Computer Aided Design Tools

Designing a (working) SoC involves numerous steps.

Each stage may employ multiple software tools.

Often many of the tools are 'integrated' in a 'framework' (such as Cadence)

- hopefully working together
- similar 'look and feel'

Designs go through several phases of development:

- Design space exploration and modelling
- RTL design
- Logic synthesis
- Layout/Place and Route
- Electrical checking

Our labs. have concentrated on the first two of these and primarily on the RTL design.

Design space exploration

The first stage of a design is deciding what to build! Presumably there is a (perceived) need for the device so this sets the initial requirements. With a 'simple' device the way forward may be 'obvious'; however modern ASICs are sufficiently expensive that few are that simple. Typically some design modelling is needed.

Transaction Level Modelling (TLM)

A high-level model can be built in anything the designer(s) choose although there are some specialist languages aimed at this purpose. This stage will decompose a potential design into large-ish blocks which communicate via **transactions**.

A transaction could be (say) the sending of a request to **read** a memory at some **address**; another transaction may be returned which is the desired **data**. This could show that a subsequent read command had to wait for the first data to be returned, or could be extended to allow multiple outstanding transactions, possible returned out-of-order etc.

At this level the design space can be explored and experiments with different architectures conducted.

Probably some of the blocks will be bought-in. For example, if a processor is desired there may be something suitable in the public domain or one might approach a company such as ARM whose business is based on **licensing Intellectual Property** (IP) and would supply a range of products from a TLM through RTL design to compilers and debug facilities.

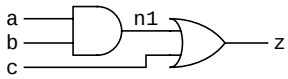
There are some specialist languages which are intended to facilitate TLM. One fairly recent one is **System-Verilog** (standardised 2009) which extends the older Verilog specifications to include features such as object-oriented programming, interfaces and multi-dimensional arrays. Another possibility is **System-C** which is a C++ extension to allow easy parallel modelling. There also exist newer, specialised tools such as **Bluespec** which is intended to allow synthesis directly from a high-level model, automating the RTL translation phase.

[The right hand column on this page applies to the next slide.]

A circuit description may form a netlist which can be simulated.
This could include HDL descriptions:

```
if (a < 4'h6) x = 8'h00; else x = 8'h0F;
```

or schematics made from 'primitive' library elements



which could be extracted and translated to HDL descriptions:

```
assign n1 = a & b;
assign z = n1 | c;
```

Hierarchy may be represented by instantiating elements multiple times.

Limitation: the hardware is going to be static: there can be no '`new`' elements created at run time because the silicon is what emerges from the foundry!

Given a Verilog description of a whole system this can be executed by various means, much like any other computer language.

Interpretation

An interpreter treats the source code as a data structure which is read to guide the interpreter in what to do at any given time. This is a slow mechanism but is quite simple to implement and allows easy 'single stepping', etc.

Compilation

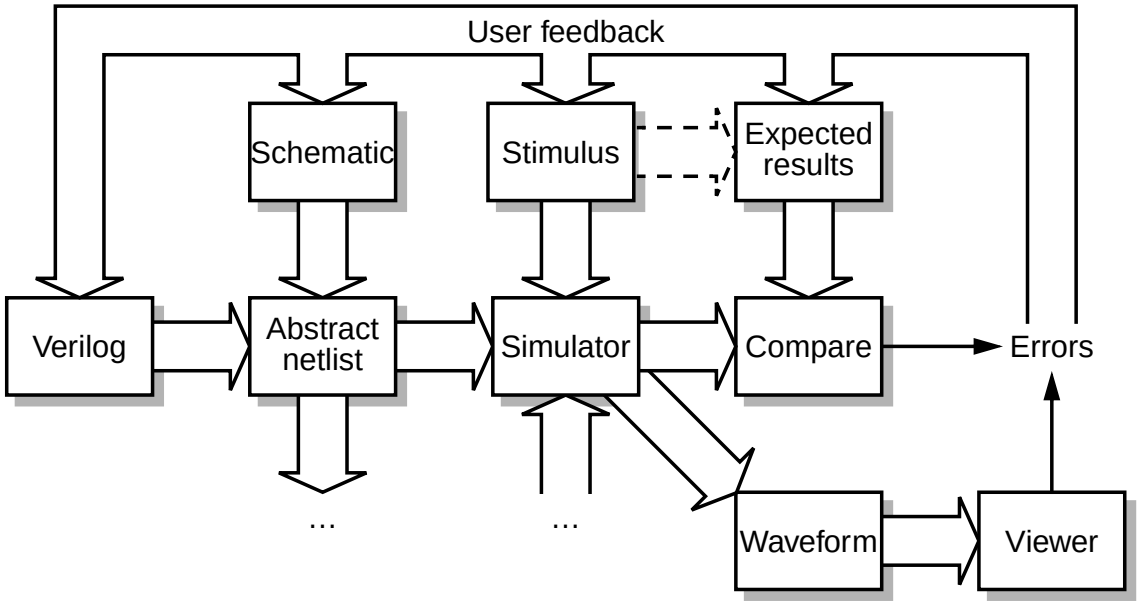
The HDL code may be translated into machine code for the computer which will simulate it (typically x86). This is more complicated than interpretation but the simulations will then run much faster. Tools will often still allow debug facilities such as breakpoints and single stepping by inserting appropriate information (e.g. statement identifiers) in the compiled code.

Scheduling

Regardless of the mode of execution HDL code is **multi-threaded**. This reflects hardware structures which are implicitly **parallel**. The scheduling of the various threads is the responsibility of the run-time system which will execute statements in *an* order which obeys the language specification. Note that the specific order of execution is not always defined so different tools may execute code differently. *Code must be written so that any permissible scheduling differences do not affect its overall operation.*

Scheduling is 'two dimensional' in that it sequences both things that happen at different times – here earlier things happen before later ones – and things which happen simultaneously in time but must be executed sequentially given a sequential simulation machine. The latter *may* be subject to different ordering.

Design flow: Behavioural



The behavioural design models the logic of the system. This can be:

- a high level description (possibly timed)
- an RTL design (cycle accurate)

Behavioural design development

The behavioural (or 'functional') design level should be familiar by now. A design is created using a **HDL description** or a **schematic entry** tool. These are combined to form a **netlist** which could be (for example) still in the form of behavioural Verilog.

This should be simulated according to some input **stimulus** file. The stimulus may be hand-generated; alternatively it can be written from a higher level model. A HDL stimulus generator can also be used, for example to produce pseudo-random inputs for Monte Carlo simulation.

The circuit and stimulus will be combined in a **simulation** by interpreting or, more likely these days, compiling the code and running the result. This can generate a **trace** file of selected signals which indicate how and when they change. It is normal to pre-select an interesting subset of the signals for tracing simply to limit the size of the generated files. Writing large data sets will also slow down the simulation as it will choke on the disc/network bandwidth.

The resultant traces can be imported into a waveform viewer which will offer services such as zoom, scroll and search. The ability to observe the evolution of signals is very useful in debugging.

As simulations become large it is impractical to view everything by ~~hand~~ eye. Data sets can be reduced by recording particularly interesting events – for example printing out the details of each memory read.

The most efficient and reliable way to check results is to automate the process. One method is to keep a record of each output and apply a simple utility such as `diff`. However a simulation environment can also be used for checking, so the **test harness** can compare results as they are generated, picking up expected outputs from a prepared file or, perhaps better, from the generator functions. Having tests which automate the checking and report possible errors (such as are used by lab. demonstrators) makes **regression testing** easy.

When errors are found the fault could be in any user input: the design, the tests or other inputs. It is necessary to review and correct these as appropriate. However once one working design is obtained the tests should only be extended so that compatibility is maintained.

It is a Good Idea to (temporarily) induce some deliberate faults in the design during this development process too. This demonstrates that the test will correctly spot when a design is wrong. A test which has never failed may not *be able* to fail, which is a fault in itself.

This process should be generally familiar from the laboratories.

Transfer formats

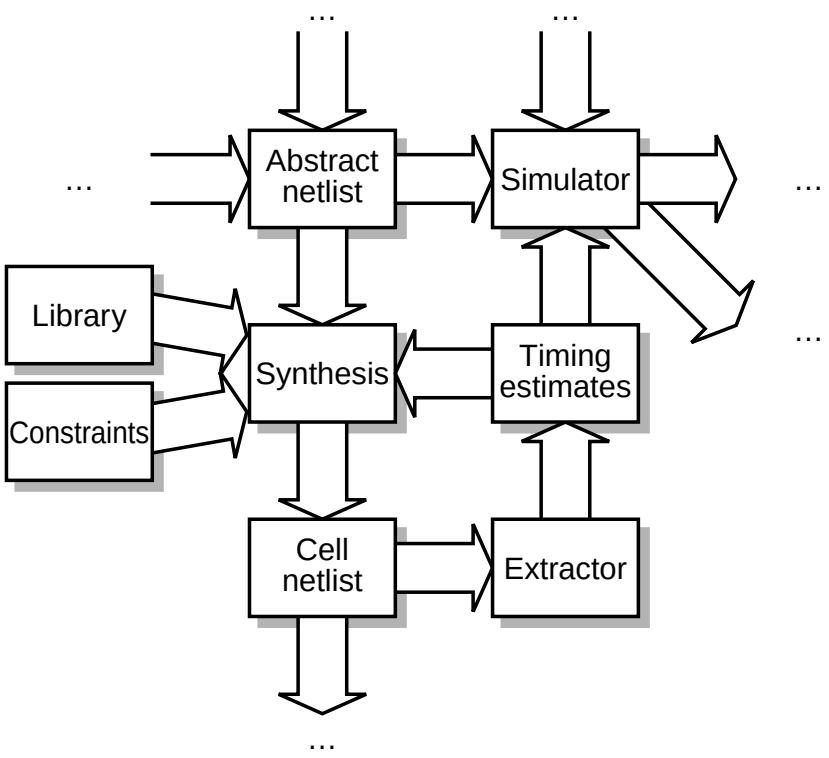
The way designs (etc.) are transferred from tool to tool requires some degree of standardisation if different software is going to be integrated. A few such standards – and, no doubt, some proprietary formats – exist. The details of these don't usually matter as they are both written and read by machines not humans although they are of obvious interest to anyone developing CAD systems themselves.

Verilog is one such system. By using structural Verilog hierarchical models can be imported into descriptions. Thus a schematic diagram can have its basic models (AND and OR gates etc.) instantiated and connected in a text description. This is easy to port to other programmes.

EDIF (Electronic Design Interchange Format) is still used extensively as an vendor independent format. As the name suggests it was developed for this purpose (latterly with considerable involvement from this School).

[This is not intended as an exclusive list.]

Design flow: Synthesis



Synthesis

Given a netlist which is believed satisfactory, this must be mapped into the target technology. This will be supplied from a library, such as a list of standard cells. The synthesizer follows rules to build (for example) an adder specified by a Verilog '+' sign into a (multi-bit) adder made of gates.

Not all Verilog functions can be handled by all synthesizers. Thus it may be that '*' is not understood and it is necessary to decompose a multiplier into simpler stages. (There are many ways to build a multiplier with different area/latency/throughput characteristics.) As it happens the Xilinx synthesizer can map multiplications as most of their FPGAs have dedicated multiplier blocks on-chip.

The synthesis process is also influenced by a set of constraints. Here the constraints will chiefly be area speed requirements. There are different ways to build (e.g.) an adder which cannot all be explored here. However: a ripple-carry adder is simple and area efficient but very slow, having a critical path across every bit in the data bus (time O(N)); adding a carry look-ahead circuit can accelerate the addition (roughly, time $O(\log_2(N))$) thus gaining maybe a factor 6 speed up on a 32-bit add but at the cost of perhaps 2-3 times the logic.

There are many similar examples.

The result of the synthesis will be another netlist where all the functions are represented (at the lowest level) by library cells. This may retain some hierarchical structure or may be *flattened*, where all the cells are instantiated. A flattened netlist gives greater representational flexibility. However there are two significant costs: the expense of having to handle much larger data files and the loss of hierarchical information if a human wishes to view later simulations in detail.

The library cell models will contain some capacitive load information and this, perhaps combined with some wiring load estimates from the fanout, can be extracted and fed back to yield more accurate timing information. For the first time in the process some idea of the possible cycle time is obtained.

The cycle time, obtained through STA (q.v.) can be fed back to the synthesizer to allow it to iterate in an attempt to satisfy the user constraints. It can also be back annotated into a higher level simulation to give more accurate (yet still fairly fast) behavioural simulations.

The netlist of cells can also be simulated, of course.

Library cells

For an ASIC the library cells will be specific to a particular (geometric) process from a particular silicon foundry. Technology mapping therefore commits the design to a certain target. Of course there is nothing to prevent synthesis of the RTL into different targets using different processes. This is usually the business model of 'fabless' silicon companies (such as ARM) who can sell IP which can be made according to your favourite process. Resynthesis is similar to recompiling software for a different processor architecture and makes porting easy – or at least easier than porting layout.

The library will probably not contain layout; instead there will be cell **phantoms** which show the size and connection points. This protects the foundry's IP. The foundry will have **characterised** all the cells using SPICE (q.v.) or something similar so there will also be accurate models of their behaviour for simulation purposes.

Constraints

When changing a **functional** design into a **physical** one there may be a ‘wish list’.

These are expressed as constraints on the synthesizer and place and route tools.

- Clock period
- Floorplan/pinout
- Area/density
- Power

These are often antagonistic so an acceptable trade-off needs to be sought.

Leads to a lot of ‘mucking about’ with parameters to see how/if the requirements can be met.

Constraints

Clock period

Clock constraints were discussed in detail the timing lecture. Fundamentally for almost all logic blocks there will be a clock period into which the gate (and wire) delays must fit.

The synthesis/P&R will apply Static Timing Analysis to its ‘proposed’ implementation and may then:

- adjust gate sizes to increase drive
- add buffers
- give up

The user is free to adjust the functional model – for example by altering the partitioning between stages – as a result of this process.

It is possible to allow logic to take multiple clock periods but these *multi-cycle paths* will need to be specified by the designer. Thus, for example, a slow memory could be allowed two cycles with the address being inserted on one cycle, its output being ignored on the next (thus set-up/hold violations would not be considered) and the output latch would enabled for the cycle after that.

Floorplan/pinout

When routeing a block it may be necessary that certain connections are in/near certain places. For example, at the top level of a chip the connection pads might have been fixed (e.g. so that PCB designers can begin work) and the appropriate signals will need to reach them. This clearly ought to influence where cells are placed.

At a smaller scale, similar constraints may be applied to sub-blocks where particular connections may need to be grouped on particular sides of the block. (These will then further constrain the next level of wiring, hopefully in a useful way.)

Area/density

It is desirable to use all the chip area for useful gates. Looked at in another way, it is expensive to build a bigger chip than necessary.

Having said that, if when the design is ‘placed’ everything is packed tightly there will be no room to enlarge cells or insert buffers to meet timing constraints. Extra space needs to be available close to existing wiring runs otherwise further wire must be added, increasing loads and exacerbating the problem. Thus attempting initial placement at 100% density will usually result in failure.

A typical starting point would be around 70% **utilisation** although the user may choose to experiment to find a point where the implementation is *just* achieved successfully.

The density achievable will, of course be affected by the timing and floorplanning constraints; a longer period implies less need for buffer insertion and densities may be higher. Naturally the user’s wants are antagonistic here!

Power

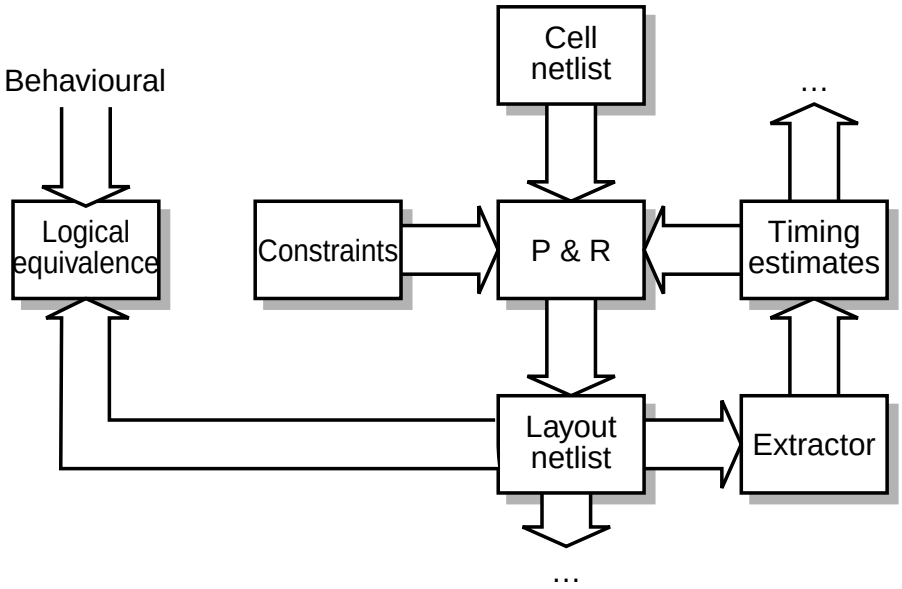
Power dissipation is an issue and there may be a peak power constraint for a chip or block. Power concerns are typically from self-heating (possibly locally, possibly the whole chip) and battery life.

The **energy** used by a CMOS transition is dominated by the supply voltage and the node capacitance (which depends on the wiring). **Power is energy/time** so power is also proportional to how often the node switches. This, in turn, depends on the clock period and the *activity* of a particular node, as most signals do not change on every clock.

A statistical activity file can be derived by running ‘representative’ simulations and producing statistical models for each node. These can then be fed forward as a constraint for the power analyser.

Excessive power dissipation may enforce lower placement density, or some other design change.

Design flow: Layout



- Layout editor
- Extraction
- Floorplanning
- Place & Route
- Buffer Insertion
- Logical equivalence checking

Layout Tools

Layout editor

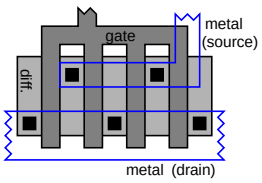
A layout editor is for 'polygon pushing'; drawing the features which will actually appear as wires, transistor-parts etc. It most cases this is now restricted to the creation of standard cells and analogue designs. It is also possible to use such an editor (or similar) for placing and connecting cells a hierarchical units.

Layout editors give a plan-view of the chip surface with the various layers depicted by different colours/shading patterns. (There may be 10+ different layers so it is normal to turn off the display of some to avoid clutter.)

Layout editing will be closely coupled with DRC (q.v.) as the resultant layout must be verified DRC-correct. When the layout is believed complete it can be *extracted*.

Extractor

A tool which interprets layout polygons as components. It will identify transistors and follow wires to find their interconnection, resulting in a schematic-like representation of what is actually present. This is more complex than it sounds in that (for example) some transistors may be physically split into multiple parallel components. (The adjacent figure shows the layout of four parallel transistors which form a single entity.)



The extractor will also capture the sizes of the components and the various load capacitances, etc.

Floorplanning

Floorplanning is similar to layout but is used to indicate the areas where sets of components may be placed rather than placing individual devices. Thus an area may be reserved for a particular (e.g.) processor and all its cells will be placed in that area.

Place and Route

'P&R' are separate operations but generally treated together. **Place**ment is the location of components (typically standard cells) on the silicon surface; when that is decided the **route**r will interconnect the cells with wires.

Placement density may be important. It is bad to waste space but if cells are packed too closely together there will be no room for in-situ changes such as buffer insertion. It is obviously undesirable to throw away a nearly correct layout because of lack of space as a rip-up-and-redo will introduce more problems and it may be that the layout will never 'converge' to something useable if this is done repeatedly. Unused space will be filled with empty 'filler' cells to maintain the standard cell structure.

The resultant layout will need extracting to find the wiring load which has now been imposed as this is topology dependent.

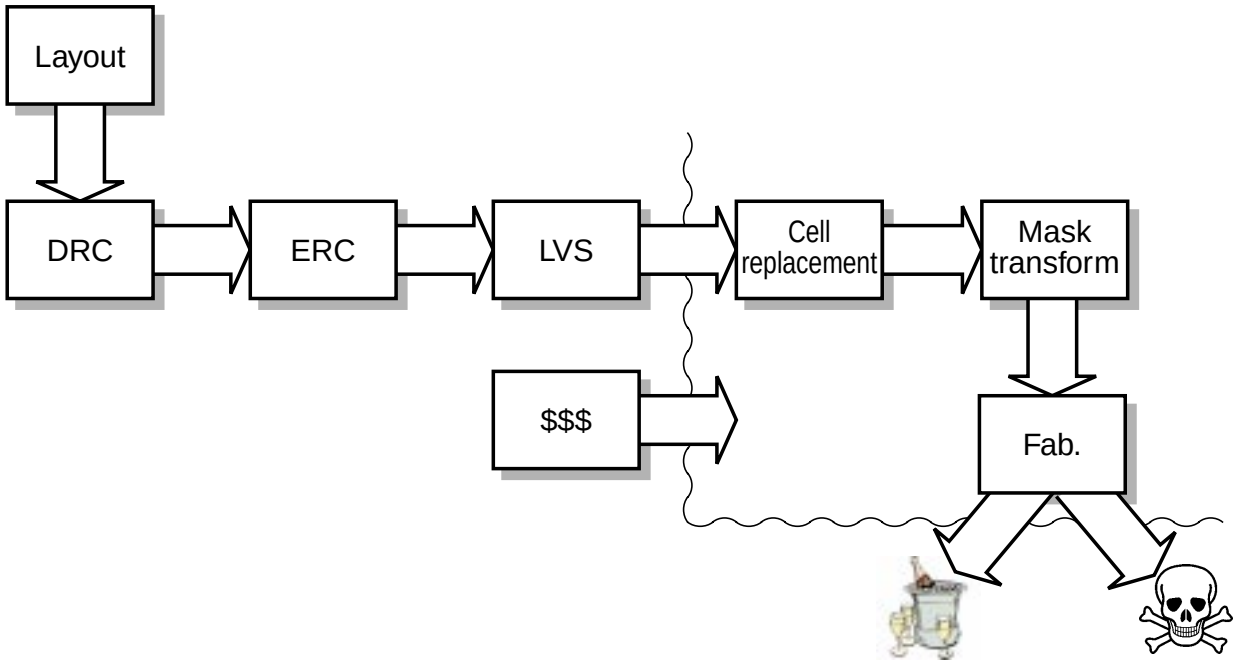
Buffer Insertion

The wiring from the router will slow down switching edges, sometimes unacceptably. To alleviate this the driving gates may be substituted by cells with a greater drive strength (these may be physically larger too, resulting in some placement disruption). If this cannot be done successfully the signal may be buffered by adding extra (amplifier) cells along its path. hopefully these can be inserted close to the desired wiring track; if not the extra wire might neutralise their benefit.

Logical equivalence checking

After all this 'messing about' it is nice to know that the circuit implements the functions you originally designed. Of course it should, but engineers like a 'belt and braces' approach. An equivalence checker is a *separate tool* which compares the synthesized netlist to the original RTL and highlights possible differences. By (hopefully) using different algorithms any obscure mistakes are likely to be different in the different flows and therefore may appear here.

Design flow: Final checks



There may be other checks too …

Foundry Processes

What the foundry does is part mystery but they definitely:

- Substitute any ‘phantoms’ with real layout
- DRC the result
- Process the ‘drawn’ polygons for **mask making**.
- Take the money

Because the feature sizes (e.g. 45 nm) are smaller than the wavelength of the light (currently 193 nm which is fairly hard UV) the masks are not simple photographic prints any more but need to take into account the diffraction patterns formed, so the ‘physics’ of the printing process is first reversed.

FPGA design flow

An FPGA design flow is similar to an ASIC. However in laboratories we keep things relatively simple by doing designs which don’t challenge the tools.

In particular our designs are constrained only by the pads on the FPGA, which are already fixed to certain functions on the Printed Circuit Board.

Our designs (should) fit comfortably onto the available chips. (If this were not the case then FPGAs come in families and larger ones may be available.) Clearly the maximum resource on a particular FPGA is fixed by what you already bought.

FPGA logic is performed by look-up tables in Combinatorial Logic Blocks (CLBs) and these have a fixed drive strength. It is not possible to increase this but it is possible to use CLBs as buffers in high fan-out or long distance signals.

Timing may be constrained but in our labs. we run quite slow[1] designs so the constraints are omitted. The synthesis process will report back estimates of the speed of each unit (and, thus, the ‘chip’ as a whole).

Power is very difficult to estimate because of the need for a routed design and activity models. However by being sufficiently conservative it can be neglected for our purposes.

1. Relative to what *could* be available.

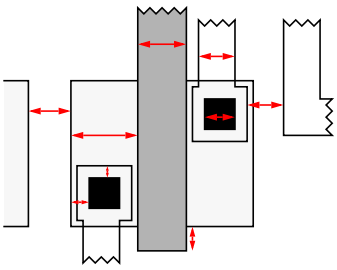
Post-layout Tools

- DRC – Design Rule Check
  - Do the ‘polygons’ all obey the necessary max/min widths and clearances
  - *Should* be okay if standard cells + placement + routers (hierarchical) all worked
  - May need *metal fill* on some tracking layers to keep chip surface planar
- ERC – Electrical Rule Check
  - Various checks on power supplies, voltage drop …
  - Signal Integrity
- LVS – Layout Versus Schematic
  - Does the resultant layout match the intended netlist?
    If process was automated it *should* …
  - If hand-intervention has taken place then this gives added confidence
- Thermal modelling ?
- EMC modelling ??
- …

Post-layout Tools

Design Rule Check

Potentially quite straightforward, the DRC verifies that the device can be made (with a reasonable confidence that it will work).



Some of the parameters which will be checked are shown in the figure, e.g. contact size, gate length (min.), gate/diff. overlap, diff. width, metal track width (min.), track spacing …

There will be other values (such as *max*. wire width) which will also be imposed. Metal dimensions will vary on each layer: the higher layers (numbered upwards from ‘metal 1’ near the silicon) will be wider because it is expected that the surface will be rougher due to the tracking underneath.

Electrical rules

When gates switch they demand charge from the power supplies so a current flows. To put this into perspective, the core of a desktop computer may use a 1 V supply on the chip and yet dissipate 50 W. This means an *average* current demand of 50 A. An electric kettle may dissipate (say) 2.3 kW from a 230 V supply which means it draws 10 A: look at the thickness of its flex.

When supplying gates on-chip the power rails are normally thicker than the minimum width and ‘gridded’ such that there are many routes from pins to cells. Nevertheless they have a finite resistance and so will ‘lose’ some of the supply voltage in transit. This loss must be kept small to avoid violating the gate operating assumptions.

It is now normal to apply a tool which estimates the current demand in each area of the chip and, from this, works out the supply voltage at each point. If the resultant drops are too large the power grid must be reinforced, perhaps by dedicating more metal layers to it.

Signal Integrity checks

Electromigration

Perhaps it sounds a bit odd but electrical currents are like water currents in that they can erode their channels. This phenomenon, called ‘electromigration’, is caused by the flowing electrons ‘impacting’ metal atoms and causing them to move slightly. It is more serious with high current density (more flow) and particularly in power supplies, where the flow is always in the same direction. (In signal wires the charge/discharge cycles tend to even out.) Current is concentrated in the thinner parts of a wire (the wire will be uneven due to manufacturing imperfections) which means any thin parts wear faster, eventually resulting in a broken wire.

Solid-state circuits are very reliable but this is a major wear-out mechanism. To prolong life current densities should be kept as low as is possible. Where power wiring is concerned ‘as much as possible’ is a good plan.

Crosstalk

Tools may also be used to hunt for potential crosstalk (q.v.) problems. Crosstalk is principally a problem when long wires (such as buses) run adjacent for long distances. With notice of this it is possible to space out or swap wire placements to reduce any harmful effect.

Tolerances

The chip must work with a range of **PVT**

- Manufacturing **P**rocess variation
- Varying operating **V**oltage
  - Typical voltage ranges may be ±10%.
- Different chip **T**emperature
  - Temperature ranges may be 0˚C-100˚C, -55˚C-125˚C etc.

You have characterised a chip under some ‘typical’ conditions…

… but does it run (and meet spec.) under variation?

Typically, run simulation at extremes

- fast-fast, high voltage, low temperature
- slow-slow, low voltage, high temperature
- The usual problems occur at the slow end of the range
- Any hold-time violations are likely to occur at the fast end of the range
- Device should operate correctly across whole range

[Sometimes tests will qualify a max. speed – e.g. processors]

Tolerances

Process

The chip manufacturing process is subject to numerous variations. Sometimes the transistor doping is ‘better’ or ‘worse’ which causes the ‘strength’ of the transistors to vary; these variations can affect P and N transistors differently. This means that rising and falling edges may be faster or slower than the design might indicate. Variations differ according to process but as a rough guide the ‘speed’ of a type of transistors may vary by (of the order of) 25%(i.e. the propagation delay in pulling up/down may 25% faster than ‘expected’ or 25% slower – and possibly a mixture).

Simulation (at some level) should test all the process ‘corners’ which are described as {fast-fast, fast-slow, slow-fast, slow-slow}; there is also a ‘typical’ ‘corner’ which is the average and is used for the majority of modelling.

Voltage

When operating the supply voltage may be uncertain (e.g. battery condition) and may vary across a chip according to current demand. The supply voltage will affect the speed, a lower voltage means slower switching. (An upper limit on voltage is imposed by charge tunnelling through the transistor gate insulator; put simply, the parts are so small that a ‘spark’ can jump through the glass[1] (Oxide) spacer.)

Temperature

Clearly operating temperature can vary from ambient (at switch on) to something somewhat higher due to self-heating. The temperature which needs to be considered is that of the transistors, which will be rather higher than that of the (much larger) package.

Higher temperatures reduce MOSFET channel conductance thus ‘slowing’ the circuit.

1. This typically isn’t Silicon Dioxide any more, now being a ‘high-k’ material. These can be made thicker yet still provide the necessary electric field coupling to operate the transistor.

Just for interest …

Dynamic Voltage Scaling (DVS)

Supply voltage affects circuit speed; an approximation is to say that speed is proportional to supply voltage (over the operating range).

Supply voltage also affects the energy dissipated in an operation; this is roughly proportional to the square of the supply voltage.

It is therefore possible to save energy by slowing down the clock and commensurately reducing the supply voltage – but the latency of the calculation increases.

This technique – known as dynamic voltage scaling – is sometimes employed in portable equipment by saving energy when the processing requirement is low but speeding up as demand increases.

There is a minimum voltage at which a CMOS circuit will operate successfully, as well as a maximum.

DVS is more complicated in larger SoCs because there are more subsystems to power; on chip variable regulators may be possible but add considerably to the complexity of the design.