ECAD IB

8 Lectures.

1. Netlists, schematics and RTL.
2. Logic modelling.
3. Further HDL (Verilog).
4. Back-end logic synthesis.
5. Chip, board and system testing.
6. ASIC Design Flow.
7. Speed, Complexity and Folding.
8. Design Example.

These eight lectures are an introductorary course on Electronic Computer-Aided Design (ECAD). These lectures also provide some of the material needed for the first half of the afternoon practical classes which run all Michaelmas term. In particular, there will be a tutorial in the use of the Verilog hardware description language (HDL) and you will need to be able to program in Verilog for the practical classes. Examination questions will expect a working knowledge of Verilog although marks will not be deducted for syntax errors. There may be no exams in ECAD in 98/99.

There is more information in these notes than will be lectured, but these notes do not form a full substitute to background reading.

Recommended books:

Randy H Katz "Contemporary Logic Design" Pub Benjamin Cummings ISBN 0 8053 2703 7

Frank Scarpion "Digital System Implementation" Prentice Hall.

U Golze "VLSI Chip Design with the Hardware Description Language Verilog" Springer.

V Sagdeo "The Complete Verilog Book" KAP.

JM Lee "Verilog Quickstart" KAP.

G Russell, D J Kinniment, E G Chester and M R McLauchlano "CAD for VLSI"

P Naish, P Bishop "Designing Asics"

S M Rubin "Computer Aids for VLSI Design"

J Mavor, M A Jack, P Denyer "Introduction to MOS LSI design"

Weste and Eshraghian "Principles of CMOS VLSI design - A systems perspective."

Donald E.Thomas and Philip Moorby. 'The Verilog Hardware Description Language' Published by Kluwer Academic Publishers. ISBN 0–7923-9126-8.

E Sternheim, R Singh, R Madhaven and Y Trivedi. 'Digital Design and Synthesis with Verilog HDL' Published by Automata. ISBN 0-9627488-2-X.

Glossary of jargon and acronyms

Here are some acronyms:

ALU	Arithemetic and logic unit.
ASIC	Application specific integrated circuit.
BICMOS	A process with both CMOS and bipolar transistors on one die.
CAD	Computer aided design.
CAE	Computer aided engineering.
CAM	Computer aided manufacture.
CLB	Configurable logic block.
CRC	Cyclic redundancy check (added to end of a data frame).
CMOS	Complementary metal oxide semiconductor.
DRAM	Dynamic random access memory.
DMA	Direct memory access.
DSP	Digital signal processor/ing.
EDO	Extended Data Out for DRAMS.
ECL	Emitter coupled logic.
EMC	Electromagnetic compatibility.
EMI	Electromagnetic ingress or i(m)missions (sic).
FET	Field effect transistor.
FSM	Finite state machine.
FPGA	Field programmable gate array.
GaAs	Gallium Arsenide.
HDL	Hardware description language.
IEEE	Institure of electrical and electronic engineers.
IPR	Intellectual Property Rights.
JTAG	Joint technical advisorary group
LSI	Large scale integration (say about 5000 gates, 10000 transistors).
LCA	Logic cell array.
MCM	Multi-chip module.
MSI	Medium scale integration (i.e. the larger ttl devices).
NRE	Non recurring engineering costs. i.e paid once per design.
OTP	One-time programmable
PAL	Programmable array logic.
PIO	Programmed Input and Output.
PCB	Printed ciruit board.
PLL	Phase-locked loop.
PIN	Personal identification number.
RTL	Register transfer level.
SRAM	Static RAM.
SIMM	Single in-line memory module.
SSI	Small scale integration (i.e. about 4 gates on a chip).
TTL	Transistor-transistor logic.
UART	Universal asynchronous recevier and transmitter.
Verilog	An HDL language in wide use (not an acronym).
VHDL	VHSIC Hardware Description Language
VHSIC	Very High Speed Integrated Circuit
VNL	Verilog netlist.
VCO	Voltage controlled oscillator.
VLSI	Very large scale integration (big chips).
XNF	Xilinx netlist format.

1 Netlists, schematics and RTL.

1.1 Abstraction

Electronic Computer Aided Design (ECAD) uses layers of abstraction. The level of complexity within a given layer of abstraction is always limited such that a human designer can comprehend its operation. The complexities and rules for reliable operation of a particular layer are hidden from the layer above by the abstraction and are encapsulated in *design rules*, which explain how and how not to use components from the layer below.

Design rules cover such aspects as:

- maximum loading of a signal,
- minimum space or maximum mask overlap during fabrication of an IC or PCB to avoid shorts or achieve connecton,
- set up and hold times for flip-flops or other pre-designed blocks.

When the design rules are not broken, the layer below behaves as specified and systems can be built like "Lego".

A better term for a layered system is a hierarchic system.

Hardware is always hierarchial, both in concept and implementation. Figure 1 shows that the only thing that is real in an ECAD system is the masks used to make the PCBs and chips and the programming information for programmable devices. Software for embedded processors is shown coming out the side of the stack, but in practice software development often does not share the same ECAD tools or original specification and much manual work is required to integrate the final product.

Future systems will generate both the hardware and the software from a system specification, with the exact mix of hardware and software (e.g. number of processors to use) being chosen by the ECAD system.

A designer works at one of these levels at one time. An abstraction of a level one lower is used as the basic component of the current level. For instance, an AND gate is typically made of transistors, but in a booby trap in an Egyptian pyramids, it might be made made using the combined weight of some marbles that have rolled down grooves. The transistors in the past were discrete but today they are on integrated circuits. Regardless of the lower layer implementation, the abstract view remains constant but parameterised — e.g. by number of inputs, speed of operation, power consumption, etc..

The abstraction is a macroscopic view describing the function. For the AND gate it would consist of

- description of how to use it, such as a list of the connection terminals or 'ports' that need to be connected,
- specification of function performed (e.g. truth table for an AND gate),
- simulation model (typically modelling the delay to operate from the input changing),
- design rules for interconnecting (e.g. its output must not be connected to another output),
- loading that it places on the signals it is connected to,
- total of resources used (e.g. board/chip area and power).

Some guide figures:

6 PCBs in a system
5 ASICS per PCB
100000 gates per ASIC
4 transistors per gate

That makes 12 million transistors in the system.

10 mask polygons per transistor

Further polygons needed for wiring.

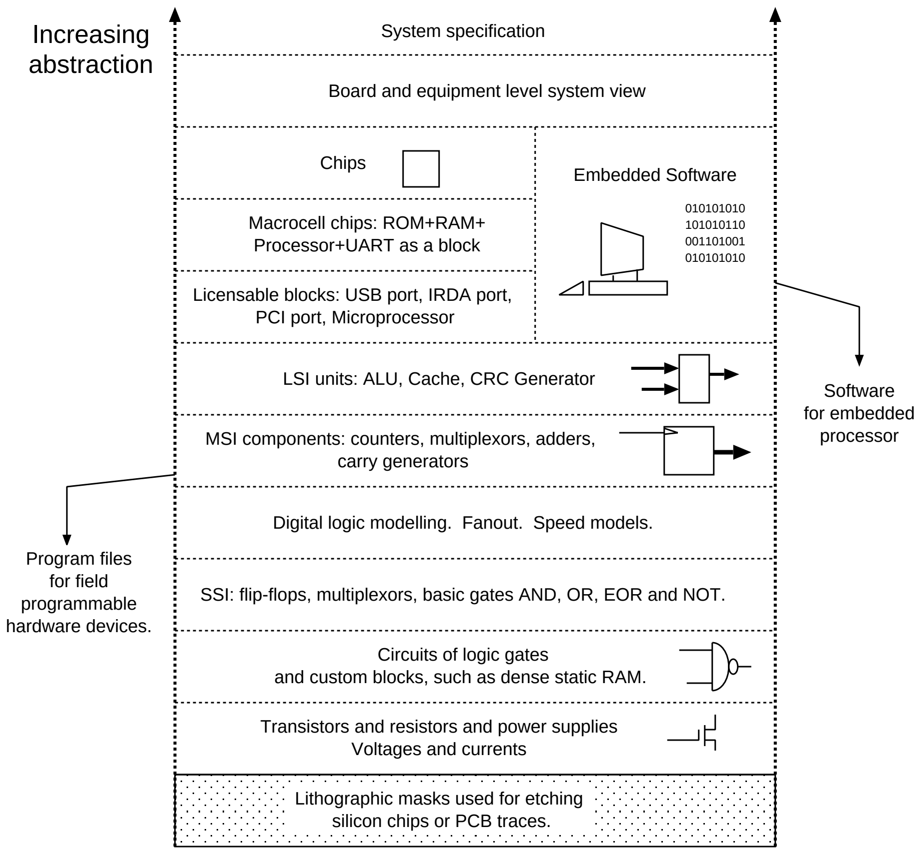


Figure 1: Layers of abstraction built on top of the masks by an ECAD system.

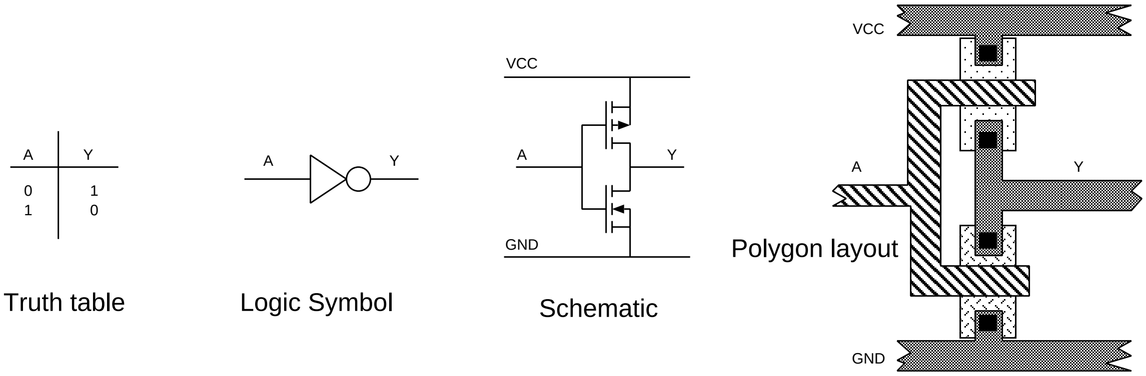


Figure 2: Four levels of abstraction on an inverter.labelfig:inverthier

Such abstractions exist at all levels and enable us to reuse parts of previous designs and give access to suites of standard libraries of abstractions at a common level.

Examples of libraries:

- the 74 series standard logic library (as used in the IA practical classes)
- off-the-shelf chips such as dynamic RAMs
- standard cells in a semi-custom gate array
- sets of boards for standard busses such as PCI or ISA.

1.2 Combinatorial and Register Transfer Level (RTL) design using schematics and Verilog HDL.

A structural hardware description with only one level of heirarchy is called a *netlist*.

1.2.1 Netlist

The netlist is perhaps the most ubiquitous level of circuit representation.

A net is a wire or conductor which connects one or more components. The points where a net joins on to a component are known as the contacts or ports. Every net has a name, its netname. The netlist is basically the circuit diagram of the hardware.

More precisely, the netlist is a list of all of the nets by netname and for each net a list of pairs which tell which components each net goes to and which contact (i.e. pin number) on that component the net goes to.

It can be stored as a list of lines of the following form

```
netname :  [ (part_instance, pin_number), (part_instance, pin_number) ... ]
```

Whenever a part is used the writer of the netlist must generate an instance name for that part instance. These part instance names must be unique. Therefore, when a particular part is used several times in a design, as is the case for simple gates etc, the part instance names will all be different.

Related to the netlist is the parts list, which gives further part type information for each part instance. The details vary from one CAE system to another, but typically have the form:

```
partname : device type, location, manufacturer
```

The location information, in the form of X/Y coordinates for each part, will only be present in the later stages of the design process when the parts have been 'placed'. When the design is complete, a *bill of materials* will be generated from the parts list and fed to the purchasing department.

The manufacturer field is not appropriate where the parts are actually just transistors or gates and the netlist represents a chip. Appropriate fields would be, for example, gate length in microns or the name of the design library that the part will be found in. The library design for a gate would define instances of transistors and their layout and interconnection. The library design for a transistor would consist of a set of polygons spread over the various manufacturing processing stages.

There are various special forms of netlist that can be identified:

- **Top level or closed netlist.** In a closed netlist there are no connections to the outside world. In any simulation of a system we must use a closed netlist, otherwise we would have to bring the effects of the outside world into our model, making it closed.

  To simulate interactions with the outside world and so close an open netlist, we introduce a/some dummy stimulus and response part(s) and build them into the netlist. For instance, if our product is a printer with a parallel port, the parallel port connector has pins on it which would be open in our top level netlist for the whole printer. To close our netlist we must introduce a dummy component which pretends to be a computer (or whatever) and connects to the connector pins.

- **Non-top level, ported netlist.** Alternatively, a netlist can represent a subcircuit of a larger unit, such as just one of the chips of the printer. To the top level, a subcircuit is just like a component; it has ports and a place in the parts instance list. The netlist for a subcircuit has a selection of its nets flagged as being available for connection to the circuit it is a subcircuit of.

- **Flattened netlist.** When the circuit of a design has subcircuits, it is said to be an *heirarchic* design. When it has none, it is said to be *flat*.

  Any hierarchic netlist can be flattened by expanding the instances of its subcircuits. Renaming of instance names and net names in the subcircuit is required in case of clashes. A clash will always occur if one subcircuit is used more than once (i.e. has more than one instance). The conventional way to rename instances is to prefix the original instance name in the circuit of the subcircuit on to all of the instance names in the subcircuit. When multiple levels of hierarchy are used, instance names can get quite long. A small example of flattening is shown in figure 3, where the two levels of heirarchy have turned into one. (The *cvnl* program used in the E+A workshops will flatten a Verilog set of modules if you request the `-vnl` output and give select the top module with `-root`.)

  Selective flattening is when some of the subcircuits of a circuit are flattened (i.e. expanded) and others are left as subcircuit instances.

- **Annotated Netlist.** An Annotated netlist has had additional information stored alongside each net. The most common use for this is when a layout has been performed and the true lengths and loadings of each net are known. The simulator is then able to more accurately match reality (Section 6.10).

Selective flattening is important for *behavioural modelling* (described in section 3.1). Parts of a system which have not yet been designed or synthesised cannot be flattened, but a designer can instead use behavioural models for these while concentrating attention on other parts of the system.

A designer will specify how much flattening and what type of model to use for each instance before a simulation of the design. The full circuit of a design is simply the flattened netlist and would never be fully printed out for anything but small designs.

Heirarchic Netlist

```
module MOD1(b, a);
  input a; output b;
  wire c;
  INV inv1(c, b);
  MODX modx1(b, c);
endmodule



module MOD2(q, s, r);
  input r, s; output q;
  wire c;
  INV inv2(c, s);
  MODY mody1(q, c, r);
endmodule



module MODTOP(r, aa, bb);
  output rr;
  input aa, bb;

  wire l, m;

  MOD1 m(l, aa);
  MOD1 n(m, bb);
  MOD2 o(rr, l, m);
endmodule
```

Equivalent Flattened Netlist

```
module MODTOP (rr, aa, bb);
  input aa, bb; output rr;
  wire l, m;
  wire m_c, n_c, o_c;

  INV m_inv1(m_c, aa);
  INV n_inv1(n_c, bb);
  INV o_inv2(o_c, l);
  MODX m_modx1(m_c, l);
  MODX n_modx1(n_c, m);
  MODY o_mody1(rr, o_c, m);

endmodule
```

For many designs the
flattened netlist
is often bigger than the
hierarchic netlist owing
to multiple instances
of the same component.
Here it was smaller.

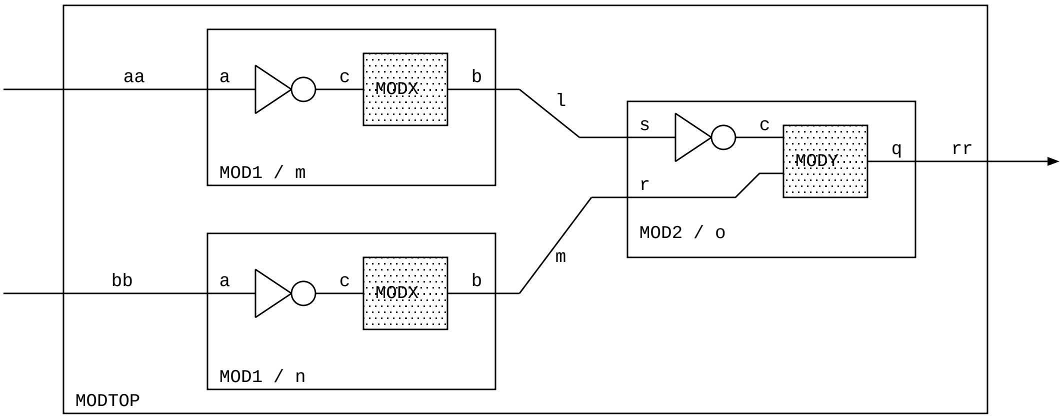


Figure 3: Verilog an schematic of a set of modules and a flattend equivalent module..

Netlists may be:

- entered manually with a text editor
- entered graphically with a schematic drawing package
- synthesised from higher level forms using tools.

A combination of all three generation methods ends up being used in many designs.

There are a number of standard formats for netlists, such as edif and OrCad, Verilog Netlist (vnl) and Xilinx netlist format (XNF).

1.3 Standardised hardware definition languages.

Verilog and VHDL are the most common HDLs today. VHDL is slightly more verbose than Verilog and is richer, allowing user defined datatypes. Each language has its fans. A small subset of Verilog language syntax, known as the structural subset, can be used to specify a hierarchic netlist (here termed vnl) (section 1.4).

Beyond the structural subset, Verilog supports register transfer level (RTL) (section 1.6) and behavioural constructs(section 3.1). We will look at all three levels in this course.

Both VHDL and Verilog contain constructs which can be simulated but which are beyond the capabilities of current compilers to convert to hardware (section 3.4). The subset of the language which can be compiled grows each year, as compilers develop. However, rather than using these languages in advanced ways, there is growing enthusiasm for generating hardware from standard imperative programming languages, such as C and Java (section 3.6).

1.4 Structural Netlists within HDLs

A hardware description language (HDL) offers a text file representation of a circuit. It lists the components along with their interconnecting wires (termed nets). Each wire or net has a name and each instance of a component has a name. For example, the general form of a line in a Cambridge HDL description is:

```
instance_name := component_name(wire1, wire2 .... wireN);
```

In VHDL we find

```
instance_name :  COMPONENT component_name PORT MAP
                 (wire1, wire2 ..... wireN);
```

This is longer, but VHDL offers many unshown options that one might sometimes wish to use. In Verilog it is

```
component instance_name (wire1, wire2 ...... wireN);
```

Here is an HDL version of the divide-by-five counter of Figure 4 in an HDL of some sort:

```
 begin circuitone;
   clock :  CLOCK(clk);
   ff1   :  DFF(clk, a, q1, qb1);
   ff2   :  DFF(clk, q1, q2, qb2);
   ff3   :  DFF(clk, q2, q3, qb3);
   nor   :  NOR2(a, q2, q3);
end circuit;
```

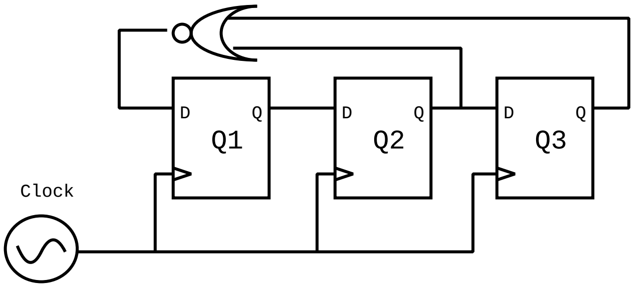


Figure 4: Schematic of ‘circuitone’: a simple divider circuit.

Note that this circuit, called *circuitone* has no (explicit) hierarchy and no inputs or outputs.

The order in which the contacts are given for a particular component must be correct and accord with a definition elsewhere, such as in the library of standard components the component is drawn from. We define part of the library in the first exercise below.

We will now turn this top-level, closed circuit into a subcircuit by giving it some external connections. We assume that the clock is from an external source and we wish to have an output from one of the flip-flops. We will also add a reset input. The connection ports are listed after the name in the begin statement, along with their type, in or out.

```
begin subcircuit(clk, rst, q2);
   input clk, rst;
   output q2;
   ff1   :  DFFR(clk, rst, a, q1, qb1);
   ff2   :  DFFR(clk, rst, q1, q2, qb2);
   ff3   :  DFFR(clk, rst, q2, q3, qb3);
   nor   :  NOR2(a, q2, q3);
end subcircuit;
```

Exercise: Write the HDL for DFFR and NOR2 using just

- INV — an inverter, and
- DFF — the simple d-type used before
- AND2 — a two input and gate.

Any number of these parts can be used in each model.
Exercise: Try to write the HDL code for INV and AND2 using whatever subcircuits you like. Make an apposite statement about ‘leaf’ models.

1.4.1 Port Typing and Interconnection Rules

Some ECAD suites have typed ports and rules about what type combinations may be interconnected by a net. These rules can either be mandatorary or just flag warnings when violated.

In the old Cambridge HDL we find the following port types:

IN an input,

OUT an output,

TRI a tri-state output,

BI a bidirectional port.

In Verilog the same types are defined with the keywords `input, output` and, for the last two, `inout`. Cambridge HDL has two Net types

IO a singly driven net,

TRI a tri-state net.

and in Verilog these net types are introduced with the keywords `wire, tri`.

Cambridge HDL had obvious rules about interconnection: Rules for a net of type IO:

- must connect to exactly one port of type OUT,
- may connect to any number of ports of type IN

Rules for a TRI net:

- may connect to any number of ports of type TRI,
- may connect to any number of ports of type BI,
- may connect to any number of ports of type IN.

Power supplies. Net names such as VCC and GND are sometimes available as OUT types. These may be connected to unused device inputs.

In VHDL, abstract types can be defined, used and checked at the higher levels, with the aim of avoiding mistakes. For instance, signals could be typed (branded) such that the compiler generates an error if an address bus wire is connected to a data bus port. Enumeration types are also useful, allowing symboic names instead of integer constants. These can be defined in VHDL whereas with Verilog a macro preprocssor is used. In the same way as for compilation of HLLs into machine code, these types need have no physical manifestation at the gate level.

1.5 Schematic entry of the netlist

Schematic entry systems allow graphical entry of a netlist. A user pulls components out of a library and plonks them on her 'page', then uses a mouse to place wires between them. Each page can have terminal ports to allow the definition of subcircuits. The componets used may be such subcircuits which have their netlists entered in the same or another way. Schematic entry is for mouse lovers, and is useful sometimes. There will be a demo of schematic entry in a later lecture.

1.6 Register Transfer Level (RTL)

A register transfer level defintion of hardware is often used today. Both Verilog and VHDL include RTL constructs.

RTL is essentially an imperative programming language where the variables are the registers in the hardware. Each register must be separately defined, along with its width. Values are transfered between registers on a single clock edge using an operator, such as the `<=` operator in Verilog. In Abel, the clock net does not need to appear in the source language and instead can be implied. In all RTLs, a rich operator set can be available to form the expressions for the right-hand side of each assignment. An infix multiply operator '`*`' could imply the generation of thousands of gates if the registers are wide.

In RTL, each variable is assigned only once in an unordered list of assignments. Conditional expressions (`? :`) and the `if` construct are synonymous in RTL. This is demonstrated in the Verilog example below by the assignments to `y0` and `y1` which are equivalent.

```
module TEST(p, q, clk);

   input clk;
   input [4:0] p, q;
   input r;

   reg [4:0] a, b, c;

   always @(posedge clk) begin
          b <= (r) ? q : a;
          c <= a-b;
          if (p == 3) a <= a + 1; else a <= q;
          ...
          q <= ...
          ...
          end
endmodule
```

When an `else` clause is missing, it is the same as having an `else` clause present which assigns the variable to itself. Note that in the Verilog example, the `begin end` could be relaced with `fork join` without changing the meaning. This would mean that all the statements are to be executed in parallel.

In a VHDL equivalent, the infinite loop, introduced by `always` is termed a *process loop* and this term can be used in all languages. Systems programmers will see a clear similarity to a multi-threaded software system, the event sensitivity at the head of the process loop typically takes the form a blocking system call to some form of `read`, `select` or `taskwait` primitive that gets the next work item.

There is no pause of flow or program counter in an RTL description. Another important aspect of an RTL description is that the writer knows and intrinsically states in his code how many registers (flip-flops) the design will use and what their functions will be.[1] RTL leaves the tools no freedom over the algorithm used to implement the target system, it just has freedom over the circuit details.

1.7 Finite state machine specification

Many hardware definition languages provide special support for easy definition of finite state machines (Verilog does not). The designer simply provides a list of the states needed, and for each state, gives the output nets which should be active and a list of input condition/next state pairs to define the transition function. The language tool allocates the states to flip-flop configurations and generates all of the necessary combinatorial logic for the next state function.

Again, the number of flip-flops has been implicitly specified by the designer. If 'one hot' encoding is used, then the number of flops is equal to the number of states, otherwise it is lower bounded by log-base-two of the number of states.

Here is an example FSM definition in a made up language with three states and three inputs. It is assumed to start in idle mode and the reset input forces it back to idle mode. The prime input causes it to go from idle to waiting. It will stay in waiting until tigger is true, when it will go to running. After running, whatever the input values (denoted by the star), it will return to idle. The coding is iven as ONEHOT and there are three states, so three flip-flops are implied. If the coding was BINARY, only two flip-flops would be required. For output from the FSM, it is assumed that the names of the states are available for use in RTL expressions outside the FSM definition. The

[1] Although flip-flops with unused outputs or obviously constant outputs will typically be deleted in post optimisations.

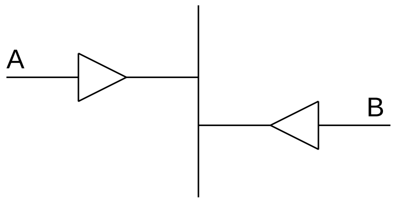
same will be true for the inputs, so either Mealy or Moore machines can be made.

```
DEFFSM
   CLK = clk;
   INPUTS = reset, prime, trigger;
   STATES = { running, waiting, idle };
   CODING = ONEHOT;
   TRANSITIONS =
        idle: prime -> waiting;
        waiting: reset -> idle, trigger -> running;
        running: * -> idle;

ENDFSM
```

1.8 Introduction to ECAD Workshops.

This section will finish with an introduction to the ECAD and architecture workshops and the associated assessed exercises. Finally there will be a demo of the *cv2* compiler.



A \ B	0	1	X	Z
0	0	X	X	0
1	X	1	X	1
X	X	X	X	X
Z	0	1	X	Z

Figure 5: The four value logic resolution function.

2 Logic modelling.

A simulator models the operation of a system. One could model the voltage on each net as a real function of time. Instead of using voltages, a digital simulator uses logic levels. The *four value logic system* is often used.

2.1 Logic modelling using four value system.

In the four value logic system each net (wire or signal), at a particular time, has one of the following logic values:[2]

value	meaning
0	logic zero
1	logic one
Z	high impedance — not driven at the moment
X	uncertain — the simulator does not know

In this model, nets jump from one value to another in an instant, wheras in real life the voltage always changes at some maximal rate (e.g. 2000 volts per microsecond).

When more than one driver is connected to a net (if allowed in the terms of acceptable netlists) then the simulator must invoke a bus resolution function to determine the value of the net. Figure 5 shows a suitable function for the four value system. It is commutative and associative.

Multiple drivers are associative.

Figure 6 shows the four value logic with common gates. Consider the values of (x and 0) and (x or 1) — they are defined sensibly. There is a certain similarity between z and x as input values.

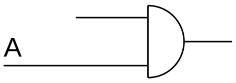
> *Exercise:* Draw truth tables for a NOR, XOR and NAND gates. Can these truth tables be generated by functional composition of the given truth tables ?
> *Exercise:* Draw a bus wire driven with two tri-state buffers, therefore having a total of 4 inputs. *You may find it helpful to condense your truth tables by introducing another value 'XX', meaning we do not know and it does not matter.*

2.2 Systems with more than 4 logic values

Many logic modelling systems include variations in signal strength, to enable modelling, for instance, of the fact that a weak logic one produced by a pullup resistor can safely be overcome by a strong logic zero produced by an open collector output. Full-blooded Verilog systems include this capability, but CSIM only uses the four value logic system. Full Verilog also includes a 'charge strength' for each net, to model its capacitance.

[2] In design specification, the letter 'x' is also used to denote 'dont-care', which allows efficient logic minimisation. In Verilog, the letter 'x' means uncertain during simulation and 'dont-care' during logic synthesis.

B

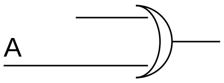


A

AND gate

A \ B	0	1	X	Z
0	0	0	0	0
1	0	1	X	X
X	0	X	X	X
Z	0	X	X	X

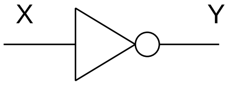
B



A

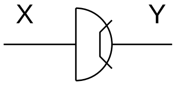
OR gate

A \ B	0	1	X	Z
0	0	1	X	X
1	1	1	1	1
X	X	1	X	X
Z	X	1	X	X

X  Y

Inverter

X	0	1	X	Z
Y	1	0	X	X

X  Y

Open collector buffer

X	0	1	X	Z
Y	0	Z	X	X

Figure 6: Four value logic with common gates.

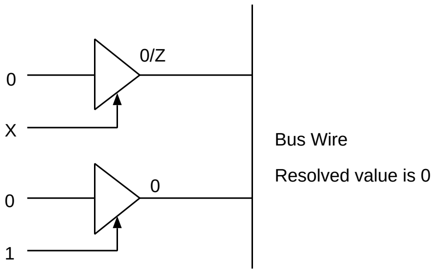


Figure 7: Example where having more than four logic values avoids a final X value.

Most more-powerful logic modelling systems can model a range of values on a single net. For instance, in figure 7, the upper tri-state buffer is producing either a zero or a Z. When this is resolved with the 0 from the lower tri-state buffer, we obtain a 0 overall. In a four value logic system, the output from the upper buffer would have to be represented as an X and this, when resolved with the lower 0, would give an X. Therefore the four value system would cause more X's to flow around the circuit and X's are a pain in practice.

In some modelling systems additional logic values are used to indicate changing nets (e.g. a logic value used for period between a zero and a one and another used between a one and a zero).

The logic values available are often built into the CAD tools, but VHDL allows the user to specify the logic values used and a system of 46 states known as `t_logic` is frequently used with VHDL designs in practice since it is used in most VHDL libraries.

Exercise: Write down the resolution function for a 6 value logic system where the two new states are a weak logic zero and a weak logic one (This is useful for pullup or pulldown resistors used in open collector logic etc). Actually, there maybe more than one sensible resolution function.
Exercise: Consider whether the 6 value logic system can model resistors in series with nets, rather than simply resistors with one end to the power supply.
Exercise: Consider the truth table for an XOR gate under the four value logic system when the two inputs are connected together. How does this reflect on the power of the four value logic system ?
Exercise: Consider the optimisations that can be made to the implementation of an FIR filter if it is known that only every fourth output value is to be used, with the three in between being ignored.

2.3 Event driven simulation.

Event driven simulation is often used in computing science. For ECAD, the events used are changes in the logic value on a net. An event simulator contains the following components:

- System interconnection representation (e.g. netlist with a slot for a logic value on each net),
- Object-oriented implementations of the behaviour of the leaf components (these are called the component *models*),
- An event list,
- A means to put some initial events in the event list, and
- Some methods to view the simulation operation.

At all times the event list is sorted in order of ascending time field, as shown in figure 8. The simulator operates using a single thread as follows. The first event is removed from the event list. The next event becomes the new first event. The removed event is executed and this typically generates more events. When the event list is empty, the simulation has finished.

For logic simulation, the event execution proceeds as follows. The event is a new logic value for a specified net. If the new value is the same as the old value already on the net, nothing happens. Otherwise, the net list is consulted and all leaf component modules which are connected to the net have a method invoked. The choice of method or the arguments passed to it indicate which terminal of the component the event happened on and what the new logic value for the net is. In response to events, the model may update internal state and it may schedule further events for the future. These new events are inserted in the event list at the appropriate place.

As a simple example, take an inverter. It has one input, and when it gets an input event, it generates an output event for the output net scheduled for some time in the future that is the current time (given by the time field in the event being executed) plus the inverter's view of its propagation delay.

Another simple example, which serves as a generator of events, is a clock. The clock must somehow be arranged to receive an initial event at the start of simulation. Thereafter, for each event it receives, the clock will generate an event on its output net of the oposite polarity and then schedule an event for itself in half a clock period's time into the future.

2.4 The Verilog/VHDL simulation cycle including delta time.

This section is not be examinable. It is useful to understand the general principle of event-based simulation. It is also important, as a Verilog user, to understand the difference between Verilog's

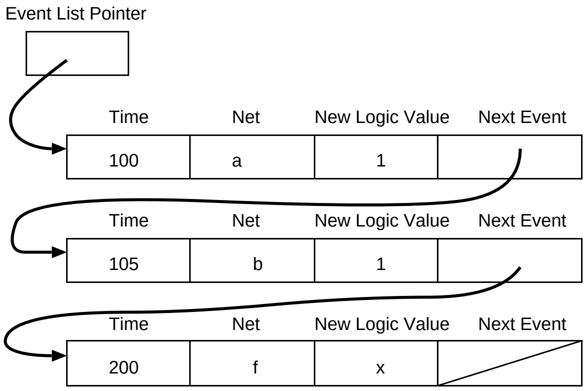


Figure 8: Main data structure for event driven simulation.

two behavioural assignment operators (as documented in the 'Learners Subset'*, but it is not really necessary to know implementation details. Indeed, what is outlined here is just one possible implementation of the Verilog/VHDL approach.*

In a Verilog simulator, the (heirarchic) netlist and the component models are written in Verilog, but the event driven simulator and the output functions are implemented in the lower layers.

The Verilog simulation cycle (and hence language semantics) can be implemented with an event driven simulation, but three enhancements are needed: delta time steps must be added to the event list, a thread package of some sort is needed (there are no stacks for these threads so the thread package is trivial) and a pending update list is needed.

In Verilog, a thread is introduced with the keyword `initial` or the keyword `always`. The latter is a shorthand for '`initial while (1)`' and thus introduces an infinitely looping thread. A thread without event control can run to completion immediately.

```
initial begin a = 1; b = 2; end
```

This will set the variables to the given values and exit.

If we deliberately create a race

```
initial a = 1;
initial a = 2;
```

then the language does not define which order the threads will run in and so a could end up with either value.

There are two forms of event control in Verilog: hash delays and signal sensitivities. Event control statements can be put in front of any other statement to delay the thread. In the hash form, the integer given means delay the thread for that number of system clocks. There is a system clock, called `$time` that is simply an integer and in an event-driven implementation, it can simply be set to the time field from the last event taken of the event list. For an example of hash delays, the following will cause a pair of pulses on the net called `a`.

```
initial begin a = 1; #10 a = 0; #10 a = 1; #10 a = 0; end
```

When a thread encounters a hash delay, the thread blocks and an event can be added to the event list to unblock the thread when the time comes.

A typical signal sensitivity has the form `@(posedge net)` where 'net' is any signal[3] and the thread

[3]The words *signal* and *net* are used interchangably in Verilog.

will be held up until a low to high transition of that net. This is very useful for modelling D-type flip-flop behaviour, but has little revelvance to this current discussion.

Delta time steps arise from the fact that using a hash delay of zero does not mean the obvious thing, which would be nothing. Instead, `# 0` essentially means that the thread should block until everything on the leading portion of the event list where the events share the same time setting as each other (and `$time`) have been executed. It is almost identical to a `#1`, in that all threads across the system will synchronise, except that the time will not advance.

An entry is made on the pending update list by a non-blocking assignment. The list is a simple list of registers and new values for them. The pending update list is critical to the way Verilog evaluates RTL sections, since the registers, which are the left-hand sides of the non-blocking assignments, must not appear to change on the right-hand side expressions until they all have been evaluated.

Non-blocking assignment is made using the '`<=`' symbol whereas blocking assignment is made using '`=`'. Whenever all threads in the whole simulation are blocked, all of the entries on the pending update list are processed by assigning the given values to the given registers. It is an amazing fact that the exact details of its thread execution are not defined in the current Verilog standard.

Here is an example.

```
initial begin
     a = 3;       // a gets 3
     a <= 4;      // a pending update to a is created
     b = 5+a;     // b gets 8, since a has its first value still
     #0           // this hash zero executes the pending updates
     b = 5+a;     // b gets 7
```

VHDL is basically exactly the same, except VHDL has user defined data types and so on.

2.5 General rule for converting Verilog to hardware.

There are many envisagable mappings between a Verilog program and a hardware circuit and, indeed, there are many hardware circuits which perform the same function. However, the generally used mapping used in a Verilog compiler is as follows.

Verilog variables become flip-flops. Each flip-flop is updated by at most one thread. The value stored in the flip-flop is the value last assigned to the Verilog variable when the thread encounters a clock event control statements, such as `(posedge clk)`.

The final details of the circuit used are determined by processes known as back-end synthesis (section 4). The first ECAD+ARCH workshop gives first-hand experience of the translation rule using the example *first.cv*.

2.6 Demo of the *csim* simulator.

We will conclude with a demo of the *csim* simulator and look closely at an example section of Verilog shown in figure 9.

```
// first.cv
// a divide by 16 counter with an exciting output

module CTR16(clk, o);

  input clk;
  output o;

  reg [3:0] count;

  initial count = 0;


  always @(posedge clk) count <= count + 4'd1;


  assign o = count[3] ^ count[1];


  PRINTBUS4 printbus(clk, count);

endmodule

// top level simulation model
module SIMSYS();
  wire clk;
  wire boo;
  CLK10MHz clock(clk);
  CTR16 counter (clk, boo);
endmodule
```

A module starts with the word module and ends with the word endmodule.

Comments start with double slash.

The module ports are listed here without range or direction.

The width spec [3:0] makes it a four bit wide register.

Direction and widths given in the declaration.

A local variable which is to be assigned to inside an initial or always block must be of type reg.

This is a four bit wide decimal one.

This initial statement is run when the simulation starts, otherwise the count variable would be uncertain (4'hx).

This is a behavioural statement with one event control followed by a single assignment.

This is a continuous assignment which produces combinatorial logic. Uparrow is xor and the subscripts access individual bits of the register

These local nets are of type wire since they are driven by instantiated components.

The module SIMSYS is a closed netlist as required to start simulation.

Here is an instance of the CTR16 defined above. The instance name is counter.

Figure 9: A Verilog section including all three forms of specification: structural instances, continuous assignment for combinatorial functions and behavioural assignmnet in the always block.

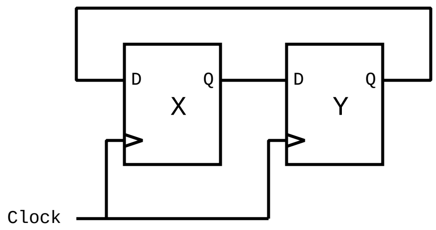


Figure 10: Circuit to swap two registers.

3 Further HDL with emphasis on Verilog.

The Verilog touched on in the first two sections is described in detail in the guide '*Learners Subset*' that is given out with the E+A workshop notes. This is the subset of the language which most people use for practical hardware designs. In this simple subset, we note that

1. each thread is either an initial thread that dies or else an always thread which spends its whole life in one big process loop,
2. the only event control used is at the head of the process loop,
3. each variable is updated in exactly one process loop.

In this section, we will consider further constructs that can be used in Verilog, some that cannot and the shortcomings of Verilog.

3.1 Beyond RTL: Behavioural descriptions of hardware.

With RTL the designer is well aware what will happen on the clock edge and of the parallel nature of all the assignments and is relatively well aware of the circuit she has generated. For instance it is quite clear that this code

```
always @(posedge clk) begin
    x <= y;
    y <= x;
    end
```

will produce the circuit of Figure 10. (If `x` and `y` were busses, the circuit would be repeated for each wire of the bus.) The semantics of the above code are that the right-hand sides are all evaluated and then assigned to the left-hand sides. The order of the statements is unimportant.

However, the same circuit may be generated using a specification where assigment is made using the `=` operator. If we assume there is no other reference to the intermediate register `t` elsewhere, and so a flip-flop named `t` is not required in the output logic. On the other hand, if `t` is used, then its input will be the same as the flip-flop for `y`, so an optimisation step will use the output of `y` instead of having a flip-flop for `t`.

```
always @(posedge clk) begin
    t = x;
    x = y;
    y = t;
    end
```

With this style of specification the order of the statements is significant.

Typically the above style of assignment statements are incorporated in a whole set of nested `if-then-else` and `case` statements. The intention of this style is to give an easy to write and understand description of the desired function, but this can result in logic output from the synthesiser which is mostly incomprehensible. Ideally, the user would never have to look at the output from the synthesiser, but in practice detailed timing and testing anaysis often requires manual inspection and intervention at the backend.

3.2 More advanced behavioural specification

```
reg din;
always
      begin
      @(posedge clk) din = 1;
      @(posedge clk) din = 1;
      @(posedge clk) din = 0;
      end
```

In this example, not only is there a point of execution, but the implied 'program counter' advances only partially around the body of the `always` loop on each clock edge. Clearly the compiler or synthesiser has to make up flip-flops not explicilty mentioned by the designer, to hold the current 'progam counter' value.

3.3 Memories and Superstates.

Many hardware designs call for memories, either RAM and ROM. Small memories can be implemented from gates and flip-flops (if RAM). For larger memories, a customised structure is preferable (their implementation will be covered in the part II VLSI course). Very large memories are best implemented using separate off-chip device where as sizes of several kilobytes can easily be integrated in ASICs. Having several smaller memories on a chip takes more space than having one larger memory because of overheads due mainly to address decoding.

In an imperative HDL, memories readily map to arrays. A primary difference between a formal memory structure and a bunch of gates is the I/O bandwidth: it is not normally possible to access more than one location at a time in a memory. Consider the following Verilog HDL

```
reg [7:0] myram [1023:0];  // 1 kbyte memory

always @(posedge clk) myram[a] = myram[a+1] + 2;
```

If `myram` is implemented as an off-the-shelf, single-ported memory array, then it is not possible to read and write it in one clock cycle. Compilers which handle RAMs in this way either do not have explicit clock statements in the user code, or else interpret them flexibly. An example of flexible interpretation, is the 'Superstate' concept introduced by Synopsys for their compiler, which splits the user specified clock intervals into as many as needed actual clock cycles.[4] With such a compiler, the above example is synthesisable.

When multiple memories are used, a scheduling algorithm must be used by the compiler to determine the best order for reading and writing the required values.

The decisions about how many memories to used and what to keep in them may be automated or manual. In Verilog, directives embedded in comments are often used to guide synthesiser tools (see section 7).

[4]Verilog is bizzare in that subscripts in square brackets applied to arrays act as entry selects where as to non-array, bussed nets, the array acts as a bit select. It is impossible in Verilog syntax to apply a bit select to an array entry without first assigning the array entry to an intermediate variable.

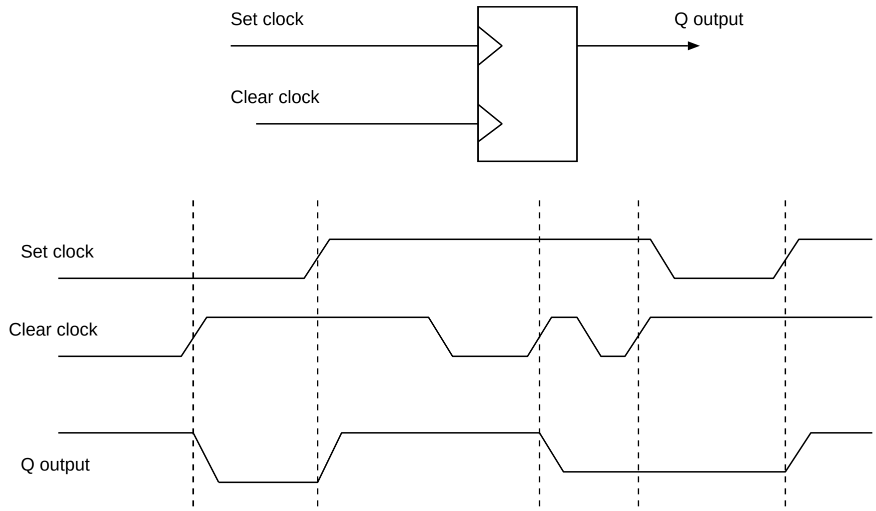


Figure 11: A component where the output depends on the positive edge of two signals.

3.4 Non-synthesisable constructs

Logic synthesisers cannot synthesise into hardware the full set of constructs of a general programming language. There are inevitable problems with:

- recursive functions
- unbounded numbers of state variables
- access to file or screen I/O

Another aspect typical synthesisers have problems with is asynchronous update to variables. For instance, in Verilog

```
reg q;
input set, clear;

always @(posedge set) q = 1;
always @(posedge clear) q = 0;
```

is beyond current general purpose logic synthesisers, yet it defines a well-known and perfectly realisable component, shown in figure 11. With the rules used in the CSYN CV2 compiler, this will result in two flip-flops for y with their outputs connected together. As their D-inputs are constant values, 0 and 1, the flip-flops could be simplifed away, replacing the output nets with the input nets and then the power supplies would be joined. Clearly an error! CV2 would spot the error and stop.

3.5 Shortcomings of Verilog and VHDL.

Verilog and VHDL are languages focussed more on simulation than logic synthesis. The normal rules for translation to hardware (section 2.5) are simple and semantically are more of a bodge than a comfortable fit.

Circuit aspects which could readily be determined or decided by the compiler are frequently explicit or directly implicit in the source Verilog text. These aspects include the number of state variables, the size of registers and the width of busses. Having these details in the source text makes the design longer and less portable.

Perhaps the major shortcoming of Verilog (and VHDL) is that the language gives the designer no help with concurrency. That is, the designer must keep in her head any aspect of handshaking between logic circuts or shared reading of register resources. This is ironic since hardware systems have much greater parallelism than software systems. The language 'Handel-C' from Oxford is designed to be compiled into hardware and does support exchange of data along *'channels'*. The handshaking signals required are generated by the compiler [www.comlab.ox.ac.uk/oucl/users/ian.page].

Finally, in Verilog, it is very hard to use the behavioural style of specification without consuming multiple clock cycles. A judicious selection and intermixing of the two assignment operators available can achieve this, but leads to highly unreadable code and easily generated bugs. In practice, Verilog programmers solve this by separating out the registered and combinatorial sections into separate threads where the combinatorial thread encompasses the complex behavioural code and the registered thread is a trivial list of parallel RTL assignments of the form `actual_Signal <= new_Signal;` for each net to be registered in the final design.

It should also be stated that neither Verilog or VHDL have a useable formal semantics and both are evolving within their definition committees faster than the changes can be documented properly.

3.5.1 Motivations to do better.

Verilog and VHDL have allowed vast ASICs to be designed, so in some sense they are successful. Better languages are needed to meet the following new ECAD aims:

- Speed of design: time to market,
- Integration of final ASIC test procedures,
- Ease of behavioural specification,
- Ease of implementation of a formal specification,
- Ease of proof of conformance to a specification
- Portability: can be compiled into software as well as into hardware.

3.6 Synthesis from C and other programing languages.

The advantages of using a general purpose language to describe both hardware and software are becoming apparent: designs can be ported easily and tested in software environments before implementation in hardware. There is also the potential benefit that software engineers can be used to generate ASICs: they are normally cheaper to employ than ASIC engineers! The practical benefit of auch approaches is not yet proven, but there is great potential.

A product called C-to-Verilog [http://www.compilogic.com] is typical of the new generation of ECAD tools. C-to-Verilog claims the following:

- Compile C to RTL Verilog for synthesis to FPGA and ASIC hardware.
- Compile C to Test-Bench for Verilog simulation.
- Compiler options to control design's size and performance.
- Global analysis optimizes C-program intentions in hardware.
- Automatic and controlled parallelism and pipelining.
- Generates readable Verilog for integration and modification.
- Options to assist tracing/debugging HDL generated.
- Includes command line and GUI programmer's workbench.

3.6.1 Semantics of a programming language when used for hardware description.

In Verilog, the rule for mapping the thread to hardware is simply to update the real flip-flops with the values found in the simulation time registers when the thread encountes the clock event control statement ('`(posedge clk)`'). In languages such as C and Java, there are no such clock statements. There are no widely-accepted rules for converting C and Java to hardware, but two suitable rules for functions and processes can be summarised as:

Combinatorial logic from functions: If a function makes no use of global, free or static variables and the number of times any loops in its body are executed can be determined (easily) at compile time, then we can generate a combinatorial logic block (network of gates) that does the same thing.

Infinite process loops: If the program contains a '`while (1)`' type header to a loop, then this will inevitably have input and output operations in the body of the loop and the whole loop can usefully be converted to a logic block which performs the same function. The number of clock cycles that the logic block consumes to loop the loop can be chosen by the compiler: it may vary on input data. The nature of the input and output statements may vary: calls to print functions are not likely to be convertable to hardware. Instead, inputs and outputs are likely to be reads and writes to static arrays which map to standard RAM blocks in the hardware implementation.

Demo: The second approach will be illustrated with a demo of compiling the C implementation of an ARM processor into gates.

Example exam question/Revision question: a) Why do designers only tend to use the RTL subset of the richer hardware description languages such Verilog? b) What are the advantages of using behavioural specification ? c) What are the advantages of using C or Java for hardware implementation? *Note: there are no ECAD exam questions in 98/99 so revision may not be sensible.*

3.7 State chart graphical 'languages'

There have been attempts to generate hardware systems via graphical entry of a finite state machine or set of machines. The action at a state or an edge is normally the execution of some software typed into a dialog box at that state, so the state machine tends to just show the top levels of the system. An example is the 'Specharts' system [IEEE Design and Test, Dec 92]. The Unified Modeling Language (UML) is promoted as *'the industry-standard language for specifying, visualizing, constructing, and documenting the artifacts of software systems'* [http://www.rational.com/uml/index.shtml]. It can be used for hardware too. Takeup of new tools is slow, especially if they are only likely to prove themselves as worth the effort on large designs, where the risk of using brand new tools cannot normally be afforded.

3.8 Synthesis from formal specification

It is desirable to eliminate the human aspect from structured hardware design and to leave as much as possible to the computer. The idea is that computers do not make mistakes, but there are various ways of looking at that!

A holy grail for CAD system designers is to restrict the human contribution towards a design to the top level entry of a specification of the system in a formal language. By 'formal' we mean a language in which it is easy to prove properties of the system. (The Part II course on hardware specification shows how to use predicate logic to do this.) The detailed design is then synthesised by the system from the specification.

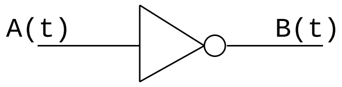
There are many ways of implementing a particular function and the number of ways of implementing a complete system is infinite. Most of these are silly, a few are sensible and one, perhaps,

Synthesis from formal specification.

The notation on this page will be covered in Logic and Proof course.

Designs can be specified using predicate calculus.

Example, an inverter

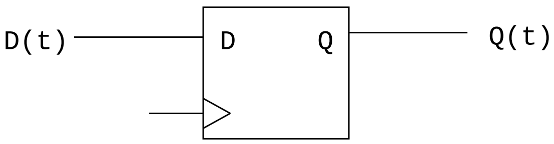


$$\forall\, t.\quad A(t) \Leftrightarrow \sim B(t)$$

Above, the digital logic values are the truth values of the proof system, but they may be separated as follows:

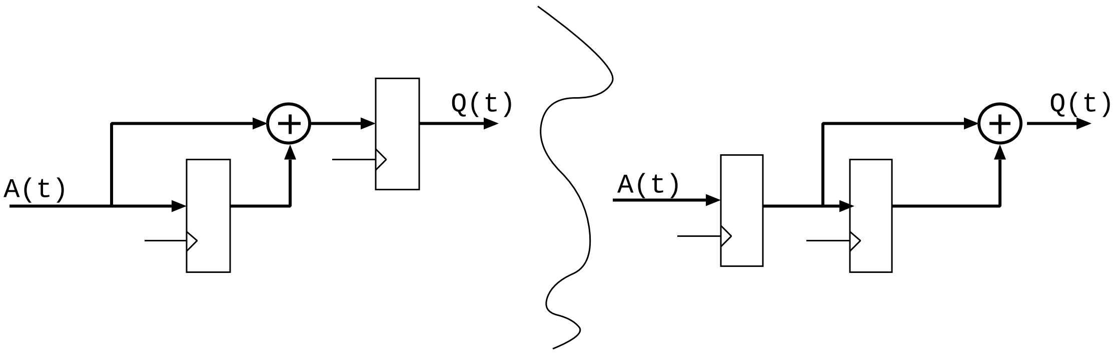
$$\forall\, t.\quad A(t)==1 \Leftrightarrow \quad B(t)==0$$

For synchronous machines with a single global clock, time may be quantised and time steps mapped to clock steps:



$$\forall\, t.\quad D(t)==x \Leftrightarrow \quad Q(t+1)==x$$

Of course, when D and Q are busses, multiple flip-flops are used forming a broadside register.



$$\forall\, t.\quad Q(t)==A(t-1) + A(t-2)$$

Using such logic-based, formal specification, it is easy to specify systems that cannot be made, for instance, systems which are

- Non causal (future input affects current output).

Figure 12: Basics of compilation from formal specification.

optimum. Research into using expert systems to select the best implementation is ongoing, but human input is needed in practical systems. The important thing is that the human input is only a guide to synthesis, chosing a particular way out of many 'formally correct' ways. Therefore errors cannot be introduced.

For instance, an inverter with input A and output B, expressed as predicates of time, can be specified as

$$\forall t. A(t) \quad \leftrightarrow \quad \neg B(t)$$

Here the logic levels of the circuit have the same notation as the logic values in the proof system, but an approach where this is not done is not significantly more complicated:

$$\forall t. A(t) == 1 \quad \leftrightarrow \quad B(t) == 0$$

When time is quantised in units equal to a tick of the global clock then a D-type flip-flop can be expressed:

$$Q(t+1) == x \quad \leftrightarrow \quad D(t) == x$$

Here we have dropped the implied, leading $\forall t$.

A complex formal specification does not necessarily describe the algorithm and hence logic structure that will be used in the implementation. Therefore, synthesis from formal specification involves a measure of inventiveness on the part of the tool.

3.8.1 Refinement from a specification to implementation.

Conversion from specification to implementation can be done with a process known as *selective refinement*. This chips away at bits of the specification until, finally, it has all be converted to logic. Some example rules for the confersion are given in figure 12. There is a vast selection of refinement rules available for application at each refinement step and the quality of the outcome is sensitive to early decisions. Therefore, fully automated techniques for this fail with anything other than trivial example designs. (This last sentence is only the author's conjecture, but he will dig up some hard evidence soon, perhaps).

It seems that a better approach is for much of the design to be specified algorithmicly by the designer (as in the above work) but for the designer to leave gaps where he is confident that a refinement-based tool will fill them. These gaps are often left by designers in their first pass at a design anyway; or else they are filled with some approximate code that will allow the whole design to compile and which is heavily marked with comments to say that it is probably wrong. These critical bits of code are often the hardest to write and easiest to get wrong and are the bits that are most relevant to meeting the design specification. An example might be the handshake logic for a bus or network protocol.

Systems which can synthesise hardware from formal specifications are not in wide commercial use, but there is a good market opportunity there and, in the long run, such systems will probably generate better designs than humans.

A final goal will be to freely mix design specifications in many forms, including functional and formal specifications. The synthesis system should basically only complain under the following situations:

- The requested system is actually impossible: e.g. the output comes before the input that caused it,
- The system is overspecified in a contradictorary way,
- The aglorithm for implementing the desired function cannot be determined.

3.9 Demo of behavioural compilation.

This section will finish with a demo of compiling Verilog into hardware where the compiler generates additional state flip-flops to serve as a program counter.

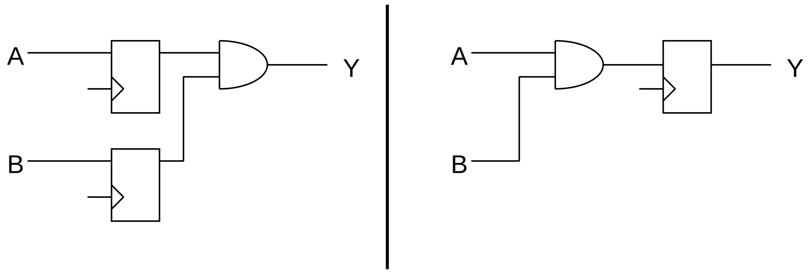


Figure 13: Example circuit before and after D-type migration.

4 Back-end logic synthesis.

The back-end of the logic synthesis procedure involves optimisation. Such optimisation can consume massive amounts of CPU time, although typically a designer will not use more than a day's worth of CPU time on any particular job. Instead, she will partition the synthesis process into day-sized jobs, or her boss will structure the team that way. The back-end synthesis procedures include

- Restructuring procedures where the contents of flip-flops changes.
- Restructuring procedures where the contents of flip-flops is unchanged and only the combinatorial logic changes.
- Insertion of special logic patterns for testing.

4.1 Synthesis goals: logic minimsation, delay area, power.

There are many digital logic circuits which achieve the same function, but they vary in

- the number of gates they consume,
- the total wiring length consumed,
- the speed of various paths through the circuit,
- the power consumption.

Typically minimisation aims to reduce the number of gates, but wiring can be reduced at the expense of gates if a gate is duplicated with identical inputs at the points where its output is needed, provided those inputs are available locally.

Speed optimisation is a valuable technique. Typically, many paths are given '*dont-exceed*' timespecs (timing specifications). The optimiser must select a pattern of gates which meets these timespecs. The paths may be specified as start-end pairs or else, for sequential logic, as '*must-be-stable*' times relative to the clock.

4.1.1 D-type migration.

A simple example of restructuring procedures where the contents of flip-flops change is shown in figure 13. Both circuits compute the same logic function (assuming all clocks commoned) but they have vastly different timing properties. Compared with the right-hand figure, the left-hand figure has a later output but can accept later inputs. This form of restructuring is known as D-type migration or '*re-pipelining*'. Note that, in the figure, the number of D-types has actually changed.

4.1.2 Restructuring to accommodate timespecs.

4.1.3 Logic Minimisation.

If the flip-flops are left alone, then we are faced with the combinatorial logic minimisation problem. Logic minimisation in itself is an extensive subject. The algorithms required to generate efficient circuits in the presence of timespecs are mature and proprietrary, although several textbooks exist.

If we consider 'straightforward' minimisation of total gate count, then there are several classes of algorithm available, depending on the run time available and how many outputs come from the logic block.

Logic minimisation can be solved with exhaustive search for small numbers of input variables (less then ten) or by Karnough maps by humans. A general principle of logic minimisation is the expansion of *cubes* to cover adjacent dont-cares, as done by humans with Karnough maps. A cube is a conjunction (ANDing) of one or more input variables or their negations. The more variables included in a cube the fewer input combinations it is true for, so cubes which expand as much as possible take fewest gates.

The Espresso algorithm [http://oak.ece.ul.ie/ colin/ce4518/espresso.html] from Berkely is widely used iterative algorithm for logic minimisation, based on repeated expansion and contraction of cubes and keeping the expansions which are broadest (minimal gates). It gives a good enough result in most cases.

For multiple outputs, the (PD) Putnam and Davis algorithm can be used, but it is rather simple. This expands all of the output expressions into sum-of-products terms and looks at the resulting terms, which are *implicants*. An implicant is a cube which, if true, means that the output must also be true. There are two simple rules for minimisation with implicants which the PD process uses. They are: two implicants which are different in only the negation of one variable (e.g. ACD and $A\overline{C}D$) can be replaced with a single one without that variable (i.e. AD), and two implicants which are the same as each other except that one has an additional literal can be replaced with just the shorter one (e.g. ABC and AB can be replaced with just AB);

Unfortunately, there are no simple rules for many of the other identities of Boolean logic which allow minimisation, although one could use apply a finite table of known identities to get a bit further. An example problem (from the carry output of a full adder) is $AB + C(\overline{A}B + A\overline{B})$, which cannot be simplified with everyday rules to $AB + AC + CB$.

The PD algorithm works by looking at all of the implicants and selecting the ones to be used (implemented in gates) based on their usefulness. Clearly any implicant which is disjoint from all the others is needed. Implicants which are used by a high number of output expressions are then selected for implementation, since the output from the implicant gate will be used several times at the OR stage. Finally, implicants which are needed but have no special merit are chosen. At each stage, several options may be available, so the whole algorithm can be run iteratively with alternate decisions being forced, and the best result selected.

The Putnam and Davis algorithm tends to generate two-level logic trees, which are often not ideal, since the high fan-in gates needed have actually to be made from multiple smaller gates. Indeed, the initial expressions or HDL which the logic specification came from may contain a degree of useful structure which could best be preserved, but which instead is lost in the conversion to sum-of-products form.

Algorithms which iterate to improve on multi-level logic structures (i.e. as opposed to sum-of-products, two-level structures) are research areas.

4.1.4 Don't care states in HDL.

Verilog allows specification of don't cares as shown in this (ALU-like) example:

```
wire [31:0] my_putput =
        (case1) ? a + b:
        (case2) ? ~a:
        (case3) ? a - b:
        32'bx;
```

Here we are asserting that if none of the case signals is true, then we do not care what the value on the bus 'my_putput' is. These don't care values will become fuel to the logic minimiser. (If you try this with cv2 you will find cv2 has a second-rate logic minimiser.) Advanced Verilog programmers will know how to tell the compiler other facts, such as that 'case1' and 'case2' will never be true at the same time, which will also cause logic simplification to be possible (such information creates further, effective dont-care points).

4.2 Instance optimisation problem and uniquify.

Often the same subcircuit is used several times in a design on a single chip. The synthesiser can either be given the whole chip to do a global job on, or else it can be given the subcircuit once and the synthesiser output will be used in multiple instances.

There are various pros and cons: for instance, if multiple instances of the same module are used, hand optimisations and test procedures will work for each instance; but if the synthesiser is allowed to work on the whole design it may optimise across the previous instance boundaries resulting in a more compact result.

If, in some instances of a subcircuit, some inputs are tied off to fixed logic values, or an output is not used, then that instance will carry redundant logic. The benefit of synthesising it only once, meaning it will only have to be debugged and analysed for testability at the gate level once, may outweigh the cost of the wasted silicon.

The command to the industry standard Synopsys compiler for making multiple instances separate, so that they are treated individually, is 'uniqify'.

4.3 Demo of schematic entry, PCB layout and autorouter.

This section will finish with a demo of a schematic entry package for printed circuit boards, but it is important to realise that schematic entry of ASICs is also used, although generally only for the higher levels of the circuit hierarchcy. The PCB layout and autorouter systems will also be demonstrated. Note that place and route is needed on ASICs in just the same way as on PCBs.

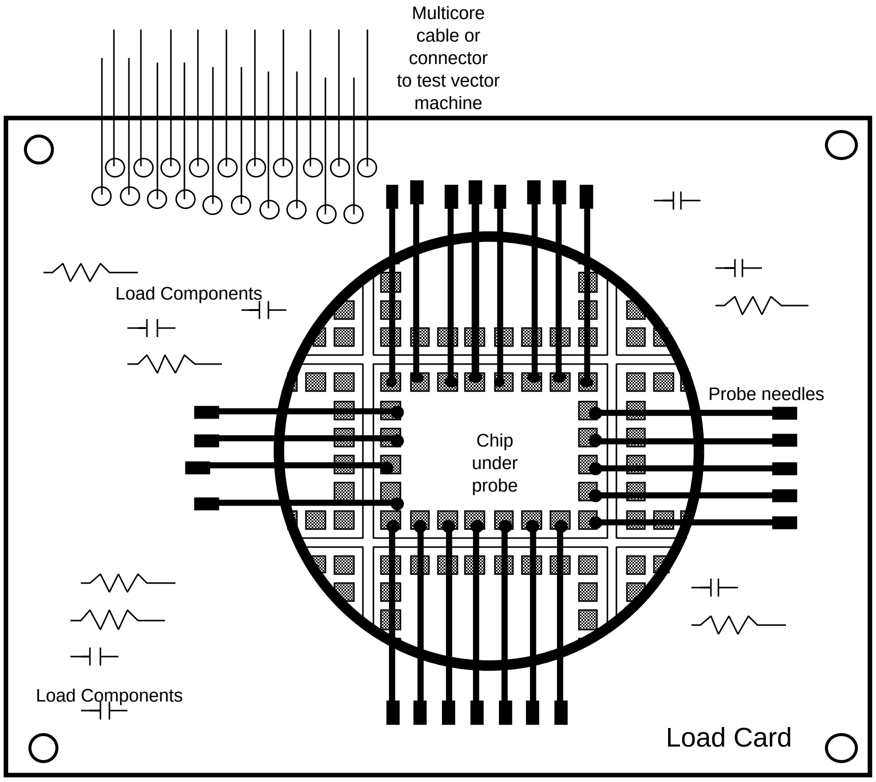


Figure 14: A load card with wafer probe pins for testing a chip before the wafer is diced.

5 Chip, board and system testing.

When a design is complete there will be a production run. For consumer goods, such as a CD player, we might make 5000 units per month for a design life of six months. The aim is that 99.9 percent of units will work.

To ensure the number of failures is acceptable it is necessary for every component and subassembly to be tested before the next stage of assembly, and, in addition, a final test program for the finished product must be used.

If we assume that faults are uniformly distributed over the design and that a reasonable fraction (say half) of the total product functionality is only occasionally used in normal operation, then it is clear that a special test program is required which is able to exercise (nearly) every function of the unit. Since testing time can be a major part of manufacturing time, the test program must run in as short a time as possible.

The fraction of functions covered by a test program is the *fault coverage*, which is normally expressed in percent.

A test program consists of a sequence of stimulus and expected results that must be applied to the device. If the actual results do not match the expected results, then a fault is detected. Of course it is possible for faults to exist which do not show up as a result of a poor test program.

A good test program will achieve 98 to 99.5 fault coverage in a few seconds of testing. Military, aerospace and medical standard equipment may need to have 100 percent coverage.

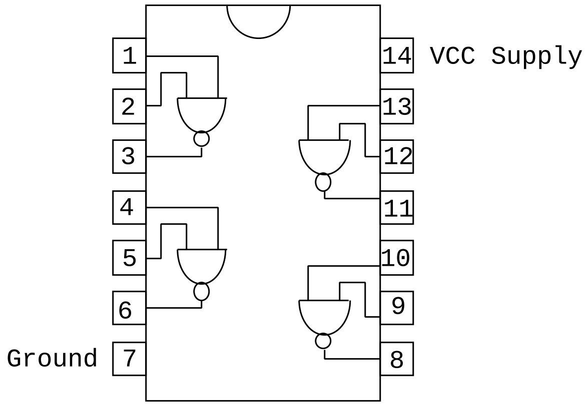


Figure 15: Pin connections of a 7400 quad NAND gate device.

5.1 Stuck-at fault model, fault coverage and fault simulation.

For a digital chip or a circuit board, a tangible subset of ‘every possible type of fault’ must be used. This is the *stuck at* model of faults. In this model, the only fault considered is for one single net of the whole design to be shorted to the power supply or ground. If there are n nets, there are $2n$ faults that could occur.

5.2 Linear Test vector format.

A test program for a chip or card consists of a set of test vectors generated from simulation or otherwise. The test program is applied to a set of test probe points. These are typically the pins or pads of a chip or a *bed of nails* that a circuit board is pushed on to.

The test program consists of a list of vectors. The length of each vector is equal to the number of probe points. The elements of each vector are an ASCII character taken from one of the following:

```
1 --- apply a logic one to this probe point
0 --- apply a logic zero to this probe point
z --- apply high impedance to this probe point
H --- expect logic one at this probe point
L --- expect logic zero at this probe point
x --- do not care what happens at this point
p --- a pin not connected to the tester, but connected to power etc.
c --- clock this pin mid-way through the cycle.
```

Typically other symbols enable special pulses or varying power supply voltages to be applied. The nature of these special signals is specified in separate tables included with the test program.

The vectors are applied in sequence at some clock rate (e.g. 10 million vectors per second) and there may be 100000 for a large chip or board.

If any of the H or L points do not match, the device under test has failed.

Example — a test of one part of a 7400 quad nand gate shown in Figure 15. We allow spaces in the vectors for clarity. For the first gate, pins 1 and 2 are inputs and 3 is the output. There are four vectors in this rather inadequate program.

```
   000 000 0 001 111 1
   123 456 7 890 123 4
 [ 00H 00x p H00 x00 p ]
 [ 01x 00H p H00 x00 p ]
```

```
[ 10x 00H p H00 x00 p ]
[ 11L 00x p H00 x00 p ]
```

> *Exercise:* Write a 100 percent coverage test program for a 7400.

5.3 The need for test modes.

For some designs it would take a very long test program to test certain functions. For example, the leap year circuitry in a digital watch is might need four years' worth of clock pulses before being exercised. Therefore, the throughput of the testing station would be low.

For other designs, the observability of internal state can be low. This occurs when there is a lot of internal state and few outputs. For example, a credit card PIN number checker need only have one output net, saying good or bad. Many different test sets would have to be presented before the affect of every part of the internal logic could be felt at the output net. This either leads to low fault coverage or long test programs.

Testability can be increased by adding either:

1. Additional outputs to bring out the state of internal nets
2. Additional inputs, which are tied off to one logic level during normal operation, but which during testing can shorten the length of counter cycles or cause more of the internal state to be brought out to the existing pins.

The addition of a single test input is often the preferred solution.

> *Exercise:* Consider the testability of devices with built in redundancy. One class of these devices is designed to automatically self-recover from partial failure: for instance, by detecting that a subsection of the logic is not working and patching in a spare, similar subsection to take its place. An example might be a RAM chip on a satellite.

5.4 Fault Simulator

The usual CAD tool which determines the fault coverage of a given test program against a given design is called the *Fault Simulator*. Its outputs are the fault coverage percentage and the list of stuck at faults not detected by the given test program. It is a varient of the normal simulator which is augmented by the keeping of a running set of which faults would have been detected at the current point in simulation. At the end of simulation, the list contains the undetectable faults and this is the output.

> *Exercise:* Design a trivial dynamic fault simulation algorithm whose expected running time is $O(n \times v)$ where there are n nets and v vectors. Can you design a better algorithm ?
> *Exercise:* Design a logic structure that is patently untestable. Would any such structure be needed in a real application ?

5.5 Test Program Generation

A test program generator is a CAD tool which generates a good test program for a design by analysing the structure of the design. It can also inform the designer that her masterpiece is difficult to test, allowing her to make modifications at an early stage.

Manual test programs are often generated from the designer's functional simulations — the ones she has used to determine whether the design meets its specification. Typically the designer concatenates each of these simulations (along with some others which she feels may bump up the fault coverage) and feeds them into the standard simulator. The simulator is run in a mode where it prints a test vector to a file at a certain fraction through every clock event and this file is then the test program. It is clear that a working device will pass this test program, but there is no guarantee that such an approach will yield sufficient fault coverage.

For largely combinatorial designs, the maximum possible fault coverage can be fairly accurately determined by a static fault coverage analysis program. This operates with algorithms based on sets of detectable faults present at the inputs of gates to determine the set of detectable faults at the output of the gate. Repeated application gives the externally observable faults.

Automatic test program generation can involve sophisticated logic analysis algorithms. The section in Kinniment and McLauchlan on this subject is very good.

5.6 Boundary scan

Test vectors have traditionally been applied to the device under test using a separate wire from the testing machine to each probe point. This is suitable when the device under test is

1. a die on a wafer
2. a packaged chip
3. a populated printed circuit board.

But today it is desirable to apply these same test programs to the component parts of a complete assembly. Examples are:

1. a multi-board finished product
2. a macrocell placed on a larger ASIC.

With boundary scan, the chip can be put into a boundary scan test mode where the IO pads of a chip are connected into a shift register. The shift register is a chain of flip-flops, one being built into each IO pad for the purpose.

In test mode, the test vector is shifted into the flip-flops. The signal on an input pad is overridden by the value in the scan flip-flop. The test path is then activated using a strobe line which latches the values on the output pads into the scan flip flops. The test vector with output data is then shifted out and examined as the next vector is shifted in.

IO pads are intrinsically big (to get the bond wire on) and slow (owing to the large transistors and diodes used in them). The speed and complexity penalty of adding boundary scan logic to them is therefore low.

Owing to the serial nature of scan path testing, multiple chips can be put in series on the scan path. These chips can be macrocells on one large piece of silicon, or spearate pieces of silicon on a circuit card.

Speed of testing can be a problem for long scan chains. Clearly the rate of exchange of test data between the device under test and the testing machine is much lower with a serial interconnection than offered by the multiple parallel paths of the bed-of-nails approach.

5.7 General scan path

General scan path testing is similar to boundary scan, but threads the scan path through all of the flip-flops within the design.

In addition to its normal connections and operations, each flip-flop must support a scan mode and has an extra input for connection to the output of the previous flip-flop in the scan path and a connection to a global control net which enables scan mode.

- Advantage — full testability and observability
- Disadvantage — increase in complexity and reduction in device speed.

5.8 JTAG boundary scan.

The IEEE 1149 (JTAG) international standard defines four wires per chip available for boundary scan. These are:

tms	—	test mode select — put high to enter boundary scan mode
tdi	—	test data input — serial data input
tck	-	test clock to clock each serial bit in
tdo	-	test data output to read back old data as new is shifted in.

For normal operation of the chip, connect the three inputs to logic zero (ground). A fifth wire, test reset, is often found in new implementations.

Figure 16 shows a boundary scan path that is inserted as part of the electronics of each input or output pad. The system includes a few gates and flip-flops at each pad and a daisychain of wiring from one pad to the next around the chip perimeter. Pads which are bidirectional essentially consist of an input pad together with an output pad where the bond pad is shared.

When the test mode select is low, the boundary scan logic has no effect, but when in test mode, the scan logic takes over from the input pads and enables monitoring of the output pads.

The scan logic forms a shift register with one bit per pin. The JTAG test data input and output are placed at the ends of the shift register and the clock is the JTAG test clock. An internal counter, not shown in the figure, counts the shifited bits until a whole new vector has been inserted and then generates strobe signals to transfer data in and out of the holding data registers in each pad.

The strobes are generated from a small counter circuit which knows the number of pads scanned and clocks the vector in parallel form once the required number of bits have been shifted.

Disadvantage: The main problem with JTAG is that four pins is quite a large number for small devices.

Advantage 1: Chips can be tested in place on their final circuit board (an *in-circuit* test).

Advantage 2: If the 'chip' is actually a macrocell built into a larger chip or module, then the 'chip' can be tested as an autonomous subunit, reusing its own standard test program.

The JTAG port of a chip is becoming used for additional functions:

- device type and serial number,
- in-circuit programming of reusable FPGAs,
- single step debugging of embedded microprocessors.

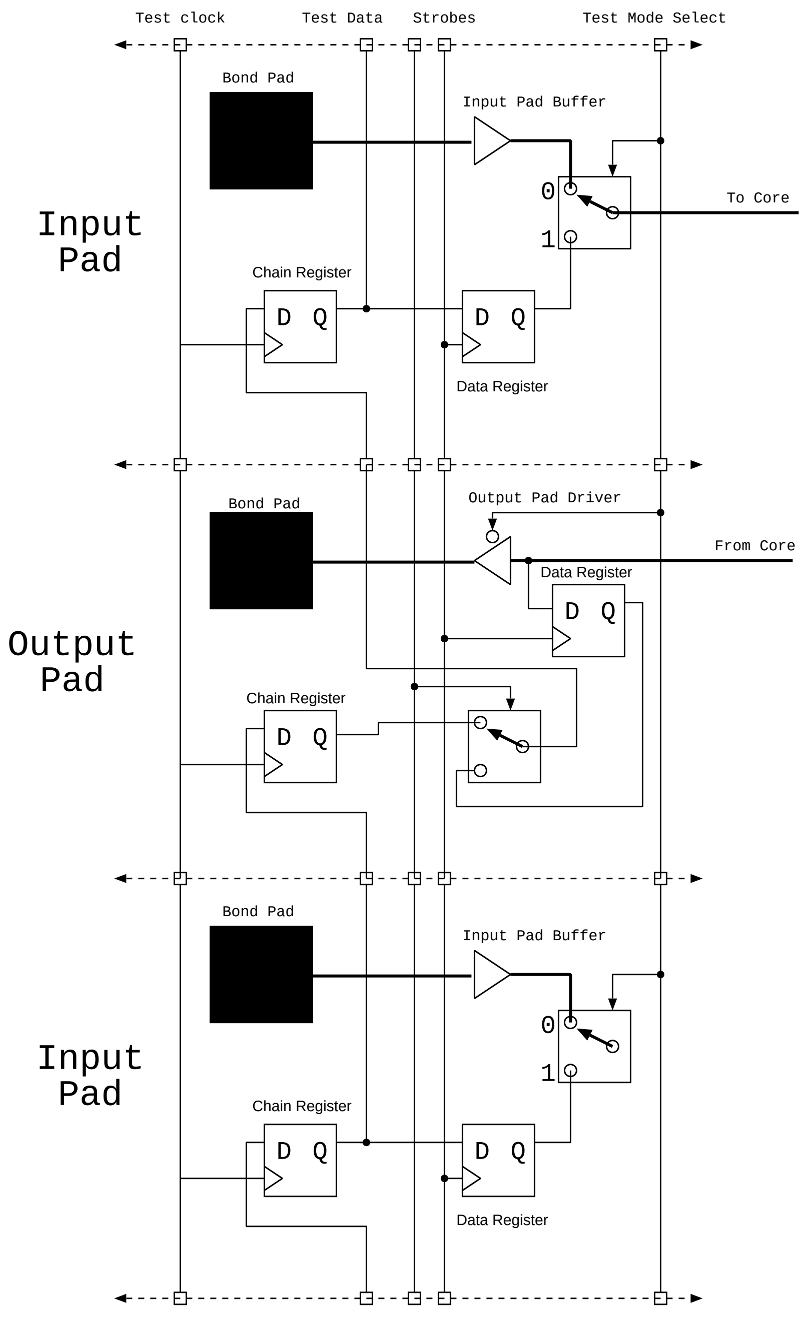


Figure 16: Circuit needed in input and output pads to achieve boundary scan

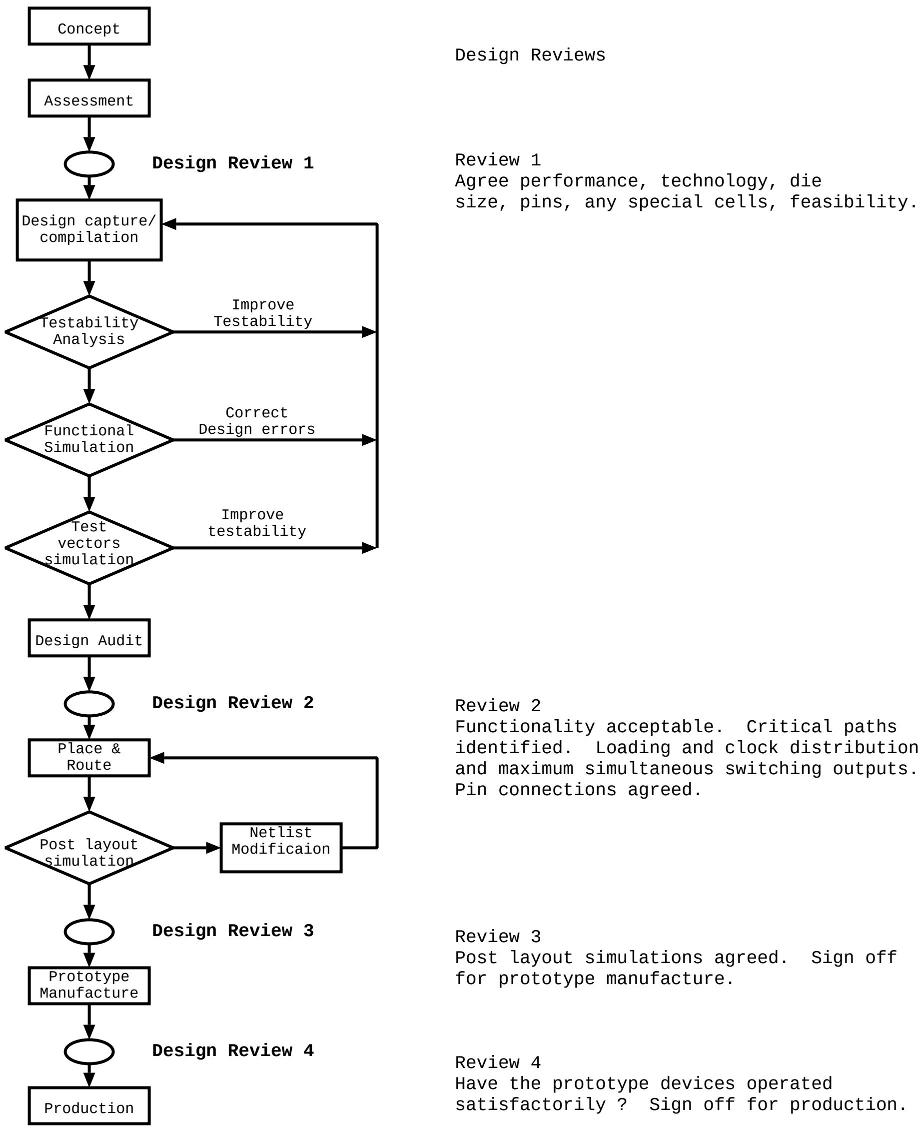


Figure 17: Illustration of contractural design flow between a design house and a foundry.

6 ASIC Design Flow

The design flow for an ASIC must be highly structured for efficiency reasons and to have a clear contractural arrangement between the companies involved. These could be as many as the designer's own company, various library and macrocell block vendors, a synthesis tool vendor and a foundry.

The design flow for complete systems or a multi-chip sub-system accords very closely.

At each step in the flow, the following rule applies: *'repeat previous steps as necessary to sort out the problems that we encounter at this step.'*

6.1 Top level circuit and sub-circuit entry

The top level file for the design must be entered into the design system database. A combination of behavioural specification, schematic entry, direct netlist entry using a text editor and generation from miscellaneous tools, such as C programs, can be used.

The designer will generally have a good idea of the design partition requirements of the system at this stage, except when using field programmable logic devices, such as PALs and FPGAs, where the exact number of such devices is often uncertain initially.

6.2 Preliminary choice of leaf parts from standard libraries

An initial selection of parts from standard libraries must be made and entered into the design database. In early iterations of the design cycle, the aim is to achieve a simulatable system, to gain confidence in the overall system design. The designer will not care which particular manufacturer's logic gate is used, or whether parts of the system are still represented behaviourally, provided simulation can start.

In final iterations, the designer will have to specify part numbers and technologies precisely, make sure that the correct libraries are present on the system and that there will be no unexpected licensing fees to pay for using them.

6.3 Design database rule checking

The system will make as many design rule checks as possible to ensure the validity of the design. It will flag any errors and warnings.

6.4 Netlist closing and stimulus generation

It is not useful to simulate a netlist with open input ports, so fictional components must be added to close the netlist. These devices may start doing things autonomously, such as sending data to the system. These input events, along with events on input devices which are proper parts of the system, such as switches and other sensors, must be modelled and are called the stimulus.

The designer generates sets of stimuli that will be used for the simulation and enters it into the CAE system. It can be as simple as a list form, giving the event and the time after the start of simulation when it should occur. Some simulators support interactive input, so the designer can enter stimuli at any point during simulation.

6.5 Simulation

The system is simulated, normally using an event driven simulator. This type of simulator keeps a list of events, which are changes in the logic state of a net, sorted into increasing time in the future. Events are created initially by clocks or stimulus generators. They are processed in turn by the simulator by communicating them to the *models* for each component instance. These models create new events in the future, which are merged into the event list. An inverter model, for instance, would generate an event for its output net to change as a result of the an event on its input net. If the input went from low to high, the output event generated by the inverter model will be a high to low change at the time of the input event plus the propagation delay of the inverter.

Software tools known collectively as *diagnostics* enable the designer to trace and log any aspect of the simulation, such as the cycle by cycle transfers along one of the busses or the series of light pulses entering a model of an optical fibre.

Many of the models will report run-time errors or warnings. Typical messages are

- tri-state net xxx has multiple drivers from time yyy to zzz.
- set up time on d input of flip-flop instance iii violated by ttt ns.
- uncertain data written to location aaa of RAM memory iii.

The correct operation of the system is sometimes checked with high level tools which compare functionality with specification but often the designer will just look at the simulated waveforms and other data and be confident.

6.6 Final partition

The precise implementation technology for every component is selected. This gives us a completed design topology and netlist, but little idea of geometrical arrangement. For the FPGA parts of the system, the number and type of devices to be used becomes clear. Functions which have been implemented in FPGAs for development will now perhaps be reimplemented in masked ASICs to reduce production cost.

The tools can estimate the overall design cost in terms of total chips and silicon area and can give estimates of power consumption.

6.7 Placement

Placement is the process of assigning (x, y, z) coordinates to each part. The goal in placement is normally to minimise the total length of conductor needed to wire up the design. The hierarchy of the design is refered to as a basis for finding closely interconnected cliques of components which can become neighbours (but this information is lost if too much flattening has occured). The Z coordinate is null for many technologies, but for printed circuit boards and complete systems we have the choice of which board and which side of the board, and sometimes little components can be put under big ones with long legs.

Placement can be done by software or manually, or with any level of human participation in between.

6.8 Routing

Routing is the process of determining the route for the conductors that make up the design, so that all the required contacts are joined up. Routing is difficult, since inserting one conductor can block the route desired by another. Routing requires fairly sophisticated algorithms which are a careful blend of a greedy algorithm and a lot of trial and error. Some routers use *shove aside*, which moves already routed conductors sideways a bit to introduce room for the current conductor. The most recent routers put all of the conductors in place without worrying about shorts in the first instance, and then perform iterated layer swaps and fragmentation of the conductors by inserting vias until no shorts remain.

Routing takes place on layers. These are layers within a multi-layer printed circuit board (four to ten layers is common) or layers on a chip (two or three layers of metal on top and one layer of polysilicon in the substrate are common resources).

Normally power supplies are given their own layer(s) (both on circuit boards and chips) and two layers of signal routing are needed. A heuristic which allocates mostly X direction sections on one layer and Y direction sections on the other is generally used, since conductors cannot cross each other within one layer.

It is desirable to have interplay between the routing and the placement stage. Typically CAD tools do not automatically do this. When routing is *impossible* on the given number of layers and with the given placement, one or other needs to be changed.

When the routing tools fail, it is common to adjust the placement manually (using a mouse and interactive tools) or just to run the placement tools again with new seeds in their random number generators.

Finding optimum routings and placements for a design has complexity of at least NP-hard.

Power supply range	4.5 to 5.5 volts
Temperature range	0 to 70 degrees ambient
Process variation	+/- 0.2 in speed

Table 1: Example parametric variations.

Clock signals and any other critical signals are sometimes routed by hand or routed first so that they can take the best paths.

6.9 Delay estimation

Once the system has been placed and routed, the length and type of each conductor is known. These facts allow fairly accurate delay models of the conductors to be generated.

6.10 Back annotation and parametric simulation

The accurate delay information is fed into the main simulator and the functionality of the chip or system is checked again. This is known as back annotation. It is possible that the new delays will prevent the system operating at the design speed.

Another tool which is useful at this time is a *static timing analyser* which, rather than simulating a circuit, walks over it summing delay paths from outputs back to inputs, in order to flag possible timing errors.

Parametric variation is often taken into account at the same time. This refers to simulating the system at the boundaries of its power supply, temperature and process specifications. Example parametric variations are shown in table 1. The total variations in component delay after back annotation and parametric variation may be 2 to 1.

It is sometimes convenient to generate behavioural models for additional, fictional components which are instantiated in the simulation. They monitor sequences of events on their inputs and flag illegal or incorrect cases. A extreme case is a *yes-no test wrapper*.

6.11 Yes/no test wrappers.

Using Verilog or similar, one can write a test wrapper for a design that produces a single bit of output: yes or no. The wrapper has built in to it a model of the design under test and, for each output compares the the values coming from that model and the model under test using '==' or similar. The two models being compared are typically versions of the same design taken from different parts of the design flow: for example, the entry level design and the back-annotated netlist.

In Verilog, a simple example where we are testing a NAND gate, is

```
reg fail;              // one such fail variable for whole test
initial fail = 0;

initial @(posedge end_of_simulation) if (fail) $display("Failed");
        else $display("Passed");


// For each output or module under test, we do the following

NAND2 device_under_test(outp, ina, inb);
assign model_output = ~(ina & inb);
always @(posedge teststrobe) if  (model_output != outp) fail <= 1;
```

Typically there will be many such statements that update the 'fail' variable. (Such a wrapper is not synthesisable because of this.)

Yes/No wrappers allow the results for many simulations run as batch jobs to be quickly analysed. The simulations typically vary in their parametric settings, enabling the fact that, for instance, a design fails at low temperature and high supply voltage to be quickly determined.

Test programmes based on test vectors can perform a similar function, but only provided the output and input sequences are identical for each run. The Yes/No wrapper allows the stimulus input sequences to be a parameter varied from simulation to simulation.

6.12 Static timing analyse.

Simulation is a time-intensive method of verifiying that a design performs its desired function.

Set-hold errors from the simulator are not a reliable means of checking that the design will run at speed, since signals may arrive after the hold time when intended to arrive before the set-up time or the critical path may not be excercised during the simulation.

A static timing analyser works from all D-inputs back along the netlist finding the longest paths and then reporting on the maximum clock speed.

Such a tool needs to know which paths are not being used in a design, or which will be static during operation, since these will typically be routed with slow nets. It takes a list of these, typically manually generated, and discounts them from the analysis.

The xdelay command on Thor is such a tool and can be used on a Xilinx .lca file. Try it.

6.13 Test program generation and fault simulation

The fault simulator is run to check the fault coverage of the test program, as described in Section 5.4. The level of fault coverage required is often a contractural parameter between a foundry or assembly house and the designer.

6.14 FPGA Implementation/Emulation

When a design involves embedded processors or when a great deal of software is needed as part of the target system, development of the software is generally hampered since the silicon implementation is not available. Faults or specification changes to the silicon could be determined if *virtual* silicon was available.

An emulation of the entire system before fabrication is possible using FPGAs. The emulated system can act as a software development and testing platform. Since FPGAs are less dense than ASICs in terms of useable gates, hundreds of FPGAs are needed to emulate the ASIC. The speed of FPGAs is lower than the equivalent ASICs and the penalty of all the interchip connections in the FPGA version reduces the speed further. Operational speeds of 1/50th the target speed are often possible.

A market leader for this time of emulation is Quickturn. The selling price of one of their boxes of FPGAs is around one million dollars. See http://www.quickturn.com/technology/emulation.htm.

6.15 Mask making or 'taping out'

The artworks for chips and printed circuit boards leave the computer and become physical entities. The artworks are photoplotted; that is, very accurately plotted on transparent film or glass. Actual size photoplots for PCBs can be used. For chips, actual size is too small, so photo-reduction is used after plotting at larger scale. The photoplots are put into various projectors (to make masks for chips and expose photo resistive coatings for PCBs etc) and everything is made and assembled.

A polygon editor can be used for minor modifications to the artwork: e.g. adding text or manually rerouting a net. After each set of such manual modifications, a *netlist extractor* runs over the artwork to read back the netlist and then this is compared against the previously known netlist to check for errors.

6.16 Alpha testing

Once the first test wafers of chips have been processed, one or two devices are soldered to a test circuit board or a prototype of the whole system is assembled and given to the designer. She switches it on and stands well back.

6.17 Product and design development

The next stage is typically for a small run of 10 or 20 units to be made — the beta prototypes. At the same time, orders will be placed for the components to be used in production and factories for assembly identified. Production test programs must be developed. The product must be measured on an emissions range for succeptibility and emission of radio waves (EMI/EMC testing). Cardboard boxes, datasheets and manuals, vendors, distributors, telephone enquiry service, field repair agents, advertising and many other things have to be put in place.

If the design is successful, you can now start on the new improved version, to hit the streets as sales of the current product drop off. If the design is bad, then you may find yourself stuck in a factory in the Far East with an oscilloscope and thousands of not working circuit boards.

We are now in the realm of production engineering and manufacturing, which is worth knowing about if you want to be a manager!

6.18 Sign offs, prototypes and production.

Figure 17 shows an overview of the design flow process for an ASIC from the point of view of interaction between a design company and an ASIC implementing company.

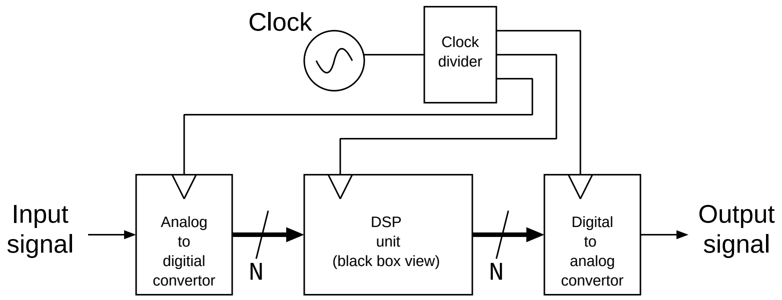


Figure 18: Simple application of digital signal processing (e.g. scratch removal on old LPs).

7 Speed, Complexity and Folding.

For most designs there are many possible implementations, but in this section we will study the plentiful and obvious options which are apparent with arithmetic-intensive designs that have a low amounts of IO. The design decisions revolve around where to store the data and how many processing elements to use.

7.1 Digital Signal Processing and Processor

The process of applying an algorithm to a stream of digital samples (quantised analog values) as a time domain series is known as digital signal processing (DSP). A digital signal processor (also called DSP) is a processor which is optimised for digital signal processing, and typically has special addressing modes for single-cycle multiply-accumulate and arithmetic that saturates rather than overflows.

Two's complement arithemetic is normally used. Floating point and sign plus log-magnitude are occasionally used.

Digital Signal Processors are not covered in this course. Instead here we consider hardwired implementations of DSP algorithms.

DSP design examples include satellite receivers, modems and video compression devices. In these examples, the data rate in and out of the unit is specified in words per second and the algorithm needed may also be precisely stated as a data flow graph (see next subsection), but there is considerable flexibility over the actual implementation. This leads to a *blackbox* diagram during top-down system design (figure 18). Often the algorithm is well known and can be found in textbooks and papers from previous decades: what is new is the potential to make a single-chip implementation. Othertimes the algorithm is new (e.g. scratch removal from LPs) and we just wish to implement it easily and with low production cost. In most cases, the degree of custom hardware, standard processors and software to use is a free design choice.

7.2 Data flow graphs for digital signal processing.

Another property of DSP applications is that many of them tend to have very little data dependency over their flow graph. That is, they perform the same sequence of operations on the input words to generate each output word, regardless of the values of the input words. When there is some data dependency, it can conveniently be expressed as a conditional expression, rather than a fork of program flow (example given in lecture). A data flow graph (DFG) is an abstract drawing which shows the processing steps and precision used (in terms of bits), but does not show clock signals or anything specific to an implementation.

For example, we will consider the finite impulse response (FIR) filter. These commonly occur in DSP systems. They might have between four and one hundred taps, depending on the application.

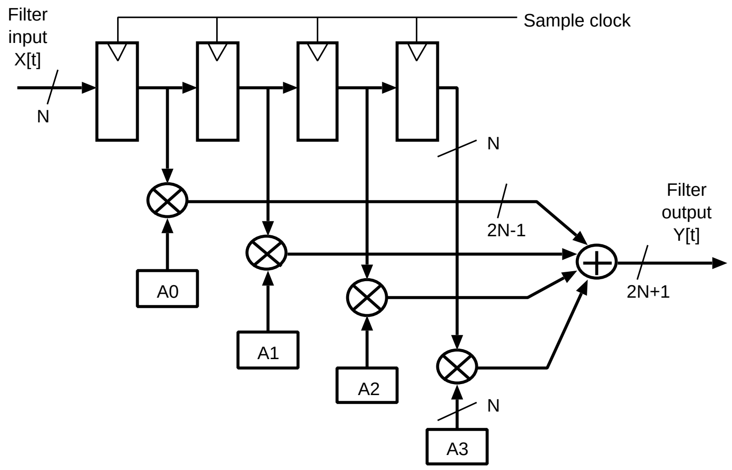


Figure 19: A 4 stage FIR filter.

Figure 19 shows one of several possible psudo data flow graph for an FIR filter. Alternative DFGs exist that compute the same result have the multipliers between the delay stages. The figure is also the circuit diagram of one implementation of the desired function. Such filters are very processing intensive, since if there are N taps, then N multiplies and an N input add are needed per sample.

In the figure, N is four and input data words enter a four stage, broadside shift register, formed by the four rectangles. The four, crossed circles denote multipliers and the values of A_i in the ovals are constants (known as the filter coefficients). When two N bit numbers are multiplied, the result is $2N$ bits for unsigned arithmetic and $2N - 1$ for signed arithmetic. (The bus widths on the diagram assume signed arithmetic using two's complement.) The output circle is an adder and when four numbers are added, the output width increases by two bits.

The filter coefficients alter the filter response. When all are zero the output is zero and when only one is non-zero there is a flat frequency response. Other settings are beyond the scope of this course [see *'Digital processing of signals'* by B Gold and CM Rader, McGrawHill].

The sample rate at the input and the output is the same and must be the rate of the illustrated sample clock. The diagram is a psudo data flow graph (DFG) in that a clock net is shown. We can turn it into a pure DFG by deleting the clock net and labeling the rectangles with the symbol Z (or in some texts Z^{-1}. Then the output o can be stated mathematically as a function of the input x

$$y = (A_3 z^3 + A_2 z^2 + A_1 z + A0)zx$$

For all data flow graphs, there is an equivalent, functional or mathematical expression.

Large functions tend to become unweildy and hard to understand when expressed functionally or as a z-domain polynomial, so either the DFG is drawn out manually or it is expressed in a *single assignment language* and a common such language is *Silage*. In a single assignment language, each variable occuring on the left hand side of an assignment only occurs once per program. (When there is a data dependency, the single assignment model can often be extended with the rule that each variable only gets assigned to once on any given flow through the program.)

Note: No aspects of the Silage language itself are examinable.

A Silage definition of the FIR filter is shown in figure 20. To make the program complete, defintions of the input X and the output Y are needed: these will refer to infinite lists of samples. The definition of A might declare it as a constant array. Also, the working variables T and S need to be defined. The construct `(i: 1..3) S` is perhaps the only obviously obscure section of code, and means that the statement S (in this case the beng-end block) should be 'macro' expanded or

```
T[0] = A[0] * X[0];
(i: 1..3) :: begin
    T[i] = A[i] * X[i];
    S[i] = S[i-1] + T[i-1]
    end;
Y = S[3] + T[3];
```

Figure 20: A Silage program for the 4 stage FIR filter.

'generated' three times over with i taking on the numbers 1..3 each time. Note that each variable (array member) is assigned only once.

In the source of this code example [Computer Lab TR287], it says that the somewhat convoluted form of the progam was chosen by the writer so that the computation of `T[i-1]` occurs the cycle before that of `S[i]`. However, the Silage program does not directly imply an implementation or mapping to clock cycles, but perhaps early implementations used only a direct and obvious mapping to implementation.

The inputs to single assignment languages are the variables which are not assigned to at all, and these input variabled implictly denote an infinite (lazy) list of actual input values.

Programs in single assigment languages can be converted by computer fairly easily into functional expressions. They can also be drawn out as a data flow graph, where each box is a either an input variable or an intermediate variable and each circle is an arithmetic or logical operation on its inputs.

7.3 Implementation decisions.

Given the input and output word rates and the algorithm, one can readily determine the number of additions and multiplies per second required to implement the function. The main design implementation decisions are then:

- What system clock rate to aim for (normally some integer multiple of the IO word rate),
- How to store the working data and intermediate results
- How many individual adders and multipliers to use

In a good compiler for Silage (such as CATHEDRAL-II), many possible implementations are considered and the best is chosen. For instance, for the FIR filter, we can use one multiplier four times per sample, two multipliers twice or four separate multipliers. Both the multiply and the addition function used have freedom over the type of carry chain generated: for instance, with a simple pipelined break in the carry chain, the top $N-1$ bits of the output might be generated the cycle after the lower N bits. This is fine if the next stage knows this fact, which it will do if it is synthesised at the same time.

In general, we wish to use the highest clock speed and the lowest amount of silicon. Decisions over the order of calculation of intermediate results are coupled with the amount of store required, since a memory location or register that is read soon after writing can then be used to store something else.

7.4 Demo of DSP compiler.

This section will conclude with a demo of a DSP compiler and we will discuss a handout which shows various implementations of a function, each with different resource use.

Exercise: Consider the optimisations that can be made to the implementation of an FIR filter if it is known that only every fourth output value is to be used, with the three in between being ignored.

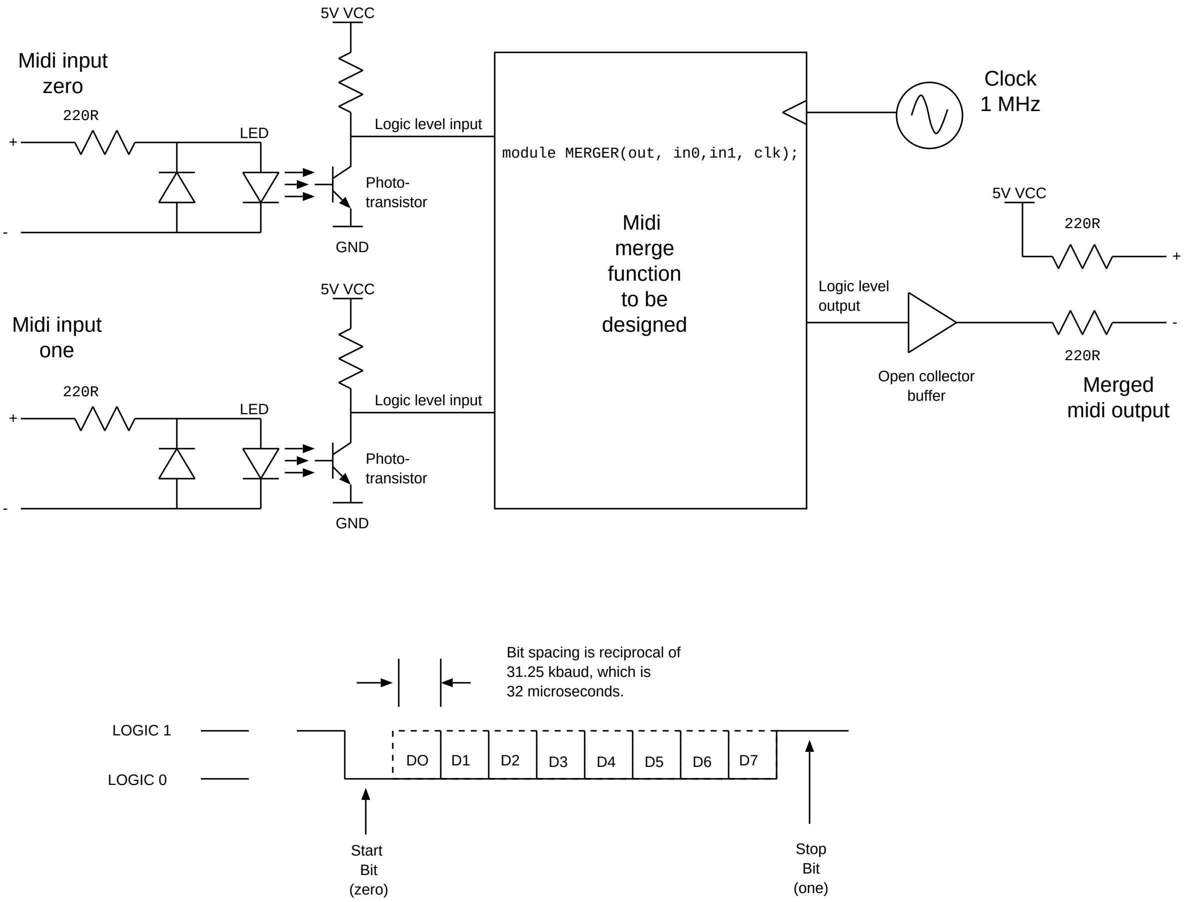


Figure 21: MIDI interface details and merge unit block diagram. MIDI serial data format.

8 Design Example.

This section conatins a design example. The design example is a MIDI merge unit. *No details in this section are examinable.*

8.1 MIDI Merge Unit Specification

MIDI is the musical instrument digital interface and it is found on most electronic musical intruments. A MIDI connection is unidirectional and contains both control and real-time traffic. In this example, we will only concern ourselves with the real-time *note* data which is sent from a keyboard to a sound generator as the keyboard is played. A three-byte note command is sent for each key pressed or released on the keyboard. The three MIDI note commands we are interested in have the following formats (in hexadecimal)

```
9n kk vv      (note on)
8n kk vv      (note off)
9n kk 00      (note off with zero velocity)
```

where n is a channel number, kk is a note number and vv is a velocity. The values of n range from 0 to 15 and allow a single MIDI link to carry 16 virtual MIDI channels of data. The values of kk are in semitones in the range 0 to 127 with 60 being middle 'C'. The values of vv are from 1 to 127 where 127 is the loudest. Note that these parameters are all less than 80 hex. The first command (note on) is generated when a key is pressed and the second (note off) when it is released. The

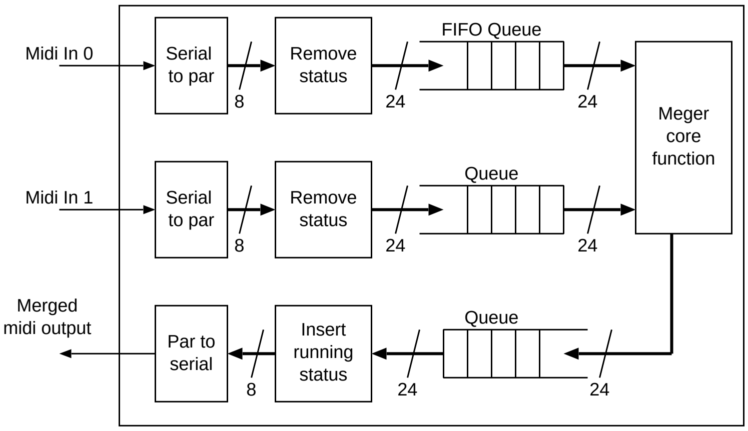


Figure 22: MIDI merge unit internal functional units.

third is an alternative to the note off command when the release velocity is not used (which is the case for most keyboards). The benefit of the alternative accrues when *running status* is used. Running status is an optimisation of the MIDI protocol which brings the average number of bytes actually sent per note command down from three to closer to two. Since this is a real-time music playing protocol, and many note commands need to be sent when a large chord is played, reducing the number of bytes in this way is helpful, but it will impact our merge function. A status byte in MIDI is a MIDI byte in the range 80 to EF, and so this includes the note on and off commands. The rule for running status is that if a status byte needs to be sent and it is the same as the last one actually sent, then it does not need to be sent.

Figure 21 shows the desired configuration of the MIDI merge unit and the format of MIDI bytes on the wire. The actual MIDI cables operate with two wires carying the send and return currents: this is known as a current loop circuit. The current is on for a zero and off for a one. The current is sufficient to drive a small led which is mounted in a dark space next to a photo-transistor. This optical link between devices avoids electrical connections which can cause earthing problems in a studio environment. When light falls on the photo-transistor, it conducts and pulls down the input wire to close to ground, creating a valid logic zero level. Otherwise the inputs are one. One is the idle line state. A MIDI bytes starts with a start bit of zero for 32 microseconds, followed by the data bits, l.s.b. first.

The function of the MIDI merge unit is to merge two streams of MIDI data: e.g. two different channels of MIDI bytes (different values of n) may need to be merged. The unit must be prepared to accept arbitrary timing and status relationships between its two inputs. If they are simultaneously active, then the data from one of the inputs will have to be slightly delayed and sent after the data from the other. If both are busy to more than 50 percent utilisation, the output cannot cope and some data must be discarded by the unit. Since this is a real-time system, the output needs to have as low a delay as possible from the input.

8.2 Top down design.

The next stage of design is to sketch an internal block diagram, as shown in figure 22. The system will assemble bits into bytes, then assemble bytes into three-byte command words. The command words will be stored in FIFO (first-in first-out) queues and then read out and merged. The reverse process is done on the transmit side.

The blocks in the block diagram are each suitable for encoding as a Verilog module. Except for the final parallel to serial block, each module will need an output guard signal which indicates when the data on the output wires of the module is valid. For most of the blocks we will define this to go high once, each time a new word is valid. For the FIFO blocks, we will implement a *handshake* on the output, where the valid signal indicates that the FIFO has some data in it (and that that data is currently sitting on the output wires) and a read signal which indicates that we have accepted the data, thereby causing the next data to move to the output.

We can define the signatures of the verilog modules as follows. First, the serial to parallel convertor

```
input clk;
output [7:0] pardata;   output guard;
```

The running status remover should be

```
input clk;
input guard_in;   input [7:0] pardata_in;
output guard_out; output [23:0] pardata_out
```

For the FIFOs we should have

```
input clk;
input guard_in; input [7:0] pardata_in;
input read; output guard_out;  output [23:0] pardata_out;
input read; output guard_out;  output [23:0] pardata_out;
```

For the merge core unit we should have

```
input clk;
input guard_in0; input [23:0] pardata_in0; output read0;
input guard_in1; input [23:0] pardata_in1; output read1;
output guard_out; output [23:0] pardata_out;
input read; output guard_out;  output [23:0] pardata_out;
```

And for the running status inserter and parallel to serial unit the signatures will be the reverse of the reciprocal units.

A full set of Verilog for the design will be put on the ECAD WWW page http://www.cl.cam.ac.uk/Teaching/1998/InroECAD. As much as possible of the design example will be covered in lectures.

8.3 Software Alternative

The function could be implemented with a single-chip processor with dual serial ports. A suitable device is a PIC from Microchip and would cost just a pound or so. See http://www.microchip.com.

> *Exercise:* Consider the pros and cons of the software implementation compared with the hardware implementation. Consider that the MIDI standard is mature and has not significantly changed in ten years. Consider the real-time response possible. *This exercise might be better undertaken after the Computer Design course.*

8.4 ECAD Challenge

Because the MIDI protocol works in real time, for the best performance, the MIDI merge unit should have a very low delay through it. The minimal possible delay might be just one clock tick, but the design presented in lectures and above always completely receives each command before sending it onwards. A good part II project would be to write a program which read in the netlist of the MIDI merge unit and wrote out a netlist of an improved design with lower delay. The program would work by applying general purpose transformations on the netlist which can manipulate it to achieve some goal, such as low delay.

End of document.