# Low-Power Verification Automation – A Practical Approach

Shaji Kunjumohamed
Broadcom Corporation
Email:shajikk@broadcom.com

Hendy Kosasih
Cadence Design Systems
Email: hkosasih@cadence.com

***Abstract*—**Today's SoCs use a variety of hardware and software power schemes to minimize power consumption. As SoCs continue to grow more complex, so do the various power schemes and the associated verification challenges. The power format files, together with the RTL and netlist, are used to transfer low-power information from one design stage to another and from one tool to another. All together, they enable the creation of complex SoCs, but the shortened design/derivative times pose additional challenges. This paper attempts to explore some common low-power design and verification issues and focuses on some methodologies that can address such problems.

## I. INTRODUCTION

Recently, the growing amount of energy consumed by electronic devices has triggered a rush towards energy efficiency. On one end, we have huge data centers that expend massive amounts of energy to power equipment along with heating and cooling costs, and on the other end, we see users trying to get the best out of their wireless devices in terms of battery life. Designers worry about the reliability of devices damaged by excessive heat. The benefits gained by implementing design power savings have generated a push towards low-power design.

Today, time-to-market requirements, shortened spin-off times, and SoC design complexity pose many challenges. Increased focus on low-power design leads to specifying power intent much earlier in a system design when the design is partitioned into hardware and software. SoC designers use a variety of static and dynamic verification techniques and various tools for implementing low-power design. We also have various power formats (CPF, UPF, and eventually, UPF2.0) to express the low-power intent of a design. These power format files, together with the RTL and netlist, are used to transfer low-power information from one design stage to another and from one tool to another.

The chain is as strong as its weakest link. For example, even though the hardware supports many power management techniques, the final product is only as good as the software that controls the hardware. For complex SoCs, unlike the power format files that capture and transfer the low-power design information between design cycles, no foolproof method exists that would accurately transfer the low-power information from underlying hardware to software.

## II. CHALLENGES

The power format files are relatively new. They are Tcl-based and convey the low-power intent of a design. Many RTL designers have not been sufficiently trained to use power formats. Moreover, each vendor independently adds features to their tools, adding to the confusion. Today's shortened chip spin times imply that design teams cannot spend valuable work-hours training everybody and developing power format files from scratch. For the chip teams, if they are dealing with multiple IPs with power format files, integration can be a nightmare. It can lead to translation errors and the introduction of new bugs because individual IPs may be leveraging heavily on tool/format specific features to express the low-power design intent. Many tools support specific power format versions, so even if a design team adopts a specific format, it can lead to issues. Backward compatibility is not guaranteed, so design reuse can also become problematic going forward. Since the power format files are Tcl-based, the same intent can be coded in many ways. There are no specific coding guidelines associated with them, further complicating IP integration. Currently, there are unification efforts among leading vendors in the industry, but those efforts have not matured around a single standard. Even if a standard is developed, it will still take a while for all the tools and vendors to make the necessary changes and adopt any kind of unified approach. Meanwhile, engineers need to muddle through all of the implementation details. What many chip teams are lacking is a good integration methodology, which is automated enough to follow through, while producing a consistent and bug-free design.

## III. ISSUES WITH DESIGN REUSE

As mentioned earlier, one approach for improving the design-cycle time is design reuse because reused IPs are already tested for reliability. Some of today's large SoCs may have 60-70 blocks, the majority of which are reused. As the complexity of any chip increases, so does the amount of associated reuse. All the IPs may undergo thorough verification by separate core teams, low power, or otherwise. The block design teams may or may not have sufficient expertise to develop the power format files and address the specific verification challenges associated with them. But they may not know how a block is interfaced with the chip. Reuse is going to be the key, so any specific rule constructs that designers develop for a block-level environment should be easily ported to the chip level. Since the same IP team supplies designs to various chip teams, the low-power intent

passed on should be scalable. The integration team does not necessarily know the specific constructs at the block level, which need to be reconfigured at the chip level. Such constructs can be identified easily using appropriate keywords. Chip teams can search for these keywords during their integration efforts.

```
# Example for Rule construct at block
# level

proc  my_core1_generic_rule1{domain1
domain2 pins cells…} {
  <Specific_rules>
}

# Rule construct is called in block
# environment

my_core1_generic_rule1 "BLK_DMN1"
"BLK_DMN2" {..} {..}..

# Same rule construct can be called in
# chip environment, now rule reflects
# conditions in chip level

my_core1_generic_rule1 "CHIP_DMN1"
"CHIP_DMN2" {..} {..}..
```

Example 1

Hardware designs are generally organized hierarchically, where the verification of subblocks can be done independently. Power formats leverage on this divide-and-conquer approach and can also be hierarchically organized. This is the right approach as design complexity increases. The adoption of this methodology should be carefully evaluated in the given environment because it can lead to translation errors during format conversion. This might not work well with tools that do not yet support a hierarchical approach and tools that tend to interpret the constructs differently, resulting in design bugs.

Power format files are Tcl-based, which is advantageous because Tcl is a powerful language. However, this advantage also allows the same specification to be coded in different ways by the user. Also, the relative immaturity and inconsistency of tool support means good coding guidelines have yet to be developed. Any coding guideline will improve readability, maintainability, and compatibility of the code. The following are examples of coding guidelines:

1. Naming convention of variables in Tcl code: how the names should reflect objects that are being referred to.
2. Naming convention for any reusable components.
3. Constructs/methods that are of common use should be wrapped into a procedural library and reused.

```
# Examples

#  supply nets with 1.0 volt
set     supply_net_1p0 [list]
lappend supply_net_1p0 TOP_VDD1P0_0
lappend supply_net_1p0 TOP_VDD1P0_1
..

# Grab values from environment varables.
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>
}
..
```

Example 2

Most companies may have already developed complex environments and flows for running simulations, as well as methodologies to invoke a variety of static and dynamic tools. In order to reduce errors, the same power format file must be used across multiple tools. The following are examples of commonly found issues.

1. Some tools used in a design-cycle stage may not support a power format file syntax that tools earlier in the design cycle had supported.
2. Tools from different vendors interpret power format files differently, such that additional tweaks in the power format file may be required to make the tool interpret format correctly.
3. Tools may need many tool-specific options or pragmas embedded in the power format files.
4. The design names or hierarchies within the power format files may need to be changed to be compatible with a variety of environments, especially in the case of design reuse.

The flow can pass various parameters into the Tcl through environment variables. As shown in Example 2, the format file then can use such parameters to configure itself for a particular tool or design. Adherence to good coding guidelines is extremely useful while automating low-power environments.

## IV. POWER FORMAT FILE AUTOMATION

Many companies have internally developed robust flows for simulating designs with a variety of simulators. These flows allow the users to launch the simulations without worrying about the underlying simulators, operating systems, and other licenses involved. We can extend the same approach to handling power format files. The power format

files typically define the power intent in terms of power domains and relationships between them. This can be quite challenging as chip complexity increases. Similar to the approach for simulators, the low-power methodology should be able to handle most of the underlying issues associated with power format files. Simulation flows are able to handle simulators from various vendors; therefore, low-power flows should be able to handle different power formats like CPF and UPF or any derivatives of them. Users should be able to specify the chip level or block level power intent in an easy to read format, and the flow should be able to do the rest of the work. It should be able to dump out the complete power format file, as chosen by the user as either CPF or UPF.

Figure 1 represents the idea described above. The input to the low-power flow is the low-power specification (or a configuration file) and some design information. The flow dumps out the power format file, which can be used either by static or dynamic tools to verify the power intent of the design.

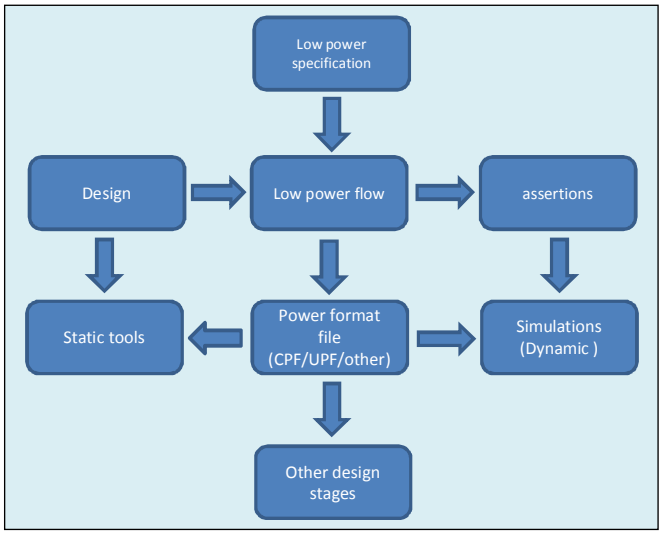



Figure 1.

For a typical design, the variables that designers have in hand for specifying the low-power intent are limited in number. The following is an example list of such variables with which a power format file can be constructed. (This list is not exhaustive.)

1. Number of power domains, domain names, supply nets, and voltage states associated with each supply net in the design.
2. Blocks associated with each power domain.
3. Shut-off condition/isolation conditions for each domain.
4. Power state table of the design.
5. Any domain-specific or pin-specific rules involving isolations, level shifters, etc.
6. Information on power switches, state retention elements, etc.

Whether it is a block-level or top-level low-power design, the list stays pretty much the same. A power format generally captures the above information and assembles it in Tcl for the tools to read. Whether it is CPF or UPF, the hardware implementation envisioned by the designer should be the same.

With the above list, a lot of information can be deduced. For example, from the voltage states of the power nets associated with each domain, it can be determined which domain is always on, switchable, etc. This can be cross-checked against other given information, such as whether a shut-off condition is specified for all switchable domains. Although these checks are simple and will be caught one way or the other later, the flagging of such errors by the flow itself will aid the designer to specify a better low-power specification. It is observed that simple issues that initially go unnoticed, but get caught later in the design cycle, require additional time and effort to fix.

Given the above variables, one can specify power intent in many different ways, especially the isolation/level shifter relationship between different domains and associated blocks. Such a specification can be easily expressed in a scripting environment such as PERL. There are various ways to express design intent. Below is example code.

```
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 => [“instA”, “instB”],
     shutoff => “domain_shut_off_1”,
     isolation => “domain_iso1”,
     …
   }
   {
     name => “domain2”,
     power_net => “vdd2”,
     ground_net => “vss”,
     power_net_volt => [ “1.5”, “0.0” ],
     ground_net_volt => [ “0.0” ],
     blocks => [“instC”, “instD”],
     shutoff => “domain_shut_off_2”,
     isolation => “domain_iso2”,
     …
   }
  …
 ]
 …
}
```

Example 3: An example of a configuration/power spec.

The power states in the design can be specified by the following code:

```
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”},},
…

 ]
}
```

Example 4

Similarly, we can come up with generic methods for specifying other relationships with power domains as below (for example, which domain crossings need to have isolation).

```
my $dsn = {
 DOMAINS => […]
 POWER_STATES => […]
 ISOLATIONS => [
     {domain1 => aon_domain,},
     {domain2 => aon_domain,},
 ]
LEVEL_SHIFTERS => [
     {domain2 => aon_domain,}
 ]
 …
}
```

Example 5

The advantage with this approach is that any user can fill in the given template easily and generate the corresponding power intent. The flow can check the specification for errors and notify users immediately; for example, when the flow is missing any components, there are mismatches in power net names between different closures, whether the state table is specified correctly, etc. The flow can even embed checks and diagnostic messages that are targeted for particular low-power architectures, and notify users immediately to correct the same. Such are difficult to incorporate with power format files, since they are only a medium to configure the underlying tools. There, these errors will be flagged only when some expert runs the tool with a design, and it might take a while to resolve such issues.

The platform can be implemented with any good scripting language that supports closures and object-oriented styles. For example, components from the above specification can be easily converted into arrays of objects and convenient data structures. Once this is done, the checking and corresponding autogeneration of the power format file becomes trivial.

```
# 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);
 }
…
}
```

Example 6

One obvious advantage of such a flow is guaranteed compatibility of the autogenerated power format file with a tool such as a simulator. In the current environment, where there are multiple power format files, nobody is sure on the outcome of any unification efforts. Certain standards can disappear, many commonly used constructs in a format can suddenly become redundant, or tools might not support some features going forward. The proposed methodology helps to avoid dependency on any specific language and tool sets while offering greater flexibility. Whoever maintains the flow should track what the various tools support and industrywide trends. The flow can be adjusted so that the same input specification can autogenerate power format files having newer constructs as demanded by tool vendor. As discussed earlier, reusability is one approach used to improve design-cycle times. The proposed flow addresses most of the reusability issues of IPs.

There can be instances where minor changes in the design, such as changes in the names of power nets and power domains, etc., can lead to the modification of the power format file in multiple places. In the above setup, the user would only need to make a corresponding change in the input configuration file at a single place. Another example is a case where the designer tries to specify isolation or a level-shifter behavior at the I/O of a subblock, based on some naming convention or regular expression pattern matching. While power format files from different vendors have the capability to apply constraints based on regular expressions, the exact syntax may vary between power formats. The flow can be configured to read in the block-level RTL files, extract the names of ports that need special handling, and put it into an autogenerated power format file as an I/O net-based rule, rather than a regular expression based rule.

During the development cycle of a chip, the design will be in a continuous state of flux. For example, block instance

names can often change, new blocks can get added or removed from power domains, power net names and power modes can change, and chip-level I/Os can change. The power format file needs to be kept up to date with the changing design because all tools using the power format file will consider it to be golden. Manually updating a power format file each time there is a change is cumbersome and can introduce new bugs. To reduce the risk, the process can be automated. This is done by capturing a *time capsule*.

1. The flow can be made in such a way that it can have the ability to read the top-level RTL of the chip, where the blocks that are partitioned to go into different power domains are instantiated. The flow hence gets the list of instances.
2. Depending on a regular expression or manual mapping provided by users, the flow can determine which instance goes to which domain. This information can be handed off to the low-power flow. If the appropriate interfaces are already built, the flow can take this information and make it part of the low power specification. Users do not need to explicitly spell out the instances in various power domains in configuration files again, as is done in Example 3.
3. The list generated in (2) can be stored in a file or database, which can be called a *time capsule*. The next time the flow is run, the time capsule during that run can be compared with the earlier time capsule. Anything different can be reported as errors. All mismatches need to be reviewed because they can indicate whether the design has changed.

The implementation of such a check is important since it can capture bugs that can be overlooked otherwise. A typical example can be the introduction of a new block in a specific power domain. If the power intent is not updated, the block may go to the default domain. The above check will catch such a scenario.

Some simulators, such as Cadence's Incisive Enterprise Simulator, can generate a report of low power intent in a design once the design and the power format file are loaded in the simulator. This report can also be used as a time capsule to check for any low-power design changes between different simulator runs.

A low-power verification environment can use assertions for automated checking of power sequencing. Some simulators can automatically generate assertions from the power format files. While these might satisfy most of the verification requirements, there can be cases where engineers need to write custom assertions for checking low power. An example is a case where a domain has multiple isolation controls and some relationships between them. In some other cases, assertions need to be enabled only after an initialization sequence to avoid false violations, so finer control may be desired. Another common scenario is the checking of the relationship between shut-off conditions and power nets associated with each power domain. Furthermore, a user might want to reuse all these assertions in the final physical netlist simulations. Since the assertions that need to be generated are common for all power domains, the flow can be configured to autogenerate assertions as well, when it generates power format files. Such assertions can be used in any simulation environment.

One of the difficulties that a team encounters is in writing the top-level power format file. The power format file can have many components, or it can be organized hierarchically, depending on the selected power format. Using the autogeneration platform, users can insert tags and other headers in the block-level format files. The flow can leverage on this for automatically generating the top-level power format file. This method is analogous to the autogeneration of top-level RTL code from the block level, which many chip-level teams are currently using. As in the case of RTL, for this to work, the interface of the block-level power format file needs to be consistent, and the autogeneration of block-level power formats must also be consistent. The low-power flow does not need to be 100% perfect, but it should be flexible enough to be tailored to suit various architectures and user needs.

The intent of the method proposed here is not to introduce a new power format. The platform can be considered as a wrapper built around various tools. The number of parameters that designers need to specify to come up with a CPF/UPF power format file is not that many. These can be captured in a template, similar to what is being proposed earlier. The interface given to designers through this platform can be of much higher level of abstraction than the actual power format file description of the design. This can be paralleled to scripts that some design teams use to generate RTL designs in Verilog. In this case, by changing a handful of parameters in the scripts, designers generate different design configurations. These parameters and the templates that capture them serve as a high level abstraction of such complex designs, but cannot serve as a replacement for the actual RTL design or Verilog language.

## V. TRANSFERRING LOW-POWER INTENT INTO SOFTWARE

The desired low-power performance of a device not only depends on the underlying hardware but also on how well the software uses the low-power features of the hardware. Software developers create low-level APIs that interface with the hardware. These APIs are used to build other complex software applications. In developing the software, the developer tries to determine the program sequence that configures the chip. Developers primarily use hardware register interface information to determine the required program sequences. In some cases, the software is tested on a hardware emulation platform until the real parts are available.

Software runs in real time. It can create many untested chip scenarios that can crash the system. Today's chips use a

variety of complex techniques to save power. This means that the sequences that use the low-power features also tend to be complex. For example, software may want to power down a particular section or feature in hardware to save power. There can be many associated dependencies. To power down a block, software must power down a domain that might include other blocks. In some cases, software must ensure that blocks in the middle of performing a function are allowed to finish before powering down. In such cases, the software might need to wait until the block reaches a known state. There can be scenarios where the software configuration changes based on some status flags. In some power-down modes, the clocks going into the hardware blocks to be powered down need to be stopped. This can affect other blocks that use the same clock tree. Hence, powering down a block means that the user must follow a variety of software sequences. These sequences tend to be complex, and they can depend on how the specification requirements are implemented in the chip. A conventional specification may not cover all these details. This calls for a better understanding of the chip's low-power architecture by software developers and the need to interact more with designers. Most of the time this is not practical; it can add to the development time of the software and introduce more bugs because of perception differences. As in hardware design, there is no vehicle similar to the power format file to transfer low-power design intent. However, this issue can be addressed by a suitable methodology.

As discussed earlier, software must account for various low-power design dependencies between different hardware blocks to create program sequences. These dependencies can be clock related power-domain related, or functional. They can be captured by means of dependency graphs, which are a hierarchical representation of dependencies between various blocks in terms of clocks, domains, and functionality. They can be consolidated into a single entity, or they can be separate, depending on the implementation. Graph relationships must be updated by designers in the design stage. The graphs can be considered a part of design because the information is passed on to another stage of product development, just as with power format files.

Figure 2 shows an example of a hierarchical relationship between different blocks. A box represents the configurations that must be done on that block to attain a particular hardware state. If a block is connected to a block above it, there is a dependency. For example, if software needs to power down Block1, the following must be done:

1. From Graph1, configurations must first be done at the top level and then on Block5 to prepare Block1 for a power-down.
2. From Graph2, the clock to Block1 must be stopped by programming the clk ctrl1 block. It is evident that Block1 and Block3 share common clocks, so Block3 will be affected by stopping the clock. Some configuration must also be done in the main ctrl block.
3. From Graph3, it looks like, in order to power down Block1, DomainA needs to be powered down. But this will also power down Block2, so the prerequisites for powering down Block2 also should be satisfied from Graph1 and Graph2, so that subsequent sequences should be constructed by traversing the same.

From the above explanation, it becomes clear that power-down sequencing of a block may get complex. After the sequencing is constructed, the flow can output a pseudocode for software developers to use. To implement the pseudocode, any scripting language or libraries that support hierarchical data structures can be used.

Software teams develop software on emulation platforms before real hardware is available for use. Software teams can use emulation platforms to test different power-down sequencing types. Combined with randomization, this can be a powerful tool to check the functionality of a design. Identifying bugs at this stage is better than finding issues in real silicon.

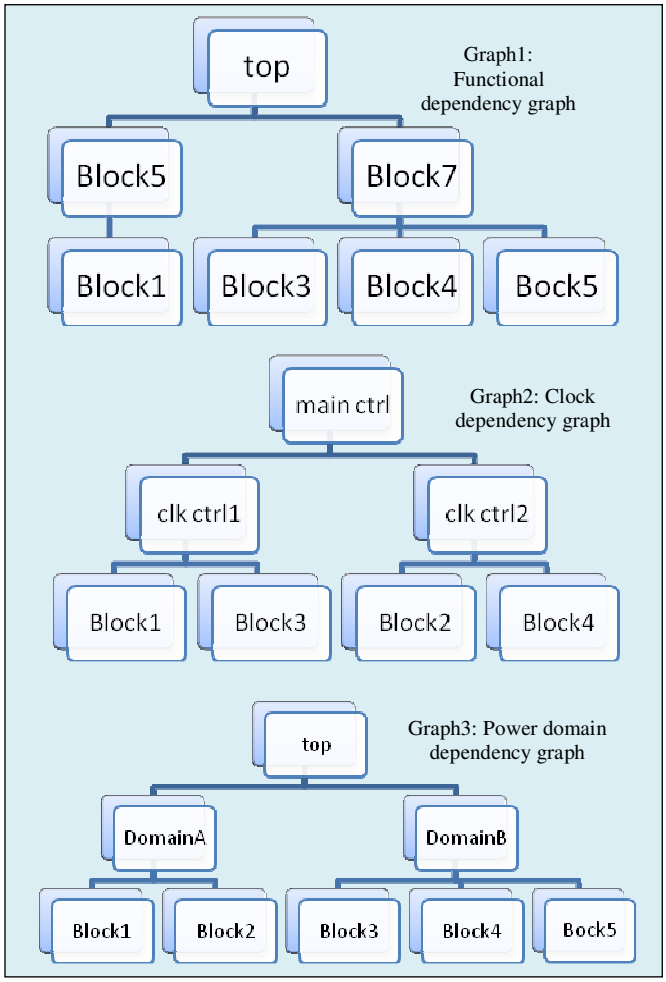


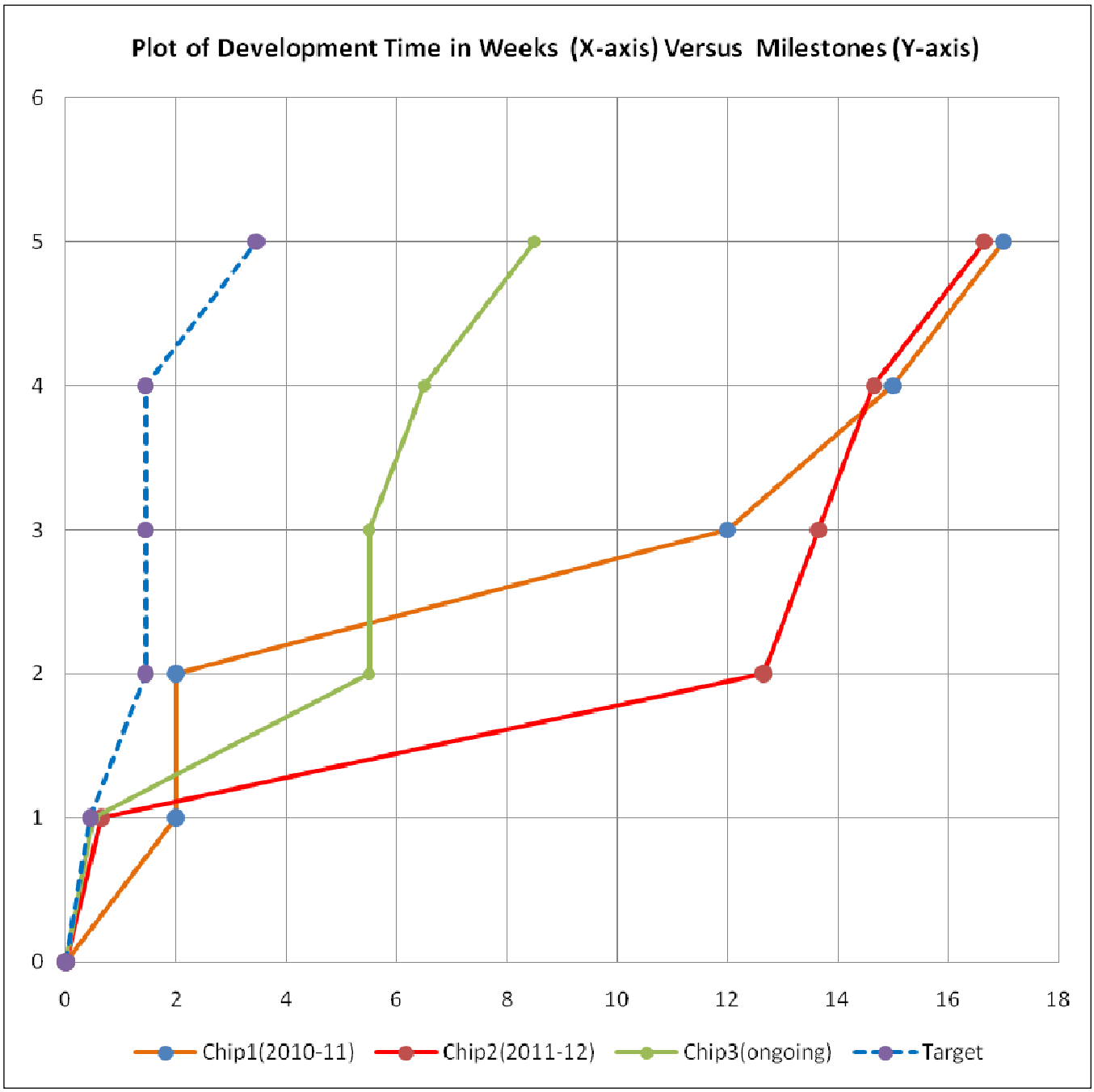

Figure 2.

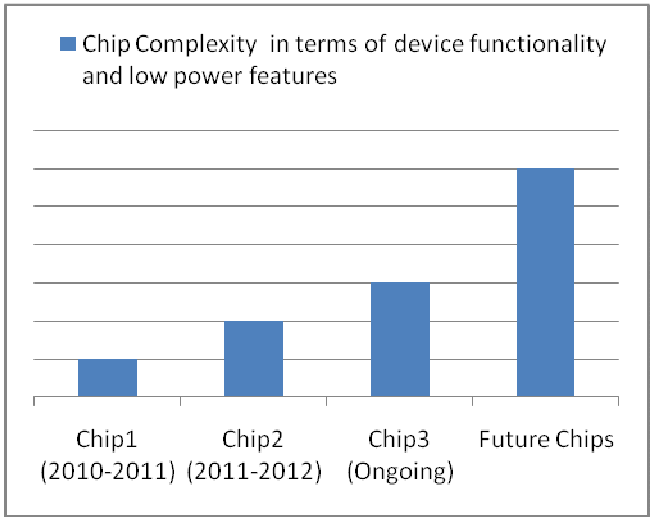
## VI. IMPLEMENTATION DATA

A graph which plots the time taken to construct a power format file against the project milestones for various projects is presented below.

Chip1 represents the initial case, where the power format file was introduced to the team as a tool for expressing low-power intent. Subsequently, there was a learning curve in terms of understanding the methodology and tool. Here, the entire power format file was hand-coded. There was significant ramp-up time.

Chip2 is an example where low-power automation was developed and introduced into the design flow. Here, power format files were generated automatically. Script infrastructure and flow took some time to develop. In the case of Chip3, the automation environment from Chip2 was reused, and there were some environment changes as well. Overall cycle time for developing an accurate low-power format file improved a lot when compared to other projects. The Target is the goal to strive for as the low-power flow matures.






| 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 |

## VII. CONCLUSION

Many issues encountered by engineers setting up a low-power test bench and verification environment have been discussed. The central theme is to come up with a good low-power methodology for users, so they will not have to deal with the power format files directly. The power format auto generation can detect issues with input low power specification provided, hence avoiding basic errors that designers often make. The result of adopting this methodology is consistency between power format files provided by different design teams and easiness in managing and integrating them into chip level environment. Different methods to capture and hand off low-power intent are presented. The implementation of such methods have saved us time in our 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.

There may be other solutions besides those proposed in this paper.

## REFERENCES

[1] Robert Meyer, Joel Artmann: “Creating a Complete Low Power Verification Strategy using the Common Power Format and UVM”, DVCon 2012.

[2] Amit Srivastava, Rudra Mukherjee, Erich Marschner, Chuck Seely, Sorin Dobre: “Low power SoC Verification: IP Reuse and Hierarchical Composition using UPF”, DVCon 2012.

[3] http://www.powerforward.org