LOW-POWER VERIFICATION AUTOMATION

A PRACTICAL APPROACH

Shaji K. Kunjumohamed, Broadcom Corporation
Hendy Kosasih, Cadence Design Systems

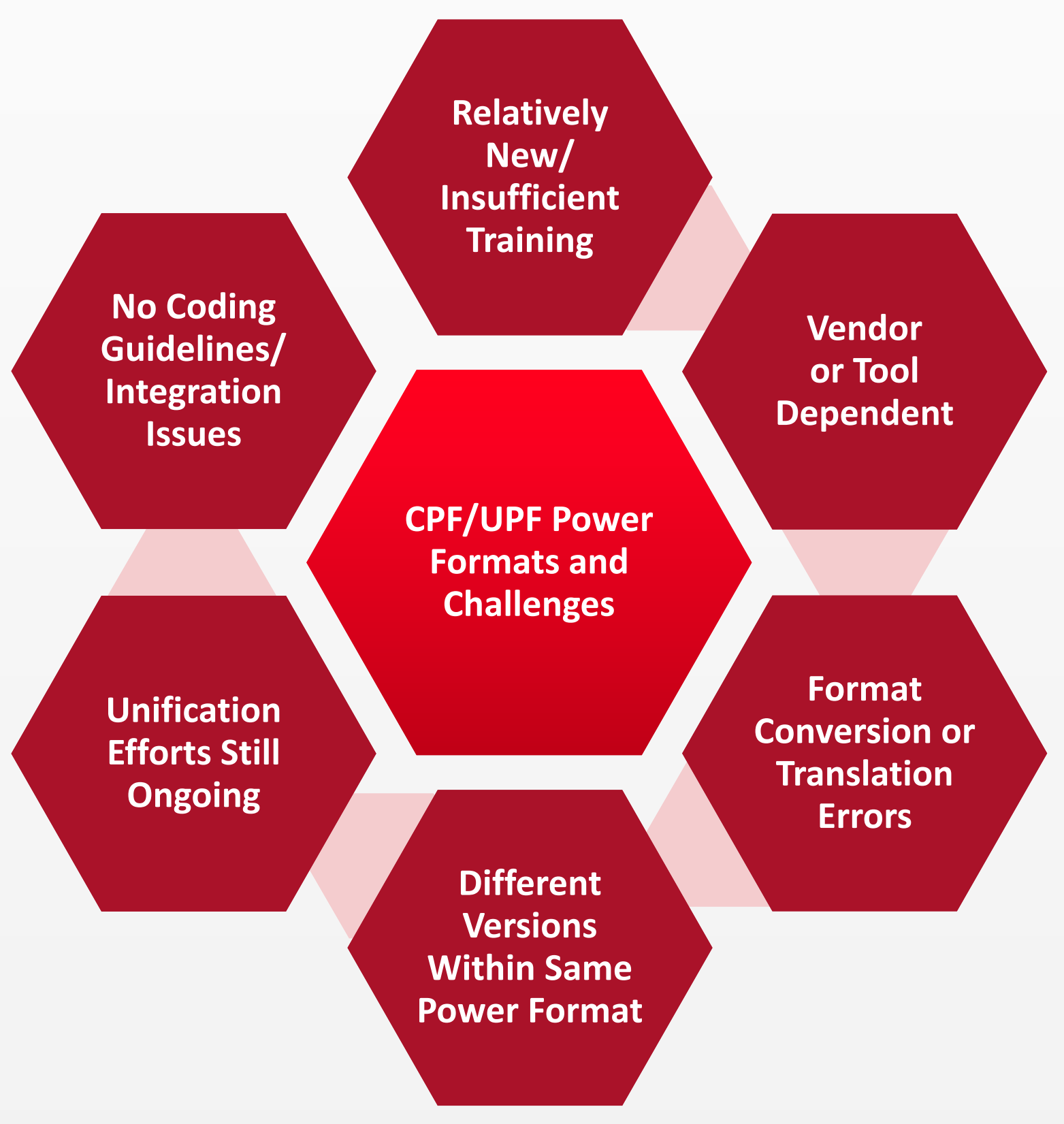
Abstract

Today's SoCs use a variety of hardware and software power schemes to minimize power consumption. This poster attempts to explore some common low-power design and verification issues, and will focus on some methodologies that can address such problems.

Introduction

- Designers use static and dynamic verification techniques and various tools to implement low power design.
- Power formats like CPF and UPF with RTL and Gate-level netlist transfers low power intent of a design from one hardware design stage to another.
- No reliable method exist that would transfer the low power information from hardware to software.

Challenges with Power Format Files


Relatively New/ Insufficient Training
Vendor or Tool Dependent
Format Conversion or Translation Errors
Different Versions Within Same Power Format
Unification Efforts Still Ongoing
No Coding Guidelines/ Integration Issues
CPF/UPF Power Formats and Challenges

Power Format – Reuse Guidelines

- **Reuse** rule constructs, that designers have developed and tested at block level, during top level integration. Constructs can be reused by wrapping inside TCL procedures and calling it from chip level.
- Hierarchically **organize** power format files, the same way of how major blocks in a chip is organized.
- **Adopt** a uniform method of coding for power format files – in terms of naming conventions for variables, reusable constructs, methods and libraries.
- **Automate** the integration process.
- **Share** same power format file between different tools, instead of having multiple copies.
- **Integrate** the low power environment into existing flows for invoking different tool chains. The infrastructure for this need to be developed.

Auto Generation of CPF/UPF

- Many companies have flows which allow users to simulate designs without worrying about various underlying simulators, OS and other licenses. Low power automation can also use similar approach.
- Users can specify power intent of a design at a higher level of abstraction than the corresponding power format file of the design. From this initial specification, either CPF or UPF file can be auto generated.
- The power format file can be auto-generated in such a way that specific tool dependent pragmas and constructs can be activated for a specific tool run. The same file can be shared between multiple tools.

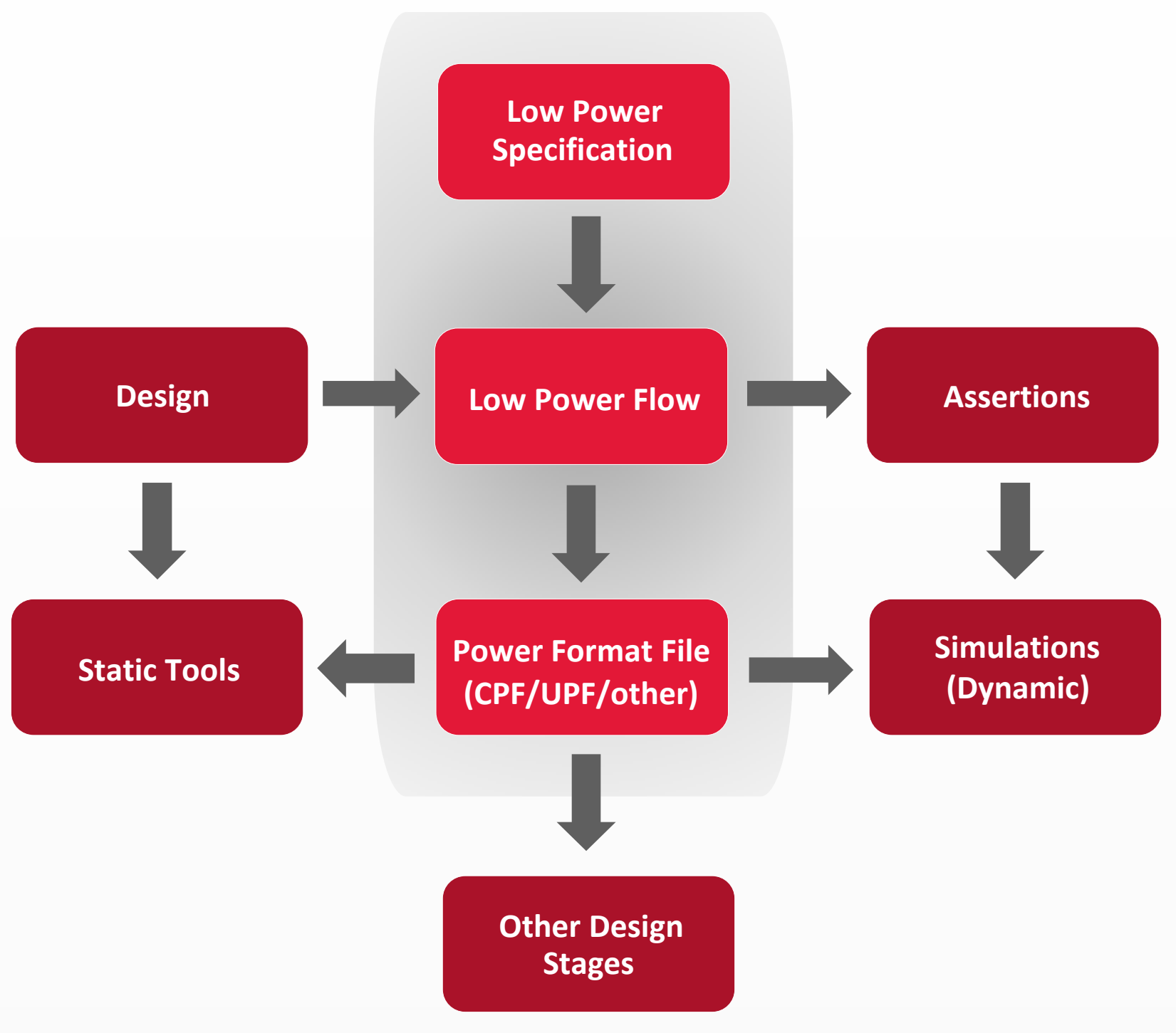
Example: Configuring power format file for multiple tools.

```
# Grab values from environment
# variables.

set design_path $env(DESIGN_PATH)
set target      $env(TOOL_TYPE)

if {$target eq "dynamic"} {
   <do settings for dynamic tools>
} else {
   <do settings for static tools>
}
```

How Automation Fits into Flow



Example Configuration Snippets

```
# Power Domains

my $dsn = {
 DOMAINS => [
   {
    name         => "aon_domain",
    default      => "yes",
    power_net    => "aon_vdd",
    ground_net   => "vss",
    power_net_volt  => [ "1.0" ],
    ground_net_volt => [ "0.0" ],
     …
   }

   {
    name         => "domain1",
    power_net    => "vdd1",
    ground_net   => "vss",
    power_net_volt  => [ "1.0",
                         "0.0" ],
    ground_net_volt => [ "0.0"],
    blocks     => ["I_A", "I_B"],
    shutoff    => "shut_off_1",
    isolation => "iso_1",
     …
   }
   …
  ]
 …
}
```

```
# State table.

my $dsn = {
 DOMAINS => [
   …
 ]
 POWER_STATES =>  [
  { state => "on",
    value => { aon_vdd => "1.0",
               vdd1    => "1.0",
               vdd2    => "1.5",
               vss     => "0.0" },},

   { state => "dmn1_off",
     value => { aon_vdd => "1.0",
                vdd1    => "0.0",
                vdd2    => "1.5",
                vss     => "0.0"},},

   { state => "dmn2_off",
     value => { aon_vdd => "1.0",
                vdd1    => "1.0",
                vdd2    => "0.0",
                vss     => "0.0"},},
     …
 ]
}
```

```
my $dsn = {

 DOMAINS => […]

 POWER_STATES => […]

 ISOLATIONS => [
     {domain1 => aon_domain,},
     {domain2 => aon_domain,},
 ]

 LEVEL_SHIFTERS => [
     {domain2 => aon_domain,}
 ]
 …
}
```

Example: on how to specify relationships between domains (isolation, level shifter rules)

- Designers can fill out the above template, any missing components will be flagged by the flow.
- Diagnostic messages targeted for particular architecture can be embedded in the setup – This is not possible with CPF/UPF since they are only a medium to control the underlying tools.

Example: The high level specification can be converted into objects and power format files can be auto generated

```
# Pseudo code
…
foreach my $d (@domains) {
 if ($d->get_default) {
   $d->dump_code_snippet($d, "default_domain");
 }
 …
 if (!$d->get_default && !$d->get_shutoff && $d->can_be_switched) {
   error("No shut off condition specified for switched domain %s\n",
   $d->get_name);
 }
…
}
```

Why this approach?

- Standards can change/become less popular.
- Tools may not support certain constructs anymore or newer ones can get added. Whoever is maintaining the platform can track such changes and industry wide trends and make changes to flow.

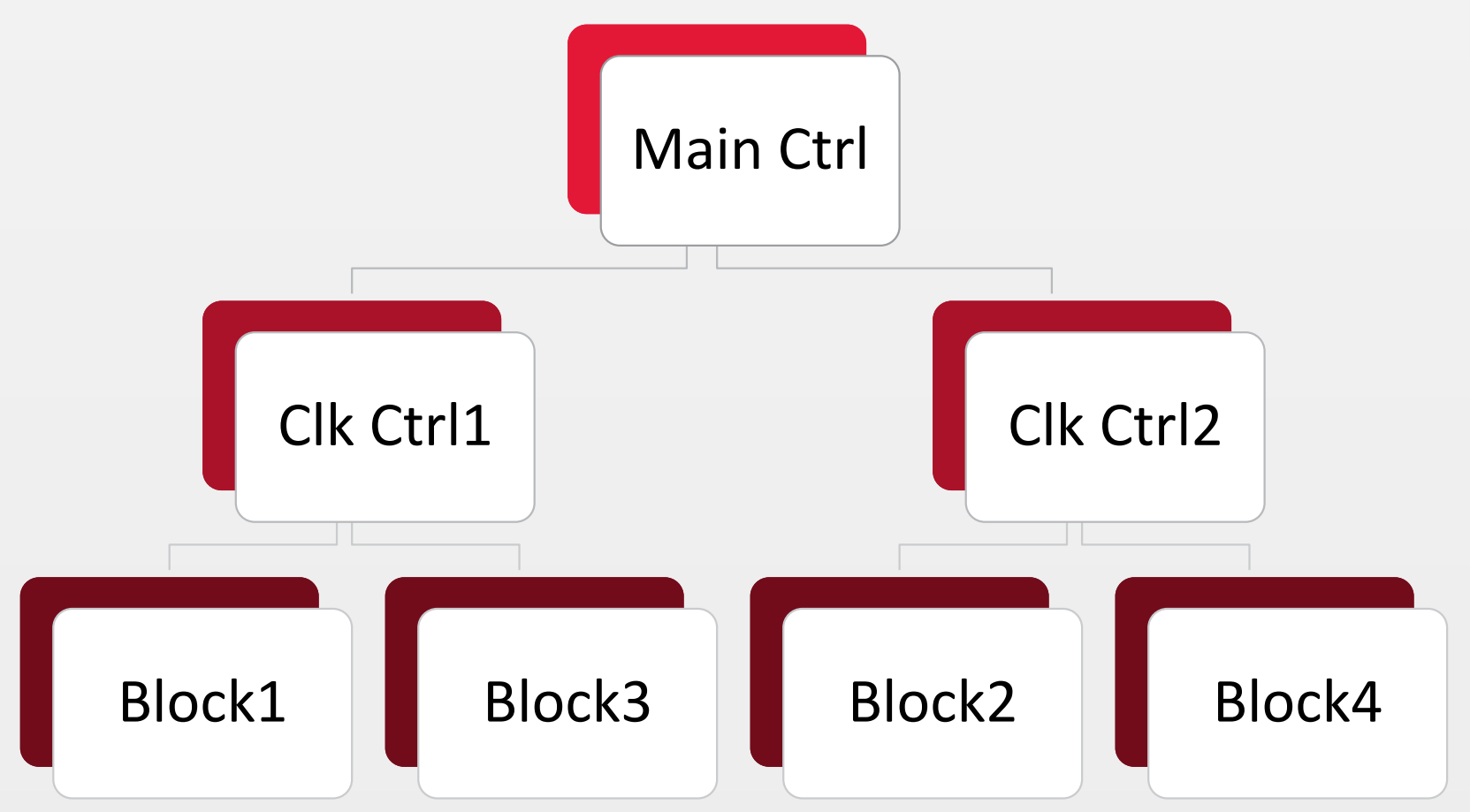
- The flow can have a feature to read in designs, extract design information based on user supplied regular expressions while also performing some design checks based on that.
- Assertions checking low power features can be automatically generated.
- Users can insert tags and headers in the power format file generated, which will be used to automatically stitch together the top level power format file. (Similar to Verilog auto instantiation scripts that designers commonly use)
- NOT a new power format – rather a wrapper which provides a higher level abstraction, built around tools.

Transferring Power Intent to S/W

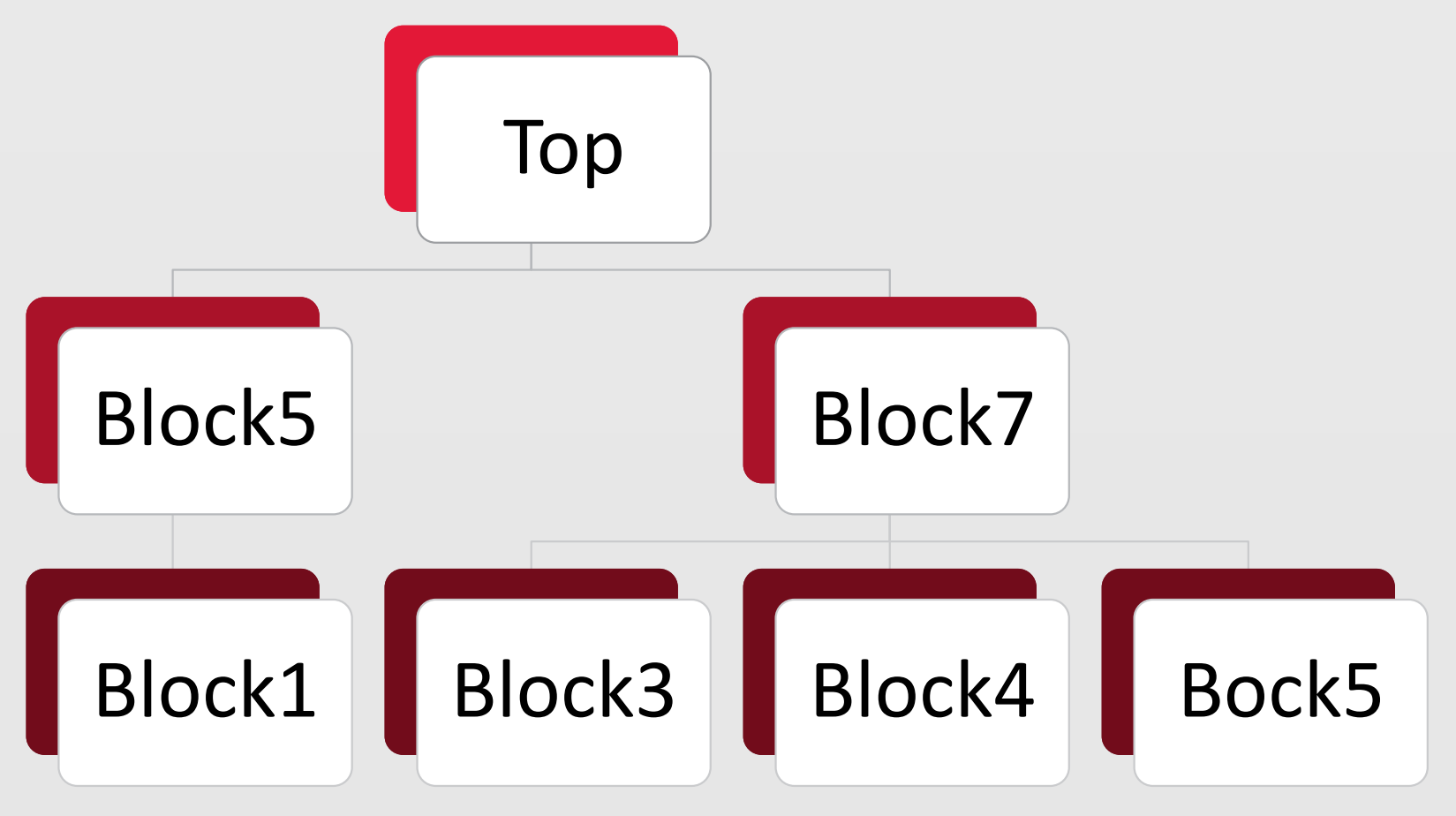
- No automated mechanism exist, this is the weakest link in the chain – No vehicle similar to power format file.
- Software need to follow complex program sequences to manage power.
- Software program sequences might also be dependent on how the specification is implemented in chip.
- Complex specifications can be captured by means of dependency graphs – Dependency between various blocks in a chip in terms of clocks, domains and functionality.

Dependency Graphs

- The dependency graphs can be captured in a data structure. This may be used for generating program sequences that will be consumed by software.

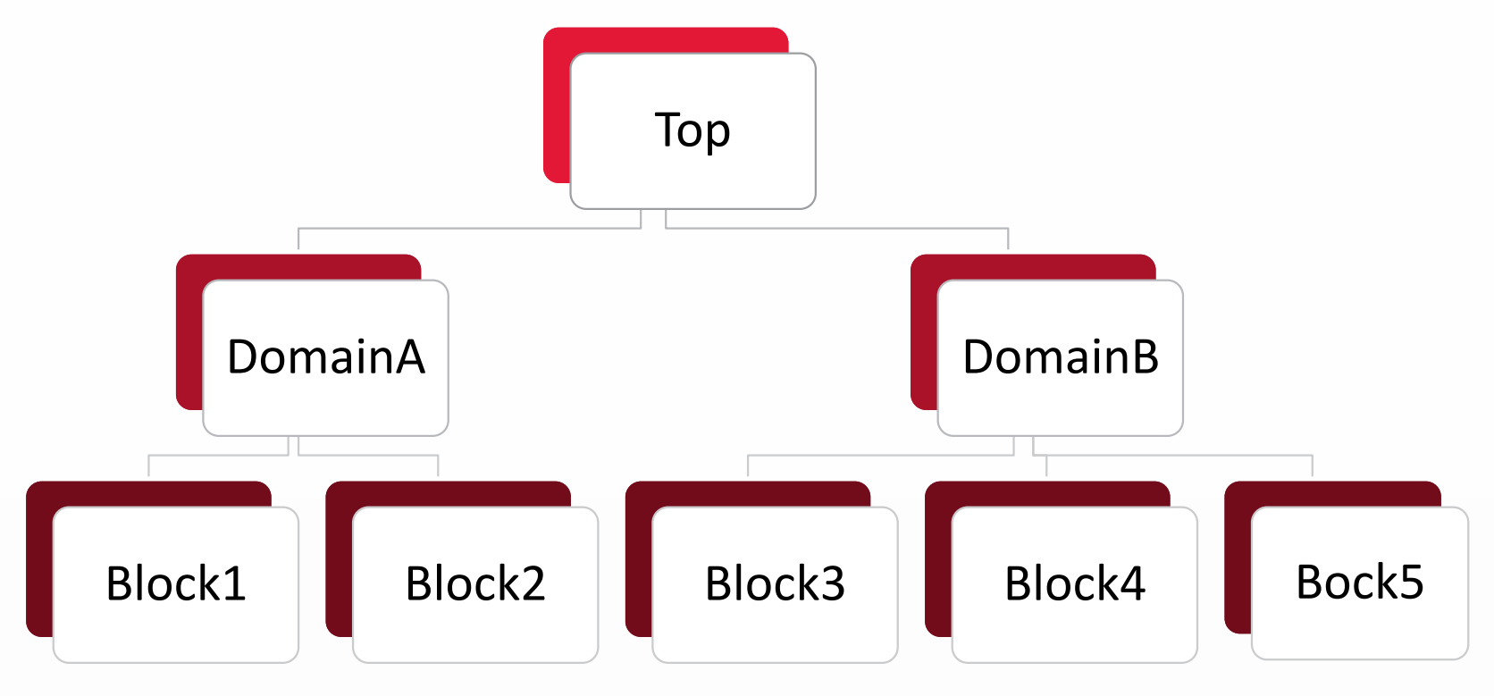


1) Before powering down a block, check for any clock dependency between blocks. Generate the program sequences for same.



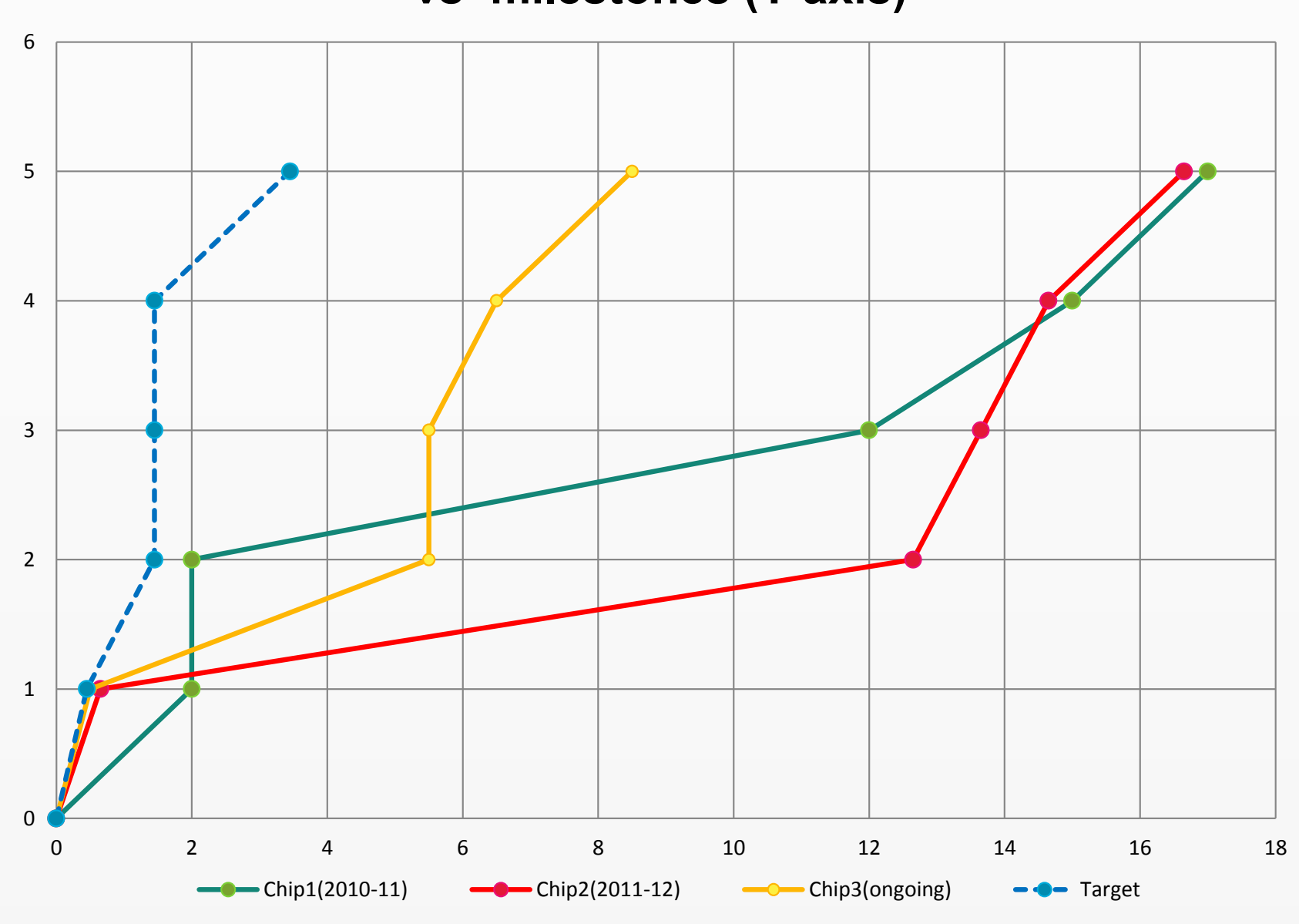
2) Analyze functional dependency between blocks. Sometimes other blocks need to be programmed before power down can begin. Generate program sequences.

3) Check for power domain dependency between blocks. Go back to 1) and 2) until all dependencies are resolved.

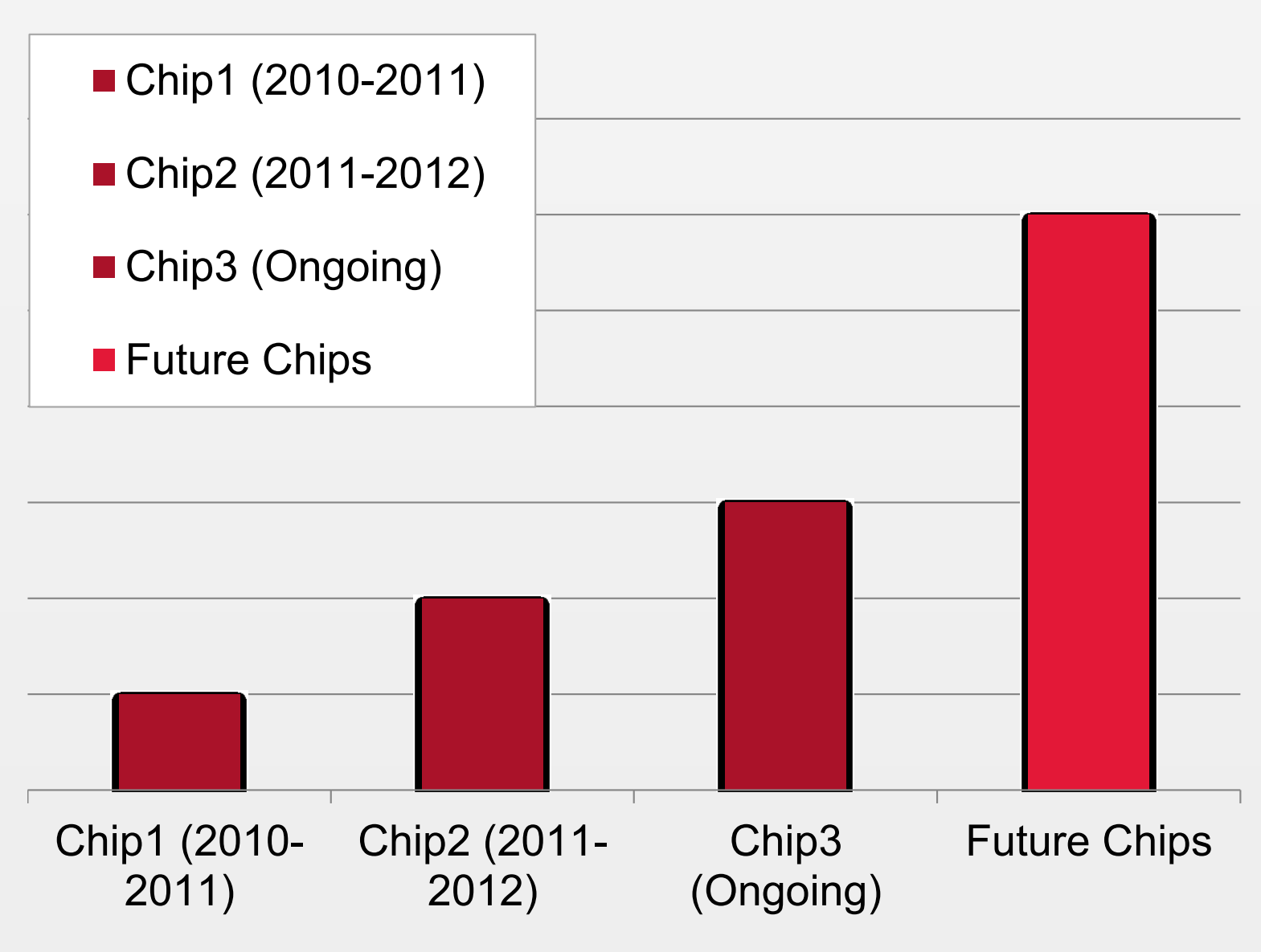


Results

Graph of Development time in weeks (X-axis) vs milestones (Y-axis)



Milestones	Description
0 to 1	Initial specification, gathering requirements
1 to 2	Generating flows, infrastructure and maintaining them
2 to 3	Availability of Simulation ready (dynamic checking) power format file
3 to 4	Availability of power format file for static tool checking
4 to 5 and possibly beyond	Final version of power format file after design/tool/flow fixes



Conclusion

This power format automation flow will be able to detect issues on the low power specification provided by the users, even before the specification evolve into any power format files and run with any tools. Hence , designers can avoid basic errors that they often make. The implementation of such method saves time in design cycle. There will be an initial investment for developing the flow, but the benefits are tremendous. This may lead to fewer bugs, improved design-cycle times, and better time-to-market.