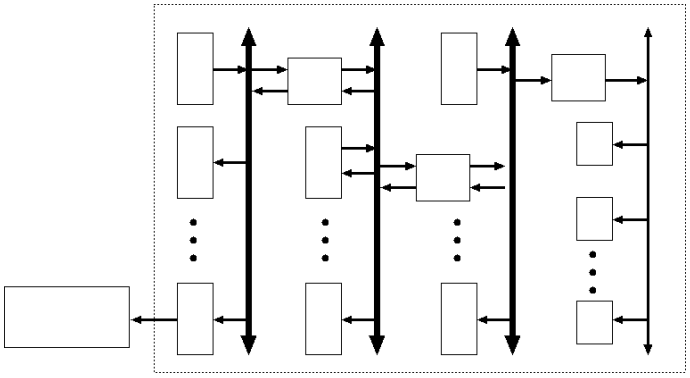
System on Chip Design and Modelling



University of Cambridge
Computer Laboratory
Lecture Notes

Dr. David J Greaves

Part II
Computer Science Tripos
Easter Term 2010

- **(1) Introduction: What is a SoC ?**
- **(2) Review/Revision of Verilog RTL**
- **(3) SystemC Components**
- **(4) Basic SoC Components**
- **(5) ESL: Electronic System Level Modelling**
- **(6) ABD - Assertion-Based Design**
- **(7) SoC DRAM and Bus/NoC Structures.**
- **(8) SoC Engineering and Associated Tools**
- **(9) High-level Design Capture and Synthesis**

0.1 Syllabus

We will cover many of the following topics:

1.Verilog RTL Design with examples. Basic RTL to gates synthesis algorithm.

2. Further examples. Event-driven simulation cycle. Using signals, variables and transactions for component inter-communication.

3. SystemC overview. Verilog synthesis and high/low-level mapping examples.

4. High-level modelling in SystemC. Bus and cache structures, DRAM interface. Design exploration.

5. Transactional modelling (ESL). Electronic systems level design. IP- XACT.

6. Processor Modelling. Instruction set simulators, cache modelling and hybrid models.

7. Assertions and Monitors. System Verilog brief tour. PSL/SVA assertions. Temporal logic compilation to FSM. Assertion-based design.

8. On Chip Interconnect. Busses (OPB (BVCI) and AXI). Glue logic synthesis. Transactor Synthesis. Network on chip.

0.2 Recommended Reading

Design And Reuse

OSCI. *SystemC tutorials and whitepapers* . Download from OSCI www.systemc.org or copy from course web site.

Ghenassia, F. (2006). *Transaction-level modeling with SystemC: TLM concepts and applications for embedded systems* . Springer.

Eisner, C. & Fisman, D. (2006). *A practical introduction to PSL* . Springer (Series on Integrated Circuits and Systems).

Foster, H.D. & Krolnik, A.C. (2008). *Creating assertion-based IP* . Springer (Series on Integrated Circuits and Systems).

Grotker, T., Liao, S., Martin, G. & Swan, S. (2002). *System design with SystemC* . Springer. Wolf, W. (2002). *Modern VLSI design (System-on-chip design)* . Pearson Education. LINK.

LG 1 — Introduction: What is a SoC ?

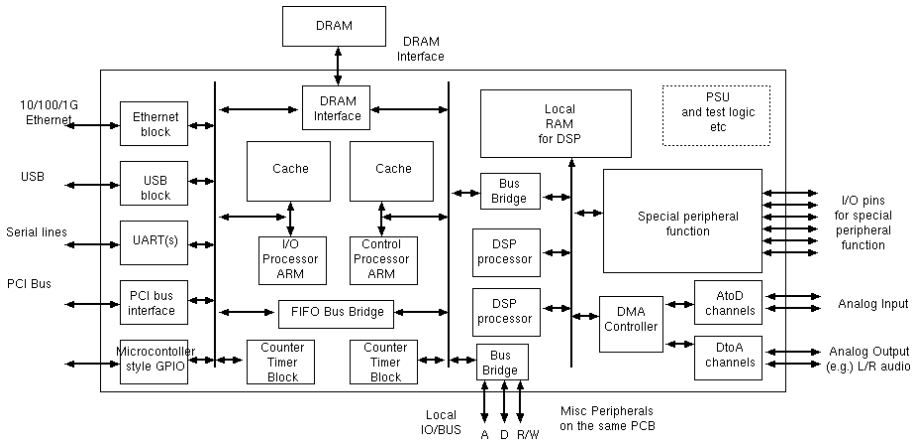


Figure 1.1: Block diagram of a multi-core 'platform' chip, used in a number of networking products.

A System On A Chip: typically uses 70 to 140 mm^2 of silicon.

SoCs are found in every consumer product, from modems, telephones, DVD players, televisions and iPODs.

A SoC is a complete system on a chip. A 'system' includes a microprocessor, memory and peripherals. The processor may be a custom or standard microprocessor, or it could be a specialised media processor for sound, modem or video applications. There may be multiple processors and also other generators of bus cycles, such as DMA controllers. DMA controllers can be arbitrarily complex, and are really only distinguished from processors by their complete or partial lack of instruction fetching.

Processors are interconnected using a variety of mechanisms, including shared memories and message-passing hardware entities such as specialised channels and mailboxes.

SoCs are found in every consumer product, from modems, mobile phones, DVD players, televisions and iPODs.

1.1 Design Flow

Design flow is divided by the **Structural RTL** level into:

- **Front End**:specify, explore, design, capture, synthesise
- **Back End**: place, route, mask making, fabrication.

1.2 Design Flow Diagram

Figure 1.2 shows a typical design and maufacturing flow that leads from design capture to SoC fabrication.

1.2.1 Front End

The design must be specified in terms of high-level requirements, such as function, throughput and power consumption.

Design capture: it is transferred from the marketing person's mind, back of envelope or or wordprocessor document into machine-readable form.

Architectural exploration will try different combinations of processors, memories and bus structures to find an implementation with good power and load balancing. A loosely-timed high-level model is sufficient to compute the performance of an architecture.

Detailed design will select IP (interlectual property) providers for all of the functional blocks, or else they will exist from previous in house designs and can be used without license fees, or else freshly written.

Logic synthesis will convert from behavioural RTL to structural RTL. Synthesis from formal high-level forms, including SysML statecharts, formal specifications of interfaces and behaviour is becoming possible.

Instruction set simulators for embedded processors are needed: purchased from third parties such as ARM and MIPS, or as a by-product of custom processor design.

The interface specifications (APIs) between components need to be stored: the IP-XACT format may be used.

High-level models that are never intended to be synthesisable and test bench components will also be coded, typically using SystemC.

1.2.2 Back End

After RTL synthesis using a target technology library we have a structural netlist that has no gate delays.

Place and route gives 2-D co-ordinates to each component and adds external I/O pads and puts wiring between the components.

RTL annotated with actual implementation gate delays gives a precise power and performance model. If performance is not up to par, design changes are needed.

A library of standard tests will be run every night and any changes that cause a previously-passing test to fail (regressions) will be automatically reported to the project manager.

Fabrication of masks is commonly the most expensive single step.

1.3 Levels of Modelling Abstraction

Our modelling system must support all stages of the design process, from design entry to fabrication. We need to mix components using different levels of abstraction in one simulation setup.

Levels commonly used are:

- **Functional Modelling:** The 'output' from a simulation run is accurate.
- **Memory Accurate Modelling:** The contents and layout of memory is accurate.
- **Untimed TLM:** No time stamps recorded on transactions.
- **Loosely-timed TLM:** The number of transactions is accurate, but order may be wrong.
- **Approximately-timed TLM:** The number and order of transactions is accurate.
- **Cycle-Accurate Level Modelling:** The number of clock cycles consumed is accurate.
- **Event-Level Modelling:** The ordering of net changes within a clock cycle is accurate.

Other terms in use are:

- **Programmer View Accurate:** The contents of visible memory and registers is as per the real hardware, but timing may be inaccurate and other registers or combinational nets that are not designated as part of the 'programmers view' may not be modelled accurately.
- **Behavioural Modelling:** Using a threads package, or other library (e.g. SystemC), hand-crafted programs are written to model the behaviour of each component or subsystem. Major hardware items such as busses, caches or DRAM controllers may be neglected in such a model.

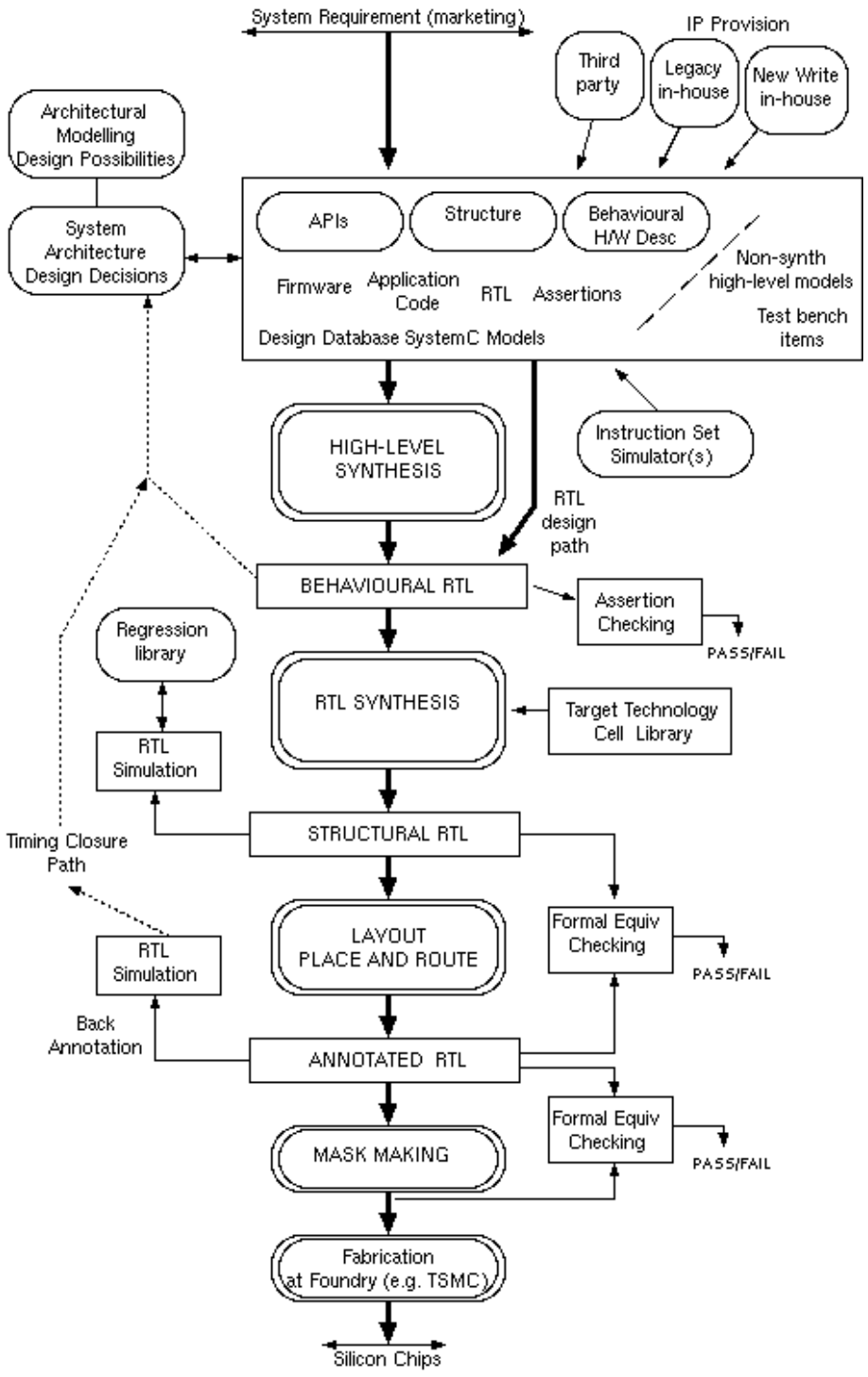


Figure 1.2: Design and Manufacturing Flow for SoC.

LG 2 — Review/Revision of Verilog RTL

2.1 RTL Summary View of Variant Forms.

From the point of view of this course, Verilog and VHDL are completely equivalent as register transfer languages (RTLs). Both support simulation and synthesis with nearly-identical paradigms. Of course, each has its proponent's.

Synthesisable Verilog constructs fall into these classes:

- **1. Structural RTL** enables an hierarchic component tree to be instantiated and supports wiring (a netlist) between components.
- **2a. Unordered, continuous assignments** describe combinational logic using a rich set of integer operators, including all those found in software languages such as C++ and Java.
- **2b. Synchronous, unordered RTL** supports the same set of complex RHS expressions as 2a.
- **3. Synthesisable behavioural RTL** uses a thread to describe behaviour where a thread may write a variable more than once. A thread is introduced with the '`always` ' keyword.

However, standards for synthesisable RTL greatly restrict the allowable patterns of execution: they do not allow a thread to leave the module where it was defined, they do not allow a variable to be written by more than one thread and they can restrict the amount of event control (i.e. waiting for clock edges) that the thread performs.

The remainder of the language contains the so-called 'non-synthesisable' constructs.

All the time values in the RTL are ignored for synthesis and zero-delay components are synthesisable. For them also to be simulatable in a deterministic way the simulator core implements the **delta cycle** mechanism.

One can argue that anything written in RTL that describes deterministic and finite-state behaviour ought to be synthesisable. However, this is not what the community wanted in the past: they wanted a simple set of rules for generating hardware from RTL so that engineers could retain good control over circuit structures from what they wrote in the RTL.

Today, one might argue that the designer/programmer should not be forced into such low-level expression or into the excessively-parallel thought patterns that follow on. Certainly it is good that programmers are forced to express designs in ways that can be parallelised, but the tool chain perhaps should have much more control over the details of allocation of events to clock cycles and the state encoding.

RTL synthesis tools are not normally expected to re-time a design, or alter the amount of state or state encodings. Newer languages and flows (such as Bluespec and Kiwi) still encourage the user to express a design in parallel terms, yet provide easier to use constructs with the expectation that detailed timing and encoding might be chosen by the tool.

2.2 Structural Verilog

Level 1/3: Structural Verilog : Structural, Heirarchic, Netlist

```
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;
```

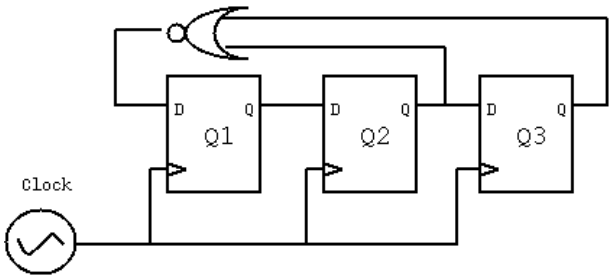


Figure 2.1: The circuit described by our structural example (a divide-by-five, synchronous counter).

Just a netlist. There are no assignment statements that transfer data between registers in structural RTL (but it's still a form or RTL).

2.3 Structure Flattening

Figure 2.2 shows structural RTL before and after flattening as well as a circuit diagram showing the component boundaries.

2.4 2a/3: Continuous Assignment.

```
input signed [31:0] c, d, e;
output signed [31:0] a;

assign a = (g) ? 33 : b * c;

assign b = d + e;
```

- Order of continuous assignments is un-important,
- Loop free, otherwise: parasitic level-sensitive latches are formed (e.g. RS latch),
- Right hand side's may range over rich operators (e.g. mux `?:` and multiply `*`),
- Bit inserts to vectors are allowed on left-hand sides (but not array writes).

```
assign   d[31:1] = e[30:0];
assign   d[0] = 0;
```

(*Sorry: this slide missing from printed notes*).

Heirarchic Netlist

```
module MOD1(b, a);
  input a; output b;
  wire c;
  INV inv1(c, a);
  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.

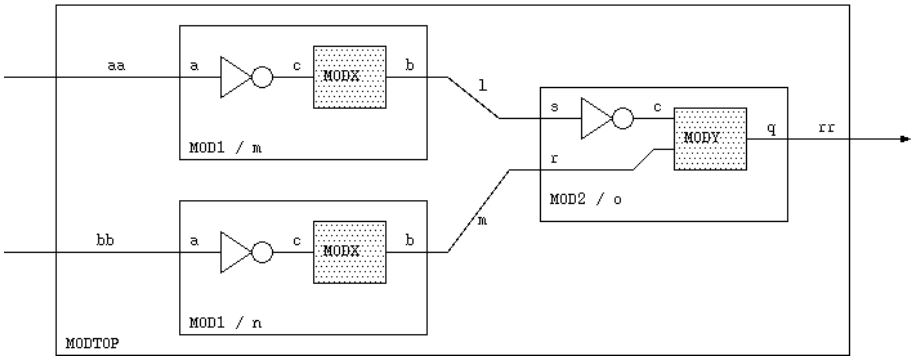


Figure 2.2: Example RTL fragment, before and after flattening.

2.5 2b/3: Pure RTL : unordered register transfers.

```
module CTR16(ck, din, o);

  input ck, din;
  output o;

  reg [3:0] count, oldcount;

  // Add a four bit decimal value of one to count
  always @(posedge ck) begin
      count <= count + 1;
      if (din) oldcount <= count;
      end

 // Note ^ is exclusive-or operator
 assign o = count[3] ^ count[1];

endmodule
```

Registers are assigned in clock domains (one shown). Each register assignment appears in exactly one clock domain. RTL synthesis does not generate special hardware for clock domain crossing (described later).

If we do not assign a register, it retains its old value. In other words, the behavioural 'if' statement is converted to the following **pure RTL** form that clearly uses a multiplexor.

```
oldcount <= (din) ? count : oldcount;
```

Order of pure RT' assignments does not matter.

Note: combinational logic (continuous assign) has no clock domain.

2.6 Elementary Examples

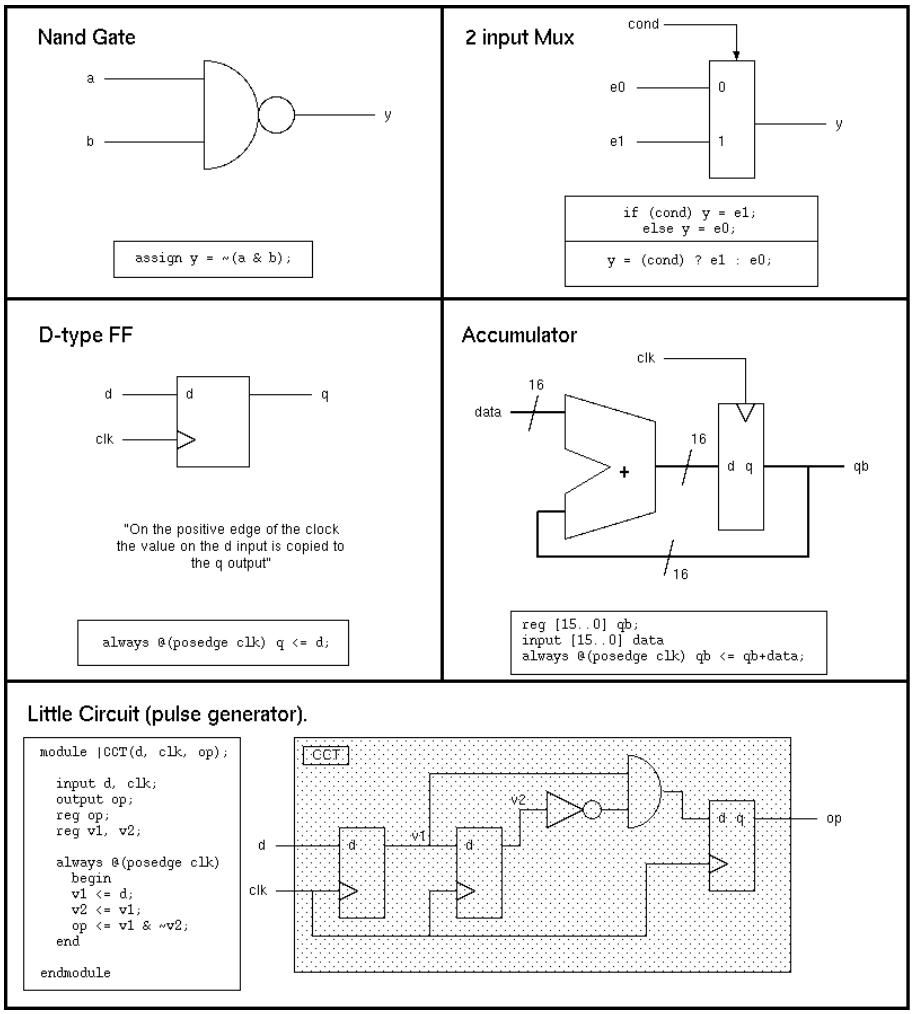


Figure 2.3: Elementary Synthesisable Verilog Constructs

Figure 2.3 shows synthesisable Verilog fragments as well as the circuits typically generated.

2.7 3/3: Behavioural RTL

Behavioural RTL resembles software.

A behavioural thread assigns to variables, makes reference to variables already updated and can re-assign new values.

Unlike 'pure RTL', the order of the statements has an effect.

For example, the following behavioural code

```
        if (k) x = y;
        z = !x;
```

can be compiled down to the following, unordered **'pure RTL'**.

```
        x <= (k) ? y: x;
        z <= !((k) ? y: x);
```

Not all behavioural RTL is defined to '**synthesisable**' (see later, and also synthesis algorithm in additional material).

2.8 Simulation And Synthesis.

An RTL program can be used both for simulation and synthesis.

Simulation: uses event-driven simulation (EDS). When using zero-delay models, we use the compute/commit paradigm where the EDS kernel is augmented to support delta cycles.

Synthesis: involves converting to a parallel form with one right-hand-side expression per variable. Then converting each expression to a logic tree, preferably taking into account sub-expression sharing and user speed/power/area requirements.

Simulation uses a top-level test bench module with no inputs.

Synthesis runs are made using points lower in the hierarchy as roots. We should certainly leave out the test-bench wrapper when synthesising and we typically want to synthesise each major component separately.

Synthesisable code uses synthesisable subset!

2.9 SRTL abstract syntax

Abstract syntax for a synthesisable RTL (Verilog/VHDL) without provision for delays:

Expressions:

```
datatype ex_t =
    Num of int
  | Net of string
  | Inv of ex_t
  | Query of ex_t * ex_t * ex_t
  | Diadic of diop_t * ex_t * ex_t
  | Subscript of ex_t * ex_t
```

Imperative commands (might also include a 'case' statement) but no loops.

```
datatype cmd_t =
    Assign of ex_t * ex_t
  | If1 of ex_t * cmd_t
  | If2 of ex_t * cmd_t * cmd_t
  | Block of cmd_t list
```

Our top level will be an unordered list of the following sentences:

```
datatype s_t =
   Sequential of edge_t * ex_t * cmd_t
|  Combinational of ex_t * ex_t
```

The abstract syntax tree for synthesisable RTL supports a rich set of expression operators but just the assignment and branching commands (no loops). (Loops in synthesisable VHDL and Verilog are restricted to so-called structural generation statements that are fully unwound by the compiler front end and so have no data-dependent exit conditions).

2.10 Behavioural - 'Non-Synthesisable' RTL

Not all RTL is officially synthesisable, as defined by language standards. However, commercial tools tend to support larger subsets than officially standardised.

RTL with event control in the body of a thread defines a statemachine. This is compilable by some tools.

This state machine requires a program counter (PC) register at runtime (implied):

```
input clk, din;
output req [3:0] q;

always begin
      q <= 1;
      @(posedge clk) q <= 2;
      if (din) @(posedge clk) q <= 3;
      q <= 4;
      end
```

How many bits of PC are needed ? Is conditional event control synthesisable ? Does the output 'q' ever take on the value 4 ?

Consider the dual-edge-triggered flip-flop in Figure 2.4.

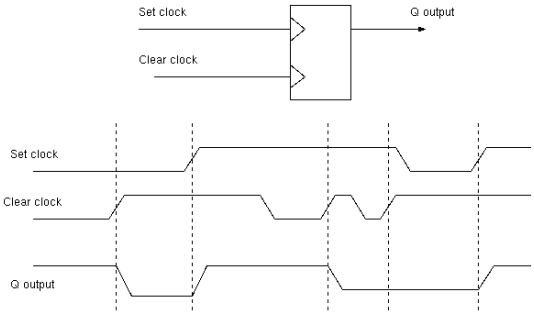


Figure 2.4: Schematic symbol and timing diagram for an edge-triggered RS flop.

```
reg q;
input set, clear;

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

Here a variable is updated by more than one thread. This component is commonly used in phase-locked loops. It can be modelled in Verilog, but is **not** supported for Verilog synthesis. A real implementation typically uses 12 NAND gates in a relatively complex arrangement of RS latches.

Another common source of non-synthesisable RTL code is testbenches. Testbenches commonly uses delays:

```
reg clk, reset;

initial begin clk=0; forever #50 clk = !clk; end

initial begin reset = 1; # 1025 reset = 0; end
```

Other non-synthesisable constructs found in RTL:

- Fork and Join
- Named events
- Variable update by more than one thread,
- File I/O and so on.

Finite state is all that should matter! In Kiwi we are finding at compile-time a fixed-point on the amount of state needed.

2.11 Structural Hazards.

We have a hazard when an operation cannot proceed because some information is not available or a resource is already in use.

- **Data hazard** - when an operand's address is not yet known or the operand has not arrived in time for use,
- **Control hazard** - when it is not yet clear whether an operation should be performed (cannot speculate on writes)
- **Structural hazard** - insufficient physical resources to do everything at once.

Resources that might present structural hazards are:

- Memories with insufficient ports,
- Memories with access latency (synchronous RAMs),
- Fetching from DRAM,
- Pipelined operator implementations (e.g. Booth Multiplier or floating point unit),
- Anything non-fully pipelined (something that goes busy).

Operations that could potentially be done in parallel have to be done serially.

Examples:

- insufficient number ALUs for all of the operations to be schedulled in current clock tick.
- insufficient number of ports on a RAM/register file.

To overcome structural hazards, additional clock cycles and holding registers are typically needed.

A **Non-fully pipelined** component: is unable to start a new operation on every clock cycle.

Such components:

- Have a start input and a busy/ready output.
- Component goes busy for a constant or variable number of cycles.

Example: fixed point multipliers and dividers (see additional material).

Example: all floating point operations tend to be implemented with multi-cycle units.

2.12 Structural Hazards in RTL

A structural hazard in an RTL design can make it non synthesisable.

Consider the following expressions that make liberal use of array subscription and the multiplier operator:

```
q <= Foo[x] + Foo[y];
q <= Foo[Foo[v]];
q <= a*b + c*d;
```

Problems arising:

- The RAM or register file Foo might not have two read ports.
- Even with two ports, can it perform the double subscription in one clock cycle?
- The cost of providing two 'flash' multipliers for use in one clock cycle while they lie idle much of the rest of the time is likely not warranted.

A multiplier that operates combinationaly in less than one clock cycle is called a 'flash' multiplier and it uses quadratic silicon area.

In addition, the RAMs may be synchronous with fixed latency and the multiplication time might be data dependent. The multipliers might not be **fully-pipelined**.

A non-fully pipelined component cannot start a new operation on every clock cycle. Instead it has handshake wires that start it and inform the client logic when it is ready.

A **holding register** is commonly manually inserted to overcome a structural hazard.

```
always @(posedge clk) begin
   pc = !pc;
   if (!pc) holding <= Foo[x];
   if (pc) ans <= holding + Foo[y];
   end
```

For greater manual control we could express the design using a behavioural style if we are allowed to pause the thread.

```
always @(posedge clk) begin
   holding <= Foo[x];
   @(posedge clk) ;
   ans <= holding + Foo[y];
   end
```

In the future, better to use tools that automatically balance the load on structural components?

Non-examinable material.

The transform in the additional material shows a behavioural transform from sequential to parallel composition.

To overcome structural hazards, folding in the reverse direction is needed, trading space for time, introducing additional registers typically called (according to useage):

- holding registers,
- pipeline stages, or
- write-back registers.

2.12.1 Notation

In these examples, sequential composition (in successive clock cycles) is denoted with the double semicolon. There is a distributive law of lockstep composition:

```
(c1 || c2) ;; (c3 || c4)  === (c1; c3) || (c2 ;; c4)
```

A lockstep parallel composition is well formed if its arguments contain the same number of sequential steps.

2.12.2 Holding Registers

A holding register stores the supporting inputs to a time-folded expression. For example, the space-using form

```
v1 <= A[e1] + e2 || v2 <= A[e3] + e4 || other_work
```

may have a structural hazard if array 'A' only has one read port, but can be rewritten using time, assuming $v1$ occurs in $e3$, $e4$ and other_work, using the **holding register** for $v1$ called h_v1:

```
(v1 <= A[e1] + e2 || h_v1 <= v1)   ;;  (v2 <= A[e3'] + e4' || other_work')
```

where $e3'$ and $e4'$ and other_work' are the rewritten forms of those expressions to refer to the holding register instead of $v1$ directly.

Similar holding registers are needed for other left-hand side variables assigned in the first clause of the sequence. Where such a left-hand side is an array, a pair (h_s, h_v) of holding registers is needed for the subscript and the old value at that location and a functional array form is needed for the substitution (i.e. reads of the form $A[e]$ are replaced with $(e = h\_s)?h\_v : A[e]$). However, the read out of the old value typically causes a new structural hazard, so it is best to instead leave assignments to arrays to the second clause of the sequence.

2.12.3 Delay Padding

The **delay padding** operation must be applied when one (or more) of two (or more) RTL expressions executing in lockstep is/are time-folded, thereby extending its/their execution time. For instance, if

```
v1 <= v2+1 || v2 <= A[v1 * 3]
```

is naively timefolded to

```
v1 <= v2+1 || (t1 <= v1 * 3 ;; v2 <= A[t1])
```

the execution times of the left-hand and right-hand sides no longer match and the result is not well formed. Instead, the left-hand side of the parallel composition must be delay padded with an input holding register, as follows:

```
(t2 <= v2 ;; v1 <= t2+1) || (t1 <= v1 * 3 ;; v2 <= A[t1])
```

or it can be padded with an output **write-back** register, as follows:

```
(v1_wb <= v2+1 ;; v1 <= v1_wb) || t1 <= v1 * 3 ;; v2 <= A[t1]
```

For an assignment with a lot of supporting input, rather than having a large number of holding registers for each of its support, having a single write-back register for its output is generally better, but optimum load balancing of expensive structural resources can be the deciding factor.

2.12.4 Generalised Pipeline Transform

Rather than delaying the input or output to a function, the function can be divided at any intermediate point with the introduction of so-called pipeline registers. As well as covercomming structural hazards, this can greatly help with timing closure. For instance,

```
v1 <= A[e1 * e2] + A[e3 * e4]
```

should be rewritten to use only one read port on 'A' as

```
t1 <= A[e1 * e2] ;; v1 <= t1 + A[e3 * e4]
```

This also has the benefit that one multiplier can be re-used for both operations using multiplexors.

2.13 Folding, Retiming & Recoding

The **time/space fold** and **unfold** operations trade execution time for silcon area. A given function can be computed with fewer clocks by 'unfolding' in the the time domain, typically by loop unwinding (and predication).

```
LOOPED (time) option:                | UNWOUND (space) option:
                                     |
for (i=0; i < 3*&i < limit; i++)  |  if (0 < limit) sum += data[0] * coef[j];
  sum += data[i] * coef[i+j];        |  if (1 < limit) sum += data[1] * coef[1+j];
                                     |  if (2 < limit) sum += data[2] * coef[2+j];
```

Sharing structural resources may require additional multiplexers and wiring: so not always worth it. A good design not only balances structural resource use between clock cycles, but also timing delays.

We can **retime** a design with and without changing its state encoding. Adding a pipeline stage can increase the amount of state without **recoding** existing state.

2.14 Critical Paths

Timing closure : Making the design meet its target clock rate.

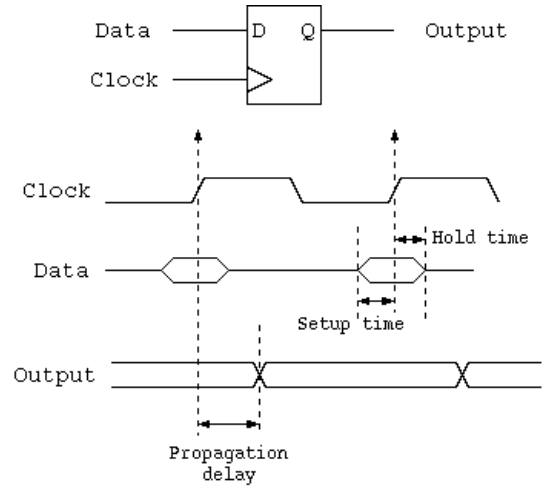


Figure 2.5: Illustrating the three, main timing parameters of a D-type flop.

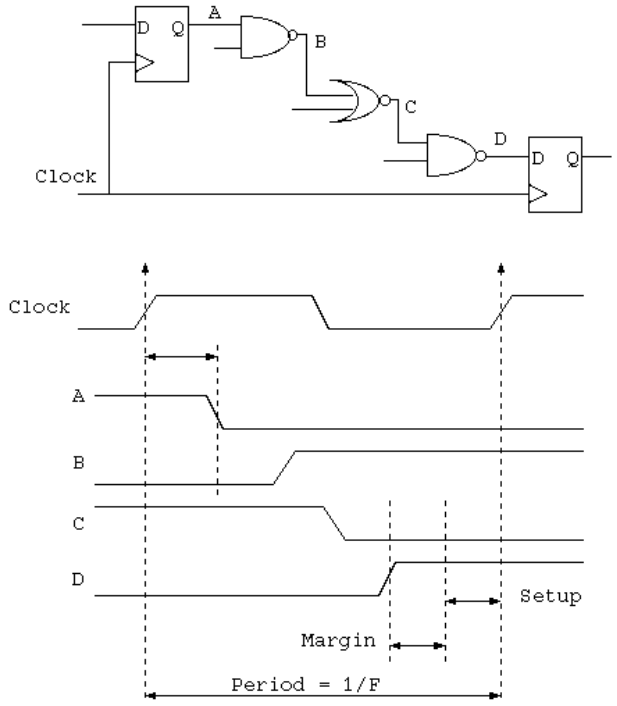


Figure 2.6: Typical nature of a critical path.

The maximum clock frequency of a synchronous machine is set by its critical path.

Introducing a pipeline stage increases latency but also the maximum clock frequency.

Flip-flop migration can affect critical paths:

```
a <= b + c;             b1 <=c; c1 <= c;
q <= (d) ? a:0;         q <= (d) ? b1+c1:0;
```

Alternatively, pushing the multiplexor back will require an earlier version of `d` that might not be available (e.g.

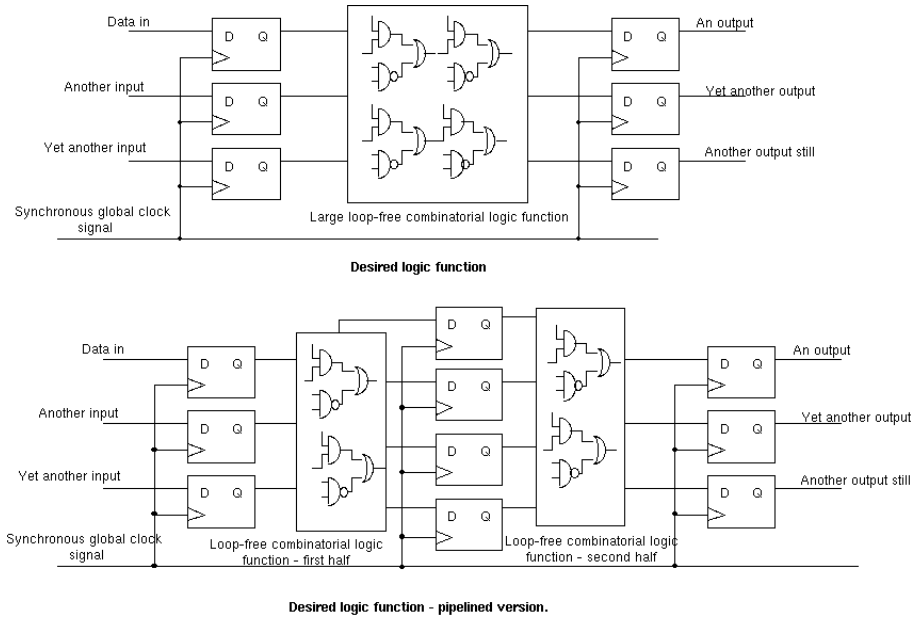


Figure 2.7: A circuit before and after insertion of an additional pipeline stage.

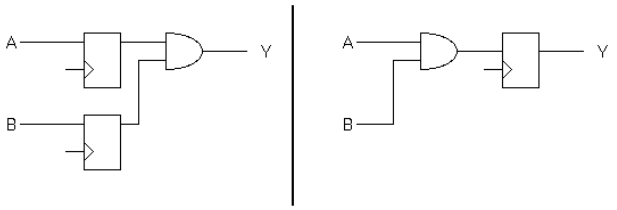


Figure 2.8: Two circuits of identical behaviour, but different state encoding.

if it were an external input).

Problems arising:

- Circuits containing loops (proper synchronous loops) cannot be pushed with a simple algorithm since the algorithm loops.
- External interfaces that do not use transactional handshakes (i.e. those without flow control) cannot tolerate automatic re-timing since the knowledge about when data is valid is not explicit.

but retiming can overcome structural hazards (e.g. the 'write back' cycle in RISC pipeline).

Other rewrites commonly use: automatically introduce one-hot and gray encoding, or invert for reset as preset.

Apart from using retiming of the design to overcome hazards, retiming is also useful for balancing logic between pipeline stages. Re-timing is therefore helpful for meeting **timing closure** which means ensuring the critical path of the design is short enough that the clock frequency can meet the envisioned target.

D-type migration is a transform that re-codes the state. For example, rather than forming the conjunction of (ANDing) two nets and registering the conjunct (feeding it through a D-type) one can instead form the conjunct of the registered inputs. A complete algebra can be used to annotate each signal with its offset from the designer's first-envisaged timing and D-types shunted around more or less at will. However, it is impossible to put a negative D-type on an input to the circuit (i.e. a gate that predicts the future), instead one must delay every other signal to compensate.

2.15 Transactional Handshaking in RTL

Legacy RTL's (Verilog and VHDL) do not provide synthesis of handshake circuits (but its one of the main innovations in Bluespec).

We'll use the word **transactional** for interfaces that support flow-control and can span between clock domains (one side may even be asynchronous).

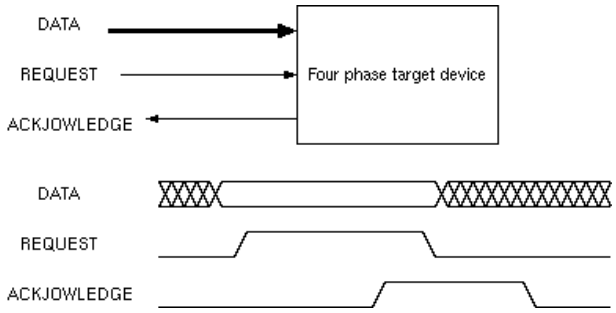


Figure 2.9: Timing diagram for an asynchronous, four phase handshake.

If tools are allowed to retime components, all interfaces between components must be transactional.

```
putbyte(char d)
{
  wait_until(!ack);
  data = d;
  settle();
  req = 1;
  wait_until(ack);
  req = 0;
}
```

```
char getbyte()
{
  wait_until(req);
  char r = data;
  ack = 1;
  wait_until(!req);
  ack = 0;
  return r;
}
```

2.16 RTL Compared with Software

Synthesisable RTL (SRTL) looks a lot like software at first glance, but we soon see many differences.

SRTL is statically allocated and defines a finite-state machine.

Threads do not leave their starting context and all communication is through shared variables that denote wires.

There are no thread synchronisation primitives, except to wait on a clock edge.

Each variable must be updated by at most one thread.

Software on the other hand uses far fewer threads: just where needed. The threads may pass from one module to another and thread blocking is used for flow control of the data.

SRTL requires the programmer to think in a massively parallel way and leaves no freedom for the execution platform to reschedule the design.

2.17 Further Synthesis Issues

There are many combinational circuits that have the same functionality. Synthesis tools can accept additional guiding metrics from the user, that affect

- Power consumption,
- Area use,
- Performance,
- Testability.

(The basic algorithm in the additional material does not consider any guiding metrics.)

Gate libraries have high and low drive power forms of most gates. The synthesis tool will chose the appropriate gate depending on the fanout and (estimated) net length during routing.

Use Quine/McCluskey or Espresso Algorithm for logic minimisation. Liberal use of the 'x' don't care designation in the source RTL allows the synthesis tool freedom to perform logic minimisation.

Can share sub-expressions or re-compute expressions locally. Reuse of sub-expressions is important for locally-derived results, but sending a 32 bit addition result more than one millimeter on the chip may use more power then recomputing it locally.

Retiming for structural hazard and timing closure avoidance is now becoming automated in modern design flows.

2.18 RTL Conclusion

RTL is not as expressive for algorithms or large structures as most software programming languages.

The concurrency model is that everything executes in lock-step. The programmer keeps all this concurrency in his/her mind.

Users must generate their own, bespoke handshaking and flow control between components.

Higher-level entry forms are ideally needed, perhaps schedulling within a thread at compile-time and between threads at run time ?

Reference algorithms for synthesis and simulation are included in the addtional material on the course web pages. (*Non-examinable.*)

LG 3 — SystemC Components

(SystemC using transactional-level modelling (TLM/ESL) is in a later section).

SystemC is a free library for C++ for hardware SoC modelling. Download from www.systemc.org

It can be used for detailed net-level modelling, but today its main uses are :

- Architectural exploration: Making a fast and quick, high-level model of a SoC to explore performance variation against various dimensions, such as bus width and cache memory size.
- Transactional level (TLM) models of systems, where handshaking protocols between components using hardware nets are replaced with subroutine calls between higher-level models of those components.
- Synthesis : some companies are generated their RTL from SystemC designs with a generation of so-called C-to-gates compilers.

SystemC was developed over the last ten years. There have been two major releases, 1.0 and 2.0. Also of importance is the recent release of the add-on TLM library, TLM 2.0.

Everything can be downloaded from www.systemc.org

SystemC includes (at least):

- A module system with inter-module channels: C++ class instances are instantiated in a hierarchy, following the circuit component structure, in the same way that RTL modules instantiate each other.
- An eventing and threading kernel that is non-preemptive and which allows user code inside components to run either in a trampoline style, returning the thread without blocking, or to keep the thread and hold state on a stack.
- Compute/commit signals as well as other forms of channel for connecting components together. The compute/commit signals are needed in a zero-delay model of hardware to avoid 'shoot-thru': i.e the scenario where one flip-flop in a clock domain changes its output before another has processed the previous value.
- A library of fixed-precision integers. Hardware typically uses all sorts of different width busses and counters that wrap accordingly. SystemC provides classes of signed and unsigned variables of any width that behave in the same way. For instance the user can define an `sc_int` of five bits and put it inside a signal. The provided library includes overloads of all the standard arithmetic and logic operators to operate on these types.
- Plotting output functions that enable waveforms to be captured to a file and viewed with a program such as `gtkwave`.
- A separate transactional modelling library: TLM 1.0 provided separate blocking and non-blocking interfaces prototypes that a user could follow and in TLM 2.0 these are rolled together into 'convenience sockets' that can convert between the two forms. See LG 4.

Problem: hardware engineers are not C++ experts but they can be faced with complex or advanced C++ error messages when they misuse the library.

Benefit: General-purpose behavioural C code, including application code and device drivers, can all be modelled in a common language.

```
SC_MODULE(mycounter)
{
   sc_in  < bool       > clk, reset;
   sc_out < sc_int<10> > myout;

   void m()
   {
      myout =  (reset) ? 0: (myout.read()+1); // Use .read() since channel contains a signal.
   }

   SC_CTOR(mycounter)
     { SC_METHOD(m);
       sensitive << clk.pos();
     }

}
```

SystemC enables a user class to be defined using the the SC_MODULE macro.

Modules inherit various attributes appropriate for an hierarchic hardware design including an instance name, a type name and channel binding capability..

The `sensitive` construct registers a callback with the EDS kernel that says when the code inside the module should be run.

A nasty feature of SystemC is the need to use the .read() method when reading a signal.

3.1 SystemC Structural Netlist

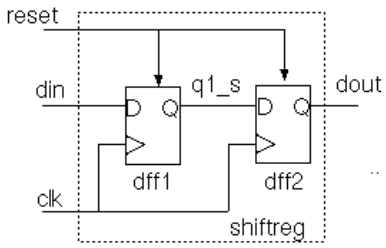


Figure 3.1: Schematic diagram for the first test example.

```
SC_MODULE(shiftreg)  // Two-bit shift register
{   sc_in  < bool >  clk, reset, din;
    sc_out < bool >  dout;

    sc_signal < bool > q1_s;
    dff dff1, dff2;      // Instantiate FFs

    SC_CTOR(shiftreg) : dff1("dff1"), dff2("dff2")
    {   dff1.clk(clk);
        dff1.reset(reset);
        dff1.d(din);
        dff1.q(q1_s);

        dff2.clk(clk);
        dff2.reset(reset);
        dff2.d(q1_s);
        dff2.q(dout);
    }
};
```

A SystemC channel provides general purpose interface between components.

The `sc_signal` channel should be used to obtain the compute/commit paradigm. This avoids non-determinacy from races in zero-delay models.

Other provided channels include FIFOs and semaphores.

Users can overload the channel class to implement channels with their own semantics if needed. A user-defined channel type can even contain other SystemC components but the importance of this is reduced when using the TLM libraries.

3.2 SystemC Channels and Signals

All primitive channels connect modules together.

Predefined channels include buffer, fifo, signal, mutex semaphore and clock.

RTL-style compute/commit is provided by the SystemC signals.

A signal is an abstract (templated) data type that has a current and next value.

Reads are of the current value. Writes are to the next value.

```
  int nv;
  sc_out    < int > data;
  sc_signal < int > mysig;
  ...
        nv += 1;
        data = nv;
        mysig = nv;
        printf("Before nv=%i, %i %i\n'', nv, data.read(), mysig.read());
        wait(10, SC_NS);
        printf("After  nv=%i, %i %i\n'', nv, data.read(), mysig.read());
  ...


Before nv=96, 95 95
After  nv=96, 96 96
```

Instantiable channel provides implementations of methods `read, write, update` and `value_changed`.

Signals have a current value and a next value. Assigns are to the next value whereas reads use the current value. The next value is copied to the current value in a commit cycle when there are no futher events of the current `tnow` on the event queue.

For faster system modelling, we do not want to enter EDS kernel for every change of every net: so is it possible to pass larger objects around, or even send threads between components, like S/W does ?

It is possible to put any datatype inside a signal and route that signal between components (provided the datatype can be checked for equality to see if current and nex are different and so on). So yes, using this approach, a higher-level model is possible, because a complete Ethernet frame or other large item can be delivered as a single event, rather than having to step though the cycle-by-cycle operation of a serial hardware implementation.

SystemC 2.0 enabled threads to be passed along the channels, breaking away from just using signals. This enables the transactional calls introduced in LG4 to be passed along structures that were previously EDS nets carrying logic values.

Can raise modelling abstraction level by passing an abstract datatype along channel.

```
sc_signal < bool > mywire; // Rather than a channel conveying just one bit,

struct capsule
{ int ts_int1, ts_int2;
  bool operator== (struct ts other)
  { return (ts_int1 == other.ts_int1)*&(ts_int2 == other.ts_int2); }
  ...
  ...  // Also some others
};

sc_signal < struct capsule > myast; // We can send two integers at once.
```

For many basic types, such as `bool, int, sc_int,` the required methods are provided in the library, but clearly not for user-defined types.

```
void mymethod()  { .... }
SC_METHOD(mymethod)
sensitive << mywire.pos();
```

When the scheduler blocks with no more events in the current time step, the pending new values are committed to the visible current values.

Future topic: TLM: wiring components together with methods instead of shared variables.

3.3 Threads and Methods

SystemC enables a user module to keep a thread and a stack but prefers, for efficiency reasons if user code runs on its own upcalls in a trampoline style.

- An `SC_THREAD` has a stack and is allowed to block.
- An `SC_METHOD` is just an upcall from the event kernel and must not block.

Comparing SC_THREADs with trampoline-style methods we can see the basis for two main programming TLM styles to be introduced later: blocking and non blocking.

The user code in an SC_MODULE is run either as an SC_THREAD or an SC_METHOD.

An SC_THREAD has a stack and is allowed to block.

An SC_METHOD is just an upcall from the event kernel and must not block.

Use SC_METHOD wherever possible, for efficiency.

Use SC_THREAD where important state must be retained in the program counter from one activation to the next or when asynchronous active behaviour is needed.

3.4 Example using an SC_THREAD

A data source that uses a net-level four-phase handshake:

```
SC_MODULE(mydata_generator)
{
  sc_out < int  > data;
  sc_out < bool > req;
  sc_in  < bool > ack;


  void myloop()
  {
    while(1)
    {
      data = data.read() + 1;
      wait(10, SC_NS);
      req = 1;
      do { wait(0, SC_NS); } while(!ack.read());
      req = 0;
      do { wait(0, SC_NS); } while(ack.read());
    }
  }
  SC_CTOR(mydata_generator)
  {
    SC_THREAD(myloop);
  }
}
```

3.5 Blocking and Eventing

A SystemC thread can block for a given amount of time using the `wait` function in the SystemC library (not the Posix namesake).

Wait on arbitrary boolean condition is harder to implement owing to the compiled nature of C++:

- C++ does not have a reflection API that enables a user's expression to be re-evaluated by the event kernel.
- Yet we still want a reasonably neat and efficient way of passing an uninterpreted function.
- Original solution: the delayed evaluation class.

```
    waituntil(mycount.delayed() > 5*&!reset.delayed());
```

Poor user had to just insert the **delayed** keyword where needed and then ignore it when reading the code.

It was too unwieldly, now removed: use the less-efficient

```
    do { wait(0, SC_NS); }  while(!((mycount > 5*&!reset)));
```

SystemC originally provided a type called lambda for building predicate functions that could be registered with the EDS kernel so that the kernel could interpret them when it wished and hence determine when a process was unblocked. However, they were cumbersome to use. They illustrate some of the issues typical to using a language like C++ for a purpose it was not designed for.

The recommended approach now is just to use spinlocks.

```
    waituntil(mycount.delayed() > 5*&!reset.delayed());
```

is replaced with the simple

```
    do { wait(0, SC_NS); }  while(!((mycount > 5*&!reset)));
```

3.6 Pre-TLM Software Style Channels

A suitable coding style for sending calls along the nets (prior to the TLM 2.0 standard):

```
class write_if: public sc_interface
{ public:
  virtual void write(char) = 0;
  virtual void reset() = 0;
};


class read_if: public sc_interface
{ public:
  virtual char read() = 0;
};


class fifodevice: sc_module("fifodevice"),
public write_if, public read_if, ...
{
  void write(char) { ... }
  void reset() { ... }

  ...
}
```

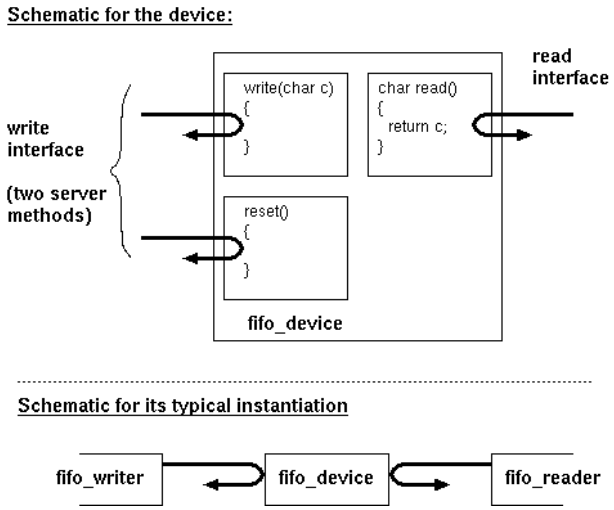


Figure 3.2: Thread-callable (TLM) interfaces exported by the FIFO example.

```
SC_MODULE("fifo_writer")
{
  sc_port<write_if> outputport;
  sc_in <bool> clk;
  void writer()
  {
    outputport.write(random());
  }

  SC_CTOR(fifo_writer} {
     SC_METHOD(writer);
     sensitive << clk.pos();
  }
}


   fifodevice myfifo("myfifo");
   fifo_writer mywriter("mywriter");

   mywriter.outputport(myfifo);
```

Here a thread passes between modules, but modules are plumbed in Hardware/EDS netlist structural style.

We can implement the object-oriented (OO) S/W concept of adding an interface to a component by inheritance.

Although there was a limited capability in SystemC 1.0 to pass threads along channels, and hence do subroutine calls along what look like wire, this was made much easier SystemC 2.0.

See the slide for full details, but the important thing to note is that the entry points in the interface class are implemented inside the fifo device and are bound, at a higher level, to the calls made by the writer device. This kind of plumbing of upcalls to entrypoints formed an essential basis for future transactional modelling styles.

However we soon run in to the well-known OO problem with multiple instances of an interface: not often needed for S/W but common enough in H/W designs.

3.7 SystemC Plotting and GUI

We can plot to industry standard VCD files and view with gtkwave (or modelsim).

```
sc_trace_file *tf = sc_create_vcd_trace_file("tracefile");

// Now call:
// sc_trace(tf, <traced variable>, <string>);

sc_signal<int> a;
float b;
sc_trace(trace_file, a, "MyA");
sc_trace(trace_file, b, "MyB");

sc_start(1000, SC_NS);  // Simulate for one microsecond
sc_close_vcd_trace_file(tr);
return 0;
```

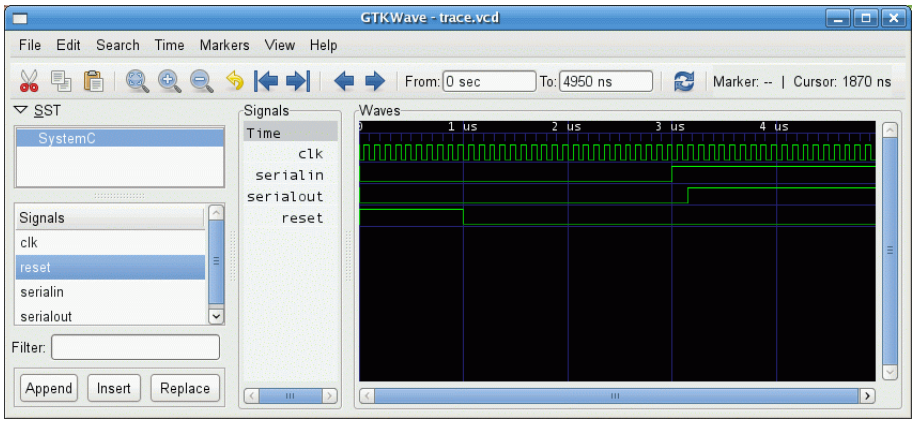


Figure 3.3: Waveform view plotted by gtkwave.

VCD can be viewed with **gtkwave** or in **modelsim**. There are various other commercial interactive viewer tools...

Try-it-yourself on PWF

LG 4 — Basic SoC Components

This section is a review of actual hardware components and bus structures found on chips. Schematics and illustrative RTL fragments are provided.

In the old-fashioned approach, we notice that the hand-crafted RTL used for the hardware implementation has no computerised connection with the firmware, device drivers or non-synthesisable models used for architectural exploration. Later we briefly look at how IP-XACT solves this.

We start with revision of basic computer architecture ...

4.1 Platform Chip (notes)

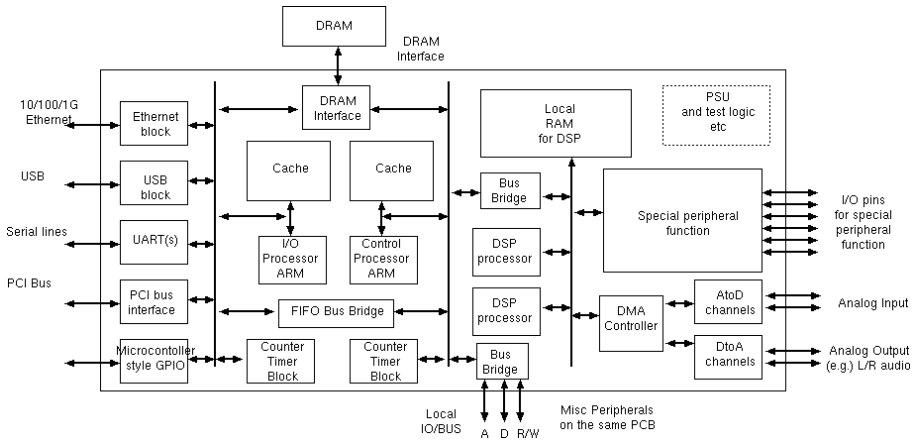


Figure 4.1: Platform Chip: Example Virata Helium 210

A platform chip is the modern equivalent of a microcontroller: it is a flexible chip that be programmed up to serve in a number of embedded applications. The set of components remains the same as for the microcontroller, but each has far more complexity: e.g. 32 bit prfocessor instead of 8. In addition, rather than putting a microcontroller on a PCB as the heart of a system, the whole system is placed on the same piece of silicon as the platform components. This gives us a system on a chip (SoC).

The example illustrated in figure 4.2 has two ARM processors and two DSP processors. Each ARM has a local cache and both store their programs and data in the same off-chip DRAM.

The left-hand-side ARM is used as an I/O processor and so is connected to a variety of standard peripherals. In any typical application, many of the peripherals will be unused and so held in a power down mode.

The right-hand-side ARM is used as the system controller. It can access all of the chip's resources over various bus bridges. It can access off-chip devices, such as an LCD display or keyboard via a general purpose A/D local bus.

The bus bridges map part of one processor's memory map into that of another so that cycles can be executed in the other's space, albeit with some delay and loss of performance. A FIFO bus bridge contains its own transaction queue of read or write operations awaiting completion.

The twin DSP devices run completely out of on-chip SRAM. Such SRAM may dominate the die area of the chip. If both are fetching instructions from the same port of the same RAM, then they had better be executing the same program in lock-step or else have some own local cache to avoid huge loss of performance in bus contention.

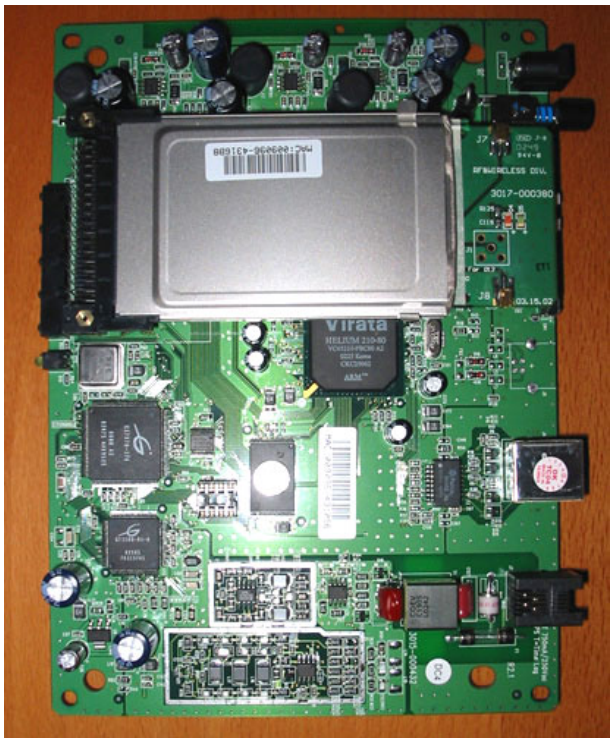


Figure 4.2: Helium chip built in to a modem.

The rest of the system is normally swept up onto the same piece of silicon and this is denoted with the 'special function peripheral.' This would be the one part of the design that varies from product to product. The same core set of components would be used for all sorts of different products, from iPODs, digital cameras or ADSL modems.

A platform chip is an SoC that is used in a number of products although chunks of it might be turned off in any one application: for example, the USB port might not be made available on a portable media player despite being on the core chip.

Generally devices must be allocated to busses with knowledge of the expected access and traffic patterns. Commonly there is one main bus master per bus. The bus master is the device that generates the address for the next data movement (read or write operation).

Busses are connected to bridges, but crossing a bridge has latency and also uses up bandwidth on both busses. So we should allocate devices to busses so that inter-bus traffic is minimised based on a priori knowledge of likely access patterns.

Lower-speed busses may go off chip.

DRAM is always an important component that used to be off chip. Today, some on-chip DRAM is being used in SoCs.

4.2 Microcomputer: Processor Bus Structure

This device is a bus master or *initiator* of bus transactions.

A basic microprocessor such as the original Intel 8008 device has a 16 bit address bus and an 8 bit data bus so can address 64 Kbytes of memory. It is an A16/D8 processor core. Internally it has instruction fetch, decode and execute logic.

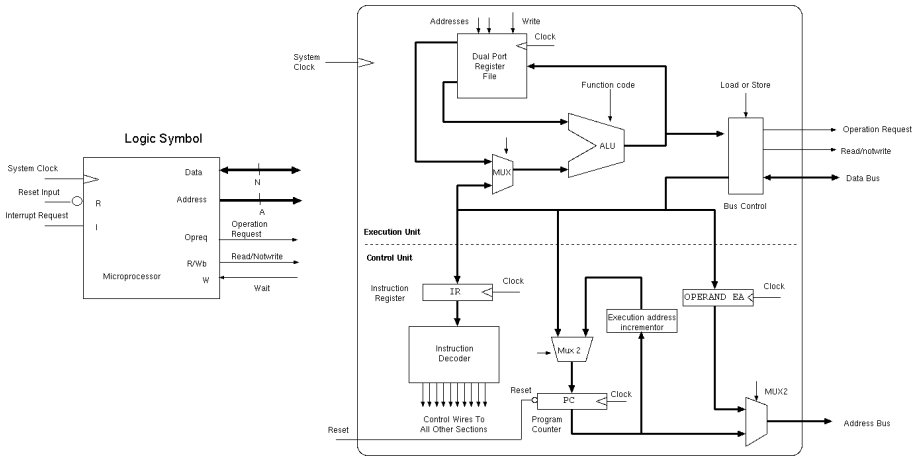


Figure 4.3: Schematic symbol and internal structure for a microprocessor (CPU).

The interrupt input makes it save its PC and load a fixed value instead: an external hardware event forces it to make a jump.

The high-order address bits are decoded to create chip enable signals for each of the connected peripherals, such as the RAM, ROM and UART.

As we shall see, perhaps the first SoCs, as such, were perhaps the microcontrollers. The Intel 8051 used in the mouse shipped with the first IBM PC is a good example. For the first time, RAM, ROM, Processor and I/O devices are all on one piece of silicon. We all now have many of these such devices : one in every card in our wallet or purse. Today's SoC are the same, just much more complex.

4.3 An historic D8/A16 Computer

Figure 4.4 shows the inter-chip wiring of a basic microcomputer (i.e. a computer based on a microprocessor).

4.4 Memory Address Mapping and Decode

```
-------  -----    ------------------------
Start    End      Resource
-------  -----    ------------------------
0000     03FF     EPROM
0400     3FFF     Unused images of EPROM
4000     7FFF     RAM
8000     BFFF     Unused
C000     C001     Registers in the UART
C002     FFFF     Unused images of the UART
-------  -----    ------------------------
```

The following RTL describes the required glue logic:

```
module address_decode(abus, rom_cs, ram_cs, uart_cs);
   input [15:14] abus;
   output rom_cs, ram_cs, uart_cs;


   assign rom_cs = (abus == 2'b00);  // 0x0000
   assign ram_cs = (abus == 2'b01);  // 0x4000
   assign uart_cs = !(abus == 2'b11);// 0xC000
endmodule
```

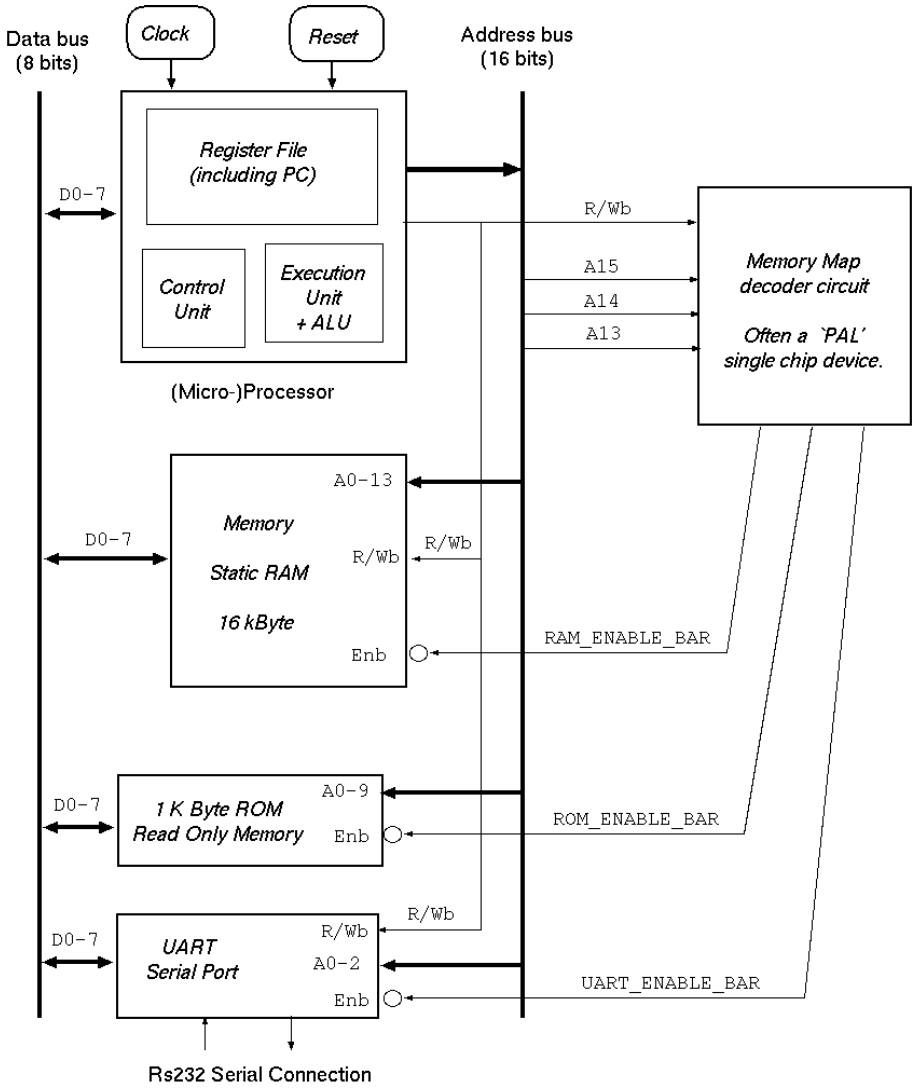


Figure 4.4: Early microcomputer structure, using tri-state busses.

The 64K memory map of the processor has been allocated to the three addressable resources as shown in the table.

The memory map must be allocated without overlapping the resources.

The ROM needs to be at address zero if this is the place the processor starts executing from when it is reset.

The memory map must be known at the time the code for the ROM is compiled. This requires agreement between the hardware and software engineers concerned.

In the early days, the memory map was written on a blackboard where both teams could see it.

For a modern SoC, there could be hundreds of items in the memory map. An XML representation called IP-XACT is being adopted by the industry and the glue logic may be generated automatically.

4.5 A Basic Micro-Controller

A microcontroller has all of the system parts on one piece of silicon. First introduced in 1989-85 (e.g. Intel 80C31).

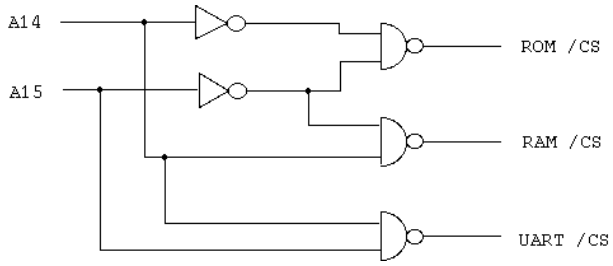


Figure 4.5: The 'glue logic' required to implement the memory map.

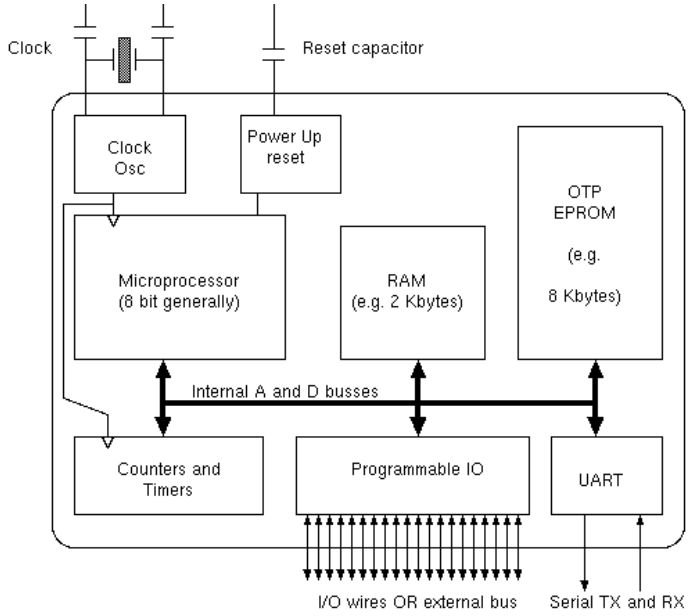


Figure 4.6: A typical single-chip microcomputer (micro-controller).

Such a micro-controller has an D8/A16 architecture and would be used in a door lock, mouse or smartcard.

4.6 Switch/LED Interfacing

Figure 4.7 shows an example of electronic wiring for switches and LEDs. Figure 4.8 shows an example of memory address decode and simple LED and switch interfacing for programmed I/O (PIO) to a microprocessor.

When the processor generates a read of the appropriate address, the tri-state buffer places the data from the switches on the data bus.

When the processor writes to the appropriate address, the broadside latch captures the data for display on the LEDs until the next write.

4.7 UART Device

The RS-232 serial port was widely used in the 20th century for character I/O devices (teletype, printer, dumb terminal).

A pair of simplex channels (output and input) make it full duplex.

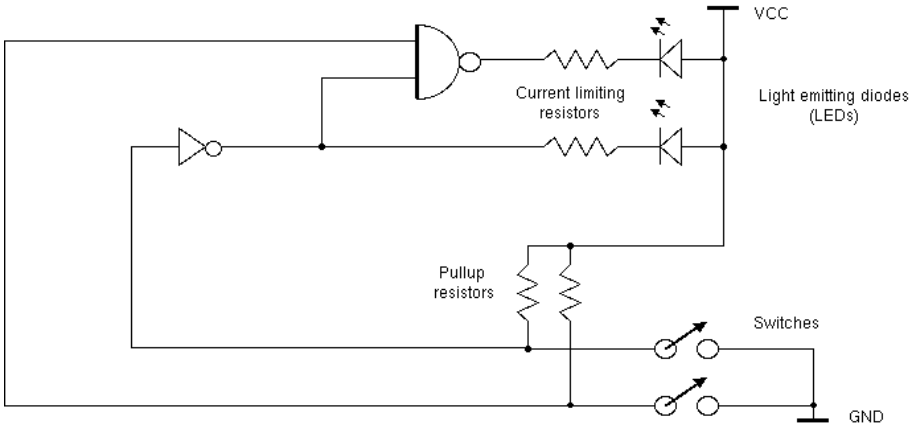


Figure 4.7: Connecting LEDs and switches to digital logic.

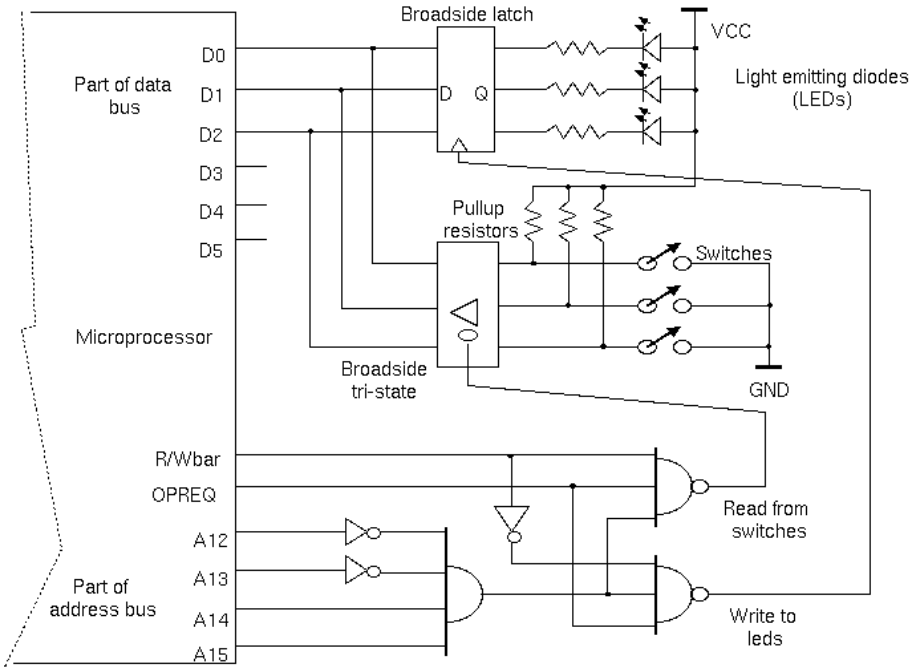


Figure 4.8: Connecting LEDs and switches for CPU programmed IO (PIO)

Additional wires are sometimes used for hardware flow control, or a software Xon/Xoff protcol can be used.

Baud rate and number of bits per words must be pre-agreeded.

4.8 Programmed I/O

Programmed Input and Output (PIO). Input and output operations are made by a program running on the processor. The program makes read or write operations to address the device as though it was memory.

Disadvantage: Inefficient - too much polling for general use. Interrupt driven I/O is more efficient.

Code to define the I/O locations in use by a simple UART device (universal asynchronous receiver/transmitter).

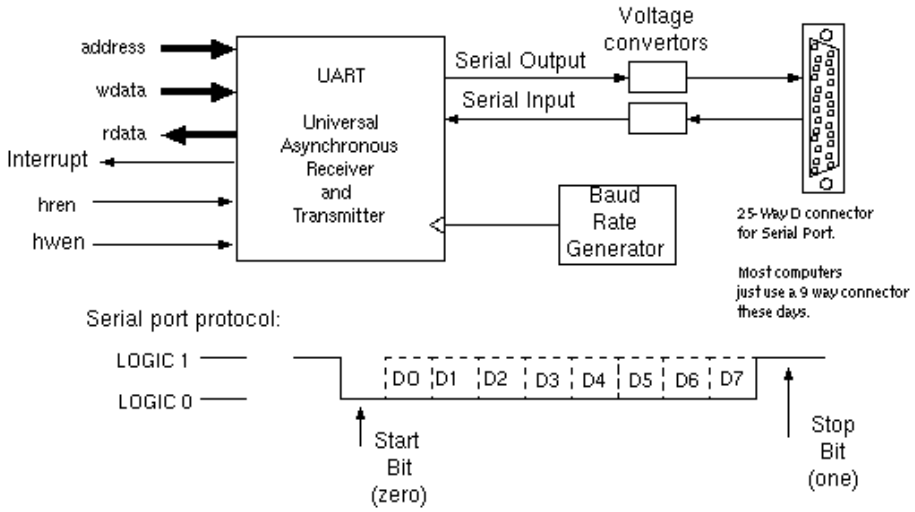


Figure 4.9: Typical Configuration of a Serial Port with UART

```
#define IO_BASE 0xFFFC1000 // or whatever


#define U_SEND    0x10
#define U_RECEIVE 0x14
#define U_CONTROL 0x18
#define U_STATUS  0x1C


#define  UART_SEND()  \
   (*((volatile char *)(IO_BASE+U_SEND)))
#define  UART_RECEIVE() \
  (*((volatile char *)(IO_BASE+U_RECEIVE)))
#define  UART_CONTROL() \
  (*((volatile char *)(IO_BASE+U_CONTROL)))
#define  UART_STATUS() \
  (*((volatile char *)(IO_BASE+U_STATUS)))


#define UART_STATUS_RX_EMPTY (0x80)
#define UART_STATUS_TX_EMPTY (0x40)


#define UART_CONTROL_RX_INT_ENABLE (0x20)
#define UART_CONTROL_TX_INT_ENABLE (0x10)
```

Code to make a blocking, polled read:

```
char uart_polled_read()
{
   while (UART_STATUS() &
      UART_STATUS_RX_EMPTY) continue;
   return UART_RECEIVE();
}
```

Code to make a blocking, polled write:

```
uart_polled_write(char d)
{
   while (!(UART_STATUS()&
     UART_STATUS_TX_EMPTY)) continue;
   UART_SEND() = d;
}
```

Interrupt driven receive routine:

```
char rx_buffer[256];
int rx_inptr, rx_outptr;

void uart_reset()
{
  rx_inptr = 0;
  rx_output = 0;
  UART_CONTROL() |= UART_CONTROL_RX_INT_ENABLE;
}


char uart_read()
{
  while (rx_inptr==rx_outptr) wait();
  char r = buffer[rx_outptr];
  rx_outptr = (rx_outptr + 1)&255;
  return r;
}

char uart_rx_isr()  // interrupt service routine
{
  while (1)
    {
      if (UART_STATUS()&UART_STATUS_RX_EMPTY) return;
      rx_buffer[rx_inptr] = UART_RECEIVE();
      rx_inptr = (rx_inptr + 1)&255;
    }
}

uart_write(char c)
{
  while (tx_inptr==tx_outptr) wait();
  buffer[tx_inptr] = c;
  tx_inptr = (tx_inptr + 1)&255;
  UART_CONTROL() |= UART_CONTROL_TX_INT_ENABLE;
}

char uart_tx_isr()
{
  while (tx_inptr != tx_outptr)
    {
      if (!(UART_STATUS()&UART_STATUS_TX_EMPTY)) return;
      UART_SEND() = tx_buffer[tx_outptr];
      tx_outptr = (tx_outptr + 1)&255;
    }
  UART_CONTROL() &=  UART_CONTROL_TX_INT_ENABLE;
}
```

The code fragment illustrates the complete set of four software routines needed to manage a pair of circular buffers for input and output to the UART. If the UART has a single interrupt output for both send and receive events, then two of the four routines are combined with a software dispatch between their bodies. Not shown above is that the ISR must be prefixed and postfixed with code that saves and restores the processor state (this is normally in assembler).

The input and output subroutines use spinlocks. The receiver spins until the empty flag in the status register goes away. Reading the data register makes the status register go empty again. The actual hardware device might have a receive FIFO, so instead of going empty, the next character from the FIFO would become available straightaway.

The output function is exactly the same in principle, except it spins while the device is still busy with any data written previously.

4.9 I/O Blocks, Common Interface Nets.

In the remainder of this section, we will consider a number of I/O blocks. All will be **targets**, most will also generate **interrupts** and some will also be **initiators**.

We use no bi-directional (tri-state) busses within our SoC: instead we use dedicated busses and multiplexor trees.

We use the following RTL net names:

- **addr** Internal adddress selection within a target,
- **hwen** Asserted during a target write,
- **hren** Asserted during a target read,
- **wdata** Input data to a target when written,
- **rdata** Output data when target is read,
- **interrupt** Asserted by target when wanting attention.

On an **initiator** the net directions will be reversed.

4.10 Interrupt Structure

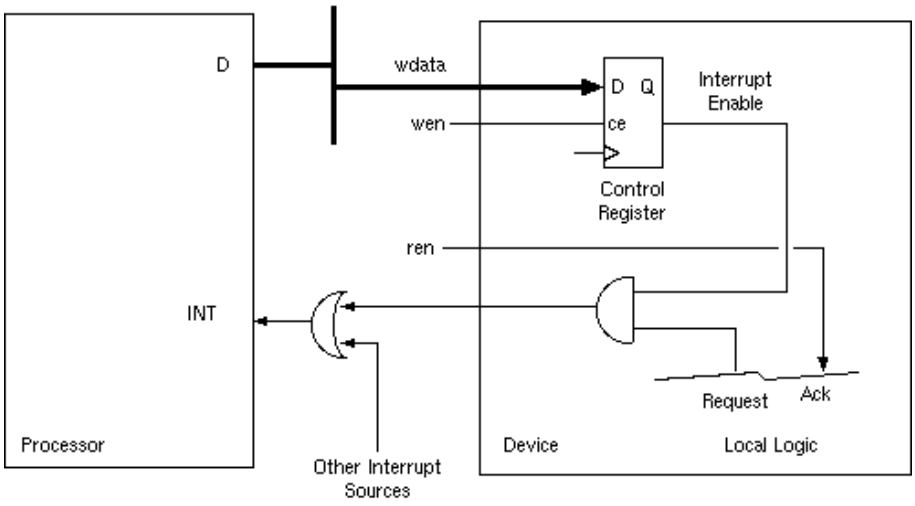


Figure 4.10: Interrupt generation: general structure within a device.

Nearly all devices have a master interrupt enable control flag that can be set and cleared by under programmed I/O by the controlling processor. Its output is just ANDed with the local interrupt source.

The programmed I/O uses the write enable (`wen`) signal to guard the transfer of data from the main data bus into the control register. A `ren` signal is used for reading.

The principal of programming is

- Receiving device: Keep interrupt enabled: device interrupts when data ready.
- Transmit device: Enable interrupt when S/W output queue non-empty: device interrupts when H/W output queue has space.

With only a single interrupt wire to the processor, all interrupt sources share it and the processor must poll around on each interrupt to find the device that needs attention.

Enchancement: a vectored interrupt makes the processor branch to a device-specific location.

Interrupts can also be associated with priorities, so that interrupts of a higher level than currently being run preempt.

4.11 RAM - on chip memory (Static RAM).

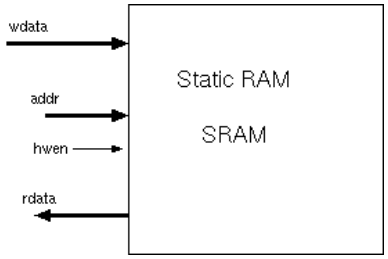


Figure 4.11: Static RAM with single port.

RAMs vary in their size and number of ports.

The 'hren' signal is not shown since the RAM is reading at all times when it is not reading. However, this wastes power, so it would be better to hold the address input stable when not needing to read the RAM.

Most RAMs in use on SoCs are synchronous with the data that is output being addressed the clock cycle before.

Most of today's SoC designs have more than fifty percent of their silicon area devoted to RAM.

Owing to RAM fabrication overheads, RAMs below a few hundred bits should typically be implemented as register files made of flip-flops. But larger RAMs have better density and power consumption than arrays of flip-flops. RAMs require special test logic.

Commonly, synchronous RAMs are used, requiring one clock cycle to read at any address. The same address can be written with fresh data during the same clock cycle, if desired.

RAMs for SoCs are normally supplied by companies such as Virage and Artizan. A 'ram compiler' tool is used for each RAM in the SoC. It reads in the user's size, shape, access time and port definitions and creates a suite of models.

High-density RAM (e.g. for L2 caches) may clock at half the main system clock rate and may need error correction logic to meet the system-wide reliability goal.

On-chip SRAM needs test mechanism:

- Can test with software running on embedded processor.
- Can have a special test mode, where address and data lines become directly controllable (JTAG or otherwise).
- Can use a built-in self test (BIST) wrapper that implements 0/F/5/A and walking ones typical tests.

Off-chip RAMS, such as DRAM and ZBT SRAM commonly used:

- Large area: would not be cost-effective on-chip.
- Specialised, proprietary or dense VLSI technology
- Non-volatile process (FLASH)
- Commodity part (DRAM, FLASH)

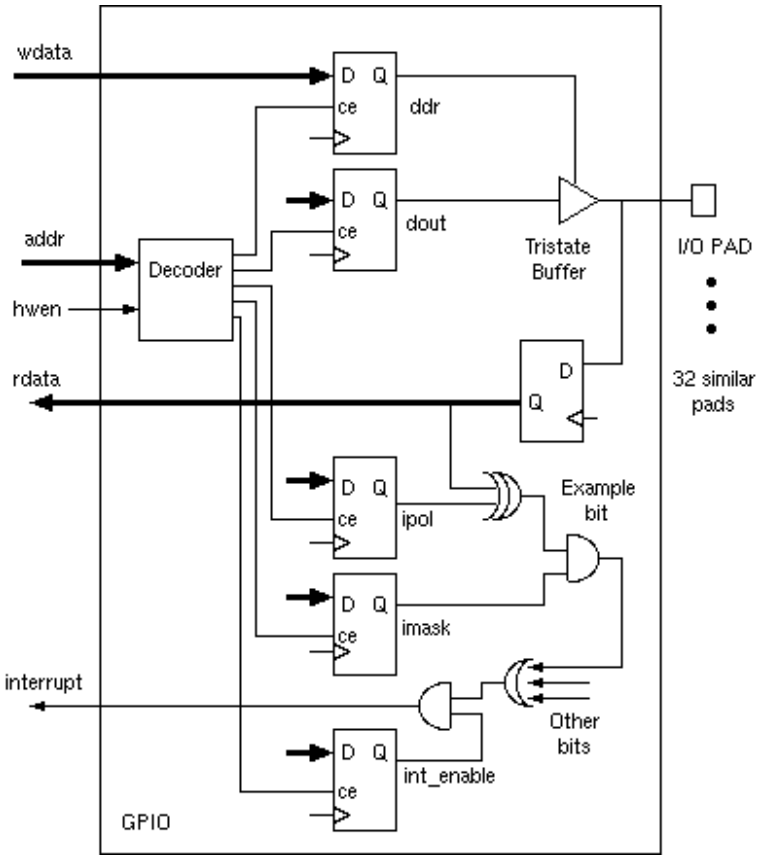


Figure 4.12: Internal implementation of general-purpose I/O pins.

4.12 GPIO - General Purpose Input/Output Pins

```
reg [31:0] ddr;    // Data direction reg
reg [31:0] pins_r; // regsiter'd pin data
reg [31:0] dout;   // output regsiter
reg [31:0] imask;  // interrupt mask
reg [31:0] ipol;   // interrupt polarities


reg int_enable, // output register

always @(posedge clk) begin
   pins_r <= pins;
   if (hwen*&addr==0) ddr <= wdata;
   if (hwen*&addr==4) dout <= wdata;
   if (hwen*&addr==8) imask <= wdata;
   if (hwen*&addr==12) ipol <= wdata;
   if (hwen*&addr==16) int_enable <= wdata[0];
   end


bufif b0  (pins[0], dout[0], ddr[0]); // Tri-state buffers.
.. // thirty others here
bufif b31 (pins[31], dout[31], ddr[31]);

wire int_pending = (|((din ^ ipol)&imask));
assign rdata = pins_r;
assign interrupt = int_pending*&int_enable;
```

Micro-controllers have a large number of GPIO pins.

Exercise: Show how to wire up a push button and write a device driver that counts how many times it is/was pressed.

The general structure of GPIO pins has not changed since the early days of the 6821 I/O controller. A number

of pins are provided that can either be input or output. A data direction register sets the direction on a per-pin basis. If an output, data comes from a data register. Interrupt polarity and masks are available on a per-pin basis for received events. A master interrupt enable mask is also provided.

The slide illustrates the schematic and the Verilog RTL for such a device. All of the registers are accessed by the host using programmed I/O.

4.13 A Keyboard Controller

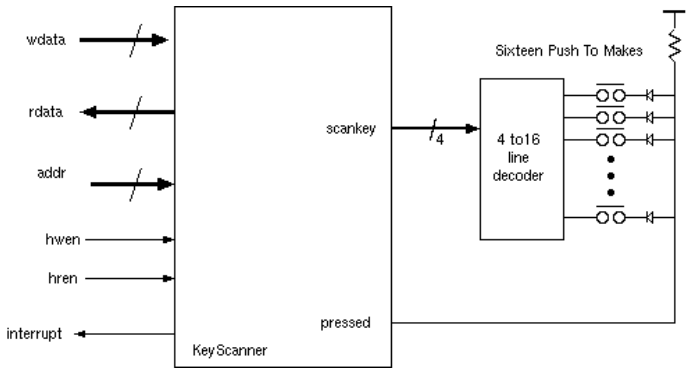


Figure 4.13: Keyboard Controller (without scan multiplexing).

```
output [3:0] scankey;
input pressed;
reg int_enable, pending;
reg [3:0] scankey, pkey;


always @(posedge clk) begin
   if (!pressed) pkey <= scankey;
   else scankey <= scankey + 1;

   if (hwen) int_enable <= wdata[0]
   pressed1 <= pressed;
   if (!pressed1*&pressed) pending <= 1;
   if (hren) pending <= 0;
   end
assign interrupt = pending*&int_enable;
assign rdata = { 28'b0, pkey };
```

The keyboard scanner scans each key until it finds one pressed. It then loads the scan code into the `pkey` register where the host finds it when it does a programmed I/O read.

The host will know to do a read when it gets an interrupt. The interrupt occurs when a key is pressed and is cleared when the host does a read `hren`.

In practice, one would not scan at the speed of the processor clock. One would scan more slowly and use extra register on asynchronous input `pressed` (see crossing clock domains). Or, typically, one might use a separate microcontroller to scan keyboard.

Note, a standard PC keyboard generates an output byte on press and release and implements a short FIFO internally.

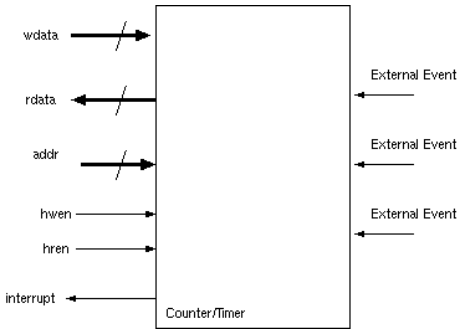


Figure 4.14: Counter/Timer Block, found in most computer systems.

4.14 Counters and Timers

```
reg [31:0] prescale, prescalar;
reg [31:0] counter, reload;
reg int_enable, ovf, int_pending;

always @(posedge clk) begin
   ovf <= (prescale == prescalar);
   prescale <= (ovf) ? 0: prescale+1;
   if (ovf) counter <= counter -1;
   if (counter == 0) begin
       int_pending <= 1;
       counter <= reload;
       end
   if (host_op) int_pending <= 0;
   end
wire host_op = hwen*&addr == 32;
assign interrupt = int_pending*&int_enable;
```

The counter/timer block is essentially a counter that counts internal clock pulses or external events and which interrupts the processor on a certain count value.

An automatic re-load register accommodates poor interrupt latency, so that the processor does not need to re-load the counter before the next event.

Timer (illustrated in the RTL) : counts pre-scaled system clock, but a counter has external inputs as shown on the schematic (e.g. car rev counter).

Four to eight, versatile, configurable counter/timers generally provided in one block.

All registers also configured as bus slave read/write resources for programmed I/O.

In this example, the interrupt is cleared by host programmed I/O (during `host_op`).

4.15 Video Controller: Framestore

```
reg [3:0] framestore[32767:0];
reg [7:0] hptr, vptr;
output reg [3:0] video;
output reg hsynch, vsynch;

always @(posedge clk) begin
     hptr <= (hsynch) ? 0: hptr + 1;
     hsynch <= (hptr == 230)
     if (hsynch) vptr <= (vsynch) ? 0: vptr + 1;
     vsynch <= (vptr == 110)
     video <= framestore[{vptr[6:0], hptr}];

     if (hwen) framestore[haddr]<= wdata[3:0];
     end
```

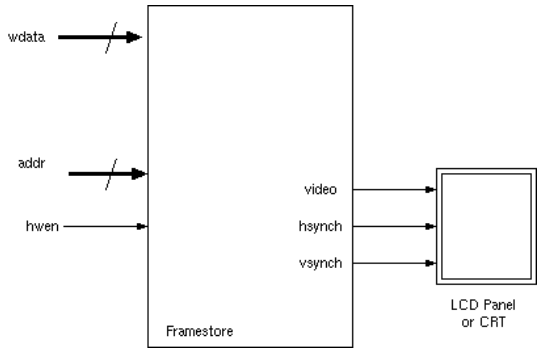


Figure 4.15: A basic, memory-mapped frame store.

The framestore reads out the contents of its frame buffer again and again. The memory is implemented in a Verilog array and this has two address ports. Another approach is to have a single address port and for the RAM to be simply 'stolen' from the output device when the host makes a write to it. This will cause noticable display artefacts if writes are at all frequent.

This framestore has fixed resolution, but real ones have programmable values read from registers instead of the fixed numbers 230 and 110.

The framestore in this example has its own local RAM. This reduces RAM bandwidth costs on the main RAM but uses more silicon area. A delicate trade off! A typical compromise, also used on audio and other DSP I/O, is to have a small staging RAM or FIFO in the actual device but to keep as much as possible in the main memory.

It's an output only device that never goes bust, so it generates no interrupts.

4.16 DMA Controller

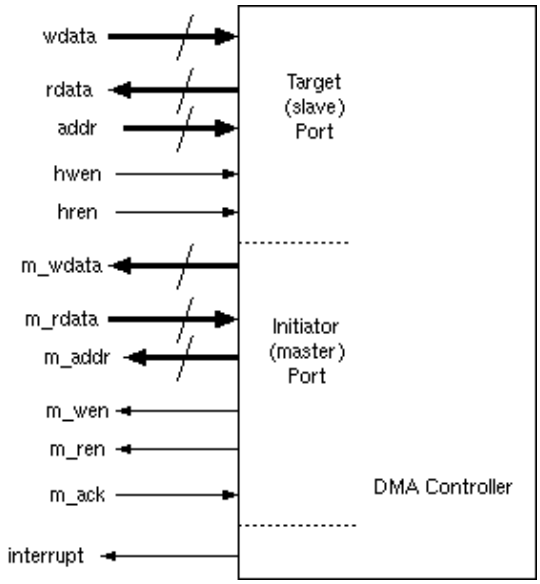


Figure 4.16: DMA controller, showing initiator and target connections.

This controller just block copies: may need to keep src and/or dest constant for device access.

DMA controllers may be built into devices: SoC bus master ports needed.

```
reg [31:0] count, src, dest, datareg;
reg int_enable, active, intt, rwbar;

always @(posedge clk) begin
   if (hwen*&addr==0) begin
       { int_enable, active } <= wdata[1:0];
       int  <= 0; rwbar <= 1;
       end

   if (hwen*&addr==4) count <= wdata;
   if (hwen*&addr==8) src <= wdata;
   if (hwen*&addr==12) dest <= wdata;

   if (active*&rwbar*&m_ack) begin
         datareg <= m_rdata;
         rwbar <= 0;
         src <= src + 4;
         end
   if (active*&!rwbar*&m_ack) begin
         rwbar <= 1;
         dest <= dest + 4;
         count <= count - 1;
         end
   if (count==1*&active*&!rwbar) begin
         active <= 0;
         intt <= 1;
         end
   end
assign m_wdata = datareg;
assign m_ren = active*&rwbar;
assign m_wen = active*&!rwbar;
assign m_addr = (rwbar) ? src:dest;
assign interrupt = intt*&int_enable;
```

The DMA controller is the first device we have seen that is a bus initiator as well as a bus target. It has two complete sets of bus connections. Note the direction reversal of all nets.

This controller just makes block copies from source to destination with the length being set in a third register. Finally, a status/control register controls interrupts and kicks of the procedure.

The RTL code for the controller is relatively straightforward, with much of it being dedicated to providing the target side programmed I/O access to each register.

The active RTL code that embodies the function of the DMA controller is contained in the two blocks qualified with the `active` net in their conjunct.

Typically, DMA controllers are multi-channel, being able to handle four or so concurrent transferrs. Many devices have their own DMA controllers built in, rather than relying on dedicated external controllers. However, this is not possible for devices connected the other side of bus bridges that do not allow mastering (initiating) in the reverse directions. This is a common-enough situation for perhipherals such as IDE disk drives.

Rather than using a DMA controller one can just use another processor. If the processor runs out of a small, local instruction RAM it will not impact on memory bus bandwidth with its fetches and it might not be that much larger in terms of silicon area.

An enhancement might be to keep either of the src or destination registers constant for streaming device access. For instance, to play audio out of a sound card, the destination address could be set to the programmed I/O address of the output register for audio samples and set not to increment.

For media with hard real-time characteristics, such as audio, video and modem signals, a small staging FIFO is likely to be needed in the device itself because the initiator port may experience latency when it is serviced. The DMA controller then initiates the next burst of its transfer when the local FIFO reaches a trigger depth.

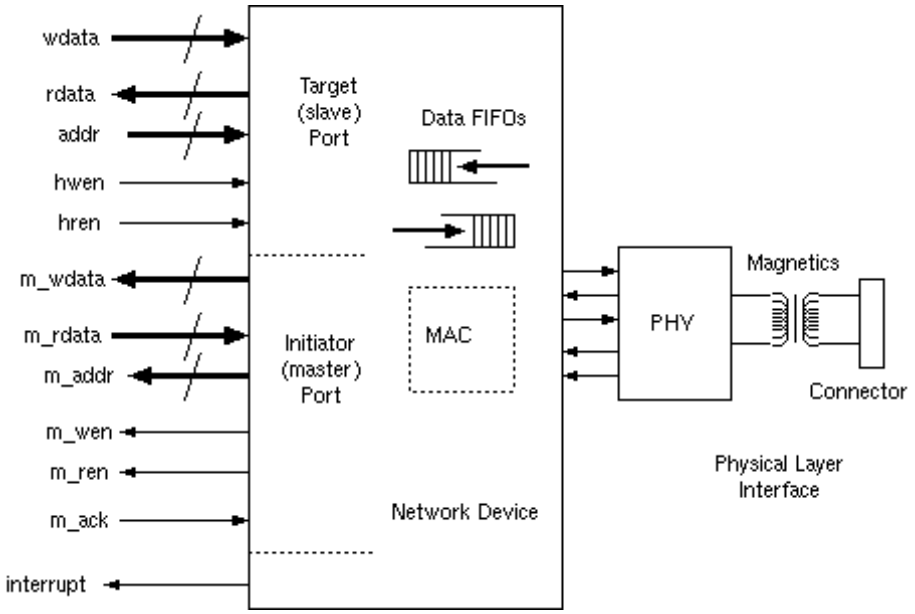


Figure 4.17: Connections to a DMA-capable network device.

4.17 Network Device

Network devices, such as Ethernet, USB, Firewire, 802.11 are rather similar to the audio/video/modem device with embedded DMA controller just discussed. For high throughput these devices should likely be bus masters or use a DMA channel.

DMA offloads work from the main processor, but, equally importantly, using DMA requires less staging RAM or FIFO in device. In the majority of cases, RAM is the dominant cost in terms of SoC area.

Another advantage of a shared RAM pool is **statistical multiplexing gain**. It is well known in queueing theory that having a monolithic server performs better than having a number of smaller servers that each are dedicated to one client. If the clients all share one server and arrive more or less at random, the system can be more efficient in terms of server utlisation. So it goes with RAM buffer allocation: having a central pool requires less overall RAM, on average, than having the RAM split around the various devices.

The DMA controller in a network device will might often have the ability to follow elaborate data structures set up by the host, linking and de-linking buffer pointers from a central pool in hardware.

4.18 Bus Bridge

The basic idea of the bus bridge is that bus operations slaved on one side are mastered on the other. The bridge need not be symmetric: speeds and data widths may be different on each side.

A bus bridge connects together two busses that are potentially able to operate indepently when traffic is not crossing. However, in some circumstances, especially when briding down to a slower bus, there may be no initiator on the other side, so it never actually operates independently.

The bridge need not support a flat address space: addresses seen on one side may be totally re-organised when viewed on the other side or unadressable. However, for debugging and test purposes, it is generally helpful to maintain a flat address space and to implement paths that are not likely to be used in normal operation.

A bus bridge might implement write posting using an internal FIFO. However it will generally block when reading. In another LG we cover networks on a chip that go further in that resepect.

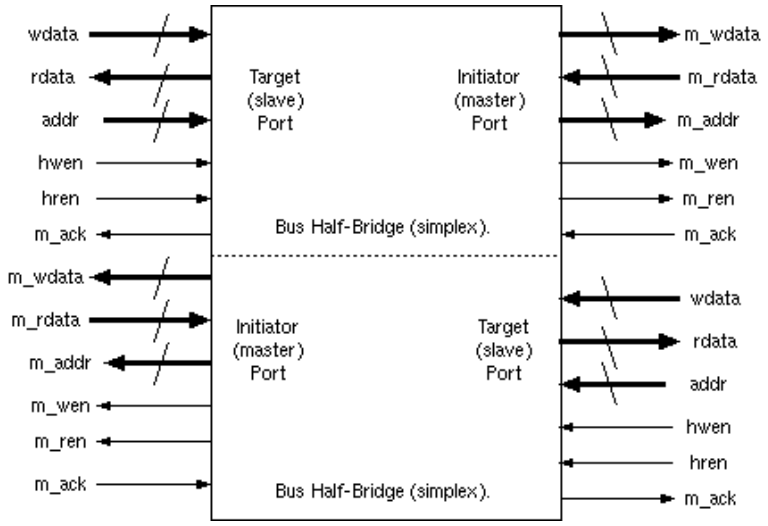


Figure 4.18: Bi-directional bus bridge, composed from a pair of back-to-back simplex bridges.

Note, the 'busses' on each side use multiplexors and not tri-states on a SoC. The multiplexors are different from bus bridges since they do not provide **spatial reuse** of bandwidth.

With a bus bridge, system bandwidth ranges from 1.0 to 2.0 bus bandwidth: inverse proportion to bridge crossing cycles.

4.19 Inter-core Interrupter (Doorbell/Mailbox)

A commonly-requried component for basic synchronisation between separate cores.

Offers a target (slave interface) for one ore more cores. Generates interrupts for the other cores (and self in symmetric situations).

One core write a register that assers and interrupt wire to another core.

Mailbox variant allows small data items to be written to a queue in the interrupter. These are read out by the (or any) core that is (or wants to) handle the interrupt.

4.20 Clock Frequency Multiplier PLL and Clock Tree

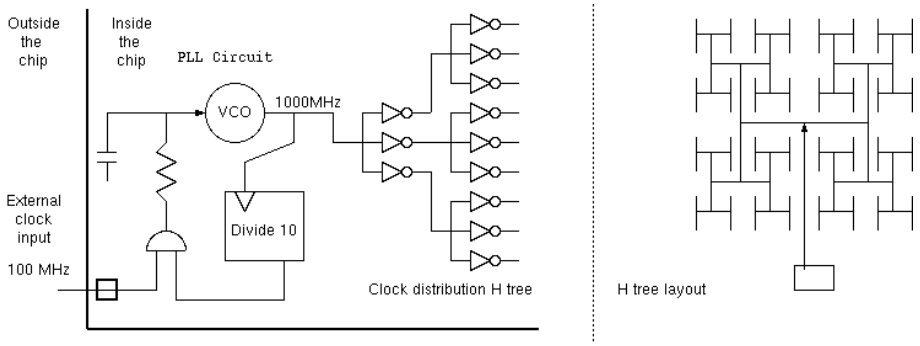


Figure 4.19: Clock multiplication using a PLL and distribution using an H-tree.

- Clock sourced from a lower-frequency external (quartz) reference.
- Multiplied up internally with a phase-locked loop.
- Dynamic frequency scaling (future topic): programmable PLL ratio.
- Skew in delivery is minimised using a balanced clock distribution tree.
- Physical layout: fractal of H's, ensuring equal wire lengths.
- Inverters are used to minimise pulse shrinkage (duty-cycle distortion).

The clock tree delivers a clock to all flops in a domain with sufficiently low skew to avoid shoot-thru. This is achieved by balancing wire lengths between the drivers.

The clock frequency is a multiple of the external reference which is commonly sourced from the piezo-effect of sound waves in a thin slice of quartz crystal.

This slide was not lectured in 2009. However, later on we talk about having a programmable clock frequency, so it's worth noting that the multiplication factor of 10 illustrated in the slide can be variable and programmed in some systems (e.g. laptops).

4.21 Clock Domain Crossing Bridge

Like a bus bridge, but different clocks on each side.

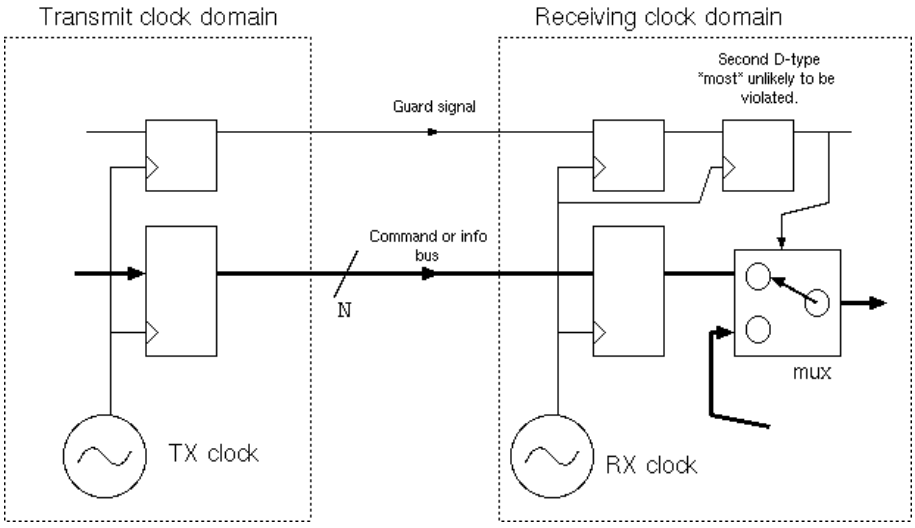


Figure 4.20: Sending parallel data between clock domains.

```
input clk;  // receiving domain clock

input [31..0] data;
input req;
output reg ack;


reg [31:0] captured_data;
reg r1, r2;
always @(posedge clk) begin
     r1 <= req;
     r2 <= r1;
     ack <= r2;
     if (r2*&!ack) captured_data <= data;
```

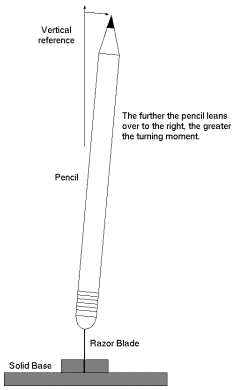


Figure 4.21: A pencil balancing on a razor blade can be metastable.

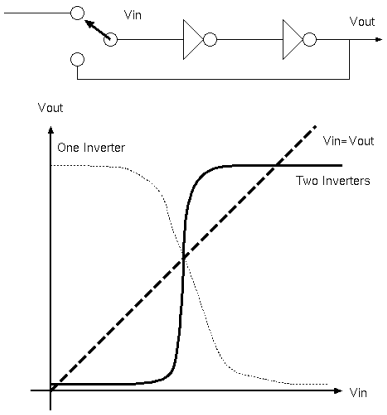


Figure 4.22: A bistable is two inverters connected in a ring, but there is also a metastable state.

A clock crossing bridge is like a bus bridge, but has different clock domains on each side.

A simplex clock domain crossing bridge carries information in only one direction. Duplex carries in both directions.

These are commonly needed when connecting to I/O devices that operate at independent speeds: for example, an Ethernet receiver sub-circuit works at the exact rate of the remote transmitter that is sending to it. Today's microprocessors also have separated clock domains for their cores viz their DRAM interfaces.

Basic idea:

- Have one signal that is a guard or qualifier signal for all the others going in that direction.
- Make sure all the other signals are settled in advance of guard.
- Pass the guard signal through two registers before using it (metastability).
- Use a wide bus (crossing operations less frequent).

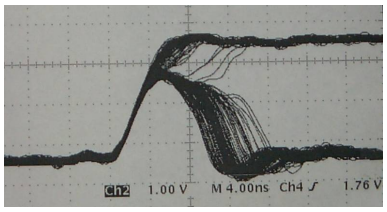


Figure 4.23: Metastable waveforms at the output of a D-type when set/hold times are sometimes violated.

The data signals can also suffer from metastability, but the multiplexer ensures that these metastable values never propagate into the main logic of the receiving domain.

Simplex: can never be sure about the precise delay.

We need a protocol with insertable/deletable padding symbols that have no semantic meaning. Or at a higher level, the protocol must have elidable idle states between transactions.

100 percent utilisation is impossible when crossing clock domains. The four-phase handshake limits utilisation to 50 percent (or 25 if registered at both sides) Other protocols can get arbitarily close to saturating one side or the other provided we know the maximum tolerance in the nominal clock rates.

Duplex: cannot reply on any precise timing relationship between the two directions. The protocol must rely on sequencing or expicit transaction tokens. In other words, we need a lot of temporal decoupling of requests and acks for crossing clock domains (and also network on chip later) (NB: This is not the same as the temporal decoupling in ESL modelling).

4.22 Arbiter

When multiple clients wish to share a resource, an arbiter is required. An arbiter decides which requester should be serviced.

Typical shared resources are busses, memories and multipliers.

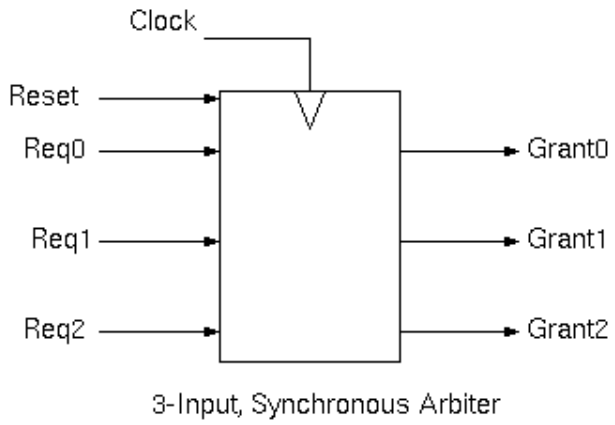


Figure 4.24: Typical Arbiter Schematic (three port/synchronous example)

There are two main arbitration disciplines:

- Static Priority - based on input port number (stateless).
- Round Robin - based on last user (held in internal state).

Another major policy variation is *preemptive* or not: can a granted resource be deassigned while the request is still asserted.

Arbiter circuits may be synchronous or asynchronous.

Complex disciplines involve dynamic priorites based on use history and to avoid starvation or might implement 'best matchings' between a number of requesters and a number of resources.

RTL implementation of synchronous, static priority arbiter with preemption.

```
module arbiter(clk, reset, reqs, grants);
  input clk, reset;
  input [2:0] reqs;
  output reg [2:0] grants;

  always @(posedge clk) if (reset) grants <= 0;
      else begin

         grants[0] <= reqs[0]; // Highest static priority
         grants[1] <= reqs[1]*&!(reqs[0]);
         grants[2] <= reqs[2]*&!(reqs[0] || reqs[1]);

      end
```

Exercise: Give the RTL code for a non-preemptive version of the 3-input arbiter..par

Exercise: Give the RTL code for a round-robin, non-preemptive version of the 3-input arbiter.

LG 5 — ESL: Electronic System Level Modelling

Aim 1: To Model whole SoC using real firmware and high-level behavioural models.

Aim 2: To allow seamless and successive replacement of model parts with low-level models/implementations when available and when interested in detail.

An ESL methodology provides:

- Tangible, lightweight **rapidly-generated prototype** of full SoC architecture.
- **Rapid Architectural Evaluation:** determine bus bandwidth and memory use for a candidate architecture. Easy to adjust major design parameters.
- **Algorithmic Accuracy:** Get real output from an early system, hosting the real application/firmware, possibly in real-time.
- **Timing information:** Get timing numbers for performance (accurate or loose timing).
- **Firmware development**: Integrate high-level behavioural models of major components with their device drivers.

Future topic: Embed assertions in the high-level models and use these assertions through to tape out (ABD).

Increasingly the industry wants to synthesise behavioural models to become the fabricated system, but today manual re-coding is the main way.

ESL is electronic system level modelling using recent developments whereby transactional models of hardware components can be called directly by device driver code without modelling processor cores or busses. This is especially useful for **architectural exploration** where a designer can rapidly experiment with different SoC configurations in terms of how many busses, what is connected to which bus and how wide the various busses and caches are.

We look at the motivational history of ESL, looking at how firmware and behavioural models were two types of IP divided from each other despite being generally in a common language: C++. We discuss architectural exploration using mixed-abstraction models.

ESL uses procedure calls between components in a S/W (software) coding style whereas traditional hardware modelling has used shared variables to model nets that connect the components. We need, at times, to convert between these S/W and H/W styles. We will need some **transactors**. These are small software entities that converts between the two modelling styles.

On the course web site, there is information on two sets of practical experiments.

- **Simple TLM 1 style:** To help investigate the key aspects of the transactional level modelling (TLM) methodology without using extensive libraries of any sort we use our own processor, the almost trivial nominalproc, and we cook our own transactional modelling library.

  This practical takes an instruction set simulator of a nominal processor and then sub-class it in two different ways: one to make a conventional net-level model and the other to make an ESL version. The nominal processor is wired up in various different example configurations, some using mixed-abstraction modelling.
- **TLM 2 style:** Using the industry standard TLM 2.0 library and the Open Cores OR1K processor. This is ultimately easier to use, but has a steeper learning curve.

We concentrate on the blocking TLM modelling style and then extend this with timing annotations to give performance estimates that vary in accuracy according to the quantum setting.

5.1 ESL Evolution: Starting point for SoC Modelling: Pre-ESL.

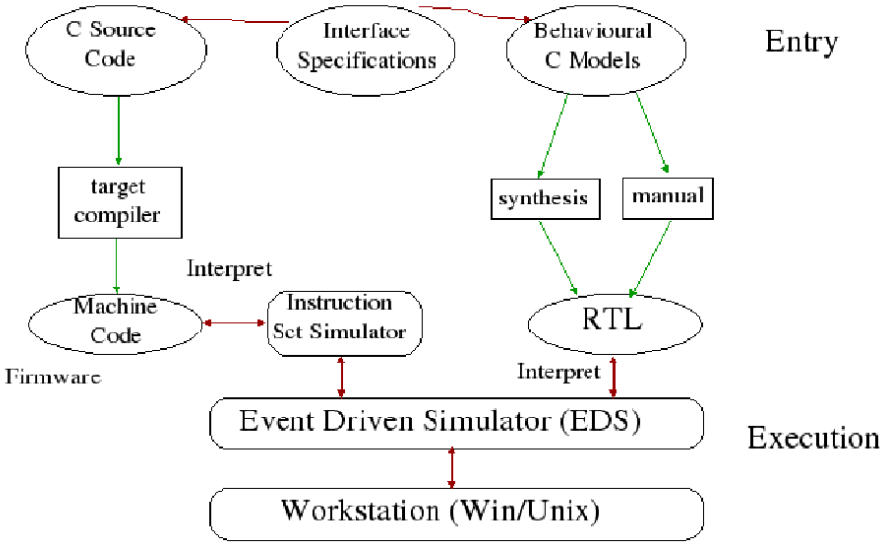


Figure 5.1: Typical SoC modelling configuration circa 1980.

In the pre-ESL era, an ISS interprets binary machine code and an RTL simulator simulates hardware devices.

Pre-ESL, the main tool was the EDS simulator. An instruction set simulator (ISS or *emulator*) coded in RTL or actual RTL processor model can be used to interpret the firmware, but this runs very slowly, perhaps a million times slower than real time, meaning operating system boot would take a day of modelling.

RTL Hardware devices compiled to cycle-accurate C models: increased performance.

High-level models integrated in the same environment: high performance.

Improvements shown in Figure 5.2 are to implement the ISS efficiently in C++ (perhaps using JIT techniques on the actual firmware) and to compile the RTL to C++ for faster simulation. The ISS can be cycle-accurate or just programmer-view accurate, where the hidden registers that overcome structural hazards or implement pipeline stages are not modelled.

Firmware cross-compiled for workstation processor: avoids ISS.

A further improvement shown in Figure 5.3 is to avoid the ISS entirely by cross compiling the firmware so that it can run naively on the modelling workstation. This can lead to a system that runs faster than real time, because the modelling workstation might be ten times faster than the true embedded processor core, depending on the activity ratios between hardware models and firmware.

ESL: Behavioural models used as simulation models.

Finally, as shown in Figure 5.4, for top speed, one has the scenario where the firmware device drivers communicates directly with high-level models of the hardware using procedure calling. This avoids modelling the processor bus operations entirely.

5.2 Using C Preprocessor to Adapt Firmware

We may need to recompile the hardware/software interface when compiling for TLM model as compared to the actual firmware.

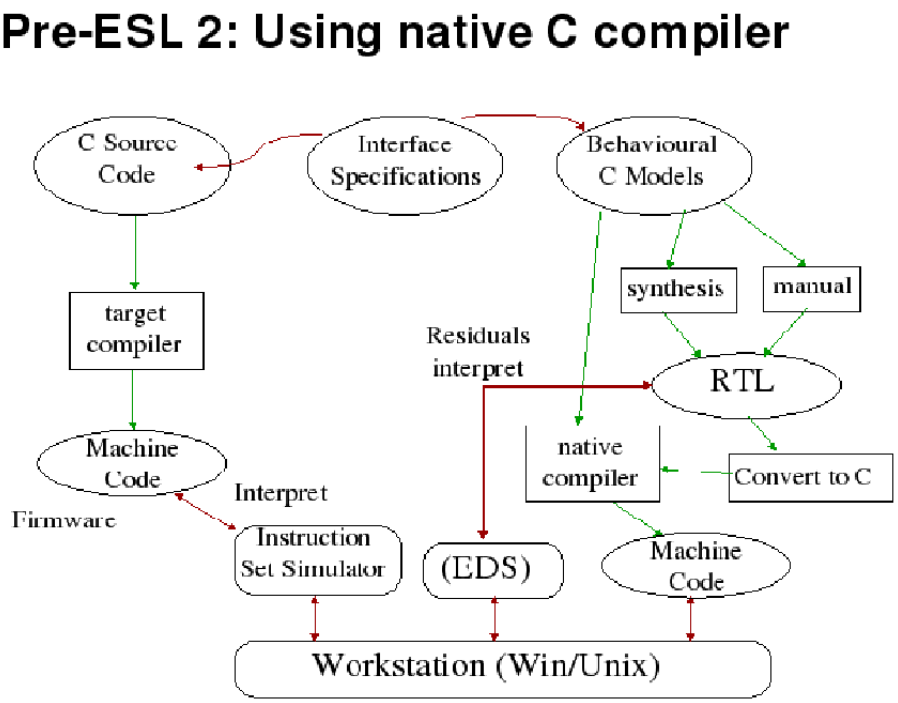


Figure 5.2: Modelling performance improved using an efficient ISS and native/C models of the custom hardware and peripherals.

Typically, differences are minor and can implemented in C preprocessor.

Device driver access to a UART device would be changed as follows:

```
#ifdef ACTUAL_FIRMWARE

  // For real system and lower-level models:
  // Store via processor bus to UART device register
  #define UART_WRITE(A, D)    (*(uart_base+A*4)) = (D)

#else

  // For high-level TLM modelling:
  // Make a direct subroutine call from the firmware to the UART model.
  #define UART_WRITE(A, D)    uart.write(A, D)

#endif
```

Alternatively, it is also possible to use the workstation VM system to trap calls from natively-compiled firmware to hardware: this requires the memory map of the embedded system to resemble that of the workstation.

5.3 Forms of ESL Model

ESL model: A model of a complete electronic system, including embedded processors.

A range of modelling styles made inter-operable using a suitable library (e.g. SystemC+TLM):

- Memory-accurate model
- Register-accurate model
- **Transactionally accurate model (TLM)**

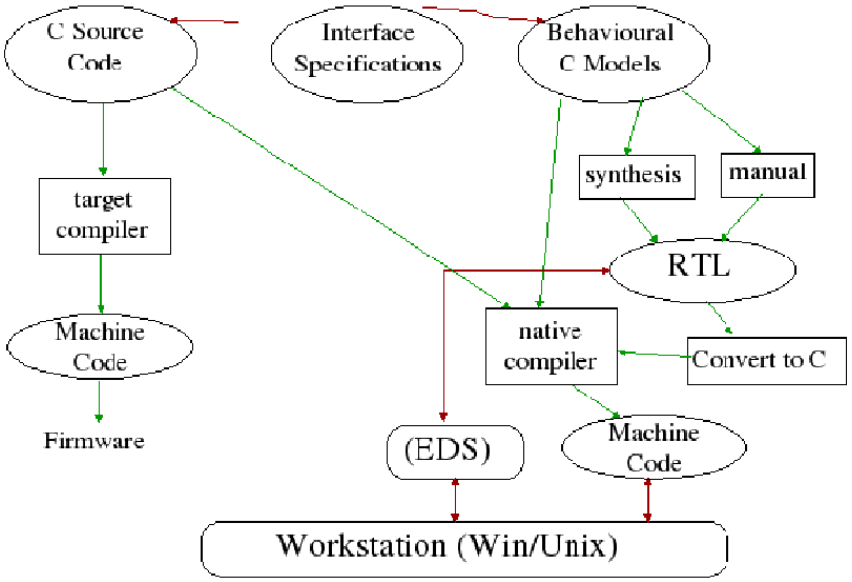


Figure 5.3: Avoiding the ISS by cross-compiling the firmware.

- Cycle-accurate model
- RTL-level model
- Net-level model
- Timed, net-level model.

In this section we concentrate on the TLM models. These can vary in their timing annotation style and their thread blocking:

- Approximately-timed, loosely-timed or untimed,
- Blocking or non-blocking styles.

5.4 Example H/W Protocol: 4/P Handshake

A commonly-used, asynchronous, simplex protocol, with flow control.

```
putbyte(char d)
{
  wait_until(!ack);
  data = d;
  settle();
  req = 1;
  wait_until(ack);
  req = 0;
}
```

```
char getbyte()
{
  wait_until(req);
  char r = data;
  ack = 1;
  wait_until(!req);
  ack = 0;
  return r;
}
```

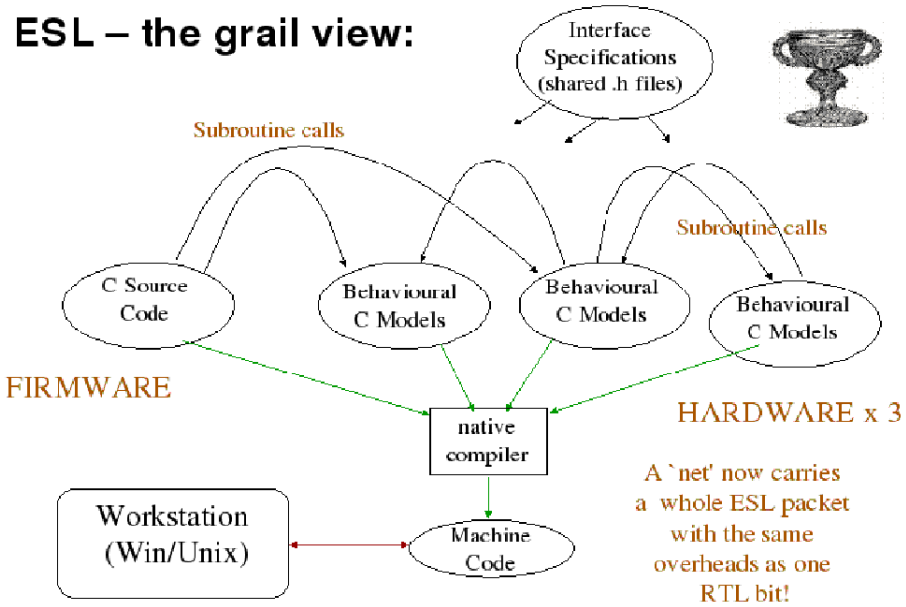


Figure 5.4: Using direct calls between firmware and behavioural models of the peripherhals.

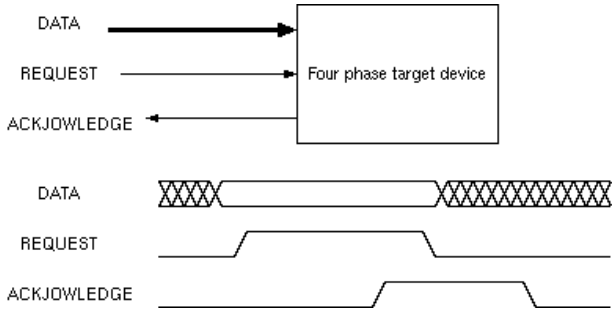


Figure 5.5: Four-phase handshake protocol, nets and protocol.

Example, untimed, **blocking transactor**: converts from transaction to pin-level modelling.

A common asynchronous protocol is the simple, four-phase handshake. Data is transferred once per complete handshake operation, which involves four phases.

Simple transactor code to convert the pin-level implementation so that software can call it can be implemented as follows:

```
putbyte(char d)
{
  wait_until(!ack);
  data = d;
  settle();
  req = 1;
  wait_until(ack);
  req = 0;
}


char getbyte()
{
  wait_until(req);        // NB: wait_until not in SystemC 2.0
  char r = data;          // See full code on web site, practical section.
  ack = 1;
  wait_until(!req);
  ack = 0;
  return r;
}
```

Other forms of transactor keep their own thread and make up-calls to user callbacks for each transaction.

5.5 Transactor Configurations

Four possible transactor's are envisonable for the 4/P handshake and in general.

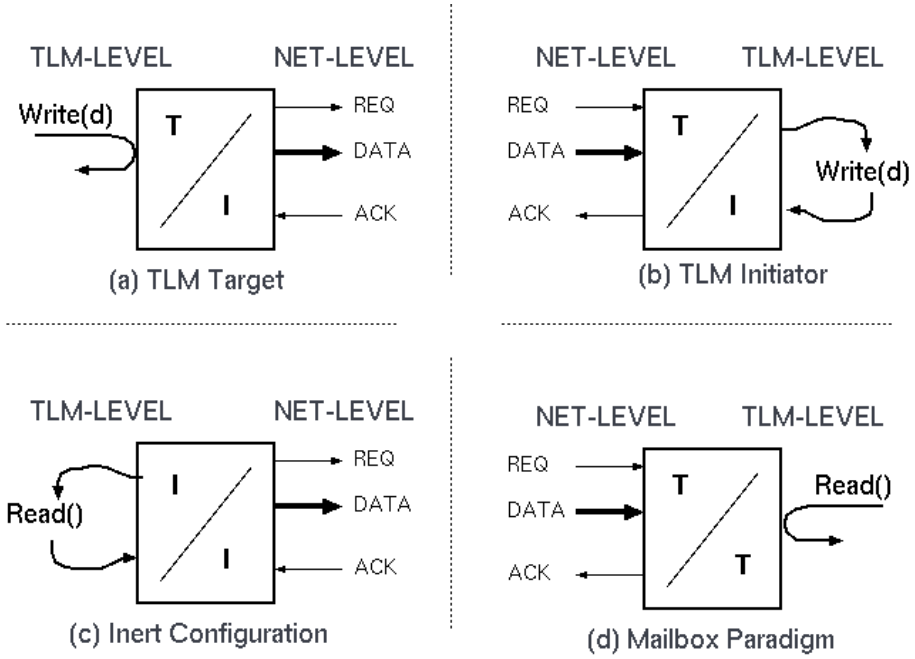


Figure 5.6: Possible configurations for simple transactors.

The form that is an initiator on boths sides will never do anything!

The form that is a target on boths sides follows the *mailbox* design pattern.

An (ESL) Electronic System Level *transactor* converts from a hardware to a software style of component representation. A hardware style uses shared variables to represent each net, whereas a software style uses callable methods and up-calls. Transactors are frequently required for busses and I/O ports. Fortunately, formal specifications of such busses and ports are becoming commonly available, so synthesising a transactor from the specification is a natural thing to do.

There are four forms of transactor for a given bus protocol. Either side may be an initiator or a target, giving four possibilities.

A transactor tends to have two ports, one being a net-level interface and the other with a thread-oriented interface defined by a number of method signatures. The thread-oriented interface may be a target that accepts calls from an external client/initiator or it may itself be an initiator that make calls to a remote client. The calls may typically be blocking to implement flow control.

The initiator of a net-level interface is the one that asserts the command signals that take the interface out of its starting or idle state. The initiator for an ESL/TLM interface is the side that makes a subroutine or method call and the target is the side that provides the entry point to be called.

5.6 What is a Transaction ?

In general, a *transaction* has atomicity, with commit or rollback. But in ESL the term means less than that. In ESL we might just mean that a thread from one component executes a method on another. However, the call and return of the thread normally achieve flow control and implement the atomic transfer of some datum, so the term remains relatively intact.

We can have blocking and non-blocking coding styles:

- **Blocking:** Hardware flow control signals implied by thread's call and return.
- **Non-blocking:** Success status returned immediately and caller must poll/retry as necessary.

In SystemC: blocking requires an SC_THREAD, whereas non-blocking can use an SC_METHOD.

Which is better: a matter of style ? Non-blocking enables finer-grained concurrency and closer to cycle-accurate timing results. (TLM 2.0 sockets will even automatically map between styles.)

5.7 Example of non-blocking coding styles:

(Have seen blocking style on earlier slide.)

Example: Non-blocking (untimed) transactor for the four-phase handshake.

```
bool putbyte_nb_start(char d)
{
  if (ack) return false;
  data = d;
  settle(); // A H/W delay for skew issues,
  // or a memory fence in S/W for
  // sequential consistency.
  req = 1;
  return true;
}

bool putbyte_nb_end(char d)
{
  if (!ack) return false;
  req = 0;
  return true;
}
```

```
bool getbyte_nb_start(char &r)
{
  if (!req) return false;
  r = data;
  ack = 1;
  return true;
}

bool getbyte_nb_end()
{
  if (req) return false;
  ack = 0;
  return true;
}
```

Both routines should be repeated by the client until returning true.

Four timing points may be of interest:

- first try start,

- succeed (last try) start,
- first try end,
- succeed (last try) end.

5.8 TLM in SystemC: TLM 1.0

Transactional-level modelling (TLM) 1.0 standard used conventional C++ concepts of multiple inheritance.

An SC_MODULE that implements an interface just inherits it.

The `sc_port` and `sc_export` constructs are used to wire TLM ports together.

Problem: no standardised structure for payloads.

Problem: no standardised timing annotation mechanism.

Problem: how to have multiple TLM ports on a component with same interface: e.g. a packet router.

5.9 SoC Component, TLM 1.0 Form Example.

Example: a one-channel DMA controller:

```
// Bus slave side, operand registers
uint32 src, dest, length;
bool busy, int_enable;

uint32 status() { return (busy << 31)
          | (int_enable << 30); }


uint32 slave_read(int a)
{
  return (a==0)? src: (a==4) ? dest:
   (a==8) ? (length) : status();
}
void slave_write(int a, uint32 d)
{
   if (a==0) src=d;
   else if (a==4) dest=d;
   else if (a==8) length = d;
   else if (a==12)
   { busy = d >> 31;
     int_enable = d >> 30; }
}
```

```
// Bus mastering side
while(1)
{
  waituntil(busy);
  while (length-- > 0)
    mem.write(dest++, mem.read(src++));
  busy = 0;
}
```

We would like to make interrupt output with an RTL-like continuous assignment:

```
 interrupt = int_enable&!busy;
```

But this will need a thread to run it

The OSCI TLM 1.0 standard used conventional C++ concepts of multiple inheritance. As illustrated earlier in the course, an SC_MODULE that implements an interface just inherits it.

SystemC 2.0 implemented an extension called `sc_export` that allows a parent module to inherit the interface of one of its children. This was a vital step needed in the common situation where the exporting module is not the top-level module of the component being wired-up.

However, TLM 1.0 had no standardised or recommended structure for payloads and no standardised timing annotation mechanisms.

There was also the problem of how to have multiple TLM ports on a component with same interface: e.g. a packet router.

The RTL coding style of the DMA controller shown earlier was a little bit hard to understand, but at least it was synthesisable.

On the other hand, the active component of an ESL version of such a DMA controller is simply:

```
// Bus mastering side
while(1)
{
  waituntil(busy);
  while (length-- > 0)
    mem.write(dest++, mem.read(src++));
  busy = 0;
}
```

In other words, it looks just like a simple block copy in C++. (Full details are in the practical class examples.)

However, we can see that that memory operations are likely to get well out of synchronisation with the real system since this copying loop just goes as fast as it can without worrying about the speed of the real hardware. It is just governed by the number of cycles the read and write calls block for, which could be none. The whole block copy might occur in zero simulation time!

This sort of modelling is useful for exposing certain types of bugs in a design, but it does not give useful performance feedback.

5.10 TLM - Modelling Contention

When more than one client wants to use a resource at once we have contention.

Real queues are used in hardware, either in FIFO memories or by flow control applying back pressure on the source to stall it until the contended resource is available. An arbiter allocates a resource to one client at a time.

Contention can be modelled using real or virtual queues:

1. In a low-level model, the real queues are modelled in detail.
2. A TLM model may queue the transactions, thereby blocking the client's thread until the transaction can be served.
3. Alternatively, the transactions can be run straightaway and an estimated delay can be added to the client's delay account.

Delay estimates can be based on dynamic measurements of utilisation at the contention point, in terms of transactions per millisecond and a suitable formula, such as $1/(1-p)$ that models the queuing delay in terms of the utilisation.

We'll use the average cost if we are not separately modelling contention. Otherwise, use the minimum (base) cost and let the contention or queueing modelling mechanism estimate the remainder.

If we are running with cross-compiled firmware, we do not have accurate instruction stream, so must use bulk estimates, on basic-block basis or coarser.

The value 'p' is the utilisation in the range 0 to 1. From queuing theory, with random arrivals, the queuing delay goes to infinity using a $1/(1-p)$ shape curve as p approaches unity. For uniform arrival and service times, the queuing delay goes sharply to infinity at unity.

5.11 TLM - Analytical or Fluid Flow Delay Modelling

Q. If we do not model clock cycles or instruction counts, how can we work out response times and resource utilisations ?

A. Attach cost annotations and useage diagnostics to whatever behaviours are actually modelled.

Hence, we will use **fluid-flow** continuous approximation to discrete costs, such as bus cycles, cache misses and DRAM access patterns.

Modelled behaviours:

- **Memory transactions by processor**: annotate with average (minimum) cost.
- **Instructions by processor**: annotate with time used on an average instruction (or per instruction) basis.
- **Packets or flits** in network on chip.
- **Contention overheads:** can use geometric (1/(1-p)) approximations.
- **Clock-cycles:** where cycle-accurate sub-systems are included.

Clearly we must attach a timing delay annotation to each operation at some level of granularity. But who is responsible for taking note of the timing annotations ?

The SystemC EDS kernel maintains the current simulation time. The current simulation time is the time of the event on head of event queue, as shown in the Toy EDS part of these notes.

```
cout << ‘‘Time now is : ‘‘ << simcontext()->time_stamp() << ‘‘ \n’’;
```

A coarse schedulling granularity gives high-speed simulation: But how often must we re-enter the SystemC kernel ?

When using an ISS, how many instructions to model in one step ?

- **Cycle-accurate model:** Every clock tick,
- **Approximately-timed model:** we block the EDS kernel every time a timing delay is encountered,
- **Loosely-timed model:** we add up the delays encountered by a thread in a local variable until we really must block or we wish to resynchronise with other parts of the system.

- **Untimed model:** Free riding: The subsystem's load is ignored.

Loosely-timed modelling, also know as **temporally-decoupled modelling**, can be highly efficient.

We can mix modelling styles in one system, but threads that pass through untimed components will appear faster than real life. This is OK if they are not of interest and not on the critical path, thereby affecting the performance of other components.

5.12 SystemC Kernel Time Stamp

Let's use an EDS kernel with its `tnow` variable defined by the head of the event queue. This is our main reference time stamp, but let's try not to use the kernel very much, only entering it when inter-module communication is needed. This reduces context swap overhead (I know it's all in user space but it is a computed branch that does not get predicted) and we can run a large number of ISS instructions at one time, giving good use of the caches on the modelling workstation.

In SystemC, we can always print the current time with:

```
cout << ``Time now is : `` << simcontext()->time_stamp() << `` \n'';
```

5.13 TLM Modelling: Adding The Timing Annotations

The naive way to add approximate timing annotations is to block the SystemC kernel until the required time has elapsed.

```
sc_time clock_period  = sc_time(5, SC_NS);  // 200 MHz clock

int read(A)
{
  int r = 0;
  if (A < 0 or A >= SIZE) error(....);
  else r = MEM[A];
  wait(clock_period * 3);   // <-- Directly model memory access time: three cycles say.
  return r;
}
```

We can also measure resource utilisation in this way:

```
write(A, D)
{
   if (A > LIM) port1.write(A-LIM, D) else port0.write(A, D)
   opcount += 1;
   if (opcount == 100)
   {
      sc_time delta = sc_time_stamp() - last_measure_time;
      logging.log(100, delta);
      last_measure_time = sc_time_stamp();
      opcount = 0;
   }
}
```

In the above, we assume logging.log knows how many bus cycles per unit time can be handled and hence can compute and record the utlisation.

A more-flexible coding style is to pass a time accumulator variable called 'delay' around for various models to augment:

```
putbyte(char d, sc_time &delay)
{
  ...
  delay += sc_time(140, SC_NS);
}
```

The leading ampersand on delay is the C++ denotation for pass by reference. At any point, any component can execute a **resynch** by performing

```
sc_wait(delay);
delay = 0;
```

Performance will be reduced if there are frequent resynchs, but transaction ordering will be modelled correctly.

For **approximate** timing:

- In the non-blocking coding style, we can make a note of the simulator `tnow` in a local variable at the start of a transaction and not return success of the transaction (or a sub-part of it) until `tnow` has sufficiently advanced.
- In the blocking coding style we can block the caller's thread for the appropriate number of cycles before returning with the result.

For **loose** timing:

- In the blocking coding style, we can compare the delay time so far accumulated with a desired maximum (the *quantum* for that thread or component or system wide quantum) and if it is exceeded then do a resynch.
- In the non-blocking coding style we can do the same, but non-blocking with loose timing is not commonly used.

The loose timing allows transactions to execute out of order (see later slides).

5.14 Loose Timing and Temporal Decoupling

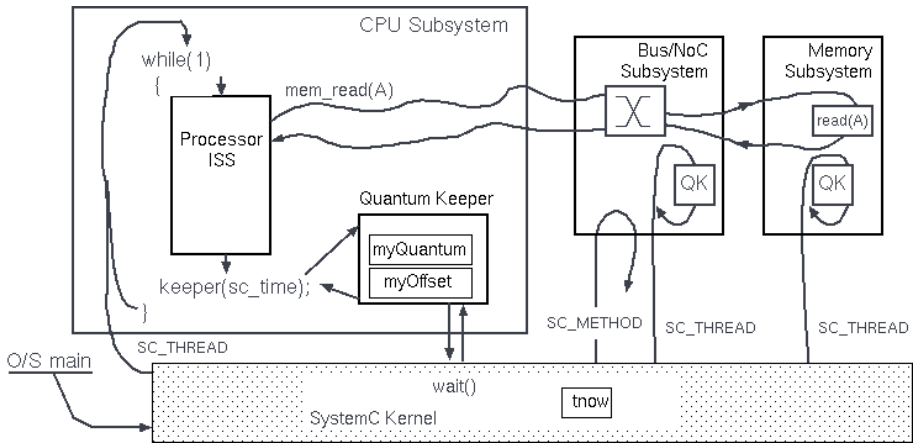


Figure 5.7: Typical structure of thread using loosely-timed modelling with a quantum keeper.

A thread is passed between components and back to the originator and only rarely enters the SystemC Kernel.

Each thread has a variable called **delay** of how far it has run ahead of kernel simulation time, and only yields when it needs an actual result from another thread or because its delay exceeds a chosen value.

Each component increments the delay field in the TLM calls it processes, according to how long it would have delayed the client thread under approximate timing.

Each component may have a quantum keeper, but only one is illustrated in Figure 5.7 at the CPU subsystem. Every thread must encounter a quantum keeper at least once in its outermost loop.

Keeper code is just a conditional resynch:

```
if (delay > myQuantum) { sc_wait(delay); delay = 0; }
```

By calling `wait(delay)` the simulation time will advance to where the caller has got to while running other pending processes. The myQuantum could be a system default value or a special value for each thread or component.

Or where a thread needs to block to wait for a result from some other thread:

```
while (!condition_of_interest)
{
  sc_wait(delay);
  delay = 0;
}
```

- **Large time quantum:** fast simulation,
- **Small time quantum:** transaction order interleaving is more accurate.

Loosely-timed is appropriate for software development. It supports modelling of timers and interrupts, sufficient to boot an operating system. Simulation time still exists, but processes may be temporally decoupled from simulation time.

Transactions may execute in a different sequence from reality: **sequential consistency** compromised ?

The loosely-timed coding style is normally used with blocking models. and is appropriate for software development or when only very rough timing is needed. However the accuracy can be altered by reducing the **quantum**, but this also reduces performance.

In the OR1K example on the course web site, the thread from the main ISS is used to make the transactions on the bus, cache and memory sub-systems. In a more-advanced setup, the processor model would continue to operate while it has outstanding transactions on the bus, and the cache model would continue to serve the processor while it has outstanding transactions on the DRAM and so on. More than one thread is then needed.

All processes will typically use the same quantum (although they need not).

Components may share quantum keepers or have their own. One just needs to make sure concurrent activities are added to different offset variables.

In summary, large time quantum means fast simulation whereas small time quantum means transaction order interleaving is more accurate.

Transactions may execute in a different sequence from reality, hence violating the **sequential consistency** of the model.

Many real systems also do not preserve sequential consistency:

- global system design may typically be robust against out-of-order transactions,

- memory fence instructions are included in modern processors to control consistency,
- but RaW hazards and cheque-not-clearing-in-time hazards can bite!

5.15 ESL TLM in SystemC: TLM 2.0

TLM2.0 (July 2008) tidies away the TLM1.0 interface inheritance using **convenience sockets** and defines the **Generic Payload**.

Also defined memory/garbage ownership and transport primitives with timing and backdoor access to RAM models.

```
trans->set_command(tlm::TLM_WRITE_COMMAND);
trans->set_address(addr);
trans->set_data_ptr(reinterpret_cast<unsigned char*>(&data));
trans->set_data_length(4);
trans->set_streaming_width(4);
trans->set_byte_enable_ptr(0);
trans->set_response_status( tlm::TLM_INCOMPLETE_RESPONSE );

socket->b_transport(*trans, delay);
```

Other standard payloads (e.g. 802.3 frame or audio sample) might be expected ?

The generic payload can be extened on a a custom basis and intermediate bus bridges and routers can be polymorphic about this: not needing to know about all the extensions but able to update timestamps to model routing delays.

It also defines memory/garbage ownership and transport primitives with timing. Finally, it defines a raft of useful features, such as automatic conversion between blocking and non-blocking styles.

However, TLM 2.0 it is a bit too complex to get a deep understanding of in the time available for the Part II course, hence the motivation for the 'Toy ESL' practicals in TLM 1.0 style.

LG 6 — ABD - Assertion-Based Design

Topics: Assertion-Based Design, Assertion Synthesis to H/W Monitors.

Declarative programming involves writing assertions that hold for all time. For instance, on an indicator panel *never is light A on at the same time as light B.*

Ideally, assertion-based design (ABD) involves:

- Writing assertions at design capture time *before* detailed coding starts.
- Writing further assertions as coding progresses.
- Structuring testing around assertions.

Assertions are (conjunctions of):

- Imperative safety checks (like assert.h in C++)
- Declarative safety properties (what must always or never happen).
- So-caled **strong** properties of liveness and deadlock (what should eventually happen or not happen).

All three can potentially be proved by formal methods.

Simulation (aka *dynamic validation*) can sometimes find safety violations and sometimes find deadlock but it cannot prove liveness.

Assertions can be imported from previous designs or other parts of the same design for global consistency.

ABD shows up corner case problems not encountered in simulation.

A formally-verified result may be *required* by the customer.

Assertion-based design is an approach that encourages writing assertions as early as possible, even *before* coding starts.

The Z notation for formal specifications is quite well known and suitable for describing properties of data structures. Other examples from software are object constraint language (OCL), Alloy and so on. Database integrity and consistency rules are also common examples of assertions.

Assertions should be machine readable and machine provable, as far as possible. There's even a school of thought that assertions should be executable, whenever possible, thereby generating example output that conforms to what is specified.

Assertions are (combinations of):

- Imperative safety checks (like assert.h in C++), that must hold when a flow of control reaches it.
- Declarative safety properties, that always hold, such as 'Never are both the inner and outer door of the airlock open at once unless we are on the ground'. Declarative safety properties normally use the keywords **never** or **always**.
- Strong properties (also declarative) about liveness and deadlock. **Strong**, in the language of PSL, means that the property cannot be checked by simulation, only by a static formal method that checks all possible executions.

All three can potentially be proved by automated provers, such as model checkers. Contemporary, industrial SoC design only uses fully automated provers, whereas research in specification and verification often uses manually-guided provers where the computer may make suggestions about proof steps buts its main role is to check the result has been derived without a false step.

Declarative assertions are written either as assertions about the current state or about state trajectories (i.e. sequences of states). In general, trajectory expressions can be compiled into a checker automata (or RTL sub-circuit) and a state assertion can be applied to the output function (terminal) of the automata (RTL module).

A declarative assertion about the current state is intrinsically universally quantified over time, so really it is about all states. Applying the same reasoning to a trajectory assertion implies that the property holds for all occurrences of the trajectory, whether overlapping or not.

Assertions can be imported from previous designs or other parts of the same design for global consistency.

ABD shows up corner case problems not encountered in simulation. And in some cases, such as medical life-support systems, a formally-verified result may be required by the customer.

6.1 ABD - The alternative: Simulation

The alternative is extensive simulation with overnight testing for **regressions**.

Can either write a RTL or ESL yes/no automaton as part of the test bench. Or one can spool the outputs to file and **diff** against **golden** with PERL script.

Downfall of simulation: it's non-exhaustive and time consuming.

ABD benefits (and challenges):

- Completeness (how to define this?)
- Scalability (tools limited in practice?),
- Rare corner situations (unusual conjunctions of events) are covered.

But: Simulations

- are needed for performance analysis and general design confidence,
- can generate some production **test vectors**.
- can be partly formal: using bus monitors for dynamic validation and Specman/VERA constrained pattern generators for stimulus.

Simulation is effective at finding many early bugs in a design. It can sometimes find safety violations and sometimes find deadlock but it cannot prove liveness.

Once the early, low-hanging bugs are fixed, formal proof can be more effective at finding the remainder. These tend to lurk in unusual corner cases, where particular alignment or conjunction of conditions is not handled correctly.

If a bug has a one in ten million chance of being found by simulation, then it will likely be missed, since fewer than that number clock cycles might typically be simulated in any run. However, given a clock frequency of just 10 MHz, the bug might show up in the real hardware in one second!

Simulation is generally easier to understand. Simulation gives performance results. Simulation can give a golden output that can be compared against a stored result to give a pass/fail result. A large collection of

golden outputs is normally built up and the current version of the design is compared against them every night to spot **regressions**.

Simulation **test coverage** is expressed as a percentage. Given any set of simulations, only a certain subset of the states will be entered. Only a certain subset of the possible state-to-state transitions will be executed. Only a certain number of the disjuncts to the guard to an IF statement may hold. Only a certain number of paths through the block-structured behavioural RTL may be taken.

There are many ways of defining coverage: for instance do we have to know the reachable state space before defining the state space coverage, or can we use all possible states as the denominator in the fraction?

In general software, a common coverage metric is the percentage of lines of code that are executed.

Scaling of formal checking is a practical problem: today's tools certainly cannot check a complete SoC in one pass. An incremental approach based around individual sub-systems is needed.

6.2 Formally Synthesised Bus Monitor

Dynamic validation: a checker flags protocol violations:

- Safety violations are indicated straightaway,
- Liveness properties can be noted as they are satisfied.

If we have a simulator (e.g. a Verilog interpreter) that does not understand temporal logic, the assertions can be compiled to an RTL checker automaton.

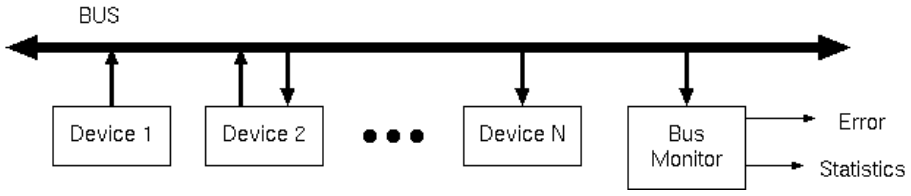


Figure 6.1: Dynamic validation: Monitoring bus operation with a hardware monitor.

A bus monitor connects to the net-level bus. (NB: No tri-states in a SoC: everything is typically multiplexors.)

For a liveness property we can indicate whether it has been tested at least once and also whether there is a pending condition that yet to be satisfied.

Monitor can keep statistics as well as detect protocol violations.

Can it be synthesised from a formal spec ? www.cl.cam.ac.uk/research/srg/han/hprls/orangepath/transactors and Bus Monitors

Busses and I/O ports often behave according to a protocol. Such protocols are usefully defined and checked using a formal method.

A bus monitor connects to the net-level bus. It is an instantiated component that looks like any other component. It can be left in for actual fabrication or, more typically, used only for simulation and removed for fabrication.

A bus monitor can keep statistics as well as detect protocol violations. It is useful to know how much data was transferred, as well as other figures, such as the average size of a data block or the percentage of time more than one initiator was contending for the bus.

Can it be synthesised from a formal spec ? Yes, the internals of the bus monitor can be normal RTL that was synthesised from the a formal specification.

www.cl.cam.ac.uk/research/srg/han/hprls/orangepath/transactors and Bus Monitors

For **safety violations** the monitor can print out an error as soon as it is detected. However, **liveness** properties cannot be checked as cleanly during simulation: we can only check to see if they have occurred and the number of occurrences printed at the end of simulation. If a liveness property has been satisfied once, it is likely that it might happen infinitely often in the future.

In other words, a checker for a safety property will assert an error output as soon as the property is violated, whereas that for a liveness property will start off flagging an error that will go away once the property has been found to hold at least once.

6.3 Is a formal specification complete ?

Is a formal specification complete ?

- Does it fully-define an actual implementation (this is overly restrictive) ?
- Does it exactly prescribe all allowable, observable behaviours ?

By 'formal' we mean a machine-readable description of what is correct or incorrect behaviour. A **complete** specification might describe all allowable behaviours and prohibit all remaining behaviours, but most formal definitions today are not complete in this sense. For instance, a definition that consists of of a list of safety assertions and a few liveness assertions might still allow all sorts of behaviours that the designer knows are wrong. He can go on adding more assertions, but when does he stop ?

One might define a 'complete specification' as one that describes all observable behaviours. Such a specification does not restrict or prescribe the internal implementation in black box terms since this is not observable.

When testing an IP block, can we compute coverage somehow: e.g. What percentage of rule disjuncts held as dominators (on their own) ? Or, e.g. What (inverse log) percentage of reachable state space was spanned?

There are no well-accepted coverage metrics for formal specifications. We could measure what percentage of rule disjuncts held as dominators (on their own) ? There is no clear definition of 100 percent coverage.

6.4 ABD - State versus Path, Concrete Versus Symbolic.

Many assertions are over **concrete state**. For instance *'Never is light A off when light B is on'* .

Other assertions need to refer to **symbolic values**. For instance *'The value in register X is always less than the value in register Y'* .

State properties describe the current state only. For instance '*Light A is off and light B is on*'.

Path properties relate successive state properties to each other. For instance '*light A always goes off before light B comes on* '.

We shall see PSL requires the symbolic values be embedded in the bottommost 'modelling layer' and that its temporal layer cannot deal with symbolic values. For instance, we cannot write '$\{A(x); B(y)\} \mid => \{C(x, y)\}$'.

Also note that the internal representation used by a checker tool for a concrete property could use a symbolic encoding (such as a BDD) to handle an exponentially-large state space using reasonable memory.

6.5 ABD - PSL Assertion, General Structure

PSL - Property Specification Language

www.project-veripage.com/psl_tutorial_2.php

Figure 6.2: General structure of a PSL assertion

As in most temporal logics, there are three main directives:

1. always and never,
2. next (family of them),
3. eventually!

The `always` directive is the most frequently used and it specifies that the following property expression should be checked every clock. The `never` directive is a shorthand for a negated `always`.

The `next` directive relates successive state properties, as qualified by the clocking event and qualifying guard.

The `eventually!` directive is for liveness properties that must happen at some time in the future. The `eventually!` directive is suffixed with a bang sign to indicate it is strong property that cannot be (fully) checked with simulation.

The general structure of a PSL assertion has the following parts:

- A name or label that can be used for diagnostic output.
- A verification directive, such as **assert**.
- When to check, such as **always** or **eventually!**.
- The property to be checked: a state expression or a temporal logic expression.
- A qualifying guard, such as a clock edge or enable signal at which time we expect the assertion to hold.

6.6 ABD - PSL Four-Level Syntax Structure

The abstract syntax of PSL uses for levels:

- Since the language is embedded in the concrete syntax of several other languages, such as Verilog, SystemVerilog and VHDL, its syntactic details vary. In particular, creating state predicates involves expressions that range over the nets and variables of the host language. The precise means for this is defined by the **MODELLING LAYER** that allows one to create state properties using RTL.

  Non-boolean sub-expressions can be used in the modelling layer to generate boolean state predicates.

```
assign tempok = temperature < 99;
```

- All high-level languages and RTLs have their own syntax for boolean operators and this can be used within the modelling layer. However boolean combinations can also be formed using the PSL **BOOLEAN LAYER**.

```
not (rd and wr); -- rd, wr are nets in the RTL (modelling layer).
```

- The PSL **TEMPORAL LAYER** allows one to define named sub-expressions and properties that use the temporal operators. For example:

```
-- Sequence definition
sequence s1 is {pkt_sop; (not pkt_xfer_en_n [*1 to 100]); pkt_eop};

sequence s2 is {pkt_sop; (not pkt_xfer_en_n [*1 to 100]); pkt_aborted};

-- Property definition
property p1 is reset_cycle_ended |=> {s1; s2};

-- Property p1 uses previously defined sequences s1 and s2.
```

- The PSL **VERIFICATION LAYER** implements the declarative language itself. It includes the main keywords, such as **assert**.

6.7 ABD - PSL Extended Regular Expressions

PSL has a rich regular expression syntax for pattern matching. These are called *SERES* or sequences. SERES stands for Sugar Extended Regular Expression, where Sugar was an older name for PSL.

Sequence elements are state properties from Modelling and Boolean layers. Core operators are (of course): disjunction, concatenation and arbitrary repetition. As a temporal logic: interpret concatenation as a time sequencing.

- **A;B** Semicolon denotes sequence concatenation
- **A[*]** Postfix asterisk for arbitrary repetition
- **A|B** Vertical bar (stile) for alternation.

Make easier to use with additional operators defined in terms of primitives:

- **A[+]** One or more occurrences: **A;A[*]**
- **A[*n]** Repeat `n` times
- **A[=n]** Repeat `n` times non-consecutively
- **A:B** Fusion concatenation (last of A occurs during first of B)

Further repetition operators denote repeat count ranges. Repeat counts must be compile-time constant (for today's standard/tools).

6.8 ABD - PSL Properties and Macros

PSL defines some simple path to state macros

- `rose(X)` means **!X; X**
- `fell(X)` means **X; !X**

Others are easy to define:

- `stable(X)` can be defined as **X; X — !X; !X**
- `changed(X)` can be defined as **X; !X — !X; X**

6.9 ABD - Naive Path to State Conversion

Compiling regular expressions to RTL is relatively straighforward. The following ML fragment handles the main operators. By converting a path expression to a state expression we can generate an RTL checker for use in dynamic validation. It can also be used for converting all path expressions to state expressions if a formal proof tool is to be used that only handles state expressions, such as a basic BDD package.

```
fun gen_pattern_matcher g (seres_statexp e) = gen_and2(g, gen_boolean e)

|   gen_pattern_matcher g (seres_diop(diop_seres_alternation, l, r)) =
    let val l' = gen_pattern_matcher g l
        val r' = gen_pattern_matcher g r
        in gen_or2(l', r') end

|   gen_pattern_matcher g (seres_diop(diop_seres_catenation, l, r)) =
    let val l' = gen_dff(gen_pattern_matcher g l)
        val r' = gen_pattern_matcher l' r
        in r' end

|   gen_pattern_matcher g (seres_diop(diop_seres_fusion, l, r)) =
    let val l' = gen_pattern_matcher g l
        val r' = gen_pattern_matcher l' r
        in r' end

|   gen_pattern_matcher g (seres_monop(mono_arb_repetition, l)) =
        let val nn = newnet()
            val l' = gen_pattern_matcher nn l
            val r = gen_or2(l', g)
            val _ = gen_buffer(nn, r)
            in r end

|   gen_pattern_matcher g (seres_diop(diop_n_times_repetition, l,
                          seres_statexp(x_num n))) =
        let fun f (g, k) = if k=0 then g else
            gen_pattern_matcher (f(g, k-1)) l
            in f (g, n) end
```

The ML fragment `gen_pattern_matcher` handles concatenation, fusion concatenation, alternation, arbitrary repetition and n-times repetition. However, this generates a one-hot automaton and there are far more efficient procedures used in practice and given in the literature.

A harder operator to compile is the length-matching conjunction (introduced shortly), since care is needed when each side contains arbitrary repetition and can declare success or failure at a number of possible times.

6.10 ABD - SERES Pattern Matching Example

Suppose four events are supposed to always happen in sequence:

Consider **always true[*]; A; B; C; D**

Basic pattern matcher applied to **A;B;C;D** generates:

```
  DFF(g0, A, clk);
  AND2(g1, g0, B);
  DFF(g2, g1, clk);
  AND2(g3, g2, C);
  DFF(g4, g3, clk);
  AND2(g5, g4, D);
> val it = x_net "g5" : hexp_t
```

Although correct, distributive laws mean that all four signals must hold constantly (ignoring start up).

It is important to note that putting a SERES as the body of an **always** statement probably does not have the desired effect: it does not imply that the contents occur sequentially. Owing to the overlapping occurrences interpretation, such an **always** statement distributes over sequencing and so implies every element of the sequence occurs at all times. Therefore, it is recommended to **always uses an SERES as part of a suffix implication** or with some other temporal layer operator.

6.11 ABD - PSL Temporal Layer Operators

The disjunction (ORing) of a pair of sequences is already supported by the SERES disjunction operator. But PSL sequences can also be combined with implication and conjunction operators in the 'temporal layer'.

- `P |-> Q` P is followed by Q (one state overlapping),
- `P |=> Q` P is followed by Q (immediately afterwards),
- `P && Q` P and Q occur at once (length matching),
- `P & Q` P and Q succeed at once,
- `P within Q` P occurred at some point during Q,
- `P until Q` P held at all times until Q started,
- `P before Q` P held before Q held.

6.12 ABD - Sequence Constraint as a Suffix Implication

Earlier example: try phrasing using suffix implication:

Perhaps this will serve as a good `always` assertion?

`always A;B |=> C;D` now gives

```
  DFF(g0, A, clk);
  AND2(g1, g0, B);
  DFF(g2, g1, clk);
  INV(g3, C);
  AND2(g4, g3, g2);  // Holds if C missing
  DFF(g5, g2, clk);
  INV(g6, D);
  AND2(g7, g5, g6);  // Holds if D missing
  OR2(g8, g7, g4);
> val it = x_net "g8" : hexp_t   // Holds on error
```

Even this is not very specific: C and D might occur at other times.

PSL has a `onehot` operator: can use it to stop more than one of these values at once.

Data conservation: At an interface we commonly want to assert that data is not lost or duplicated. Is PSL any help? Not realy, ideal one should use a language that can range over symbolic data and tagged streams of data.

6.13 ABD - A Simple Model Checker

Formal checker for a state safety property:

Algorithm: **'Find reachable state space'**: Add successors of current set until closure.

1. S := { q_0 } // initial state
2. S := S $\cup$ $\{q' \mid \exists\ \sigma \in \Sigma, q \in S\ .\ NSF(q,\sigma) = q'\ \}$
3. If safety property does not hold in any $q \in S$ then flag error.
4. If S increased in step 2 then goto step 2.

S can be held explicitly in bit map form or symbolically as a BDD.

Variation: property to check might be a path property, so either

- Compile it to a checking automaton (becomes a state property of expanded NSF), or
- Expand it as we go (using modal mu calculus).

The PSL strong assertions need to be checked with a formal proof tool. Model checking is normally used.

A model checker explores every possible execution route of a finite-state system by exploring the behaviour over all possible input patterns.

There are two major classes of model checker: explicit state and symbolic. Explicit state checkers actually visit every possible state and store the history in a very concise bit array. If the bit array becomes too big they use probabilistic and hashing techniques. The main example is Spin. Symbolic model checkers manipulate expressions that describe the reachable state space and these were famously implemented as BDDs in the SMV checker. There are also other techniques, such as bounded model checking, but the internal details of model checkers is beyond the scope of this course.

The most basic model checker only checks state properties. To check a path property it can be compiled into an automaton and included as part of the system itself.

To check safety over all reachable states, one can either find the reachable state space and then see if all of it is safe, or one can check the safety predicate after each step in creating the reachable state space. The algorithm for the reachable state space, given on the slide, is simply to start with the initial state and repeatedly add any successors until closure.

To check liveness formally is beyond the scope of this course, but one algorithm is to repeatedly trim cul-de-sacs from the state transition graph so that only a core where all states are reachable from all others remains.

6.14 ABD - Boolean Equivalence Checker

Often we have two implementations of a combinational circuit to check for equivalence. For example, they might be:

1. RTL version: pre-synthesis,
2. Gate-level version: post-synthesis or post-layout.

Sources of difference between the designs might be manual implementation of one of them, manual edits to synthesiser outputs and EDA tool faults.

Boolean equivalence: do the two functions produce the same output?

- For all input combinations ?
- For a subset of input combinations (some input patterns are don't cares).

Then: feed negation of mitre to a SAT solver.

Result: if there are no solutions, designs are equivalent.

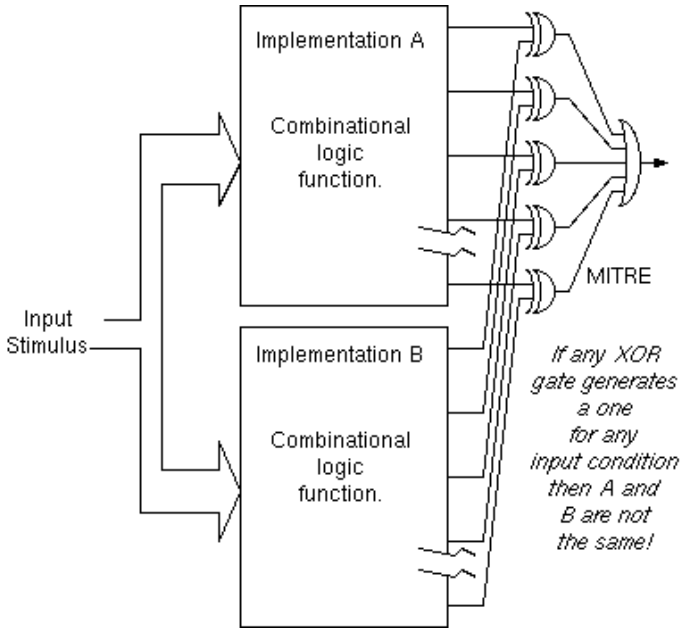


Figure 6.3: A mitre compares the outputs from a pair of supposedly-equivalent combinational components.

Often we have two implementations to check for equivalence, for instance, when RTL is turned into a gate-level netlist by synthesis we have:

- RTL version: pre-synthesis, and
- Gate-level version: post-synthesis.

After place and route operations, it is common to extract the netlist out from the masks and check that for correctness, so this is another source of the same netlist.

There two main sources of potential errors: 1. manual edits at any point may upset correctness. 2. EDA tools used in the flow may have bugs.

The **boolean equivalence problem** is do two functions produce the same output. However, are we interested for all input combinations? No, normally we are only interested in a subset of input combinations (because of don't care conditions).

The method, shown in Figure 6.3, is to create a **mitre** of the two designs using a disjunction of XOR gate outputs. Then, feed negation of mitre to a SAT solver to see if it can find any input condition that produces a one on the output.

SAT solving is a matter of trying all input combinations, so has exponential cost in theory and is NP complete. However, modern solvers such as **zChaff** essentially exploit the intrinsic structure of the problem so that they normally are quite quick at finding the answer.

Result: if there are no input combinations that make the mitre indicate a functionality difference, then the designs are equivalent.

Commercial example Formality

6.15 ABD - Sequential Logic Equivalence: Example

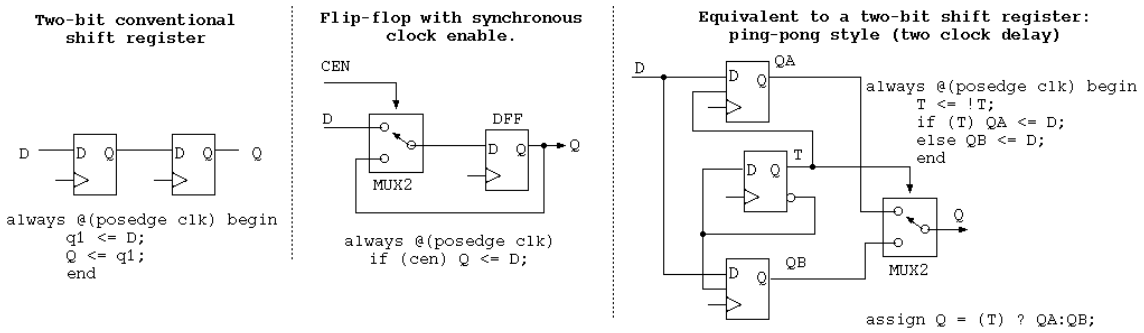


Figure 6.4: Two circuits that use different amounts of internal state to achieve the same functionality.

The figure shows implementations of a two-bit shift register. They differ in amount of internal state. They have equivalent observable behaviour (ignoring glitches). Note, to implement larger delays, the design based on multiplexors might use less power then the design based on shifting, since fewer nets toggle on each clock edge.

6.16 ABD - Sequential Equivalence Checker

Do a pair of designs follow the same state trajectory ?

- Considering the values of all state variables ?
- Considering a re-encoding of the state variables ?
- For an observable subset of the state (e.g. at an interface) ?
- When interfacing with a given reactive automaton ?

Other freedoms that could be allowed within the notion of equivalence:

- Temporally floating ports - latency independence. With floating ports we do not consider the relative timing of events between ports, only the relative timing of events within each port.
- Synchronous or asynchronous (turn-taking) composition. If a pair of circuits are combined, do they share a common clock or take it in turns to move?
- Strong or weak bi-simulation (stuttering equivalence). A stuttering equivalence between a pair of designs may exist if we disregard the precise number of clock cycles each took to achieve the result (such as different implementations of a microprocessor).

Practical problem: Designs may only be equivalent in the used portion of the state space. Hence we may need a number of side conditions that specifiy the required operating conditions.

6.17 ABD - Sequential Logic Simplification

A finite-state machine may have more states than it needs to perform its observable function because some states are totally equivalent to others in terms of output function and subsequent behaviour. Note that one-hot coding does not increase the reachable state space and so is **not** an example of that sort of redundancy.

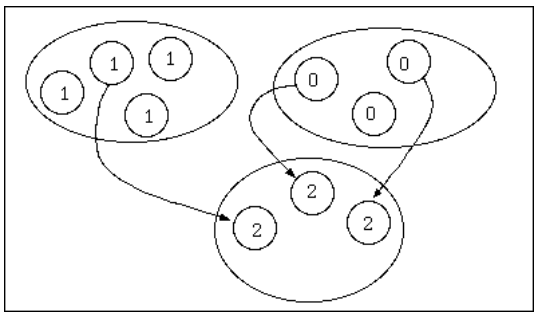


Figure 6.5: Illustrating the basic step of one bisimulation reduction algorithm.

A Moore machine can be simplified by the following procedure:

- 1. Partition all of the state space into blocks of states where the observable outputs are the same for all members of a block.
- 2. Repeat until nothing changes (i.e. until it closes) For each input setting:
  - 2a. Chose two blocks, B1 and B2.
  - 2b. Split B1 into two blocks consisting of those states with and without a transition from B2.
  - 2c. Discard any empty blocks.
- 3. The final blocks are the new states.

Alternative algorithm: start with one partition per state and repeatedly conglomerate. The best algorithms use a mixture of the two approaches to meet in the middle.

en.wikipedia.org/wiki/Wikipedia: Formal Equivalence Checking

For instance CADP package: developed by the VASY team at INRIA.

Products: Conformal by Cadence, Formality by Synopsys, SLEC by Calypto.

One future use of this sort of procedure might be to generate an instruction set simulator for a processor from its full RTL implementation. This sort of de-pipelining would give a non-cycle accurate, higher-level model that runs much faster in simulation.

Different implementations of a circuit may vary in their state encoding or even in the amount of state they keep in terms of bits. One might be simpler or better than the other for a given purpose. At times we need to check the equivalence of deigns.

For a synchronous clock domain, if two designs are known to have the same state encodings, then the problem degenerates to that of combinational equivalence checking of their respective next-state functions. For each D-type flip-flop we need to check the combinational equivalence of its sourcing circuit in the two designs. However, even if the state encodings are the same, there can be un-used portions of the state space which must be treated as don't-cares for this check.

If the state encoding is known to be changed, then what can be compared? Perhaps we can compare the trajectory of states between two designs, building up our own mapping between the encodings in each design. Generally, two designs will have the same set of output terminals and so the basis of the mapping is equivalence classes formed around each possible setting of the output terminals.

This leads to the concept of full equivalence in terms of external **observable behaviour**. Do the two designs behave the sames as each other in black box testing: that is, without any knowledge of the internal behaviour. If so, they are said to **bi-simulate** each other.

Again, not all of the reachable state space may be used. The circuit might always be wired up externally so that one input is a delayed version of one output. Therefore the question arises, do a pair of designs follow the same state trajectory when interfacing with a specified reactive automaton ?

Commonly, the number of clock cycles of delay through a sub-system (its latency) is not important and perhaps can be disregarded in equivalence checking. This leads to the concept of **temporally floating ports** (not lectured in 2008/9), where a pair of designs are equivalent if the timing behaviour inside a port (subset of the external collections) appears equivalent, even though we would see differences if we looked at the relative timing of operations with respect to other ports. For example, the precise order in which an Ethernet hub chip sends a broadcast packet to each of its output ports does not matter, as long as it is actually delivered to each port, and from any given port this ordering cannot be perceived without peeking at the other ports. This sort of floating is similar to the temporal decoupling ides in the loosely-timed models, in that systems are designed, at the high level, to be tolerant to timing and performance variations in their major components. .

Two other variations in the problem definition arise for systems where the exact number of clock cycles is not considered important. This clearly also applies to asynchronous systems without a clock.

- When a pair of sub-systems are wired to each other, the composition may be a synchronous or asynchronous (turn-taking) composition. The algorithms used for making the different combinations may vary (a research topic of DJG) and the possible behaviours may also be different. For instance, it might only be possible to enter a given state (on the next clock edge) without the simultaneous change of two communicating nets, but if these are sourced in different components that share the same clock this might never happen. This is rather like the two main types of auction: where participants either bids in turn, with knowledge of the previous bid, or all bid together without knowledge (competitive tender) and then another round is used when there are tie-breaks needed.
- Strong or weak bi-simulation: a pair of designs might differ in the number of clock cycles they take to produce a response to a given input. As mentioned, this can be hidden by putting the input and output in different temporal groups, or it can be considered a difference (strong bi-simulation) or it can be considered not a difference if no nets other than the clock have changed in the meantime: a **stuttering equivalence** (a form of weak bi-simulation).

6.18 ABD - Automated Stimulus Generation

Testbench automation: generate pseudo-random input under constraining assertions.

Products: Verisity's Specman Elite, Synopsys Vera.

www.verisity.com/products/specman.htmlwww.open-vera.com

Simulations and test programs require **stimulus**. This is a sequence of input signals, including clock and reset, that exercise the design.

Given that formal specifications for many of the input port protocols might exist, one can consider automatic generation of the stimulus, from random sources, within the envelope defined by the formal specification. Several commercial products do this, including Verisity's Specman Elite, Synopsys Vera.

Here is an example of some code in Specman's own language, called 'e', that defines a frame format used in

networking. Testing will be inside envelope defined by **keep** statement.

```
struct frame {
  llc: LLCHeader;
  destAddr: uint (bits:48);
  srcAddr: uint (bits:48);
  size: int;
  payload: list of byte;
  keep payload.size() in [0..size];  };
```

Sequences of bits that conform to the frame structure are accepted at an input port of the design under test.

An heirarchy of specifications and constraints is supported. One can compose and extend one specification to reduce its possible behaviours:

```
  extend frame { keep size == 0;  };
```

6.19 ABD - Conclusion

ABD today is often focussed on safety and liveness properties of systems and formal specifications of the protocols at the ports of a system. However, there are many other useful properties we might want to ensure or reason about, such as those involving counting and/or data conservation. These are less-well embodied in contemporary tools.

PSL deals with concrete values rather than symbolic values. Many interesting properties relate to symbolic data (e.g. specifying the correct behaviour of a FIFO buffer). Using PSL, all symbolic tokens must be wrapped up in the modelling layer which is not the core language.

Formal methods are taking over from simulation, with the percentage of bugs being found by formal methods growing. However, there is a lack of formal design entry. Low-level languages such as Verilog do not seamlessly mix with automatic synthesis from formal specification and so double-entry of designs is common.

LG 7 — SoC DRAM and Bus/NoC Structures.

The delay and power problem, physical parameters:

- Speed of light on silicon and on a PCB is 200 metres per microsecond.
- A clock frequency of 2 GHz has a wavelength of 2E8/2E9 = 10 cm.
- Synchronous digital clock domain requires connections to be less than (say) 1/10th of a wavelength.
- RC time constants must also be overcome, so need to register a signal in several D-types if it passes from one corner of an 8mm chip to the other!
- DRAM is several centimeters away from the SoC and has significant internal delay.

Hence we need to use protocols that are tolerant to being registered (passed through D-type pipeline stages). The four-phase handshake has one datum in flight and degrades with reciprocal of delay. We need something a bit like TCP that keeps multiple datums in flight.

But first let's revist the simple hwen/rwen system used in the 'socparts' section.

7.1 Basic Bus: One initiator.

The bus protocol in the eailer slides that used **addr, hwen, hren, wdata** and **rdata** does not tolerate registering for reads, but if a **ready** or other acknowledgement signal were added, it would be like the four phase handshake and work correctly, but poorly for long distances over the chip.

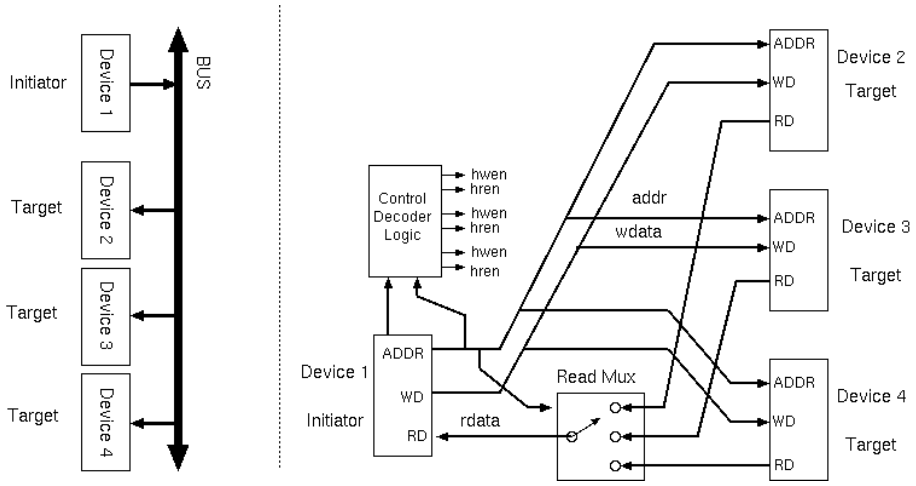


Figure 7.1: Example where one initiator addresses three targets.

Figure 7.1 shows such a bus with one initiator and three targets.

No tri-states are used: on a modern SoC address and write data outputs use wire joints or buffers, read data uses multiplexors. There is only one initiator, so no bus arbitration is needed.

Max throughput is unity (i.e. one word per clock tick). Typical SoC bus capacity: 32 bits $\times$ 200 MHz = 6.4 Gb/s, but owing to protocol degrades with distance. This figure can be thought of as unity (i.e. one word per clock tick) in comparisons with other configurations we shall consider.

The most basic bus has one initiator and several targets. The initiator does not need to arbitrate for the bus since it has no competitors.

Bus operations are reads or writes. In reality, most on-chip busses support burst transactions, whereby multiple consecutive reads or writes can be performed as a single transaction with subsequent addresses being implied as offsets from the first address.

Interrupt signals are not shown in these figures. In a SoC they do not need to be part of the physical bus as such: they can just be dedicated wires running from device to device. (For ESL higher-level models and IP-XACT representation, interrupts need management in terms of allocation and naming in the same way as the data resources.)

Un-buffered wiring can potentially serve for the write and address busses, whereas multiplexors are needed for read data. Buffering is needed in all directions for busses that go a long way over the chip.

7.2 Basic bus: Multiple Initiators.

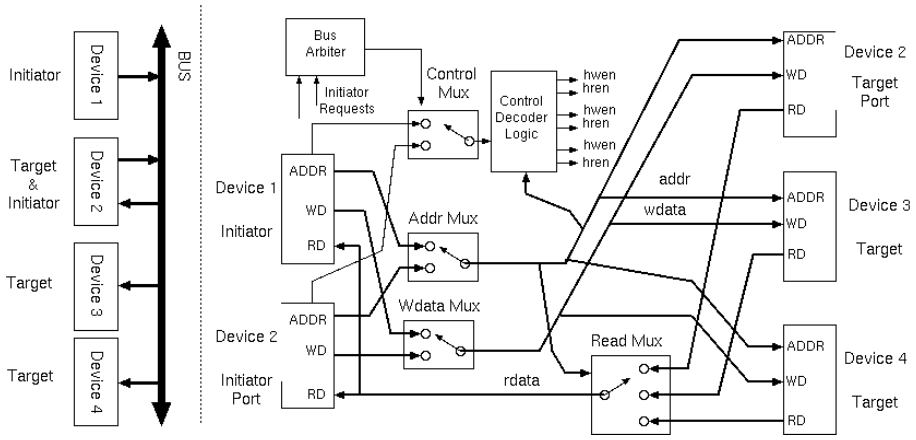


Figure 7.2: Example where one of the targets is also an initiator (e.g. a DMA controller).

Basic bus, but now with two initiating devices.

Needs arbitration between initiators: static priority, round robin, etc.. With multiple initiators, the bus may be busy when a new initiator wants to use it, so there are various arbitration policies that might be used. Preemptive and non-preemptive with static priority, round robin and so on.

The maximum bus throughput of unity is now shared among initiators.

Since cycles now take a variable time to complete we need acknowledge signals for each request and each operation (not shown).

How long to hold bus before re-arbitration ? Commonly re-arbitrate after every burst. The latency in a non-preemptive system depends on how long the bus is held for. Maximum bus holding times affect response times for urgent and real-time requirements.

7.3 Bridged Bus Structures.

To make use of the additional capacity from bridged structures we need at least one main initiator for each bus. However, a low speed bus might not have its own initiators: it is just a slave to one of the other busses.

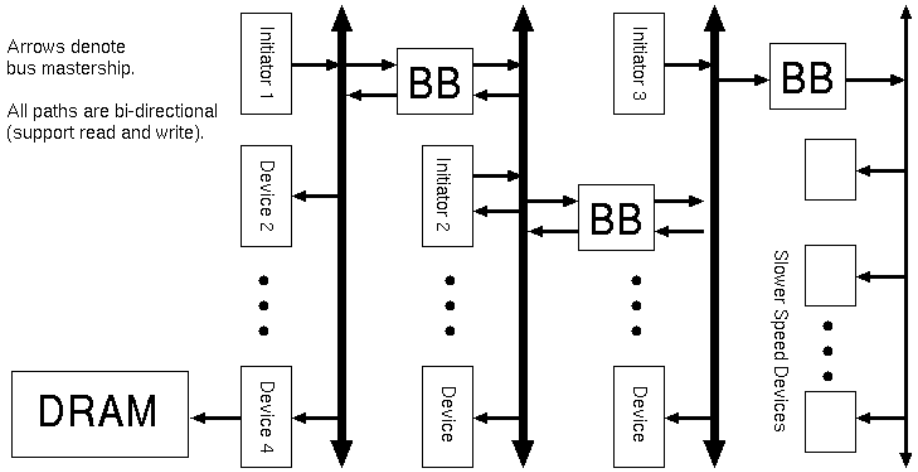


Figure 7.3: A system design using three main busses.

Bus bridges provide full or partial connectivity and some may write post. Global address space, non-uniform access time (NUMA). Some busses might be slower, narrower or in different clock domains from others.

The maximum throughput is the sum of that of all the busses that have their own initiators, but the achieved throughput will be lower if the bridges are used a lot: a bridged cycle consumes bandwidth on both sides.

How and where to connect DRAM is always a key design issue. The DRAM may be connected via a cache. The cache may be dual ported on to two busses, or more.

Bus bridges and top-levels of structural wiring automatically generated. An example tool that does this is ARChitect2 from ARC International (now part of Virage Logic).

7.4 Network on Chip: Simple Ring.

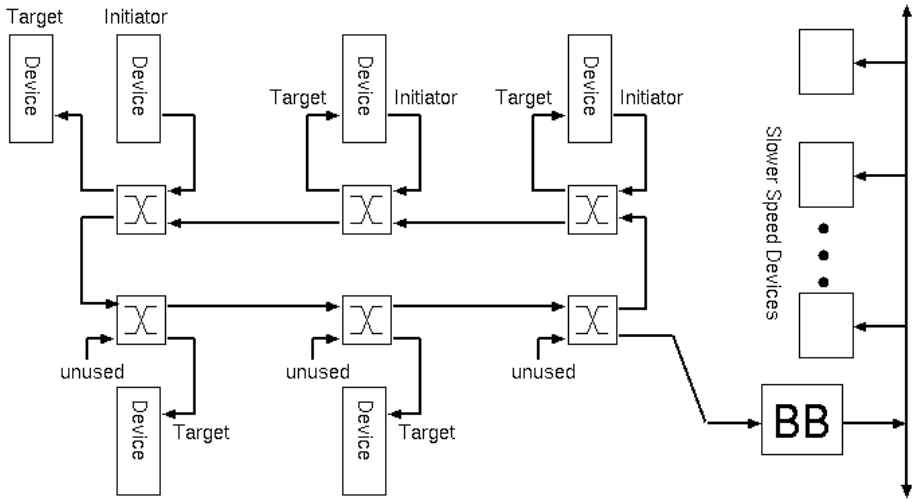


Figure 7.4: A ring network: a low-complexity network on chip structure.

Two-by-two switch element enables formation of rings and other nocs.

Switch element is registered: hence network can span the chip.

Network needs to carry decoupled requests and response packets.

Local arbitration in each element. Global policies required to avoid deadlock and starvation.

Give priority to traffic already on the ring: LAN-like buffering at source.

Does not carry interrupts or other sideband signals.

Single ring: throughput=2. counter-rotating ring: throughput=4.

Switched networks require switching elements. With a 2x2 element it is easy to build a ring network. The switching element may contain buffering or it may rely on back-pressure to make sources reduce their load.

Single ring: throughput=2. Counter-rotating ring (one ring in each direction): throughput=4 since a packet only travels 1/4 of the way round the ring on average.

Using a network, the delay may be multiple clock cycles and so a **write posting** approach is reasonable. If an initiator is to have multiple outstanding read requests pending it must put a token in each request that is returned in the response packet for identification purposes.

Although there can be effective local arbitration in each element, a network on a chip can suffer from deadlock. Some implementations uses separate request and response networks, so that a response is never held up by new requests, but this just pushes deadlock to the next higher logical level when some requests might not be servicable without the server issuing a subsidiary request to a third node. Global policies and careful design are required to avoid deadlock and starvation.

7.5 Network on chip: Switch Fabrics.

A simple ring is not very effective. Instead, richer meshes of elements are used and the elements can have a higher radix, such as 4x4. There are a number of well-known switch wiring schemes, whth names such as Benes, Clos, Shuffle, Delta.

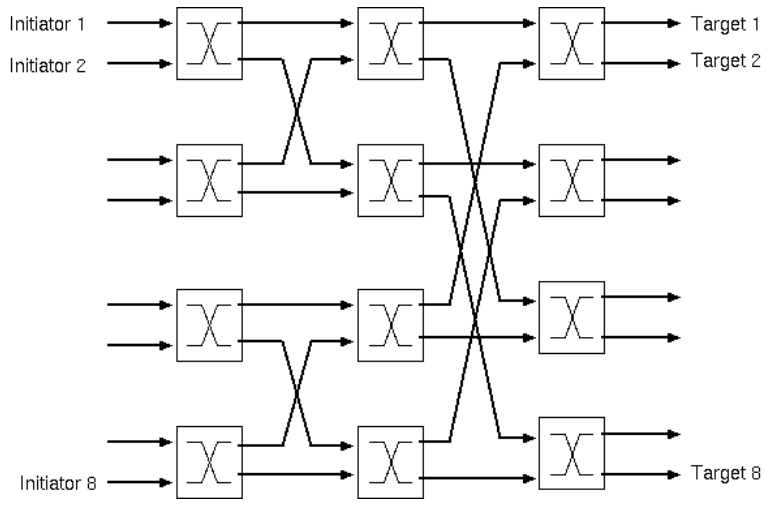


Figure 7.5: A more-complex switching fabric: more wiring, more bandwidth and less contention.

Two-by-two switch element connects eight devices in three stages.

Can use a larger radix: benes, clos, shuffle, delta, (even batcher-banyan).

Problem: typically we will not need quite as many initiators as targets.

The throughput is potentially equal to the number of ports, but the fabric may partially block and there may be uneven traffic flows leading to receiver contention. These effects reduce throughput. Typically will not need quite as many initiators as targets, so a symmetric switch system will be over provisioned.

Can be overly complex on the small scale, but scale up well.

Network On Chip Synthesis Tool: Mullins et al. NetGen Network Generator.

7.6 Network on Chip: Higher Dimensions.

Can we consider higher-dimensional interconnect ?

Hypercube has lowest diameter for number of customers. But it has excessive wiring.

Chips are two-dimensional so perhaps use a 2-D network ? But this may be overly conservative.

Maybe use 2.5-D ? have a small number of 'multi-hop' links?

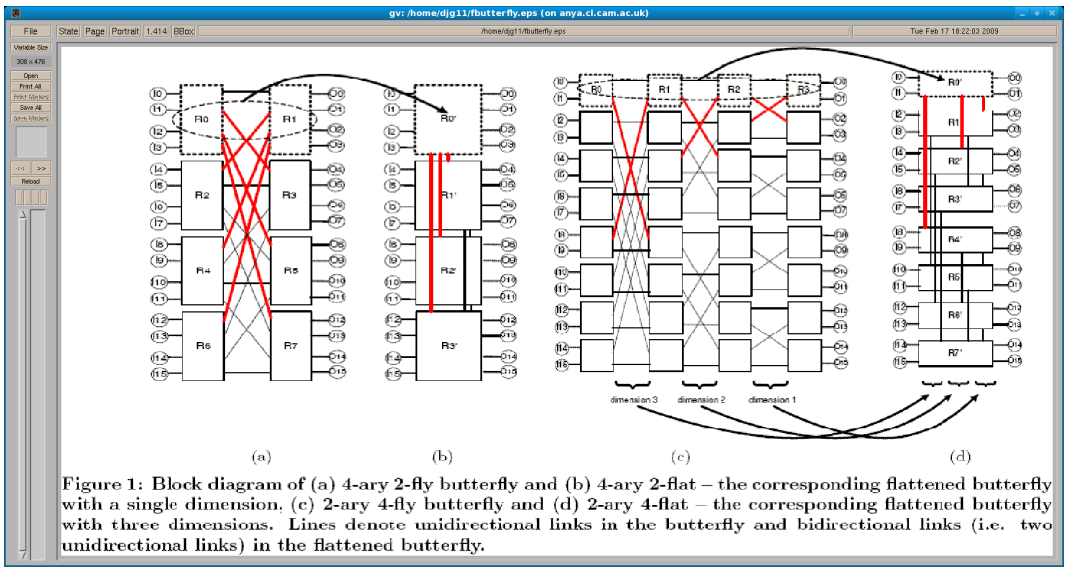


Figure 1: Block diagram of (a) 4-ary 2-fly butterfly and (b) 4-ary 2-flat – the corresponding flattened butterfly with a single dimension. (c) 2-ary 4-fly butterfly and (d) 2-ary 4-flat – the corresponding flattened butterfly with three dimensions. Lines denote unidirectional links in the butterfly and bidirectional links (i.e. two unidirectional links) in the flattened butterfly.

Figure 7.6: The 'Flattened Butterfly' network topology.

On benign (load-balanced) traffic, the flattened butterfly approaches the cost/performance of a butterfly network and has roughly half the cost of a comparable performance clos network.

The advantage over the clos is achieved by eliminating redundant hops when they are not needed for load balance.

Flattened butterfly : a cost-efficient topology for high-radix networks. John Kim, William J. Dally, Dennis Abts

7.7 Practical Bus Protocols on IP Blocks

Many IP blocks today are wired up using OCP's BVCI and ARM's AHB. Although the port on the IP block is fixed, in terms of its protocol, it can be connected to any system of bus bridges and on chip networks.

Open core protocol (OCP): freely available, bus-independent protocol for IP blocks.

Download full OCP documents from OCIP.org. See also bus-protocols-limit-design-reuse-of-ip

- All IP blocks can sport this interface.
- Separate request and response ports.
- Data is valid on overlap of `req` and `ack`.
- Temporal decoupling of directions:
- Allows pipeline delays for crossing switch fabrics or crossing clock domains.
- Sideband signals: interrupts, errors and resets: vary on per-block basis.
- Two complete instances of the port if block is both an initiator and target.
- Large arrows indicate signal directions on initiator.

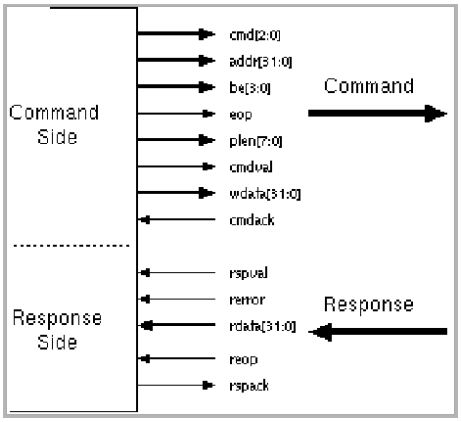


Figure 7.7: Core nets of the BVCI interface, showing its two separate portions. (Arrows indicate net directions on initiator).

The Open Core Protocol (OCP) is a freely available bus-independent protocol. Download full spec from OCPIP. See also D & R article.

A prominent feature is totally separate request and response ports. This makes it highly tolerant of delays over the network and amenable to crossing clock domains. However requests and responses must not get our of order since there is no id token. Older-style protocols where targets had to respond within a prescribed number of clock cycles cannot be used in these situations.

For each half of the port there are request and acknowledge signals, with data being transferred on any positive edge of the clock where both are asserted.

If a block is both an initiator and a target, such as our DMA controller example, then there are two complete instances of the port.

Operations are qualified with conjunction of `req` and `ack`.

Response and acknowledge cycles maintain respective ordering.

Bursts are common. Successive addressing may be implied.

7.8 Other on-chip busses.

The AMBA AHB bus from ARM Cambridge was widely used: but is quite complex (e.g. when resuming from a split burst transaction) and has no temporal decoupling.

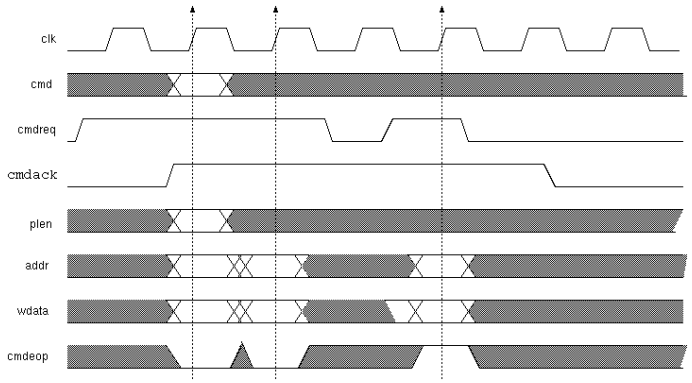


Figure 7.8: BVCI Protocol

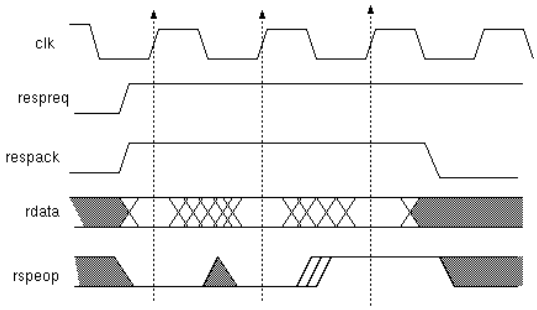


Figure 7.9: BVCI Response Portion Protocol

The BVCI supports temporal decoupling, but requests and responses must not overtake: hence it can cross clock domains and tolerate pipeline stages. But it cannot tolerate out of order responses from, say, a cache or a DRAM.

The ARM AXI bus includes tags on each operation for request/response association: hence it is suitable for pipelined, on-chip networks.

The Wishbone bus and IBM CoreConnect bus: used by various public domain IP bocks and various designs in the OpenCores project. The OR1K in the practical materials on the course web site uses Wishbone. Wikipedia Wishbone Core Connect

The OSCI TLM2.0 generic payload and the GreenSocs bus are higher-level specifications, perhaps with future vision of automatic synthesis of all glue logic?

7.9 Dymanic RAM : DRAM

DRAMs for use in PCs are mounted on SIMMS or DIMMS, but for embedded applications, often just soldered to the main PCB. Normally one bank of DRAM is shared over many sub-systems in, say, a mobile phone. SoC DRAM compatibility might be a generation behind workstation DRAM: e.g. using DDR2 instead of DDR3

Typical DRAM pin connections:

Figure 7.10: DRAM single-in-line memory module (SIMM).

Clk+/-	Clock
Ras-	Row address strobe
Cas-	Column address strobe
We-	Write enable
dq[63:0]	Data in/out
reset	Power on reset
wq[7:0]	Write lane qualifiers
ds[7:0]	Data stobes
dm[7:0]	Data masks
cs-	Chip select
addr[15:0]	Address input
bs[2:0]	Bank select
spd[3:0]	Serial presence detect

High bandwidth: 64 bits times 400 MHz giving 25.6 Gb/s peak. High capacity: Example 1 Gbyte DIMM made of 8 chips. High latency: 20 clock cyles access time to a closed bank. Worse if a bank is already open at the wrong place.

7.10 DRAM Internal Block Diagram

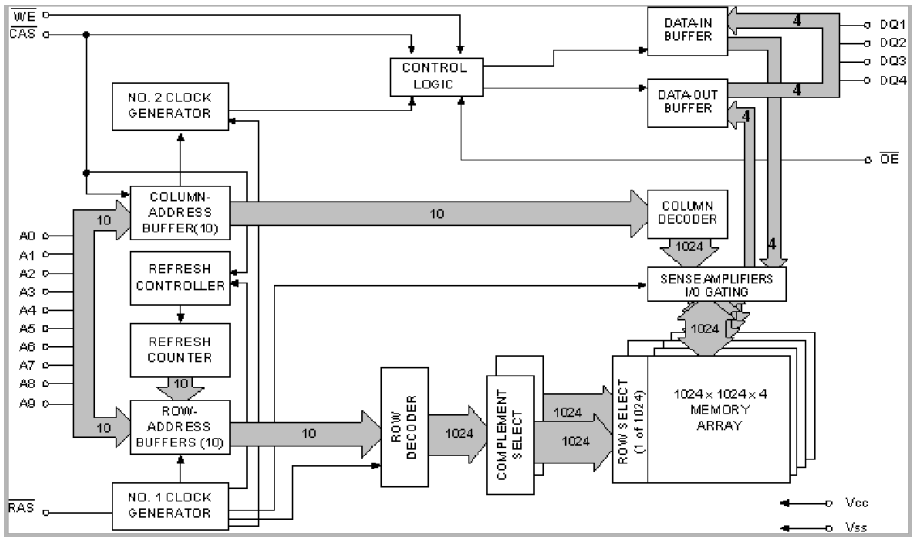


Figure 7.11: DRAM Chip Internal Block Diagram.

This DRAM has four data I/O pins and four internal planes, so no bank select bits. Modern, larger capacity DRAMs have multiple such structures on their die and hence additional bank select inputs select which one is

addressed.

Row and column addresses are supplied successively (giving overall a four part address: bank, row, column and bit lane).

Dynamic RAM keeps data in capacitors. The data will stay there reliably for up to four milliseconds and hence every location must be read out and written back (refresehed) within this period. The data does not need to leave the chip for refresh, just transferred to the edge of its array and then written back again. Hence a whole row of each array is refreshed as a single operation.

DRAM is not normally put on the main SoC chip(s) owing to its specialist manufacturing steps, large area needs and commodity-style marketing. Instead a standard part is put down and wired up.

A row address is first sent to a bank in the DRAM and then one has random access to the columns of that row using different column addresses. The DRAM cells internally have destructive read out because the capacitors get discharged into the row wires when accessed. Therefore, whenever finished with a row, the bank containing it goes busy while it writes back the data and gets ready for the next operation (charing row wires to mid-way voltage etc.).

DRAM is slow to access and certainly not 'random access' compared with on-chip RAM. A modern PC might take 100 clock cycles to access a random part of DRAM, but the ratio is not as severe in typical embedded systems owing to lower system clocks. Nonetheless, we typically put a cache on the SoC as part of the memory controller. The controller may have error correction logic in controller as well.

The cache will access the DRAM in localised bursts, saving or filling a cache line, and hence we arrange for cache lines to lie within DRAM rows.

Modern parts have programmable compensation for differing delays the PCB tracking: set up in a calibrate phase.

May have error correction logic in controller.

The controller may keep multiple banks open at once to exploit tempro-spatial access locality.

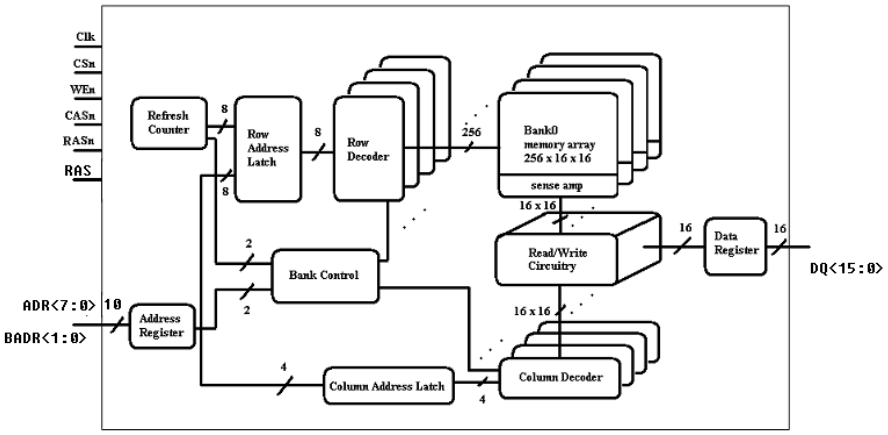


Figure 7.12: Another DRAM chip.

DRAM controller is typically coupled with a cache or at least a write buffer.

DRAM: high latency and write back overhead dictate preference for large burst operations. It is best if clients make available several operations for processing at once: up to number of banks. It is best if clients can tolerate responses out of order.

Controller must

- set up DRAM control register programming,
- calibrate delay lines,
- implement RAS to CAS latencies,
- and ensure refresh happens.

Controller might contain a tiny CPU to interrogate serial device data.

DRAM refresh overhead has minimal impact on bus throughput. For example, if 512 refresh cycles are needed in 4 ms and the cycle rate is 200E6 the overhead is 0.1 percent.

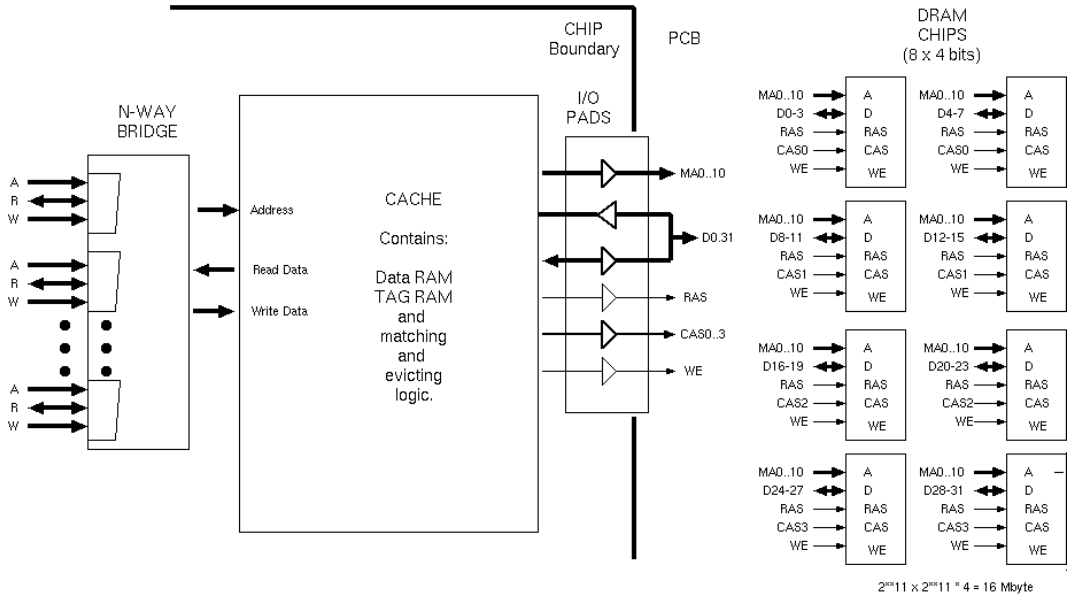


Figure 7.13: Typical structure of a small DRAM subsystem.

Figure 7.13 shows a 32 bit DRAM subsystem. Four CAS wires are used so that byte writes are possible. For large DRAM arrays, need also to use multiple RAS lines to save power by not sending RAS to un-needed destinations.

7.11 Cache Design

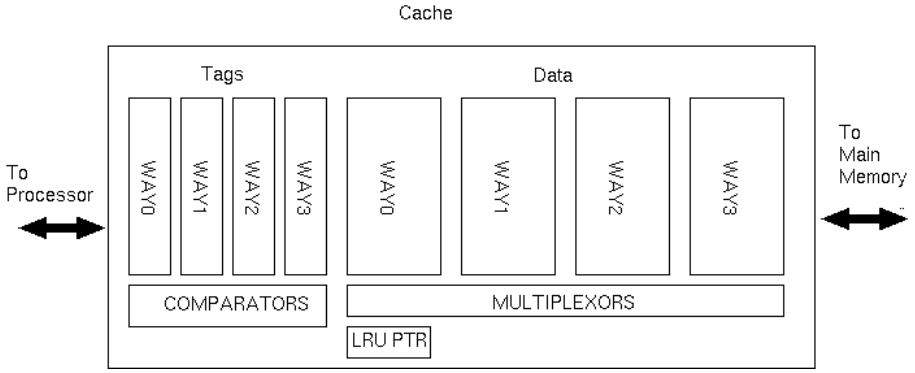


Figure 7.14: Memory blocks and tag comparator needed for a 4-way, set-associative cache.

Implementing 4-way, set-associative cache is relatively straightforward. One does not need an associative RAM macrocell: just synthesise four sets of XOR gates from RTL using the '==' operator!

```
reg [31:0] data0 [0:32767], data1 [0:32767], data2 [0:32767], data3 [0:32767];
reg [14:0] tag0 [0:32767], tag1 [0:32767], tag2 [0:32767], tag3 [0:32767];

always @(posedge clk) begin
      miss = 0;
      if (tag0[addr[16:2]]==addr[31:17]) dout <= data0[addr[16:2]];
      else if (tag1[addr[16:2]]==addr[31:17]) dout <= data1[addr[16:2]];
      else if (tag2[addr[16:2]]==addr[31:17]) dout <= data2[addr[16:2]];
      else if (tag3[addr[16:2]]==addr[31:17]) dout <= data3[addr[16:2]];
      else miss = 1;
      end
```

Of course we also need a write and evict mechanism... (not shown). Rather than implement least recently used (LRU) one tends to do 'random' replacement which can be as simple as using keeping a two bit counter to say which 'way' to evict next.

Comp-arch exercise: add a 'way prediction cache' that avoids the double lookup latency. A way cache records which set was last accessed and optimistically forwards the result from that, giving access times closer to that of a directly-mapped cache, without the aliasing overheads.

7.12 Cache Modelling

A cache can be modelled at various levels of abstraction:

- Not at all - afterall it does not affect functionality,
- Using an estimated hit ratio and randomly adding delay to main memory transactions accordingly,
- Fully modelling the tags and their lookup (while making backdoor access to the main memory for the data),
- Modelling the cache data RAMs as well, thereby generating an accurate transaction sequence on the main memory.

Depending on our needs, we may want to measure the hit ratio in the I or D caches, or the effect on performance from the misses, or neither, or all such metrics. [Virtutec Simics.]

An instruction cache (I-cache), when modelled, may or may not be accessed by an emulator or instruction set simulator (ISS). For instance, the ISS may use backdoor access to the program in main memory, or it might use JIT (just-in-time) techniques where commonly executed inner loops of emulated code are converted to native machine code for the modelling workstation.

LG 8 — SoC Engineering and Associated Tools

In this section we look at engineering aspects and associated tools used in SoC design and modelling. A lot of the effort is dedicated to maximising performance and minimising power dissipation.

8.1 Static Timing Analyser Tool

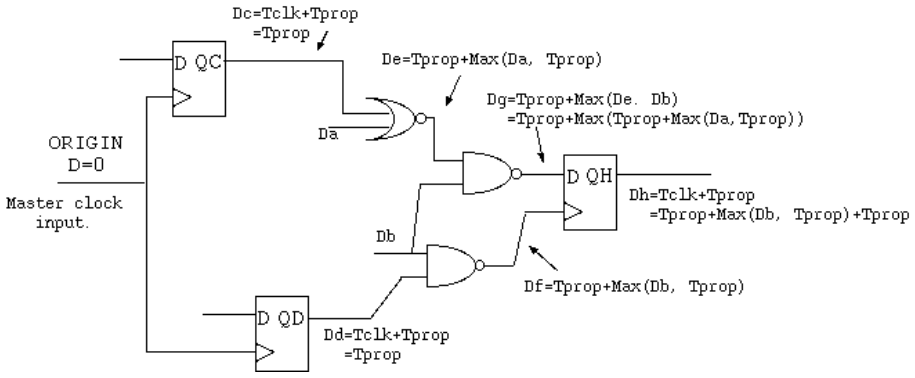


Figure 8.1: An example circuit with static timing annotations

A static timing analyser computes the longest event path through logic gates and clock-to-Q paths of edge-triggered flops.

The longest path is generally the critical path that sets the maximum clock frequency. However, sometimes this is a false result, since this path might never be used during device operation.

Starting with some reference point, taken as D=0, such as the master clock input to a clock domain, we compute the relative delay on the output of each gate and flop.

For a combinational gate, the output delay is the gate's propagation time plus the maximum of its input delays.

For an edge-triggered flop, such as a D-type or a JK, there is no event path to the output from the D or JK inputs, so it is just the clock delay plus the flop's clock-to-Q delay.

There are event paths from asynchronous flop inputs however, such as preset, reset or transparent latch inputs.

Propagation delays may not be the same for all inputs to a given output and for all directions of transition. For instance, on deassert of asynchronous preset to a flop there is no event path. Therefore, may typically keep separate track of high-to-low and low-to-high delays.

8.2 RAM Macrocell Compiler Tool

The average SoC is 71 percent RAM memory. The RAMs are typically generated by a RAM compiler. The input parameters are:

- Size: Word Length and Number of Words.
- Port description: Each port has an address input and is one of r, w, r/w.
- Clocking info: Frequency, latency, or access time for asynchronous RAM.

- Resolution: What to do on write/write and write/read conflicts.

The outputs are a datasheet for the RAM, high and low detail simulation models and something that turns into actual polygons in the fabrication masks.

```
// Example low-level model for a RAM
module R1W1RAM(din, waddr, clk, wen, raddr, dout);
  input clk, wen;
  input [14:0] waddr, raddr;
  input [31:0] din;
  output [31:0] dout;

  reg [31:0] myram [32767:0];  // 32K words of 32 bits each.
  always @(posedge clk) begin
       dout <= myram[raddr];
       if (wen) myram[waddr] <= din;
       end
```

```
// Example high-level model for a RAM
SC_MODULE R1W1RAM()
{
   uint32_t myram [32768];
   int readme(int A) { return myram[A]; }
   writeme(int A, int D) { myram[A] = D; }
}
```

Sometimes self test modules are also generated. For example Mentor's MBIST Architect(TM) generates an SRTL BIST with the memory and ARM/Artisan's Generator will generate a wrapper that implements self repair of the RAM by diverting access from a fault row to a spare row. ARM Artisan

Other related generator tools can be provided: e.g. a FIFO generator would be similar and a masked ROM generator or PLA generator.

8.2.1 Dynamic Clock Gate Insertion Tool

Clock trees consume quite a lot of the power in an ASIC and considerable savings can be made by turning off the clocks to small regions. A region of logic is idle if all of the flip-flops are being loaded with their current contents, either through synchronous clock enables or just through the nature of the design.

Instead of using synchronous clock enables, current design practice is to use a clock gating insertion tool that gates the clock instead.

Care must be taken not to generate glitches on the clock as it is gated and transparent latches in the clock enable signal can re-time it to prevent this.

How to generate clock enable conditions ? One can have software control (additional control register flags) or automatically detect. Automatic tools compute 'clock needed' conditions. A clock is 'needed' if any register will change on a clock edge.

A lot of clock needed computation can get expensive, resulting in no net saving, but it can be effective if computed once at head of a pipeline.

Beyond just turning off the clock or power to certain regions, in LG8 we look at further power saving techniques: dynamic frequency and voltage scaling.

8.3 Test Program Generator Tool

A test program generator works out a short sequence of tests that will reveal 'stuck-at' and other faults in a subsystem.

Not lectured in 2009/10.

8.4 Scan Path Insertion and JTAG standard test port.

A scan path insertion tool replaces the user's D-type flip-flops with a scan path, connected to the external JTAG test access port for post-fabrication testing.

Not lectured in 2009/10.

8.5 Architectural Exploration and Design Partition

A collection of algorithms and functional requirements must be implemented using one or more pieces of silicon. Each major piece of silicon contains one or more custom or standard microprocessors. Some silicon is custom for a high-volume product, some is shared over several product lines and some is third party or standard parts.

Design Partition: Deciding on the number of processors, number of custom processors, and number of custom hardware blocks.

Co-design and co-synthesis: two basic methods (can do different parts of the chip differently):

- Co-design: Manual partition between hardware and software,
- Co-synthesis: Automatic partition: simple 'device drivers' are created automatically:

Co-synthesis not currently used in practice.

(MPEG alogorithm)

8.6 H/W to S/W Interfacing Techniques

The primary ways of connecting H/W to S/W are:

- Programmed I/O to pin-level PIO register,
- Programmed I/O to FIFOs,
- Interrupts (hardwired or dynamically dispatched),
- Packet channel mapped into register file,
- DMA,
- Psudo-DMA (processor generates addresses only.

Dissected Cellphone: Motorola e770V Physical components:

- Display (touch sensitive) + Keypad + Misc buttons
- Audio ringers and speakers, microphone(s) (noise cancelling),
- Infra-red IRDA port

- Multi-media codecs (A/V capture and replay in several formats)
- Radio Interfaces: GSM (three bands), BlueTooth, 802.11.
- Power Management: Battery Control, Processor Speed, on/off/flight modes.
- Camera,
- Memory card slot,
- Physical connectors: USB, Power, Headset,
- Java VM and operating system.

Diverse IP Suppliers at chip and *Soctronics* level.

8.7 H/W Design Partition

A number of separate pieces of silicon are combined to form the product. Reasons for H/W design partition:

- Modular Engineering At Large (Revision Control/Lifetime/Sourcing/Reuse),
- Size and Capacity (chips 6-11 mm in size),
- Technology mismatch (Si/GaAs/HV/Analog/Digital/RAM/DRAM/Flash)
- Supply chain: In-house versus Standard Part.
- Isolation of sensitive RF signals,
- Cost: a new chip spin of old IP is still expensive.

8.8 Chip Types and Classifications

Chips can be classified by function: Analog, Power, RF, Processors, Memories, Commodity: logic, discretes, FPGA and CPLD, SoC/ASIC, Other high volume (disk drive, LCD, ...).

Manufacturers can be classified as well:

1. Major chip makers such as IBM and Intel that design, manufacture and sell their chips (Integrated Device Manufacturers / IDM).
2. Fabless manufacturers such as NVIDIA and Xilinx that design and sell chips but outsource manufacturing to foundry comp anies.
3. Foundry companies (such as TSMC and UMC) that manufacture chips designed and sold by their customers.

The world's major foundries are SMC and TSMC: Taiwan Semiconductor Manufacturing Company Limited

Figure 8.2 presents a taxonomy of chip design approaches. The top-level division is between standard parts, ASICs and field-programmable parts. Where a standard part is not suitable the choice between full-custom and semi-custom and field-programmable approaches has to be made, depending on performance, production volume and cost requirements.

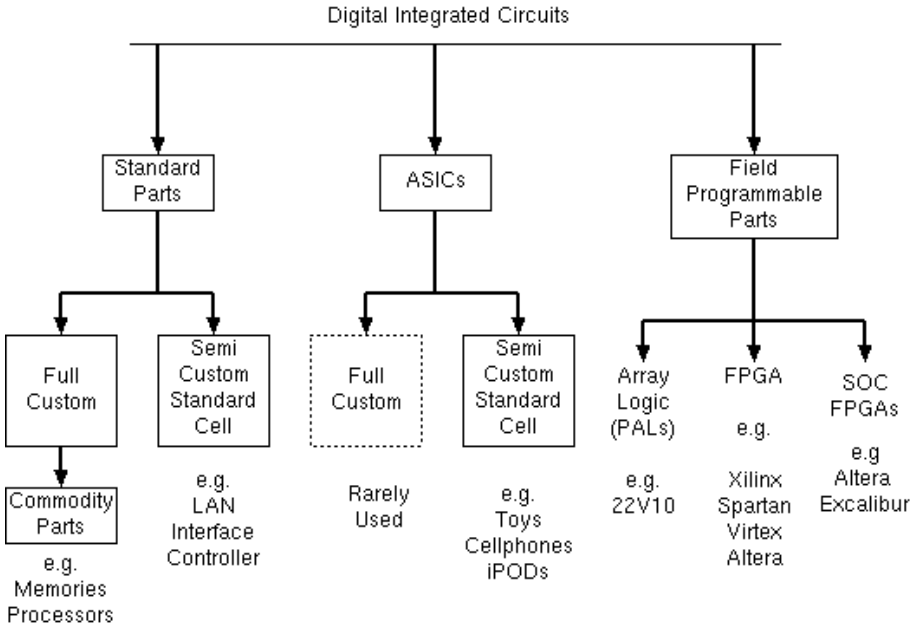


Figure 8.2: A taxonomy of integrated circuits.

8.8.1 Standard Parts

A **standard part** is essentially any chip that a chip manufacturer is prepared to sell to someone else along with a datasheet and EDA models. The design may actually previously have been an ASIC for a specific customer that is now on general release. However, most standard parts are general-purpose logic, memory and microprocessor devices. These are frequently full-custom designs designed in-house by the chip manufacturer to make the most of in-house fabrication line, perhaps using optimisations not made available to others who use the line as a foundry. Other standard parts include graphics controllers, LAN controllers, bus interface devices, and miscellaneous useful chips.

8.8.2 Masked ASICs.

A masked ASIC (application specific integrated circuit) is a device manufactured for a customer involving a set of masks where at least some of the masks are used only for that device. These devices include full-custom and semi-custom ASICs and masked ROMs.

A full-custom chip (or part of a chip) has had detailed manual design effort expended on its circuits and the position of each transistor and section of interconnect. This allows an optimum of speed and density and power consumption.

Full-custom design is used for devices which will be produced in very large quantities: e.g. millions of parts where the design cost is justified. Full-custom design is also used when required for performance reasons. Microprocessors, memories and digital signal processing devices are primary users of full-custom design.

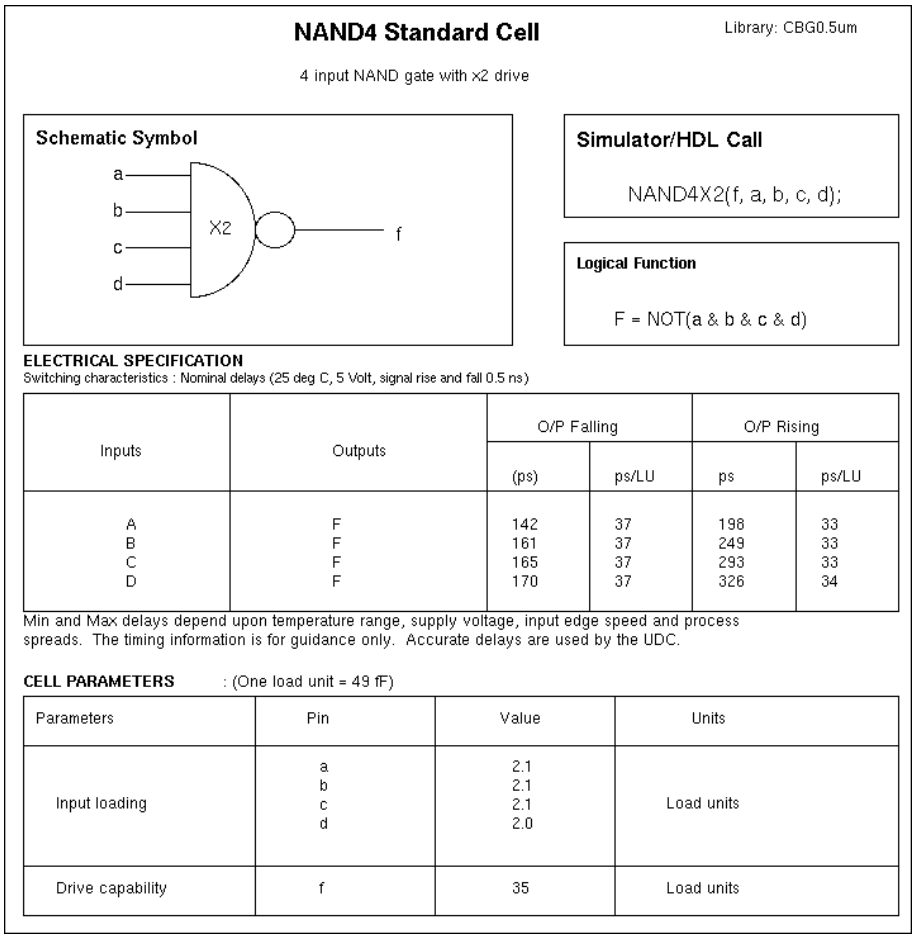
In semi-custom design, each cell has a fixed design and is repeated each time it is used, both within a chip and across many devices which have used the library. This simplifies design, but drive power of the cell is not optimised for each instance.

Semi-custom is achieved using a library of logic cells and is used for general-purpose VLSI design.

8.9 Semi-custom (cell-based) Design

A library of standard logic functions is provided. Cells are placed on the chip and wired up by the user, in the same way that chips are placed on the PCB.

- Standard Cell - free placement and free routing of nets,
- Gate Array - fixed placement, masked or electrical programmable wiring.



NAND4 Standard Cell

Library: CBG0.5um

4 input NAND gate with x2 drive

Schematic Symbol

Simulator/HDL Call

NAND4X2(f, a, b, c, d);

Logical Function

F = NOT(a & b & c & d)

ELECTRICAL SPECIFICATION

Switching characteristics : Nominal delays (25 deg C, 5 Volt, signal rise and fall 0.5 ns)

Inputs	Outputs	O/P Falling		O/P Rising	
		(ps)	ps/LU	ps	ps/LU
A	F	142	37	198	33
B	F	161	37	249	33
C	F	165	37	293	33
D	F	170	37	326	34

Min and Max delays depend upon temperature range, supply voltage, input edge speed and process spreads. The timing information is for guidance only. Accurate delays are used by the UDC.

CELL PARAMETERS : (One load unit = 49 fF)

Parameters	Pin	Value	Units
Input loading	a b c d	2.1 2.1 2.1 2.0	Load units
Drive capability	f	35	Load units

Figure 8.3: Typical cell data sheet from a standard cell library.

The figure shows a cell from the data book for a standard cell library. This device has twice the 'normal' drive power, which indicates one of the compromises implicit in standard cell over full-custom, which is that the size (driving power) of transistors used in a cell is not tuned on a per-instance basis.

Historically, there were two types of semi-custom devices:

- standard cell (for high volume)
- gate array (for volume less than say 5000 parts).

but now the mask-programmed gate array has been replaced with the field-programmed FPGA.

In standard cell designs, cells from the library can freely be placed anywhere on the device and the number of IO pads and the size of the die can be freely chosen. Clearly this requires that all of the masks used for a device are unique to that device and cannot be used again. Mask making is one of the largest costs in chip design.

8.10 Gate Arrays and Field-Programmable Logic.

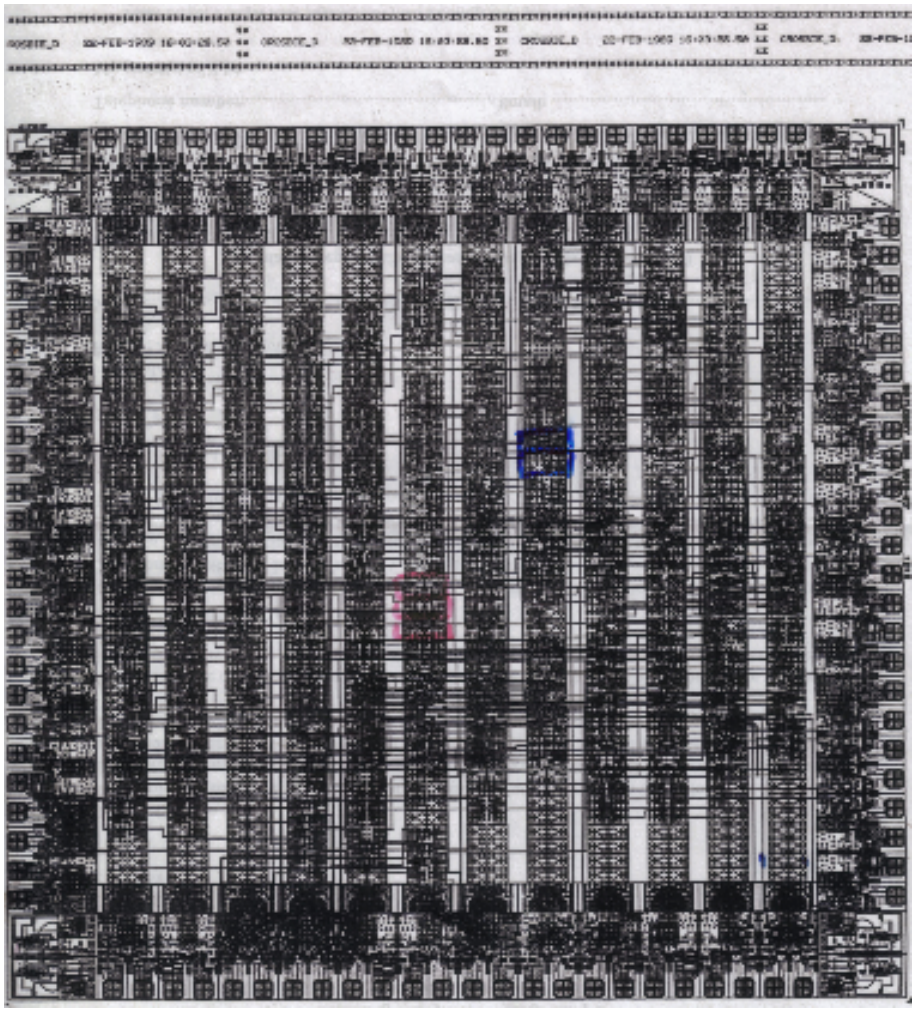


Figure 8.4: A Gate Array I Designed: (Backbone Ring ECL Gate Array)

Figure 8.4 reveals the regular layout of a masked gate array showing bond pads around the edge and wasted silicon area (white patches).

A gate array comes in standard die sizes containing a fixed layout of configurable cells. Historically, there were two main forms of gate array:

- Mask Programmable,
- Field Programmable (FPGA).

In gate array designs, the silicon vendor offers a range of chip sizes. Each size of chip has a fixed layout and the location of each transistor, resistor and IO pad is common to every design that uses that size. Gate arrays are configured for a particular design by wiring up the transistors, gates and other components in the desired way. Many cells will be unused. For mask-programmed devices, the wiring up was done with the top two or three layers of metal wiring. Therefore only two or three custom masks were needed be made to make a new design. In FPGAs the programming is purely electronic (RAM cells control pass transistors).

The disadvantage of gate arrays is their intrinsic low density of active silicon.

Standard cell designs use a set of well-proven logic cells on the chip, much in the way that previous generations of standard logic have been used as board-level products, such as Texas Instruments' System 74.

A variation on the gate array is to include full-custom macrocells such as processor cores in fixed positions on the die.

About 25 to 40 percent of chip sale revenue now comes from field programmable logic devices. These are chips which can be programmed electronically on the user's site to provide the desired function. The Xilinx FPGA parts used in the Part 1B E+A classes are one of the most important examples of field-programmable logic.

Field-programmable devices may be volatile (need programming every time after power up), reprogrammable or one-time programmable. This depends on how the programming information is stored inside the devices, which can be in RAM cells or in any of the ways used for ROM, such as electrostatic charge storage (e.g. FLASH).

Except for niche applications FPGAs are now always used instead of masked gate arrays.

8.11 FPGA - Field Programmable Gate Array

Currently DJ Greaves is using the the large Xilinx XC5VLX110T. There are four of these on the BEE3 Boards. Smaller FPGAs are used in most applications however.

Technical/Catalog Information	XC5VLX110T-2FFG1136C
Vendor	Xilinx Inc
Category	Integrated Circuits (ICs)
Number of Gates	110000
Number of I /O	640
Number of Logic Blocks/Elements	8640
Package / Case	1136-FCBGA
Operating Temperature	0C 85C
Voltage - Supply 1.14 V 3.45 V	

65 nm technology, 6-input LUT, 64 bit DP RAMs.

An FPGA (field-programmable gate array) consists of an array of configurable logic blocks (CLBs), as shown in Figure 8.5. Not shown is that the device also contains a good deal of hidden logic used just for programming it. Some pins are also dedicated to programming. Such FPGA devices have been popular since about 1990.

Each CLB (configurable logic block) (Figure 8.6) typically contains one or two flip-flops, and has a few (five shown) general purpose inputs, some special purpose inputs (only a clock is shown) and two outputs. The illustrated CLB is of the look-up table type, where the logic inputs index a small section of pre-configured RAM memory that implements the desired logic function. For five inputs and one output, a 32 by 1 SRAM is needed. Some FPGA families now give the designer write access to this SRAM, thereby greatly increasing the amount of storage available to the designer. However, it is still an expensive way to buy memory.

All CLBs within a FPGA generally have the same structure, but FPGAs are available with lower and higher functionality CLBs. The best size of CLB is not yet clear. Some designs of FPGA have a hierarchy of CLB interconnection patterns, giving CLB clusters within clusters. Most designs support special paths for fast carry adders and multiplier structures.

An FPGA is very like a mask-programmed gate array to use. The design flow and CAD tools are virtually identical. However, the expenditure before the designer has the first device in her hands might be 1000 times lower. The cost of further devices used to be at least 10 times higher than mask-programmed devices, owing to the programming cost and wasted die area devoted to the programming activities. However, modern mask costs make the mask-programmed gate array unattractive.

FPGAs tend to be quite slow, owing to larger die areas than an ASIC equivalent and because the signals pass through hidden logic used only for configuration.

Sometimes a company would build prototypes and early production units using FPGAs and then use a drop-in mask-programmed equivalent once the design is mature and sales volumes are very large.

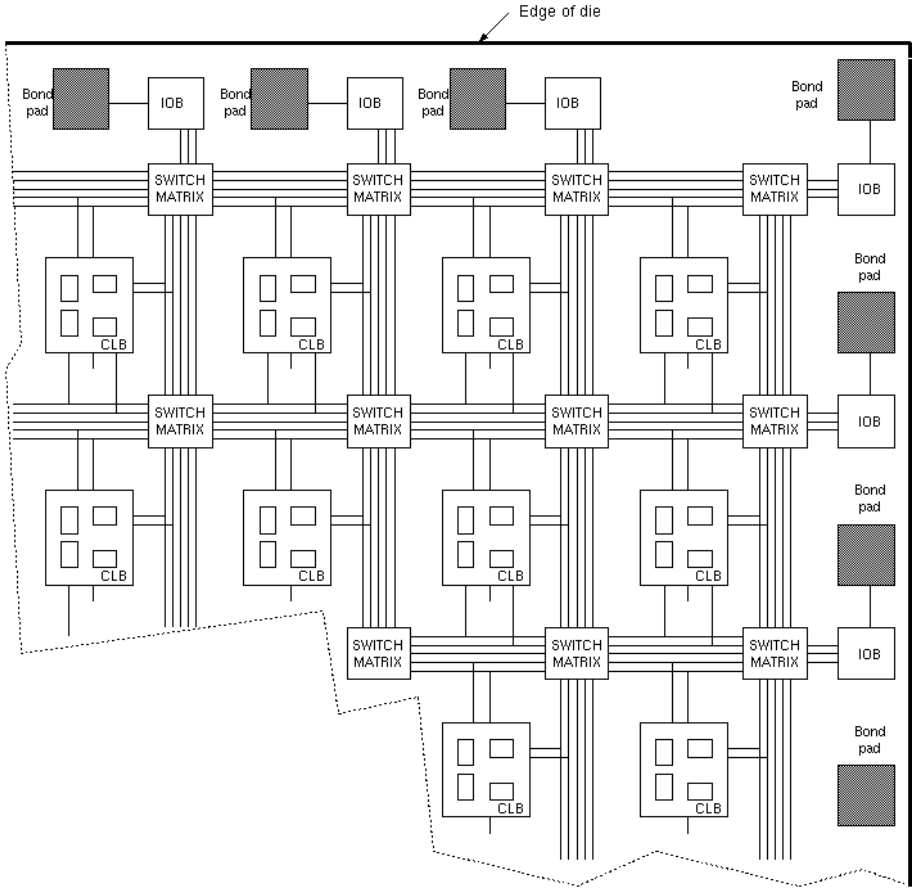


Figure 8.5: Field-programmable gate array structure, showing IO blocks around the edge, interconnection matrix blocks and configurable logic blocks.

8.12 PALs and CPLDs

PALs are Programmable Array Logic and CPLDs (Complex Programmable Logic Devices) achieve very low delay in return for simple, nearly fixed, wiring structure. All expressions are expanded to SOP form with limited number of products. Expanding to sum-of-products form can cause near-exponential area growth (e.g. ripple carry converted to fast carry).

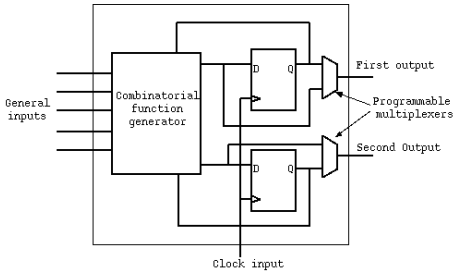


Figure 8.6: A configurable logic block for a look-up-table based FPGA.

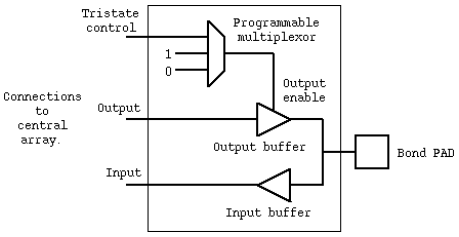


Figure 8.7: A simple IO block for an FPGA.

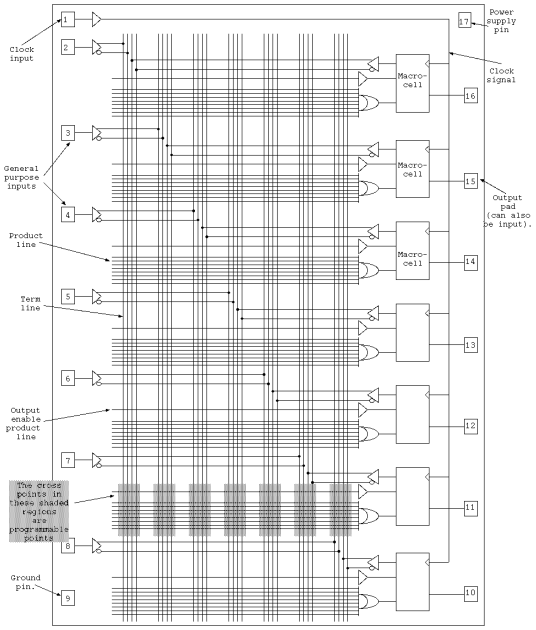


Figure 8.8: A typical PAL with 7 inputs and 7 I/Os.

```
pin 16 = o1;
pin 2 = a;
pin 3 = b;
pin 4 = c

o1.oe = ~a;
o1 = (b&o1) | c;

-x-- ---- ---- ---- ---- ---- ----  (oe term)
--x- x--- ---- ---- ---- ---- ----  (pin 3 and 16)
---- ---- x--- ---- ---- ---- ----  (pin 4)
xxxx xxxx xxxx xxxx xxxx xxxx xxxx
xxxx xxxx xxxx xxxx xxxx xxxx xxxx
xxxx xxxx xxxx xxxx xxxx xxxx xxxx
xxxx xxxx xxxx xxxx xxxx xxxx xxxx
xxxx xxxx xxxx xxxx xxxx xxxx xxxx
x                                   (macrocell fuse)
```

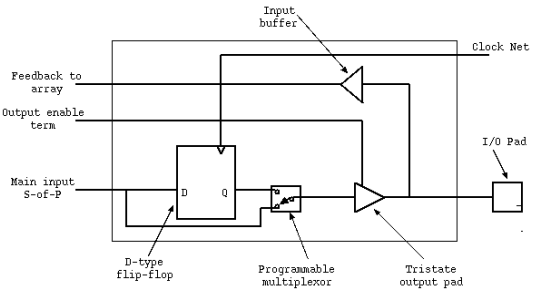


Figure 8.9: Contents of the example PAL macrocell.

A PAL is programmable array logic device. Figure 8.8 shows a typical device. Such devices have been popular since about 1985. The illustrated device has 8 product terms per logic function, and so can support functions of medium complexity. Such devices are very widely used and can feature high speed operation with clock rates of above 100 MHz. They are really just highly structured gate arrays. Every logic function must be multiplied out into sum-of-products form and hence is achieved in just two gate delays.

Programmable *macrocells* (Figure 8.9) enable the output functions to be either registered or combinatorial. Small devices (e.g. with up to 10 macrocells) offer one clock input; larger devices with up to about 100 macrocells are also available, and generally offer several clock options. Often some macrocells are not actually associated with a pin, providing a so called *buried state* flip-flop.

Mini design example: As entered by a designer in a typical PAL language, and part of the fuse map that would be generated by the PAL compiler. Each product line has seven groups of four fuses and produces the logical AND of all of the signals with intact fuses. An 'x' denotes an intact fuse and all of the fuses are left intact on an unused product lines in order to prevent the line ever generating a logical one (a gets ANDed with abar etc.). The fuse map is loaded into a programming machine (in a file format known as JEDEC), an unused PAL is placed in the machine's socket and the machine programs the fuses in the PAL accordingly.

PALs achieve their speed by being highly structured. Their applicability is restricted to small finite state machines and other *glue logic* applications.

8.13 H/W versus S/W Design Partition Principles

The cost of developing an ASIC has to be compared with the cost of using an existing part. The existing part may not perform the required function exactly, requiring either a design specification change, or some additional *glue logic* to adapt the part to the application.

More than one ASIC may be needed under any of the following conditions:

- application specific functions are physically distant
- application specific functions require different technologies
- application specific functions are just too big for one ASIC
- it is desired to split the cost and risk or reuse part of the system later on.

Factors to consider on a per chip basis:

- power consumption limitation (powers above 5 Watts need special attention)
- die size limitation (above 11 mm on a side might escalate cost per mm^2)
- speed of operation — clock frequencies above 1 GHz raise issues
- special considerations :
  - special static or dynamic RAM needs
  - analogue parts - can these also be integrated onto the ASIC ?
  - high power/voltage output capabilities for load control: e.g. motors.
- availability of a developed module for future reuse.

Many functions can be realised in software or hardware. Decide what to do in hardware:

- Physical I/O (line drivers/transducers/media interfaces),

- Highly compute-intensive, fixed functions,

what to do on custom processors:

- Bit-oriented operations,
- Highly compute-intensive SIMD,
- Other algorithms with custom data paths,
- Algorithms that might be altered post tape out.

and what to do in S/W on standard cores:

- Highly-complex, non-repetitive functions,
- Low-throughput computations of any sort,
- Functions that might be altered post tape out.
- As much as possible.

When designing a sub-system we must choose what to have as hardware, what to have as software and whether custom or standard processors are needed. When designing the complete SoC we must think about sharing of sub-system load over processors. Example: if we are designing a digital camera, how many processors should it have and can the steadicam and motion estimation processing be done in software ?

- The functions of a system can be expressed in a programming language or similar form and this can be compiled fully to hardware or left partly as software
- Choosing what to do in hardware and what to do in software is a key decision. Hardware gives speed (throughput) but software supports complexity and flexibility.
- Partitioning of logic over chips or processors is motivated by interconnection bandwidth, raw processing speed, technology and module reuse.

8.14 Legacy H/W S/W Design Partition

In the past (ninteen-eightees), it was best to use a standard processors as a separate chip. Today, it is no problem to put down one or more 'standard' processors on a SoC. It is also quite easy to design your own, so MIPS, Tensilica, ARM and other CPU core providers have to compete against in-house design teams. For instance, we use the the totally free OR 1000 in the practical materials of this course.

8.15 An old example example: The Cambridge Fast Ring two chip set.

Two devices were developed for the CFR local-area network (1983), illustrating the almost classical design partition required in high-speed networking. They were never given grander names than the *ECL* chip and the *CMOS* chip. The block diagram for an adaptor card is shown in the Figure 8.11.

The ECL chip clocked at 100 MHz and contained the minimal amount of logic that needed to clock at the full network clock rate. Its functions were:

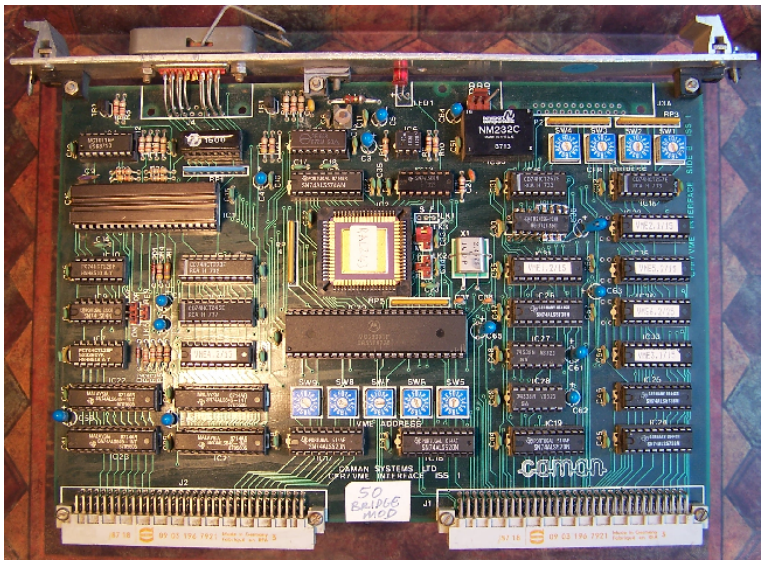


Figure 8.10: The two-chip CFR set using PALs as glue logic for the VME bus.

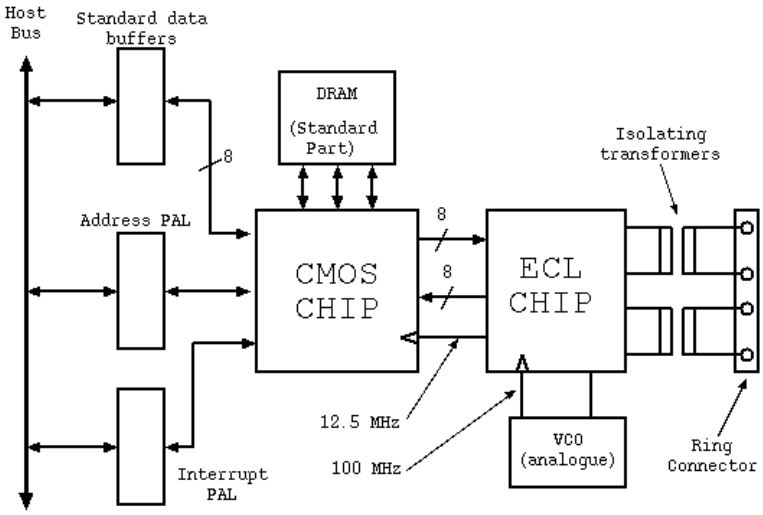


Figure 8.11: Example of a design partition — the adaptor card for the Cambridge Fast Ring.

- implement serial transmission modulator and demodulator,
- convert from 1 bit wide to 8 bits wide and the other way around,
- perform reception byte alignment (when instructed by logic in the CMOS chip).

Other features:

- ECL logic can support analogue line receivers at low additional cost so can receive the incoming signal directly on to the chip.
- ECL logic has high output power if required (1 volt into 25 ohms) and so can drive outgoing twisted pair lines directly.

The CMOS chip clocks at one eighth the rate and handles the complex logic functions:

- CRC generation

- full/empty bit protocol
- minipacket storage in on-chip RAM
- host processor interface
- ring monitoring and maintenance functions.

The ECL chip had at least 50 times the power consumption of the CMOS chip. The CMOS chip had more than 50 times the gates of the ECL chip. Rolling forward to 2010, we might make a similar design partition with a high-performance bipolar subsystem clocking at 4 GHz connected to a CMOS 'baseband' component running where some small parts operating at 500 MHz and the remainder at 250 MHz.

Standard parts were used to augment the CFR set: the DRAM chip incorporates a dense memory array which could not have been achieved for anywhere near the same cost onboard the CMOS chip and the VCO (Voltage Controlled Oscillator) device used for clock recovery was left off the ECL chip since it was a difficult-to-design analogue component where the risk of having it on the chip was not desired.

PALs are used to 'glue' the network interface itself to a particular host system bus. Only the glue logic needs to be redesigned when a new machine is to be fitted with the chipset. PALs have a short design turn-around time since they are field-programmable.

For a larger production run, the PALs would be integrated onto a custom variant of the CMOS chip.

8.16 Partitioning example: An external RS-232/POTS Modem.

Figure 8.12: A POTS modem.

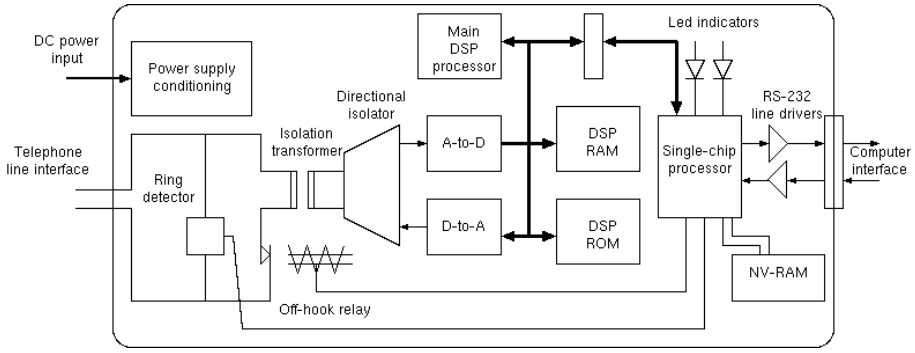


Figure 8.13: Example of a design partition — internal structure of the original modem.

Figure 8.13 shows the block diagram of a typical modem circa 1985. The illustrated device is an external modem, meaning that it sits in a box beside the computer and has an RS-232 serial connection to the computer. It also requires its own power supply.

The device contains a few analog components which behave broadly like a standard telephone, but most of it is digital. A relay is used to connect the device to the line and its contacts mirror the 'off-hook' switch which is part of every telephone. It connects a transformer across the line. The relay and transformer provide isolation of the computer ground signal from the line voltages. Similarly the ringing detector often uses a opto-coupler to provide isolation. *Clearly, these analog aspects of the design are particular to a modem and are designed by a telephone expert.*

Modems from the 1960's implemented everything in analog circuitry since microprocessors and DSP were not available. In 1985, two microprocessors were often used.

Note that the non-volatile RAM required (and still does) a special manufacturing processing step and so is not included as a resource on board the single-chip processor. Similarly, the RS-232 drivers need to handle voltages of +/- 12 volts and so these cannot be included on chip without increasing the cost of the rest of the chip by using a fabrication process which can handle these voltages. The NV-RAM is used to store the owner's settings, such as whether to answer an incoming call and what baud rate to attempt a first connection, etc..

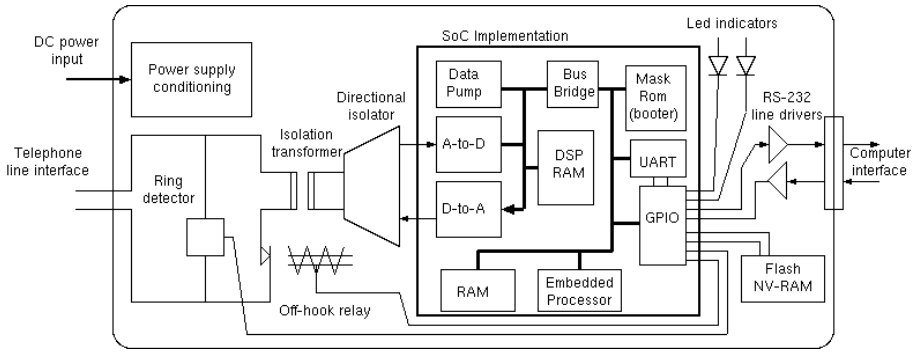


Figure 8.14: Typical structure of the modem product today (using a SoC approach).

A modern implementation would integrate all of the RAM, ROM, ADC and DAC and processors on a single SoC. The RS-232 remains off chip owing to 24 volt and negative supply voltages whereas the SoC itself may be run at 3.3 volts. The NV store is a large capacity Flash ROM device with low-bandwidth serial connection. At system boot, the main code for both processors is copied from the Flash to the two on-chip RAMS by the small, mask-programmed booter. Keeping the firmware in Flash allows the modem to be upgraded to correct bugs or encompass new communications standards.

GPIO is used for all of the digital I/O, with the UART transmit and receive paths being set up as special modes of two of the GPIO connections.

8.17 Partitioning example: A Bluetooth Module.

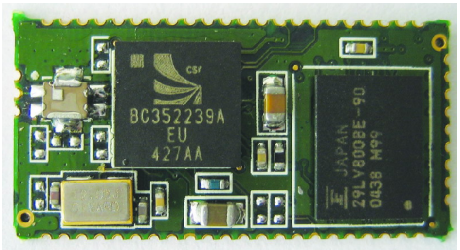


Figure 8.15: Broadcom (Cambridge Silicon Radio) Bluetooth Module circa 2000.

An initial implementation of the Bluetooth radio was made of three pieces of silicon bonded onto a small fibreglass substrate...

An initial implementation of the Bluetooth radio was made of three pieces of silicon bonded onto a small

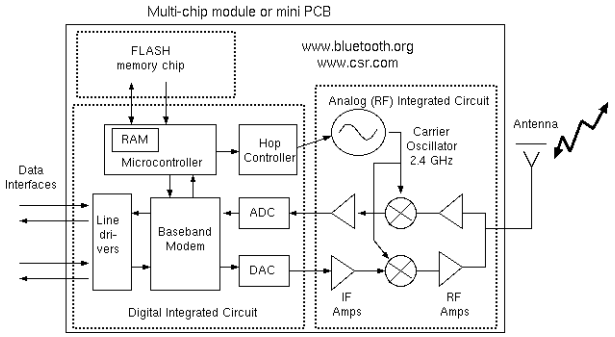


Figure 8.16: Example of a design partition — Block diagram of Bluetooth radio module (circa 2000).

fibreglass substrate with overall area of 4 square centimetres.

The module was partitioned into three pieces of silicon partly because the overall area required would give a low yield, but mainly because the three sections used widely different types of circuit structure.

The analog integrated circuit contained amplifiers, oscillators, filters and mixers that operate in the 2.4 GHz band. This was too fast for CMOS transistors and so bipolar transistors with thin bases were used. The module amplifies the radio signals and converts them using the mixers down to an intermediate frequency of a few MHz that can be processed by the ADC and DAC components on the digital circuit.

The digital circuit had a small amount of low-frequency analog circuitry in its ADC and DACs and perhaps in its line drivers if these are analog (e.g. HiFi). However, it was mostly digital, with random logic implementations of the modem functions and a microcontroller with local RAM. The local RAM holds a system stack, local variables and temporary buffers for data being sent or received.

The FLASH chip is a standard part, non-volatile memory array that can hold firmware for the microcontroller, parameters for the modem and encryption keys and other end application functions. The flash memory is a standard 29LV800BE (Fujitsu) - 8m (1m X 8/512 K X 16) Bit

Today, the complete Bluetooth module can be implemented on one piece of silicon, but this still presents a major technical challenge owing to the diverse requirements of each of the sub-components.

8.18 Cell Library Tour

In the lecture we will have a look at the following documents: A cell library in the public domain: TANNER AMIAnother VLSI TECHAnother Mosis 0.5 u Cell Library

Things to note: there's a good variety of basic gates, including quite a few multi-level gates, such as AND-OR gate. There's also I/O pads, flip-flops and special function cells. Many gates are available with various output powers.

For each gate there are comprehensive figures that enable one to predict its delay, taking into account its track loading, how many other gates it is feeding and the current supply voltage.

8.19 Silicon Power and Technology

Switching speed is dominated by electron mobility (drift velocity) in transistor gates. We can improve by shifting to faster materials, such as GaAs, or just by making the gates smaller. How small can we go: what is the *silicon end point* ?

Rule of thumb: the product of delay and power consumption of a gate is largely constant, leading to a design

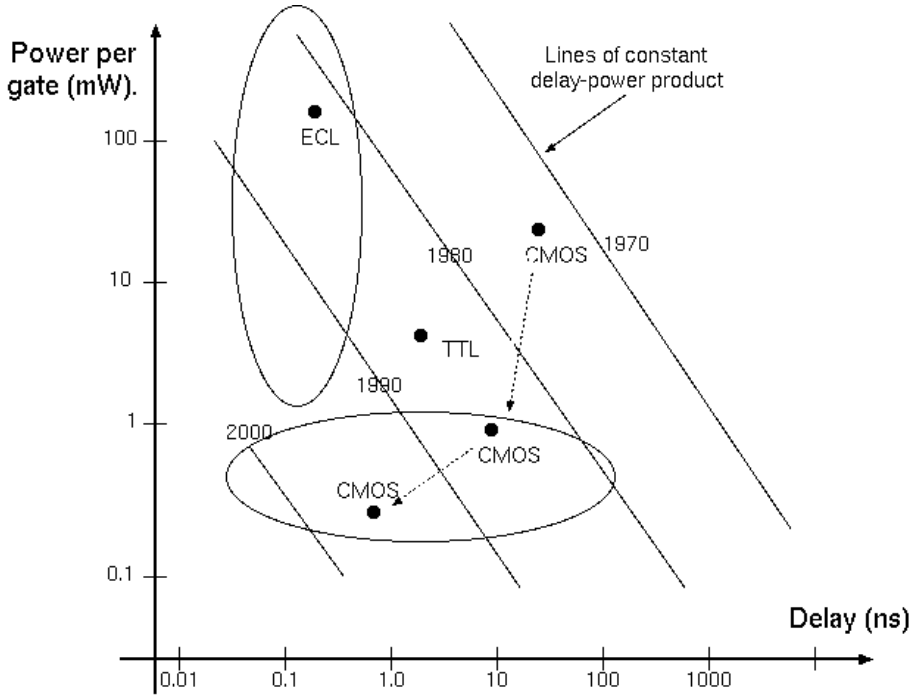


Figure 8.17: Energy used when a gate switches: delay-power product.

trade off. (Also called the speed-power product). Units are the Joule: the energy for a logic transition in the gate.

Total consumption = Gate Power + Wiring Power.

Electric charge in the wiring nets is proportional to their capacitance and hence their length and width.

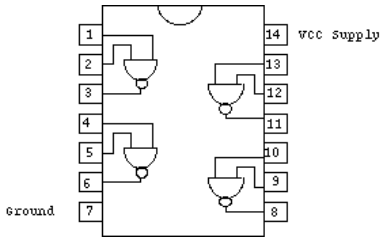


Figure 8.18: The 7400 standard part: has been in manufacture using this pinout for 40 or so years.

At any one time, there is a choice of implementation technologies. Here is the speed-power product for three versions of the 7400-format quad NAND gate, fabricated from different contemporary technologies in 1985:

Year	Technology	Device	Propagation delay (ns)	Power (mW)	Product (pJ)
1985	CMOS	74HC00	7 ns	1 mW	7 pJ
1985	TTL	74F00	3.4 ns	5 mW	17 pJ
1985	ECL	SP92701	0.8 ns	200 mW	160 pJ
2007	CMOS	74LVC00A	2.1 ns	120 uW	0.25 pJ

CMOS has been dominant, and in 2007 is the only surviving technology: 74LVC00A.pdf

The 5 volt CMOS gate has the property that it consumes virtually no power when not changing its output.

Today's lower voltage CMOS does not turn the transistors off as much, leading to significant static leakage currents.

The ECL gate is an older technology, with a higher speed-power product, but it is still useful since it is the fastest.

Gates of medium complexity or larger (rather than SSI gates as these are) tend to be an order better in speed or power, since they do not have output stages designed for driving long nets.

Alternatives to silicon, such as GaAs have been proposed for general purpose logic. GaAs has four times higher electron mobility and so transistors of a given size switch on and off that much faster. However, increases in the speed of silicon, simply by making things smaller, have turned out to be a more effective way forward. So far!

8.20 90 Nanometer Gate Length.

The mainstream VLSI technology in the period 2004-2008 was 90 nm. Parameters from a 90 nanometer standard cell library:

Parameter	Value	Unit
Drawn Gate Length	0.08	μm
Metal Layers	6 to 9	layers
Max Gate Density	400K	gates/mm^2
Track Width	0.25	μm
Track Spacing	0.25	μm
Tracking Capacitance	1	fF/mm
Core Supply Voltage	0.9 to 1.4	V
FO4 Delay	51	ps
Leakage current		nA/gate

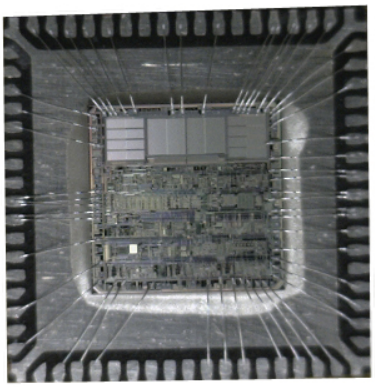


Figure 8.19: A wire-bonded silicon chip.

Typical processor core: 200k gates + 4 RAMs: one square millimeter. Typical SoC chip area is 50-100 mm^2 $\rightsquigarrow$ 20-40 million gates. Actual gate and transistor counts are higher owing to custom blocks (RAMs mainly).

Now the industry is using 35-45 nanometer.

- 2007: Dual-core Intel Itanium2: 1.6 billion transistors (90 nm).
- 2010: 8-core Intel Nehalem: 2.3 billion transistors (45 nm).
- 2010: Altera Stratix IV FPGA: 2.5 billion transistors (40 nm).

Moore's Law Transistor Count

The slide shows typical parameters from a 90 nanometer standard cell library. This figure refers to the width of the gate in the field effect transistors. The smaller this width, the faster than transistor can operate, but also it will consume more power as static leakage current. The 90 nm figure has been the mainstream VLSI technology in the period 2004-2008, but now the industry has moved to a 40-45 nanometer technology.

Typical processor core: 200k gates + 4 RAMs: one square millimeter.

A typical SoC chip area is 50-100 mm^2 with 20-40 million gates. Actual gate and transistor count would be higher owing to custom blocks (RAMs mainly), that achieve a better denisty than standard cells.

The FO4 delay is the delay through an inverter that is feeding four other inverters (fan out of four).

Moore's Law has been tracked for the last two plus decades, but have we now reached the *Silicon End Point*? That is, can we no longer make things smaller (at the same cost)? Modern workstation processors have certainly demonstrated a departure from the previous trend of ever rising clock frequencies: instead they have several cores.

8.21 Delay Estimation Formula.

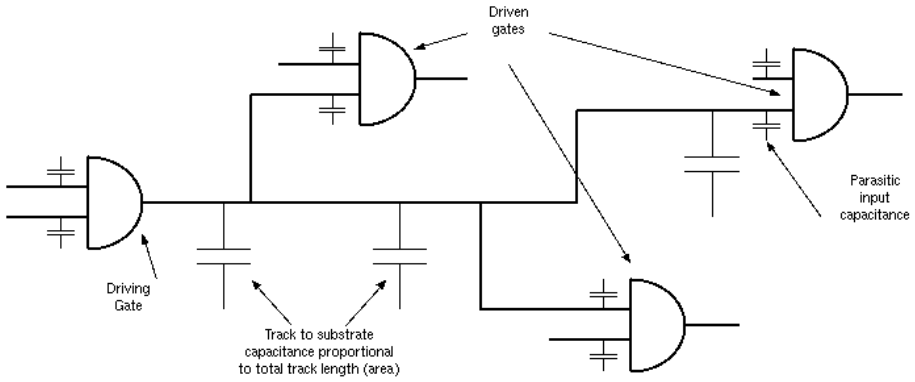


Figure 8.20: Logic net with tracking and input load capacitances illustrated.

Both the power consumption and effective delay of a gate driving a net depend mainly on the length of the net driven.

$$\text{device delay} = (\text{intrinsic delay}) + (\text{output load} \times \text{derating factor}).$$

The track-dependent output loading is a library constant times the track area.

The load-dependent part is the sum of the input loads of all of the devices being fed.

For short, non-clock nets (less than 0.1 wavelength), we just include propagation delay in the gate derating and assume the signal arrives at all points simultaneously.

Precise track lengths are only known after place and routing (Figure 1.2). Pre-layout and pre-synthesis we need RTL-level heuristics.

Figure 8.20 shows a typical net, driven by a single source. To change the voltage on the net, the source must overcome the stray capacitance and input loads. The fanout of a gate is the number of devices that its output feeds. The term *fanout* is also sometimes used for the maximum number of inputs to other gates a given gate is allowed to feed, and forms part of the design rules for the technology.

The speed of the output stage of a gate, in terms of its propagation delay, decreases with output load. Normally, the dominant aspect of output load is capacitance, and this is the sum of:

- the capacitance proportional to the area of the output conductor,
- the sum of the input capacitances of the devices fed.

To estimate the delay from the input to a gate, through the internal electronics of a gate, through its output structure and down the conductor to the input of the next gate, we must add three things:

- the internal delay of the gate, termed the intrinsic delay
- the reduction in speed of the output stage, owing to the fanout/loading, termed the derating delay,
- the propagation delay down the conductor.

The propagation delay down a conductor obeys standard transmission line formula and depends on the distributed capacitance, inductance and resistance of the conductor material and adjacent insulators. For circuit board traces, resistance can be neglected and the delay is just the *speed of light* in the circuit board material: about 7 inches per nanosecond, or 200 metres per microsecond. Hence, for short nets on chip, less than one tenth a wavelength long, we commonly assume the signal arrives at all destinations at once and model the propagation delay as an additional component of the gate derating.

8.22 Power Estimation Formula

Power is measured in Watts and $P = V \times I = E \times f$

Gate current I = Static Current (leakage) + Dynamic Current.

Early CMOS (VCC 5 volts): negligible static current, but today at VCC of 1.3 volts it's 30

Dynamic current = Short circuit current + Dynamic charge current.

Dynamic charge current computation:

- All energy in a net/gates is wasted each time it goes from one to zero.
- The energy in a capacitor is $E = CV^2/2$.
- Dominant capacitance is proportional to net length.
- Gate input and output capacitance also contribute to C.

Powers 03

Some additional dynamic current is consumed as 'short-circuit current' which is current consume when both the P and N transistors are on at once, during switching, but we ignore that here.

Activity ratio, a: is the percentage of clock cycles that see a transition. The net toggle rate = Operating frequency of the chip $f \times a$;

Useful article: POWER MANAGEMENT IN CPU DESIGN

- 1 W/cm^2 can be dissipated from a plastic package.
- 2-4 W/cm^2 required a heat sink.

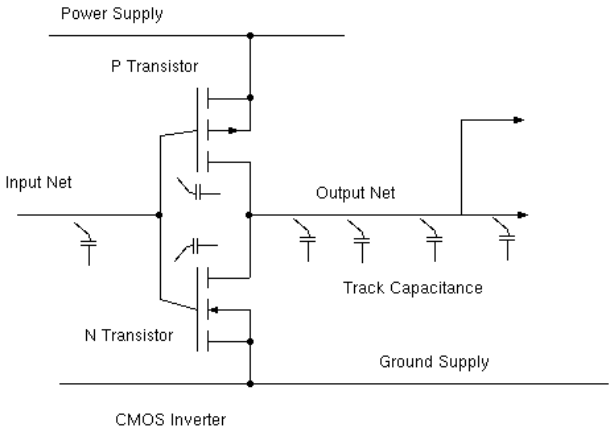


Figure 8.21: CMOS inverter with parasitic capacitances illustrated.

- More than 8 W/cm^2 requires forced cooling.

Workstation microprocessors dissipate tens of Watts: hence cooling fans.

Example: core area 64 mm^2; average net length 0.1 mm; 400K gates/mm^2, $a = 0.25$.

Net capacitance = 0.1 mm $\times$ 1 fF/mm $\times$ 400K $\times$ 64 mm^2 = 2.5 nF.

Vcc Volts	Freq MHz	Static Power mW	Dynamic Power mW	Total Power mW
0.8	100	40	24	64
1.35	100	67	68	135
1.35	200	67	136	204
1.8	100	90	121	211
1.8	200	90	243	333
1.8	400	90	486	576

- 1 W/cm^2 can be dissipated from a plastic package.
- 2-4 W/cm^2 required a heat sink.
- More than 8 W/cm^2 requires forced cooling.

Workstation microprocessors dissipate tens of Watts: hence cooling fans.

The table shows example power consumption for a circuit when clocked at different frequencies and voltages. The important thing to ensure is that the supply voltage must be sufficient for the clock frequency in use: too low a voltage means that signals do not arrive at D-type inputs in time to meet set up times.

Compare 1.35V to 1.8V: twice the power and twice the clock frequency.

In the past, chips were often core-bound or pad-bound. Pad-bound meant that the chip had too many I/O signals for its core logic area: the number of I/O's puts a lower bound on the perimeter of the chip. Today's VLSI technology allows I/O pads in the middle of the chip and designs are commonly power-bound.

8.23 Dynamic Clock Gating

Clock trees consume quite a lot of the power in an ASIC and considerable savings can be made by turning off the clocks to small regions. A region of logic is idle if all of the flip-flops are being loaded with their current contents,

either through synchronous clock enables or just through the nature of the design. EDA DESIGNLINE

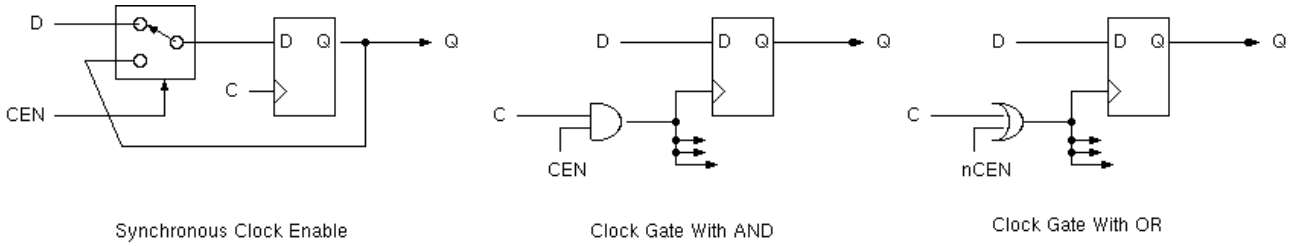


Figure 8.22: Clock enable using multiplexor, AND and OR gate.

Instead of using synchronous clock enables, current design practice is to use a clock gating insertion tool that gates the clock instead.

One logic gate serves a number of neighbouring flip-flops: state machine or broadside register.

Problem with AND gate: if CEN changes when clock is high: causes a glitch. Problem with OR gate: if CEN changes when clock is low: causes a glitch. Hence, care must be taken not to generate glitches on the clock as it is gated. transparent latches in the clock enable signal can re-time it to prevent this.

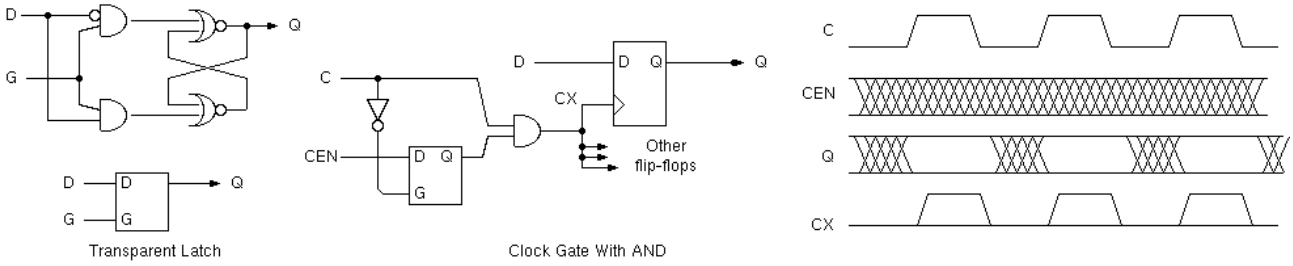


Figure 8.23: Illustrating a transparent latch and its use to supress clock gating glitches.

Care needed to match clock skew when crossing to/from non-gated domain: avoid *shoot-through* by building out the non-gated parts as well. Shoot-through occurs when a D-type is supposed to register its current D input value, but this has already changed to its new value before the clock signal arrives.

How to generate clock enable conditions ?

One could have software control for complete blocks (additional control register flags, as per power gating). But today's designs automatically detect on a finer-grain basis. Synthesiser tools can automatically insert clock required conditions and insert the additional logic. Automatic tools compute 'clock needed' conditions.

A clock is 'needed' if any register will change on a clock edge.

A lot of clock needed computation can get expensive, resulting in no net saving, but it can be effective if computed once at head of a pipeline.

Need to be sure there are no 'oscillating' stages or else know their settling time. The maximum settling time, if it exists, in terms of clock cycles before no further changes can be statically determined for a logic circuit using automatic analysis.

Beyond just turning off the clock or power to certain regions, we can consider further power saving techniques: dynamic frequency and voltage scaling.

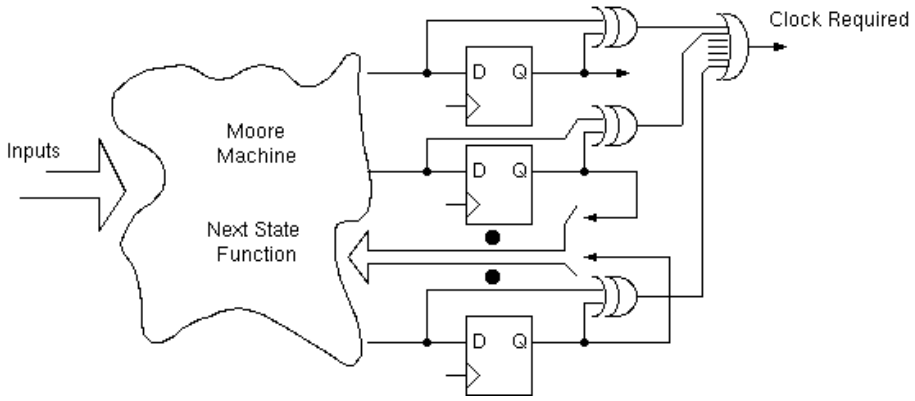


Figure 8.24: Using XOR gates to determine whether a clock edge would have any effect.

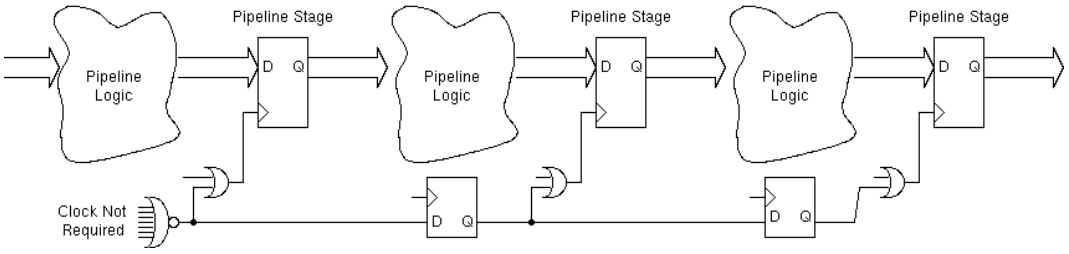


Figure 8.25: Clock needed computations forwarded down a pipeline.

8.24 Dynamic Power Gating

Increased tendenacy towards multi-product platform chips means large functional blocks on silicon may be off for complete product lifetime. Battery powered, portable devices can also use macro-scale block power down (e.g. the audio or video input and output subsystems).

These techniques typically require some sequencing: several clock cycles to power up/down a region.

Fujitsu Article

Previously we looked at dynamic clock gating, but we can also turn off power supply to regions of a chip, allbeit with coarser grain. We use power gating cells in series with supply rails.

Use signal isolation and retention cells (t-latches) on nets that cross in and out of the region. There is no register and RAM data retention in a block while the power is off. This technique is most suitable for complete sub-systems of a chip, that are not in use on a particular product or for quite a long time, such as a bluetooth tranceiver or audio input ADC.

Generally, power off/on is controlled by software or top-level input pads to the SoC. It requires some sequencing to activate the enables to the retention cells in the correct order and hence several clock cycles or more are needed to power up/down a region.

A common pracite is to power off a whole chip except for a one or two RAMs and register files. This was particularly common before FLASH memory was invented, when a small battery is/was used retain contents using a lower supply (CMOS RAM data holding voltage). Today, most mobile phones and PC mother cards have a second, tiny battery that maintains a small amout of logic when the main power is off or battery removed. This can run the real-time clock (RTC) as well.

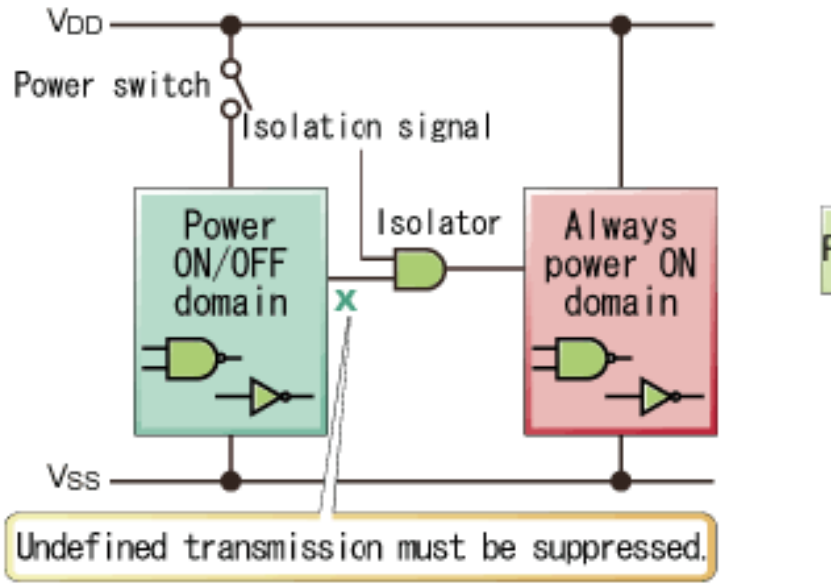


Figure 8.26: An isolation gate needed on an output from a power-gated region.

8.25 Dynamic Frequency Scaling

Let's adjust the clock frquency (while keeping VCC constant for now). Does this help ?

Let's compare with dynamic clock gating:

	Clock Gating.	Frequency Adjustment.
Control:	automatic,	manual.
Granularity:	register / FSM,	macroscopic.
Clock Tree:	mostly free runs,	slows down.
Response time:	instant,	acceptable.
Can vary voltage:	no,	yes.

To compute quickly and halt we need a higher frequency clock but consume the same number of active cycles. So the work-rate product, af, unchanged, so no power difference ?

Actually un-stopped regions consume power proportional to f.

Zeno: Tortoise and Achilles ? Tortoise is best: keep going steadily and end just in time. (He appeals even more when we vary the voltage.)

But, dynamic clock gating still good for: bursty, localised activity.

Consider adjusting the clock frquency (while keeping VCC constant for now). What does this acheive? For a fixed task, it will take longer to complete. If the processor is to halt at the end of the task, it will spend less time halted. If the main clock treee keeps going while halted, yet most of the chip uses local clock gating, then we do save some power in that fewer useless clock cyles are executed by the main clock tree.

This sort of frequency scaling can be software controlled: update PLL division ratio. Figure 4.19 illustrates the PLL. The PLL has inertia: e.g. 1 millisecond, but this is similar to the rate at which an operating system services interrupts, and hence the clock frequency to a system can be ramped up as load arrives. This is how most laptops now work.

Let's compare with dynamic clock gating: the table shows the main differences, but the most important difference is still to come: we can reduce the supply voltage if we have reduced the clock frequency.

8.26 Dynamic Voltage Scaling

Looking at the derating graph for the standard cell libraries, we see that in the operating region, the frequency/voltage curve is roughly linear. CMOS delay is inversely proportional to supply voltage.

Logic with higher-speed capabilities is smaller which means it consumes greater leakage current which is being wasted while we are halted. Also leakage power is largely proportional to supply voltage.

If we vary the voltage to a region dynamically, while keeping f constant, a higher supply voltage uses more power (square law) but would allow a higher f.

Let's only raise VCC when we ramp up f.

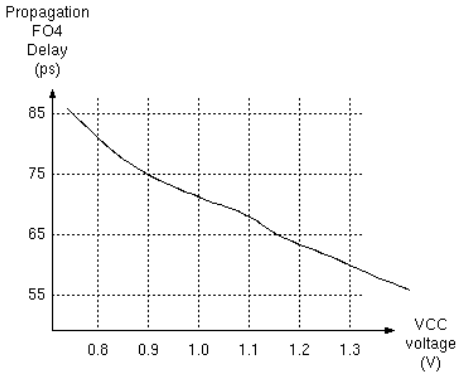


Figure 8.27: Variation of gate delay with supply voltage

Method:

1. Adjust f for just-in-time completion (e.g. in time to decode the next frame of a real-time video),
2. then adjust VCC so logic just works.

But Zeno applies still: always aim for a close to unity and a low work rate.

Overall: power will then have cubic dependence on f.

Hence, we still obtain peak performance under heavy loads, yet avoid cubic overhead when idle. We adjust VCC so that, at all times, the logic just works. However, we need to keep close track of whether we are meeting real-time deadlines.

Combinational logic cannot be clock gated (e.g. PAL and PLA). For large combinational blocks: can dip power supply to reduce static current when block is completely idle (detect with XORs).

So a typical SoC uses not only dynamic clock gating, but also manual and automatic frequency and voltage variation. Power isolation is used on a longer-scale.

LG 9 — High-level Design Capture and Synthesis

In this final section of the course we look at high-level design entry methods and automatic synthesis from high-level descriptions.

9.1 Spirit IP XACT

IP-XACTis an XML Schema for IP Block Documentation.

It is being developed by the SPIRIT Consortium as a standard for automated configuration and integration of IP blocks.

It describes interfaces and attributes of a block (e.g. terminal and function names, register layouts and non-functional attributes).

It includes separate RTL and ESL/TLM descriptions (future work to integrate these).

It aims to provide all the front-end infrastructure for rapid SoC assembly from diverse IP supplies, support for assertions and and perhaps even some glue logic synthesis.

9.2 IP XACT Tool Flow

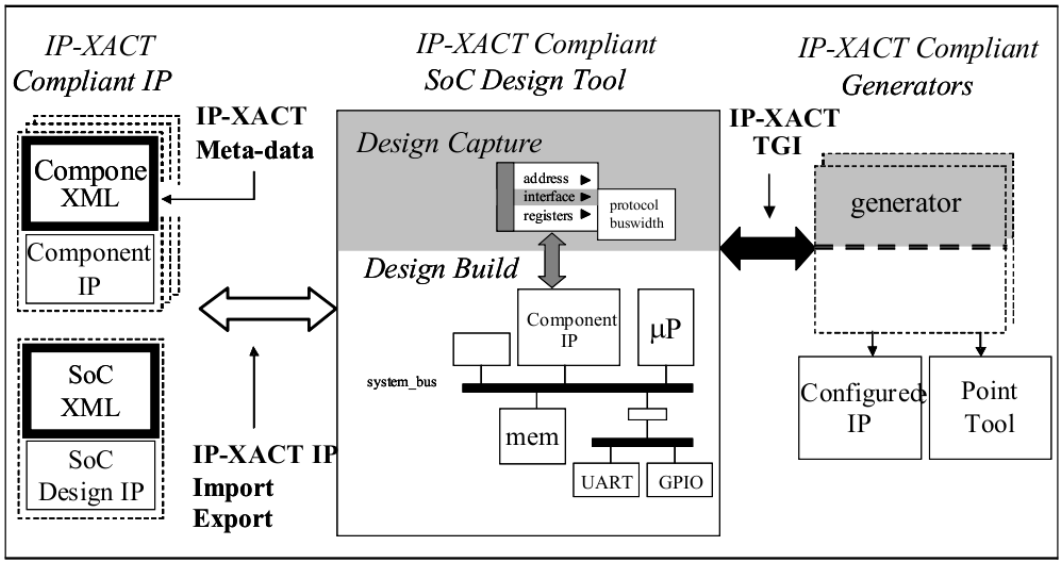


Figure 9.1: Reference Model for design capture and synthesis using IP-XACT blocks.

IP blocks are stored in libraries indexed using IP-XACT information. The SoC design is also described in conformant XML. A design capture editor supports creation of a high-level block diagram of the SoC. Various synthesis plugins, termed 'generators' produce the actual RTL and other outputs, such as power and frequency estimates or user manuals.

Automatic generation of memory maps is also useful. Header files in RTL and C can be kept in synch. (All modern PC motherboards do automatic generation of memory maps as part of the BIOS plug-and-play service.)

Try out the free plugin(s) for Eclipse!

9.3 High-Level Synthesis

Manual RTL design expression needs

- Human comprehension of the state encoding, and
- Human comprehension of the cycle-by-cycle concurrency.

Performing a Time for Space re-folding (i.e. doing the same job with more/less silicon over less/more time) requires a **complete** redesign at this level!

Can we do better ? Want to use **High-Level Synthesis**.

If one considers an embedded processor connected to a ROM, it may be viewed as one large FSM. Since for any given piece of software, the ROM is unlikely to be full and there are likely to be resources in the processor that are not used by that software, the application of a good quality logic minimiser to the system, while it is in the design database, could trim it greatly. In most real designs, this will not be helpful: for instance, the advantages of full-custom applied to the processor core will be lost. In fact, the minimisation function may be too complex for most algorithms to tackle on today's computers.

On the other hand, algorithms to create a good static scheduling of a fixed number of hardware resources work quite well. A processing algorithm typically consists of multiple processing stages (e.g. called pre-emphasis, equalisation, coefficient adaptation, FFT, deconvolution, reconstruction and so on). Each of these steps normally has to be done within tight real-time bounds and so parallelism through multiple instances of ALU and register hardware is needed. The Cathedral DSP compiler was an early tool for helping design such circuits. Such tools can perform time/space folding/unfolding of the algorithm to generate the static shedule that maps operations and variables in a high-level description to actual resources in the hardware. Data dependencies can cause variations in the time for certain steps, so a potentially a dynamic schedule could make better use of resources but the overhead of dynamic scheduling can outweigh the cost of the resources saved if the data dependencies are rare.

9.4 Higher level: Behavioural or Logical ?

There are two primary, high-level styles we can consider, and we can also consider blends of them:

- **Behavioural Expression**: Using imperative software-like code, where threads have stacks and pass between modules, and so on...
- **Declarative/Logical Expression**: Constraining assertions about the allowable behaviour are given, but any ordering constraints are implicit.

There is a related subject of back-end synthesis: netlist generation, re-encoding and re-pipelining to meet timing closure and power budgets.

9.5 Behavioural Expression

Using the first of these, behavioural expression, we express the algorithm and steps to be performed as an executable program

- using an **imperative** program (containing loops and assigments), or
- a **functional** program (where control flow is implicit).

Either way, the tool chain may:

- **re-order** the operations while preserving semantics, and/or
- **re-encode** the state and modify memory layouts.

Examples discussed:

- Synopsys Behavioural Compiler,
- Handel-C,
- BlueSpec System Verilog,
- CtoV : C-To-Verilog, SystemCrafter, Catapult, Kiwi, ...
- Statecharts.

9.6 Beyond Pure RTL: Behavioural descriptions of hardware.

What has 'synthesisable' RTL traditionally provided ?

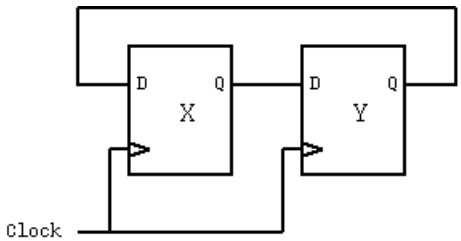


Figure 9.2: A circuit to swap two registers.

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 9.2. (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 and typically such assignment statements are incorporated in various nested `if-then-else` and `case` commands. This allows hardware designs to be expressed using the conventional imperative programming style that is familiar to software programmers. 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 if inspected by hand.

The word 'behavioural', when applied to a style of RTL or software coding, tends to simply mean that a sequential thread is used to express the sequential execution of the statements.

Despite the apparent power available using this form of expression, there are severe limitations in the offically synthesisable subset of Verilog and VHDL that might also be manifest in basic C-to-gates tool. Limitations are, for instance, each variable must be written by only one thread and that a thread is unable to leave the current file or module to execute subroutines/methods in other parts of the design.

The term '*behavioural model*' is used to denote a short program written to substitute for a complex subsection of a structural hardware design. The program would produce the same useful result, but execute much more quickly because the values of all the internal nets and pipeline stages (that provide no benefit until converted to actual parallel hardware form) were not modelled. Verilog and VHDL enable limited forms of behavioural models to serve as the source code for the subsection, with synthesis used to form the netlist. Therefore limited behavioural models can sometimes become the implementation.

9.7 More-advanced behavioural specification:

Many RTL synthesisers support an implied program counter (state machine inference).

```
reg [2:0] yout;
always
      begin
      @(posedge clk) yout = 1;
      @(posedge clk) yout = 4;
      @(posedge clk) yout = 3;
      end
```

In this example, not only is there a thread with current 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 explicitly mentioned by the designer, to hold the current 'program counter' value.

None of the event control statements is conditional in the example, but the method of compilation is readily extended to support this: it amounts to the program counter taking conditional branches. For example, the middle event control could be prefixed with 'if (din)'.

```
      if (din) @(posedge clk) yout = 4;
```

9.8 Static and Dynamic Scheduling

(N.B. Some of the material in this section is repeated from the RTL section of the notes.)

As mentioned in the RTL section of these notes, RAM ports, ALUs, non fully-pipelined components and other shared resources can cause *Structural Hazards*.

Structural Hazard: Cannot proceed with an operation because a resource is in use.

To overcome hazards we must use shedulling and arbitration:

- **Schedulling**: deciding the operation order in advance,
- **Arbitrating**: granting access dynamically, as requests arrive.

One schedulling decision impacts on another: ideally need to find a global optimum.

The schedulling and arbitration operations can often be done at **compile time**, (e.g. for operations performed by a single behavioural thread). Remainder must be done at **run time** according to actual input data since the relative interleaving of different threads is often unpredictable.

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. Large memories are best implemented using separate off-chip device where as sizes of hundreds of 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, but, where data can be partitioned (i.e. we know something about the access patterns) having several smaller memories gives better bandwidth and less contention and uses less power for a given performance.

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 Behavioural Compiler, which splits the user specified clock intervals into as many as needed actual clock cycles. With such a compiler, the above example is synthesisable using a single-ported RAM.

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. Advanced tools (e.g. C-to-Gates tools and Kiwi) generate a complete 'datapath' that consists of various ALUs, RAMs and register files. This is essentially the execution unit of a custom VLIW (very-long instruction word) processor, where the control unit is replaced with a dedicated finite-state controller.

The decisions about how many memories to use and what to keep in them may be automated or manual overrides might be specified.

9.9 Synopsys Behavioural Compiler

... was an advanced (for the late 90's) compiler that extended RTL synthesis semantics.

- Provided compile-time loop unrolling,
- Operations on variables freely moved between clock cycles,
- Additional cycles to overcome hazards (user's clock is called a 'super state'),
- Provided temporally floating I/O with variable pipelining between ports.

Existing RTL paradigms **not** preserved within the same source file: existing syntax has new meaning. Ulitmately, it seems designers felt they had lost control over detailed structure in critical places.

Synopsys Behavioural Compiler Tutorial

Citations:

- Understanding Behavioral Synthesis, A Practical Guide to High Level Design by John P Elliott; Kluwer Academic Publishers ISBN 0-7923-8542-X
- Behavioral Synthesis, Digital System Design Using the Synopsys Behavioral Compiler by David W. Knapp, Prentice Hall, ISBN 0-13-569252-0

9.10 Shortcomings of Verilog and VHDL (for H/L Synthesis).

Verilog and VHDL are languages focused more on simulation than logic synthesis. The rules for translation to hardware that define the 'synthesisable subset' were standardised post the definitions of the language.

Circuit aspects that 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.

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

- Speed of design: time to market,
- Facilitate richer behavioural specification,
- Readily allow time/space folding experiments,
- Greater freedom and hence scope for optimisation in the compiler,
- Facilitate implementation of a formal specification,
- Facilitate proof of conformance to a specification,
- Allow rule-based programming (i.e. a logic-programming sub-language),
- Support modern synchronisation primitives (e.g. join patterns)
- Portability: can be compiled into software as well as into hardware.

9.11 Channel Communications

Using shared variables to communicate between threads requires that the user abides by self-imposed protocol conventions.

Typical patterns are:

- always ready,

- simplex guard with reader always faster than writer,
- four-phase handshake,
- two-phase handshake.

As mentioned elsewhere in these notes, some protocols cannot be pipelined, some degrade throughput when pipelined and others are designed for it Some approaches completely ban shared variables and enforce use of channels (Handel-C and the main Bluespec dialect). (LINK: Handlec.pdf)

The Bluespec language infers channel-like behaviour from user syntax that looks like conventional reads and writes of shared variables.

Handel-C uses explicit Occam/CSP-like channels (pling to write, query to read):

```
while (1)              while(1)              while(1)
{                      {                     {
  ch1 ! (x);             ch2 ! (ch1? + 2)      $display(ch2?);
  x += 3;              }                     }
}
```

Using channels makes concurrency explict and allows synthesis to re-time the design.

In both cases, all of the handshaking signals potentially required are generated by the compiler and then trimmed away again if they would have constant values owing to certain components being always ready.

Bluespec Verilog also provides rule-based design expression (see later notes).

9.12 H/W Synthesis from C and other Programming Languages.

Can we convert arbitrary or legacy programs to hardware ? Not very well. Can we write new C programs that compile to good hardware ? Yes. Can we use software-style constructs in new C-like languages ? Yes.

Typical restrictions:

- **Program must be finite state**,
- all recursion bounded,
- all dynamic storage allocation outside of infinite loops (or deallocated again in same loop),
- use only boolean logic and integer arithmetic,
- limited string handling,
- very limited standard library support,
- be explicit over which loops have run-time bounds.

Baseline example DJG C-To-V compiler from 1995. Bubble Sorter Example

Commercial products available : SystemCrafter, Catapult, SimVision, CoCentric, ... others.

Try out an online demo on your own fragment of C at C-to-Verilog.com

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 such approaches is not fully proven, but there is great potential.

The software programming paradigm, where a serial thread of execution runs around between various modules is undoubtedly easier to design with than the forced parallelism of expressions found in RTL-style coding. Ideally, a new thread should only be introduced when there is a need for concurrent behaviour in the expression of the design.

A product from COMPILOGIC is typical of the new generation of such EDA tools. It 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.

We cannot compile general C/C++ programs to hardware.

A given function can generally be done in half as many clock cycles using twice as much silicon, although name aliases and control hazards (dependence on run time input data) can limit this. As well as the C/C++ input code we require additional directives over speed, area and perhaps power. The area directives may specify the number of RAMs or how to map arrays into shared DRAM.

Trading time for space is basically a matter of unwinding loops or introducing new loops.

Hazards can limit the amount of unrolling possible, including limited numbers of ports on RAMs and user-set budgets on the number of certain components instantiated, such as adders or multipliers.

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 encounters 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 circuit (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 intended for conversion to hardware. Instead, inputs and outputs are likely to be reads and writes to channels or static shared variables that map to standard registers and RAM blocks in the hardware implementation.

9.13 Kiwi : Compiling Concurrent Programs to Hardware

Current project led by David Greaves and Satnam Singh: Web Site

Kiwi is developing a methodology for hardware design using the parallel programming constructs of the C# language. Specifically, Kiwi consists of a run-time library for native simulation of hardware descriptions within C# and a compiler that generates RTL from stylised .net bytecode.

The designer uses more concurrency than 'natural' for software. This is mapped to concurrent hardware by the Kiwi tools.

For example: Times Table demo.

9.14 State charts and Graphical 'languages'

Synthesis from diagrams (especially UML/SysML) is useful:

- Full schematic entry at the gate level was once popular,
- Still popular for high-level system block diagrams,
- Also popular for state transition diagrams.

The stategraph general form is:

```
stategraph graph_name()
{
  state statename0 (subgraph_name, subgraph_entry_state), ... :

        entry:  statement;
        exit:   statement;
        body: statement;

        statement;
        ...             // implied 'body:' statements
        statement;

        c1 -> statename1: statement;
        c2 -> statename2: statement;
        c3 -> exit(good);
        ...

        exit(good) -> statename3: statement;
        exit(bad) -> statename4: statement;

        ...


   endstate

   state statename2:
   ...
   ...
   endstate


   state abort: // A special state that can be
                // forced remotely (also called disable).

   ...

}
```

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'* [Rational]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.

Schematic entry of netlists is now only applicable to specialised, 'hand-crafted' sub-circuits, but graphical methods for composing system components at the system-on-a-chip level is growing in popularity.

9.14.1 Statechart Details (from my H2 Language).

A state may contain tagged statements, each of which may be a basic block if required. They are distinguished using three tag words. The 'entry' statement is run on entry to the state and the 'exit' statement is run on exit. The 'body' statement is run while in the state. A 'body' statement must contain idempotent code, so that there is no concept of the number of times it is run while in the state. Statements with no tag are treated as body tagged statements. Multiple occurrences of statements with the same tag are allowed and these are evaluated as though executed in the textual order they occur or else in parallel.

A state contains transition definitions that define the successor states. Each transition consists of a boolean guard expression, the name of one of the states in the current stategraph and an optional statement to be executed when taking the transition. In situations where multiple guard expressions currently hold, the first holding transition is taken.

The guard expressions range over the inputs to the stategraph, which are the variables and events in the current textual scope, and the exit labels of child stategraphs.

When a child stategraph becomes active, it will start in the starting state name is given as an argument to the instantiation, or the first state of no starting name is given.

A child stategraph becomes inactive when its parent transitions, even if the transition is to the current state, in which case the child stategraph becomes inactive and active again and so transitions to the appropriate entry state.

A child stategraph can cause its parent to transition when the child transitions to an exit state. There may be any number, including zero, of exit states in a child stategraph but never any in a top-level stategraph. The parent must define one or more transitions to be taken for all possible exit transitions of its children. An exit state is either called 'exit' or 'exit(id)' where 'id' is an exit tag identifier. Exit tags used in the children must all be matched by transitions in the parent, or else the parent must transition itself under the remaining exit conditions of the child or else the parent must provide an untagged exit that is used by default.

9.15 Behavioural H/L Synthesis Summary

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

- unbounded recursive functions,
- unbounded heap use
- other sources of unbounded numbers of state variables,
- many library functions: access to file or screen I/O.

Generating good hardware requires global optimisation of the major resources (ALUs, Multipliers and Memory Ports) and hence automatic time/space folding. New techniques are needed that note that wiring is a dominant power consumer in today's ASICs

9.16 Synthesis from Declarative Specifications

Rather than specify the algorithm (behaviour) we specify the required outcome. Rather like constraint-based linear programming, the design is a piece of hardware that satisifes a number of simultaneous assertions.

Examples covered:

- Refinement Synthesis from Formal Specs,
- Rule-based hardware generation (BlueSpec),
- SAT-based logic Synthesis,
- Automatic Synthesis of Transactors and Bus Monitors (in additional materials),
- Automatic Synthesis of Glue and Interface Automata (in additional materials).

9.17 Synthesis from Formal Specification

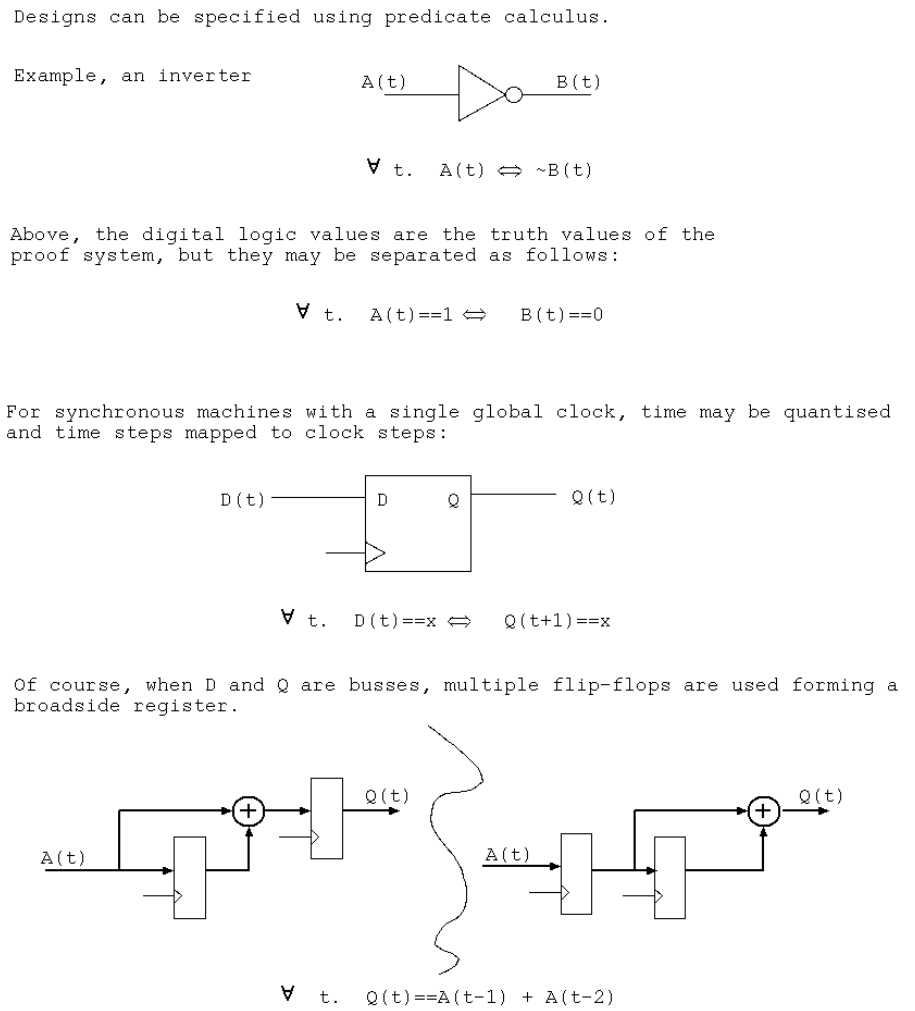


Figure 9.3: Fragments: compilation from formal specifications.

It is desirable to eliminate the human aspect from 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 tend to mean a declarative language based on set theory and typically one 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, optimum. Research using expert systems to select the best implementation is ongoing, but human input is needed in practical systems. But the human input should only be a guide to synthesis, choosing 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 declaratively 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 they are separate might is typically needed when don't care states are encompassed.

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

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 does not describe the 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.

See whitepaper from OneSpin-Solutions.com

Wikipedia: program refinement

9.18 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 conversion are given in Figure 9.3.

There are a vast number of refinement rules available for application at each refinement step and the quality of the outcome is sensitive to early decisions. Therefore, it's hard to make this fully automated.

Perhaps a good approach is for much of the design to be specified algorithmically 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. Practical examples are the handshake and glue logic for bus or network protocols.

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

The synthesis system should allow a free mix of design specifications in many forms, including behavioural fragments and functional specifications. and only complain or fail when:

- the requested system is actually impossible: e.g. the output comes before the input that caused it,
- the system is over-specified in a contradictory way,
- the algorithm for implementing the desired function cannot be determined afterall.

9.19 Rule-based hardware generation (BlueSpec)

In the last two years, Bluespec System Verilog has successfully raised the level of abstraction in RTL design in the industry.

- A Bluespec design is expressed as a list of declarative rules,
- Shared variables are mostly replaced with one-place FIFO buffers with automatic handshaking,
- Rules are allocated a static schedule at compile time and some that can never fire are reported,
- The current tight control of clock cycle (time/space folding) might be relaxed by future compilation strategies.

LINK: Small Examples

First basic example: two rules: one increments, the other exits the simulation. This example looks very much like RTL: provides an easy entry for hardware engineers.

```
module mkTb (Empty);

   Reg#(int) x <- mkReg (23);

   rule countup (x < 30);
      int y = x + 1;
      x <= x + 1;
      $display ("x = %0d, y = %0d", x, y);
   endrule

   rule done (x >= 30);
      $finish (0);
   endrule

endmodule: mkTb
```

Second example uses a pipeline object that could have aribtrary delay. Sending process is blocked by implied handshaking wires (hence less typing than Verilog) and in the future would allow the programmer or the compiler to retime the implementation of the pipe component.

```
module mkTb (Empty);

   Reg#(int) x    <- mkReg ('h10);
   Pipe_ifc  pipe <- mkPipe;

   rule fill;
      pipe.send (x);
      x <= x + 'h10;
   endrule

   rule drain;
      let y = pipe.receive();
      $display ("    y = %0h", y);
      if (y > 'h80) $finish(0);
   endrule
endmodule
```

But, behavioural expressing using a conceptual thread is also useful to have!

9.20 Synthesis from Rules (SAT-based idea).

Crazy idea ? If we program an FPGA we are generating a bit vector. SAT solvers produce bit vectors that conform to a conjunction of constraints.

Let's specify the design as a set of constraints over a fictional FPGA... We can also convert structural and behavioural design expressions to very-tight constraints and add those in.

The SAT solution wires up the FPGA and we can then apply logic trimming. LINK: SAT Logic Synthesis (Greaves)

Main poblem: how large an FPGA to start with? Redundant logic might need a bi-simulation erosion to remove it.

Seems to work for generating small custom protocols.