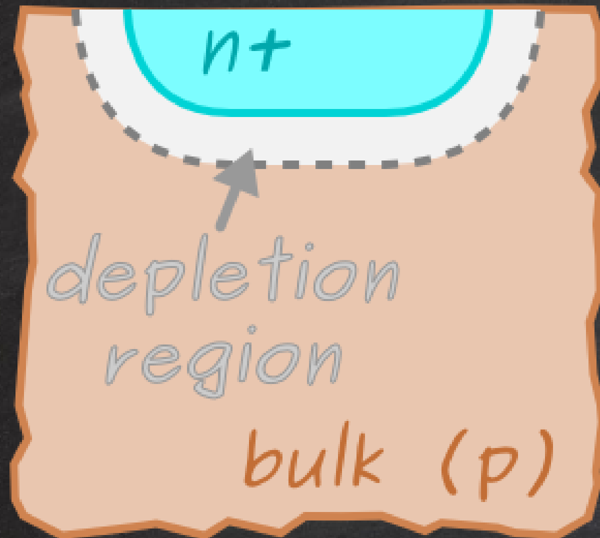


Radiation effects

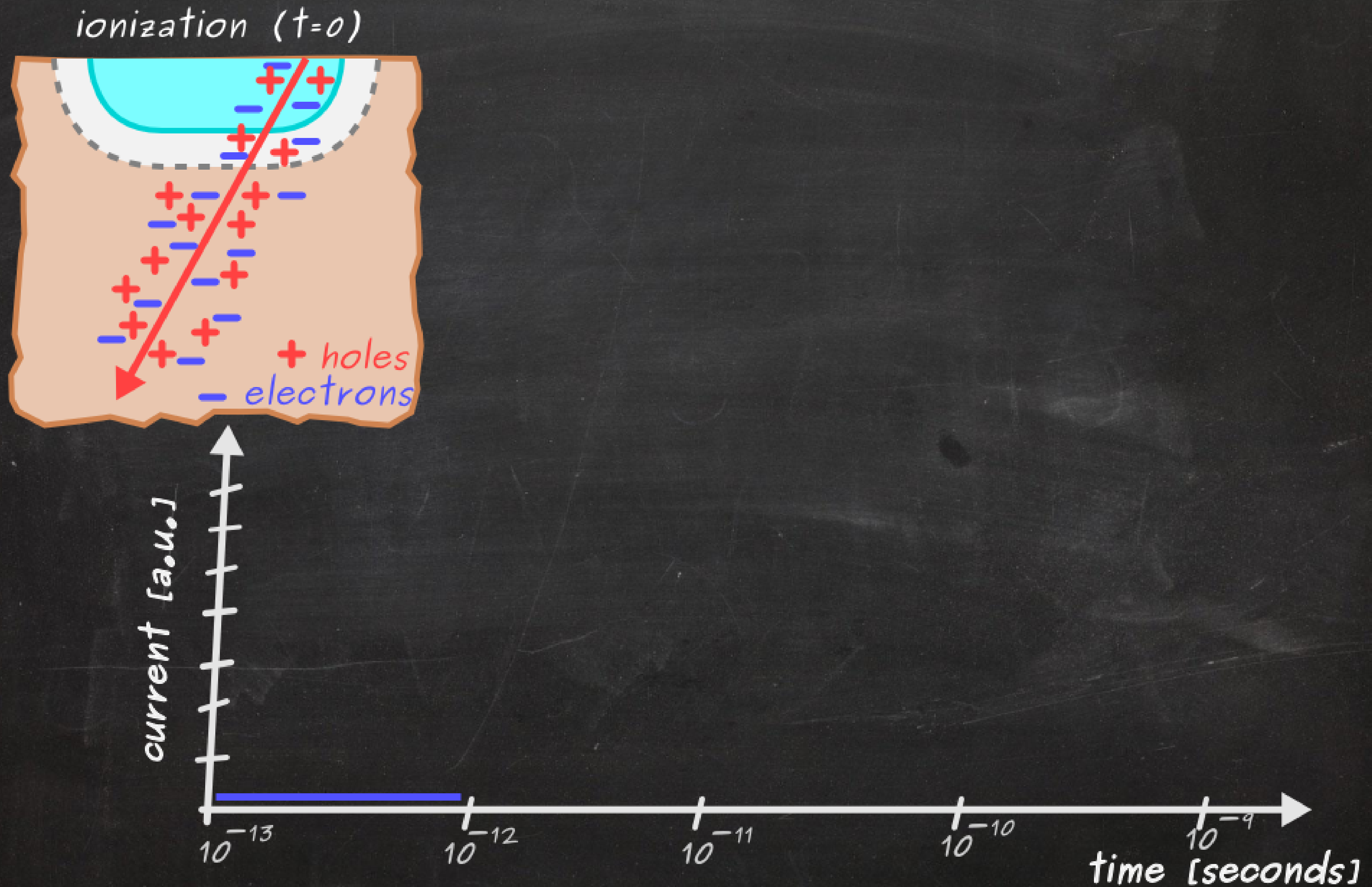


Single Event Effects

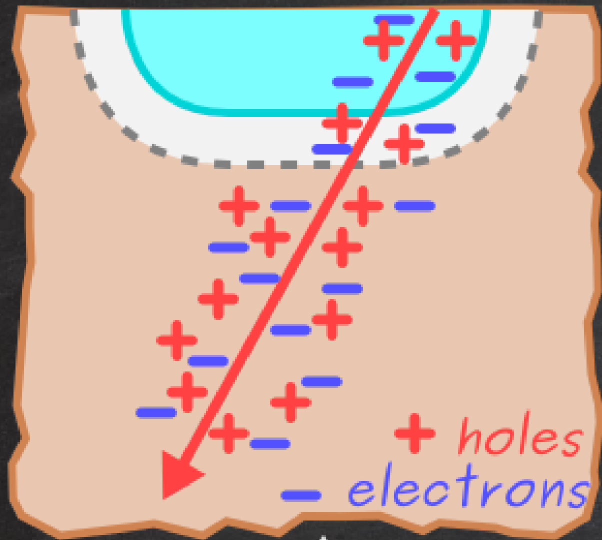
TMRG

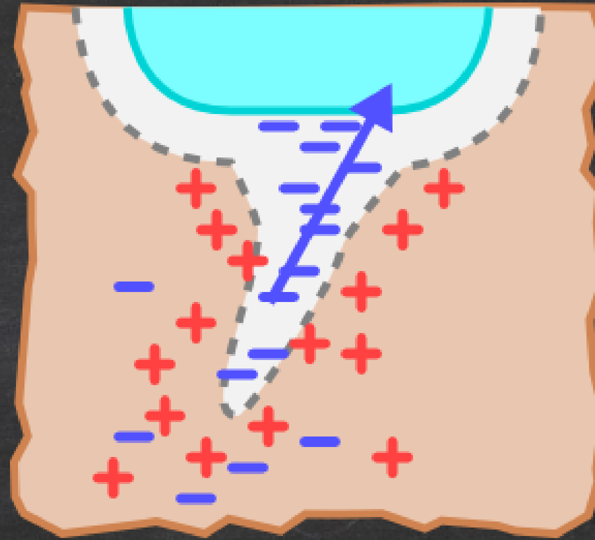


Single Event Effects



Single Event Effects





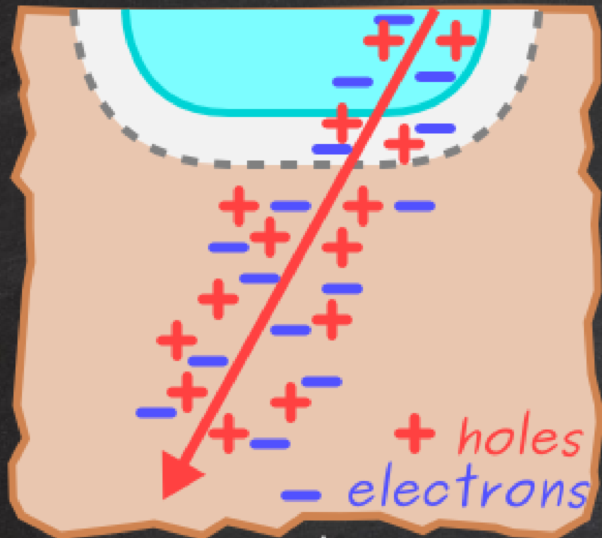
current [a.u.]

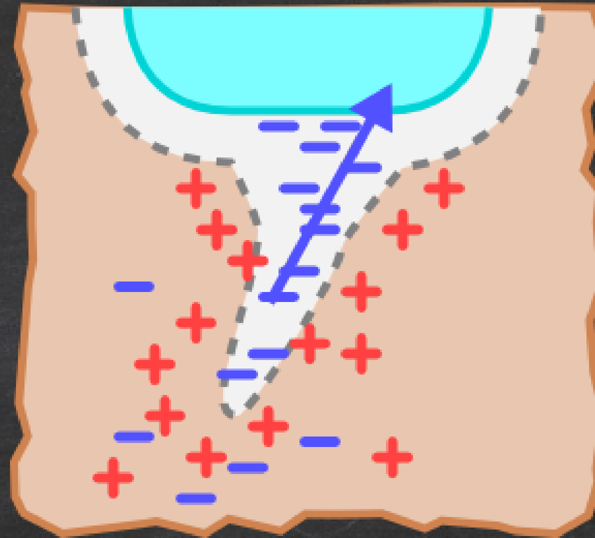
10^{-13} 10^{-12} 10^{-11} 10^{-10} 10^{-9}

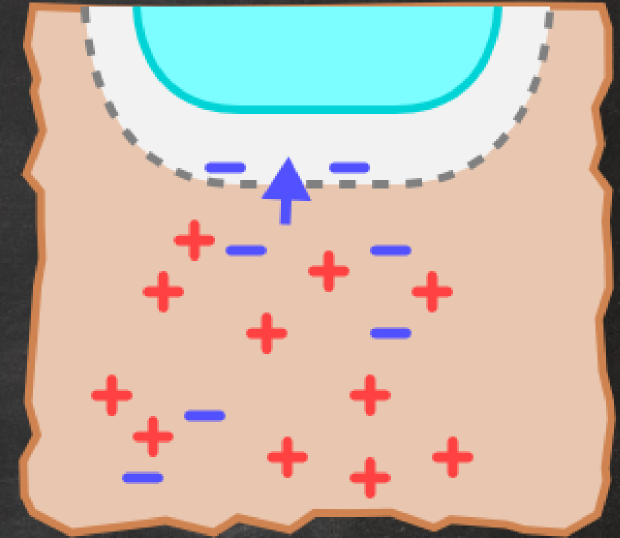
time [seconds]

Charge collection has a prompt (drift)

Single Event Effects







current [a.u.]

10^{-13} 10^{-12} 10^{-11} 10^{-10} 10^{-9}

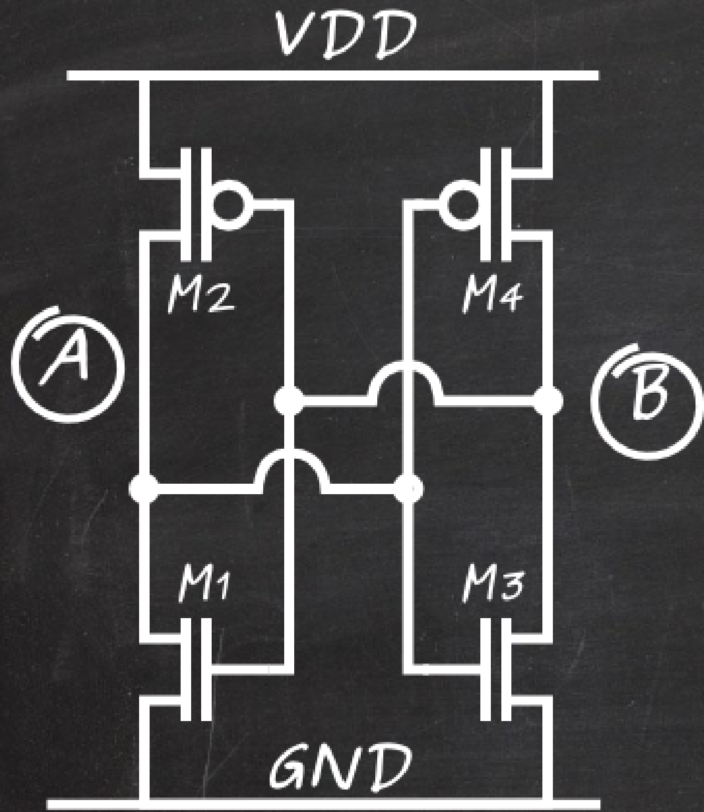
time [seconds]

Charge collection has
a prompt (drift) and
a slow (diffusion) components

Single Event Effects – example

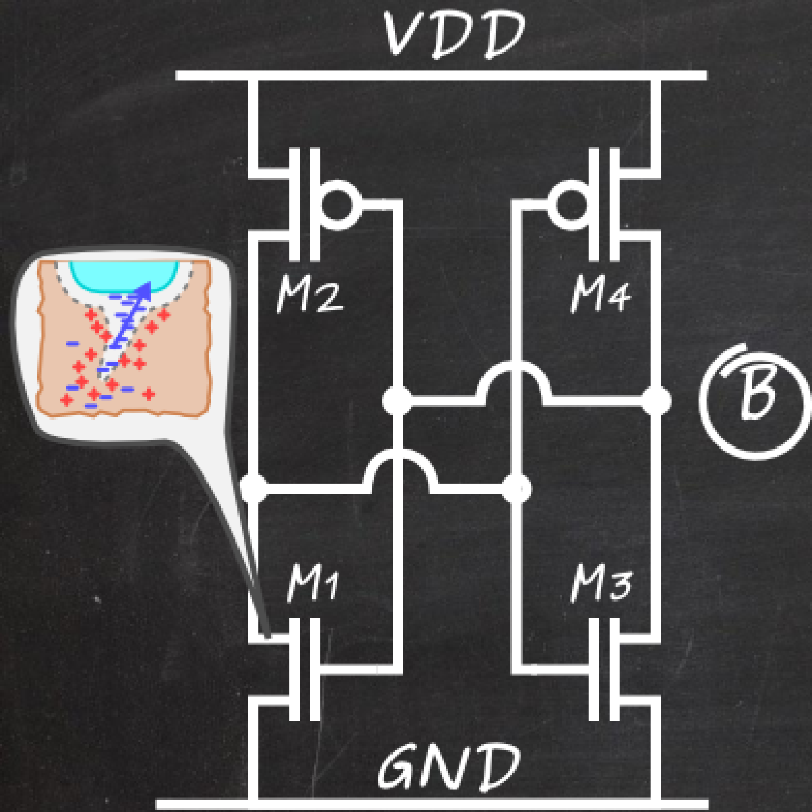
TMRG

1) initial state : A=VDD, B=GND



SRAM cell
(pass gates missing)

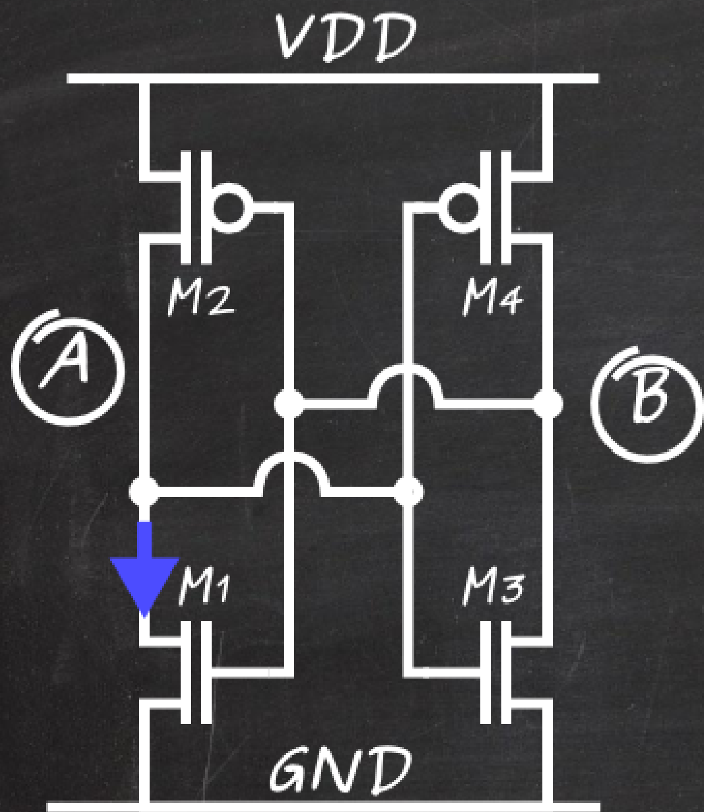
Single Event Effects – example



SRAM cell
(pass gates missing)

1) initial state : A=VDD, B=GND
2) charge deposited at drain of M1

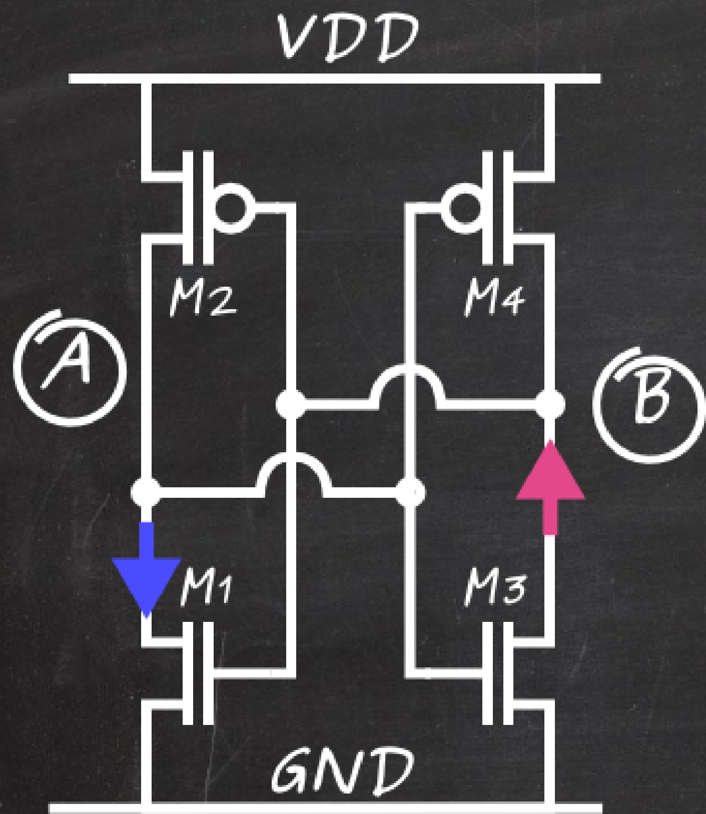
Single Event Effects – example



SRAM cell
(pass gates missing)

1) initial state : A=VDD, B=GND
2) charge deposited at drain of M1
3) transient current changes temporary the state of node A (VDD-> GND)

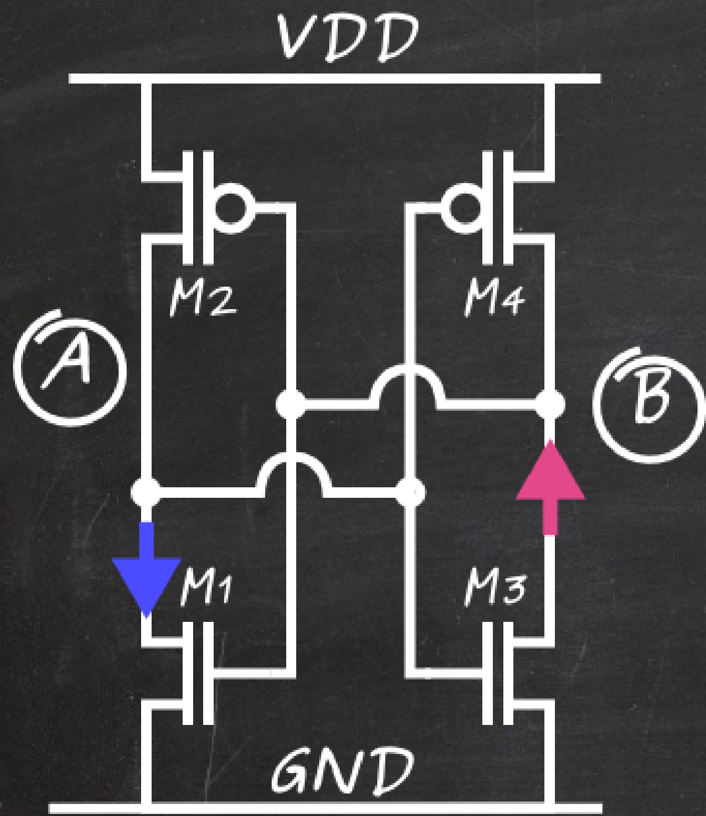
Single Event Effects – example



SRAM cell
(pass gates missing)

1) initial state : A=VDD, B=GND
2) charge deposited at drain of M1
3) transient current changes temporary the state of node A (VDD-> GND)
4) before the desposited charge is evacuated, the second inverter (M3–M4) switches (node B GND->VDD)

Single Event Effects – example

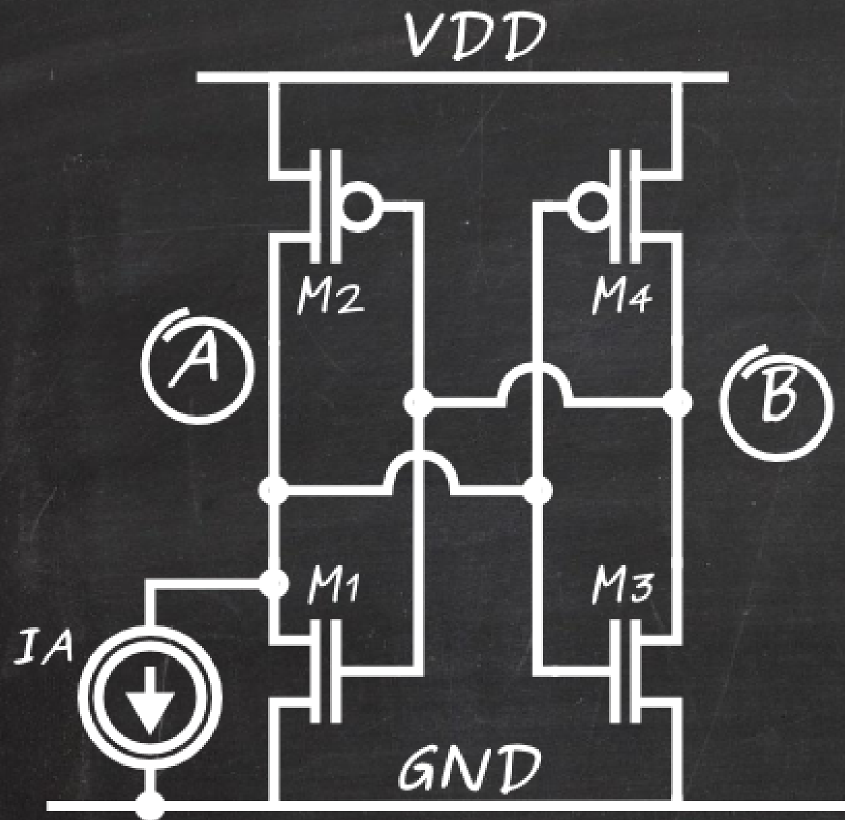


SRAM cell
(pass gates missing)

1) initial state : A=VDD, B=GND
2) charge deposited at drain of M1
3) transient current changes temporary the state of node A (VDD–> GND)
4) before the desposited charge is evacuated, the second inverter (M3–M4) switches (node B GND–>VDD)
5) The change of node B enforces the wrong state at node A –> the error is latched into the memory cell

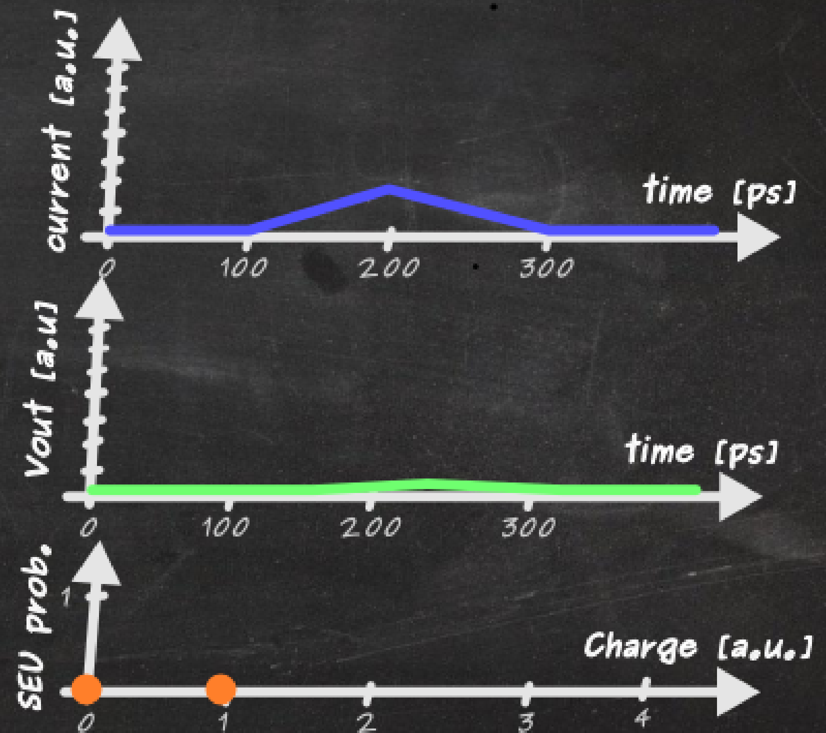
How much charge is needed to flip the value?

Critical charge

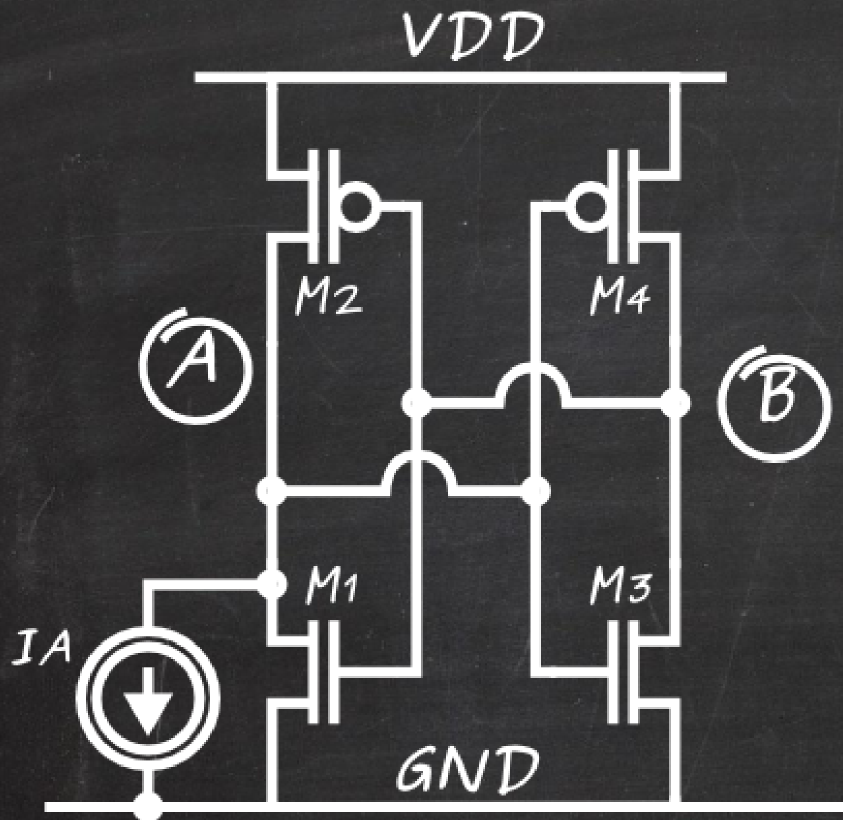


SRAM cell
(pass gates missing)

SEE can be represented by current spikes (IA) in SPICE simulation. (tringular shape is a good starting point)

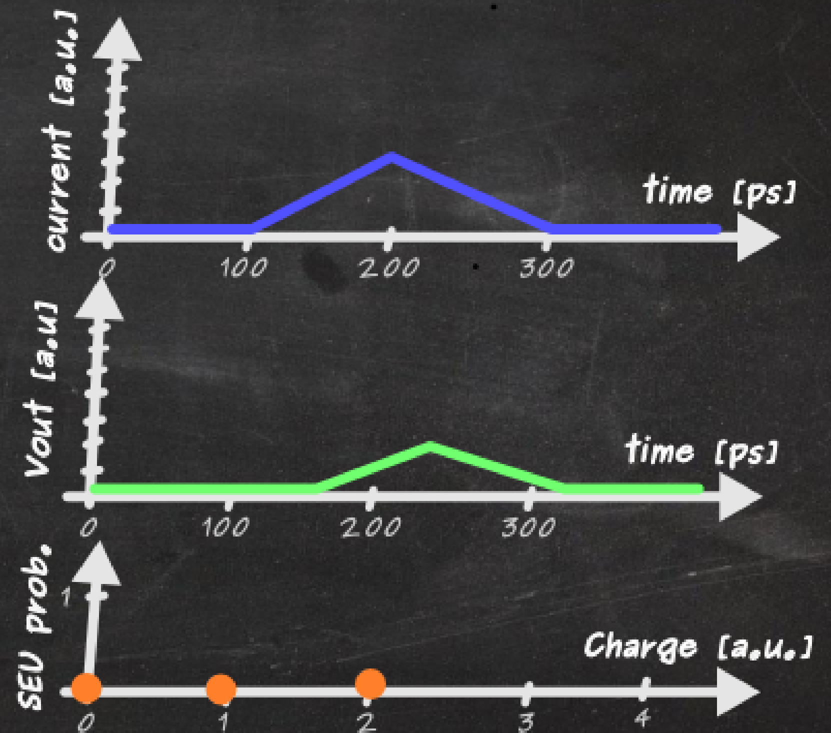


Critical charge

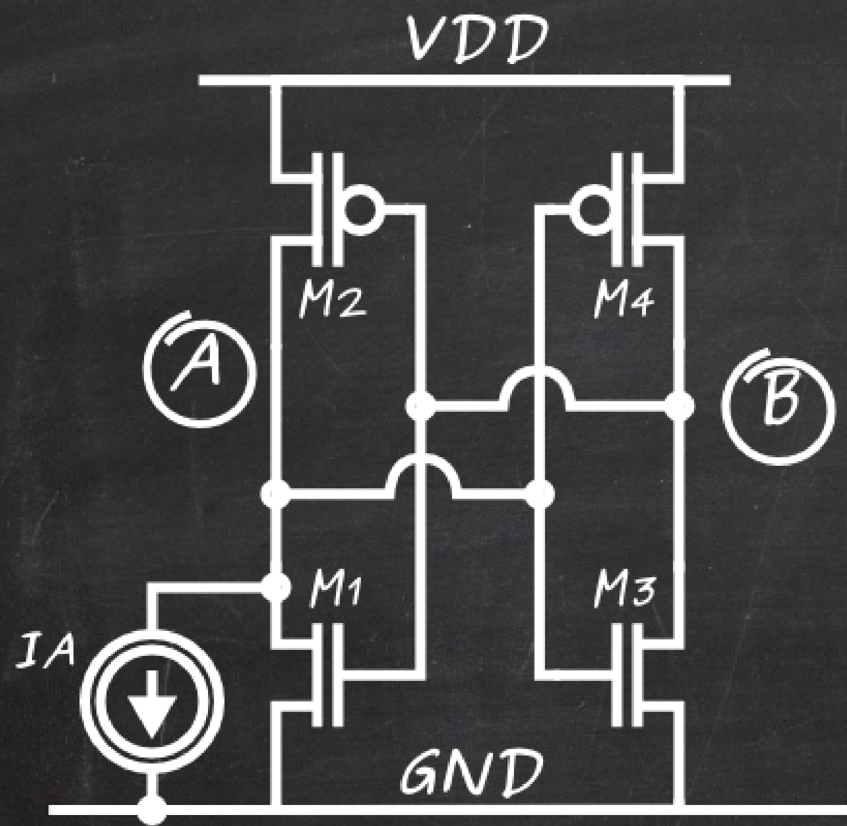


SRAM cell
(pass gates missing)

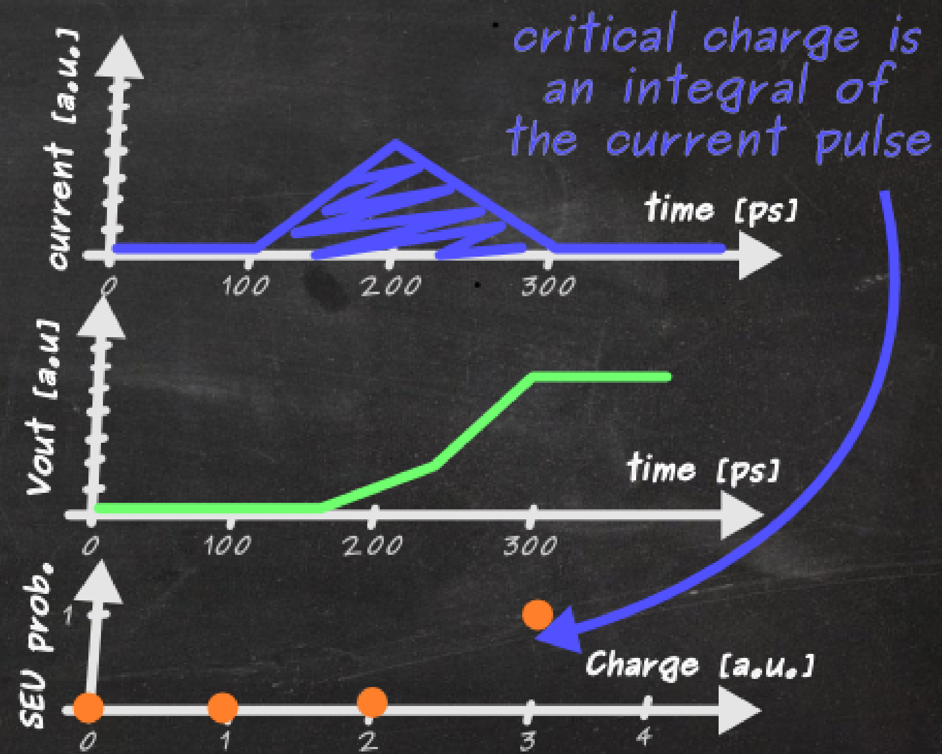
Amplitude of the current is increased until the upset is observed in simulation



Critical charge

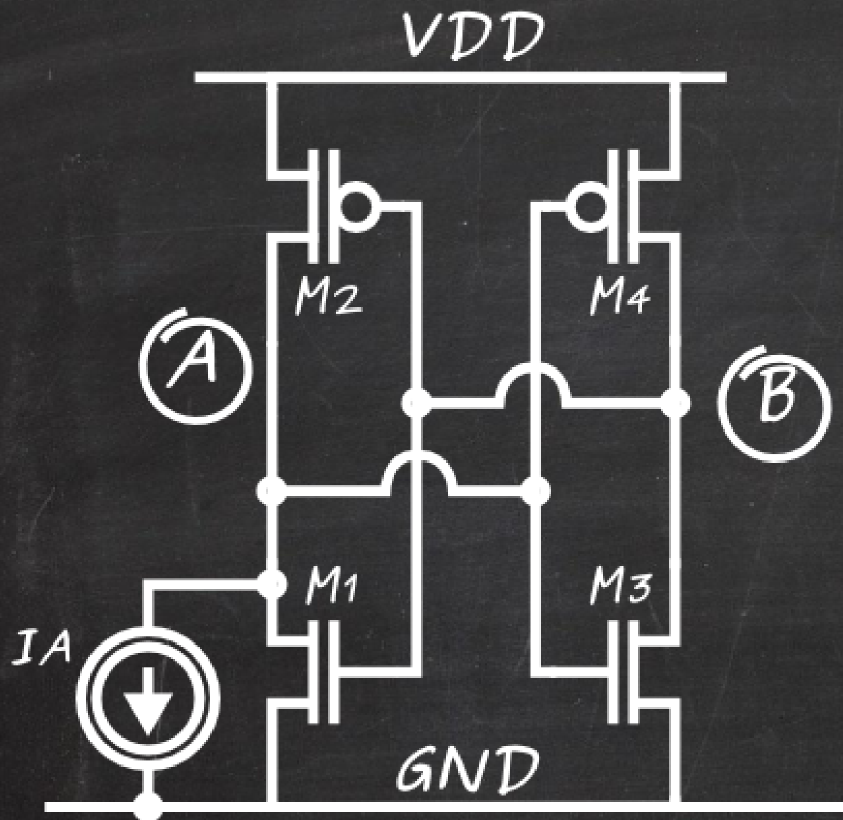


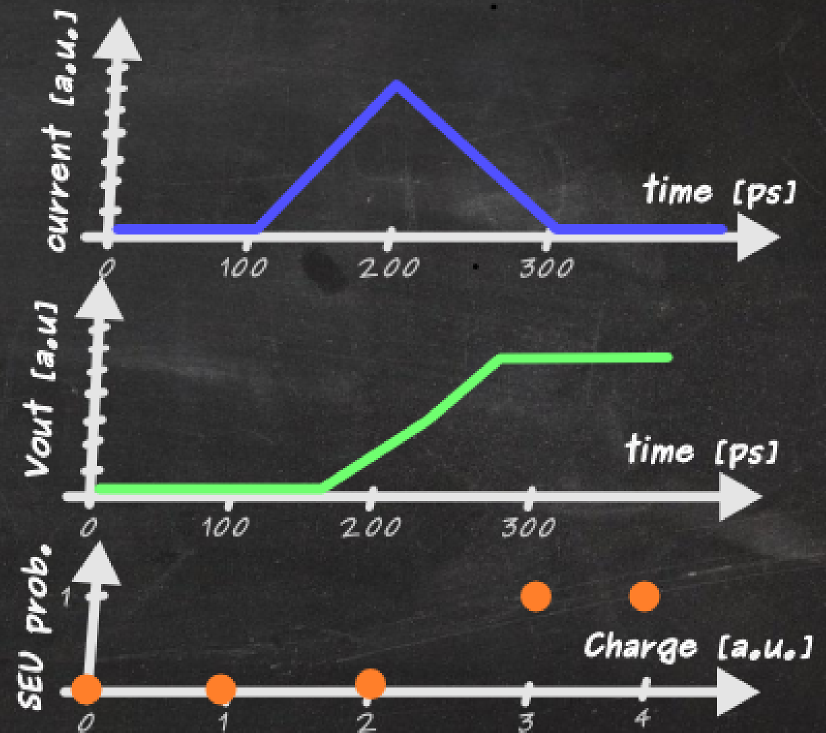
SRAM cell
(pass gates missing)



Critical charge

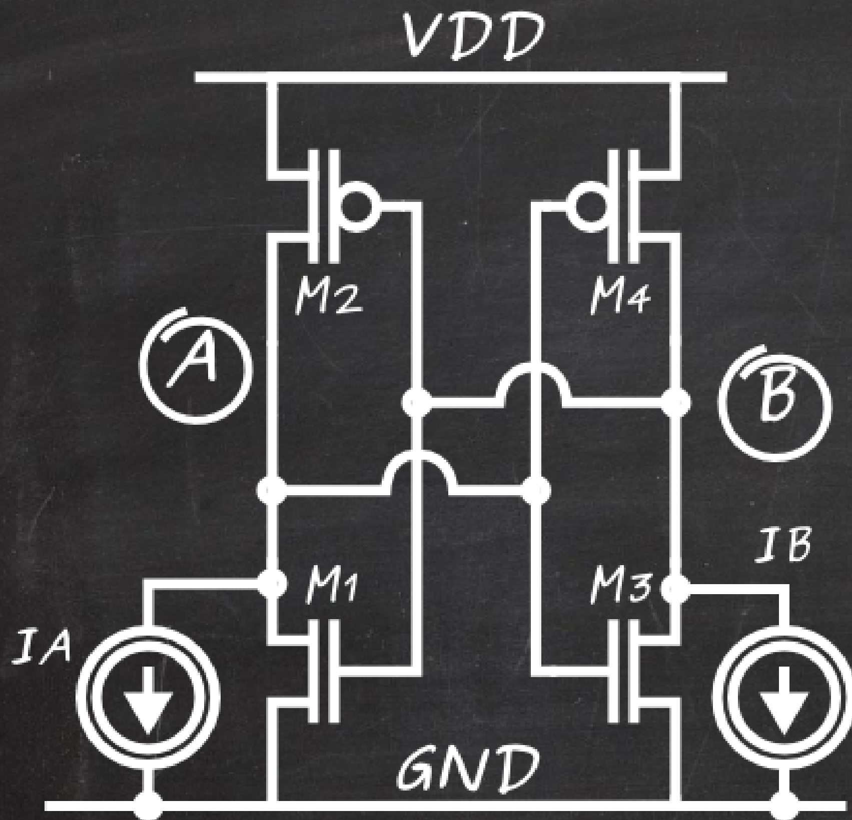
TMRG





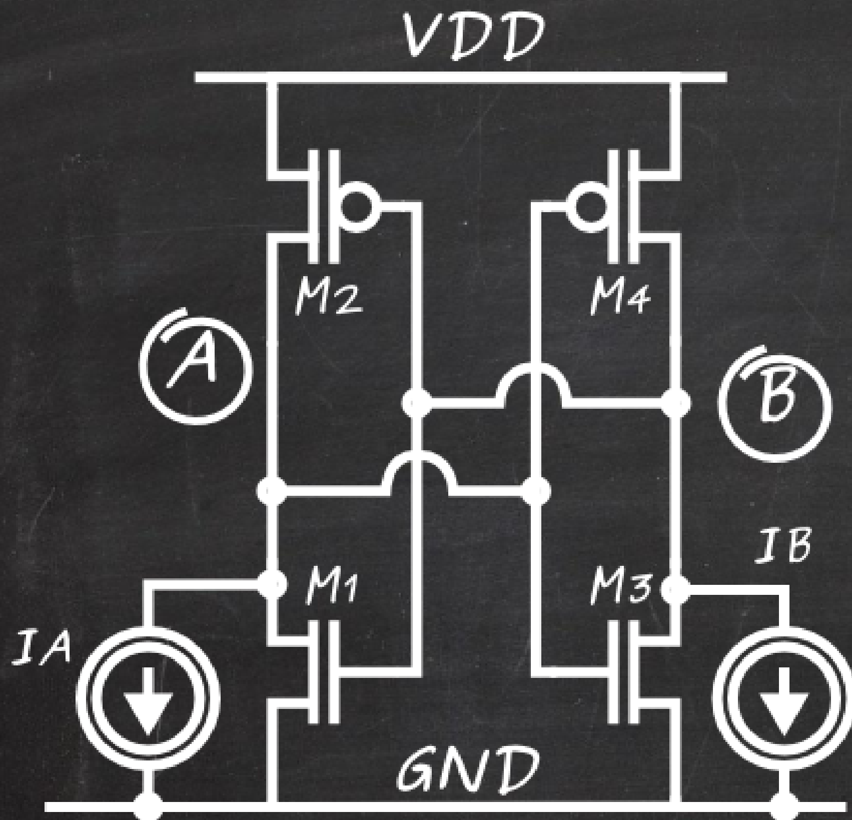
SRAM cell
(pass gates missing)

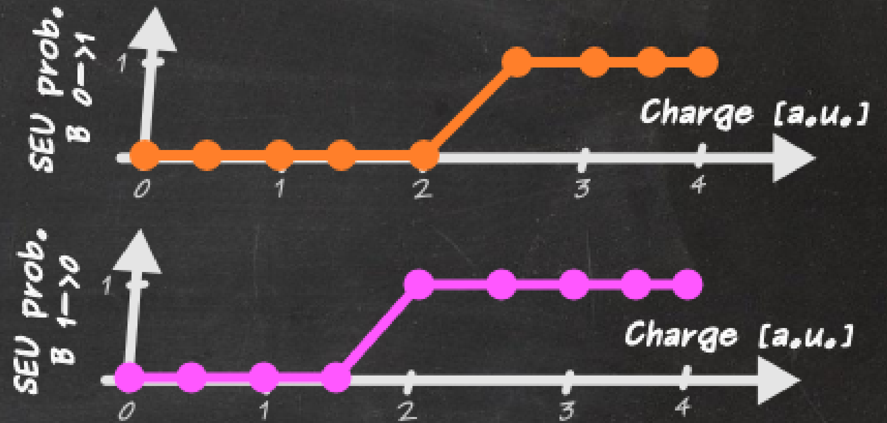
Critical charge



SRAM cell
(pass gates missing)

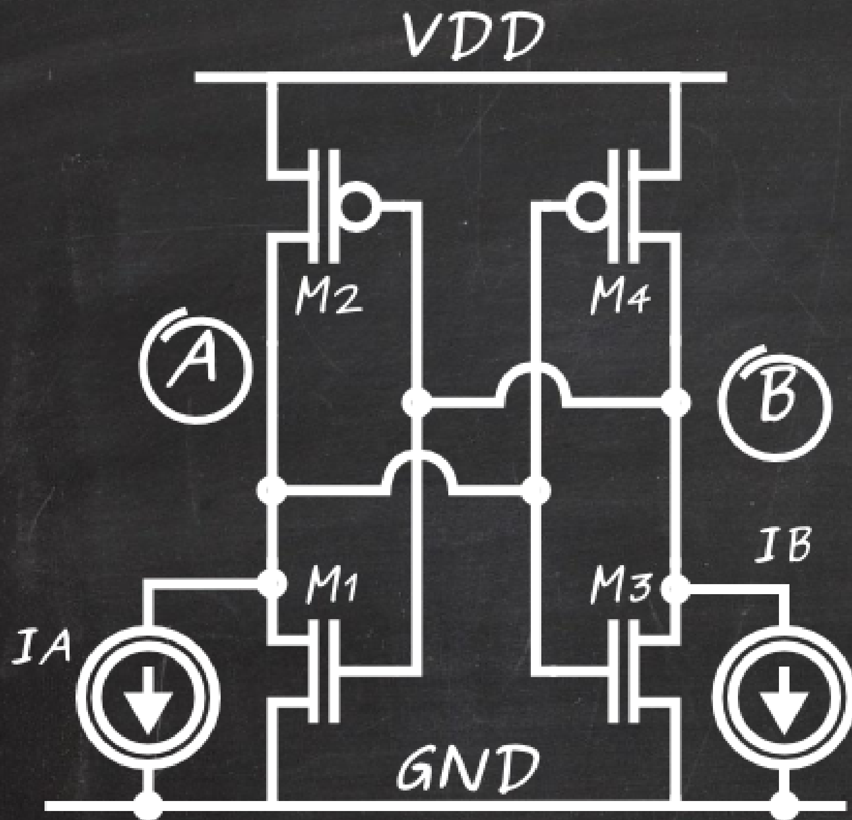
Critical charge

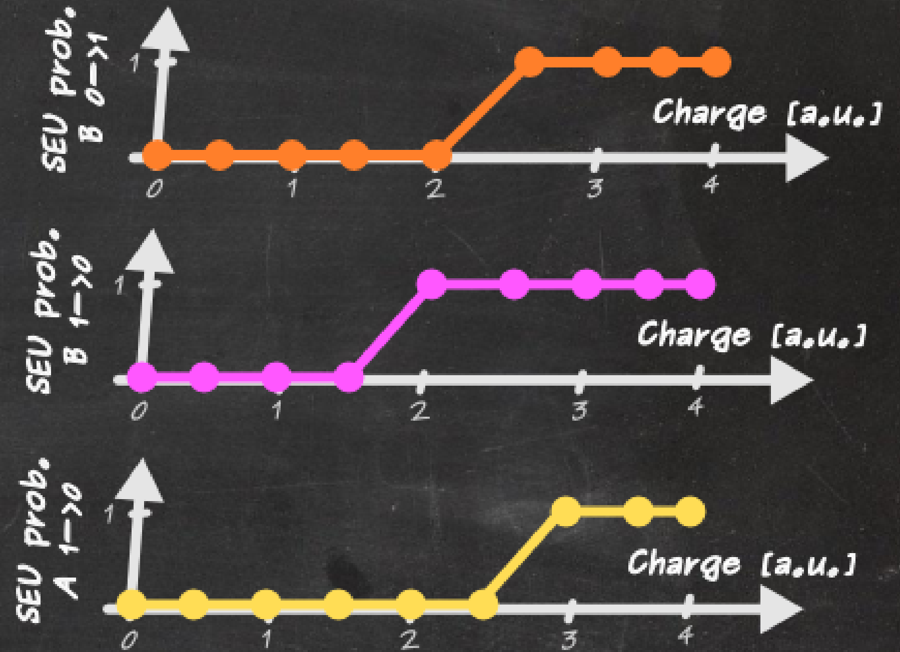




SRAM cell
(pass gates missing)

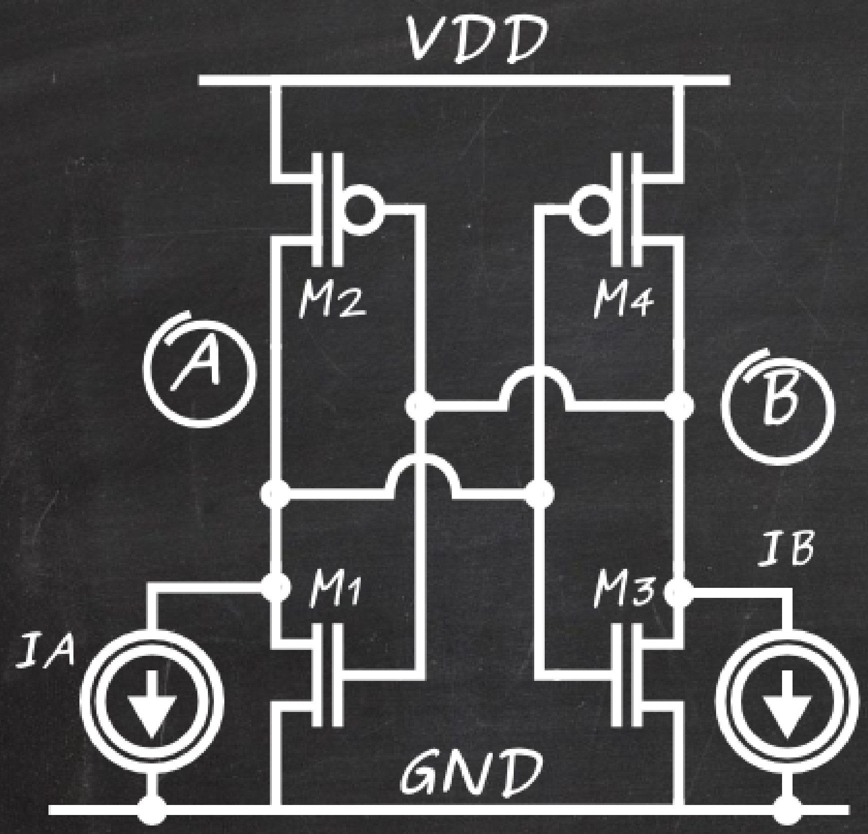
Critical charge

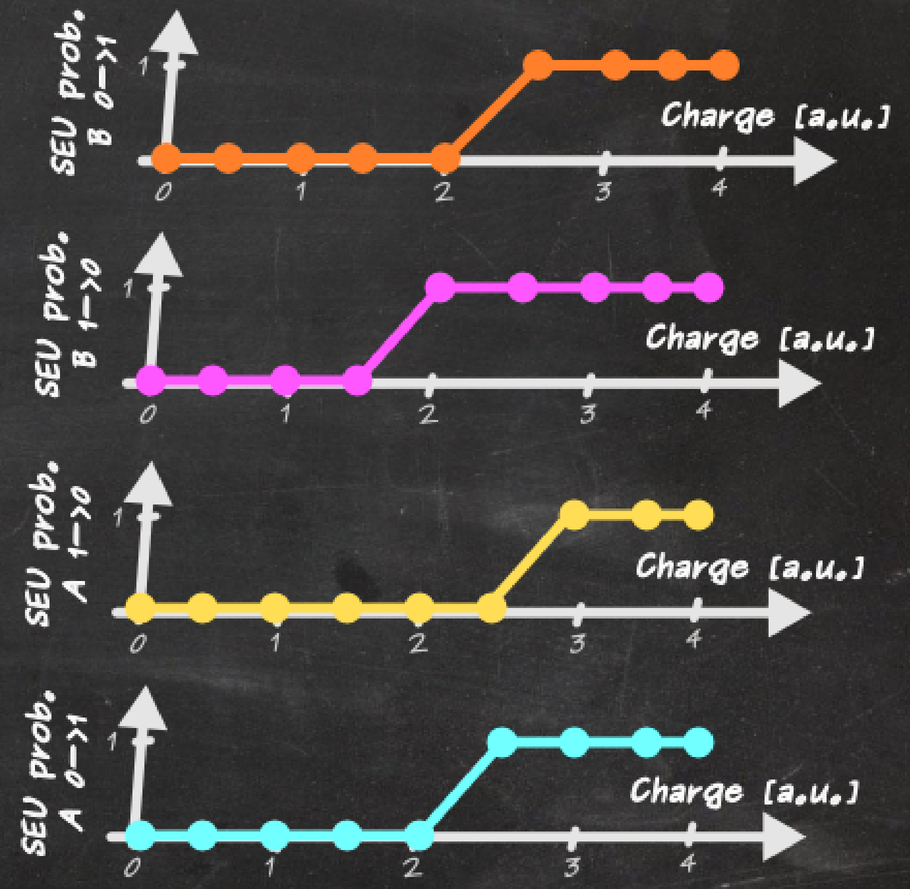




SRAM cell
(pass gates missing)

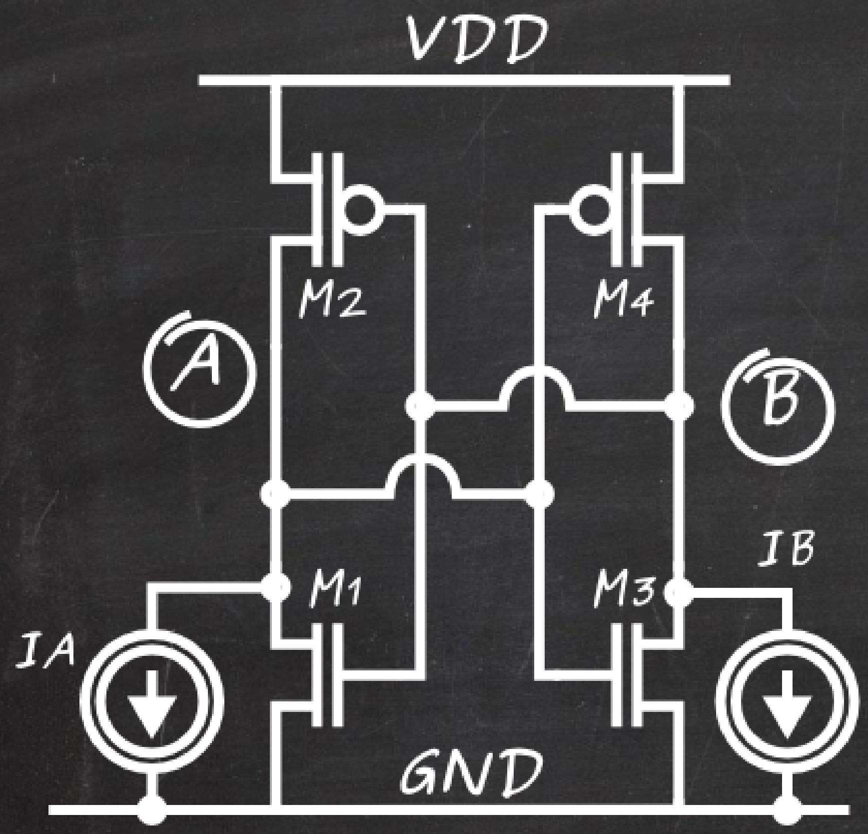
Critical charge



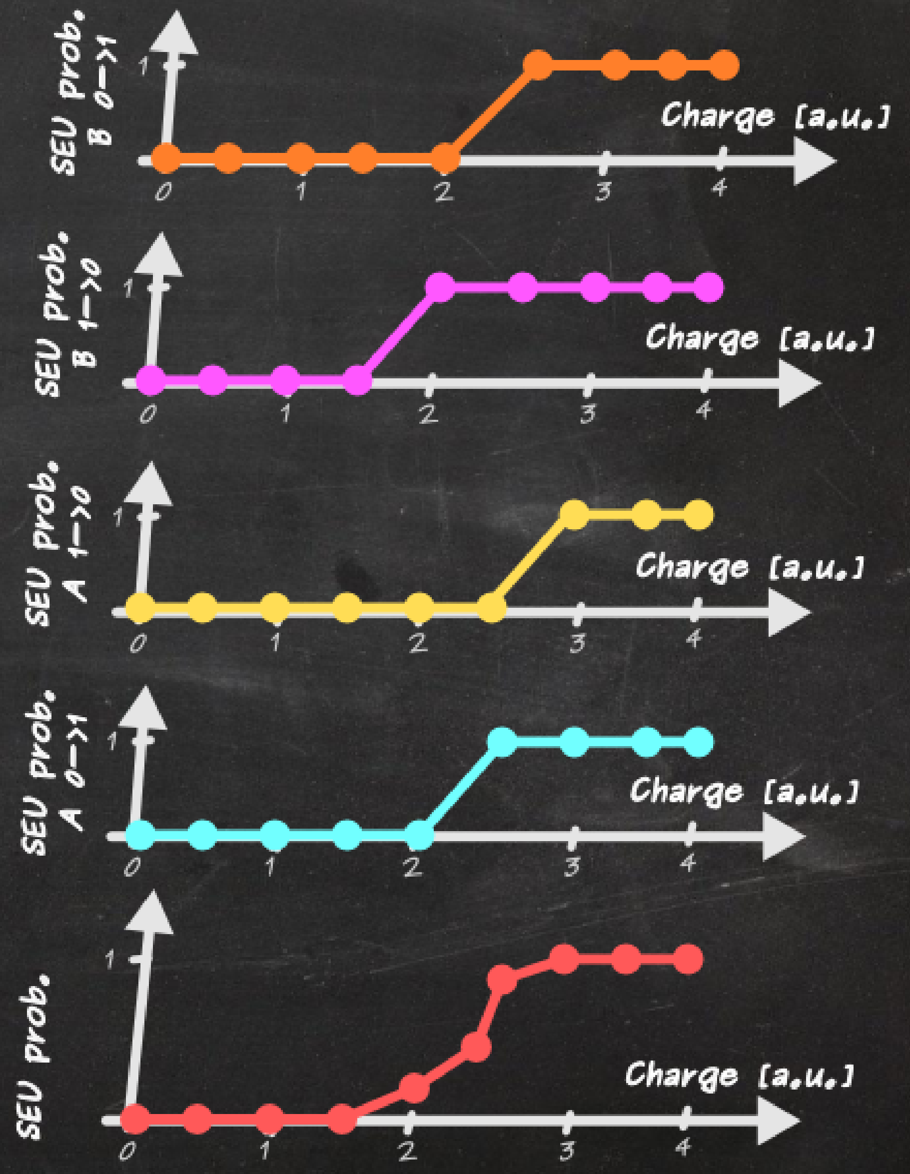


SRAM cell
(pass gates missing)

Critical charge



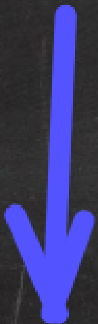
SRAM cell
(pass gates missing)



SEU and scaling

TMRG

Scaling facts:

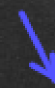


Supply Voltage

Node Capacitance

Critical energy
(less charge needed to change the state)

Physical dimensions



SEU cross section
(less likely that particle hits the sensitive area)

Overall effect depends on circuit topology and radiation environment

SEE mitigation techniques

TMRG

Technology level

minimizing sensitive depth (like SOI)

SEE mitigation techniques

TMRG

Technology level

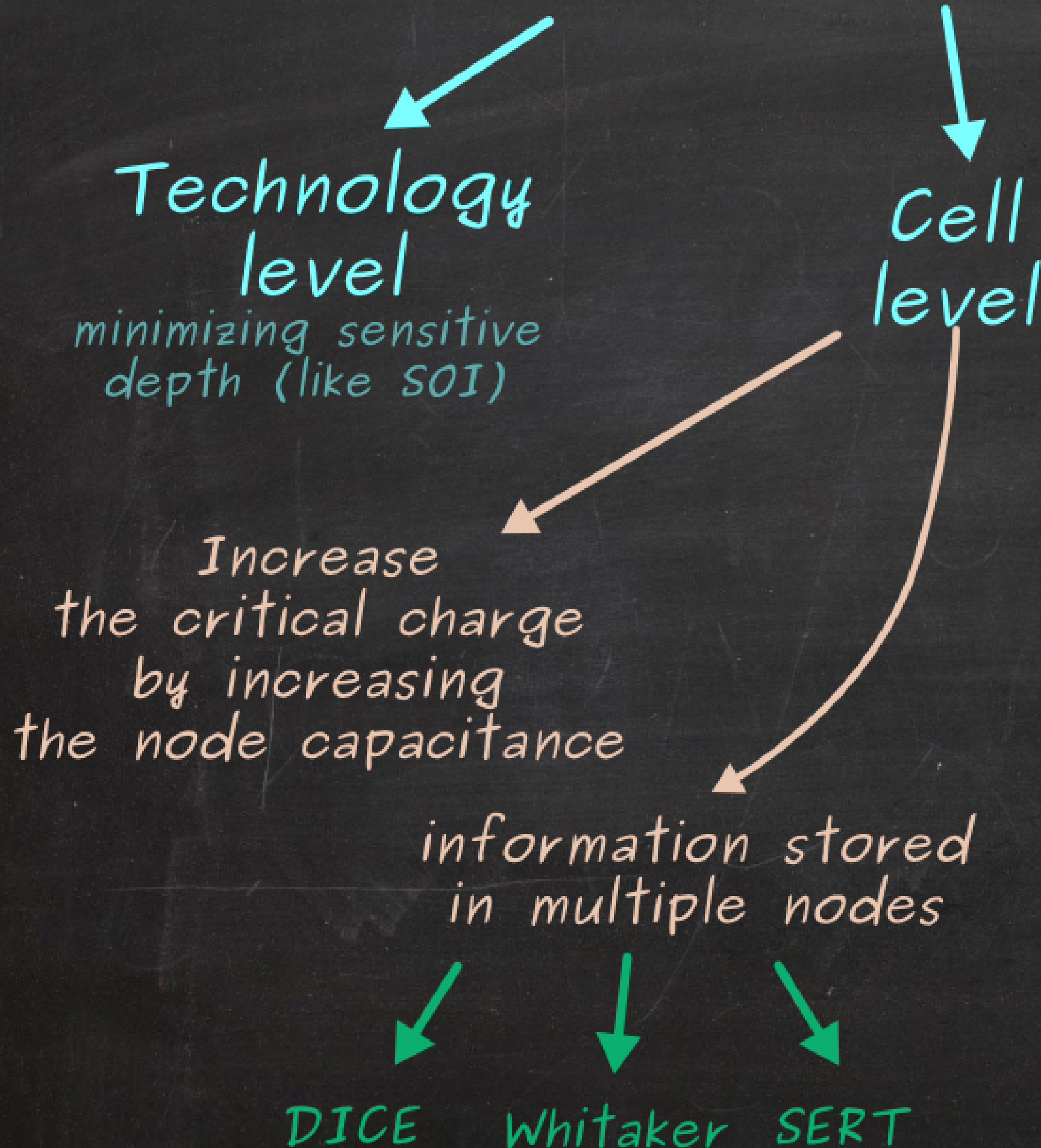
minimizing sensitive depth (like SOI)

Cell level

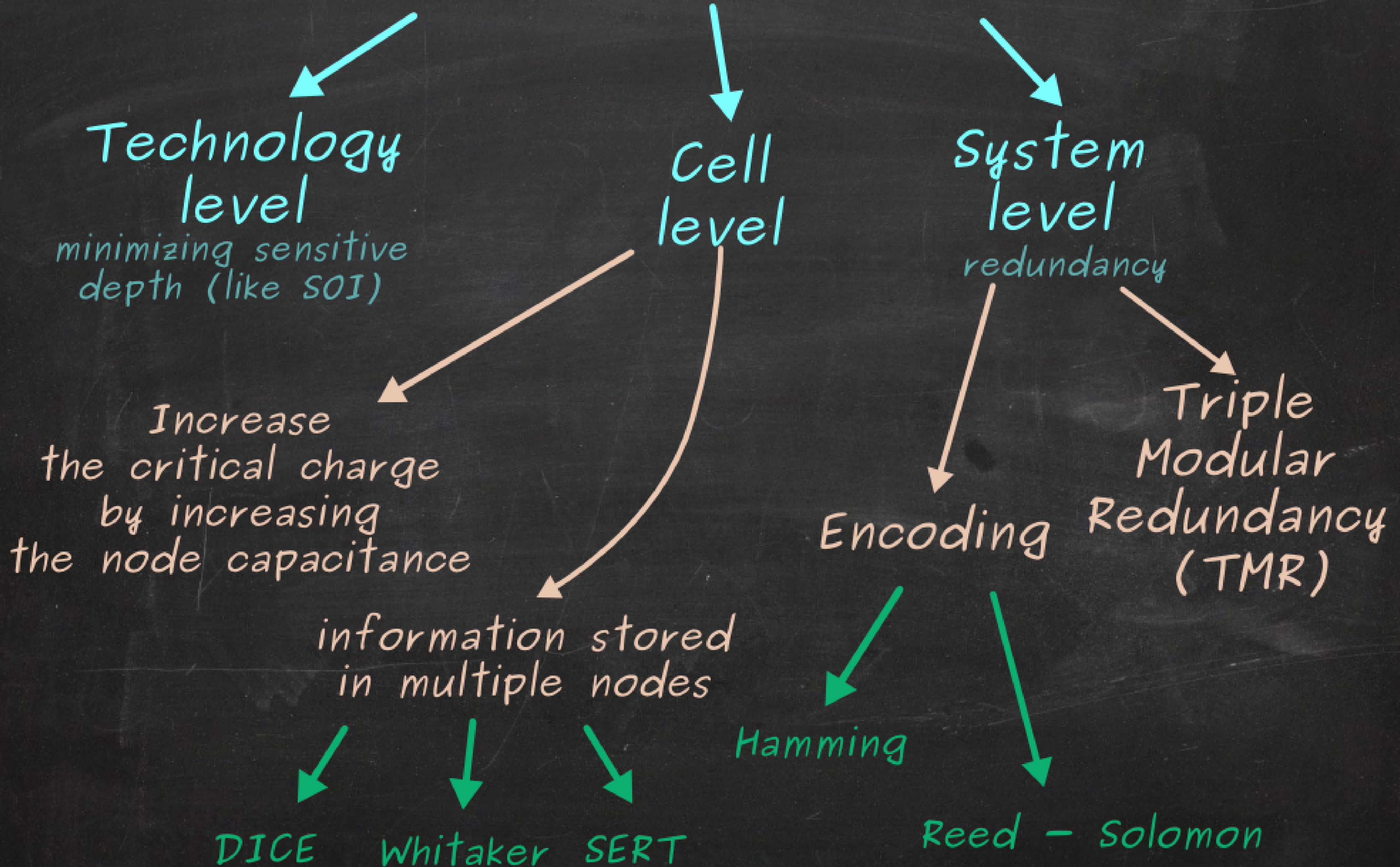
Increase the critical charge by increasing the node capacitance

-> larger transistors
("collection electrode" also gets bigger)

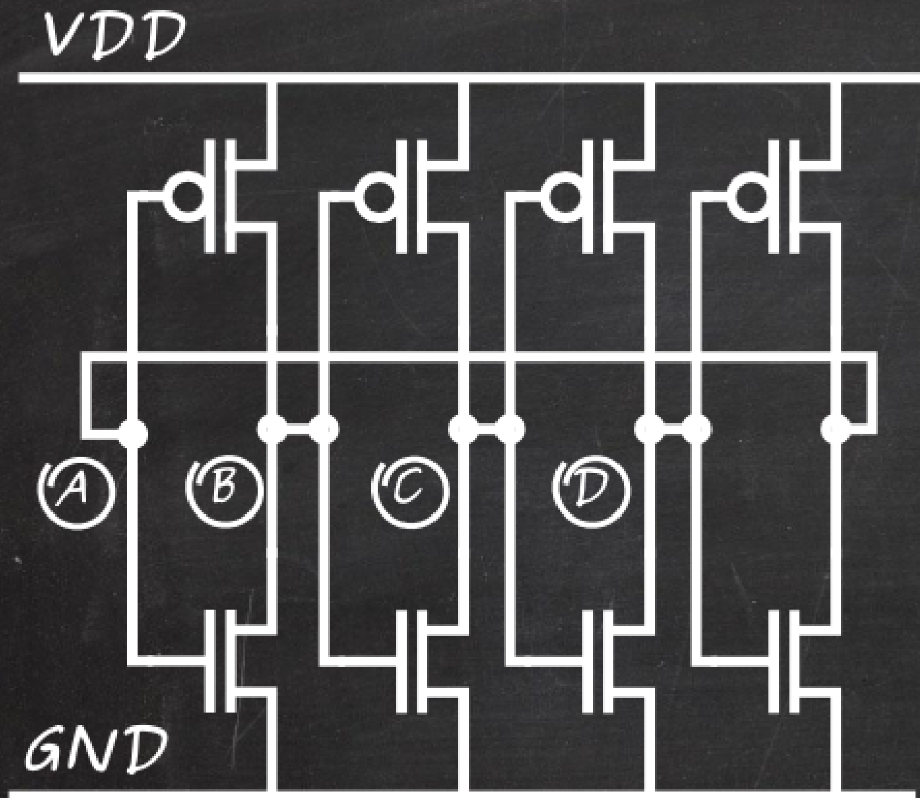
-> extra capacitance on metal layers


Technology level
minimizing sensitive depth (like SOI)
Cell level
Increase the critical charge by increasing the node capacitance
information stored in multiple nodes
DICE
Whitaker
SERT

SEE mitigation techniques



Doubled SRAM cell

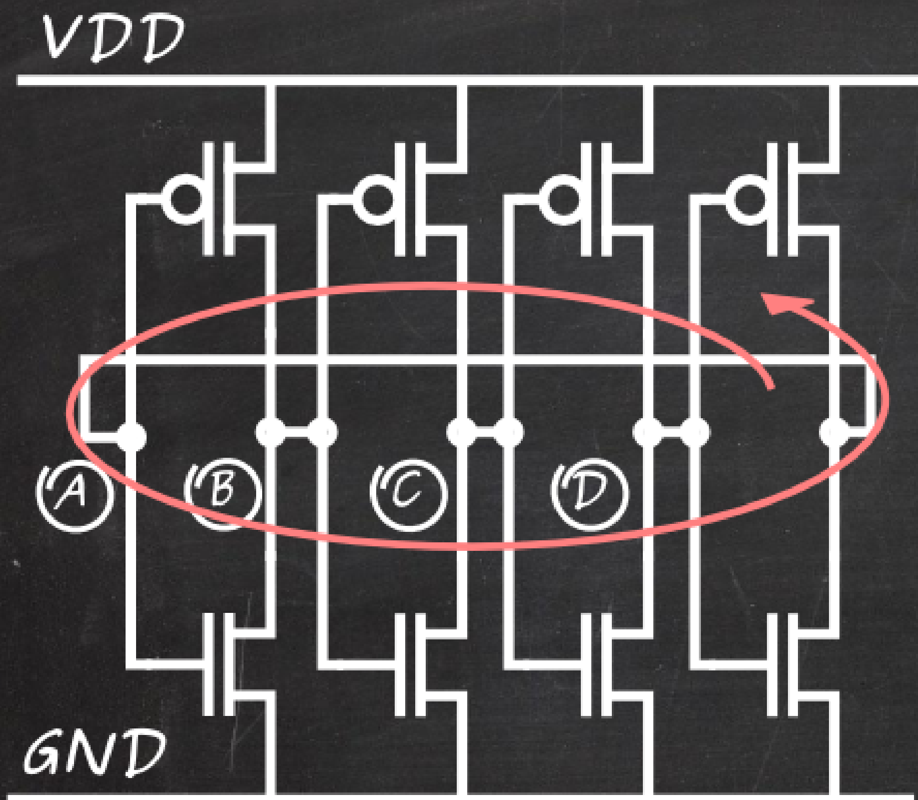


* information is stored in 4 nodes

"doubled" SRAM cell
(pass gates missing)

Doubled SRAM cell

TMRG



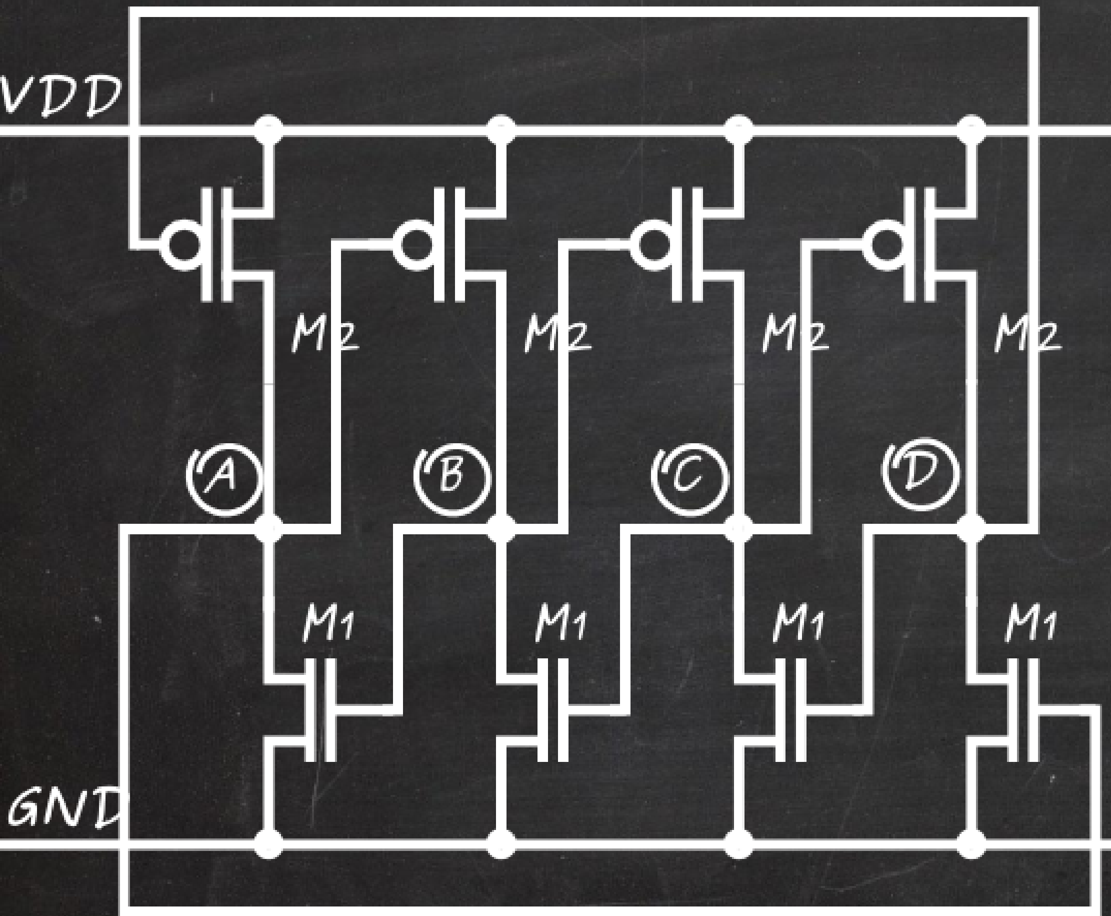
* information is stored in 4 nodes

"doubled" SRAM cell (pass gates missing)

this configuration does not offer any additional protection (an error can propagate through the loop to all nodes)

Dual Interlocked Cell (DICE)

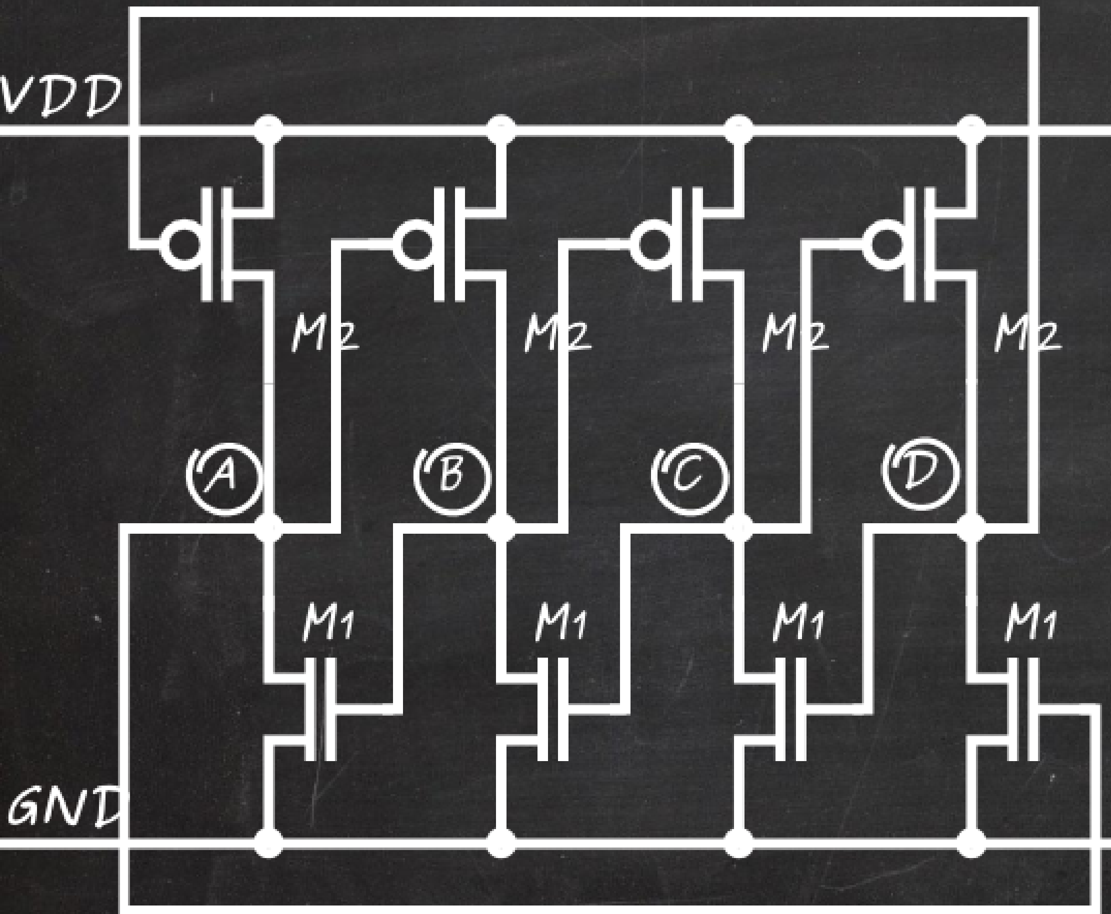
TMRG



DICE cell
(pass gates missing)

* information is stored in 4 nodes (A,B,C,D)
* two stable configurations (0,1,0,1) and (1,0,1,0)
* data can propagate in two directions:
  - low level -> left
  - high level -> right
* no logic value can propagate for more than one stage in the same direction

Dual Interlocked Cell (DICE)

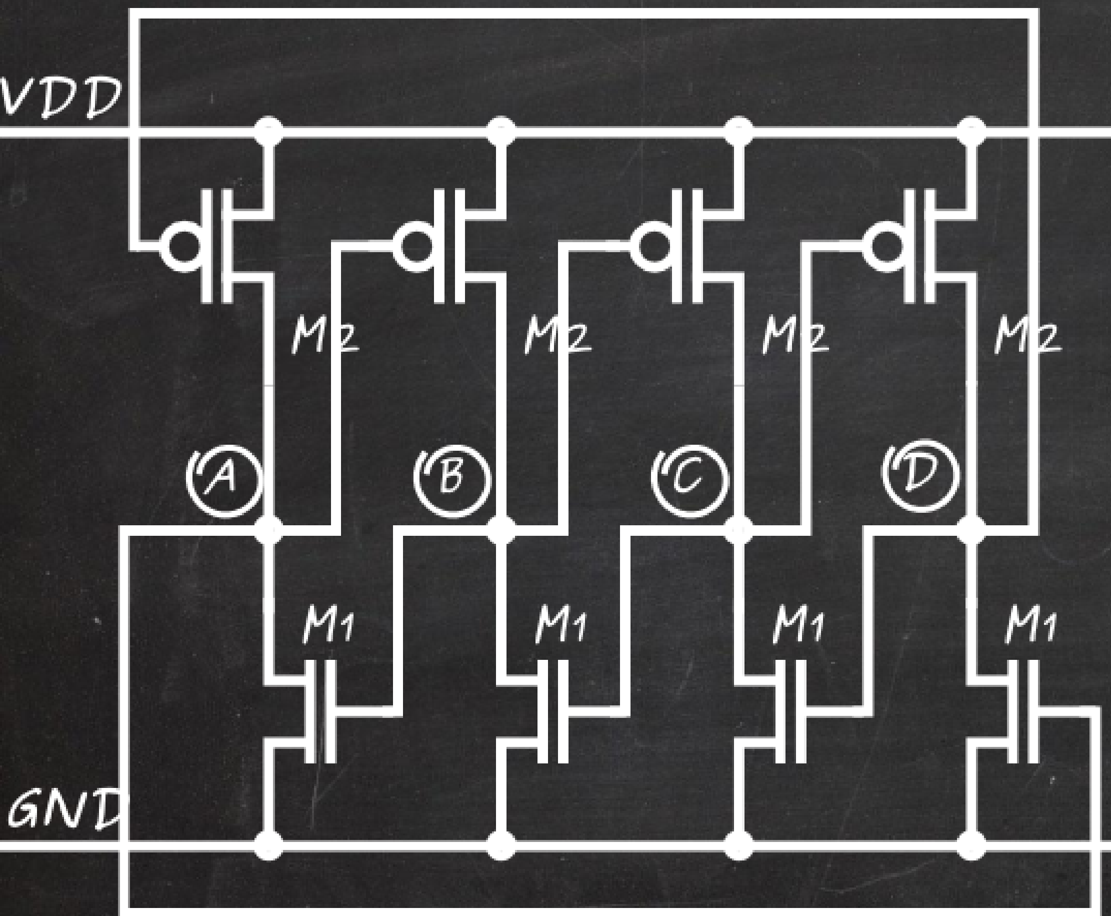


DICE cell
(pass gates missing)

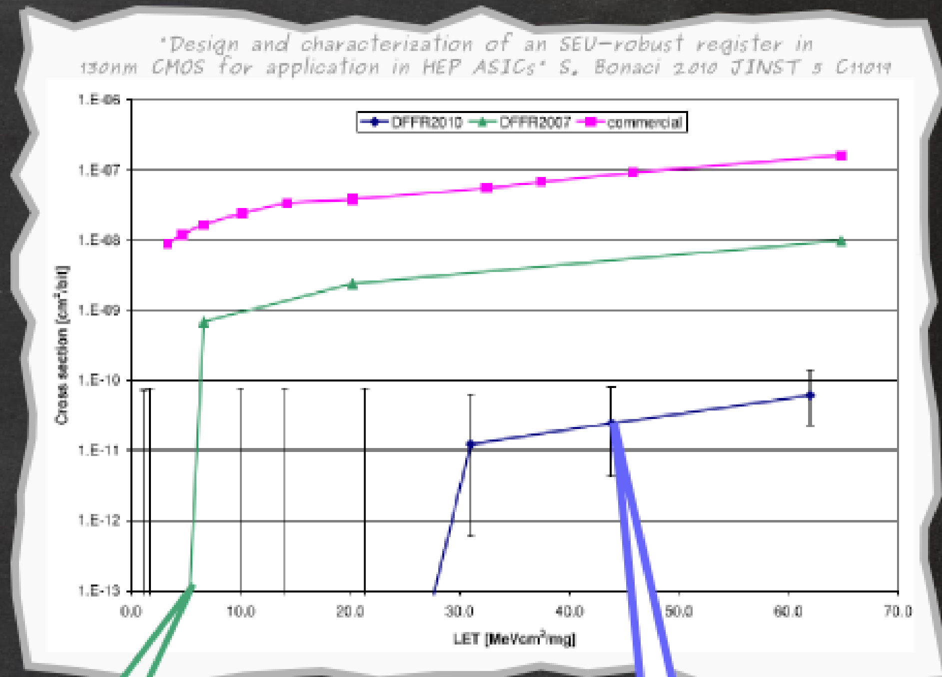
- information is stored in 4 nodes (A,B,C,D)
- two stable configurations (0,1,0,1) and (1,0,1,0)
- data can propagate in two directions:
  - low level -> left
  - high level -> right
- no logic value can propagate for more than one stage in the same direction
- drawbacks:
  - write requires access to two nodes
  - output glitch during SEU
  - rising clock during recovery time can latch the wrong value

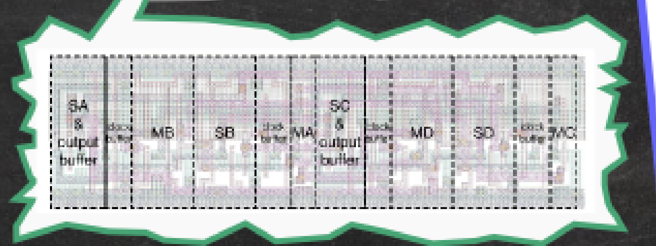
Dual Interlocked Cell (DICE)

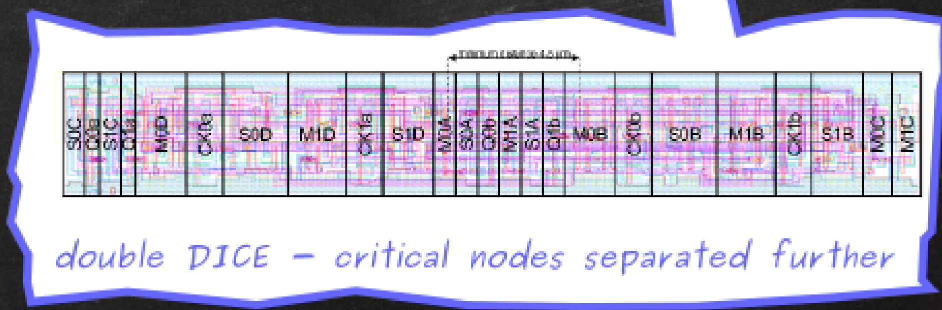
TMRG



DICE cell
(pass gates missing)







cell layout critical!

double DICE – critical nodes separated further

Hamming code example

Data: d1 d2 d3 d4

Message:	1 p1	2 p2	3 d1	4 p3	5 d2	6 d3	7 d4	
p1	x		x		x		x	p1⊕d1⊕d2⊕d4
p2		x	x			x	x	p2⊕d1⊕d3⊕d4
p2				x	x	x	x	p3⊕d2⊕d3⊕d4

Hamming code example

TMRG

Data: d1 d2 d3 d4

Message:

	1 p1	2 p2	3 d1	4 p3	5 d2	6 d3	7 d4	
p1	x		x		x		x	p1⊕d1⊕d2⊕d4
p2		x	x			x	x	p2⊕d1⊕d3⊕d4
p2				x	x	x	x	p3⊕d2⊕d3⊕d4

Data: 1 0 1 0

(tx) Message: 1 0 1 1 0 1 0

Hamming code example

TMRG

Data: d1 d2 d3 d4

Message:	1 p1	2 p2	3 d1	4 p3	5 d2	6 d3	7 d4	
p1	x		x		x		x	$p1 \oplus d1 \oplus d2 \oplus d4$
p2		x	x			x	x	$p2 \oplus d1 \oplus d3 \oplus d4$
p2				x	x	x	x	$p3 \oplus d2 \oplus d3 \oplus d4$

Data: 1 0 1 0

(tx) Message: 1 0 1 1 0 1 0

(rx) Message: 1 0 1 1 0 0 0

$s0 = p1 \oplus d1 \oplus d2 \oplus d4 = 0$

$s1 = p2 \oplus d1 \oplus d3 \oplus d4 = 1$

$s2 = p3 \oplus d2 \oplus d3 \oplus d4 = 1$

err pos = {s2,s1,s0} = 6 -> d3

Hamming code example

Data: d1 d2 d3 d4

Message:	1 p1	2 p2	3 d1	4 p3	5 d2	6 d3	7 d4	
p1	x		x		x		x	$p1 \oplus d1 \oplus d2 \oplus d4$
p2		x	x			x	x	$p2 \oplus d1 \oplus d3 \oplus d4$
p2				x	x	x	x	$p3 \oplus d2 \oplus d3 \oplus d4$

Data: 1 0 1 0

(tx) Message: 1 0 1 1 0 1 0

(rx) Message: 1 0 1 1 0 0 0

$s0 = p1 \oplus d1 \oplus d2 \oplus d4 = 0$

$s1 = p2 \oplus d1 \oplus d3 \oplus d4 = 1$

$s2 = p3 \oplus d2 \oplus d3 \oplus d4 = 1$

err pos = {s2,s1,s0} = 6 -> d3

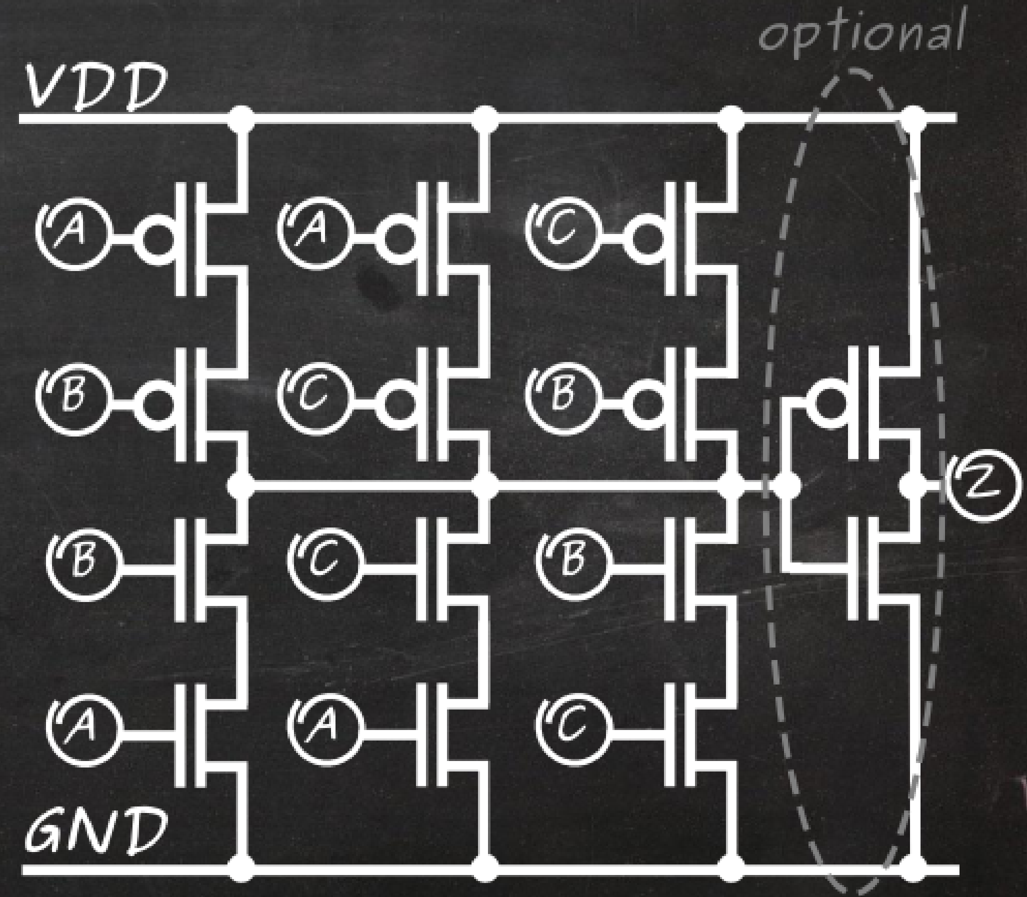
2^n bits -> n+1 check bits

Triple Modular Redundancy (TMR)

- TMR is a technique based on a **majority voter** cell
  - (2n+1) inputs (usualy 3)
  - 1 output equal to at least (n+1) inputs

A	B	C	Z
0	0	0	0
0	0	1	0
0	1	0	0
1	0	0	0
0	1	1	1
1	0	1	1
1	1	0	1
1	1	1	1

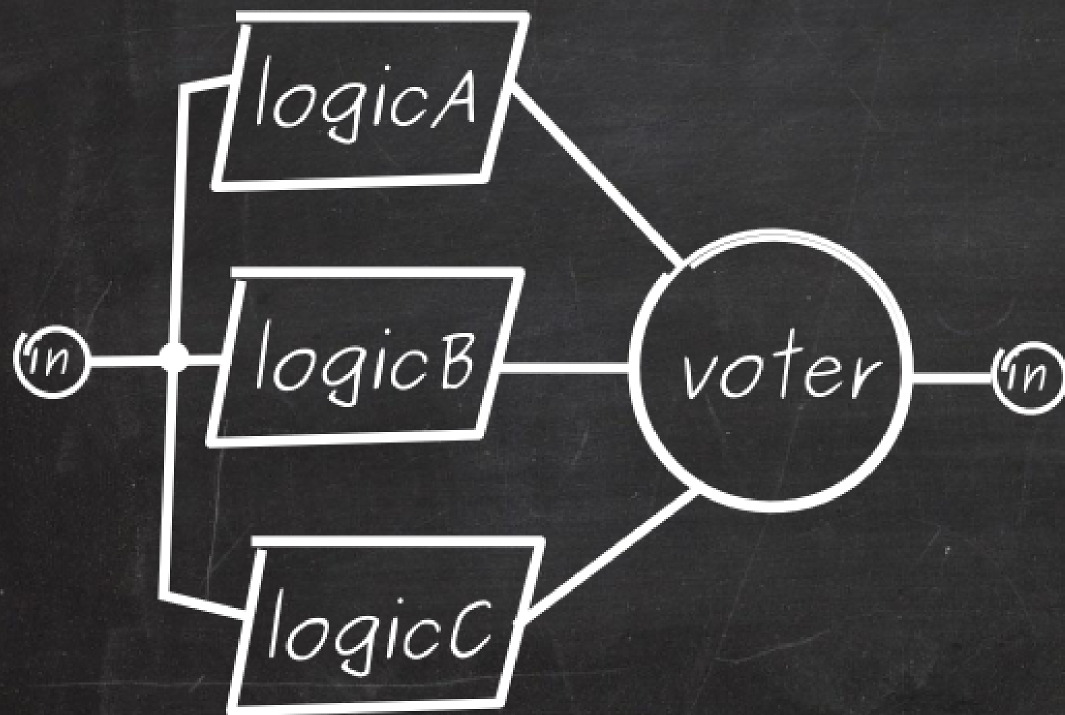
truth table



Triple Modular Redundancy (TMR)

TMRG

Normaly, the three blocks give the same output



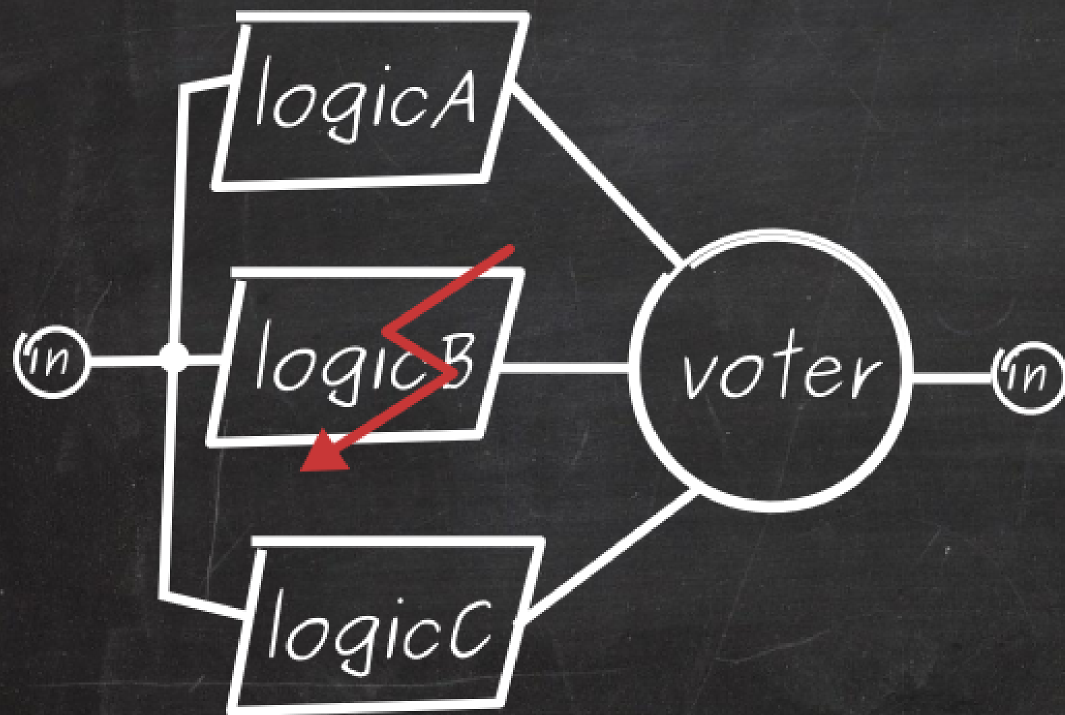
A	B	C	Z
0	0	0	0
0	0	1	0
0	1	0	0
1	0	0	0
0	1	1	1
1	0	1	1
1	1	0	1
1	1	1	1

voter truth table

Triple Modular Redundancy (TMR)

TMRG

An upset in one block (e.g. B) is masked by the voter and is not seen at the output!

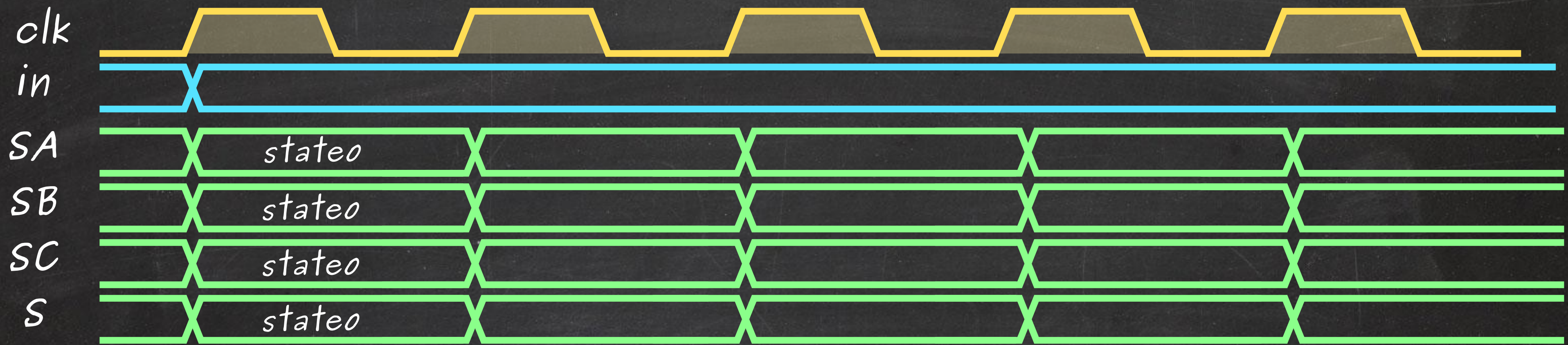


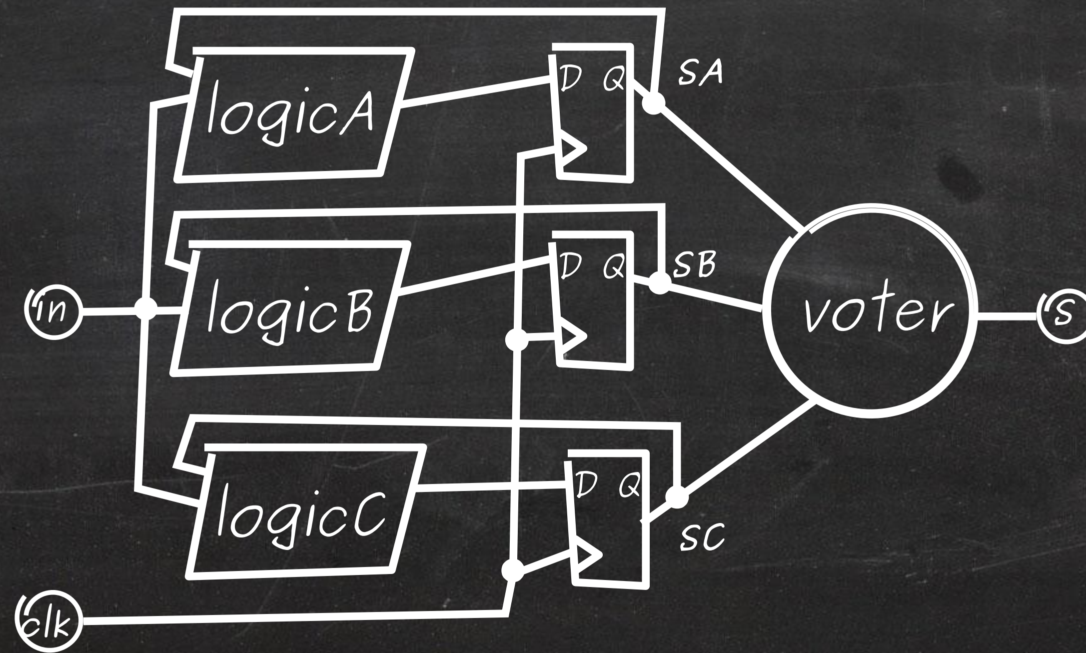
	A	B	C	Z
	0	0	0	0
	0	0	1	0
0->1	0	1	0	0
	1	0	0	0
	0	1	1	1
1-> 0	1	0	1	1
	1	1	0	1
	1	1	1	1

voter truth table

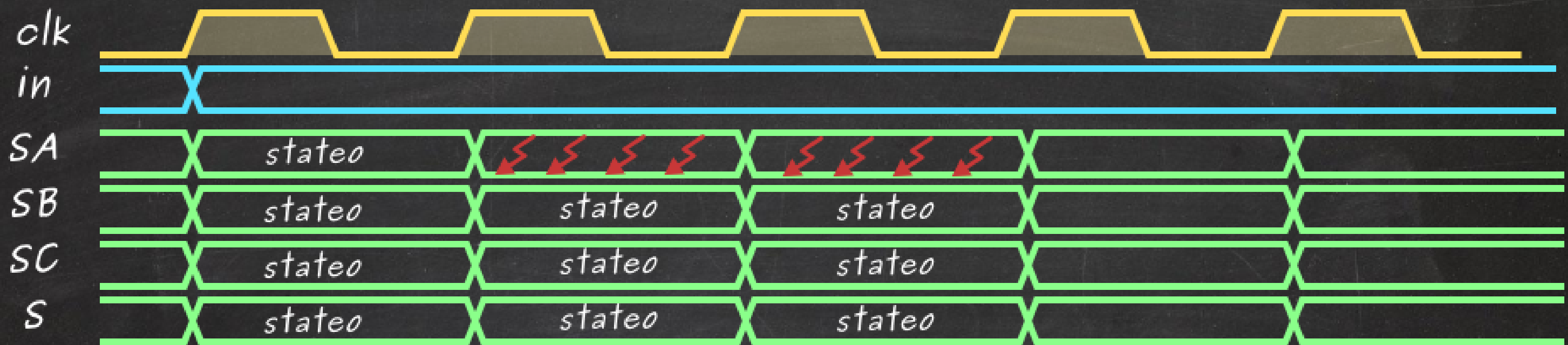
TMR: how to triplicate FSM

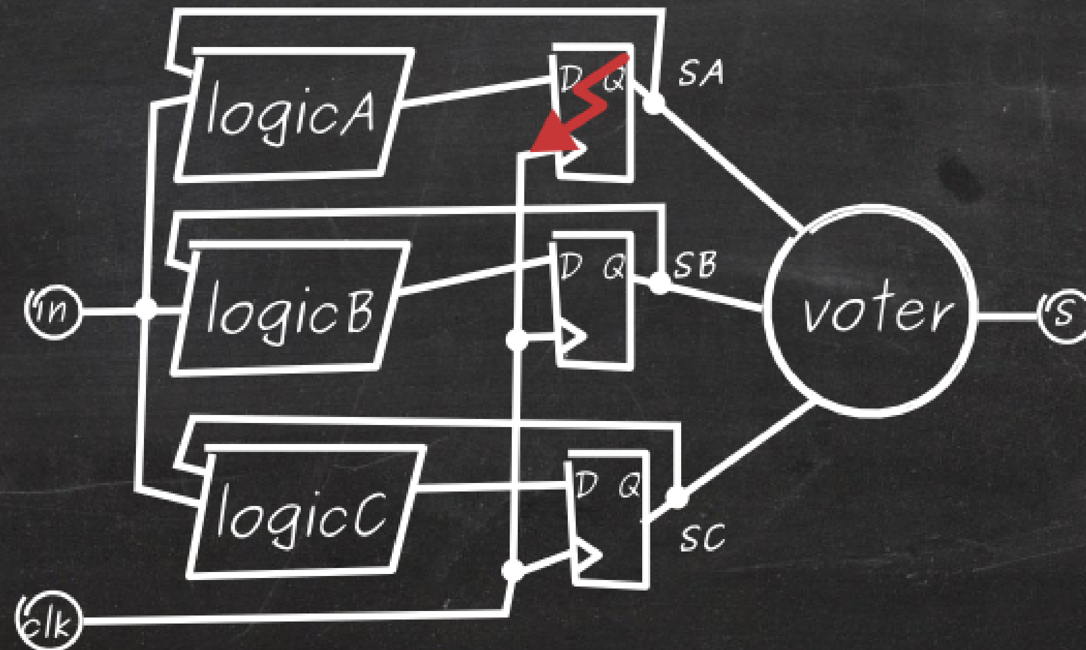
TMRG





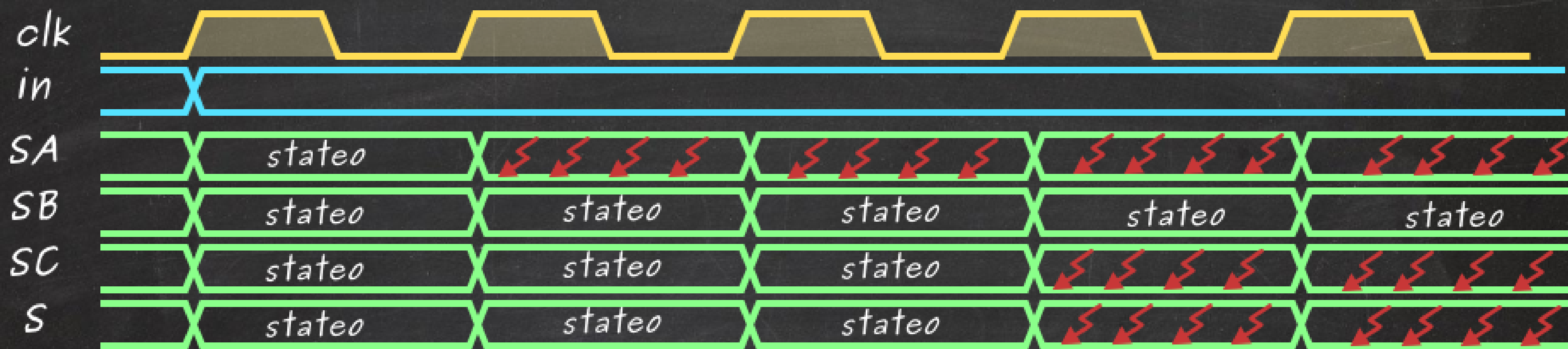
TMR: how to triplicate FSM

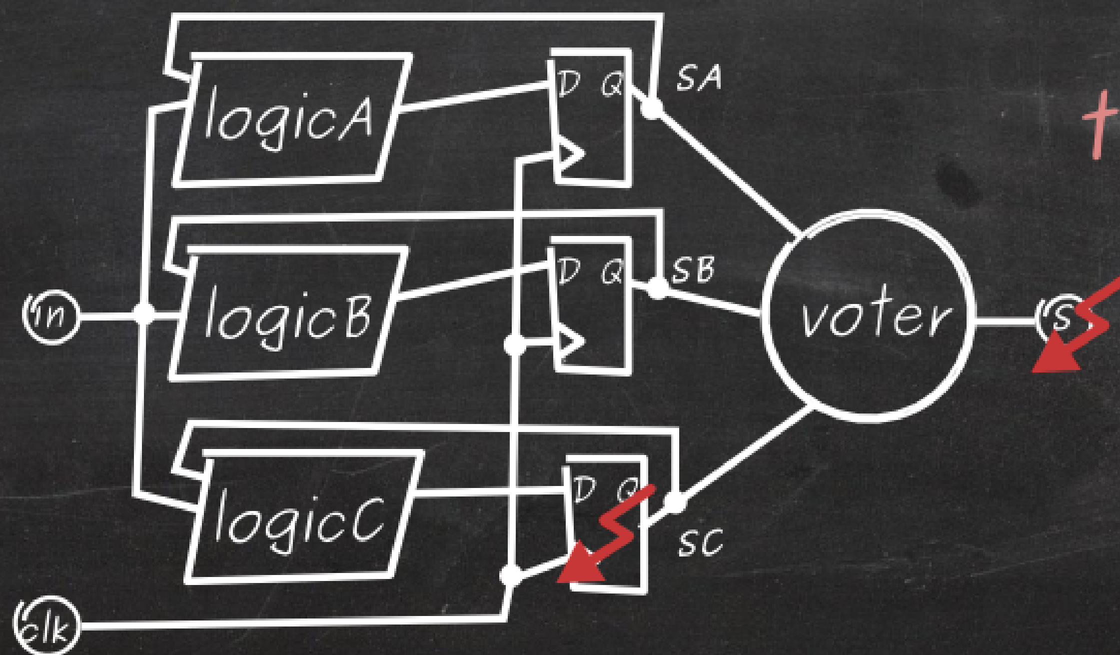




TMR: how NOT to triplicate FSM

TMRG



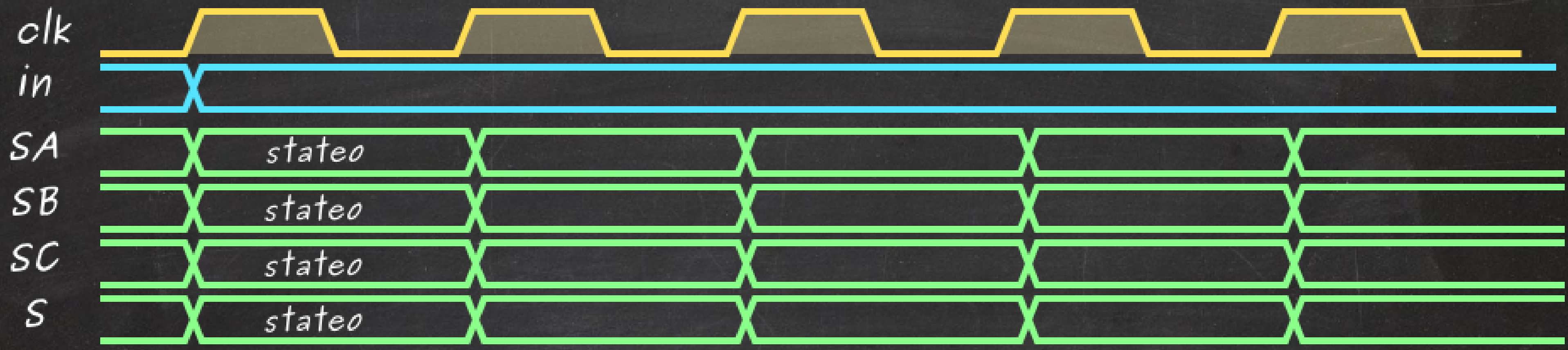


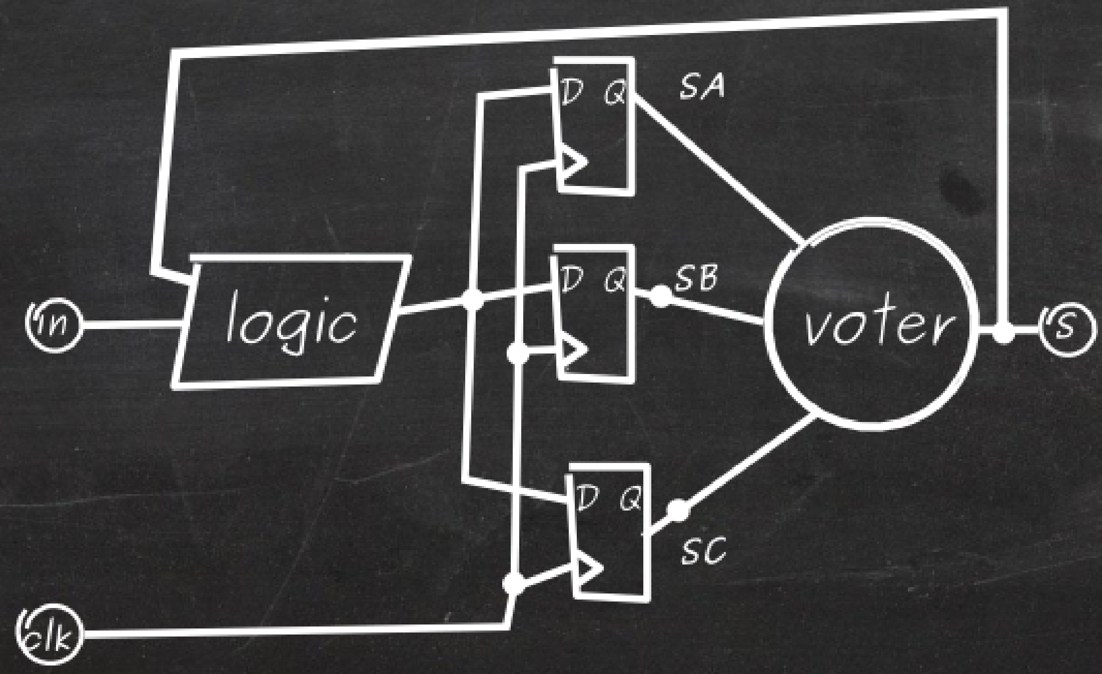
error at the output

An error in flip-flop may never be corrected

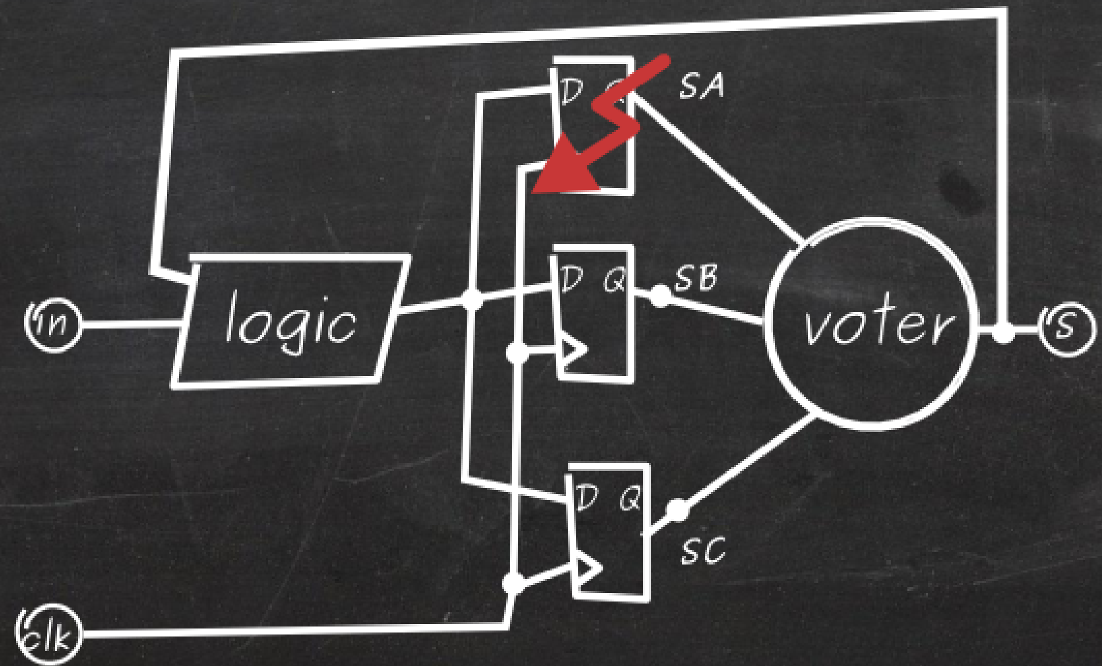
TMR: how to triplicate FSM

TMRG

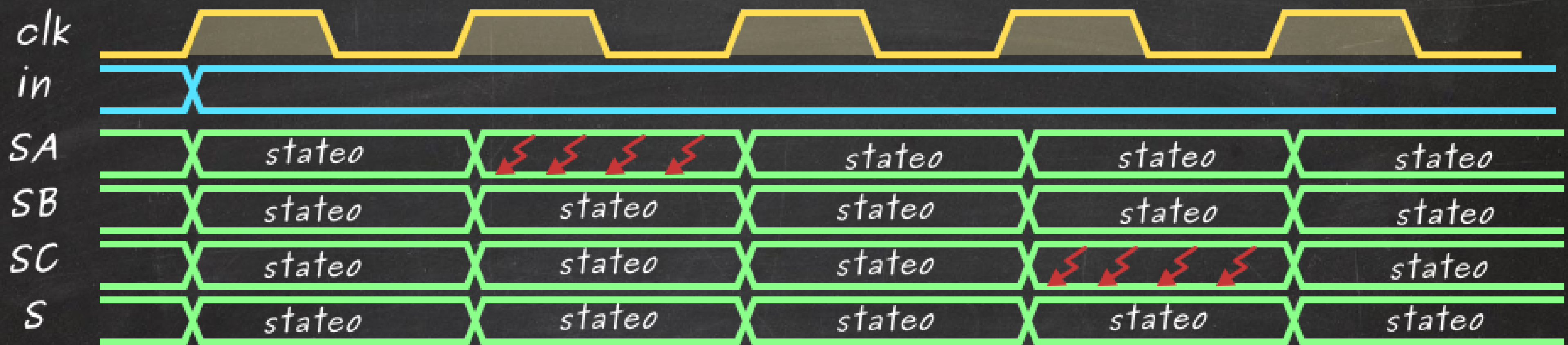


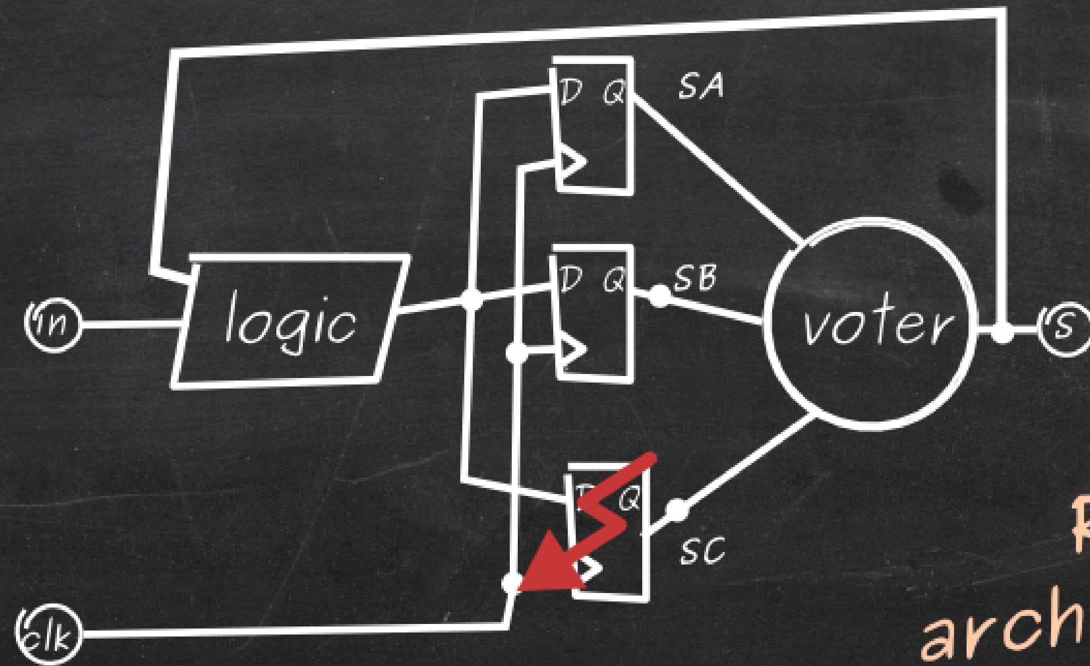


TMR: how to triplicate FSM



TMR: how to triplicate FSM

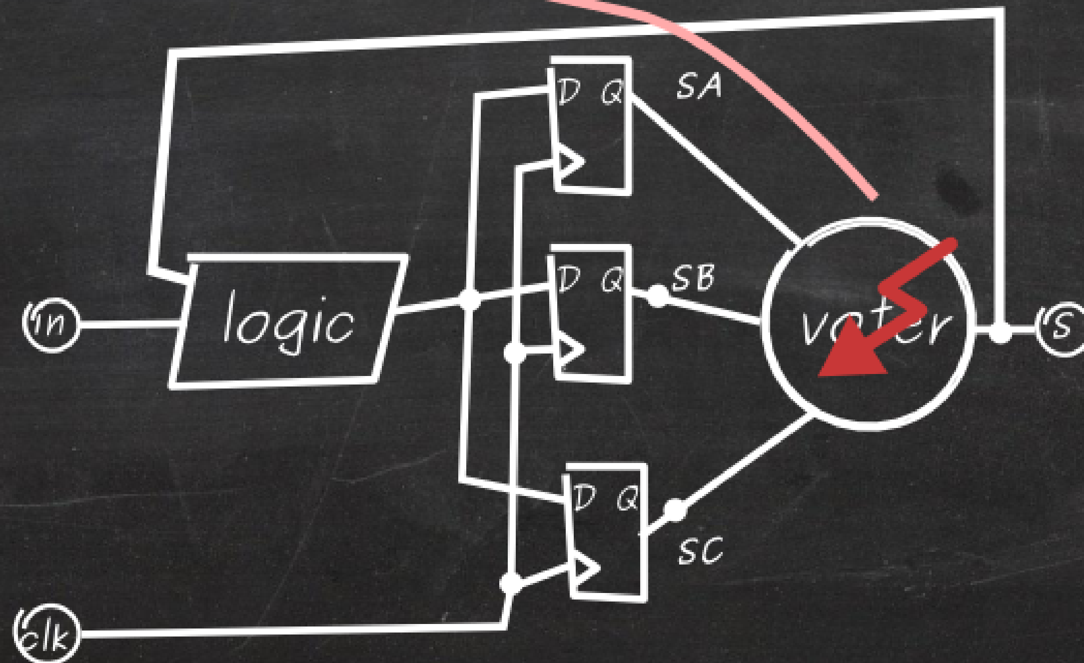




Robust architecture ?

An error is removed from the system after one clock cylce

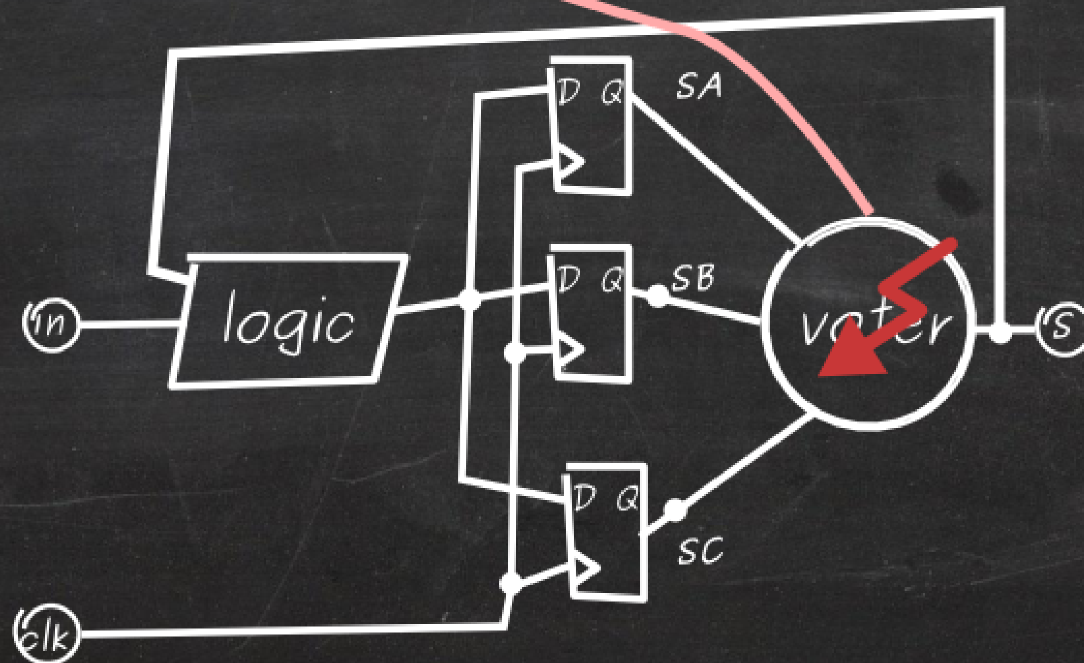
TMR: how to triplicate FSM



Single Event Transients (SET) in the voter can results is a short glitch at the output, but the state should not be affected

TMR: how to triplicate FSM

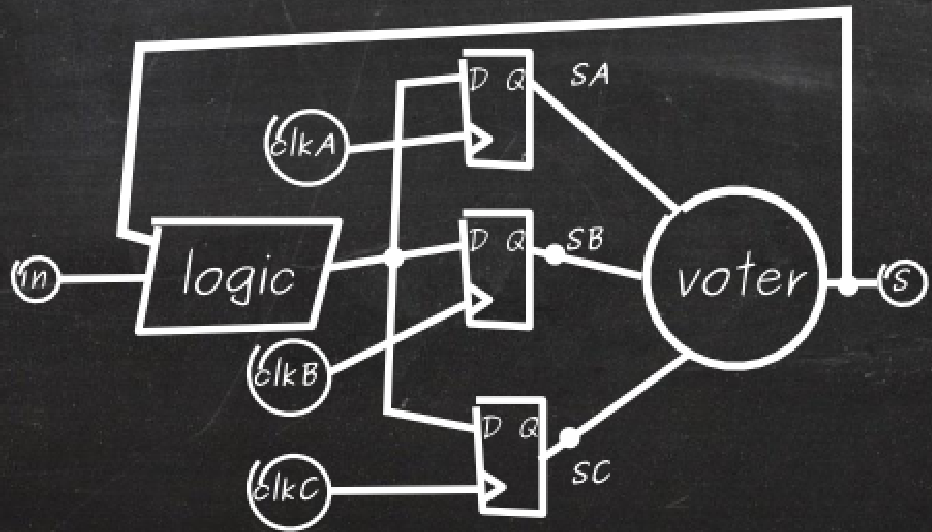
TMRG



This architecture is sensitive to Single Event Transients (SET) in combinatorial logic and/or voter occurring close to the clock edge

TMR: how to triplicate FSM (clock skew)

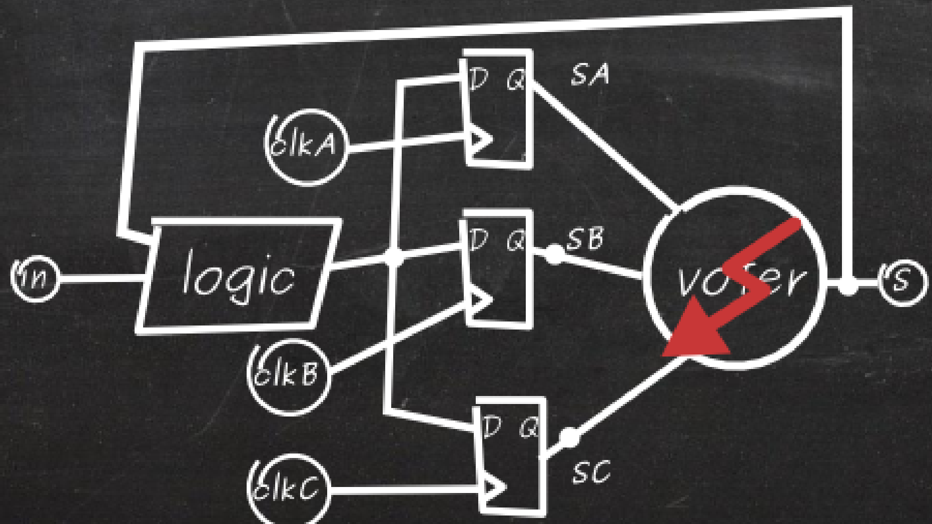
in
clkA
clkB
clkC
SA state0
SB state0
SC state0
S state0



TMR: how to triplicate FSM (clock skew)

TMRG

in
clkA
clkB
clkC
SA state0 state0
SB state0 state0 state0
SC state0 state0 state0
S state0 state0 state0

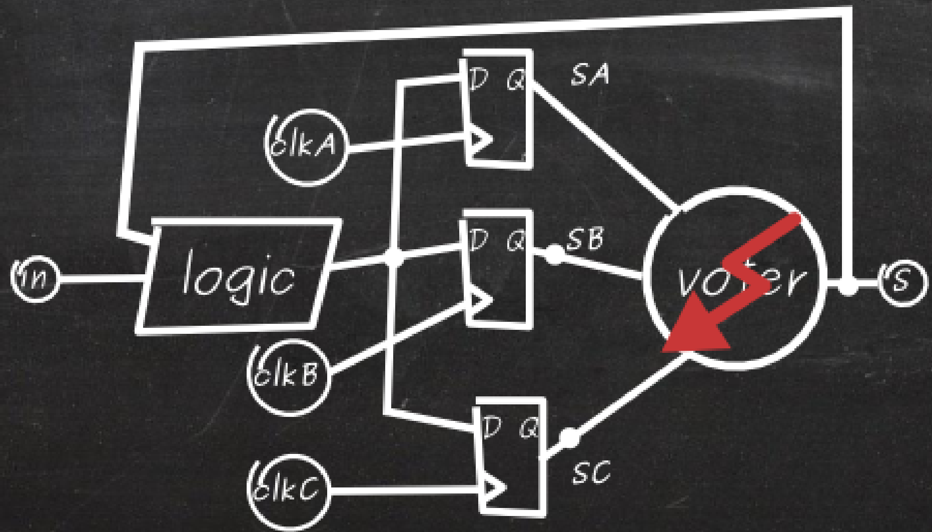


SET can be latched only in one flip-flop

TMR: how to triplicate FSM (clock skew)

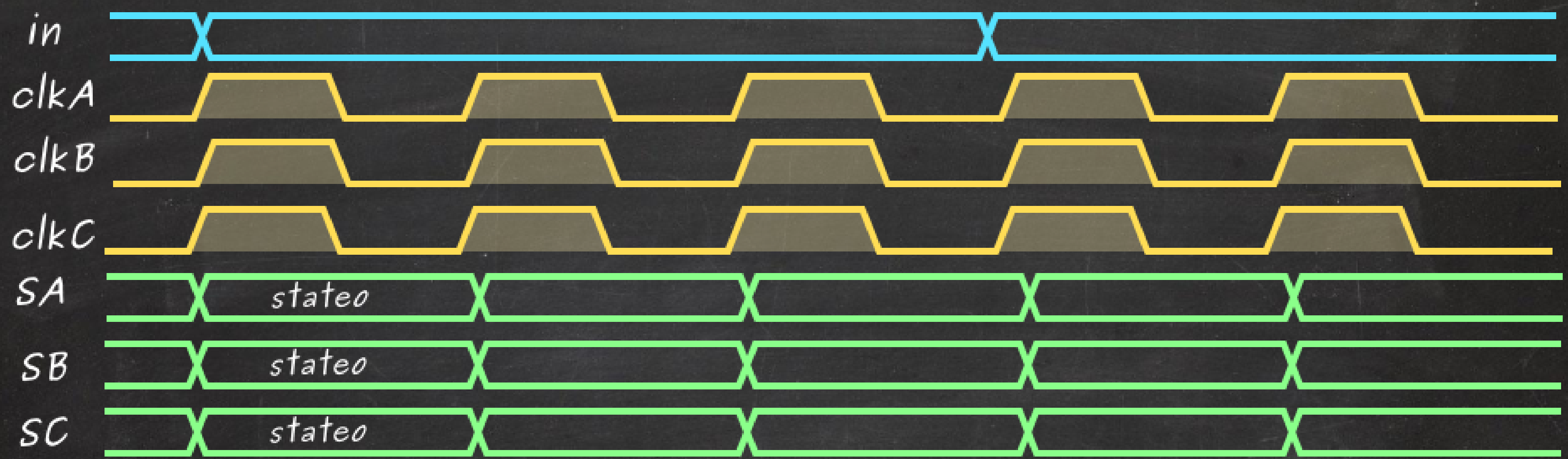
in
clkA
clkB
clkC
SA: state0 | state0 | state1 | ? ? ? ? ? ?
SB: state0 | state0 | state0 | ? ? ? ? ? ?
SC: state0 | state0 | state0 | ? ? ? ? ? ? | ? ? ? ? ? ?
S: state0 | state0 | state0 | ? ? ? ? ? ? ? | ? ? ? ? ? ?

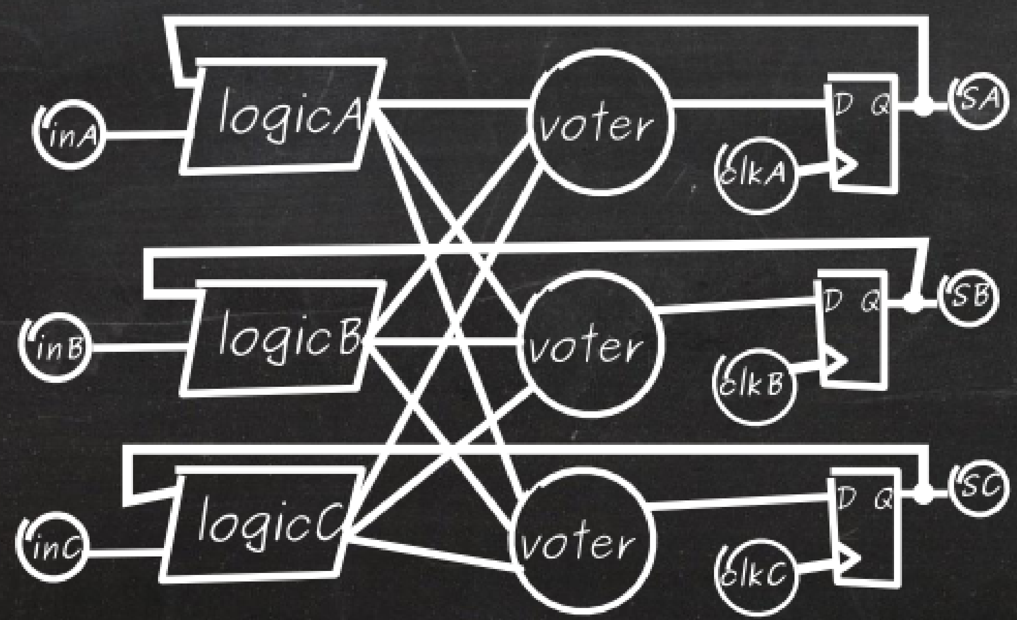
(race condition)



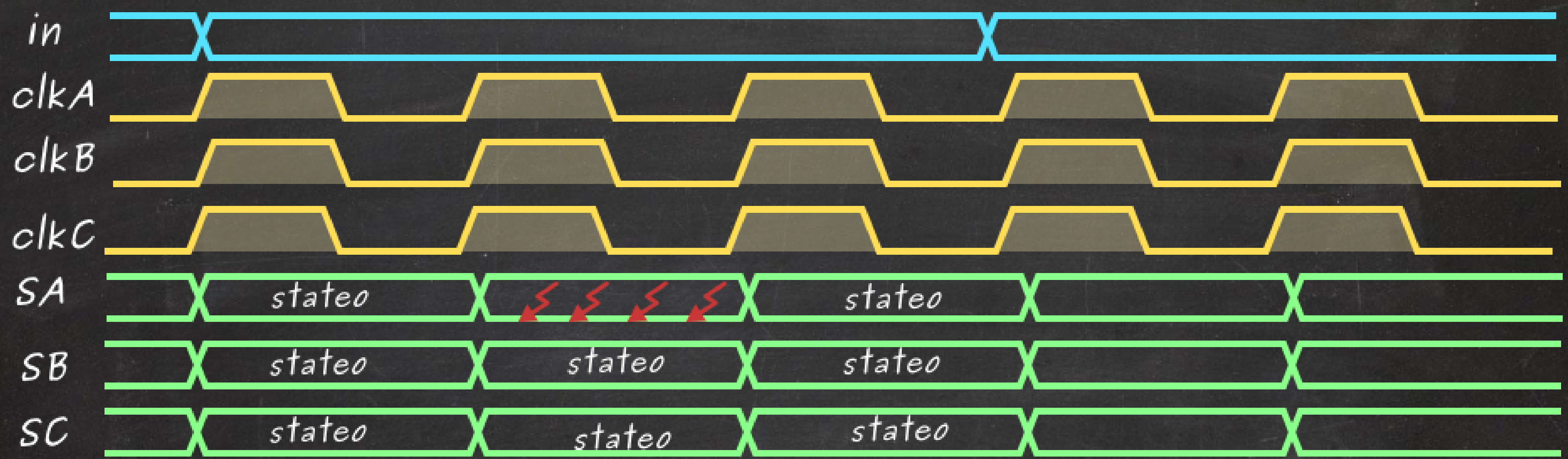
Not fully predictable

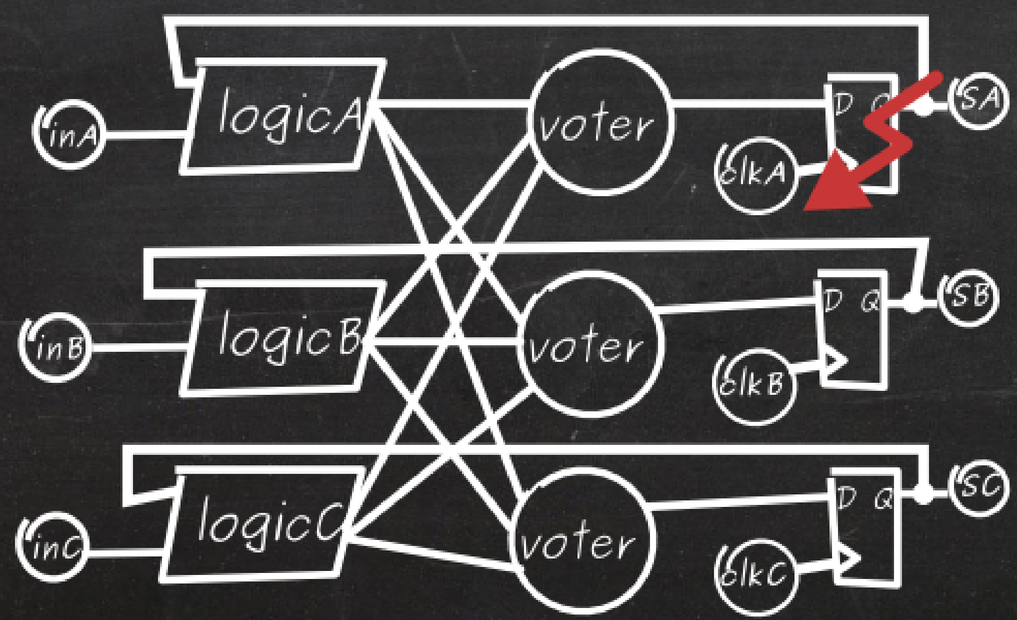
TMR: how to triplicate FSM (full TMR)



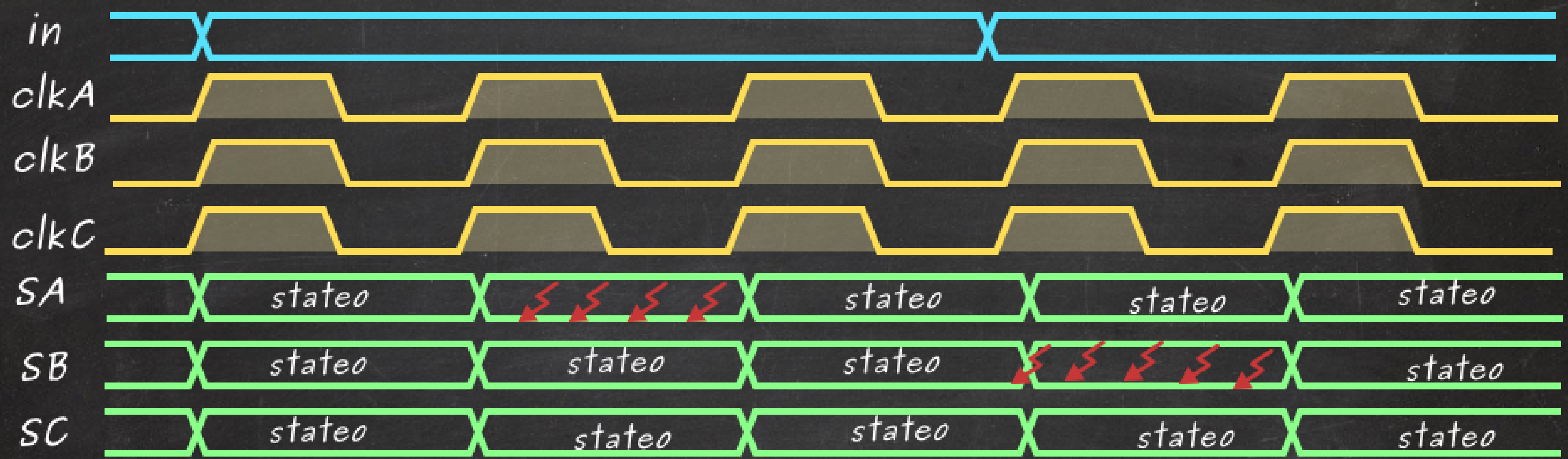


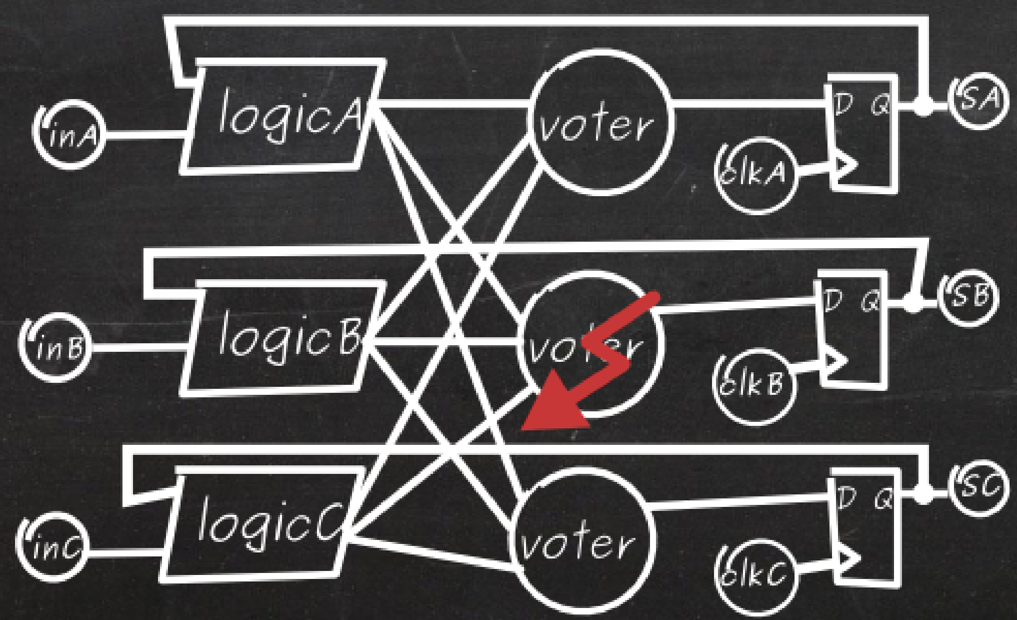
TMR: how to triplicate FSM (full TMR)



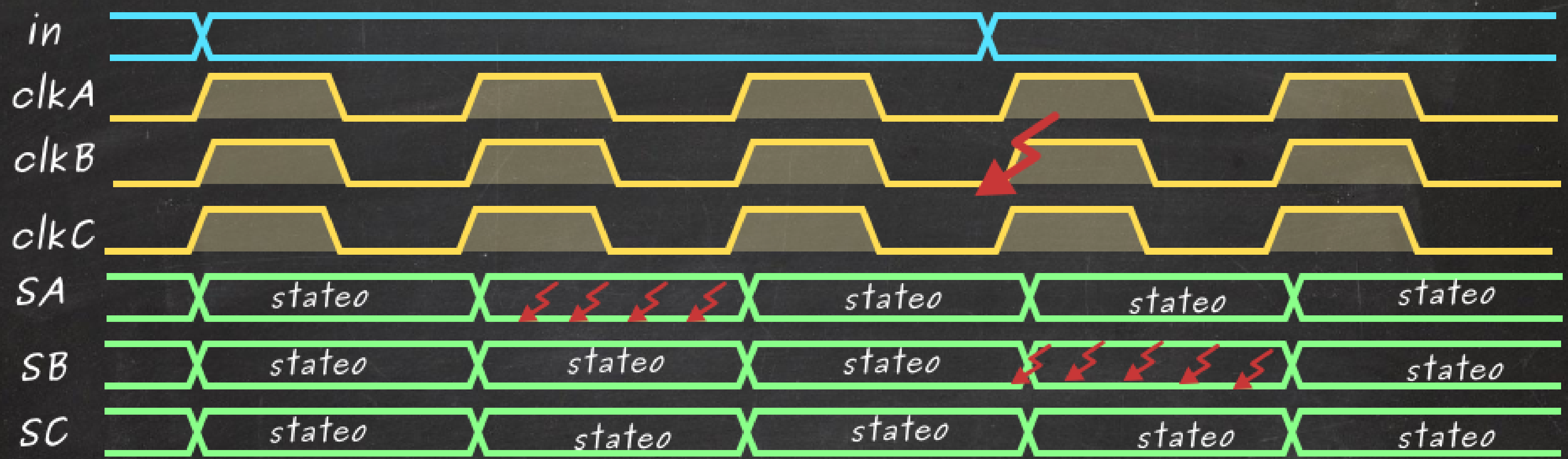


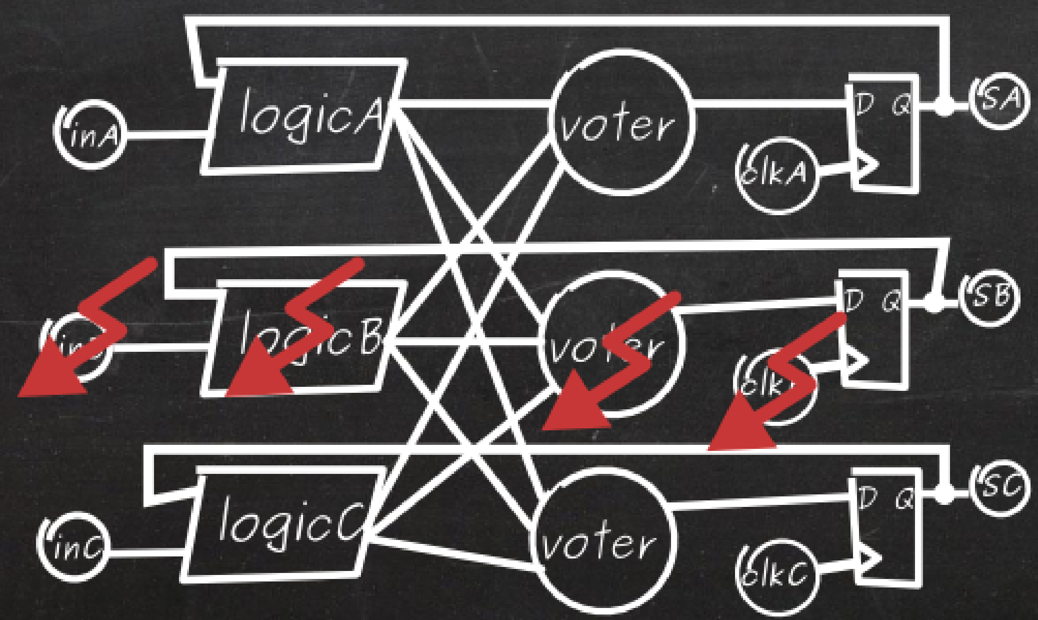
TMR: how to triplicate FSM (full TMR)





TMR: how to triplicate FSM (full TMR)





Ultimate protection ?

TMR Variants

TMRG

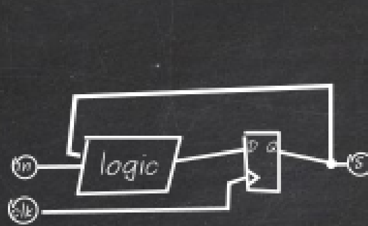
		Triplicated Registers	Triplicated Registers + clock skew	Full TMR
Resoures (power, area)	FF	x3	x3	x3
	logic	x1	x1	x3
	voters	x1	x1	x3
	clocks	x1	x3	x3
Speed		:-| +voter delay	:-(+voter delay +clock skew	:-| +voter delay
Protection		:-(	:-|	:-)

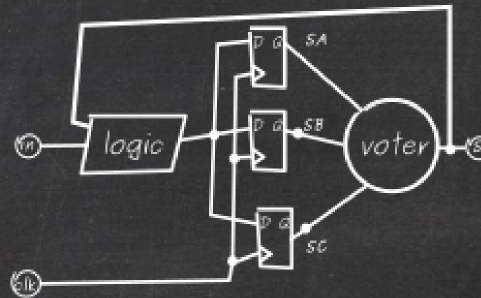
Which one to use ?

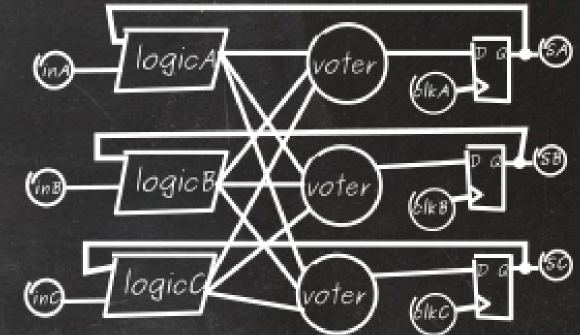
TMR Variants – example

TMRG

* 8 Bit accumlator @ 100 MHz
* 65 nm CMOS technology, 9 track library, typical corner





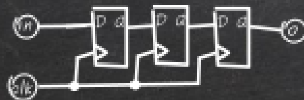


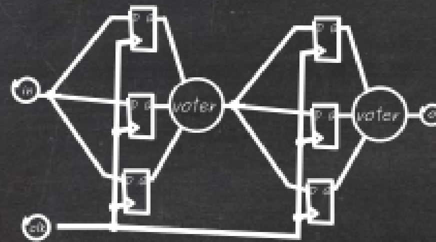
	Not Triplicated	Triplicated Registers	Full TMR
Power [µW]	20.2	57.8 (x2.8)	111.9 (x5.5!)
Slack [ns]	8.01	7.5	7.63
Area [μm^2]	130	348 (x2.6)	486 (x3.7)

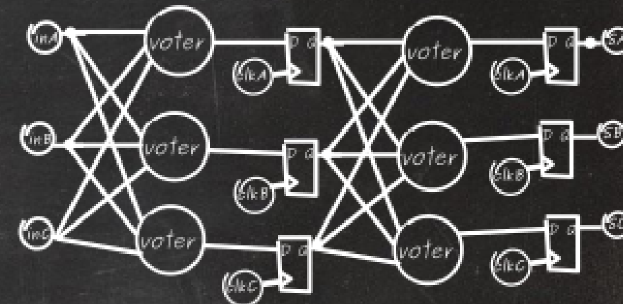
*) estimations after synthesis

TMR Variants – example

* 8 Bit shift register @ 100 MHz
* 65 nm CMOS technology, 9 track library, typical corner







	Not Triplicated	Triplicated Registers	Full TMR
Power [μW]	9.2	26.1 (x2.8)	41.2 (x4.5!)
Area [μm²]	64	201 (x3.1)	288 (x4.5!)
Slack [ns]	9.67	9.17	9.28
fmax [GHz]	3.03	1.20 (39%)	1.38 (45%)

*) estimations after synthesis

TMR Variants – guidelines

TMRG

- Full TMR is recommended whenever possible
- In the case of limited resources (area, power):
  - try to use full TMR or triplicated registers for state machines
- Always make sure that all states in the FSM are defined (default: nextState=reset;)

How to TMR the circuit ?

```
module inverter(
  input D,
  output ZN);
assign ZN=!D;
endmodule
```

Process:
1) copy & paste
2) add postfixes (A,B,C)

```
module inverterTMR(
  input DA,
  input DB,
  input DC,
  output ZNA,
  output ZNB,
  output ZNC);
assign ZNA=!DA;
assign ZNB=!DA;
assign ZNC=!DC;
endmodule
```

How to TMR the circuit ?

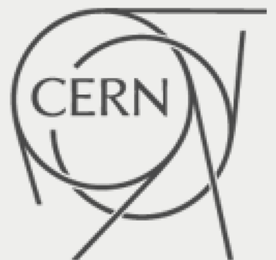
```
module inverter(
  input D,
  output ZN);
assign ZN=!D;
endmodule
```

Process:
1) copy & paste
2) add postfixes (A,B,C)

```
module inverterTMR(
  input DA,
  input DB,
  input DC,
  output ZNA,
  output ZNB,
  output ZNC);
assign ZNA=!DA;
assign ZNB=!DA;
assign ZNC=!DC;
endmodule
```

Drawbacks of manual triplication:
- time consuming
- error prone

Triple Modular Redundancy Generator



TMRG : motivation

The purpose of the TMRG tool set is to automatize the process of triplicating digital circuits.

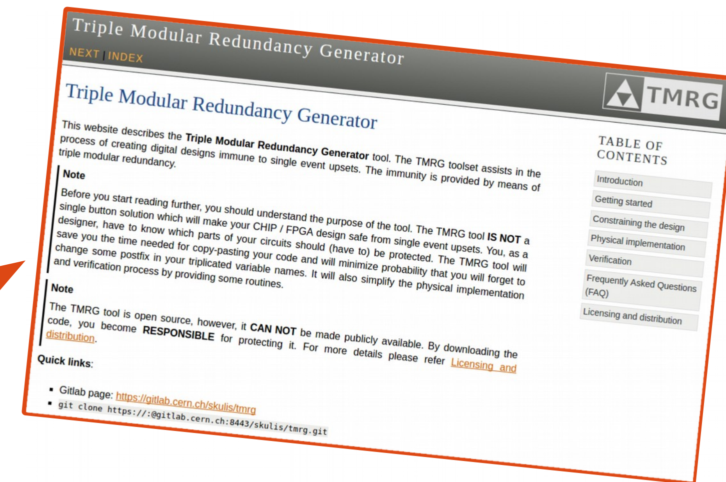
Requirements, the tool should:

- be compatible with the ASIC design flows used in the HEP community (Verilog RTL, Cadence tool chain)
- not over constraint the user's coding style (the source Verilog must be synthesizable)
- allow to obtain various flavors of TMR (registers only, full triplication, …)
- assists in the physical implementation stage (synthesis, P&R)
- assists the designer in the verification process (generation of SEE)
- it can be run in batch mode (fully automatic flow)

TMRG : project status

TMRG

- Project started: **Jan 2015**
- Project size: **>13000 lines of code**
- Documentation size: **69 pages (pdf&html)**
- Active user base: **>7 designers**
- **“Open source”**, hosted in CERN **git repository (700+ commits)**



Triple Modular Redundancy Generator

NEXT | INDEX

TMRG

Triple Modular Redundancy Generator

This website describes the **Triple Modular Redundancy Generator** tool. The TMRG toolset assists in the process of creating digital designs immune to single event upsets. The immunity is provided by means of triple modular redundancy.

Note

Before you start reading further, you should understand the purpose of the tool. The TMRG tool **IS NOT** a single button solution which will make your CHIP / FPGA design safe from single event upsets. You, as a designer, have to know which parts of your circuits should (have to) be protected. The TMRG tool will save you the time needed for copy-pasting your code and will minimize probability that you will forget to change some postfix in your triplicated variable names. It will also simplify the physical implementation and verification process by providing some routines.

Note

The TMRG tool is open source, however, it **CAN NOT** be made publicly available. By downloading the code, you become **RESPONSIBLE** for protecting it. For more details please refer Licensing and distribution.

Quick links:

- Gitlab page: https://gitlab.cern.ch/skulis/tmrg
- `git clone https://:@gitlab.cern.ch:8443/skulis/tmrg.git`

TABLE OF CONTENTS

- Introduction
- Getting started
- Constraining the design
- Physical implementation
- Verification
- Frequently Asked Questions (FAQ)
- Licensing and distribution

Activity Index

140
120
100
80
60
40
20
0

Jan 15, Apr 15, Jul 15, Oct 15, Jan 16, Apr 16, Jul 16, Oct 16, Jan 17, Apr 17

Initial development

Minor bug fixes

New features and new syntax support

Projects using TMRG

TMRG

- Chips already **submitted** and tested:
  - **GBLD10+** – 10 Gbps laser driver
  - **LDQ10** – Quad array laser driver (4x10 Gbps)
  - **VLAD** – Quad array laser driver (4x10 Gbps)
  - **DRAD** – Digital radiation test chip
  - **ePLL-CDR** – PLL/CDR circuit macro block
- (relatively big) Chips to be submitted in following months
  - **lpGBT** – 10 Gbps transceiver
  - **MPA** – Macro Pixel ASIC for CMS tracker
  - **SSA** – Strip Senor ASIC for CMS tracker
  - **SALT** – Silicon ASIC for LHCb Tracking

Before starting ...

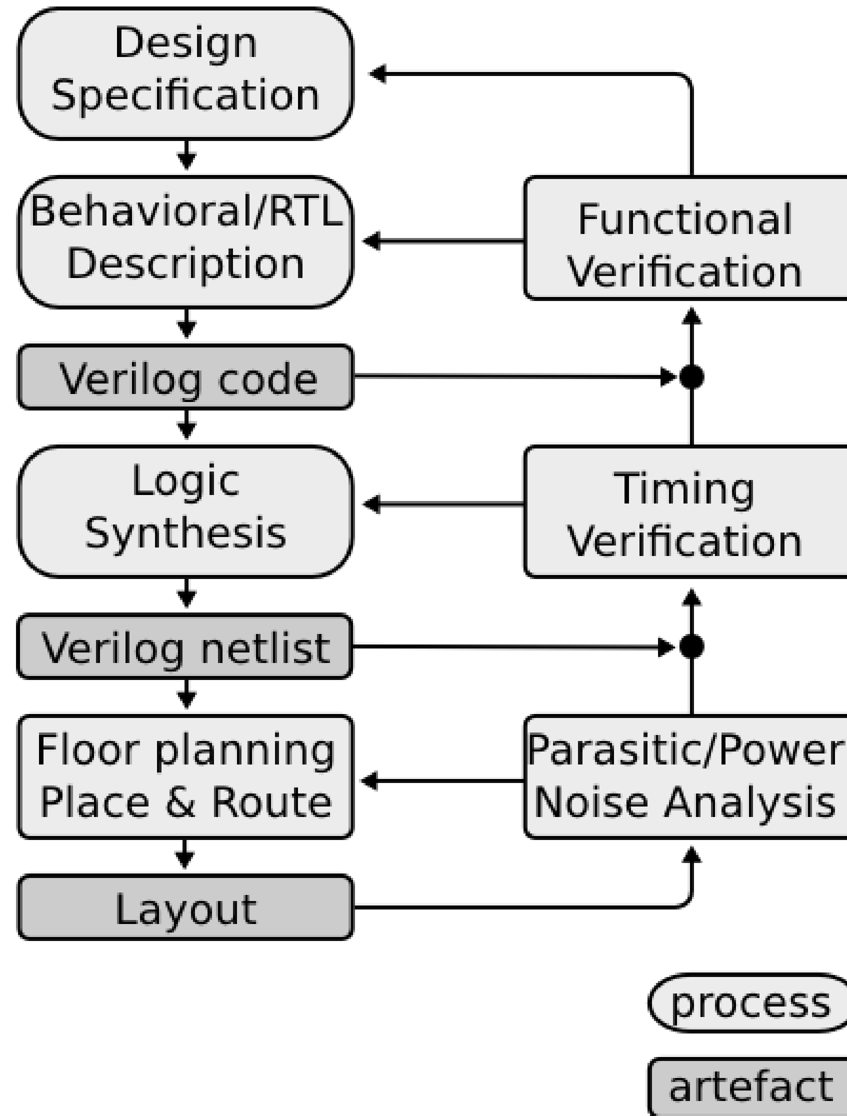
TMRG

The TMRG tool IS NOT a single button solution which will make your CHIP design safe from single event upsets.
You, as a designer, have to know which parts of your circuits should (have to) be protected. The TMRG tool will save you the time needed for copy-pasting your code and will minimize probability that you will forget to change some postfix in your triplicated variable names. It will also simplify the physical implementation and verification process by providing some routines.

The TMRG is open source, however, it **CAN NOT** be made publicly available. The tool can be considered as **dual-use item** as it can be used to produce electronic circuits which are resistant to radiation.

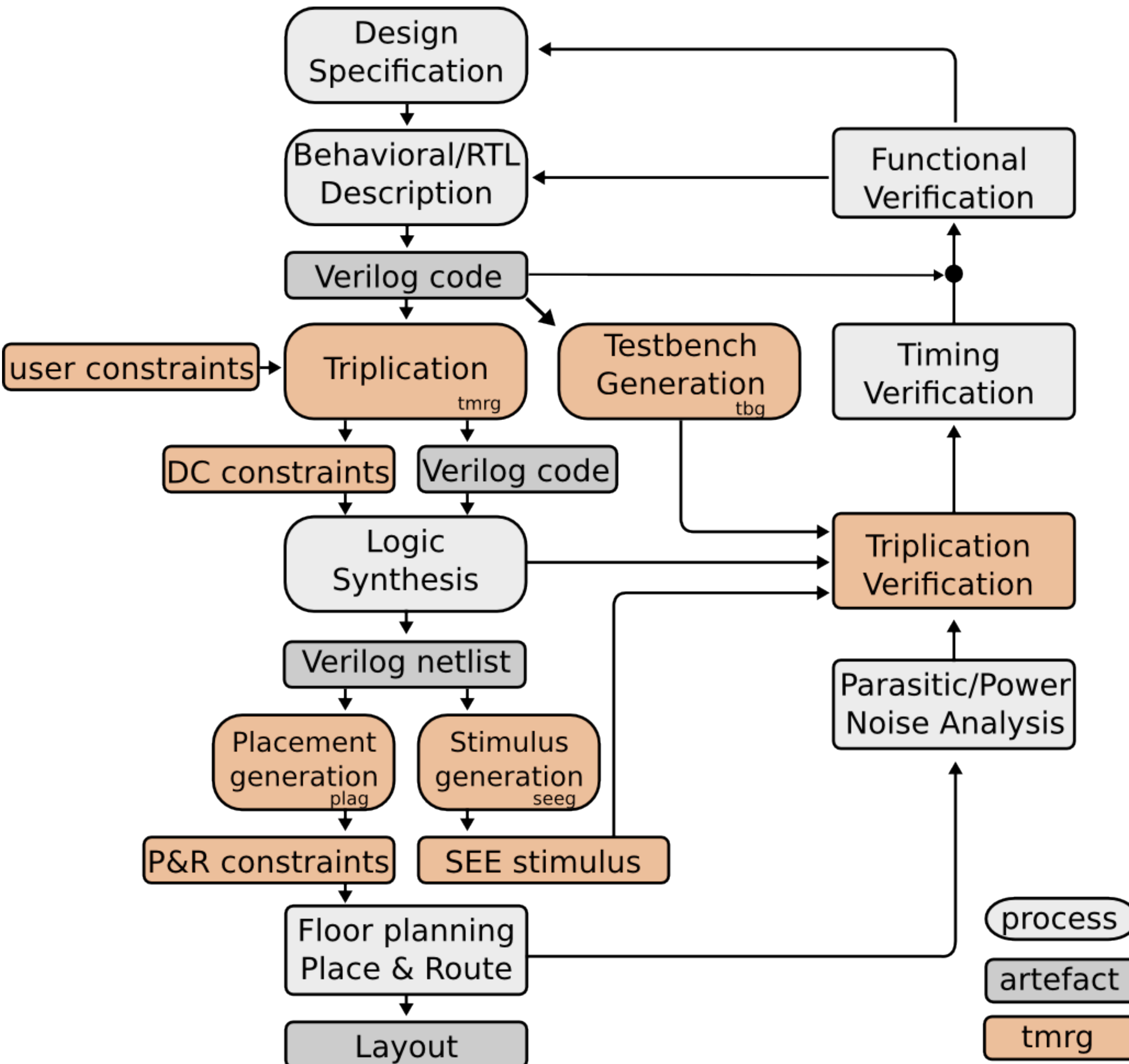
If you find any problem with the tool chain please report it!
Only by having your feedback we will be able to improve the tool chain!

Digital design flow



Digital design flow

TMRG

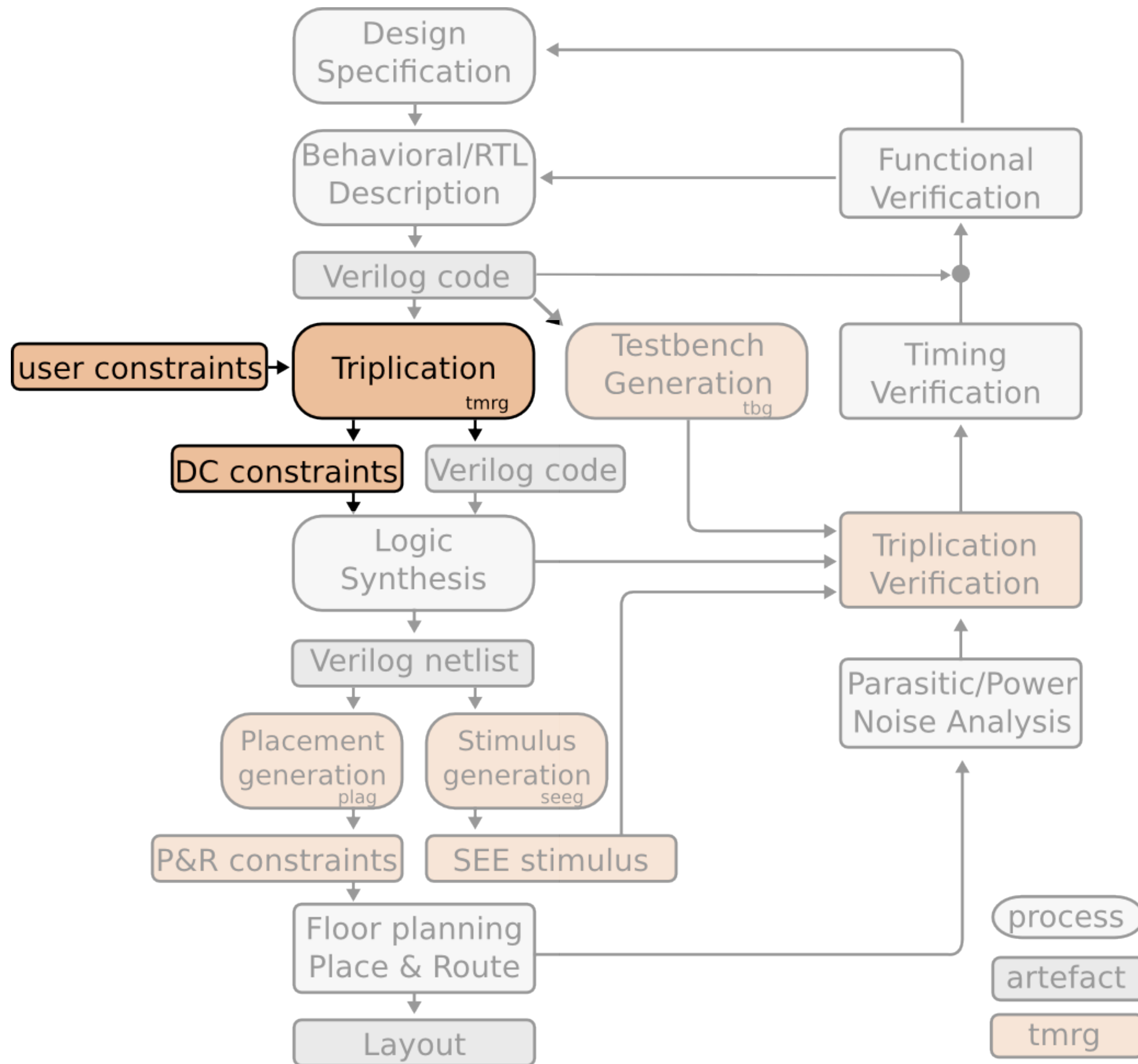


Toolset:

- **tmrg** – triplicates the Verilog code and generates synthesis constrains (for Design Compiler)
- **tbg** – generates generic test bench template (with /without TMR, SEE injection, post synthesis, post PNR)
- **plag** – generates placement directives (for Encounter)
- **seeg** – generates Single Event Effects stimulus to be used for transient simulations

TMRG

triple modular redundancy generator



Constraining the design

TMRG

The TMRG tool:

- lets the designer decide which blocks and signals are to be **triplicated by using TMRG directives** (placed in Verilog code):

```
1 // tmrg triplicate netName
2 // tmrg do_not_triplicate netName
3 // tmrg default [triplicate|do_not_triplicate]
```

- automatizes the **“conversion”** between triplicated and not triplicated signals:
  - if a non triplicated signal is connected to a triplicated signal a **passive fanout** is added
  - if a triplicated signal is connected to a non triplicated signal a **majority voter** is added

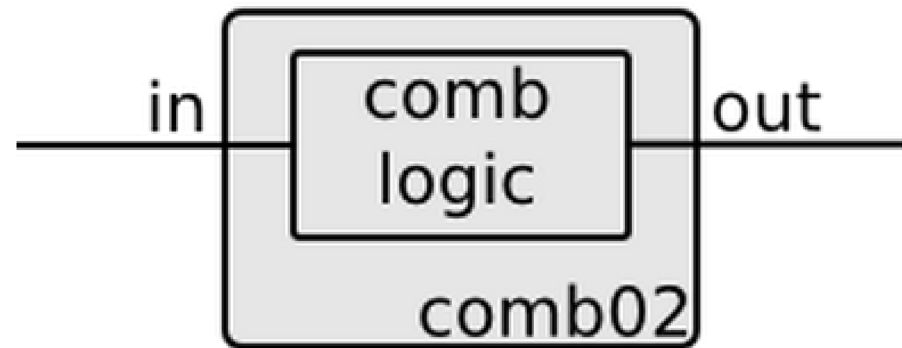
Signal source / Signal sink	non triplicated	triplicated
non triplicated	1 wire connection	fanout
triplicated	majority voter	3 wires connection *)

**) see full TMR option later*

TMRG Example1

Lets consider simple combinatorial module:

```
1 module comb01 (in,out);
2   input in;
3   output out;
4   wire combLogic;
5   assign combLogic = ~in;
6   assign out = combLogic;
7 endmodule
```



The module models an inverter, which contains only one input and one output.

TMRG Example1: Triplicating everything

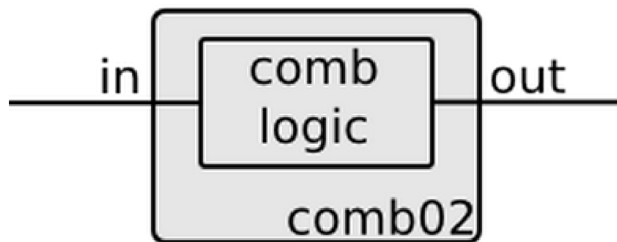
TMRG

```verilog
module comb02 (in,out);
  // tmrg default triplicate
  input in;
  output out;
  wire combLogic;
  assign combLogic = ~in;
  assign out = combLogic;
endmodule
```

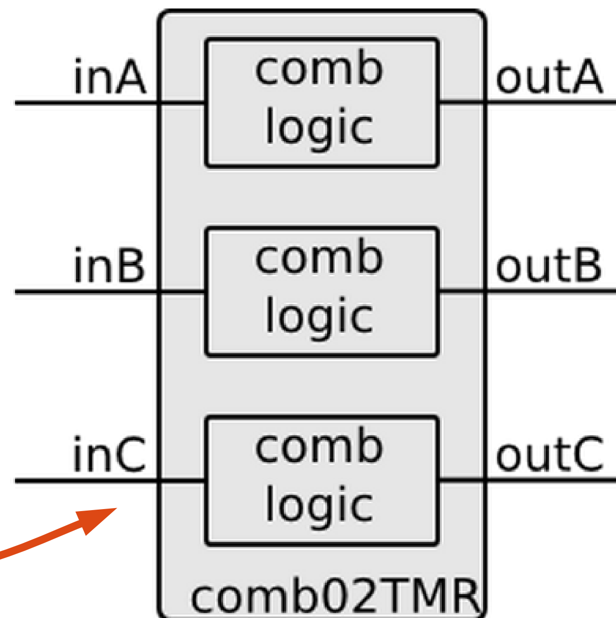
TMRG directive

TMRG

```verilog
module comb02TMR(
  inA,
  inB,
  inC,
  outA,
  outB,
  outC
);
input inA;
input inB;
input inC;
output outA;
output outB;
output outC;
wire combLogicA;
wire combLogicB;
wire combLogicC;
assign combLogicA =  ~inA;
assign combLogicB =  ~inB;
assign combLogicC =  ~inC;
assign outA =  combLogicA;
assign outB =  combLogicB;
assign outC =  combLogicC;
endmodule
```



TMRG



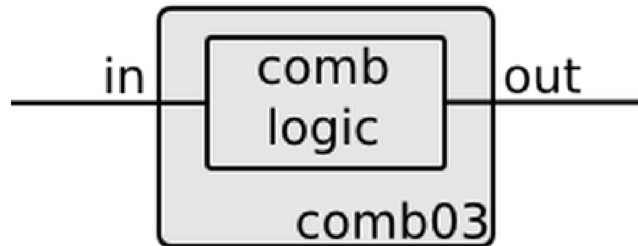
TMRG Example1: Logic and output triplication

TMRG

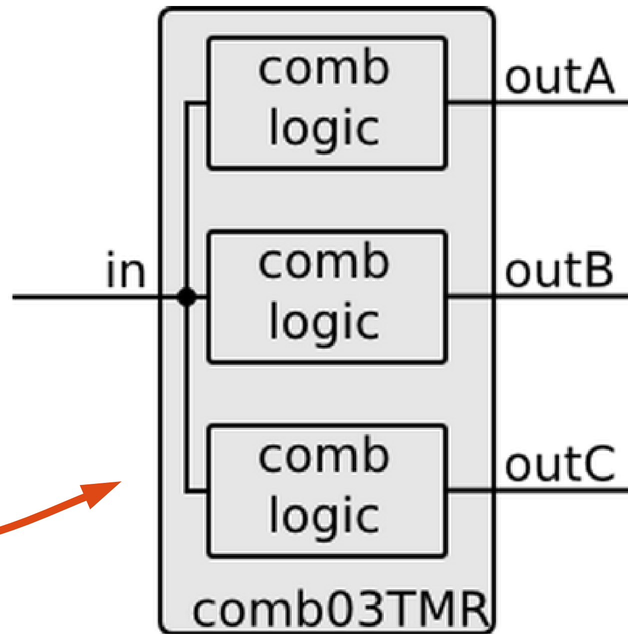
```
1 module comb03 (in,out);
2   // tmrg default triplicate
3   // tmrg do_not_triplicate in
4   input in;
5   output out;
6   wire combLogic;
7   assign combLogic = ~in;
8   assign out = combLogic;
9 endmodule
```

TMRG

Non triplicated signal connected to triplicated signal



TMRG



fanout for **in** net

```
1 module comb03TMR(
2   in,
3   outA,
4   outB,
5   outC
6 );
7 wire inC;
8 wire inB;
9 wire inA;
10 input in;
11 output outA;
12 output outB;
13 output outC;
14 wire combLogicA;
15 wire combLogicB;
16 wire combLogicC;
17 assign combLogicA =  ~inA;
18 assign combLogicB =  ~inB;
19 assign combLogicC =  ~inC;
20 assign outA =  combLogicA;
21 assign outB =  combLogicB;
22 assign outC =  combLogicC;
23
24 fanout inFanout (
25   .in(in),
26   .outA(inA),
27   .outB(inB),
28   .outC(inC)
29 );
30 endmodule
```

TMRG Example1: Input and logic triplication

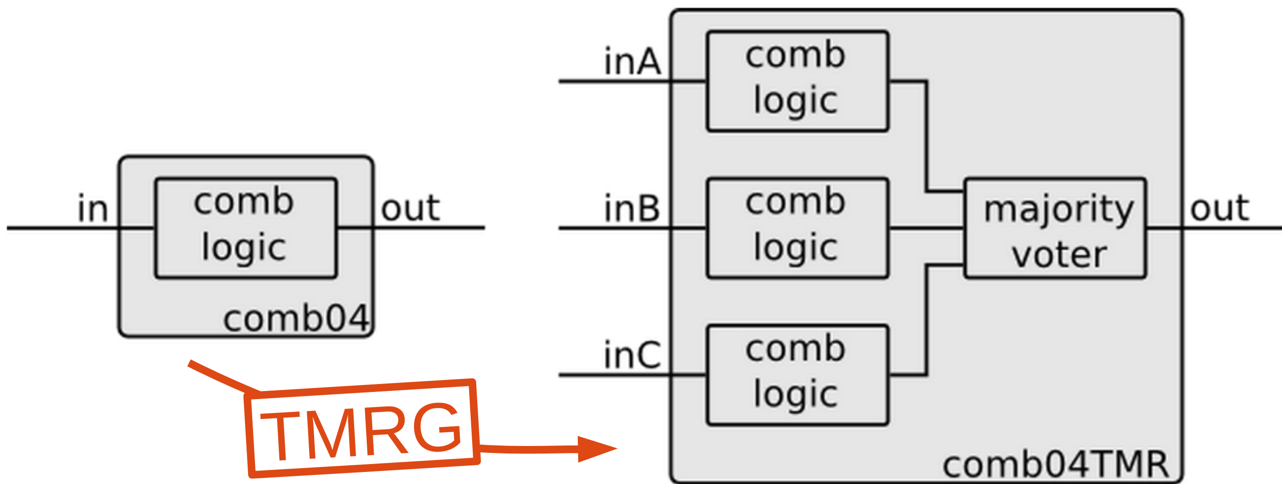
TMRG

```verilog
1 module comb04 (in,out);
2   // tmrg default triplicate
3   // tmrg do_not_triplicate out
4   input in;
5   output out;
6   wire combLogic;
7   assign combLogic = ~in;
8   assign out = combLogic;
9 endmodule
```

TMRG

Triplicated signal connected to non triplicated signal

```verilog
1 module comb04TMR(
2   inA,
3   inB,
4   inC,
5   out
6 );
7 wire combLogic;
8 input inA;
9 input inB;
10 input inC;
11 output out;
12 wire combLogicA;
13 wire combLogicB;
14 wire combLogicC;
15 assign combLogicA =  ~inA;
16 assign combLogicB =  ~inB;
17 assign combLogicC =  ~inC;
18 assign out =  combLogic;
19
20 majorityVoter combLogicVoter (
21   .inA(combLogicA),
22   .inB(combLogicB),
23   .inC(combLogicC),
24   .out(combLogic)
25 );
26 endmodule
```



voter for comb logic net

TMRG Example1: Logic triplication

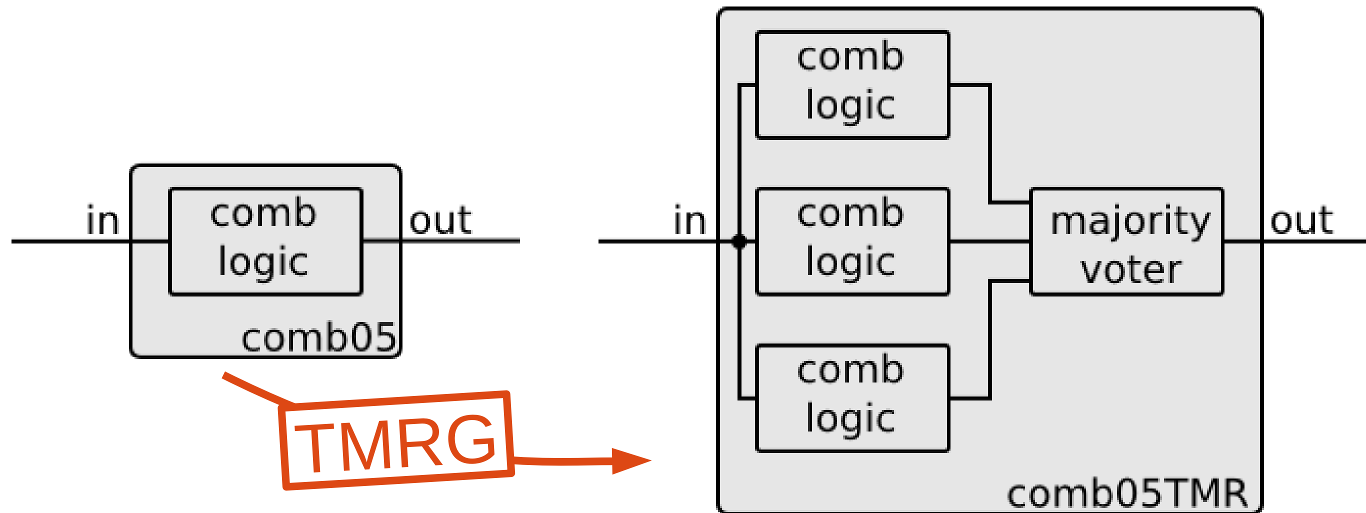
```verilog
1 module comb05 (in,out);
2   // tmrg default do_not_triplicate
3   // tmrg triplicate combLogic
4   input in;
5   output out;
6   wire combLogic;
7   assign combLogic = ~in;
8   assign out = combLogic;
9 endmodule
```

TMRG

Non triplicated signal connected to triplicated signal

Triplicated signal connected to non triplicated signal

```verilog
1 module comb05TMR(
2   in,
3   out
4 );
5 wire inC;
6 wire inB;
7 wire inA;
8 wire combLogic;
9 input in;
10 output out;
11 wire combLogicA;
12 wire combLogicB;
13 wire combLogicC;
14 assign combLogicA =  ~inA;
15 assign combLogicB =  ~inB;
16 assign combLogicC =  ~inC;
17 assign out =  combLogic;
18
19 majorityVoter combLogicVoter (
20   .inA(combLogicA),
21   .inB(combLogicB),
22   .inC(combLogicC),
23   .out(combLogic)
24 );
25
26 fanout inFanout (
27   .in(in),
28   .outA(inA),
29   .outB(inB),
30   .outC(inC)
31 );
32 endmodule
```



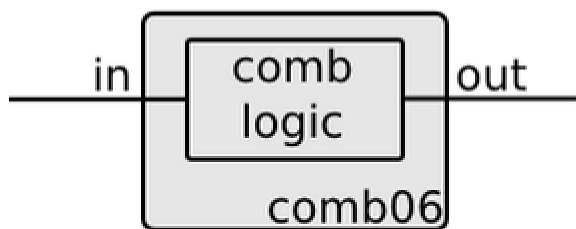
TMRG Example1: Input and output triplication

TMRG

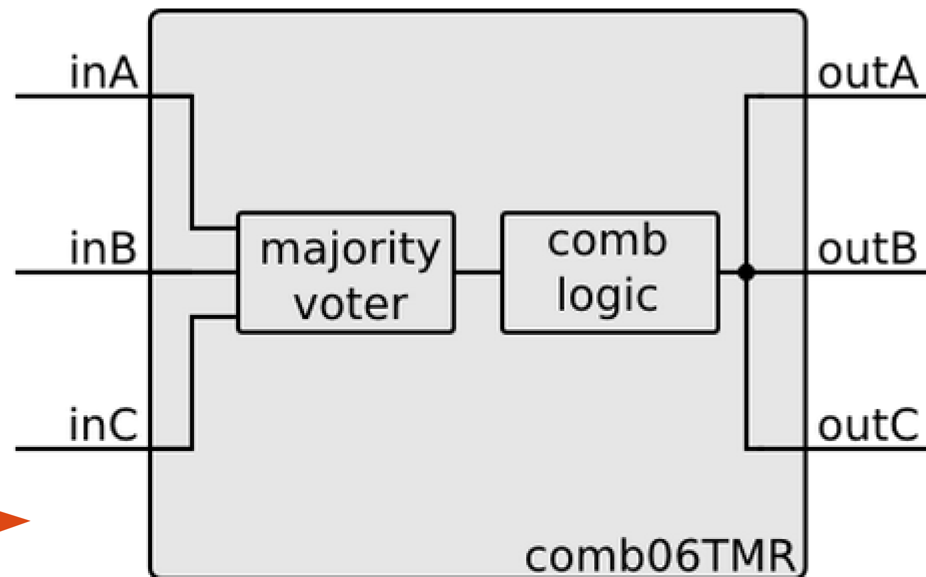
```
1 module comb06 (in,out);
2   // tmrg default triplicate
3   // tmrg do_not_triplicate combLogic
4   input in;
5   output out;
6   wire combLogic;
7   assign combLogic = ~in;
8   assign out = combLogic;
9 endmodule
```

TMRG

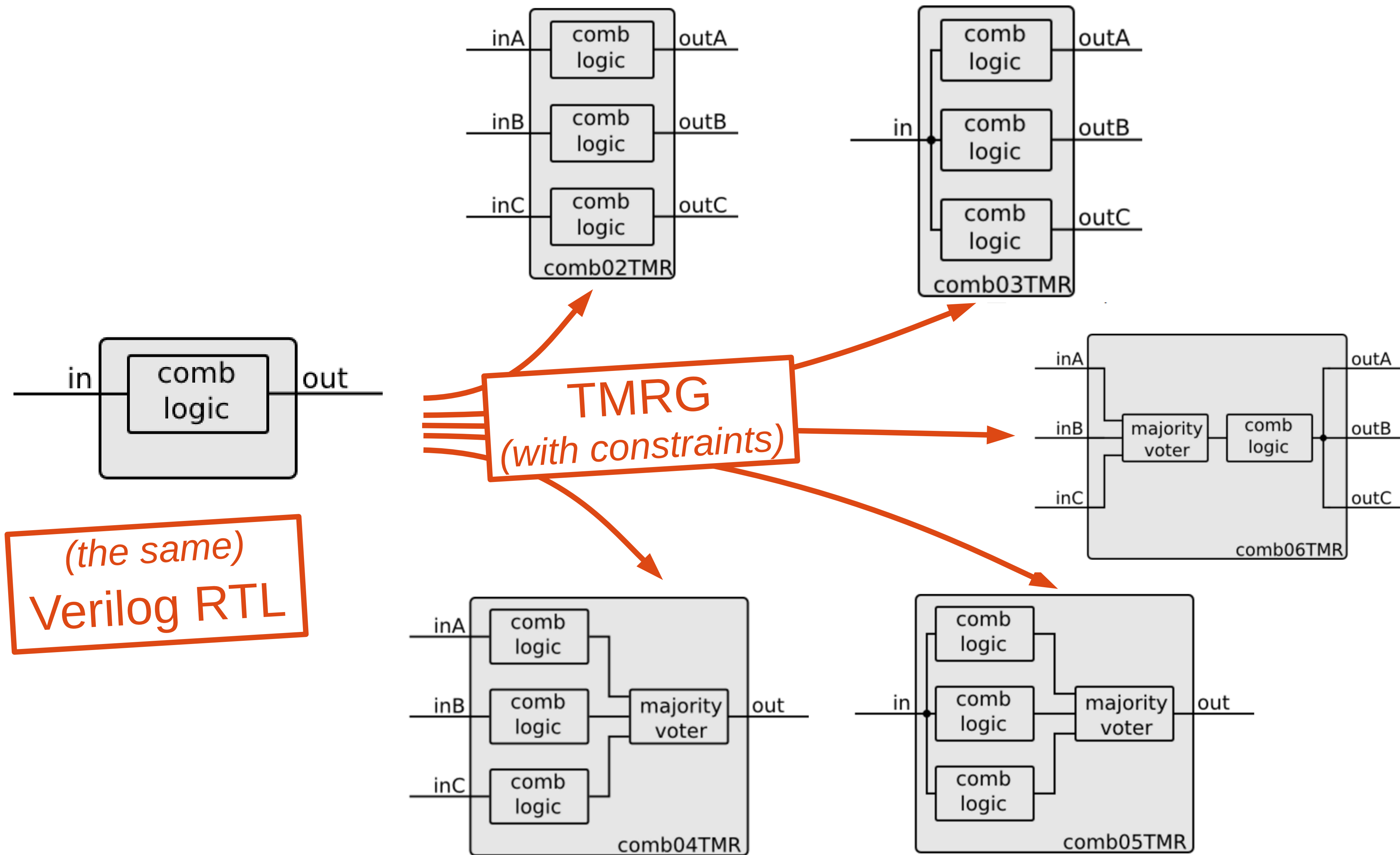
```
1 module comb06TMR(
2   inA,
3   inB,
4   inC,
5   outA,
6   outB,
7   outC
8 );
9 wire combLogicC;
10 wire combLogicB;
11 wire combLogicA;
12 wire in;
13 input inA;
14 input inB;
15 input inC;
16 output outA;
17 output outB;
18 output outC;
19 wire combLogic;
20 assign combLogic =  ~in;
21 assign outA =  combLogicA;
22 assign outB =  combLogicB;
23 assign outC =  combLogicC;
24
25 majorityVoter inVoter (
26   .inA(inA),
27   .inB(inB),
28   .inC(inC),
29   .out(in)
30 );
31
32 fanout combLogicFanout (
33   .in(combLogic),
34   .outA(combLogicA),
35   .outB(combLogicB),
36   .outC(combLogicC)
37 );
38 endmodule
```



TMRG



TMRG Example: Summary



TMRG tool behavior can be controlled by TMRG constrains

Majority voter, fanout ?

TMRG

Some definitions:

common/voter.v

```
1 module majorityVoter (inA, inB, inC, out, tmrErr);
2   parameter WIDTH = 1;
3   input   [(WIDTH-1):0]   inA, inB, inC;
4   output  [(WIDTH-1):0]   out;
5   output                  tmrErr;
6   reg                     tmrErr;
7   assign out = (inA&inB) | (inA&inC) | (inB&inC);
8   always @(inA or inB or inC)
9   begin
10    if (inA!=inB || inA!=inC || inB!=inC)
11      tmrErr = 1;
12    else
13      tmrErr = 0;
14  end
15 endmodule
```

common/fanout.v

```
1 module fanout (in, outA, outB, outC);
2   parameter WIDTH = 1;
3   input   [(WIDTH-1):0]   in;
4   output  [(WIDTH-1):0]   outA,outB,outC;
5   assign outA=in;
6   assign outB=in;
7   assign outC=in;
8 endmodule
```

If needed this definitions are added to the output file, can also be replaced by user defined modules.

FSM Example: triplication without voting

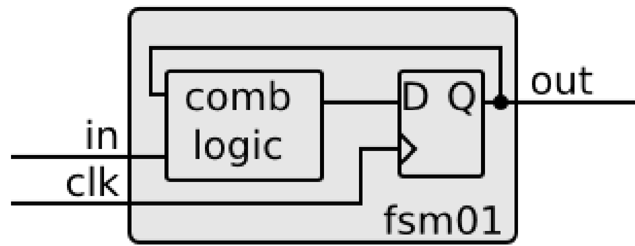
TMRG

```
 1 module fsm01 (in,out,clk);
 2   // tmrg default triplicate
 3   input in;
 4   input clk;
 5   output out;
 6   reg state;
 7   reg stateNext;
 8   assign out=state;
 9
10   always @(posedge clk)
11     state <= stateNext;
12
13   always @(state or in)
14     stateNext = in ^ state;
15 endmodule
```

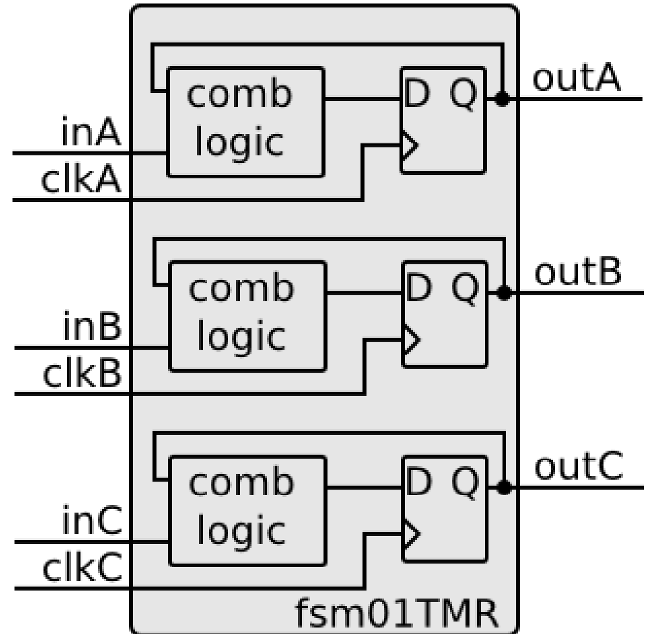
flip flop

comb logic

We know that this is not what we want ...



TMRG




```
 1 module fsm01TMR(
 2   inA,
 3   inB,
 4   inC,
 5   outA,
 6   outB,
 7   outC,
 8   clkA,
 9   clkB,
10   clkC
11 );
12 input inA;
13 input inB;
14 input inC;
15 input clkA;
16 input clkB;
17 input clkC;
18 output outA;
19 output outB;
20 output outC;
21 reg stateA;
22 reg stateB;
23 reg stateC;
24 reg stateNextA;
25 reg stateNextB;
26 reg stateNextC;
27 assign outA =  stateA;
28 assign outB =  stateB;
29 assign outC =  stateC;
30
31 always @(posedge clkA)
32   stateA <= stateNextA;
33
34 always @(posedge clkB)
35   stateB <= stateNextB;
36
37 always @(posedge clkC)
38   stateC <= stateNextC;
39
40 always @(stateA or inA)
41   stateNextA =  inA^stateA;
42
43 always @(stateB or inB)
44   stateNextB =  inB^stateB;
45
46 always @(stateC or inC)
47   stateNextC =  inC^stateC;
48 endmodule
```

If an error occurs in one branch, it will propagate along the branch. If there is no repair mechanism, after the first error the effective cross section is doubled with respect to the non triplicated circuit. In order to eliminate this problem, a voted feedback is needed!

FSM Example: triplicating only the register

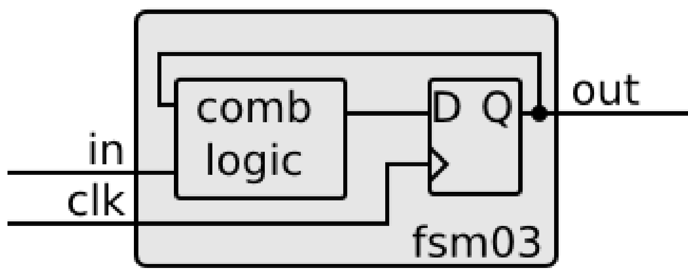
TMRG

```
1 module fsm03 (in,out,clk);
2   // tmrg default do_not_triplicate
3   // tmrg triplicate state
4   input in;
5   input clk;
6   output out;
7   reg state;
8   reg stateNext;
9   assign out=state;
10
11  always @(posedge clk)
12    state <= stateNext;
13
14  always @(state or in)
15    stateNext = in ^ state;
16
17 endmodule
```

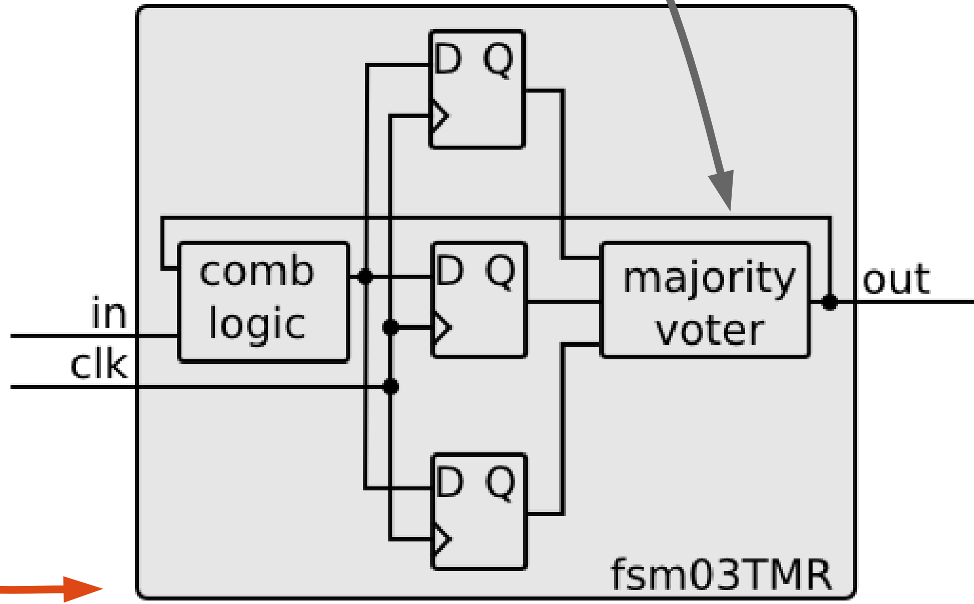
Triplicate state

Majority Voter added automatically (conversion from triplicated to non triplicated signal)

```
1 module fsm03TMR(
2   in,
3   out,
4   clk
5 );
6 wire stateNextC;
7 wire stateNextB;
8 wire stateNextA;
9 wire clkC;
10 wire clkB;
11 wire clkA;
12 wire state;
13 input in;
14 input clk;
15 output out;
16 reg stateA;
17 reg stateB;
18 reg stateC;
19 reg stateNext;
20 assign out =  state;
21
22 always @(posedge clkA)
23   stateA <= stateNextA;
24
25 always @(posedge clkB)
26   stateB <= stateNextB;
27
28 always @(posedge clkC)
29   stateC <= stateNextC;
30
31 always @(state or in)
32   stateNext =  in^state;
33
34 majorityVoter stateVoter (
35   .inA(stateA),
36   .inB(stateB),
37   .inC(stateC),
38   .out(state)
39 );
40
41 fanout clkFanout (
42   .in(clk),
43   .outA(clkA),
44   .outB(clkB),
45   .outC(clkC)
46 );
47
48 fanout stateNextFanout (
49   .in(stateNext),
50   .outA(stateNextA),
51   .outB(stateNextB),
52   .outC(stateNextC)
53 );
54 endmodule
```



TMRG



FSM Example: triplicating the register and clock skew

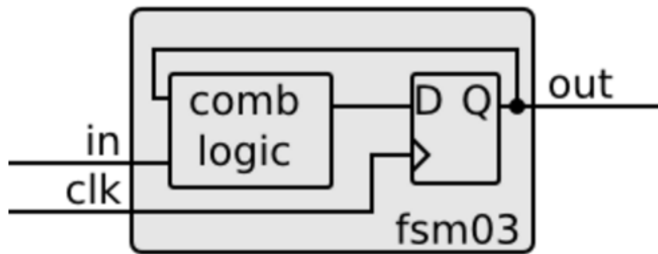
TMRG

```verilog
1  module fsm03 (in,out,clk);
2    // tmrg default do not triplicate
3    // tmrg triplicate state
4    // tmrg triplicate clk
5    input in;
6    input clk;
7    output out;
8    reg state;
9    reg stateNext;
10   assign out=state;
11
12   always @(posedge clk)
13     state <= stateNext;
14
15   always @(state or in)
16     stateNext = in ^ state;
17
18 endmodule
```

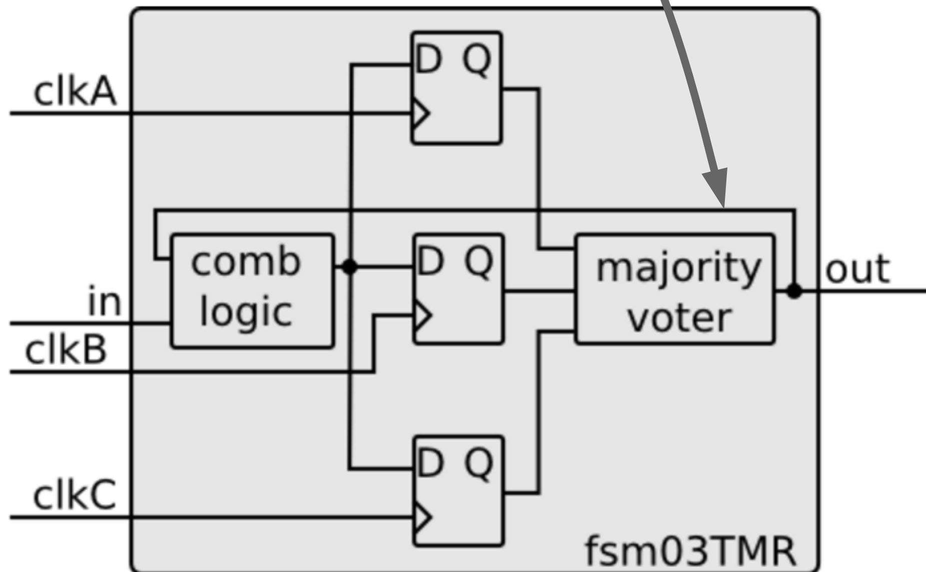
Triplicate state
Triplicate clk

TMRG

Majority Voter added automatically (conversion from triplicated to non triplicated signal)



TMRG




```verilog
1  module fsm03TMR(
2    in,
3    out,
4    clkA,
5    clkB,
6    clkC
7  );
8  wire stateNextC;
9  wire stateNextB;
10 wire stateNextA;
11 wire state;
12 input in;
13 input clkA;
14 input clkB;
15 input clkC;
16 output out;
17 reg stateA;
18 reg stateB;
19 reg stateC;
20 reg stateNext;
21 assign out =  state;
22
23 always @(posedge clkA)
24   stateA <= stateNextA;
25
26 always @(posedge clkB)
27   stateB <= stateNextB;
28
29 always @(posedge clkC)
30   stateC <= stateNextC;
31
32 always @(state or in)
33   stateNext =  in^state;
34
35 majorityVoter stateVoter (
36     .inA(stateA),
37     .inB(stateB),
38     .inC(stateC),
39     .out(state)
40     );
41
42 fanout stateNextFanout (
43     .in(stateNext),
44     .outA(stateNextA),
45     .outB(stateNextB),
46     .outC(stateNextC)
47     );
48 endmodule
```

Introducing full TMR

TMRG

To generate **full TMR (3 interconnected majority voters)** a net declaration with a specific name (**Voted** postfix) has to be used:

```
1 wire netVoted = net;
```

This syntax ensures that non triplicated Verilog code can be simulated and/or synthesized.

Full TMR: Voting triplicated signals

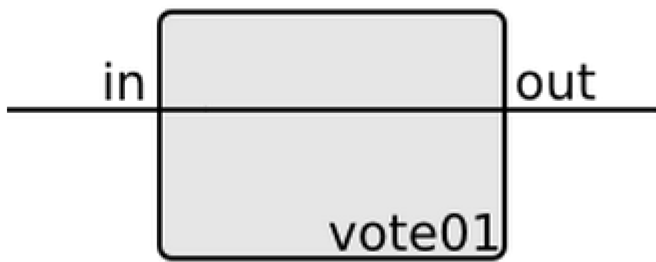
TMRG

```
1 module vote01 (in,out);
2   // tmrg default triplicate
3   input in;
4   output out;
5   wire inVoted = in;
6   assign out = inVoted;
7 endmodule
```

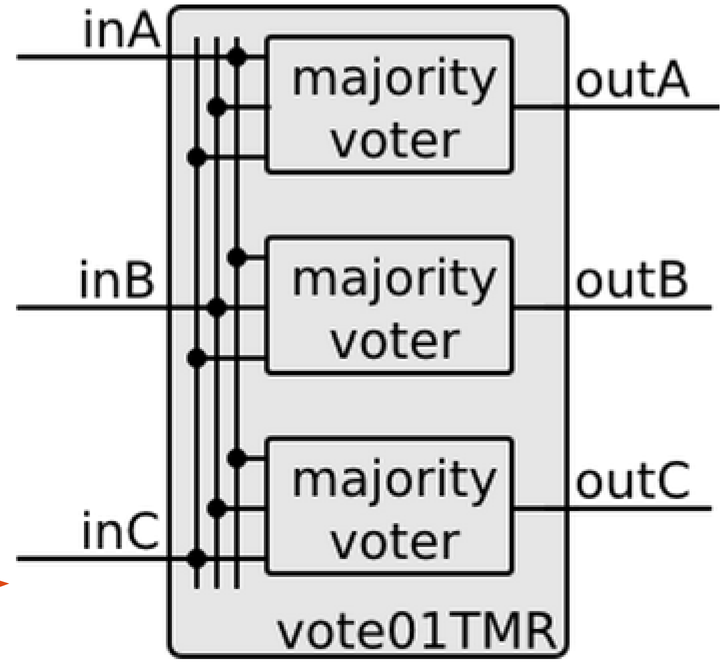
Insert 3 voters

TMRG

```
1 module vote01TMR(
2   inA,
3   inB,
4   inC,
5   outA,
6   outB,
7   outC
8 );
9 input inA;
10 input inB;
11 input inC;
12 output outA;
13 output outB;
14 output outC;
15 assign outA =  inVotedA;
16 assign outB =  inVotedB;
17 assign outC =  inVotedC;
18
19 majorityVoter inVoterA (
20   .inA(inA),
21   .inB(inB),
22   .inC(inC),
23   .out(inVotedA)
24 );
25
26 majorityVoter inVoterB (
27   .inA(inA),
28   .inB(inB),
29   .inC(inC),
30   .out(inVotedB)
31 );
32
33 majorityVoter inVoterC (
34   .inA(inA),
35   .inB(inB),
36   .inC(inC),
37   .out(inVotedC)
38 );
39 endmodule
```



TMRG



Full TMR: logic triplication and voting

TMRG

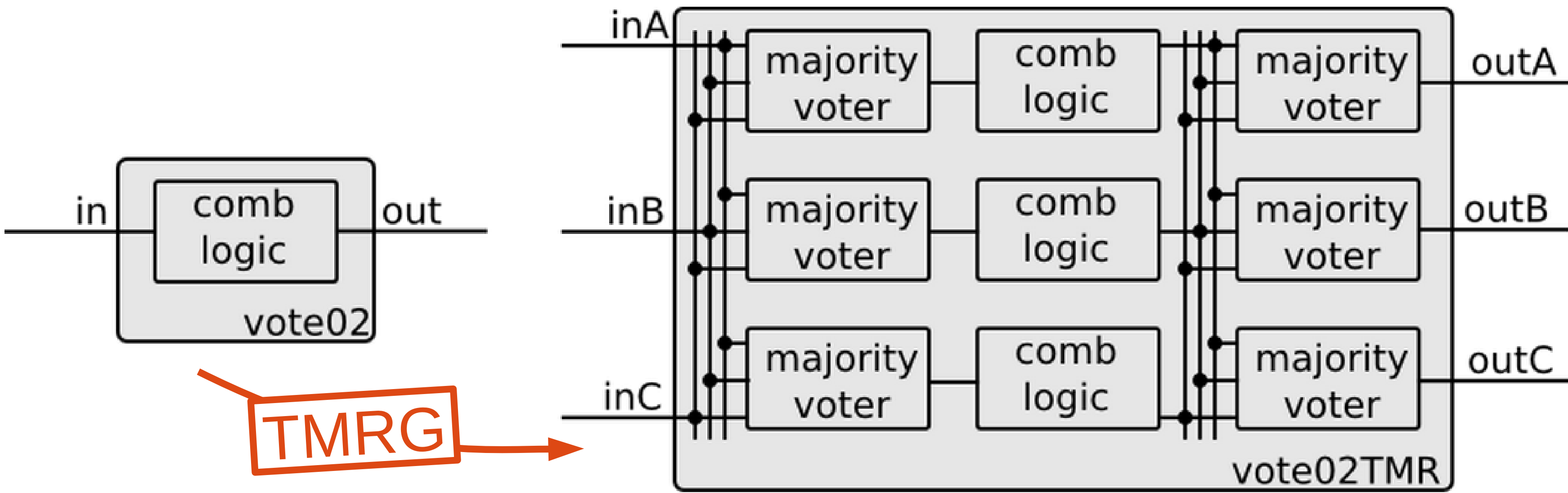
```
1 module vote02 (in,out);
2   input in;
3   output out;
4   wire combLogic;
5   wire inVoted = in;
6   assign combLogic = ~inVoted;
7   wire combLogicVoted = combLogic;
8   assign out = combLogicVoted;
9 endmodule
```

TMRG

Insert 3 voters for input signals

Insert 3 voters for output signals

```
1 module vote02TMR(
2   inA,
3   inB,
4   inC,
5   outA,
6   outB,
7   outC
8 );
9 input inA;
10 input inB;
11 input inC;
12 output outA;
13 output outB;
14 output outC;
15 wire combLogicA;
16 wire combLogicB;
17 wire combLogicC;
18 assign combLogicA =  ~inVotedA;
19 assign combLogicB =  ~inVotedB;
20 assign combLogicC =  ~inVotedC;
21 assign outA =  combLogicVotedA;
22 assign outB =  combLogicVotedB;
23 assign outC =  combLogicVotedC;
24
25 majorityVoter inVoterA (
26   .inA(inA),
27   .inB(inB),
28   .inC(inC),
29   .out(inVotedA)
30 );
31
32 majorityVoter combLogicVoterA (
33   .inA(combLogicA),
34   .inB(combLogicB),
35   .inC(combLogicC),
36   .out(combLogicVotedA)
37 );
38
39 majorityVoter combLogicVoterB (
40   .inA(combLogicA),
41   .inB(combLogicB),
42   .inC(combLogicC),
43   .out(combLogicVotedB)
44 );
45
46 majorityVoter inVoterB (
47   .inA(inA),
48   .inB(inB),
49   .inC(inC),
50   .out(inVotedB)
51 );
52
53 majorityVoter inVoterC (
54   .inA(inA),
55   .inB(inB),
56   .inC(inC),
57   .out(inVotedC)
58 );
59
60 majorityVoter combLogicVoterC (
61   .inA(combLogicA),
62   .inB(combLogicB),
63   .inC(combLogicC),
64   .out(combLogicVotedC)
65 );
66 endmodule
```



Do you want to have voter before and after the logic? Probably not … but you CAN.

FSM Example: triplication and voting

TMRG

```
 1 module fsm02 (in,out,clk);
 2   // tmrg default triplicate
 3   input in;
 4   input clk;
 5   output out;
 6   reg state;
 7   reg stateNext;
 8   wire stateNextVoted=stateNext;
 9   assign out=state;
10
11   always @(posedge clk)
12     state <= stateNextVoted;
13
14   always @(state or in)
15     stateNext = in ^ state;
16
17 endmodule
```

Insert 3 voters

flip flop

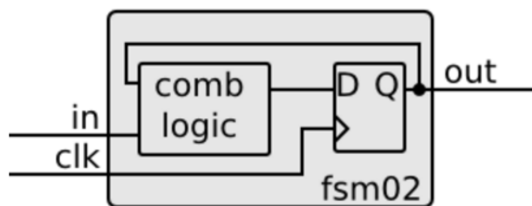
comb logic

TMRG

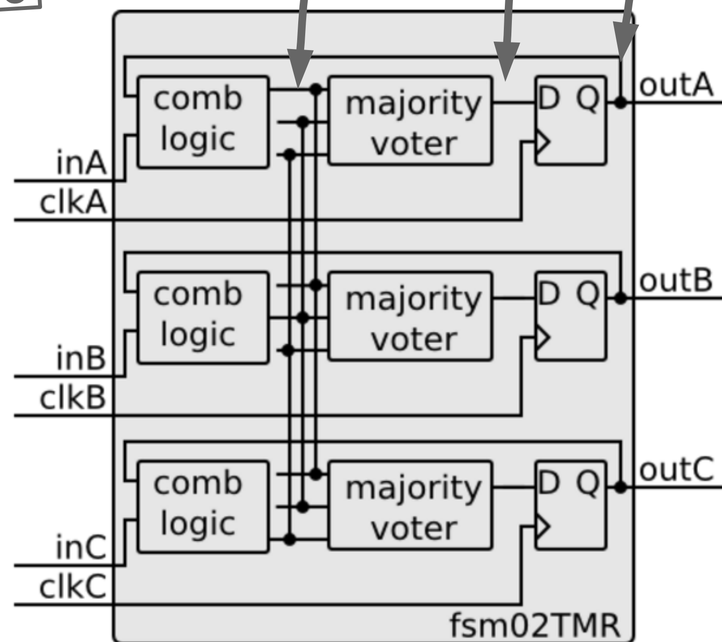
nextState

nextStateVoted

state



TMRG




```
 1 module fsm02TMR(
 2   inA,   inB,   inC,
 3   outA,  outB,  outC,
 4   clkA,  clkB,  clkC
 5 );
 6 input inA,inB,inC;
 7 input clkA,clkB,clkC;
 8 output outA,outB,outC;
 9 reg stateA,stateB,stateC;
10 reg stateNextA,stateNextB,stateNextC;
11
12 assign outA =  stateA;
13 assign outB =  stateB;
14 assign outC =  stateC;
15
16 always @(posedge clkA)
17   stateA <= stateNextVotedA;
18
19 always @(posedge clkB)
20   stateB <= stateNextVotedB;
21
22 always @(posedge clkC)
23   stateC <= stateNextVotedC;
24
25 always @(stateA or inA)
26   stateNextA =  inA^stateA;
27
28 always @(stateB or inB)
29   stateNextB =  inB^stateB;
30
31 always @(stateC or inC)
32   stateNextC =  inC^stateC;
33
34 majorityVoter stateNextVoterA (
35   .inA(stateNextA),
36   .inB(stateNextB),
37   .inC(stateNextC),
38   .out(stateNextVotedA)
39 );
40
41 majorityVoter stateNextVoterB (
42   .inA(stateNextA),
43   .inB(stateNextB),
44   .inC(stateNextC),
45   .out(stateNextVotedB)
46 );
47
48 majorityVoter stateNextVoterC (
49   .inA(stateNextA),
50   .inB(stateNextB),
51   .inC(stateNextC),
52   .out(stateNextVotedC)
53 );
54 endmodule
```

Module instantiations

TMRG

General concepts related to module instantiation triplication:

- **Only named connections** are supported for module instantiation!
- **All modules must be known at the time of triplication.**
- If a module is **not be triplicated internally** (e.g. library cell, analog macro cell) one has to add directive **`do_not_touch`** in the module body.
- For all other modules (not from library and not having `do_not_touch` constrain):
  - **triplication is always done inside the module,**
  - a new (triplicated) module has a **TMR postfix** appended to the name,
  - I/O names may change.

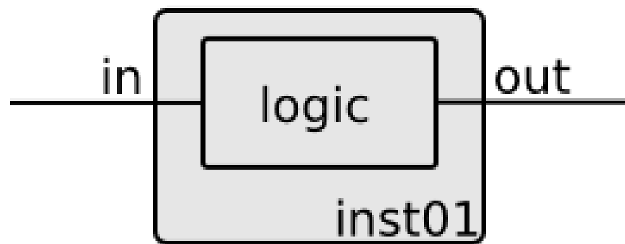
Triplicating a fixed macro cell

TMRG

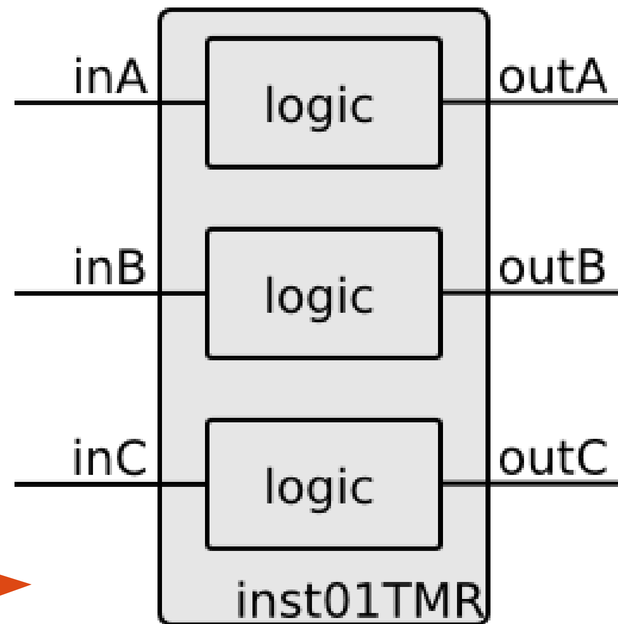
```
1  module mlogic(I,ZN);
2    // tmrg do_not_touch
3    input I;
4    output ZN;
5    assign ZN=~I;
6  endmodule
7
8  module inst01 (in,out);
9    // tmrg default triplicate
10   input in;
11   output out;
12   mlogic logic01(.I(in),.ZN(out));
13 endmodule
```

TMRG

```
1  module mlogic(
2    I,
3    ZN
4  );
5  input I;
6  output ZN;
7  assign ZN =  ~I;
8  endmodule
9  module inst01TMR(
10   inA,
11   inB,
12   inC,
13   outA,
14   outB,
15   outC
16 );
17 input inA;
18 input inB;
19 input inC;
20 output outA;
21 output outB;
22 output outC;
23
24 mlogic logic01A (
25   .I(inA),
26   .ZN(outA)
27 );
28
29 mlogic logic01B (
30   .I(inB),
31   .ZN(outB)
32 );
33
34 mlogic logic01C (
35   .I(inC),
36   .ZN(outC)
37 );
38 endmodule
```



TMRG



Not triplicating a fixed macro cell

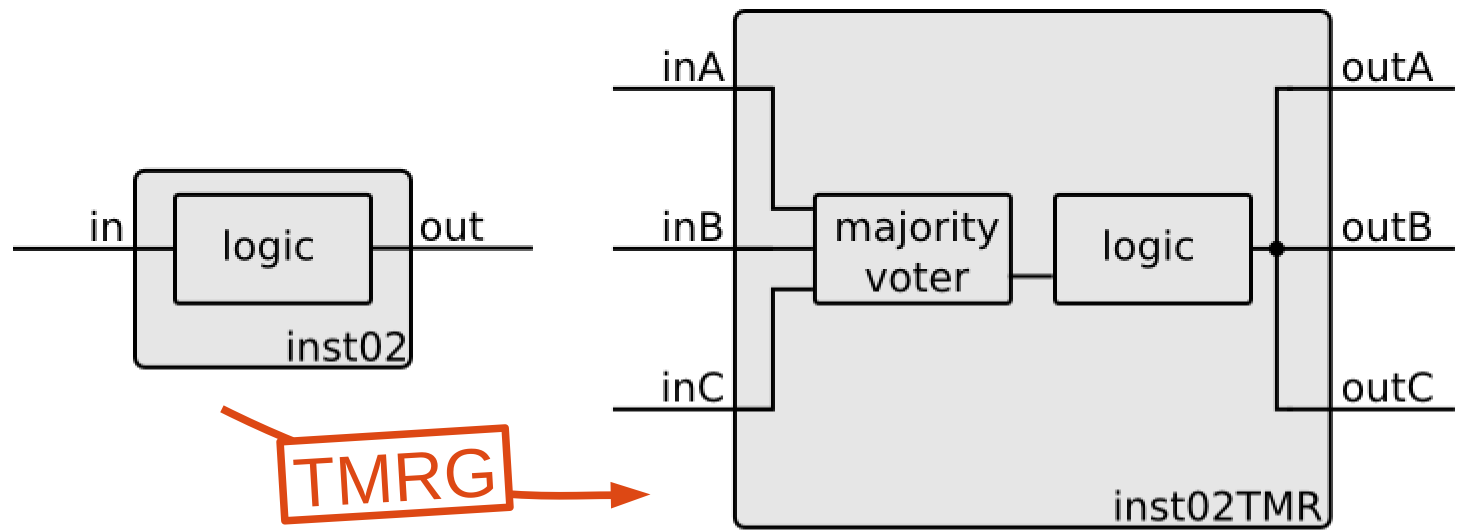
TMRG

```
1  module mlogic(I,ZN);
2    // tmrg do_not_touch
3    input I;
4    output ZN;
5    assign ZN=~I;
6  endmodule
7
8  module inst02 (in,out);
9    // tmrg default triplicate
10   // tmrg do_not_triplicate logic01
11   input in;
12   output out;
13   mlogic logic01(.I(in),.ZN(out));
14 endmodule
```

Do not triplicate logic01 instance

```
1  module mlogic(
2    I,
3    ZN
4  );
5  input I;
6  output ZN;
7  assign ZN =  ~I;
8  endmodule
9  module inst02TMR(
10   inA,
11   inB,
12   inC,
13   outA,
14   outB,
15   outC
16 );
17 wire out;
18 wire in;
19 input inA;
20 input inB;
21 input inC;
22 output outA;
23 output outB;
24 output outC;
25
26 mlogic logic01 (
27   .I(in),
28   .ZN(out)
29 );
30
31 majorityVoter inVoter (
32   .inA(inA),
33   .inB(inB),
34   .inC(inC),
35   .out(in)
36 );
37
38 fanout outFanout (
39   .in(out),
40   .outA(outA),
41   .outB(outB),
42   .outC(outC)
43 );
44 endmodule
```

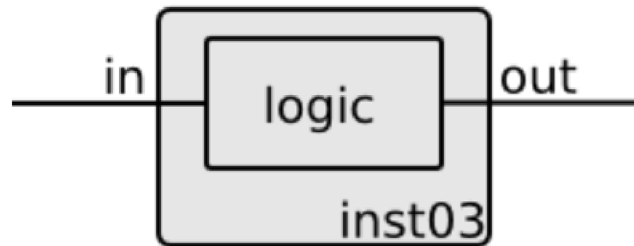
Majority Voter and fanout added automatically
(conversion between triplicated and non triplicated signals)



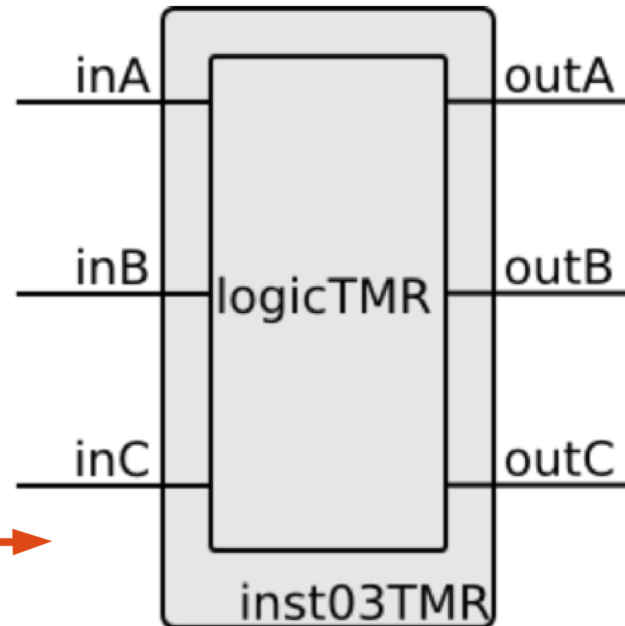
Triplicating user's modules

```
 1 module mlogic (I,ZN);
 2   // tmrg default triplicate
 3   input  I;
 4   output ZN;
 5   assign ZN=~I;
 6 endmodule
 7
 8 module inst03 (in,out);
 9   // tmrg default triplicate
10   input in;
11   output out;
12   mlogic logic01(.I(in),.ZN(out));
13 endmodule
```

TMRG



TMRG




```
 1 module mlogicTMR(
 2   IA,
 3   IB,
 4   IC,
 5   ZNA,
 6   ZNB,
 7   ZNC
 8 );
 9 input IA;
10 input IB;
11 input IC;
12 output ZNA;
13 output ZNB;
14 output ZNC;
15 assign ZNA =  ~IA;
16 assign ZNB =  ~IB;
17 assign ZNC =  ~IC;
18 endmodule
19 module inst03TMR(
20   inA,
21   inB,
22   inC,
23   outA,
24   outB,
25   outC
26 );
27 input inA;
28 input inB;
29 input inC;
30 output outA;
31 output outB;
32 output outC;
33
34 mlogicTMR logic01 (
35   .IA(inA),
36   .IB(inB),
37   .IC(inC),
38   .ZNA(outA),
39   .ZNB(outB),
40   .ZNC(outC)
41 );
42 endmodule
```

When the module being instantiated is a subject of triplication, only connections are modified and voters and fanouts are added if necessary.

Triplication brings new features - tmrError

TMRG

- **It may be desirable to know if one of the triplicated signals is different from the other two** *(whether an single event upset has happened or not)*
- The TMRG tool always generates:
  - **tmrError** output associated with each voter. For a signal mem the voter can look like:
  - Combination (OR) of all error signals inside given module

```
1 majorityVoter memVoter (
2     .inA(memA),
3     .inB(memB),
4     .inC(memC),
5     .out(mem),
6     .tmrErr(memTmrError)
7     );
```

```
assign tmrErrorC =  dff01tmrErrorC|dff02tmrErrorC|in3TmrErrorC;
```

- To make use of the signal (particular or global) it is enough to make a declaration:

```
1 wire memTmrError=1'b0;
2 wire tmrError=1'b0;
```

This definition will be removed by the TMRG tool and the wire will be connected directly to the error output of the voter. By declaring tmrError the designer gains access to the signal and can implement the required functionality. Moreover, assigning zero value ensures that the non triplicated circuit is not affected and can be simulated.

- If user does not use the tmrError functionality it will be optimized out by the synthesizer

tmrError Example

TMRG

```
1 module ff(input clk,input in, output out, input load);
2   // tmrg default do_not_triplicate
3   // tmrg triplicate mem
4   reg mem;
5   wire tmrError=1'b0;
6   wire loadInt=load|tmrError;
7   wire memNext=(load)? in : out;
8   assign out=mem;
9   always @(posedge clk)
10    if (loadInt)
11      mem <= memNext;
12 endmodule
```

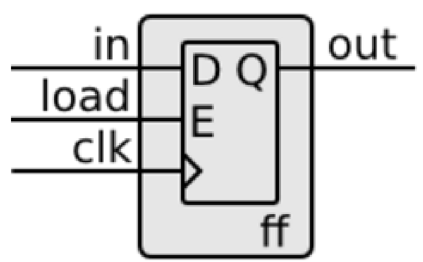
Definition for tmrError

TMRG

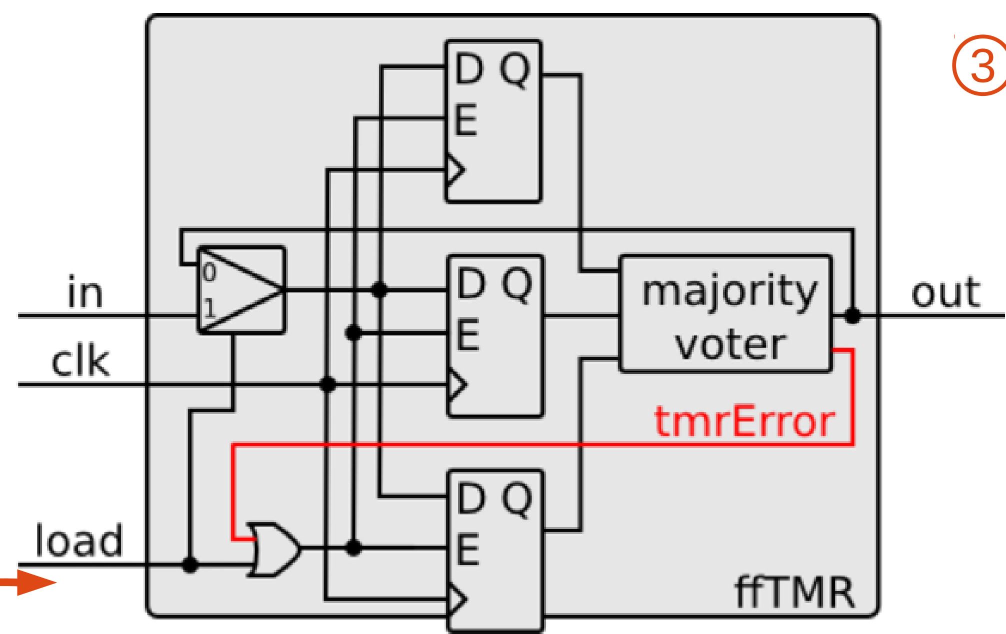
```
1 module ffTMR(
2   input  clk,
3   input  in,
4   output  out,
5   input  load
6 );
7 wire loadIntA,loadIntB,loadIntC;
8 wire memNextA,memNextB,memNextC;
9 wire clkA,clkB,clkC;
10 wire tmrError;
11 wor memTmrError;
12 wire mem;
13 reg  memA,memB,memC;
14 wire loadInt =  load|tmrError;
15 wire memNext =  (load) ? in : out;
16 assign out =  mem;
17
18 always @( posedge clkA )
19   if (loadIntA)
20     memA <= memNextA;
21
22 always @( posedge clkB )
23   if (loadIntB)
24     memB <= memNextB;
25
26 always @( posedge clkC )
27   if (loadIntC)
28     memC <= memNextC;
29
30 majorityVoter memVoter (
31     .inA(memA),
32     .inB(memB),
33     .inC(memC),
34     .out(mem),
35     .tmrErr(memTmrError)
36     );
37 assign tmrError =  memTmrError;
38 [...]
```

Assigning zero value to tmrError ensures that the non triplicated circuit is not affected and can be simulated.

③ ② ①



TMRG



TmrError can be used to implement SEU counter!

Other features

TMRG

- Accessing individual signals from a triplicated bus
(e.g. power on reset monitoring)
- Generating a triplicated bus from other signals
(e.g. clock gating for production testing)
- Generating identical slices of logic (timing critical logic)
(e.g. feedback divider for PLL)
- Specify majority voter and fanout cells on per module basis
(e.g. different voter for clock multiplexer)
- Integrated with SVN / CLIOSOFT sos version control systems

How to constrains the design ?

Constrains do not have to be places in the source code directly. **Constrains can be**

- **loaded from a configuration file**
  - The configuration file uses standard INI file format. It is a simple text file with a basic structure composed of sections, properties, and values. An example file may look like:

```
[modName]
  default : triplicate
  net : triplicate
  net : do_not_triplicate
  tmr_error : true
```

  - To **load a configuration file**, you have to specify its name as a command line argument:

```
$ tmrg -c config.cfg [other_options]
$ tmrg --config config.cfg [other_options]
```

- **provided as a command line arguments**. This approach is not very effective for constraining the whole project, but may be really handy in the initial phase. A possible constrains are shown bellow:

```
$ tmrg -d "default triplicate modName" [other_options]
$ tmrg -d "triplicate modName.net" [other_options]
$ tmrg -d "do_not_triplicate modName.net" [other_options]
$ tmrg -d "tmr_error true modName" [other_options]
```

Preserving TMR code during synthesis

TMRG

Most of the code generated by TMRG tool **is redundant**
→ **synthesizer will want to remove it (undesirable behavior!)**

The TMRG generates a set of constrains for you which will force Design Compiler not to discard the redundant logic:

```
$ tmrg --generate_sdc --sdc_headers comb06.v
```

As a result, a file `comb06TMR.sdc` will be generated. The file is a SDC file which can be loaded from the RC.

```
set sdc_version 1.3
set_dont_touch /designs/comb06TMR/nets/combLogicA
set_dont_touch /designs/comb06TMR/nets/combLogicB
set_dont_touch /designs/comb06TMR/nets/combLogicC
set_dont_touch /designs/comb06TMR/nets/combLogic
set_dont_touch /designs/comb06TMR/nets/inA
set_dont_touch /designs/comb06TMR/nets/inB
set_dont_touch /designs/comb06TMR/nets/inC
set_dont_touch /designs/comb06TMR/nets/in
```

Constrains may affect the logic optimization

Problem: how to make sure that the synthesis tool does not remove a specific cell?

Example Verilog code:

```
[..]
INVD1 instName(.I(myInput), .ZN(myOutput));
[..]
```

SDC constrain file:

```
set_dont_touch INVD1
```

- Any of INVD1 instances **will not be touched!**

RC/Genus script:

```
set_attribute preserve true /path/instName
```

- Specific instance **will be preserved!**

TMRG performance

TMRG

- **Small projects/files** - the TMRG tool should be usually very fast. As the TMRG does not invoke a complex runtime environment, the execution time should be **well below 1s**

- **Medium / large chip (in HEP community):**

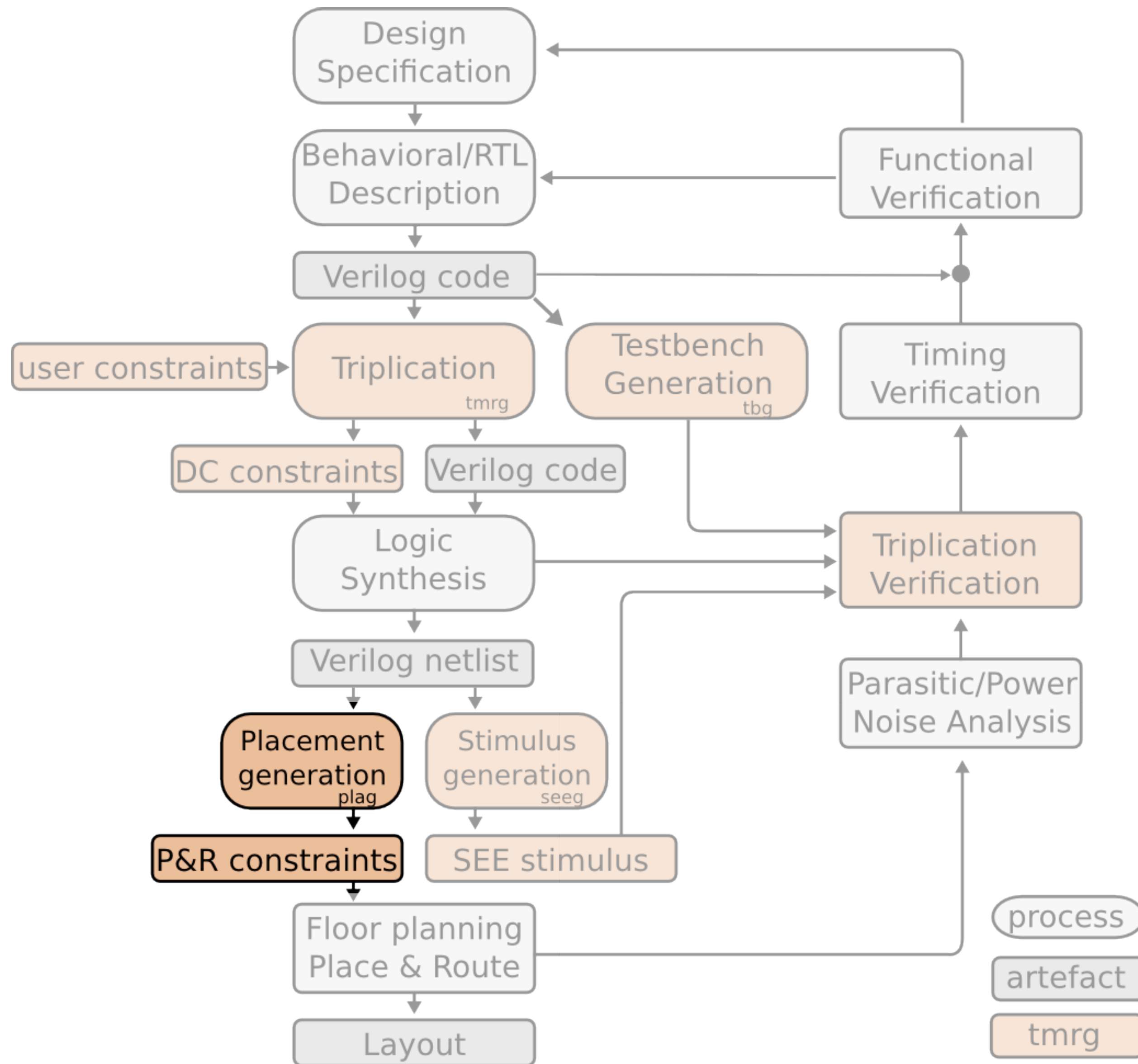
Chip	Die size [mm^2]	RTL Lines	TMR RTL Lines	TMRG run time [s]	Synthesis time [s]
lpGBT	4.5x4.5	36368	55776	173.3	T.B.D.
MPA	12x25	8937	21617	42.1	9000
SSA	11x4.5	3708	9081	20	6945
SALT	4x11	11612	20447	61.6	T.B.D.

*) The RTL lines count and die size are given only to indicate the chip size and complexity level.

- **The TMRG tool is suited for triplicating large chips in one go.**
- **The triplication time is almost negligible when compared to the synthesis time.**

PLAG

placement generator



Placement Generator

TMRG

- Majority voters before (or after) flip-flops cause the P&R tool to place instances of triplicated flip-flops close together (in order to keep the routing short).

- **Multiple bit upsets** can lead to malfunctioning of the triplicated design:
  → one has to ensure that the triplicated instances of the same element are **placed far from each each other.**

- The **PLAG** tool:
  - can assign registers (or other type of cells) to a specific Instances Group. A minimum distance is enforced, while leaving Encounter freedom to optimize the placement,
  - operates on a final netlist.

PLAG: Example

TMRG

```
# plag --lib libs/tcbn65lp.v -o tmrPlace.tcl r2g.v
```

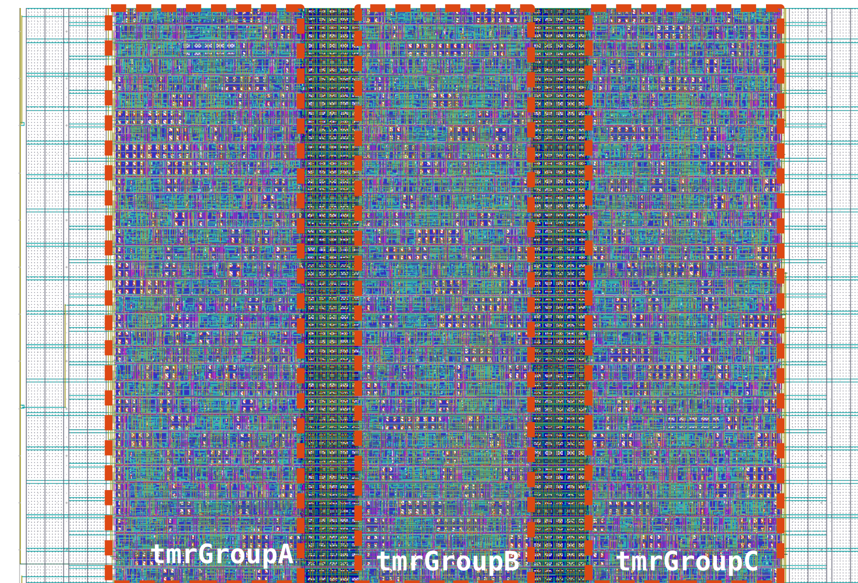
tmrPlace.tcl:

[..]

```
addInstToInstGroup tmrGroupA {fsm02TMR/stateA_reg}
addInstToInstGroup tmrGroupB {fsm02TMR/stateB_reg}
addInstToInstGroup tmrGroupC {fsm02TMR/stateC_reg}
```

[..]

[example layout]



Encounter flow:

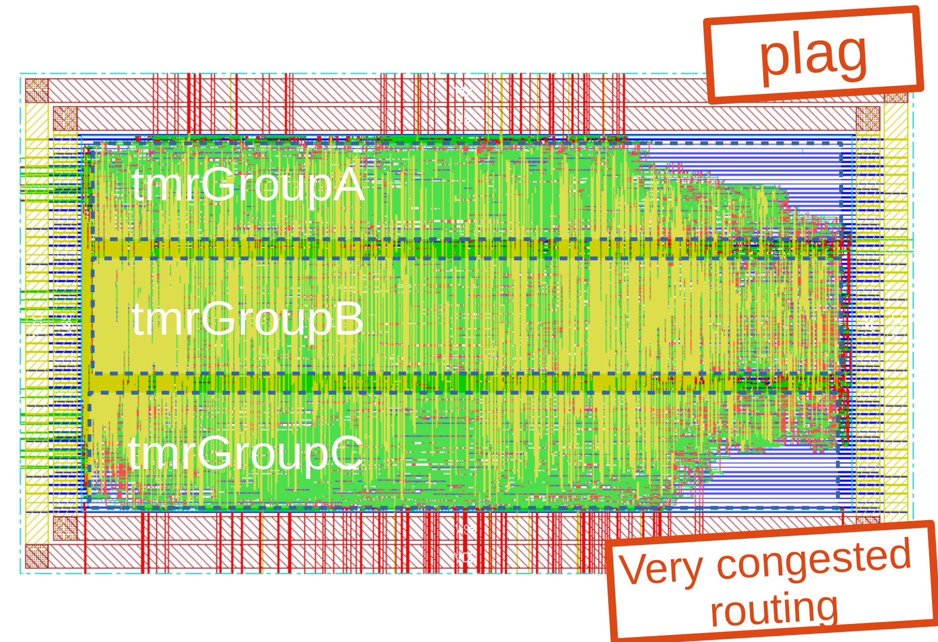
[..]

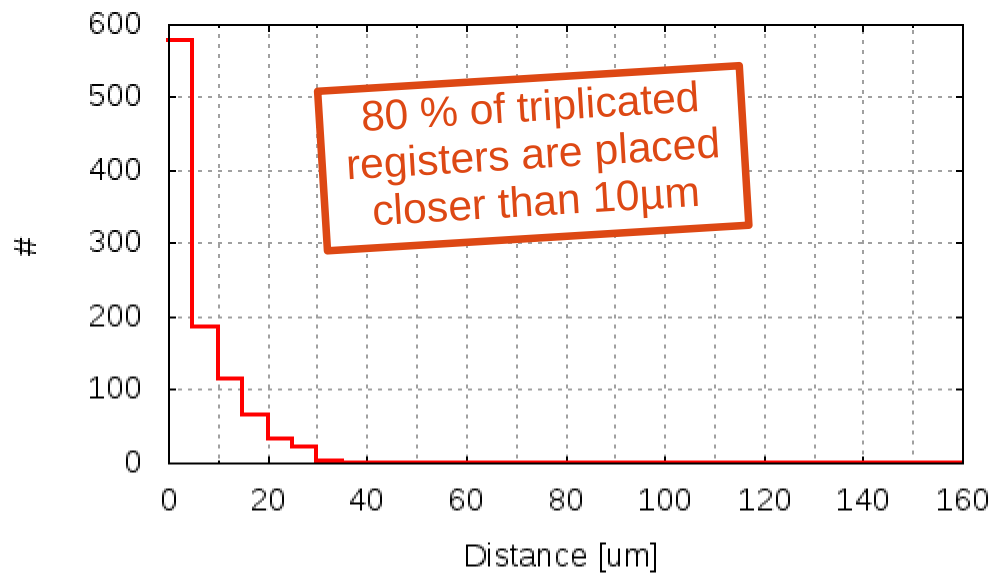
```
createInstGroup tmrGroupA -region 0 0 10 10
createInstGroup tmrGroupB -region 10 0 20 10
createInstGroup tmrGroupB -region 20 0 30 10
source tmrPlace.tcl
```

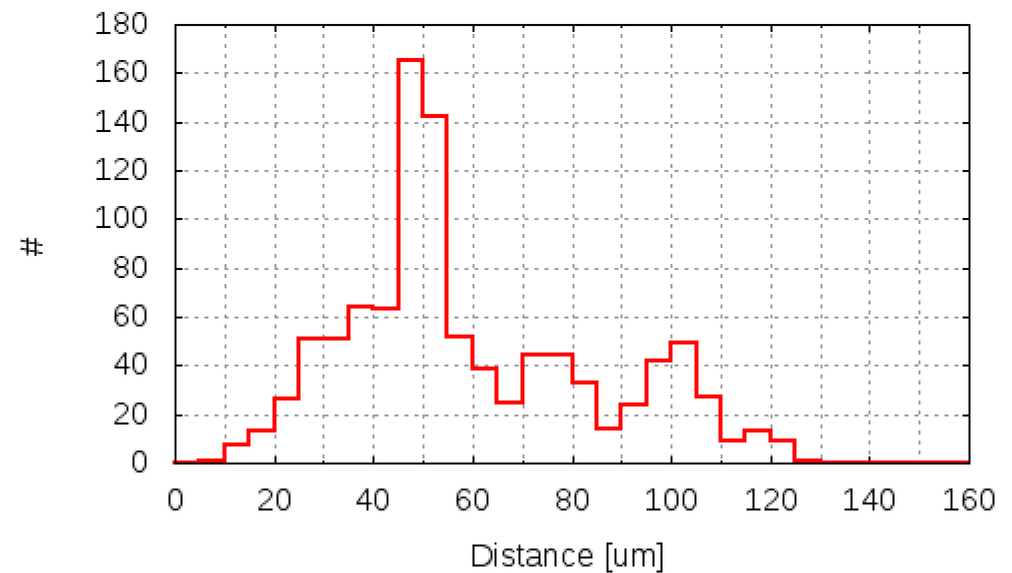
[..]

Room for improvement

PLAG: Example







Innovus tips & tricks

TMRG

Similar behaviour can be obtained with the newest version of Innovus P&N tool using command:

```
create_inst_space_group groupName
                        -inst listOfInstances
                        -spacing <value>
```

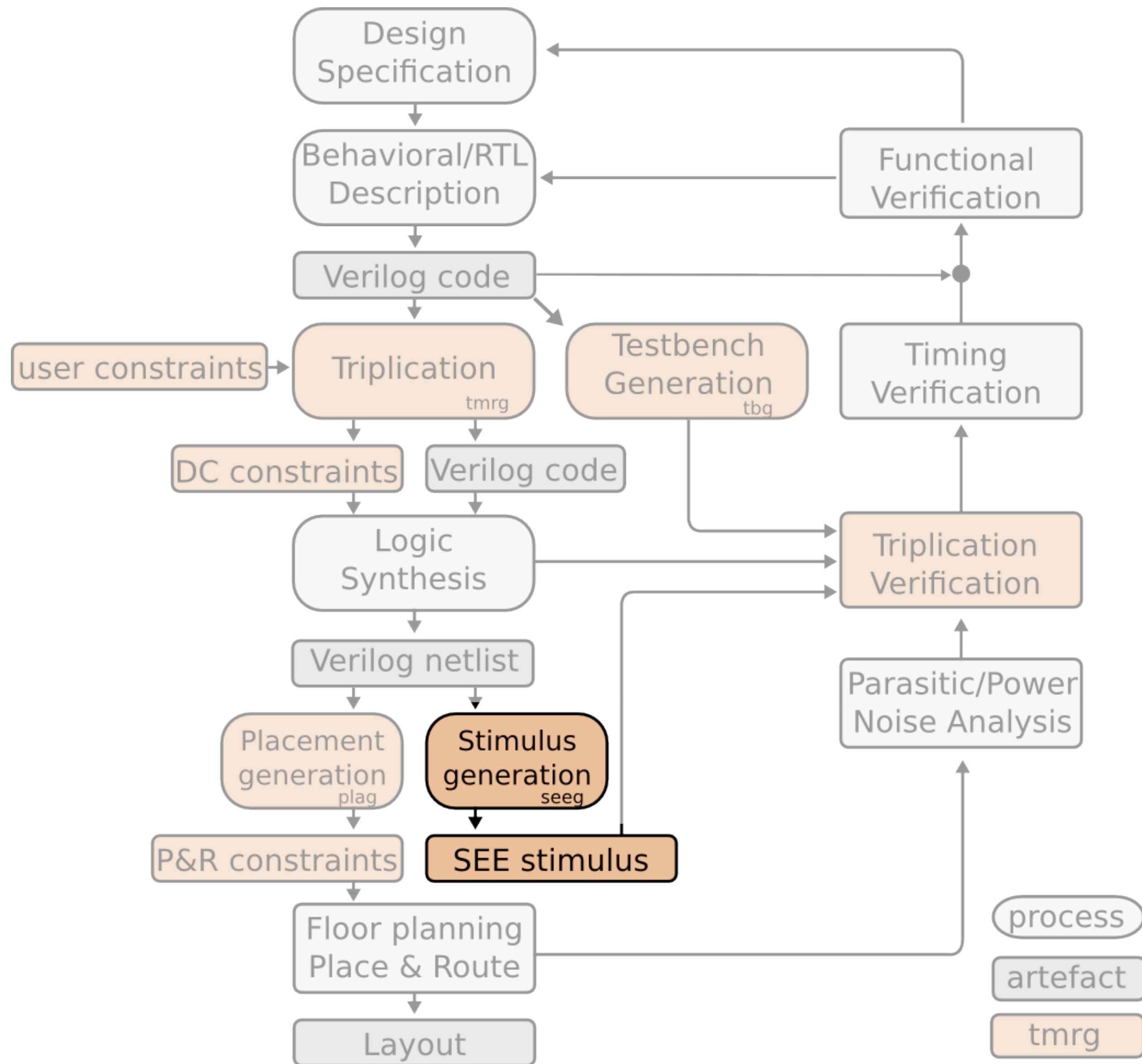
Creates space group for instances with a specific vertical-distance constraint.

- `-inst  listOfInstances` - specifies all instances that belong to the same space group by name.
- `-spacing verticalDistance` - specifies the vertical distance, in microns, between each specific instance.

SEEG

TMRG

single event effects generator



SEE simulation

TMRG

Once the design is implemented one should **verify** that the design still works as intended and that the design in **immune to SEE**.

How can we inject errors?

- Verilog:

```
force name=value;

release name;
```

- System Verilog:

```
$deposit(name, value);
```

Single Event Effects Generator

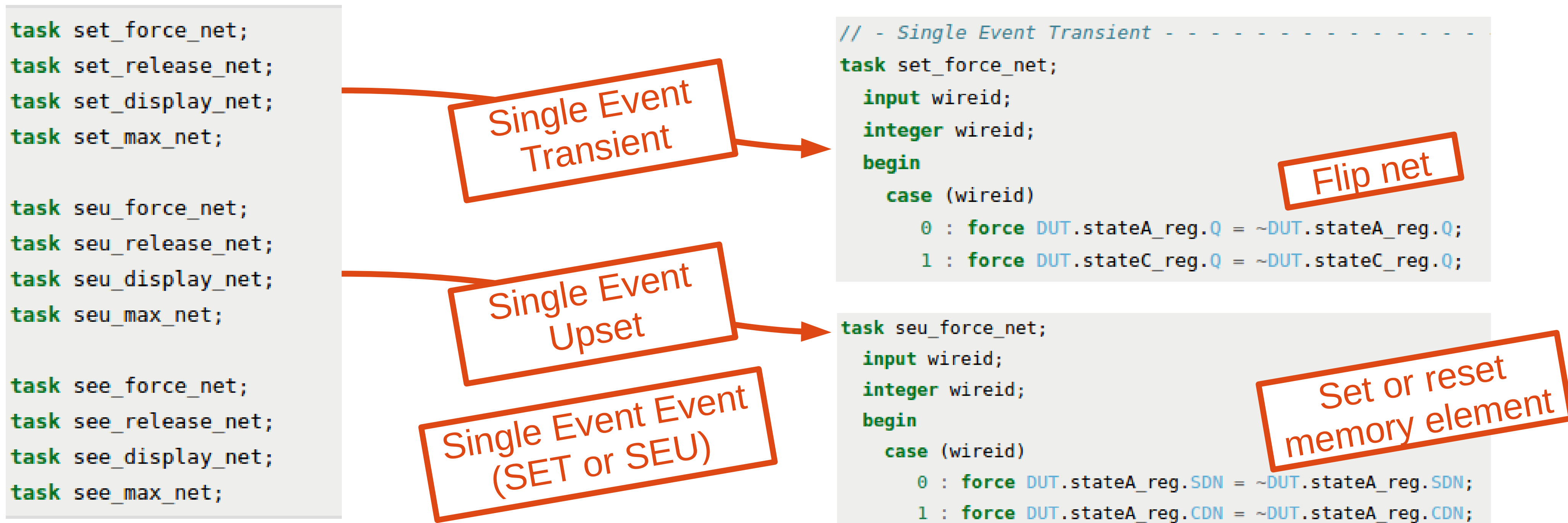
TMRG

SEEG tool streamlines the verification process.

An example usage of the tool for the netlist generated for an example fsm02 can look like:

```
# seeg --lib libs/tcbn65lp.v --output see.v r2g.v
```

The SEEG generates a file (see.v) which contains several verilog tasks, which can toggle nets (to simulate SET) or toggle flip-flops state (to simulate SEU) or both:



The approach has been verified with 65nm-HEP CMOS standard cell library and with custom standard cell library characterized with Liberate (lpGbtxHsLib).

Single Event Effects Generator

TMRG

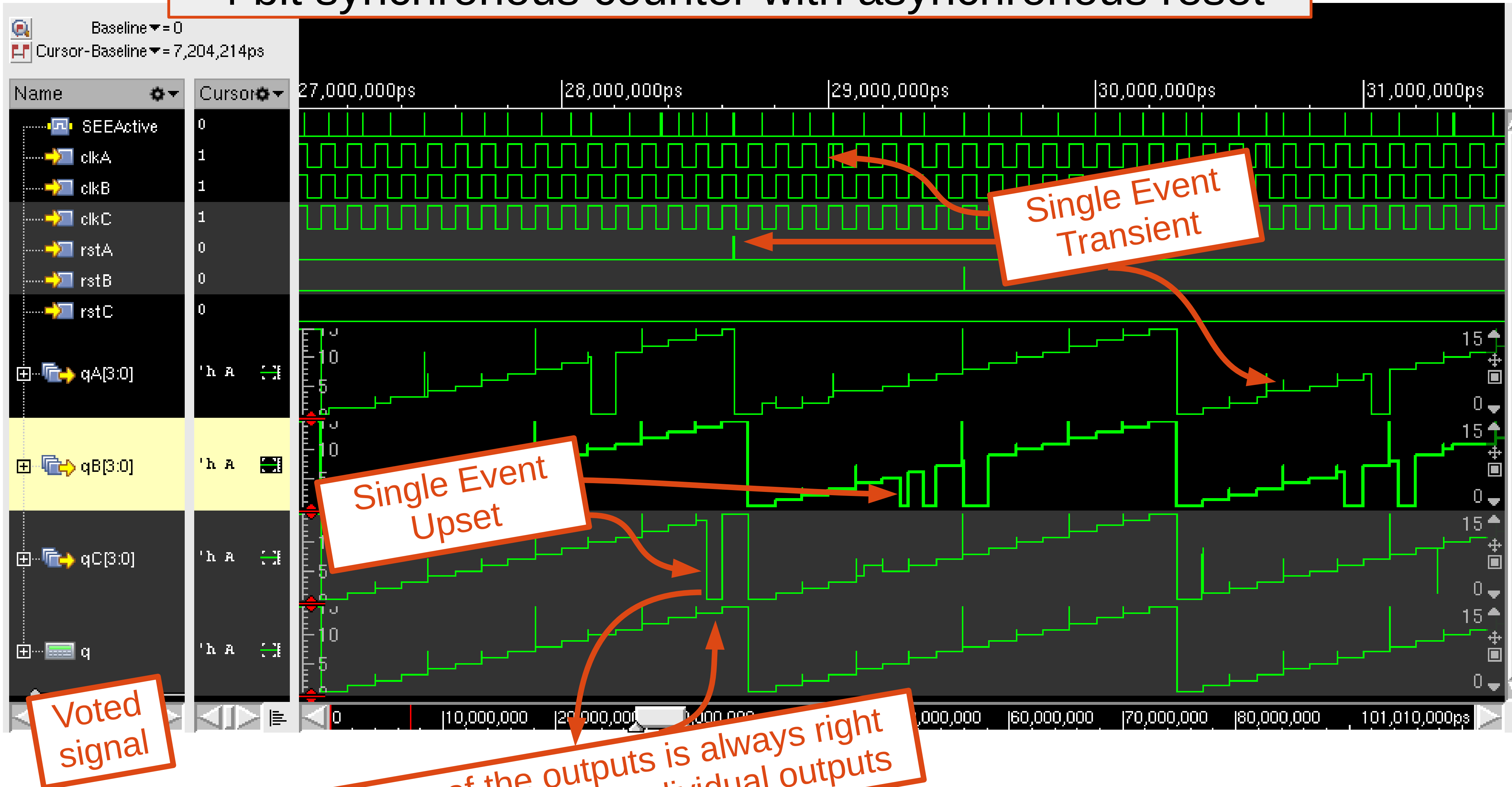
Example usage of SEEG generated tasks:

- Randomize :
  - the delay until next event
  - the length of the next event
  - Randomize the node (wire) to be affected (see / seu / set)
- Activate “upset” (force)
- Deactivate upset (release)

This is only a template which may be used as a starting point.

```
module stimulus;

 fsm02TMR DUT(...);

 [...]

 integer SEEEnable=1;      // enables SEE generator
 integer SEEnextTime;      // time until the next SEE event
 integer SEEduration;      // duration of the next SEE event
 integer SEEwireId;        // wire to be affected by the next SEE event
 integer SEEmaxWireId;     // number of wires in the design which can be affected by SEE event
 integer MAX_UPSET_TIME=10; // 10 ns  (change if you are using different timescale)
 integer SEEDel=100;       // 100 ns (change if you are using different timescale)
 integer SEECounter;       // number of simulated SEE events
 reg     SEEActive=0;      // high during any SEE event

 // get number of wires
 initial
   see_max_net (SEEmaxWireId);

 `include "fsm02TMR_see.v"

 always
   begin
     if (SEEEnable)
       begin
         // randomize time, duration, and wire of the next SEE
         SEEnextTime = SEEDel/2 {$random} % SEEDel;
         SEEduration = {$random} % (MAX_UPSET_TIME-1) + 1;  // SEE time is from 1 - MAX_UPSET_TIME
         SEEwireId   = {$random} % SEEmaxWireId;

         // wait for SEE
         #(SEEnextTime);

         // SEE happens here! Toggle the selected wire.
         SEECounter=SEECounter+1;
         SEEActive=1;
         see_force_net(SEEwireId);
         see_display_net(SEEwireId); // probably you want to comment this line ?
         #(SEEduration);
         see_release_net(SEEwireId);
         SEEActive=0;
       end
     else
       #10;
   end
endmodule
```

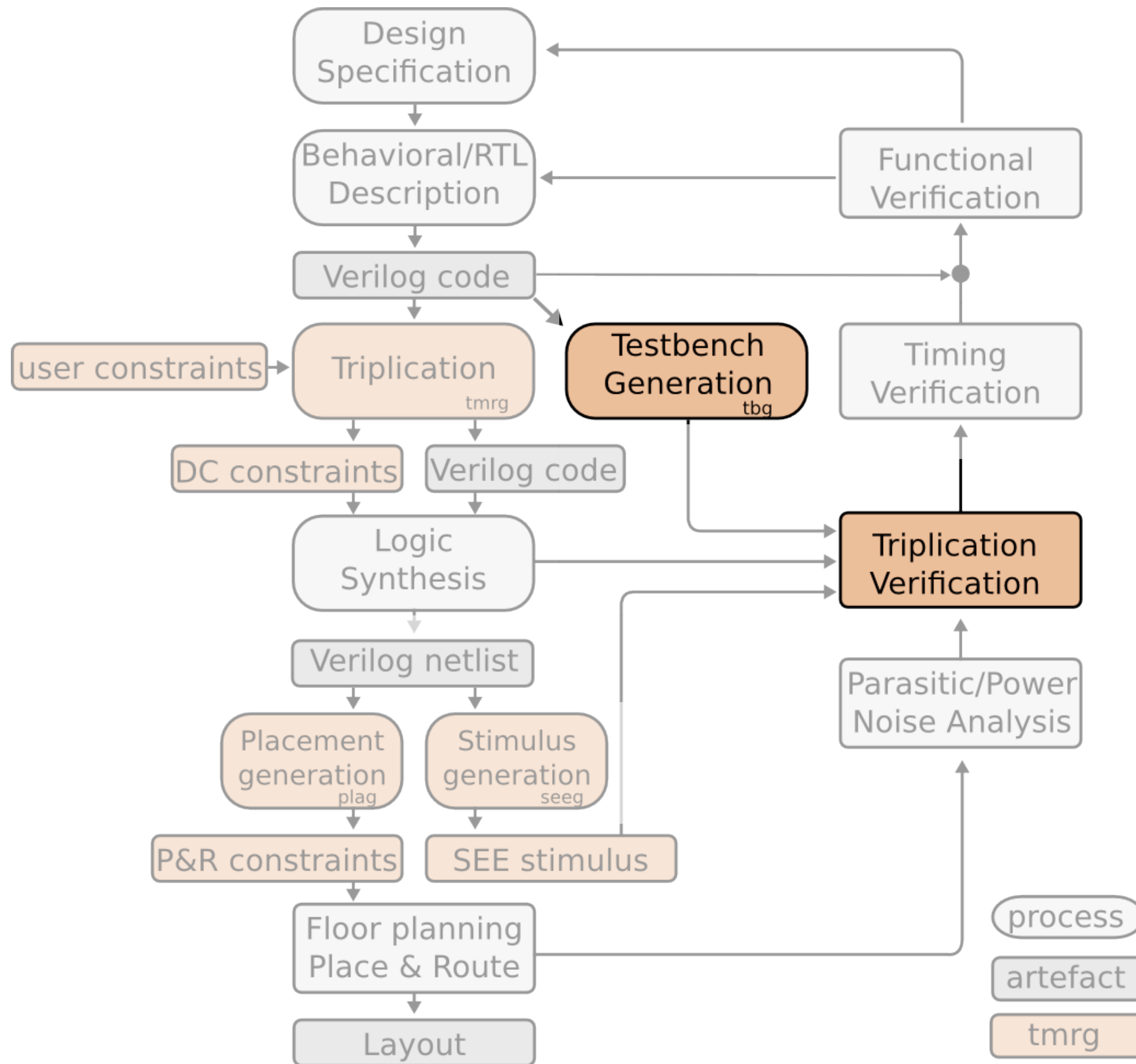
SEEG - example



TBG

TMRG

test bench generator



Test Bench Generator

TMRG

TBG generates a generic test bench

- RTL description / Netlist
- Non triplicated / triplicated version
  (automatic fanout/majority voter insertion for inputs/outputs)
- SDF timing annotation

```
`ifdef SDF
 initial
   $sdf_annotate("r2g.sdf", DUT, ,"sdf.log", "MAXIMUM");
`endif
```

- SET/SEU/SEE generation

```
`ifdef SEE
  reg     SEEEnable=0;           // enables SEE generator
  reg     SEEActive=0;           // high during any SEE event
  integer SEEnextTime=0;         // time until the next SEE event
  integer SEEduration=0;         // duration of the next SEE event
  integer SEEwireId=0;           // wire to be affected by the next SEE event
  integer SEEmaxWireId=0;        // number of wires in the design which can be af
  integer SEEmaxUpaseTime=1000;  // 1 ns  (change if you are using different time
  integer SEEDel=100_000;        // 100 ns (change if you are using different tim
  integer SEECounter=0;          // number of simulated SEE events

  `include "see.v"               // include tasks generated by seeg
  [..]
`endif
```

- simple clock/reset generators

 # tbg topModule.v -o topModule_test.v

```
`ifdef TMR
  // fanout for clk
  wire  clkA=clk;
  wire  clkB=clk;
  wire  clkC=clk;
  // fanout for in
  wire  inA=in;
  wire  inB=in;
  wire  inC=in;
  // voter for out
  wire  outA;
  wire  outB;
  wire  outC;
  wire outtmrErr;
  majorityVoterTB outVote
    .inA(outA),
    .inB(outB),
    .inC(outC),
    .out(out),
    .tmrErr(outtmrErr)
  );
  fsm02TMR DUT (
    .clkA(clkA),
    .clkB(clkB),
    .clkC(clkC),
    .inA(inA),
    .inB(inB),
    .inC(inC),
    .outA(outA),
    .outB(outB),
    .outC(outC)
  );
`else
  fsm02 DUT (
    .clk(clk),
    .in(in),
    .out(out)
  );
`endif
```

DEMO

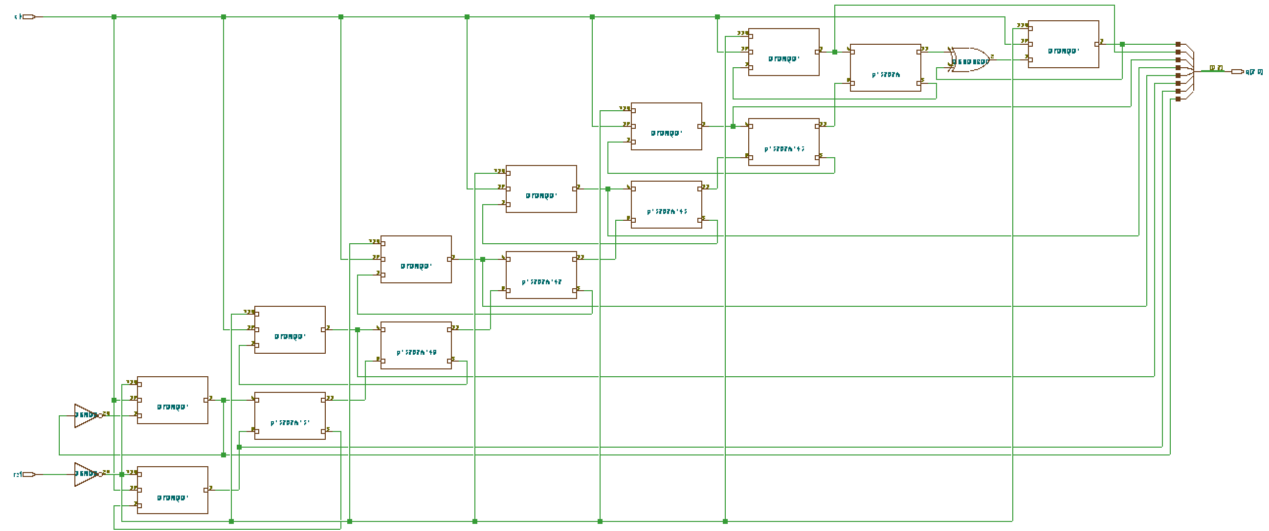
DEMO

TMRG

```
# source tmrg/etc/tmrg.sh

# tmrg --help
Usage: tmrg [options] fileName

Options:
  --version             show program's version number and exit
  -h, --help            show this help message and exit
  -v, --verbose         More verbose output (use: -v, -vv, -vvv..)
  --doc                 Open documentation in web browser

[…]

TMRG toolset:
  tmrg - Triple Modular Redundancy Generator
         (triplicates verilog netlist)
  seeg - Single Event Effects Generator
         (helps in the verification of triplicated netlist)
  plag - Placement Generator
         (helps with placement of triplicated circuit)
  tbg  - Testbench Generator
         (creates template for the testbench)
```

DFF example | RTL

TMRG



```
# cat counter.v

module counter(
  input d,
  input clk,
  input rst,
  output reg [7:0] q
);
  always @(posedge clk or posedge rst)
    if (rst)
      q<=0;
    else
      q<=q+1;
endmodule
```

no tmrg directives → default triplicate
Is it what we want ??

```
# tmrg counter.v
```

UNIX-like convention: no output → no errors!

```
# ls -l
counter.v counterTMR.v counterTMR.sdc
```

DFF example | Verbose

 # tmrg -v counter.v

```
[INFO    ] Loading file 'counter.v'
[INFO    ]
[INFO    ] Elaborating counter.v
[INFO    ] Module counter (counter.v)
[INFO    ] Port mode : ANSI
[INFO    ]
[INFO    ] Checking the design hierarchy
[INFO    ] [counter]
[INFO    ]
[INFO    ] Applying constrains
[INFO    ] Module counter
[INFO    ]  | tmrErrOut : False (configGlobal:False)
[INFO    ]  | net rst : True (configGlobalDefault:True)
[INFO    ]  | net q : True (configGlobalDefault:True)
[INFO    ]  | net clk40M : True (configGlobalDefault:True)
[INFO    ]  | net d : True (configGlobalDefault:True)
[INFO    ]
[INFO    ] Applying constrains by name
[INFO    ] Module counter
[INFO    ]
[INFO    ] Module:counter
[INFO    ] +#####################################################+######################+############+
[INFO    ] | Nets                                                |        range         |    tmr     |
[INFO    ] +#####################################################+######################+############+
[INFO    ] | rst                                                 |                      |    True    |
[INFO    ] | q                                                   |        [7:0]         |    True    |
[INFO    ] | clk40M                                              |                      |    True    |
[INFO    ] | d                                                   |                      |    True    |
[INFO    ] +-----------------------------------------------------+----------------------+------------+
[INFO    ] Triplciation starts here
[INFO    ]
[INFO    ] Triplicating file counter.v
[INFO    ] Generating SDC constraints file ./counterTMR.sdc
```

Going verbose “-v”

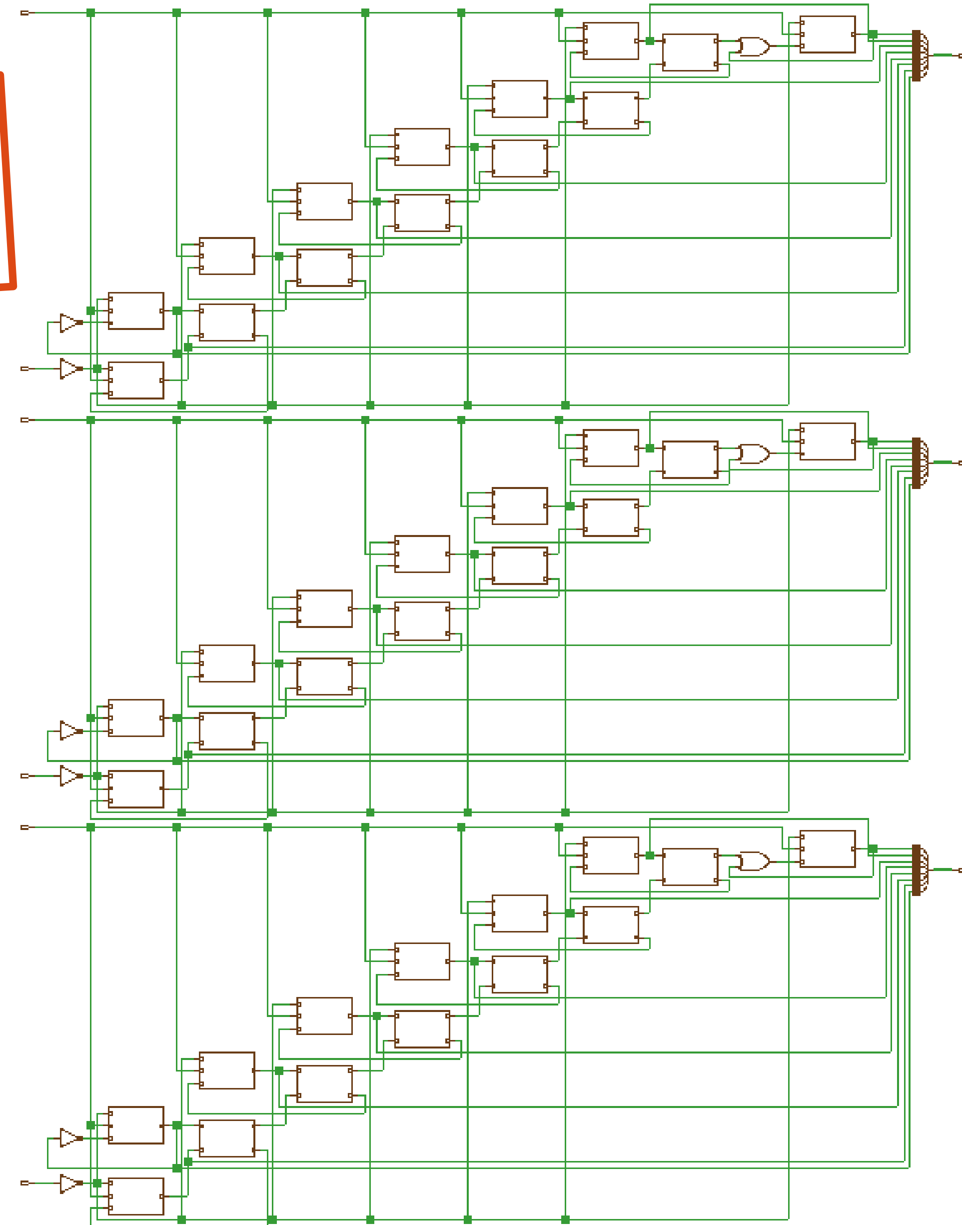
Going even more verbose “-vv”

DFF example | Triplication

TMRG

```
# cat counterTMR.v
module counterTMR(
  input  dA,
  input  dB,
  input  dC,
  input  clk40MA,
  input  clk40MB,
  input  clk40MC,
  input  rstA,
  input  rstB,
  input  rstC,
  output reg [7:0] qA,
  output reg [7:0] qB,
  output reg [7:0] qC
);

always @(posedge clk40MA or posedge rstA)
  if (rstA)
    qA <= 0;
  else
    qA <= qA+1;
always @(posedge clk40MB or posedge rstB)
  if (rstB)
    qB <= 0;
  else
    qB <= qB+1;
always @(posedge clk40MC or posedge rstC)
  if (rstC)
    qC <= 0;
  else
    qC <= qC+1;
endmodule
```

Is this type of triplication safe in this case?



DFF example | Place

```
# plag --lib /homedir/skulis/tmrg/trunk/libs/tcbn65lp.v  \
       dffTMR_rc/r2g.v
```

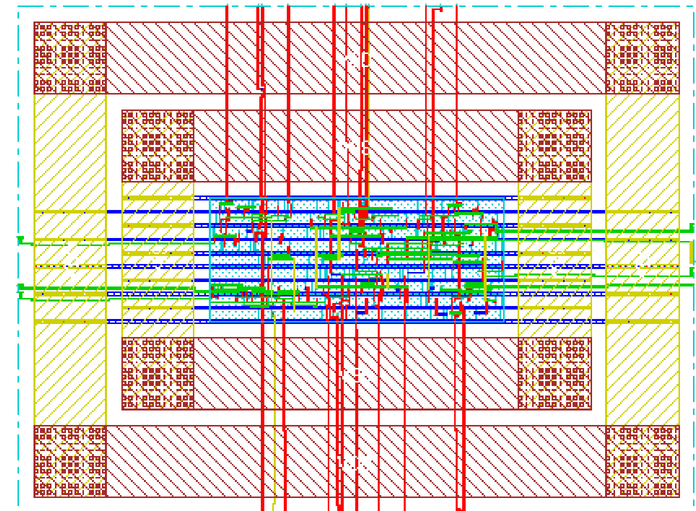
use -v for verbose output

```
# cat  tmrPlace.tcl
```

```
addInstToInstGroup tmrGroupA {dffTMR/qA_reg}
addInstToInstGroup tmrGroupB {dffTMR/qB_reg}
addInstToInstGroup tmrGroupC {dffTMR/qC_reg}
```

In the encounter flow:

```
[..]
createInstGroup tmrGroupA -region 0 0 10 10
createInstGroup tmrGroupB -region 10 0 20 10
createInstGroup tmrGroupB -region 20 0 30 10
source tmrPlace.tcl
[..]
```



DFF example | Verification

TMRG

```
# seeg --lib /homedir/skulis/tmrg/trunk/libs/tcbn65lp.v  \
       r2g.v
# cat  see.v
```

use -v for verbose output

```
task set_force_net;
  input wireid;
  integer wireid;
  begin
    case (wireid)
       0 : force DUT.qC_reg.Q = ~DUT.qC_reg.Q;
       1 : force DUT.dVoterA.Fp9999955A.ZN = ~DUT.dVoterA.Fp9999955A.ZN;
       2 : force DUT.dVoterA.p214748365A.ZN = ~DUT.dVoterA.p214748365A.ZN;
       3 : force DUT.dVoterC.Fp9999955A.ZN = ~DUT.dVoterC.Fp9999955A.ZN;
       4 : force DUT.dVoterC.p214748365A.ZN = ~DUT.dVoterC.p214748365A.ZN;
       5 : force DUT.dVoterB.Fp9999955A.ZN = ~DUT.dVoterB.Fp9999955A.ZN;
       6 : force DUT.dVoterB.p214748365A.ZN = ~DUT.dVoterB.p214748365A.ZN;
       7 : force DUT.qB_reg.Q = ~DUT.qB_reg.Q;
       8 : force DUT.qA_reg.Q = ~DUT.qA_reg.Q;
    endcase
  end
endtask

task set_release_net;
task set_display_net;
task set_max_net;

task seu_force_net;
task seu_release_net;
task seu_display_net;
task seu_max_net;

task see_force_net;
task see_release_net;
task see_display_net;
task see_max_net;
```

SET

SEU

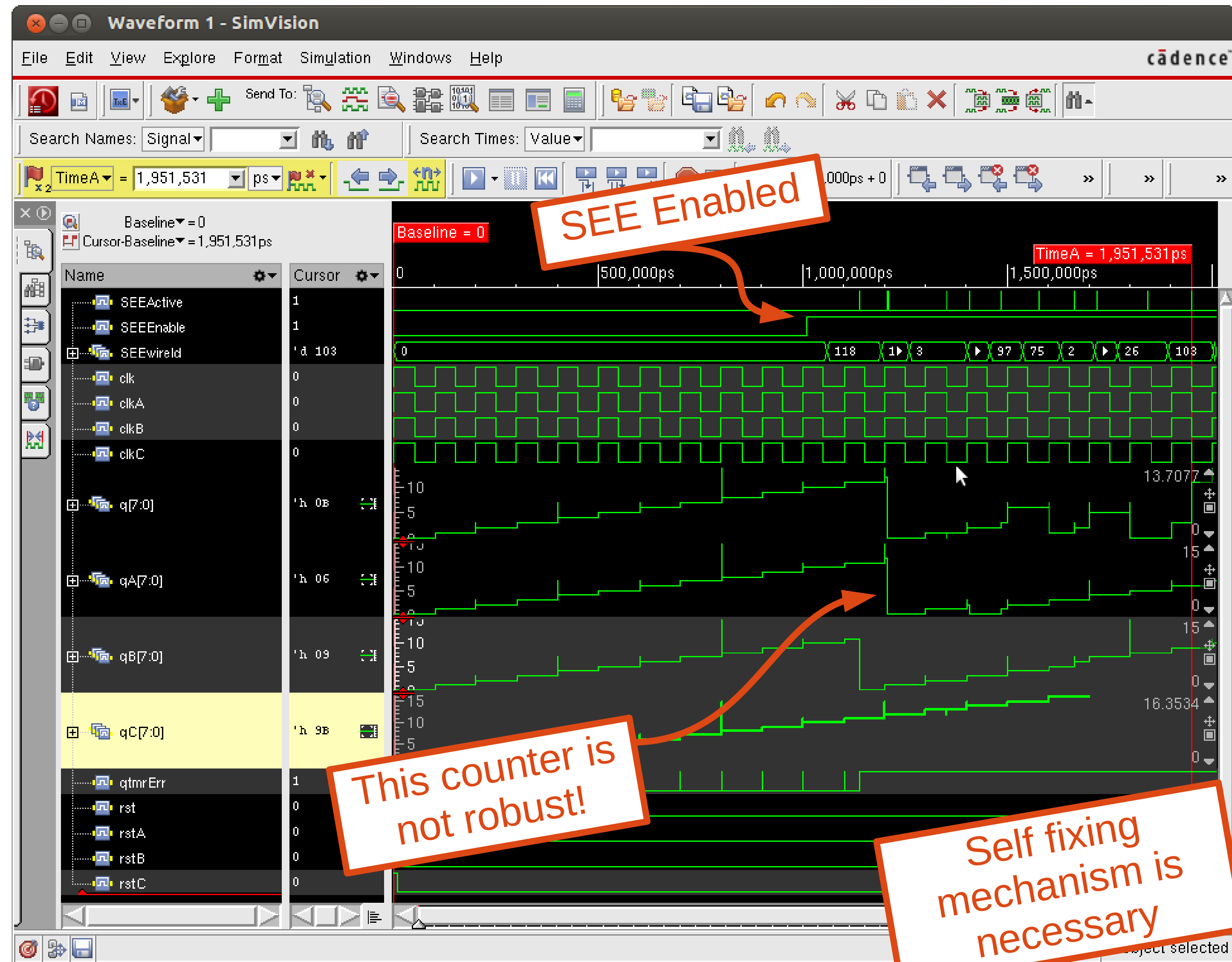
SEE

DFF example | Verification

TMRG

```
# tbg counter.v -o counter_test.v
# cat counter_test.v
```

```
`timescale 1 ps / 1 ps
[..]

module counter_test;

// Input/Output section
  reg  clk;
  wire [7:0] q;
  reg  rst;

// Device Under Test section

`ifdef TMR
  [...]
  counterTMR DUT (
    .clkA(clkA),
    .clkB(clkB),
    .clkC(clkC),
    .qA(qA),
    .qB(qB),
    .qC(qC),
    .rstA(rstA),
    .rstB(rstB),
    .rstC(rstC)
  );
`else
  counter DUT (
    .clk(clk),
    .q(q),
    .rst(rst)
  );
`endif
```



DEMO Summary

Conclusions:
- TMRG tool chain is **easy to use**
- TMRG flow can be **fully automatized**
- TMRG tool should be used with **caution**

Current Limitations

TMRG

- The TMRG supports only a subset of the Verilog language (sophisticated constructions may lead to incorrect results).
- The tool is not able to handle all possible coding styles:
  - Concatenation of triplicated and not triplicated variables on the left hand side of an assignment
  - unnamed connections for the module instantiation
- System Verilog syntax is not supported
- Parser error messages are not very verbose
- Some constants are hard-coded in the source code (like A,B,C postfixes for triplicated signals names)
- Higher order replication (5, 7, 9, ...) is not supported

Still maturing but a very useful tool!

TMRG

```
1 [...]
2 localparam k=8;
3 // tmrg do_not_triplicate dataMux
4 reg [2:0] sel;
5 reg [k-1:0] data [2*k-1:0];
6 wire [k-1:0] dataMux = data[sel];
7 [...]
```

TMRG

```
1 [...]
2 localparam  k=8;
3 wor dataTmrError;
4 wire [k-1:0] data [ 2*k-1 : 0 ] ;
5 wor selTmrError;
6 wire [2:0] sel;
7 reg  [2:0] selA;
8 reg  [2:0] selB;
9 reg  [2:0] selC;
10 reg  [k-1:0] dataA [ 2*k-1 : 0 ] ;
11 reg  [k-1:0] dataB [ 2*k-1 : 0 ] ;
12 reg  [k-1:0] dataC [ 2*k-1 : 0 ] ;
13 wire [k-1:0] dataMux =  data[sel] ;
14
15 majorityVoter #(.WIDTH(3)) selVoter (
16     .inA(selA),
17     .inB(selB),
18     .inC(selC),
19     .out(sel),
20     .tmrErr(selTmrError)
21     );
22
23 genvar gen_dataVoter;
24 generate
25   for(gen_dataVoter =  ((0>2*k-1) ? 2*k-1 : 0);gen_dataVoter<((0>2*k-1) ? 0 : 2*k-1);gen_dataVoter =  gen_dataVoter+1)
26     begin : gen_dataVoter_fanout
27
28       majorityVoter #(.WIDTH(((k-1)>(0)) ? ((k-1)-(0)+1) : ((0)-(k-1)+1))) dataVoter (
29           .inA(dataA[gen_dataVoter] ),
30           .inB(dataB[gen_dataVoter] ),
31           .inC(dataC[gen_dataVoter] ),
32           .out(data[gen_dataVoter] ),
33           .tmrErr(dataTmrError)
34           );
35     end
36 endgenerate
37 [...]
```

Summary

TMRG

Single Event Effects **mitigation techniques**:

- **Technology level** (minimizing sensitivity depth)
- **Cell level** (increasing critical charge, information stored on multiple nodes)
- **System level**:
  - **Encoding** (Hamming, Reed – Solomon, …)
  - **Triple Modular Redundancy**
    - Registers only
    - Registers and clocks
    - Full triplication

Summary

TMRG

- **TMRG tool chain** assists user along the design process of electronics resistant to Single Event Effects
  - It is compatible with the typical ASIC design flows used in the HEP community
  - It does not over constraint the user's coding style
  - It allows to obtain various flavors of TMR (registers only, full triplication, …)
  - it assists in the physical implementation stage (synthesis, P&R)
  - It assists in the verification process (generation of SEE)
  - It can be run in batch mode (fully automatic flow)
- To get started check the documentation at : **cern.ch/tmrg**
- **Development still continues, your feedback is essential!**

Thank you very much for your attention!

questions ?

suggestions ?

remarks ?

requests ?

Please visit : cern.ch/tmrg

Further reading

TMRG

- **Single event effects in static and dynamic registers in a 0.25 µm CMOS technology**
  F. Faccio; K. Kloukinas; A. Marchioro; T. Calin; J. Cosculluela; M. Nicolaidis; R. Velazco
  IEEE Transactions on Nuclear Science, Year: 1999, Volume: 46, Issue: 6

- **An SEU-Robust Configurable Logic Block for the Implementation of a Radiation-Tolerant FPGA**
  S. Bonacini; F. Faccio; K. Kloukinas; A. Marchioro
  IEEE Transactions on Nuclear Science, Year: 2006, Volume: 53, Issue: 6

- **Computational method to estimate Single Event Upset rates in an accelerator environment**
  M Huhtinen and F Faccio
  NIMA, Year: 2000, Volume:450, Issue:1

- **Characterization of a commercial 65 nm CMOS technology for SLHC applications**
  S Bonacini, P Valerio, R Avramidou, R Ballabriga, F Faccio, K Kloukinas and A Marchioro
  Journal of Instrumentation, Volume 7, January 2012

- ***Single-event upset sensitivity of latches in a 90nm dual and triple well CMOS technology***
  L Pierobon, S Bonacini, F Faccio and A Marchioro
  Journal of Instrumentation, Volume 6, December 2011

- **Design and characterization of an SEU-robust register in 130nm CMOS for application in HEP ASICs**
  S Bonaci
  Journal of Instrumentation, Volume 5, November 2010

Backup slides

Single Event Effects

Single-event effect (SEE) is a phenomena triggered by a charged particle passing through an electronic device. Traversing particle ionizes the matter producing electron-holes pairs. The amount of charge being generated depends on particle type, particle energy, incident angle, material. The charge can be then collected by a drain/source diffusion and can modify its voltage, changing its logical value (from zero to one or vice versa).

Traditionally we distinguish two types of upsets:

- **Single-event transient (SET)** is a phenomena in which an error happens in a combinatorial logic. It appears as a short glitch on a net. The proper value is restored within short time (~ns). Importance of SET increases with increasing clock frequency when the duration of SET becomes comparable with a clock period.
- **Single-event upsets (SEU)** are errors induced in memory cells (like flip-flop). In contradiction to SET, the value of the memory cell does not recover after SEU.

Tripple Module Redundancy

TMRG

There have been several techniques proposed in order to protect the circuit against events caused by the ionizing particles. Virtually all techniques relay on a **data redundancy**. It is assumed, that if the bit of information is stored in several places (nodes) the information can be properly reconstructed even if some of these places (nodes) get disturbed. There are some circuit techniques, based on **hardening standard cells** while the other techniques address the problem at the **system level**, by utilizing error-correcting coding (ECC), temporal redundancy, or Tripple Module Redundancy.

Tripple Module Redundancy (TMR) concept was originally developed by Von Neumann, with the main purpose of enhancing reliability of electronic circuits. This concept was later applied in microelectronics for protection against ionizing particles.

The purpose of TMRG tool is to automatize process of triplicating digital circuits.

Accessing individual signals from a triplicated bus

TMRG

In some very spetial cases you may want to access signals after the triplication individually. Imagine that you are designing a reset circuit. You want to have a

- Power-on reset (POR) block
- and an external reset signal.

As you do not want that SET in POR block resets your chip, you may decide to triplicate the block. For practical reasons, you still want to keep only one external reset pin.

```
 1 module powerOnReset(z);
 2   // tmrg do_not_touch
 3   output z;
 4 endmodule
 5
 6 module resetBlock01 (rstn,rst);
 7   // tmrg default triplicate
 8   // tmrg do_not_triplicate rstn
 9   input rstn;
10   output rst;
11   wire porRst;
12   assign rst=!rstn | porRst;
13   powerOnReset por(.z(porRst));
14 endmodule
```

```
 1 module powerOnReset(
 2   z
 3 );
 4 output z;
 5 endmodule
 6 module resetBlock01TMR(
 7   rstn,
 8   rstA,
 9   rstB,
10   rstC
11 );
12 wire rstnC;
13 wire rstnB;
14 wire rstnA;
15 input rstn;
16 output rstA;
17 output rstB;
18 output rstC;
19 wire porRstA;
20 wire porRstB;
21 wire porRstC;
22 assign rstA =  !rstnA|porRstA;
23 assign rstB =  !rstnB|porRstB;
24 assign rstC =  !rstnC|porRstC;
25
26 powerOnReset porA (
27   .z(porRstA)
28 );
29
30 powerOnReset porB (
31   .z(porRstB)
32 );
33
34 powerOnReset porC (
35   .z(porRstC)
36 );
37
38 fanout rstnFanout (
39   .in(rstn),
40   .outA(rstnA),
41   .outB(rstnB),
42   .outC(rstnC)
43 );
44 endmodule
```

There is not magic so far.

Accessing individual signals from a triplicated bus

If you decided that you would like to able to check during normal operation what is the status of the POR output, the straight forward way of doing that would be:

```
 1 module powerOnReset(z);
 2   // tmrg do_not_touch
 3   output z;
 4 endmodule
 5
 6 module resetBlock (rstn,rst,porStatus);
 7   // tmrg do_not_triplicate rstn
 8   input rstn;
 9   output rst;
10   output porStatus;
11   wire porRst;
12   assign porStatus=porRst;
13   assign rst=!rstn | porRst;
14   powerOnReset por(.z(porRst));
15 endmodule
```

```
 1 module powerOnReset(
 2   z
 3 );
 4 output z;
 5 endmodule
 6 module resetBlockTMR(
 7   rstn,
 8   rstA,
 9   rstB,
10   rstC,
11   porStatusA,
12   porStatusB,
13   porStatusC
14 );
15 wire rstnC;
16 wire rstnB;
17 wire rstnA;
18 input rstn;
19 output rstA;
20 output rstB;
21 output rstC;
22 output porStatusA;
23 output porStatusB;
24 output porStatusC;
25 wire porRstA;
26 wire porRstB;
27 wire porRstC;
28 assign porStatusA =  porRstA;
29 assign porStatusB =  porRstB;
30 assign porStatusC =  porRstC;
31 assign rstA =  !rstnA|porRstA;
32 assign rstB =  !rstnB|porRstB;
33 assign rstC =  !rstnC|porRstC;
34
35 powerOnReset porA (
36   .z(porRstA)
37 );
38
39 powerOnReset porB (
40   .z(porRstB)
41 );
42
43 powerOnReset porC (
44   .z(porRstC)
45 );
46
47 fanout rstnFanout (
48   .in(rstn),
49   .outA(rstnA),
50   .outB(rstnB),
51   .outC(rstnC)
52 );
53 endmodule
```

You may see that 'porStatus'' signal got triplicated which is of course what we want. Lets think if this is what you really want. If you connect it to some kind of digital bus, most likely you will have some voting on the way, so you will not have an information about individual signals.

How to constrain the design ?

TMRG

A brief summary of all constrains, ways of specifying it, and priorities is shown in Table below.

directive in code (lowest priority)	configuration file (medium priority)	command line argument (highest priority)
`module modName(..);`	`[modName]`	-
`// tmrg default triplicate`	`default : triplicate`	`-d "default triplicate modName"`
`// tmrg triplicate net`	`net : triplicate`	`-d "triplicate modName.net"`
`// tmrg do_not_triplicate net`	`net : do_not_triplicate`	`-d "do_not_triplicate modName.net"`
`// tmrg tmr_error true`	`tmr_error : true`	`-d "tmr_error true modName"`
`// tmrg tmr_error_exclude net`	`tmr_error_exclude : net`	`-d "tmr_error_exclude modName.net"`
`// tmrg slicing`	`slicing : net`	`-d "slicing modName"`
`// tmrg majority_voter_cell name`	`majority_voter_cell : name`	`-d "majority_voter_cell name modName"`
`// tmrg fanout_cell name`	`fanout_cell : name`	`-d "fanout_cell name modName"`

How to constrains the design (debugging) ?

TMRG

As there are several ways of specifying constrains and one constrain can be overwritten by another, there is mechanism which can ensure the designer that all his intentions are interpreted properly.

Lets consider the comb06 module from the above example. Lets write a configuration files comb06.cnf:

```
[comb06]
default : do_not_triplicate
in : triplicate
out : do_not_triplicate
combLogic : triplicate
tmr_error : true
```

When you run TMRG with additional options and constrains as shown below:

```
$ tmrg -vv \
       -w "default triplicate comb06" \
       -w "tmr_error false comb06" \
       -w "triplicate comb06.combLogic" \
       -c comb06.cnf \
       comb06.v
```

How to constrains the design (debugging) ?

TMRG

The **detailed log** of what is being done can be generated (-v option)

The table at the end of the listing summarizes all discovered signals and applied constrains.

Step by step process of applying constrains can be used to understand at which point something went wrong:

- Used configuration files
- Command line constrains
- Constrains evaluation for given nets

```
[..]
[DEBUG   ] Loading master config file from /home/skulis/work/tmrg/trunk/bin/../etc/tmrg.cfg
[DEBUG   ] Loading user config file from /home/skulis/.tmrg.cfg
[DEBUG   ] Loading command line specified config file from comb06.cnf
[..]
[INFO    ] Command line constrain 'directive_default' for module 'comb06' (value:True)
[INFO    ] Command line constrain 'directive_tmr_error' for module 'comb06' (value:False)
[INFO    ] Command line constrain 'directive_triplicate' for net 'combLogic' in module 'comb06'
[..]
[INFO    ] Applying constrains
[INFO    ] Module comb06
[INFO    ]  | tmrErrOut : False (configGlobal:False
                          -> configModule:True
                          -> cmdModule:False)
[INFO    ]  | net combLogic : True (configGlobalDefault:True
                          -> srcModuleDefault:True
                          -> configModuleDefault:False
                          -> cmdModuleDefault:True
                          -> src:False
                          -> config:True
                          -> cmd:True)
[INFO    ]  | net in : True (configGlobalDefault:True
                          -> srcModuleDefault:True
                          -> configModuleDefault:False
                          -> cmdModuleDefault:True
                          -> config:True)
[INFO    ]  | net out : False (configGlobalDefault:True
                          -> srcModuleDefault:True
                          -> configModuleDefault:False
                          -> cmdModuleDefault:True
                          -> config:False)
[..]
[INFO    ] Module:comb06
[INFO    ] +####################################################+##################+############+
[INFO    ] | Nets                                                |       range      |    tmr     |
[INFO    ] +####################################################+##################+############+
[INFO    ] | combLogic                                           |                  |    True    |
[INFO    ] | in                                                  |                  |    True    |
[INFO    ] | out                                                 |                  |   False    |
[INFO    ] +----------------------------------------------------+------------------+------------+
[..]
```

Summary table

Accessing individual signals from a triplicated bus

TMRG

In order to solve the problem, you have to “code” some triplication manually. If you declare a wire with a special name and with a special assignment (like bellow) you gain access to the signal after triplication:

```
1 wire myWire;
2 wire myWireA=myWire;
3 wire myWireB=myWire;
4 wire myWireC=myWire;
```

This convention ensures that you can still simulate and synthesize you original design. TMRG will convert this declarations during elaboration process to the desired ones. Lets see how we can use this in our resetBlock example.

```
1  module powerOnReset(z);
2    // tmrg do_not_touch
3    output z;
4  endmodule
5
6  module resetBlock (rstn,rst,porStatus);
7    // tmrg do_not_triplicate rstn
8    input rstn;
9    output rst;
10   output [2:0] porStatus;
11   wire porRst;
12   wire porRstA=porRst;
13   wire porRstB=porRst;
14   wire porRstC=porRst;
15   assign porStatus={porRstC,porRstB,porRstA};
16   assign rst=!rstn | porRst;
17   powerOnReset por(.z(porRst));
18 endmodule
```

vector!

```
1  module powerOnReset(
2    z
3  );
4  output z;
5  endmodule
6  module resetBlockTMR(
7    rstn,
8    rstA,
9    rstB,
10   rstC,
11   porStatusA,
12   porStatusB,
13   porStatusC
14 );
15 wire rstnC;
16 wire rstnB;
17 wire rstnA;
18 input rstn;
19 output rstA;
20 output rstB;
21 output rstC;
22 output [2:0] porStatusA;
23 output [2:0] porStatusB;
24 output [2:0] porStatusC;
25 wire porRstA;
26 wire porRstB;
27 wire porRstC;
28 assign porStatusA =  {porRstC,porRstB,porRstA};
29 assign porStatusB =  {porRstC,porRstB,porRstA};
30 assign porStatusC =  {porRstC,porRstB,porRstA};
31 assign rstA =  !rstnA|porRstA;
32 assign rstB =  !rstnB|porRstB;
33 assign rstC =  !rstnC|porRstC;
34
35 powerOnReset porA (
36   .z(porRstA)
37 );
38
39 powerOnReset porB (
40   .z(porRstB)
41 );
42
43 powerOnReset porC (
44   .z(porRstC)
45 );
46
47 fanout rstnFanout (
48   .in(rstn),
49   .outA(rstnA),
50   .outB(rstnB),
51   .outC(rstnC)
52 );
53 endmodule
```

As you can see, we ended up with triplicated bus (9 signals!).

Accessing individual signals from a triplicated bus

In the previous example it was shown how to fanout a signal in order to access sub-signals in a triplicated signal. Now let us consider opposite situation, how to generate triplicated signal from arbitrary combination of other signals.

To make example easier to understand, lets take real-life problem: we want to make a clock gating circuit. A simple implementation with only one gating signal may look like:

```
1 module clockGating (clkIn,clkOut,clkGate);
2   // tmrg default triplicate
3   input clkIn;
4   output clkOut;
5   input clkGate;
6   assign clkOut=clkIn&clkGate;
7 endmodule
```

```
1 module clockGatingTMR(
2   clkInA,
3   clkInB,
4   clkInC,
5   clkOutA,
6   clkOutB,
7   clkOutC,
8   clkGateA,
9   clkGateB,
10  clkGateC
11 );
12 input clkInA;
13 input clkInB;
14 input clkInC;
15 output clkOutA;
16 output clkOutB;
17 output clkOutC;
18 input clkGateA;
19 input clkGateB;
20 input clkGateC;
21 assign clkOutA =  clkInA&clkGateA;
22 assign clkOutB =  clkInB&clkGateB;
23 assign clkOutC =  clkInC&clkGateC;
24 endmodule
```

No magic so far.

Accessing individual signals from a triplicated bus

TMRG

If we want to be able to gate individual sub-signals in a triplicate clock, we have to use similar trick as in the resetBlock.

```
 1 module clockGating (clkIn,clkOut,clkGate);
 2   // tmrg default triplicate
 3   input clkIn;
 4   output clkOut;
 5   input [2:0] clkGate;
 6   wire gateA=clkGate[0];
 7   wire gateB=clkGate[1];
 8   wire gateC=clkGate[2];
 9   wire gate=gateA;
10   assign clkOut=clkIn&gate;
11 endmodule
```

vector!

TMRG

```
 1 module clockGatingTMR(
 2   clkInA,
 3   clkInB,
 4   clkInC,
 5   clkOutA,
 6   clkOutB,
 7   clkOutC,
 8   clkGateA,
 9   clkGateB,
10   clkGateC
11 );
12 wire [2:0] clkGate;
13 input clkInA;
14 input clkInB;
15 input clkInC;
16 output clkOutA;
17 output clkOutB;
18 output clkOutC;
19 input [2:0] clkGateA;
20 input [2:0] clkGateB;
21 input [2:0] clkGateC;
22 wire gateA =  clkGate[0];
23 wire gateB =  clkGate[1];
24 wire gateC =  clkGate[2];
25 assign clkOutA =  clkInA&gateA;
26 assign clkOutB =  clkInB&gateB;
27 assign clkOutC =  clkInC&gateC;
28
29 majorityVoter #(.WIDTH(3)) clkGateVoter (
30   .inA(clkGateA),
31   .inB(clkGateB),
32   .inC(clkGateC),
33   .out(clkGate)
34 );
35 endmodule
```