Unleashing the Full Power of UPF Power States

Erich Marschner - Mentor Graphics
John Biggs - ARM Ltd.
Agenda

- Overview of UPF Power States
- Issues with UPF Power States
- What Power States Represent
- Power State Refinement
- Active and Current Power States
- Power State Definition Rules and Guidelines
- Power State Enhancements in UPF 3.0
What is UPF?

• An Evolving Standard
  – Accellera UPF in 2007 (1.0)
  – IEEE 1801-2009 UPF (2.0)
  – IEEE 1801-2013 UPF (2.1)
  – IEEE 1801a-2014 UPF (2.2)
  – IEEE 1801-2015 UPF (3.0)
    • (In development now)

• Based upon Tcl
  – Tcl syntax and semantics
  – Can be mixed with non-UPF Tcl

• And HDLs
  – SystemVerilog, Verilog,
  – VHDL, and (in 3.0) SystemC

• For Power Intent
  – To define power management
  – To optimize power consumption

• For Power Analysis (in 3.0)
  – Component Power Modeling

• For Verification
  – Simulation or Emulation
  – Static/Formal Verification

• For Implementation
  – Synthesis, DFT, P&R, etc.
Power Mgmt Concepts

• **Power Domains**
  – Independently powered regions
  – Enable application of different power reduction techniques in each region

• **State Retention**
  – To save essential data when power is off
  – To enable quick resumption after power up

• **Isolation**
  – To ensure correct electrical/logical interactions between domains in different power states

• **Level Shifting**
  – To ensure correct communication between different voltage levels
Power States in UPF 2.0

- Apply to **power domains** and **supply sets**
- Represent
  - Capacity of a supply set to provide power
  - Operating mode of a component that consumes power
- **Power states** are actually independent **predicates**
  - Object is in a state iff its defining expression = True
  - An object can be in multiple states at once (not mutex)
    - Enables modeling abstract states, state refinement
- **Modeling power state relationships correctly** is critical
add_power_state

• Defines power states of supply set or power domain
• Power states have
  – a name
  – a logic expression
• Supply set power states can also have
  – a supply expression
  – a simstate (indicates simulation behavior in this state)
• Power states may be legal or illegal
• Power states may be defined incrementally (-update)
Supply Set Power States

Power states for the primary supply set of power domain PDA

```
add_power_state PDA.primary -supply \\
    -state {ON -simstate NORMAL \ 
        -logic_expr {SW_ON} \\
        -supply_expr { power == {FULL_ON 0.8} && \\
                        ground == {FULL_ON 0.0} } } \\
    -state {OFF -simstate CORRUPT \ 
        -logic_expr {!SW_ON} \\
        -supply_expr { power == OFF || \\
                        ground == OFF } } \\
    -update adds another state definition
```

```
add_power_state PDA.primary -supply -update \\
    -state {SLOW -simstate CORRUPT_STATE_ON_CHANGE \ 
        -logic_expr {SW_ON && interval(clk posedge negedge)>= 100ns} \\
        -supply_expr { power == {FULL_ON 0.8} && \\
                        ground == {FULL_ON 0.0} } } 
```
Updating Power States

Power states for the primary supply set of power domain PDA

```
add_power_state PDA.primary -supply \\
  -state {SLOW -simstate CORRUPT_STATE_ON_CHANGE \ 
    -logic_expr {SW_ON && interval(clk posedge negedge)>= 100ns} \ 
    -supply_expr { power == {FULL_ON 0.8} && \ 
      ground == {FULL_ON 0.0} } } }
```

```
add_power_state PDA.primary -supply -update \ 
  -state {SLOW -supply_expr { nwell == {FULL_ON 1.0} } } }
```

```
add_power_state PDA.primary -supply \\
  -state {SLOW -simstate CORRUPT_STATE_ON_CHANGE \ 
    -logic_expr {SW_ON && interval(clk posedge negedge)>= 100ns} \ 
    -supply_expr { power == {FULL_ON 0.8} && \ 
      ground == {FULL_ON 0.0} && \ 
      nwell == {FULL_ON 1.0} } } }
```

-"update modifies existing state definition"
Power Domain Power States

Power states for the power domain PDA

```
add_power_state PDA -domain \
    -state {RUN} -logic_expr { primary == ON && !sleep } } \
    -state {SLEEP} -logic_expr { primary == ON && sleep } } \
    -state {SHUTDOWN} -logic_expr { primary == OFF } }
```

defined in terms of PDA.primary power states

Power states for the power domain PDTOP

```
add_power_state PDTOP -domain \
    -state {S1} -logic_expr { PDA == RUN && PDB == RUN } } \
    -state {S2} -logic_expr { PDA == SLEEP || PDB == SLEEP } } \
    -state {S3} -logic_expr { PDA != RUN && PDB != SHUTDOWN } }
```

defined in terms of PDA, PDB power states
Issues with UPF Power States

- **Non-mutual exclusion**
  - Can be unintended and unwanted
    - Power states are NOT “states” in the general case

- **Unrestricted defining expressions**
  - Can be arbitrarily complex and difficult to understand
    - Contributes to unintended state overlap

- **Update can cause unexpected side effects**
  - Can change the meaning of a state used in defining some other state

- **Update semantics are not sufficient**
  - Needs to be branching (hierarchical), not just linear

---

Need to clarify principles of power state definition and define rules to enforce them

Need a better method than -update to support power state refinement
What is a “Power State”?

A named set of object states

- Each state has a “defining expression”
- It refers to values of the object’s “characteristic elements”
- Some characteristic elements may be don’t cares for a given state
- Multiple object states may satisfy the defining expression

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>S1</strong></td>
<td><code>A==1'b0</code> &amp; <code>B==1'b0</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0 0 0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0 0 1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0 1 0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0 1 1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td><strong>S2</strong></td>
<td><code>(A xor B) ==1'b1</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1 0 0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1 0 1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1 1 0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1 1 1</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

**don’t cares**
Power States as Sets

- Largest set = all possible object states
- Some of these states are legal states
- **Subsets** represent “more specific” (or more refined) power states
  - Refinement creates subsets by adding more conditions to satisfy
  - The innermost subset containing a given object state represents the most specific power state of that object
- **Supersets** represent “more general” (or more abstract) power states
- Non-overlapping subsets represent mutually exclusive power states
- Subset containment implies **non-mutex** power states (subset => superset)
Applying These Concepts

- Same level states must be mutually exclusive
- Superstates contain (overlap) substates - non-mutex
- These principles allow state partitioning, hierarchical refinement

Fundamental Power States
- Working
  - G0: Working
- Sleeping
  - G1: Sleeping
    - S1: Power on Suspend
    - S2: CPU off
    - S3: Standby
    - S4: Hibernation
- Soft Off
- Mech Off

Refined Power States (Substates)
- S0: Awaymode
- S1: Power on Suspend
- S2: CPU off
- S3: Standby
- S4: Hibernation

{All States} represents the set of all possible states; the fundamental states are subsets of {All States}
Refinement by Derivation

• Define new state(s) instead of updating existing state
  – Avoids side effects
  – Allows for branching refinement

• Define new state by adding another condition
  – Amounts to subsetting object’s state space

• Group related states as derivatives of same parent
  – Enables power state differentiation as design evolves

• Ensure all derivatives of same state are mutex
  – Enables identification of a unique “current state”
Example

Power states for the CPU

```plaintext
add_power_state CPU -domain \
    -state {UP \n            -logic_expr {primary==ON }}

add_power_state CPU -domain -update \
    -state {RUN \n            -logic_expr {CPU==UP && busy==1}}

add_power_state CPU -domain -update \
    -state {IDLE \n            -logic_expr {CPU==UP && busy==0 && clkg==1}}

add_power_state CPU -domain -update \
    -state {CLKGATED \n            -logic_expr {CPU==UP && clkg==0}}

...
Classes of Power States

- **Definite Power State**
  - represents a *specific binding* of values to object elements
  - has a defining expression that
    - contains only operators `==` and `&&`
    - and refers only to other Definite states (of same or other objects)

- **Indefinite Power State**
  - represents a *set of possible bindings* of values to object elements
  - has a defining expression that
    - contains operators `!`, `!=`, or `||`
    - or refers to an Indefinite state (of the same or another object)

- **Deferred Power State**
  - a Definite State that is not yet fully defined
  - has a name but no defining expression yet
Examples

• **Definite Power State**
  - \{power==FULL_ON && ground==FULL_ON\}
  - \{primary==OFF && retention==ON\}
  - \{CPU==Running && Memory==Operational\}

• **Indefinite Power State**
  - \{CPU==Running && !(Memory==Sleep)\}
  - \{primary!=ON && retention==ON\}
  - \{power==OFF || ground==OFF\}

Similar to PST states

Can be interpreted as a set of assignments

Cannot be interpreted as assignments without making choices

Negation requires closed-world assumption

This additional power (and complexity) is new with add_power_state
PSTs and Definite States

• Power State Tables (PSTs) use Definite States

```plaintext
create_pst PST1 -supplies { VDD  VDDsw  VSS  }
add_pst_state S0 -pst PST1 -state { on10  *     on00 }
add_pst_state S1 -pst PST1 -state { on10  off   on00 }
add_pst_state S2 -pst PST1 -state { on10  on08  on00 }
add_pst_state S3 -pst PST1 -state { on08  on08  on00 }
```

Implies defining expressions:

```plaintext
S0 = { VDD == 0n10 && VSS == on00 }
S1 = { VDD == 0n10 && VDDsw == off && VSS == on00 }
S2 = { VDD == 0n10 && VDDsw == on08 && VSS == on00 }
S3 = { VDD == 0n08 && VDDsw == on08 && VSS == on00 }
```
PSTs and Refinement

• Power State Tables (PSTs) can model Refinement

```plaintext
create_pst PST1 -supplies { VDD  VDDsw  VSS  }
add_pst_state S0 -pst PST1 -state { on10  *     on00  }
add_pst_state S1 -pst PST1 -state { on10  off   on00  }
add_pst_state S2 -pst PST1 -state { on10  on08 on00  }
add_pst_state S3 -pst PST1 -state { on08 on08 on00  }
```

Implies defining expressions:

```plaintext
S0 = { VDD == 0n10 && VSS == on00  }
S1 = { PST1== S0 && VDDsw == off  }
S2 = { PST1== S0 && VDDsw == on08  }
S3 = { VDD == 0n08 && VDDsw == on08 && VSS == on00  }
```

(Conceptual; not legal)

But PSTs are not hierarchical, cannot include control expressions, and only refer to supply ports/nets
Active and Current States

Power states for the CPU

- **UP**: \{primary==ON\}
- **RUN**: \{CPU==UP && busy\}
- **P0**: \{CPU==RUN && perf==2'b00\}
- **P1**: \{CPU==RUN && perf==2'b01\}
- **P2**: \{CPU==RUN && perf==2'b10\}
- **IDLE**: \{CPU==UP && !busy && clkg==1\}
- **CLKGATED**: \{CPU==UP && clkg==0\}
- **DOWN**: \{primary==OFF\}
- **RET**: \{CPU==OFF && retention==ON\}

If a state is active, every abstraction of that state is also active.

The most refined active state is the current power state.
Undefined and Error States

• **UNDEFINED**
  – Represents all other states not explicitly defined
  – Useful for early stages in the flow
  – Active (and current) only if no other state is active

• **ERROR**
  – Catches unintended non-mutex states
  – Necessary for dynamic verification
  – Active (and current) when two different fundamental states are active
Power State Definition Rules

You can:

- Define (legal) states
- Define explicitly illegal states
- Specify -complete to make undefined states illegal
- Define **Definite** subset states (existing state AND new condition)
- Define **Indefinite** superstates ([X]OR of existing states)
- Mark existing legal states illegal

You cannot:

- Create legal states in illegal state space
- Define superstates that are the AND of two or more existing states
Adopting This Approach

- In UPF 2.x
  - Define mutually exclusive fundamental states
  - Use Definite or Deferred states wherever possible
  - Use Refinement by Derivation to create refined states
  - Ensure that all refinements are mutually exclusive
  - Use more conservative simstates for more refined states
  - Define UNDEFINED, ERROR states for all objects
Changes Coming in UPF 3.0

• Predefined power states
  – UNDEFINED, ERROR for all objects
  – ON, OFF for supply sets

• Current State precedence rules
  – Replacing existing simstate precedence rules

• New name syntax for power state refinement
  – Dotted names for power states (e.g., UP.RUN.P0)

• Clarification of state transition semantics
  – Based on active and current state definitions

• Error checks for new power state concepts
Summary

• UPF 2.0 `add_power_state` is a powerful command
  – Perhaps too powerful if used without care

• Power states should be defined methodically
  – Should be mutually exclusive at any given level
  – Should use refinement by derivation to refine states

• Refinement by derivation works in UPF 2.x
  – Can be used now to create a power state hierarchy

• UPF 3.0 will reinforce this methodology