Flow Engineering for Physical Implementation: Theory and Practice

Steve Golson
Trilobyte Systems

Pete Churchill
Magma Design Automation

June 10, 2008
45th Design Automation Conference
Define: flow

flow *n.*

1. the act or manner of flowing
2. the smooth motion characteristic of fluids
3. anything that flows; stream or current
4. a continuous production
5. the sequence in which operations are performed

from http://education.yahoo.com
Define: audience

- design engineer
- EDA vendor
- manager
- CAD
Define: flow

- **Management perspective**
  
  flow *n.*
  
  a four-letter word often associated with work that can’t be scheduled and seems to never end

- **CAD perspective**

  flow *n.*
  
  a four-letter word representing an endless project whose result is underappreciated and underused by the design community

- **Design engineer perspective**

  flow *n.*
  
  a four-letter word representing a necessary tool which you are not given time to work on, even though what you have is not adequate or complete

- **EDA vendor perspective**

  flow *n.*
  
  a four-letter word representing all the stuff that customers do with other vendor’s tools, often making us look bad
Flow (our definition)

flow n.

- all the files and infrastructure required to implement a methodology

"The CAD team has created a new flow for us to use on our next design."

- a sequence of tool invocations

"You need to run the flow for clock synthesis."
Define: physical implementation

- “IC design” or “chip design” or “IC implementation”
- Everything from RTL ➔ GDSII
- Everything *other than* functionality
- *Not* writing RTL, *not* functional verification
- Hold function constant, change the representation
Define: engineering

engineering *n.*

- The application of scientific and mathematical *principles* to practical ends such as the design, manufacture, and operation of efficient and economical structures, machines, processes, and systems.

from http://education.yahoo.com
The Principles of Flow Engineering

#1 Flows are hard

#2 Flows are iterative

#3 Flows are automated

#4 Flows fail more than they succeed
Principle #1: Flows are hard

Flows are expensive

- We spend as much on flow as we do on EDA tools themselves

“The IDMs and large fabless companies are spending between $1 and $2 on CAD support activities for every dollar they spend on the tools themselves. In fact, there are nearly as many CAD engineers inside the semiconductor industry as there are in all of the EDA vendors... combined!”

Thomas Harms
Infineon Technologies
SCDsource
12 Dec 2007
Principle #1: Flows are hard

We’ve been designing chips for a long time
Why hasn’t this been solved already?

- EDA perspective:
  - There’s no money in it
  - Vendors can only work with their own tools

- CAD perspective:
  - Too many tools to support them all
  - Too many process nodes/libraries to support them all
  - Customization capabilities are difficult to include
"Principle #1: Flows are hard"

“We build chips, not flows”

- This is only true if you *reuse* a flow
- Flows are only reusable when they are *designed* to be reused
- Making changes and improvements to your flow for reuse will *not* make your current chip project finish sooner...

however your *next* chip will finish sooner...

- but only if you actually reuse your flow and don’t start over from scratch!
Principle #1: Flows are hard

Flow must change/adapt as the design matures

- Prototype flows needed in the beginning
- Production flows needed once implementation matures
- Hopefully we are building the process while the RTL is still being written
Principle #1: Flows are hard

- EDA tools are complex and expensive
- Combining them into a flow is *also* complex and expensive
- Always lots of time pressure
  - it seems there is never any time to develop what we need
  - instead, we have to make do
“Can’t we use the flow from the last project?”

“Nope, we just hacked something together to get the chip out.”

“You should just use our recommended flow.”
Principle #2: Iteration

*Iteration* is an overloaded term
We have two competing definitions:

**iteration** $n$.

- hand-fixing a single netlist or placement or routed block in order to meet your design goals (e.g., timing, DRC)

  These are *micro iterations*

  Goal: single iteration convergence using automated flow

- running your automated flow when changes occur to RTL, floorplan, constraints, etc.

  These are *macro iterations*

Reality: many many runs
Physical implementation...

- is *not* like building a bridge
  because you only build a bridge once

Physical implementation...

- is like building a *machine* that builds a bridge

Imagine building a new bridge *every day*!

Our goal is to build a *flow* that builds a chip (RTL $\rightarrow$ GDSII)
Principle #2: Iteration

- Iterate early, iterate often
- Iterations will overlap
- How fast can you iterate?
- Each iteration must be
  - repeatable
  - reliable
  - robust

How can we accomplish this?
“The RTL freeze was three weeks ago.”

“What about the five changes since then?”

“Those weren’t releases, they were ECOs.”
Principle #3: Automation

Automation means your computer does the work

- Scripted, not GUI
  - No GUI implies no hand edits
- Batch, not interactive
- Self-checking, not user judgment
- Repeatable, reliable, robust
- Predictable results

Can I sleep while this is running?
“Success in physical design will come from relentless automation.”

Paul Rodman
DAC 2002
“We’ve got scripts to automate your ECO flow. The documentation is on our website.”

“146 pages of documentation. How long will it take us to learn this?”

“I don’t need an ECO flow. Get me a six-pack of Red Bull and I’ll have it ready in the morning.”
Principle #4: Failure

_Failure_ is an overloaded term

We have two competing definitions:

failure _n._

- when a particular flow (sequence of tool invocations) does not generate tapeout-quality results

  Reality: Most iterations _fail_ for one reason or another

  Reality: Many many more failures than tapeouts

failure _n._

- when a particular flow (all the scripts and infrastructure) is painfully awkward and difficult to use

  Reality: Many flows _fail_ although we rarely admit it
Principle #4: Failure (of scripts and infrastructure)

Fallacy: Flows never fail

- You always get the chip out the door, so by definition your flow “works”
- The only flows that fail are the ones that are developed *without* a chip

Who develops flows without a chip project?
Principle #4: Failure (of scripts and infrastructure)

Dangerous fallacies:
- Our flow is a success
- The CAD department’s flow is a failure
- The EDA vendor’s flow is a failure

Outrageous truths:
- Failure of your flow infrastructure is bad failure
- Acknowledge what is broken
- Consider what else might work
Principle #4: Failure (of sequence of tool invocations)

- This is *good* failure
- Failure is inherent in the process
- Failure is your friend
Principle #4: Failure

Success

- How to define success?
  - Must have something to measure
  - Zero errors (DRC, timing, etc.)
  - Success metrics will change as the design matures

- We should not expect *first* pass success
  - rather, we want *last* pass success
  - eventually, *every* pass success
“Look at the cool wrapper we built around the tool. Now it’s much easier for you to use.”

“I got a segmentation fault. What do I do now?”

“Can you provide a test case? Without your wrapper?”
Principle #4: Failure

- All tools suck
Goals

Building from the 4 principles:

- What are the *goals* for our flow?
- How should it look and work?
- Goals are used to drive decisions during flow development and ongoing maintenance
“Our goal is to show you the best way to use our tools.”

“Our goal is to build flows, not chips.”

“Our goal is to meet the original schedule.”

“Our goal is to build chips, not flows.”
# The Goals of Flow Development

#1 Design independent flow

#2 Tool independent flow

#3 Technology independent flow

#4 User independent flow
Principle: iteration
Goal: user independence

- implies isolated sandbox (aka workspace)
- all inputs, libs, etc.
- no external sources
- no common files (except: EDA tools, OS)
- no rug to get pulled out
- isolation for each iteration (multiple sandboxes)
**Iteration**

- can you exactly repeat what you did before?
  - if you can’t repeat it, then you are not iterating, you are flailing

- if your result is a failure, you want to make sure you don’t do it that way again

- if your result is a success you want to repeat it

- what files went into your successful run? this requires...
Configuration management

- Configuration management means which...
  - RTL?
  - scripts?
  - libraries?
  - EDA tool versions?
- ...means any user should be able to repeat
  - same user rerun
  - different user rerun
  - rerun next year
  - EDA vendor rerun (e.g., test case)
- ...implies revision control
Revision control

- all sorts of these have been used for IC design
  - copy directory structure
  - SCCS
  - RCS
  - CVS
  - Subversion
  - Synchronicity, Clearcase, AccuRev, ICManage, ClioSoft, etc.
Revision control

What files need to be under revision control?

Some that you might not immediately consider:
- libraries
- derived files: netlist, layout
- anything that the next step in the flow might need
- anything that you might modify for an ECO
Revision control needs to handle:

- large binary files
- directories as primary objects
- symbolic links as primary objects
- CVS doesn’t cut it...
Revision Tags

- Configuration management uses *revision tags* or equivalent
- Must uniquely identify each iteration
- Bad names:
  - Latest
  - Bronze, silver, gold
- Good names:
  - Anything that will never run out
  - Integers
  - Dates
    - but be sure and include the year
    - and allow for multiple names per day
Consider the user’s *shell environment*:

- It should *not* affect your flow, because it isn’t under revision control!
- Tool version selection should *not* be determined by your shell search path
Automation + tool independence

- The flow should work with different tools and any number of copies of the tools you have.
- There is one precious resource that you *must* manage in an automated fashion...

Licenses!
License management

- Job should *never* fail due to lack of license
- Implies job queueing system (LSF, SGE)
  - even if you have only one license!
  - even if you have only one execution host!
  - even if you have only one user!
- Your job control system *must* be license-aware
- Each command within a tool
  may require a different set of licenses
- Far easier to be efficient with licenses
  if the tool invocations are *short* and *focused*
Define: target

target \( n. \)
  An invocation of a tool

modular target \( n. \)
  An invocation of a tool
to perform a single focused task
Why modular targets are a good idea

- simple to specify/understand/write/reuse
- more efficient license use (check in/out)
- more efficient host use
  - different steps have different resource requirements (e.g., memory, threading)
- enables parallelism: execute targets in parallel
- faster results (critical target first)
- ease of modification
- self-checking targets can detect a problem earlier in the flow
- noncritical target can fail and rest can complete
Example modular targets

- dc_read_sdc
  1. read ddc database
  2. read SDC
  3. write ddc database

- dc_compile
  1. read ddc database
  2. compile
  3. write ddc database

- dc_write_verilog
  1. read ddc database
  2. write out netlist

- pt_analysis_using_sdc
  1. read netlist
  2. read parasitics (optional)
  3. read SDC
  4. update timing
  5. save session

- talus_fix_time
  1. read constrained volcano
  2. fix time
  3. write optimized volcano

- talus_fix_cell
  1. read floorplan volcano
  2. fix cell
  3. write placed volcano
Sequence management

- GNU make
  - Understands and manages dependencies
  - Supports parallel execution
  - Free, and available on all the tool platforms
flowfile: a make-based target sequence

dc_load_design_i1:
  $$\$(start-target)
  $$\$(MAKE) --no-print-directory \\n  \MAKEFILES=../../flow/design_compiler/design_compiler_master.mk \n  TARGET_DIR=${TARGET_DIR} \n  TARGET_INSTANCE=${TARGET_INSTANCE} \n  DC_TOPO="1" \\
  \(start-target) \\
  dc_load_design
  $$\$(end-target)

dc_load_constraints_i1: dc_load_design_i1
  $$\$(start-target)
  $$\$(MAKE) --no-print-directory \\n  \MAKEFILES=../../flow/design_compiler/design_compiler_master.mk \n  TARGET_DIR=${TARGET_DIR} \n  TARGET_INSTANCE=${TARGET_INSTANCE} \n  \(start-target) \\
  input_ddc="$(DERIVED_DIR)/dc_load_design_i1/$(BLOCK_NAME).ddc" \n  constraints_file="flowfiles/cpu_constraints.tcl" \\
  DC_TOPO="0" \\
  dc_load_constraints
  $$\$(end-target)

dc_compile_i1: dc_load_constraints_i1
  $$\$(start-target)
  $$\$(MAKE) --no-print-directory \\n  \MAKEFILES=../../flow/design_compiler/design_compiler_master.mk \n  TARGET_DIR=${TARGET_DIR} \n  TARGET_INSTANCE=${TARGET_INSTANCE} \n  \(start-target) \\
  input_ddc="$(DERIVED_DIR)/dc_compile_i1/$(BLOCK_NAME).ddc" \\
  \MAKEFILES=../../flow/design_compiler/design_compiler_master.mk \n  TARGET_DIR=${TARGET_DIR} \n  TARGET_INSTANCE=${TARGET_INSTANCE} \n  \(start-target) \\
  DC_TOPO="0" \\
  dc_compile
  $$\$(end-target)

dc_syn_reports_i2: dc_compile_i1
  $$\$(start-target)
  $$\$(MAKE) --no-print-directory \\n  \MAKEFILES=../../flow/design_compiler/design_compiler_master.mk \n  TARGET_DIR=${TARGET_DIR} \n  TARGET_INSTANCE=${TARGET_INSTANCE} \n  \(start-target) \\
  input_ddc="" \\
  DC_TOPO="0" \\
  dc_syn_reports
  $$\$(end-target)

dc_compile_incremental_i1: dc_syn_reports_i2
  $$\$(start-target)
  $$\$(MAKE) --no-print-directory \\n  \MAKEFILES=../../flow/design_compiler/design_compiler_master.mk \n  TARGET_DIR=${TARGET_DIR} \n  TARGET_INSTANCE=${TARGET_INSTANCE} \n  \(start-target) \\
  input_ddc="$(DERIVED_DIR)/dc_compile_incremental_i1/$(BLOCK_NAME).ddc" \\
  \MAKEFILES=../../flow/design_compiler/design_compiler_master.mk \n  TARGET_DIR=${TARGET_DIR} \n  TARGET_INSTANCE=${TARGET_INSTANCE} \n  \(start-target) \\
  DC_TOPO="0" \\
  dc_compile_incremental
  $$\$(end-target)

dc_verilog_out_i1: dc_compile_incremental_i1
  $$\$(start-target)
  $$\$(MAKE) --no-print-directory \\n  \MAKEFILES=../../flow/design_compiler/design_compiler_master.mk \n  TARGET_DIR=${TARGET_DIR} \n  TARGET_INSTANCE=${TARGET_INSTANCE} \n  \(start-target) \\
  \MAKEFILES=../../flow/design_compiler/design_compiler_master.mk \n  TARGET_DIR=${TARGET_DIR} \n  TARGET_INSTANCE=${TARGET_INSTANCE} \n  \(start-target) \\
  DC_TOPO="0" \\
  dc_verilog_out
  $$\$(end-target)

post_compiler_reports: dc_compile_incremental_i1
  $$\$(start-target)
  $$\$(MAKE) --no-print-directory \\n  \MAKEFILES=../../flow/design_compiler/design_compiler_master.mk \n  TARGET_DIR=${TARGET_DIR} \n  TARGET_INSTANCE=${TARGET_INSTANCE} \n  \(start-target) \\
  \MAKEFILES=../../flow/design_compiler/design_compiler_master.mk \n  TARGET_DIR=${TARGET_DIR} \n  TARGET_INSTANCE=${TARGET_INSTANCE} \n  \(start-target) \\
  DC_TOPO="0" \\
  dc_syn_reports
  $$\$(end-target)

complete: parallel_2
How would you create and maintain these sequences?

- A *flowfile* is a makefile that describes your flow (a sequence of tool invocations, i.e., targets)

- flowMaker
  - Tcl/Tk script written to build and maintain *flowfiles*
  - Only thing we couldn’t get off the shelf
  - Runs on all tool platforms (e.g., RHEL)
  - Download a copy at www.flowMaker.info
flowMaker window
Select a master target
Activate the master target
Added new target instance
New target instance
Target menu
Rename the target instance
Target instance renamed
Option Selector

FlowMaker

Select the desired target switches below:

- File Inputs
- Target Variables
- Job Submission Settings
- File Outputs

- **dc_load_design_i**
  - DDC_FILE: Design database in DDC format

- **dc_load_constraints_i**
  - input_ddc: "dc_load_design_i":DDC_FILE
  - constraints_file: "newfile/cpu Constraints.tcl"

- **dc_compile_i**
  - input_ddc: "dc_load_constraints_i":DDC_FILE
  - DDC_FILE: Compiled design database in DDC format

- **dc_verilog_out_i**
  - input_ddc: "dc_compile_i":DDC_FILE
  - NETLIST_FILE: Output design netlist in Verilog format
  - DDC_FILE: Output design database in DDC format

- **post_compile_reports**
  - input_ddc: 
  - DDC_FILE: Output design database in DDC format
Derived file forwarding
Job Submission Settings

Select the desired target switches below:

- File Inputs
- Target Variables
- Job Submission Settings
- File Outputs

DC_TOPO
DC_FILE

DDC_FILE: Compiled design database in DDC format.

dc_compile

input_ddc: dc_compile/1 DDC_FILE

TOOL_VERSION
TOOL_RESOURCES
DC_TOPO

dc_verilog_out

input_dcc: dc_compile/1 DDC_FILE

TOOL_VERSION
TOOL_RESOURCES
DC_TOPO
NETLIST_FILE: Output design netlist in Verilog format.
DDC_FILE: Output design database in DDC format.

post_compile_reports

input_dcc: dc_compile/1 DDC_FILE

TOOL_VERSION
TOOL_RESOURCES
DC_TOPO
Change tool version for a target
Target menu
Parallel targets
Add another target
Rearrange target order
Update target options
Add duplicate target
How to run a flow

% make FLOW=dc complete

% make FLOW=dc complete -j -k >& make.log &
Anatomy of a flowMaker master target

dc_syn_reports:
## Basic post-synthesis Design Compiler reports
#
# in input_ddc
#  input_ddc(type) = ddc_file
#  input_ddc(comment) = Input design database in DDC format
#
# job DC_TOPO
#  DC_TOPO(type) = boolean
#  DC_TOPO(value) = 0
#  DC_TOPO(comment) = Enable DC topographical mode
#
$(call dispatch, dc_shell \ 
  $(if $(findstring $(DC_TOPO),1),-topographical) \ 
  -f $(FLOW_ROOT)/design_compiler/dc_syn_reports.tcl \ 
  ,$(TARGET_DIR)/$(TARGET_INSTANCE).log)
Anatomy of a flowMaker master target

talus_fix_time:
## Reads in volcano, runs fix time
#
# in input_volcano
# input_volcano(type) = volcano
# input_volcano(comment) = Input design database in volcano format
# var target_slack
# target_slack(type) = string
# target_slack(value) = 0p
# target_slack(comment) = Target slack with time unit
# var timing_effort
# timing_effort(type) = menu
# timing_effort(value) = medium
# timing_effort(allowedvalues) = low medium high
# timing_effort(comment) = timing effort option to fix time
# out VOLCANO
# VOLCANO(type) = volcano
# VOLCANO(filename) = $(BLOCK)%fix-time.volcano
# VOLCANO(comment) = Output design database in volcano format
#
$(call dispatch, \
    talus -f $(FLOW_ROOT)/magma/$@.tcl -logdir $(TARGET_DIR) \ 
    , $(TARGET_DIR)/$(TARGET_INSTANCE).log )
Master target options

Select the desired target switches below:

- File Inputs
- Target Variables
- Job Submission Settings
- File Outputs

**input_volcano**
- talus_fix_rtl_i1:VOLCANO
  - volcano

**VOLCANO**
- Fix netlist volcano <volcano>

**input_volcano**
- talus_fix_netlist_i1:VOLCANO
  - volcano

**VOLCANO**
- Output design database in volcano format <volcano>

**input_volcano**
- talus_dft_scan_insert_i1:VOLCANO
  - volcano

**3DC**
- write_sdc_i1:3DC
  - sdc_file

**VOLCANO**
- Constrained design database in volcano format

**input_volcano**
- talus_import_constraints_i1:VOLCANO
  - volcano

**target_slack**
- 200p (string)

**timing_effort**
- medium (menu)

**VOLCANO**
- Output design database in volcano format <volcano>
Different flowMaker users

- Just runs an existing flowfile. Invokes “make” from the command line.
- Uses flowMaker to edit the target options, then runs the modified flowfile.
- Uses flowMaker to edit the target options, edit the target sequence (add, delete, rearrange targets), then runs the modified flowfile.
- Writes new master targets to support new EDA tools and new features.
Things to consider

- What sort of directory structure do you have?
- What sort of revision control?
- What sort of configuration management and tags?
- How do you manage job control?
- How do you manage licenses?
- How do you manage tool versions?
- How do you manage your project?
Notes on project management

What you do *not* need: MS Project

- IC design is an iterative process, particularly if your flow is not stable
  - Flows will never be stable when used on an immature design
  - PERT charts are linear tools
  - OK to have major milestones but minor milestones are too detailed

- Don’t be afraid of another iteration
  - with automation, rerun is easy
A suggested management method

- Make a list of everything you haven’t done yet for your design (past successes don’t count)
- Make a second list of everything not automated in your flow yet
- Schedule each item from both lists to occur concurrent with maturation of the RTL
- How long will the implementation take?
  - You mean to run through the tools the final time, or to get all the items in the list achieved?
  - You won’t know the run time until both lists are empty
  - Hopefully both lists will be empty before the RTL stops changing
- Increase frequency of RTL releases as project matures
  - You need time between releases early on to work on the two lists
  - Towards the end, you need to iterate to make sure you are capable of running it all through
Managing iterations

- Iterate early, iterate often
- The last change in the RTL will be small
- A big change in the RTL won’t be the last change
- How do you tell if you are done?
- How do you measure progress?
- Consider your process from RTL ➔ GDSII:
  - Do you know the theoretical minimum time?
  - Do you know exactly how many steps are required?
### Simple progress report

#### min_delay

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>SLOW</td>
<td>2328</td>
<td>2471</td>
<td>1703</td>
<td>2146</td>
<td>1318</td>
<td>1376</td>
</tr>
<tr>
<td>TYP</td>
<td>5412</td>
<td>5788</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FAST</td>
<td>4933</td>
<td>4923</td>
<td>81732</td>
<td>71500</td>
<td>62504</td>
<td>60014</td>
</tr>
<tr>
<td>DFT.scan_setup</td>
<td>212</td>
<td>209</td>
<td>209</td>
<td>1573</td>
<td>1602</td>
<td>1646</td>
</tr>
<tr>
<td>SLOW</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TYP</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FAST</td>
<td>538</td>
<td>535</td>
<td>535</td>
<td>1592</td>
<td>1666</td>
<td>1671</td>
</tr>
<tr>
<td>DFT.scan_shift</td>
<td>14</td>
<td>14</td>
<td>27</td>
<td>15</td>
<td>15</td>
<td>15</td>
</tr>
<tr>
<td>SLOW</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TYP</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FAST</td>
<td>54</td>
<td>44</td>
<td>49</td>
<td>21</td>
<td>42</td>
<td>43</td>
</tr>
<tr>
<td>DFT.stuck-at_scan_capture</td>
<td>203</td>
<td>152</td>
<td>171</td>
<td>433</td>
<td>125</td>
<td>178</td>
</tr>
<tr>
<td>SLOW</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TYP</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FAST</td>
<td>751</td>
<td>517</td>
<td>933</td>
<td>292</td>
<td>349</td>
<td></td>
</tr>
<tr>
<td>DFT.AC_scan_capture</td>
<td>405</td>
<td>354</td>
<td>1730</td>
<td>1423</td>
<td>1476</td>
<td></td>
</tr>
<tr>
<td>SLOW</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TYP</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FAST</td>
<td>1214</td>
<td>992</td>
<td>2219</td>
<td>1552</td>
<td>1609</td>
<td></td>
</tr>
<tr>
<td>DFT.boundary_scan</td>
<td>44</td>
<td>51</td>
<td>1607</td>
<td>1651</td>
<td>1696</td>
<td></td>
</tr>
<tr>
<td>SLOW</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TYP</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FAST</td>
<td>370</td>
<td>431</td>
<td>1843</td>
<td>1921</td>
<td>1926</td>
<td></td>
</tr>
<tr>
<td>DFT.dft_p1500</td>
<td>200</td>
<td>3868</td>
<td>7711</td>
<td>7711</td>
<td>7711</td>
<td></td>
</tr>
<tr>
<td>SLOW</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TYP</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FAST</td>
<td>526</td>
<td>3868</td>
<td>7711</td>
<td>7711</td>
<td>7711</td>
<td></td>
</tr>
<tr>
<td>DFT.sms_p1500</td>
<td>200</td>
<td>659</td>
<td>6785</td>
<td>7067</td>
<td>7184</td>
<td></td>
</tr>
<tr>
<td>SLOW</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TYP</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FAST</td>
<td>526</td>
<td>904</td>
<td>7224</td>
<td>7280</td>
<td>7281</td>
<td></td>
</tr>
<tr>
<td>total</td>
<td>12518</td>
<td>19992</td>
<td>84426</td>
<td>115043</td>
<td>129411</td>
<td>127999</td>
</tr>
</tbody>
</table>
The Magic 8-Ball as a management tool

“Will there be problems with my new IP?”
“Is two weeks enough time to complete a floorplan?”
“Will we really see new timing problems if we turn on SI?”
“Will this chip go out on time?”
Checklist

Let’s make up a checklist for physical implementation!

- Something like the Reuse Methodology Manual and Verification Methodology Manual

- All the things you need:
  - Software
  - Hardware
  - Other stuff
What you need: Other stuff

- Books: Tcl, perl, make, shell, UNIX, revision control
- Training
- Tool documentation
- Design documentation
What you need: Hardware and Infrastructure

- CPU -- compute farm (how many?)
- memory (how much?)
- disk (1Mbyte per placeable instance?)
- backups, printers, power, maintenance...
- desktop: usually PC so you can have office apps (email, Word)
- VNC or other connection to CPU
  how easily can you transfer files to/from desktop to CPU?
  how easily can you cut-and-paste?
- remote access: VPN
- bug tracking
- team communication (email, conferences, video links)
What you need: Software

- configuration management: revision control
- license management: job control
- flow management: flowMaker
- dependence management: make
- tool version management: job control
- compute farm management: job control
- error management: logscan + self-checking
- physical design tools
- libraries (standard cell, IO, memory, analog, 3rd party IP)
"I still don’t see why we can’t use the flow from the last chip."

"How about a flow from the EDA vendor? Can’t we start with what they offer?"

"Didn’t the guys in Bangalore just tape out? We can take their flow, right?"
Thanks

- These companies loaned software keys to build up the example flow:
  - Magma
  - Synopsys
  - Cadence
  - CLK Design Automation

- Designs and libraries provided by:
  - Sun Microsystems
  - Artisan/ARM
  - Nangate

- Special thanks to Narendra Shenoy, Leon Stok and David Reda
Flow engineering: Things to remember

- **Principles**
  - #1 Flows are hard
  - #2 Iterate
  - #3 Automate
  - #4 Failure is your friend

- **Goals**
  - #1 Design independent flow
  - #2 Tool independent flow
  - #3 Technology independent flow
  - #4 User independent flow