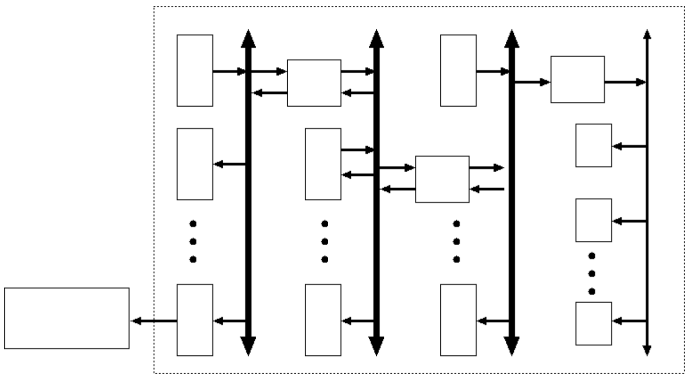
System on Chip
Design and Modelling



University of Cambridge
Computer Laboratory
Lecture Notes
KG4 Handout

Dr. David J Greaves

Part II
Computer Science Tripos
Michaelmas Term 2017/18

KG 4 — RTL, Interfaces, Pipelining and Hazards

An RTL description is at the so-called **Register Transfer Level**

A hardware design consists of a number of modules interconnected by wires known as 'nets' (short for networks). The interconnections between modules are typically structured as mating interfaces. An interface nominally consists of a number of terminals but these may have no physical manifestation since they are typically obfuscated during logic optimisation. In a modern design flow, the protocol at an interface is ideally specified once in a master file that is imported for the synthesis of each module that sports it. But a lot of low-level manual entry of RTL design is still (2017) used.

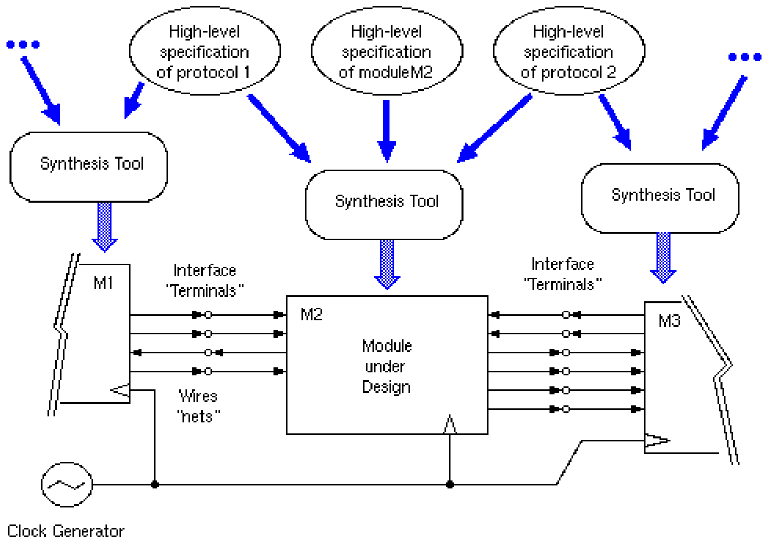


Figure 4.1: Generic (net-level) Module Interconnection Using Protocols and Interfaces.

A clock domain is a set of modules and a clock generator. Within a synchronous clock domain all flip-flops have their clocks commoned.

4.1 Protocol and Interface

At the electrical/net level, a **port** consists of an **interface** and a **protocol**. The interface is the set of pins or wires that connect the components. The protocol defines the rules for changing the logic levels and the meaning of the associated data. For example, an asynchronous interface might be defined in RTL as:

```
Transmit view of interface:       Receive view of interface:      // This is a four-phase asynchronous interface
  output [7:0] data;                input [7:0] data;             // where the idle state has strobe and ack
  output strobe;                    input strobe;                 // deasserted (low) and data is valid while
  input ack;                        output ack;                   // the strobe signal is asserted (high).
```

Ports commonly implement **flow-control** by handshaking. Data is only transferred when both the sender and receiver are happy to proceed.

A port generally has an **idle** state which it returns to between each transaction. Sometimes the start of one transaction is immediately after the end of the previous, so the transition through the idle state is only nominal. Sometimes the begining of one transaction is temporaly overlaid with the end of a previous, so the transition through idle state has no specific duration.

Additional notes:

There are four conceivable clock strategies for an interface:

Left Side	Right Side	Name
1. Clocked	Clocked	Synchronous (such as Xilinx LocalLink)
2. Clocked	Different clock	Clock Domain Crossing (see later)
3. Clocked	Asynchronous	Hybrid.
3. Asynchronous	Clocked	Hybrid (swapped).
4. Asynchronous	Asynchronous	Asynchronous (such a four-phase parallel port)

Any interface that does not have any or the same clock on boths sides is asynchronous. (For example, the clock-domain crossing bridge discussed elsewhere in these notes.)

4.1.1 Transactional Handshaking

The mainstream RTL languages, Verilog and VHDL, do not provide synthesis of handshake circuits (but this is one of the main innovations in some more recent HLDs such as Bluespec). We'll use the word **transactional** for protocol+interface combinations that support flow-control. If synthesis tools are allowed to adjust the delay through components, all interfaces between components must be transactional and the tools must understand the protocol semantic.

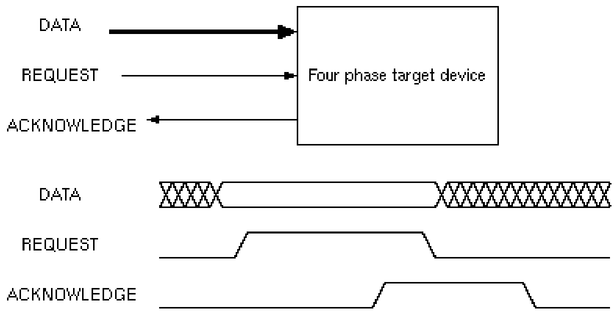


Figure 4.2: Timing diagram for an asynchronous, four-phase handshake.

Here are two imperative (behavioural) methods (non-RTL) that embody the protocol for Figure 4.2:

```
//Output transactor:
  putbyte(char d)
  {
    wait_until(!ack); // spin till last complete.
    data = d;
    settle(); // delay longer than longest data delay
    req = 1;
    wait_until(ack);
    req = 0;
  }
```

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

Code like this is used to perform programmed I/O (PIO) on GPIO pins (see later). It can also be used as an ESL transactor (see later). It's also sufficient to act as a formal specification of the protocol.

4.1.2 Transactional Handshaking in RTL (Synchronous Example)

The four-phase handshake just described is suitable for asynchronous interfaces. It does not refer to a clock.

A very common paradigm for synchronous flow control of a uni-directional bus is to have a handshake net in each direction with bus data being qualified as valid on **any positive clock edge where both**

handshake nets are asserted. The nets are typically called 'enable' and 'ready'. This paradigm forms the essence of the LocalLink protocol from Xilinx and AXI-streaming protocols defined by ARM.

Like the four-phase handshake, LocalLink has contra-flowing The interface nets for an eight-bit transmitting interface are:

```
input clk;
output [7:0] xxx_data; // The data itself
output xxx_sof_n;      // Start of frame
output xxx_eof_n;      // End of frame
output xxx_src_rdy_n;  // Req
input  xxx_dst_rdy_n;  // Ack
```

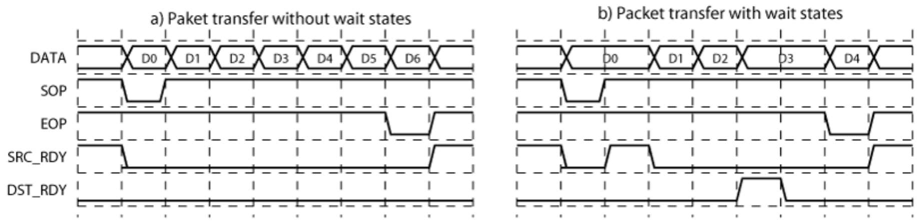

a) Paket transfer without wait states
b) Packet transfer with wait states
DATA
SOP
EOP
SRC_RDY
DST_RDY
D0
D1
D2
D3
D4
D5
D6
D0
D1
D2
D3
D4

Figure 4.3: Timing diagram for the synchronous LocalLink protocol.

To impose packed delineation on the bus, LocalLink defines start and end of frame signals. Note: all control signals are active low (denoted with the underscore `n` RTL suffix) in LocalLink.

Additional notes:

Here is a data source in Verilog RTL for LocalLink that generates a stream of packets containing arbitrary data with arbitrary gaps.

```
module LocalLinkSrc(
                 input reset,
                 input clk,
                 output [7:0] src_data,
                 output src_sof_n,
                 output src_eof_n,
                 output src_src_rdy_n,
                 input src_dst_rdy_n);


   // The source generates 'random' data using a pseudo random sequence generator (prbs).
   // The source also makes gaps in its data using bit[9] of the generator.
   reg [14:0] prbs;
   reg       started;
   assign src_data = (!src_src_rdy_n) ? prbs[7:0] : 0;
   assign src_src_rdy_n = !(prbs[9]);

   // The end of packet is arbitrarily generated when bits 14:12 have a particular value.
   assign src_eof_n =  !(!src_src_rdy_n && prbs[14:12]==2);

   // A start of frame must be flagged during the first new word after the previous frame has ended.
   assign src_sof_n =  !(!src_src_rdy_n && !started);


  always @(posedge clk) begin
     started <= (reset) ? 0: (!src_eof_n) ? 0 : (!src_sof_n) ? 1 : started;
        prbs <= (reset) ? 100: (src_dst_rdy_n) ? prbs: (prbs << 1) | (prbs[14] != prbs[13]);
   end
endmodule
```

And here is a corresponding data sink:

```
module LocalLinkSink(
                  input reset,
                  input clk,
                  input [7:0] sink_data,
                  input sink_sof_n,
                  input sink_eof_n,
                  output sink_src_rdy_n,
                  input sink_dst_rdy_n);

   // The sink also maintains a prbs to make it go busy or not on an arbitrary basis.
   reg [14:0] prbs;
   assign sink_dst_rdy_n = prbs[0];

   always @(posedge clk) begin
      if (!sink_dst_rdy_n && !sink_src_rdy_n) $display(
        "%m LocalLinkSink sof_n=%d eof_n=%d  data=0x%h", sink_sof_n, sink_eof_n, sink_data);
      // Put a blank line between packets on the console.
      if (!sink_dst_rdy_n && !sink_src_rdy_n && !sink_eof_n) $display("\n\n");
      prbs <= (reset) ? 200: (prbs << 1) | (prbs[14] != prbs[13]);
     end

endmodule // LocalLinkSrc
```

Additional notes:

And here is a testbench that wires them together:

```
module SIMSYS();

  reg reset;
  reg clk;
  wire [7:0] data;
  wire sof_n;
  wire eof_n;
  wire ack_n;
  wire req_n;

  // Instance of the src
  LocalLinkSrc src (.reset(reset),
                     .clk(clk),
                     .src_data(data),
                     .src_sof_n(sof_n),
                     .src_eof_n(eof_n),
                     .src_src_rdy_n(req_n),
                     .src_dst_rdy_n(ack_n));

  // Instance of the sink
  LocalLinkSink sink (.reset(reset),
                      .clk(clk),
                      .sink_data(data),
                      .sink_sof_n(sof_n),
                      .sink_eof_n(eof_n),
                      .sink_src_rdy_n(req_n),
                      .sink_dst_rdy_n(ack_n)
                     );



   initial begin clk =0; forever #50 clk = !clk; end
   initial begin reset = 1; #130 reset=0; end

endmodule // SIMSYS
```

4.2 Architecture: Bus and Device Structure

Transmitting data consumes energy and causes delay. Basic physical parameters:

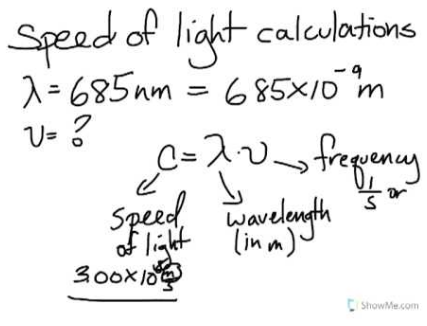


Figure 4.4: Speed of light is a constant (and in silicon it is lower).

- 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.
- Within a synchronous digital clock domain we require connections to be less than (say) 1/10th of a wavelength.
- Conductor series resistance further slows signal propagation and is dominant source of delay.
- So need to register a signal in several D-types if it passes from one corner of an 8mm chip to the other!

- Can have several thousand wires per millimetre per layer: fat busses (128 bits or wider) are easily possible.
- Significant DRAM is several centimetres0 away from the SoC and also 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. (Die stacking and recent DRAM-on-SoC approaches reduce wire length to a few mm for up to 500 MB of DRAM.)

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

4.2.1 A canonical D8/A16 Micro-Computer from 1980's

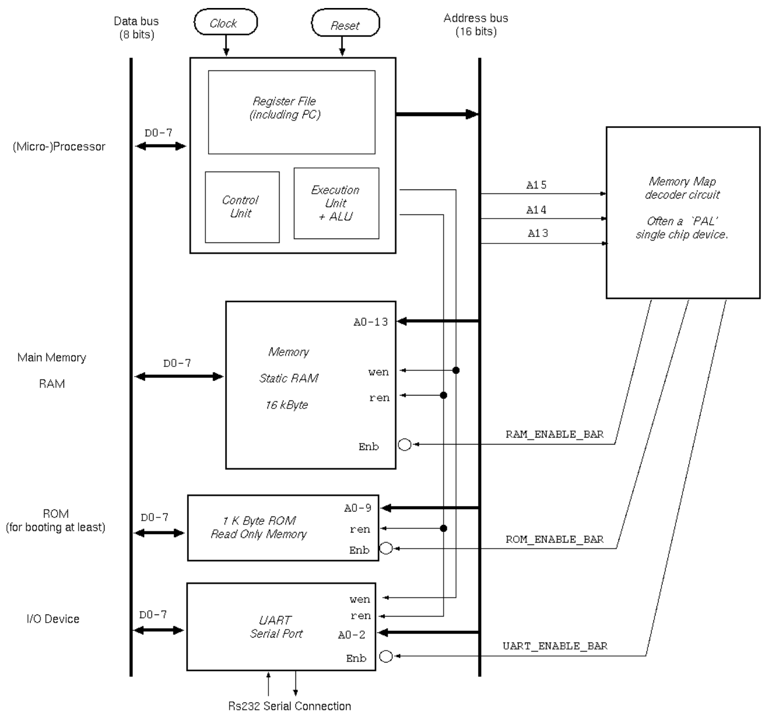


Figure 4.5: Early microcomputer structure, using a bidirectional/tri-state data bus.

Figure 4.5 shows the inter-chip wiring of a basic microcomputer (i.e. a computer based on a microprocessor). Busses of this nature were about 30cm long and had a cycle time of 250 ns or so.

4.2.2 Basic Bus: One initiator (II).

The bus in our early microcomputer was a true bus in the sense that data could get on and off at multiple places. SoCs do not use tri-states but we still use the term 'bus' to describe the point-to-point connections used today between IP blocks.

Figure 4.6 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.

Max throughput is unity (i.e. one word per clock tick). Typical SoC bus capacity: 32 bits × 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.

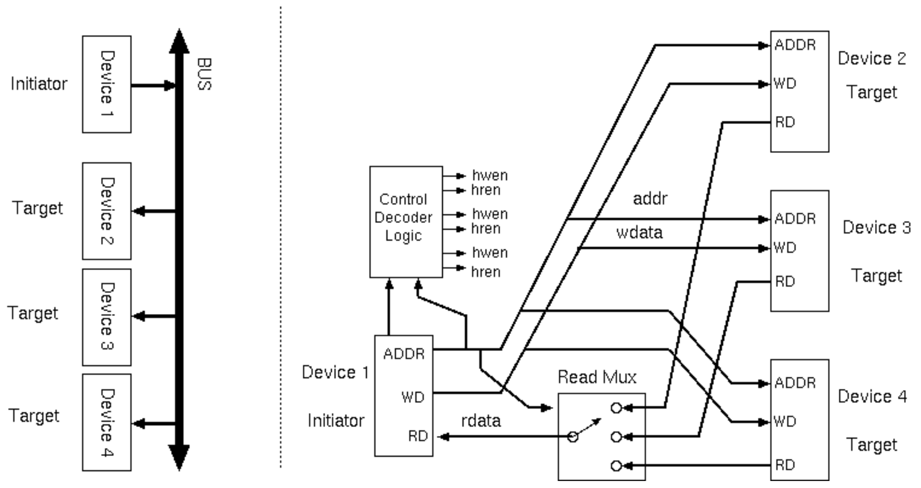


Figure 4.6: Example where one initiator addresses three targets.

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, 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.

4.2.3 Basic bus: Multiple Initiators (II).

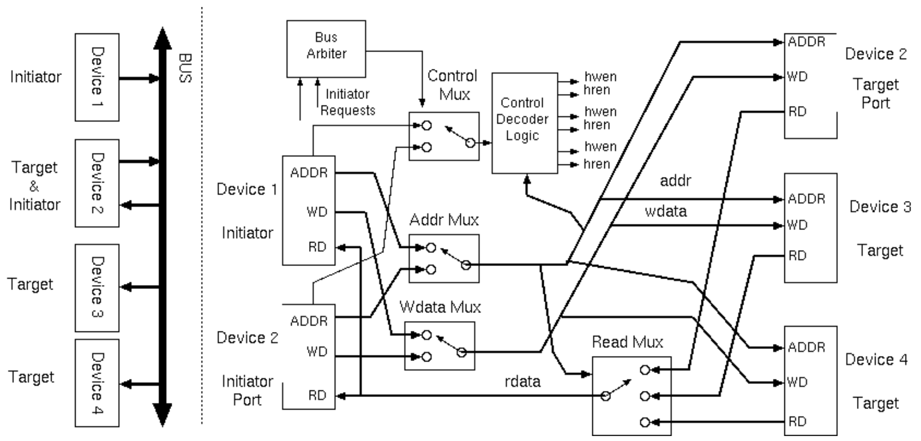


Figure 4.7: Example where one of the targets is also an initiator (e.g. a DMA controller).

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, owing to contention, we certainly 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.

4.2.4 Bridged Bus Structures.

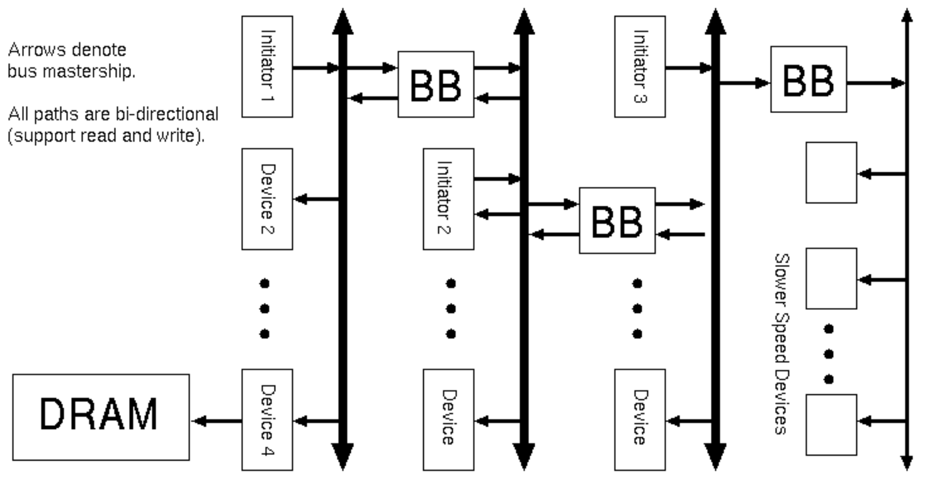


Figure 4.8: A system design using three main busses.

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.

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, other glue components and top-levels of structural wiring are typically instantiated by an automated tool inside the SoC designer's favourite design environment. Schematic entry for the SoC top level is common. From the open source world, this may be an IP-XACT plugin for Eclipse. All major EDA vendors also supply their own design environment tools.

4.2.5 Classes of On-Chip Protocol

1. Reciprocally-degrading: such as handshake protocols studied earlier: throughput is inversely proprotional to target latency in terms of clock cycles,
2. Delay-tolerant: such as AXI-lite and OCP's BVCI (below): new commands may be issued while awaiting responses from earlier,
3. Reorder-tolerant: such as full AXI: responses can be returned in a different order from command issue: helpful for DRAM access and needed for advanced NoC architectures.

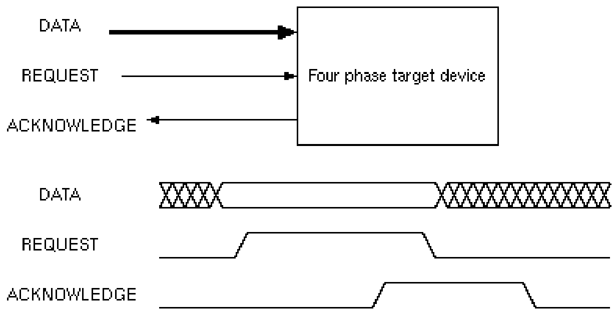


Figure 4.9: Timing diagram for an asynchronous, four-phase handshake.

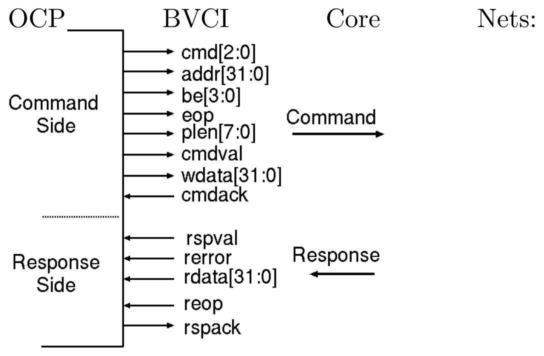
4. Virtual-circuit: (beyond scope of this course): rather than putting a destination address and port number in each message, traffic is routed at each hop via a short circuit identifier (or routing tag) where mappings have been set up in advance in the routing nodes.
5. Separated send and acknowledge circuits: A decoupling between send and reply or send and acknowledge, perhaps using a priority mechanism or perhaps using physical separation of the two directions of flow, exists, to ensure responses can always be returned, hence avoiding a form of deadlock.
6. Credit flow controlled: (beyond scope of this course): each source has a credit counter per destination or per destination/port number pair, controlling how many packets it can send without receiver buffer over-run.

Lables or tags need to be added to each transaction to match up commands with responses.

The EACD+ARCH part Ib classes use the 'Avalon' bus on the Altera devices: Avalon Interface Specifications

For those interested in more detail: Comparing AMBA AHB to AXI Bus using System Modelling

Last year you used the Altera Avalon bus in part IB ECAD+Arch workshops. Many real-world 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. 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 are neeed if block is both an initiator and target.
- Arrows indicate signal directions on initiator. All are reversed on target.

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. Older-style handshake protocols where targets had to respond within a prescribed number of clock cycles cannot be used in these situations. However BVCI requests and responses must not get our of order since there is no `id` token.

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.

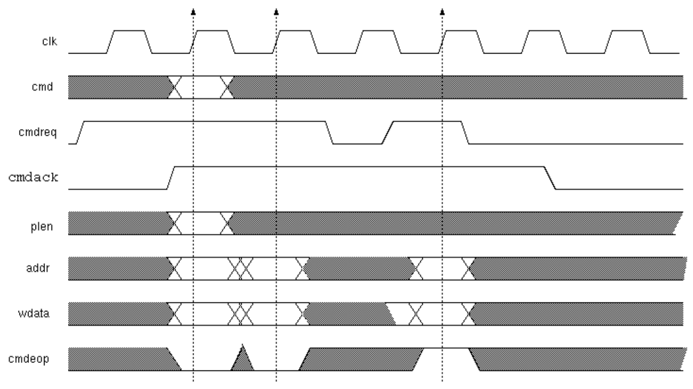
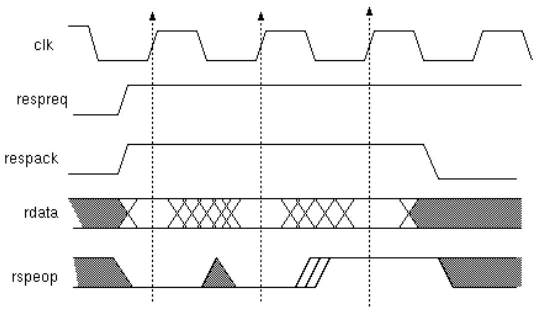


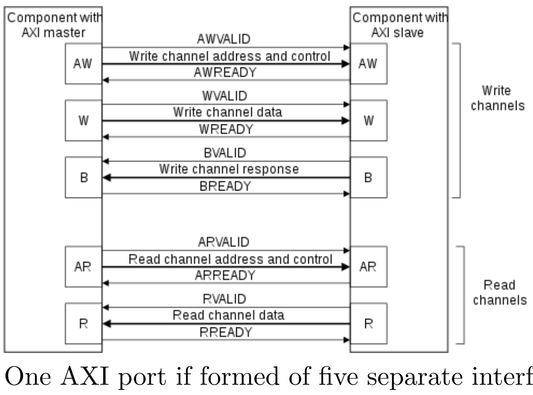
Figure 4.10: BVCI Protocol, Command Timing Diagram

Operations are qualified with conjunction of `req` and `ack`. Response and acknowledge cycles maintain respective ordering. Bursts are common. Successive addressing may be implied.



BVCI Response Portion Protocol Timing Diagram

4.2.6 ARM AXI Bus: The Current Favourite

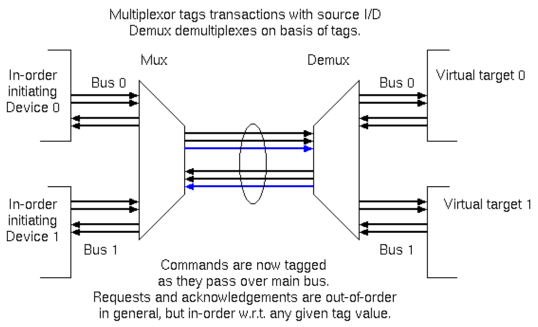


AXI-lite Bus Structure - Five temporally floating sub ports. One AXI port if formed of five separate interfaces that are called channels: two for read, three for write. Each of the five has its own contra-directional READY/VALID pair with all other nets running in the VALID direction and qualified by the conjunction of ready and valid on a clock edge. In simple applications, the address and data channels for write will run close to lockstep, making a more natural total of four logical interfaces. Sequential consistency: the complete decoupling of the read and write aspects immediately raises the prospect of RaW and WaR hazards. A RaW hazard is where a read does not connect with the most-recently written data in the correct/nominal temporal order. AXI can be used with and without (AXI-lite) ordering tokens/tags.

AXI is widely used even for non-ARM products ARM AXI

4.2.7 Supporting out-of-order operation using tags.

Some initiators, particularly out-of-order CPU cores, issue multiple outstanding reads and can do useful work as soon as any of them are serviced. Some targets, particularly DRAM, can provide better performance by servicing requests out-of-order. If we multiplex a pair of in-order busses onto a common bus, yet tag all of the traffic from each bus on the common bus according to its in-order initiator, we have a tagged, out-of-order bus.


Multiplexor tags transactions with source I/D
Demux demultiplexes on basis of tags.
Mux
Demux
In-order initiating Device 0
Bus 0
Bus 0
Virtual target 0
In-order initiating Device 1
Bus 1
Bus 1
Virtual target 1
Commands are now tagged as they pass over main bus.
Requests and acknowledgements are out-of-order in general, but in-order w.r.t. any given tag value.

Out-of-order bus with tags.

The semantics are that for any given tag the requests and replies must be kept in order. The devices on the left may be separate initiator blocks, like processors and DMA controllers, or they may be different load/store ports on a common IP block, or, in theory, any mix. For the targets on the right, there is no difference between demultiplexing to separate in-order targets and a single target that understands tags.

The tags above are used to correlate results to replies over an out-of-order bus. To preserve sequential consistency between, say, separate load/store ports on a CPU, which would have their own IDs a fence mechanism of some sort is also needed. Fences preserve RaW and WaW orderings: no transaction is allowed to overtake others in a way that would make it jump over a fence in the time domain. (In the OCP/BVCI bus, tag numbers are/were used in a different way from AXI: a fence is implied when an initiator increased a tag number.) On the AXI bus there are no fences. Instead, an initiator must just wait for all outstanding responses to come back before issuing a transaction on any of its load/store ports that is after a fence.

For AXI load-linked, store-conditional and other atomic operations, the command fields in the issueing channels contain additional nets and code points that indicate whether the transaction is exclusive in this way.

4.3 RTL: Register Transfer Language

Everybody attending this course is expected to have previously studied RTL coding or at least taught themselves the basics before the course starts.

The Computer Laboratory has an online Verilog course you can follow: Cambridge SystemVerilog TutorPlease note that this now covers ‘System Verilog’ whereas most of my examples are in plain old Verilog. There are a few, unimportant, syntax differences.

RTL is compiled to logic gate instances in a target library using a process called **Logic Synthesis**. RTL is also simulatable pre and post synthesis.

4.3.1 RTL Summary View of Variant Forms.

For the sake 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.
- **2. Lists of pure (unordered) register transfers** where the r.h.s. expressions describe potentially complex logic using a rich set of integer operators, including all those found in software languages such as C++ and Java. There is one list per synchronous clock domain. A list without a clock domain is for combinational logic (continuous assignments).
- **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.

Additional notes:

The numerical value of any time values in RTL are ignored for synthesis. Components are synthesisable whether they have delays in them or not. For zero-delay components 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.

Level 1/3: Structural Verilog: a structural netlist with hierarchy.

```
module subcircuit(input clk, input rst, output q2);
   wire q1, q3, a;
   DFFR Ff_1(clk, rst, a, q1, qb1),
        Ff_2(clk, rst, q1, q2, qb2),
        Ff_3(clk, rst, q2, q3, qb3);
  NOR2 Nor2_1(a, q2, q3);
endmodule
```

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

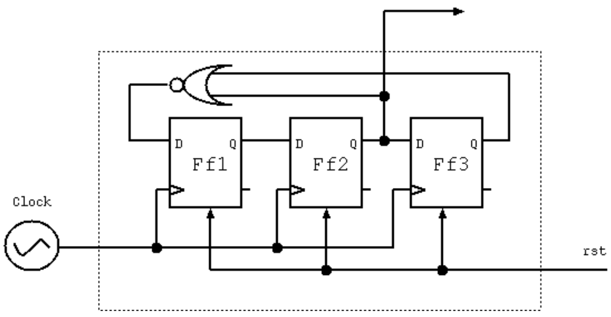


Figure 4.11: The circuit described by our structural example (a divide-by-five, synchronous counter).

All hardware description languages and RTLs contain some sort of **generate statement**. A generate statement is an iterative construct that is executed (elaborated) at compile time to generate multiple instances of a component and its wiring. In the recent Chisel and Bluespec languages, a powerful, higher-order functional language is available, but in SystemVerilog we follow a more mundane style such as:

```
wire dout[39:0];
reg[3:0] values[0:4] = {5, 6, 7, 8, 15};

 generate
   genvar i;
   for (i=0; i < 5; i++)  begin
     MUT mut[i] (
     .out(dout[i*8+7:i*8]),
     .value_in(values[i]),
     .clk(clk),
     );
   end
 endgenerate
```

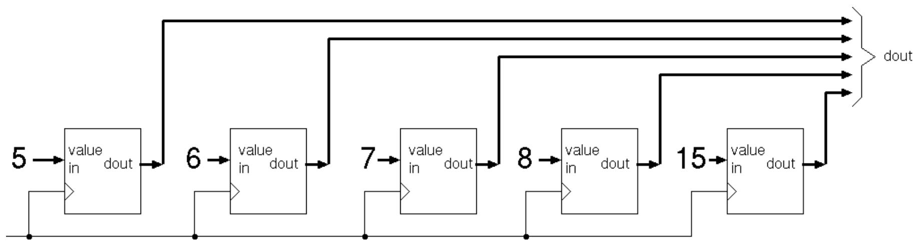
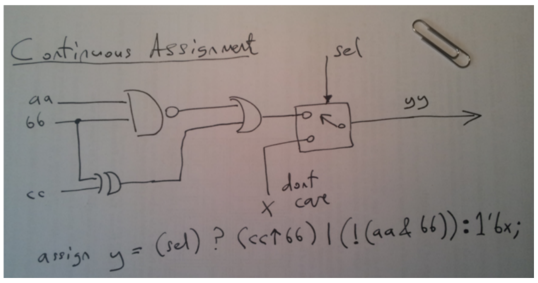


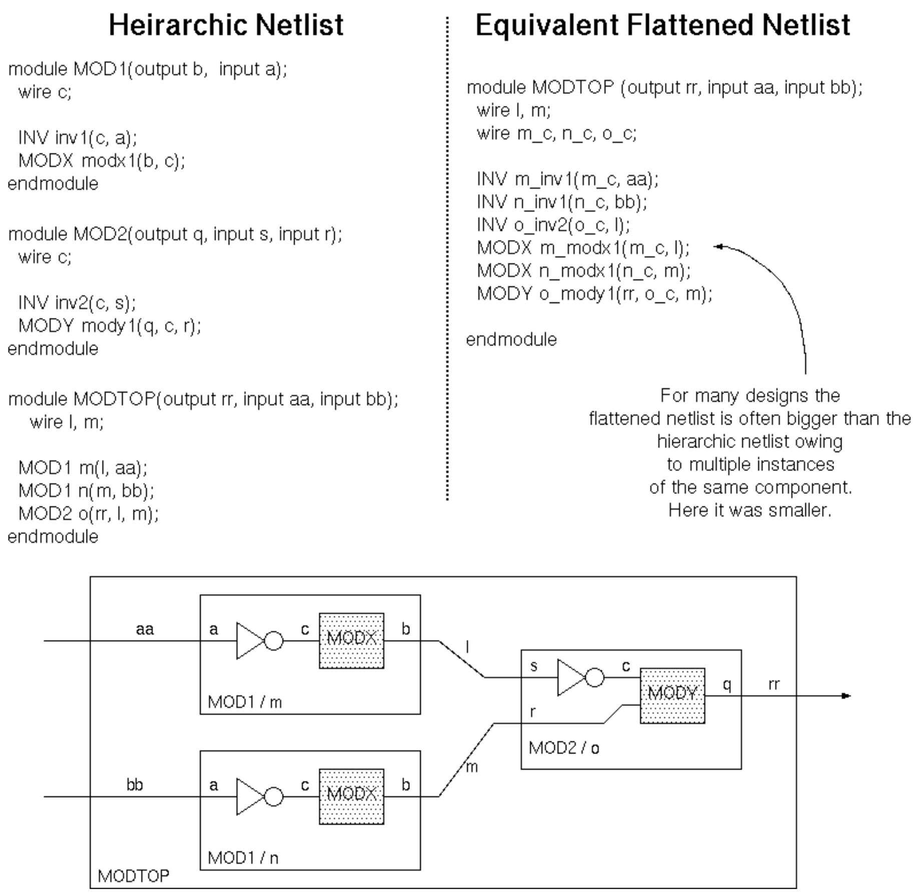
Figure 4.12: Example Generate Statement in RTL.

Figure 4.13 shows structural RTL before and after flattening as well as a circuit diagram showing the component boundaries.

2a/3: Continuous Assignment: an item from a pure RT list without a clock domain.



```
  // Continuous assignments define combinational logic circuit:
  assign a = (g) ? 33 : b * c;
  assign b = d + e;
```



Heirarchic Netlist

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

  INV inv1(c, a);
  MODX modx1(b, c);
endmodule

module MOD2(output q, input s, input r);
  wire c;

  INV inv2(c, s);
  MODY mody1(q, c, r);
endmodule

module MODTOP(output rr, input aa, input bb);
    wire l, m;

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

Equivalent Flattened Netlist

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

Figure 4.13: Example RTL fragment, before and after flattening.

- Order of continuous assignments is un-important,
- Tools often insist that continuous logic is loop-free, otherwise: intentional or un-intentional 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 combinational array writes).

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

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

Two coding styles (it does not matter whether these transfers are each in their own always statement or share over whole clock domain):

```
      always @(posedge clk)  a <= b ? c + d;
      always @(posedge clk)  b <= c - d;
      always @(posedge clk)  c <= 22-c;
```

```
// or
      always @(posedge clk) begin
        a <= b ? c + d;
        b <= c - d;
        c <= 22-c;
      end
```

In System Verilog we would use `always_ff` in the above cases.

Typical example (illustrating pure RT forms):

```
    module CTR16(
      input mainclk,
      input din,
      output o);

      reg [3:0] count, oldcount;

      always @(posedge mainclk) begin
          count <= count + 1;
          if (din) oldcount <= count; // Is 'if' pure ?
          end

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

   endmodule
```

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

In a stricter form of this pure RTL, we cannot use '`if`', so when we want a register to sometime retain its current value we must assign this explicitly, leading to forms like this:

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

3/3: Behavioural RTL: a thread encounters order-sensitive statements.

In 'behavioural' expression, a thread, as found in imperative languages such as C and Java, assigns to variables, makes reference to variables already updated and can re-assign new values.

For example, the following behavioural code (inside an always block)

```
        if (k) foo = y;
        bar = !foo;
```

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

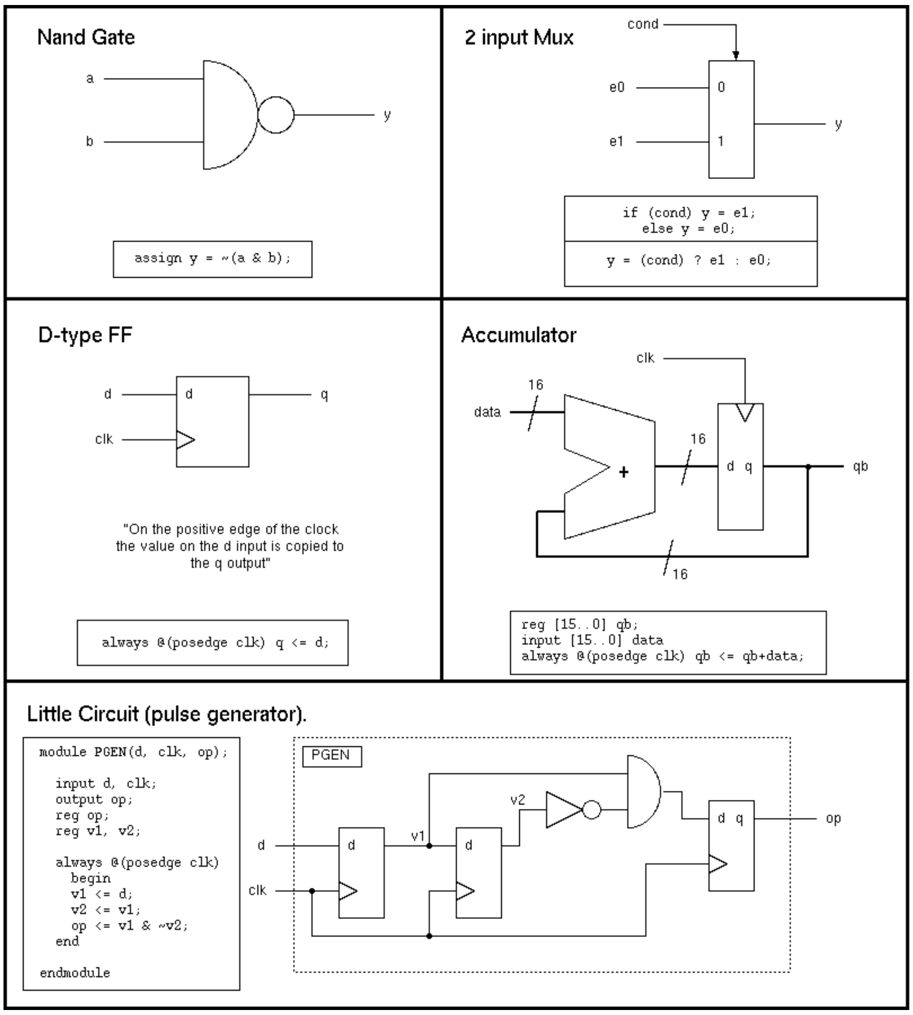
```
        foo <= (k) ? y: foo;
        bar <= !((k) ? y: foo);
```

Figure 4.14 shows synthesisable Verilog fragments as well as the circuits typically generated. The 'little circuit' uses old-style sytnax for input and output designations but this is still valid today.

The RTL languages (Verilog and VDHL) are used both for simulation and synthesis. Any RTL can be simulated but only a subset is standardised as '**synthesisable**' (although synthesis tools can generally handle a slightly larger synthesisable subset).

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.


Nand Gate
a
b
y
2 input Mux
cond
e0
e1
0
1
y
D-type FF
d
clk
d
q
"On the positive edge of the clock the value on the d input is copied to the q output"
Accumulator
clk
16
data
16
+
d q
qb
16
Little Circuit (pulse generator).
PGEN
d
clk
d
v1
d
v2
d q
op

```
assign y = ~(a & b);
```

```
if (cond) y = e1;
else y = e0;
```

```
y = (cond) ? e1 : e0;
```

```
always @(posedge clk) q <= d;
```

```
reg [15..0] qb;
input [15..0] data
always @(posedge clk) qb <= qb+data;
```

```
module PGEN(d, clk, op);

  input d, clk;
  output op;
  reg op;
  reg v1, v2;

  always @(posedge clk)
    begin
    v1 <= d;
    v2 <= v1;
    op <= v1 & ~v2;
  end

endmodule
```

Figure 4.14: Elementary Synthesisable Verilog Constructs

4.3.2 Synthesisable RTL

Additional notes:

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

Expressions:

```
datatype ex_t =                         // Expressions:
    Num of int                          //  integer constants
  | Net of string                       //  net names
  | Not of ex_t                         //  !x    - logical not
  | Neg of ex_t                         //  ~x    - one's complement
  | Query of ex_t * ex_t * ex_t         //  g?t:f - conditional expression
  | Diadic of diop_t * ex_t * ex_t      //  a+b   - diadic operators + - * / << >>
  | Subscript of ex_t * ex_t            //  a[b]  - array subscription, bit selection.
```

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

```
datatype cmd_t =                       // Commands:
    Assign of ex_t * ex_t              //   a = e;  a[x]=e;  - assignments
  | If1 of ex_t * cmd_t                //   if (e) c;        - one-handed IF
  | If2 of ex_t * cmd_t * cmd_t        //   if (e) c; else c - two-handed IF
  | Block of cmd_t list                //   begin c; c; .. end - block
```

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

```
datatype s_t =                         // Top-level forms:
   Sequential of edge_t * ex_t * cmd_t //    always @(posedge e) c;
|  Combinational of ex_t * ex_t        //    assign e1 = e2;
```

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 the structural generation statements, mentioned above, that are fully unwound by the compiler front end and so have no data-dependent exit conditions).

An example of RTL synthesis:

Example input:

```
module TC(clk, cen);
  input clk, cen;
  reg [1:0] count;
  always @(posedge clk) if (cen) count<=count+1;
endmodule
```

Results in structural RTL netlist:

```
module TC(clk, cen);
  wire u10022, u10021, u10020, u10019;
  wire [1:0] count;
  input cen;   input clk;
  CVINV  i10021(u10021, count[0]);
  CVMUX2  i10022(u10022, cen, u10021, count[0]);
  CVDFF  u10023(count[0], u10022, clk, 1'b1, 1'b0, 1'b0);
  CVXOR2  i10019(u10019, count[0], count[1]);
  CVMUX2  i10020(u10020, cen, u10019, count[1]);
  CVDFF  u10024(count[1], u10020, clk, 1'b1, 1'b0, 1'b0);
endmodule
```

Here the behavioural input was converted, by a **Logic Synthesis Compiler**, also known as an **RTL Compiler**, to an implementation technology that included inverters, multiplexors, D-type flip-flops and XOR gates. For each gate, the output is the first-listed terminal.

Verilog RTL Synthesis Algorithm: 3-Step Recipe:

1. First we remove all of the blocking assignment statements to obtain a 'pure' RTL form. For each register we need exactly one assigment (that becomes one hardware circuit for its input) regardless of however many times it is assigned, so we need to build a multiplexor expression that ranges over all its sources and is controlled by the conditions that make the assignment occur.

   For example:

   ```
      if (a) b = c;
      d = b + e;
      if (q) d = 22;
   ```

   is converted to

   ```
      b <= (a) ? c : b;
      d <= q ? 22 : ((a) ? c : b) + e;
   ```

2. For each register that is more than one bit wide we generate separate assignments for each bit.

This is colloquially known as 'bit blasting'. This stage removes arithmetic operators and leaves only boolean operators. For example, if v is three bits wide and a is two bits wide:

```
v <= (a) ? 0: (v>>1)
```

is converted to

```
v[0] <= (a[0]|a[1]) ? 0: v[1];
v[1] <= (a[0]|a[1]) ? 0: v[2];
v[2] <= 0;
```

3. Build a gate-level netlist using components from the selected library of gates. (Similar to a software compiler when it matches operations needed against instruction set.) Sub-expressions are generally reused, rather than rebuilding complete trees. Clearly, logic minimization (Karnaugh maps and Espresso) and multi-level logic techniques (e.g. ripple carry versus fast carry) as well as testability requirements affect the chosen circuit structure. Gate Building, ML fragment

When generating gates a target technology cell library needs to be read in by the logic synthesiser. Likewise, when generating FPGA logic, the details of the CLBs and limitations of the programmable wiring need to be known by the logic synthesiser.

(The details of the algorithms on these links and being able to reproduce them is not examinable but being able to draw an equivalent gate-level circuit for a few lines of RTL is examinable).

Further detail on selected constructs:

Additional notes:

1. How can we make a simple adder ?

The following ML fragment will make a ripple carry adder from lsb-first lists of nets:

```
fun add c (nil, nil) = [c]
|   add c (a::at, b::bt) =
    let val s = gen_xor(a, b)
        val c1 = gen_and(a, b)
        val c2 = gen_and(s, c)
        in (gen_xor(s, c))::(add (gen_or(c2, c1)) (at, bt))
        end
```

2. Can general division be bit-blasted ? Yes, and for some constants it is quite simple.

For instance, division by a constant value of 8 needs no gates - you just need wiring! For dynamic shifts make a **barrel shifter** using a succession of broadside multiplexors, each operated by a different bit of the shifting expression. See link Barrel Shifter, ML fragment.

3. Can we do better for constant divisors? To divide by a constant 10 you can use that 8/10 is 0.11001100 recurring, so if n and q are 32 bit unsigned registers, the following computes n/10:

```
   q = (n >> 1) + (n >> 2);
   q += (q >> 4);
   q += (q >> 8);
   q += (q >> 16);
   return q>>3;
```

4.3.3 Arrays and RAM Inference in RTL

RTL languages support bits, bit vectors (words) and arrays of bit vectors (RAMs). Arrays in the RTL can be synthesised to structural instances of RAM memories or else to register files made of flip flops. Certain patterns of array use are defined to trigger **RAM inference**, where a RAM is instantiated in the net list. RAM inference is supported in FPGA logic synthesis tools. ASIC synthesis tools require the use of the alternativ, which is for the RTL to contain explicit structural instances.

```
   reg [31:0] myram [32767:0];  // 32K words of 32 bits each.
   // To execute the following in one clock cycle needs two RAM ports
   always @(posedge clk) myram[a] <= myram[b] + 2;
```

Even when RAM inference is available, it is sometimes easiest to write a leaf module that behaves like the RAM and then structurally instantiate that in the main RTL. The RAM inference will then just act inside the leaf module and edits to the main RTL cannot violate the pattern that triggers the inference procedure.

The pattern needed to trigger RAM inference is nothing more than an RTL model of the real RAM. This example is for a dual-ported (one read, one write) SRAM. SRAM is synchronous RAM with a read latency. Here the latency is one cycle.

```
  module R1W1RAM(din, waddr, clk, wen, raddr, dout);   // This is both a behavioural model
    input clk, wen;                                    // of the SRAM and a pattern that
    input [14:0] waddr, raddr;                         // should trigger RAM inference in
    input [31:0] din;                                  // FPGA tools.
    output [31:0] dout;

    reg [31:0] myram [32767:0];          // 32K words of 32 bits each.
    always @(posedge clk) begin
         dout <= myram[raddr];           // Data out is registered once without otherwise being
         if (wen) myram[waddr] <= din;   // looked at. Write data in is sychronous with the write
         end                             // address.
endmodule
```

The behavioural model will be replaced with a RAM macrocell in the silicon implementation. Each port of a RAM has an address input. The two essential rules for RAM inference are that

1. there is one expression that is clearly recognisable as the address for each port, and
2. the data read out is registered by the required number of pipeline broadside registers to match the latency of the target technology without any use (peeking) of the data in that pipeline.

Similar rules facilitate automated deployment of other structural resources (or FUs as we shall call them later). One example is the clock-enable flip flop (as per clock gating) and another is multiplier inference. The FPGA's DSP unit, which is essentially a pipelined multiplier, will be deployed where the tools can make sufficient structural matches

4.3.4 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 state machine. This is compilable by some tools. This state machine requires a program counter (PC) register at runtime (implied):

```
   input clk, din;
   output reg [3:0] q; // Four bits of state are define here.

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

How much additional state in the form of PC bits are needed? Is conditional event control synthesisable? Does the output 'q' ever take on the value 4?

As a second non-synthesisable example, consider the dual-edge-triggered flip-flop in Figure 4.15.

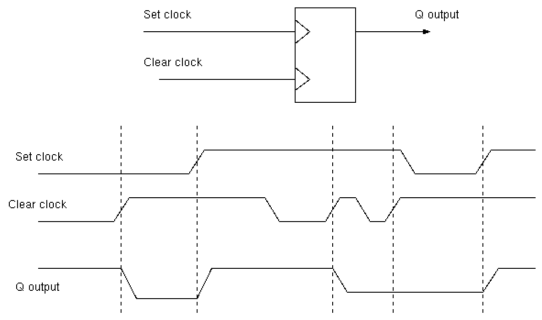


Figure 4.15: Schematic symbol and timing diagram for an edge-triggered RS flop.

```
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 used mainly in specialist phase-locked loops. It can be modelled in Verilog, but is **not** supported for Verilog synthesis. A real implementation typically uses 8 or 12 NAND gates in a relatively complex arrangement. We do not expect general-purpose logic synthesis tools to create such circuits: they were hand-crafted by experts of previous decades.

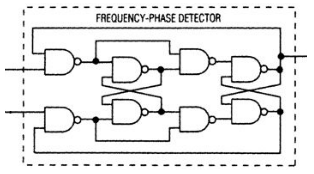


Figure 4.16: Hand-crafted circuit for the edge-triggered RS flop used in practice.

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

```
// Typical RTL testbench contents:

// Set the time in seconds for each clock unit.
‘timescale 1 ns

reg clk, reset;
initial begin clk=0; forever #5 clk = !clk; end   // Clock source 100 MHz
initial begin reset = 1; # 125 reset = 0; end     // Power-on reset generator
```

Take-away summary: The industry has essentially zeroed-in on a very narrow synthesisable RTL subset. Behavioural input forms are essentially ‘syntactic sugar’ that are mapped down to pure RTL before logic minimisation and gate mapping.

4.3.5 Further Logic 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 (with automatic clock gating synthesis),
- Area use,

- Performance,
- Testability (with automatic test modes generation).

Our basic 3-step recipe did not have an optimisation function weighted by such metrics.

Gate libraries have high and low drive stength forms of most gates (see later). The synthesis tool will chose the appropriate gate depending on the fanout and (estimated) net length during routing. Some leaf cells are broadside and do not require bit-blasting.

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

```
   reg[31:0] y;
   ...
   if (e1) y <= e2;
   else if (e3) y <= e4;
   else y <= 32'bx;              // Note, assignment of 'x' permits automated logic minimisation.
```

Can share sub-expressions or re-compute expressions locally. Reuse of sub-expressions is important for locally-derived results, but with today's VLSI, sending a 32 bit addition result more than one millimeter on the chip may use more power then recomputing it locally! Logic synthesis is an underconstrained optimisation problem. Both choosing what cubes to use in a Boolean expression and finding which subexpressions are useful when generating several output functions are exponentially complex. Iteration and hill climbing must be used. Also, we can sometimes re-encode state so that output function is simple to decode (covered in ABD notes not lectured this year but critical for HLS where the controlling sequencer hugely benefits). (The most famous logic synthesiser is Design Compiler from Synopsys which has been used for the majority of today's chips.)

4.4 Simulation

Simulation of real-world systems generally requires quantisation in time and spatial domains.

There are two main forms of simulation modelling:

- FDS: finite-difference time-domain simulation, and
- EDS: event-driven simulation.

Finite-difference simulation is used for analogue and fluid-flow systems. An example is the SPICE simulator used in the Power section of these notes to model an inverter. It is rarely used in SoC design (just for low-level electrical propagation and crosstalk modelling). Variable element size (and variable temporal step size) can be used to make finite-element simulations approximate even-driven behaviour.

```
Finite-element difference equations (without midpoint rule correction):
   tnow += deltaT;
   for (n in ...) i[n] = (v[n-1]-v[n])/R;
   for (n in ...) v[n] += (i[n]-i[n+1])*deltaT/C;
```

Basic finite-difference simulation uses fixed spatial grid (element size is deltaL) and fixed time step (deltaT seconds). Each grid point holds a vector of instantatious local properties, such as voltage, temperature, stress, pressure, magnetic flux. Physical quantities are divided over the grid. Three examples:

1. Sound wave in wire: C=deltaL*mass-per-unit-length, R=deltaL*elasticity-per-unit-length

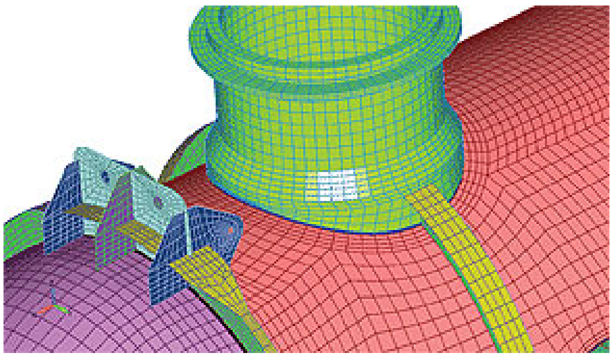


Figure 4.17: Finite Element Grid

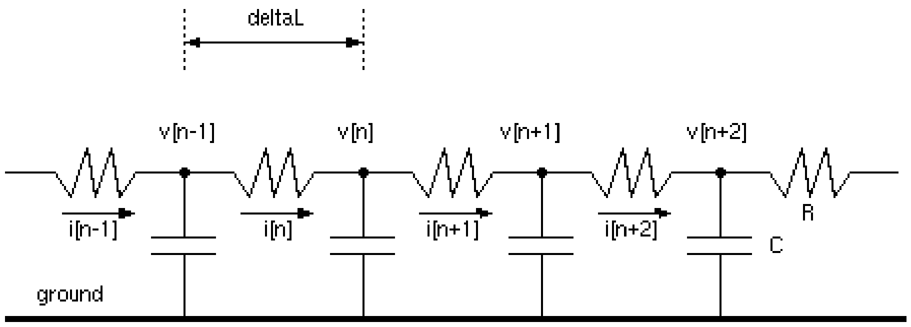


Figure 4.18: Baseline finite-difference model for bidirectional propagation in one dimension.

2. Heat wave in wire: C=deltaL*heat-capacity-per-unit-length, R=deltaL*thermal-conductance-per-unit-length
3. Electrical wave in wire: C=deltaL*capacitance-per-unit-length, R=deltaL*resistance-per-unit-length

Larger modelling errors with larger deltaT and deltaL, but faster simulation. Keep them less than 1/10th wavelength for good accuracy.

Generally use a 2D or 3D grid for fluid modelling: 1D ok for electronics. Typically want to model both resistance and inductance for electrical system. When modelling inductance instead of resistance, then need a '+=' in the $i[n]$ equation. When non-linear components are present (e.g. diodes and FETs), SPICE simulator adjusts deltaT dynamically depending on point in the curve.

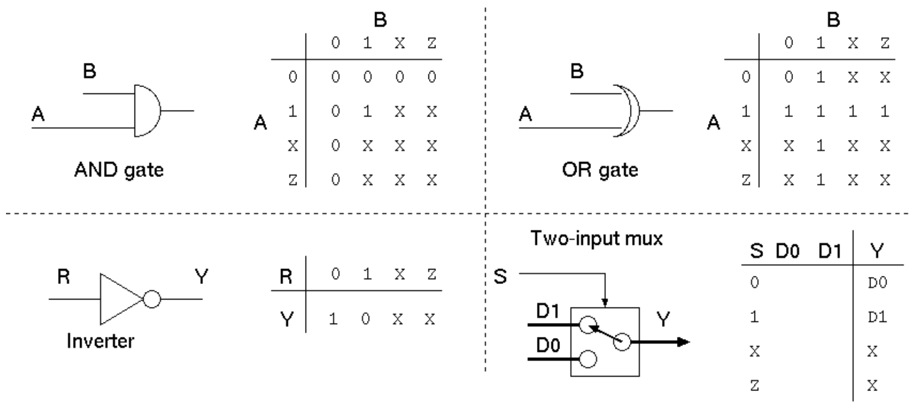
Finite element simulation is not examinable for Part II System-on-Chip.

4.4.1 Digital Logic Modelling

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

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

In this model, nets jump from one value to another in an instant. Real nets have a transit time.



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

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

R	0	1	X	Z
Y	1	0	X	X

S D0 D1	Y
0	D0
1	D1
X	X
Z	X

Figure 4.19: Illustratring the four-value logic level encoding for common gates.

The symbol ‘X’ has a different meaning according to tool applied: it means ‘uncertain’ during simulation and ‘dont-care’ during logic synthesis. The dont-care in logic synthesis enables logic minimisation (as done visually with Karnaugh maps).

Verilog and VHDL generally use more-complex logic than the four-value logic system, with various strengths of logic zero and one so that weak effects such as pull-up resistors and weedy pass transistors can be modelled. This enables a net-resolution function to be applied when a net is driven by more than one source. For instance, an equal drive-strength one and zero will resolve to an X but the stronger will win when strengths are not matched.

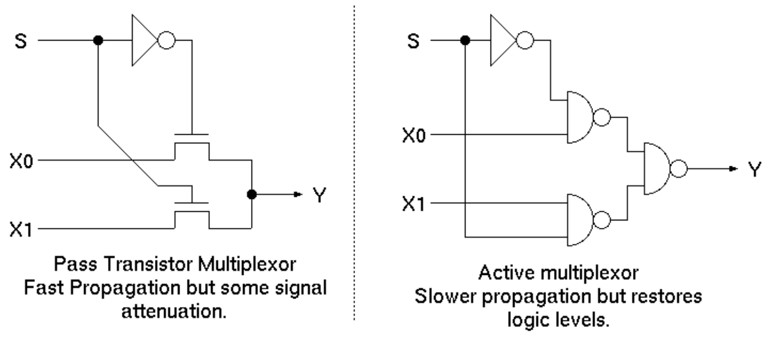

S
X0
X1
Y
Pass Transistor Multiplexor
Fast Propagation but some signal attenuation.
Active multiplexor
Slower propagation but restores logic levels.

Figure 4.20: Pass transistor multiplexor compared with an active path multiplexor that needs more silicon area, especially when the control inverter is amortised over a word.

The **pass transistor** is a cheap (in area terms) and efficient (in delay terms) form of programmable wiring, but it does not amplify the signal.

4.4.2 Event Driven Simulation

The following ML fragment demonstrates the main datastructure for an EDS kernel. EDS ML fragments

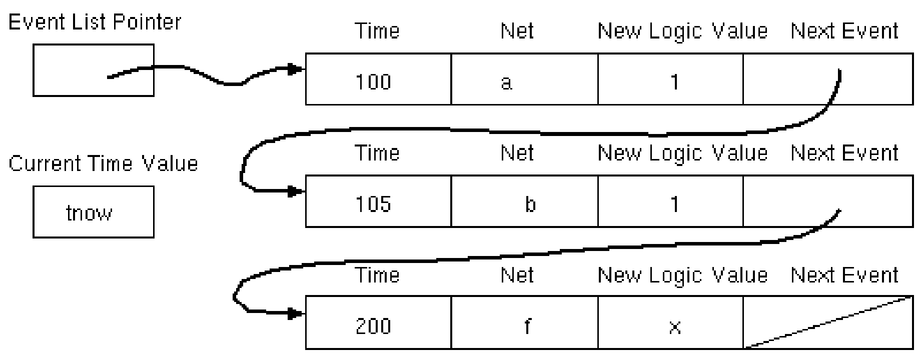


Figure 4.21: Event queue, linked list, sorted in ascending temporal order.

```
// A net has a string name and a width.
// A net may be high z, dont know or contain an integer from 0 up to 2**width - 1.
// A net has a list of driving and reading models.

type value_t = V_n of int | V_z | V_x;

type net_t = {
   net_name:      string;           // Unique name for this net.
   width:         int;              // Width in bits if a bus.
   current_value: value_t ref;      // Current value as read by others
   net_inertia:   int;              // Delay before changing (commonly zero).
   sensitives:    model_t list ref; // Models that must be notified if changed.
};

// An event has a time, a net to change, the new value for that net and an
// optional link to the next on the event queue:
type event_t = EVENT of int * net_t * value_t * event_t option ref
```

This reference implementation of an event-driven simulation (EDS) kernel maintains an ordered queue of events commonly called the **event list**. The current simulation time, **tnow**, is defined as the time of the event at the head of this queue. An event is a change in value of a net at some time in the future. Operation takes the next event from the head of the queue and dispatches it. Dispatch means changing the net to that value and chaining to the next event. All component models that are sensitive to changes on that net then run, potentially generating new events that are inserted into the event queue.

The full version of these notes covers two variations on the basic EDS algorithm: intertial delay and delta cycles. But these are not examinable CST 15/16 onwards.

Code fragments (*details not examinable*):
Create initial, empty event list:

```
val eventlist = ref [];
```

Constructor for a new event: insert at correct point in the sorted event list:

```
fun create_and_insert_event(time, net, value) =
    let fun ins e = case !e of
                (A as EMPTY) => e := EVENT(time, net, value, ref A)
              | (A as EVENT(t, n, v, e')) => if (t > time)
                   then e := EVENT(time, net, value, ref A)
                   else ins e'
        in ins eventlist
        end
```

Main simulation: keep dispatching until event list empty:

```
fun dispatch_one_event() =
        if (!eventlist = EMPTY) then print("simulation finished - no more events\n")
        else let val EVENT(time, net, value, e') = !eventlist in
        ( eventlist := !e';
          tnow := time;
          app execute_model (net_setvalue(net, value))
        ) end
```

4.4.3 Mixed Analog/Digital Simulation (Verilog-AMS)

Analogue and Mixed simulation is not examinable for Part II CST.

Hybrid System simulations: typically a digital (embedded) controller interacts with analogue plant.

Example: Vehicle hybrid power system and automatic braking system.

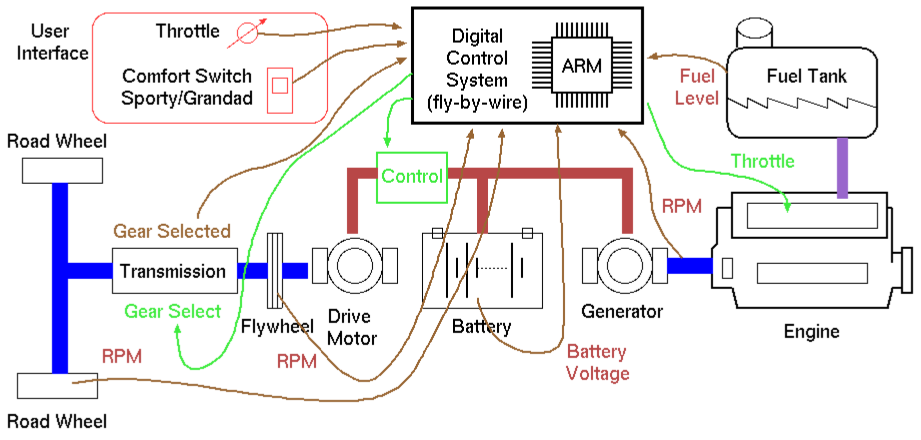


Figure 4.22: Hybrid Automobile Transmission System.

We need to be able to combine digital (event-driven) simulation with analogue models, such as fluid-flow nodal analysis and FDTD modelling.

Cyberphysical tends: Cyberphysical Functional Mock Up InterfaceRelevant tools: Verilog-AMS, VHDL-AMS, Modelica, Simulink and FMI. Wikipedia: Verilog-AMS

Verilog Analogue and Mixed Signal (AMS) extends RTL to support:

- Signals of both analogue and digital types can be declared in the same module.
- Initial, always and **analogue procedural blocks** can appear in the same module.
- Digital signal values can be set (write operations) from any context outside of an analogue procedural block
- Analogue potentials and flows can only receive contributions (write operations) from inside an analogue procedural block
- An **analog initial begin ... end** statement sets up initial voltages on capacitors or levels in a fuel tank.
- A new sensitivity enables triggering actions **always @(cross(Vt1 - 2.5)) begin ... end** .

Examples:

```
// Three 1.5 cells in series make a 4.5 volt battery.
module Battery4V5(input voltage anode, output voltage cathode);
  voltage t1, t2;
  analog begin
    V(anode) <+ 1.5 + V(t2);
    V(t2) <+ 1.5 + V(t1);
    V(t2) <+ 1.5 + V(cathode);
  end
endmodule
```

```
module resistor (inout electrical a, inout electrical b);
  parameter real R = 4700;
  analog V(a,b) <+ R * I(a,b);
endmodule
```

Verilog simulation cycle extended to support solving nodal flow equations. When we potentially de-queue a time-advancing event from EDS queue we first roll forward the FDTD simulations which themselves may contain ‘cross’ and similar sensitivity that insert new events on the EDS queue.

4.4.4 Mixed Analog/Digital Simulation: An interesting problem attributable to Zeno?

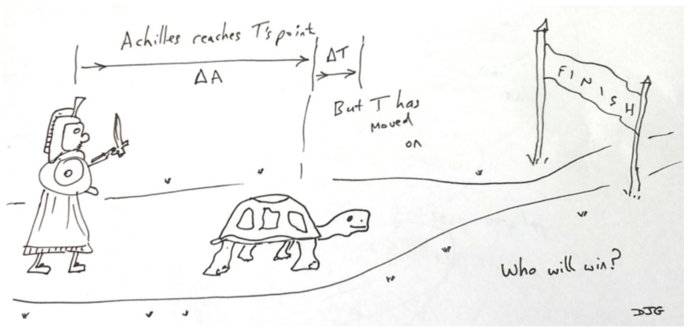


Figure 4.23: Archilles can never catch the tortoise? Sum of GP never converges?

Problem: Remember Zeno’s paradox, with Achilles and the tortoise? *And so you can never catch up, the Tortoise concluded sympathetically.* AMS simulations often suffer from unintentionally revisiting that story. A Zeno hybrid system model is a hybrid system with an execution that takes an infinite number of discrete transitions during a finite time interval.

```
// AMS simulation of of a ball bouncing -> infinite bounce frequency!
module ballbounce();
  real height, velocity;

  analog initial begin height = 10.0; velocity = 0.0; end

  analog begin // We want auto-timestep selection for this FDTD
     height   <+ -velocity;   // Falling downwards
     velocity <+ 9.8;         // Acceleration due to gravity.
  end

  // We want discrete event triggered execution here
  always @(cross height) begin
        velocity = -0.9 * velocity; // Inelastic bounce
        height = 0.000001;          // Hmmm some fudge here!
        end
endmodule
// NB: Precise syntax above may not be accepted by all tools.
```

Enclosing the Behavior of a Hybrid System up to and Beyond a Zeno Point: Michael Konecny et al—

A simple heuristic on the minimum timestep competes directly with adaptive timestep tuning needed to accurately model critical inflection points. Zeno suppression research is ongoing.

Note, the ball bounce example does not involve any nodal equations but the problem is fairly common with real-world examples that do tend to have wire and pipes splitting and joining.

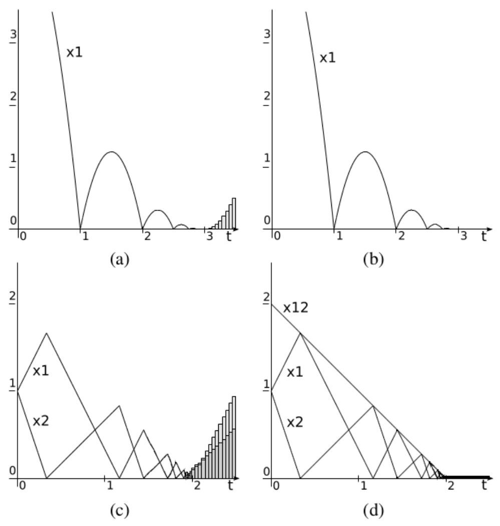


Fig. 2: Enclosures for systems with Zeno behavior. Figures (a) and (c) show the results for the bouncing ball and water tank systems of Lygeros [33, Figures 3.1 and 3.7]. However, we make these models more challenging to simulate by replacing some inequalities with equalities (such as the bouncing conditions). Figures (b) and (d) show the results for the examples when additional (redundant) constraints are added to achieve more precise enclosures.

Figure 4.24: Bounding boxes in one approach to useful simulation of Zeno-generating systems (Konecny).

4.4.5 Higher-level Simulation

Simulating RTL is slow. Every net (wire) in the design is modelled as a shared variable. When one component writes a value, the overheads of waking up the receiving component(s) may be severe. The event-driven simulator kernel contains an indirect jump instruction which itself is very slow on modern computer architectures since it will not get predicted correctly.

Much faster simulation is achieved by disregarding the clock and making so-called TLM calls between the components. Subroutine calls made between objects convey all the required information. Synchronisation is achieved via the call and its return. This is discussed in the ESL section of this course.

4.5 Hazards

Definitions (some authors vary slightly):

- **WaW hazard** - write-after-write: one write must occur after another otherwise the wrong answer persists,
- **RaW or WaR hazard** - write and read of a location are accidentally permuted,
- **Other Data hazard** - when an operand simply has not arrived in time for use,
- **Control hazard** - when it is not yet clear whether the results of operation should be committed (computation might still start speculatively),
- **Name Alias hazard** - we do not know if two array subscripts are equal,
- **Structural hazard** - insufficient physical resources to do everything at once.

(Where the address to a register file has not yet arrived we have a data hazard on the address itself, but this could be regarded as a control hazard for the register file operation itself (read or write).)

We have a structural hazard when an operation cannot proceed because a resource is already in use. Resources that might present structural hazards are:

- Memories and register files with insufficient ports,
- Memories with variable latency, especially DRAM,
- Insufficient number of ALUs for all the arithmetic to be schedulled in current clock tick,
- Anything **non-fully pipelined** i.e. something that goes busy, such as long multiplication (e.g. Booth Multiplier or division or a floating point unit).

A **fully-pipelined** component can start a new operation on every clock cycle. It will have fixed latency (pipeline delay). These are commonly encounted and are easiest to form schedules around. A non-fully pipelined components generally have handshake wires that start it and inform the client logic when it is busy. This is needed for computations better performed with variable latency. Another form that is non-fully pipelined has a **reinitiation inverval** greater than one: for example, it might accept new data every third clock cycle, but still be fixed-latency.

Synchronous RAMs, and most complex ALUs excluding divide, are generally fully pipelined and fixed-latency.

An example of a component that cannot accept new input data every clock cycle (i.e. something that is non-fully-pipelined) is a sequential long multiplier, that works as follows:

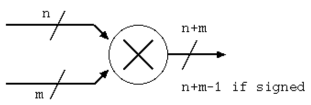
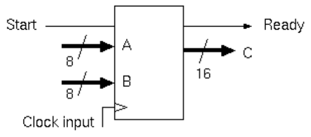


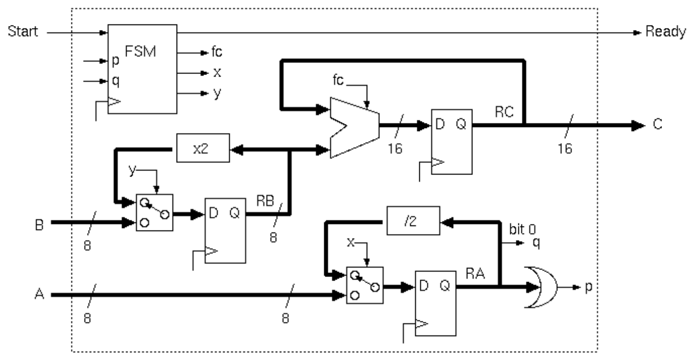
Figure 4.25: Multiplier schematic symbol.



Behavioural algorithm:

```
while (1)
{
   wait (Start);
   RA=A; RB=B; RC=0;
   while(RA>0)
   {
     if odd(RA) RC=RC+RB;
     RA = RA >> 1;
     RB = RB << 1;
   }
   Ready = 1;
   wait(!Start);
   Ready = 0;
 }

(Either HLS or hand coding can
give the illustrateddatapath
and sequencer structure:)
```



This implements conventional long multiplication. It is certainly not fully-pipelined, it goes busy for many cycles, depening on the log of the A input. The illustration shows a common design pattern consisting of a **datapath** and a **sequencer**. Booth's algorithm (see additional material) is faster, still using one adder but needing half the clock ticks.

4.5.1 Hazards From Array Memories

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:

Structural hazard sources are numbered:

```
always @(posedge clk) begin

   q0 <= Boz[e3]           // 3

   q1 <= Foo[e0] + Foo[e1]; // 1

   q2 <= Bar[Bar[e2]];      // 2

   q3 <= a*b + c*d;         // 4

   q4 <= Boz[e4]            // 3
   end
```

1. The RAMs or register files Foo Bar and Boz might not have two read ports.
2. Even with two ports, can Bar perform the double subscription in one clock cycle?
3. Read operations on Boz might be a long way apart in the code, so hazard is hard to spot.
4. 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.

RAMs have a small number of ports but when RTL arrays are held in RAM it is easy to write RTL expressions that require many operations on the contents of a RAM in one operation, even from within one thread. For instance we might need three operations on a RAM to implement

```
A[x] <= A[y + A[z]]
```

Because RTL is a very-low-level language, RTL typically requires the user to do manual schedulling of port use. (However, some current FPGA tools do a certain amount of schedulling for the user.)

Multipliers and floating point units also typically present hazards.

To overcome hazards automatically, stalls and holding registers must be inserted. The programmer's original model of the design must be stalled as ports are re-used in the time domain, using extra clock cycles to copy data to and from the holding registers. This is not a feature of standard RTL so it must either be done by hand or automatically (see HLS section of this course).

Expanding blocking assignments can lead to **name alias** hazards:

Suppose we know nothing about *xx* and *yy*, then consider:

```
begin
   ...
   if (g) Foo[xx] = e1;
   r2 = Foo[yy];
```

To avoid **name alias** problems, this must be compiled to non-blocking pure RTL as:

```
begin
   ...
   Foo[xx] <= (g) ? e1: Foo[xx];
   r2 <= (xx==yy) ? ((g) ? e1:  Foo[xx]): Foo[yy];
```

Quite commonly we do know something about the subscript expressions. If they are compile-time constants, we can decidedly check the equality at compile time. Suppose that at ... or elsewhere beforehand we had the line '`yy = xx+1;`' or equivalent knowledge? Then with sufficient rules we can realise at compile time they will never alias. However, no set of rules will be complete (decidability). And commonly they are a linear function of a loop variable of an enclosing loop (an induction expression) and, after strength reduction, the `xx+k` pattern is readily manifest.

4.5.2 Overcoming Structural Hazards using Holding Registers

One way to overcome a structural hazard is to deploy more resources. These will suffer correspondingly less contention. For instance, we might have 3 multipliers instead of 1. This is the **spatial** solution. For RAMs and register files we need to add more ports to them or mirror them (i.e. ensure the same data is written to each copy).

In the **temporal solution**, a **holding register** is commonly inserted to overcome a structural hazard (by hand or by a high-level synthesis tool). Sometimes, the value that is needed is always available elsewhere in the design (and needs forwarding) or sometimes an extra sequencer step is needed.

If we know nothing about $e0$ and $e1$:

```
always @(posedge clk) begin
   ...
   ans = Foo[e0] + Foo[e1];
   ...
   end
```

then load holding register in additional cycle:

```
always @(posedge clk) begin
   pc = !pc;
   ...
   if (!pc) holding <= Foo[e0];
   if (pc)  ans <= holding + Foo[e1];
   ...
   end
```

If we can analyse the pattern of $e0$ and $e1$:

```
always @(posedge clk) begin
   ...
   ee = ee + 1;
   ...
   ans = Foo[ee] + Foo[ee-1];
   ...
   end
```

then, apart from first cycle, use holding register to forward value from previous iteration (**loop forwarding**):

```
always @(posedge clk) begin
   ...
   ee <= ee + 1;
   holding <= Foo[ee];
   ans <= holding + Foo[ee];
   ...
   end
```

We can implement the program counter and holding registers as source-to-source transformations, that eliminate hazards, as just illustrated. One algorithm is to first to emit behavioural RTL and then to alternate the conversion to pure form and hazard avoidance rewriting processes until closure.

For example, the first example can be converted to old-style behavioural RTL that has an implicit program counter (state machine) as follows:

```
always @(posedge clk) begin
   holding <= Foo[e0];
   @(posedge clk) ;
   ans <= holding + Foo[e1];
   end
```

The transformations illustrated above are NOT performed by mainstream RTL compilers today: instead they are incorporated in HLS tools such as Kiwi. KiwiC Structural Hazard ExampleSharing 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 critical path timing delays.

These example fragments handled one hazard and used two clock cycles. They were localised transformations. When there are a large number of clock cycles, memories and ALUs involved, a global search and optimise procedure is needed to find a good balance of load on structural components. Although these examples mainly use memories, other significant structural resources, such as fixed and floating point ALUs also present hazards.

4.6 Folding, Retiming & Recoding

Generally we have to chose between high performance or low power. (We can see this also in the selection of drive strengths for standard cell gates.) 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).

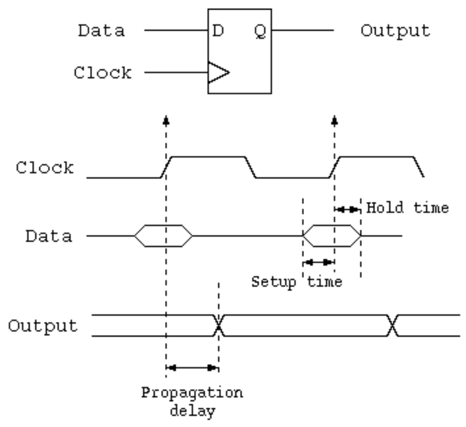
```
LOOPED (time) option:                | UNWOUND (space) option:
                                     |
for (i=0; i < 3 and i < limit; i++)  |  if (0 < limit) sum += data[0] * coef[j];
   sum += data[i] * coef[i+j];       |  if (1 < limit) sum += data[1] * coef[1+j];
                              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

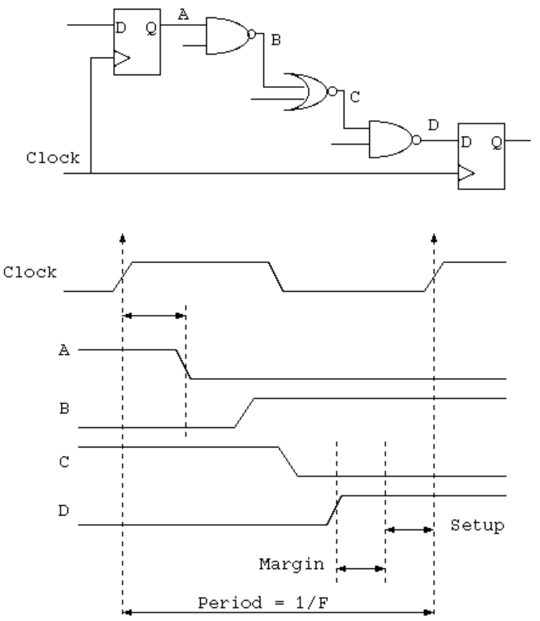
The '+=' operator is an **associative reduction** operator. When the only interactions between loop iterations are outputs via such an operator, the loop iterations can be executed in parallel. On the other hand, if one iteration stores to a variable that is read by the next iteration or affects the loop exit condition then unwinding possibilities are reduced.

We can **retime** a design with and without changing its state encoding. We will see that adding a pipeline stage can increase the amount of state without **recoding** existing state. *Note: some of this material is/would be better presented in the HLS section of the course, now it exists!*

4.6.1 Critical Path Timing Delay

Meeting **timing closure** is the process of manipulating a design to meet its target clock rate (as set by the Marketing Department for instance).





The maximum clock frequency of a synchronous clock domain is set by its critical path. The longest path of combinational logic must have settled before the setup time of any flip-flop starts.

Pipelining is a commonly-used technique to boost system performance. Introducing a pipeline stage increases latency but also the maximum clock frequency. Fortunately, many applications are tolerant to the processing delay of a logic subsystem. Consider a decoder for a fibre optic signal: the fibre might be many kilometers long and a few additional clock cycles in the decoder increase the processing delay by

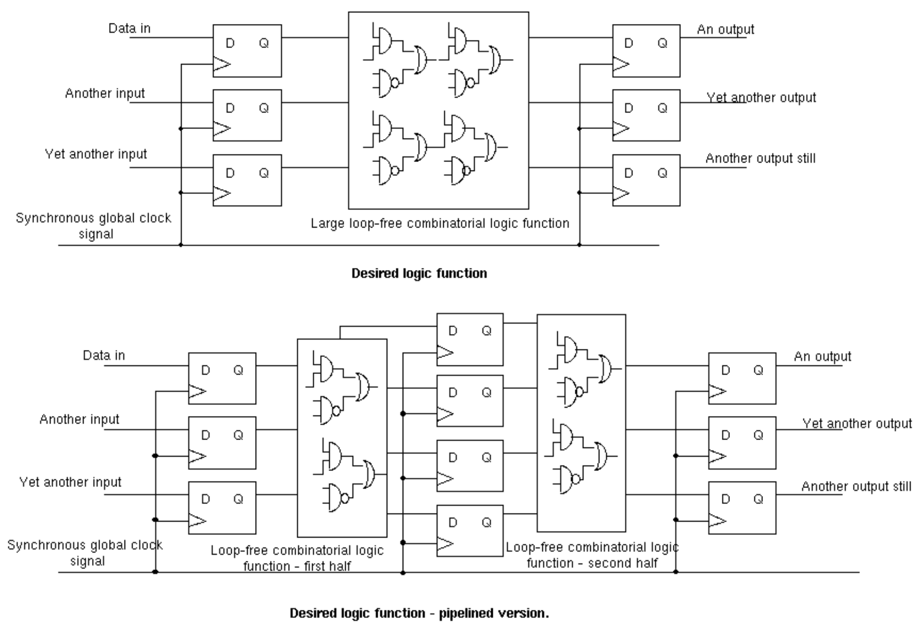

Data in
Another input
Yet another input
Synchronous global clock signal
Large loop-free combinatorial logic function
An output
Yet another output
Another output still
Desired logic function
Loop-free combinatorial logic function - first half
Loop-free combinatorial logic function - second half
Desired logic function - pipelined version.

Figure 4.26: A circuit before and after insertion of an additional pipeline stage.

an amount equivalent to a few coding symbol wavelengths: e.g. 20 cm per pipeline stage for a 1 Gbaud modulation.

Pipelining introduces new state but does not require existing state flip-flops to change meaning.

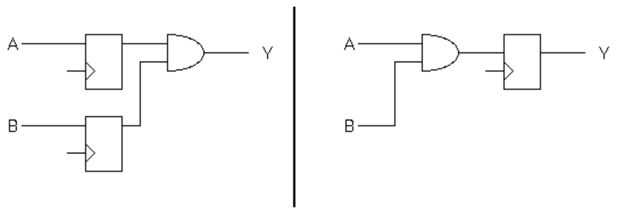


Figure 4.27: Flip-flop migration: two circuits of identical behaviour, but different state encoding.

Flip-flop migration does alter state encoding. Migration may be manually turned on or off during logic synthesis by typical RTL compiler tools. It exchanges delay in one path for delay in another - aim to achieve balance. A sequence of such transformations can lead to a shorter critical path overall.

In the following example, the first migration is a local transformation that has no global consequences:

```
Before:                Migration 1:             Migration 2 (non causal):
  a <= b + c;            b1 <= b; c1 <= c;          q1 <= (dd) ? (b+c): 0;
  q <= (d) ? a:0;        q <= (d) ? b1+c1:0;        q  <= q1;
```

The second migration, that attempts to perform the multiplexing one cycle earlier will require an earlier version of `d`, here termed `dd` that might not be available (e.g. if it were an external input we need knowledge of the future). An earlier version of a given input can sometimes be obtain by delaying all of the inputs (think of delaying all the inputs to a bookmakers shop), but this cannot be done for certain applications where system response time (in-to-out delay) is critical.

Problems arising:

- Circuits containing loops (proper synchronous loops) cannot be pushed very far (for example, the control hazard in a RISC pipeline).

- 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.
- Many structures, including RAMs and ALUs, have a pipeline delay (or several), so the hazard on their input port needs resolving in a different clock cycle from hazards involving their result values.

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

Other rewrites commonly used: automatically recode for one-hot or gray encoding, or invert for reset as preset. Large FSMs are generally recoded by FPGA tools by default so that the output function is easy to generate. This is critical for good performance with complex HLS sequencers.

4.6.2 Static Timing Analyser Tool

A static analysis tool does not run a program or simulate a design - instead it 'stares' at the source code.

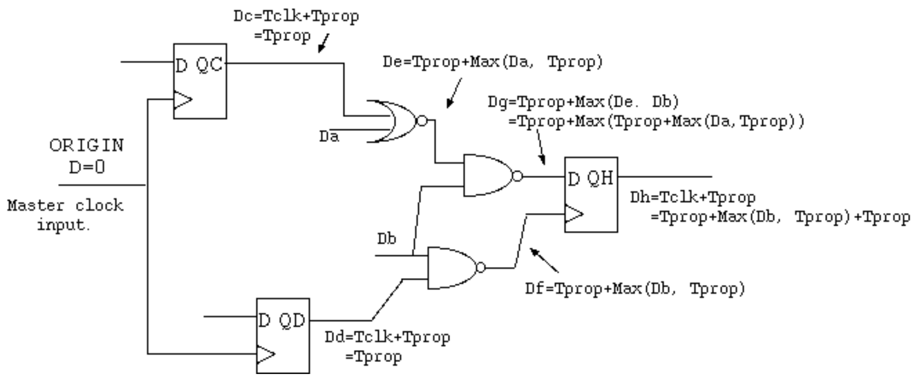


Figure 4.28: An example circuit with static timing annotations

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, a tool may typically keep separate track of high-to-low and low-to-high delays.

4.6.3 Back Annotation and Timing Closure

Once the system has been placed and routed, the length and type of each conductor is known. These facts allow fairly accurate delay models of the conductors to be generated (Section 1.1.5).

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

The marketing department have commonly pre-sold the product with an advertised clock frequency. Making the actual product work at this frequency is known as meeting **timing closure**.

With low-level RTL, the normal means to achieve timing closure is to migrate logic either side of an existing register or else to add a new register - but not all protocols are suitable for registering (Section 4.2.5).

With transactional interfaces, a one-place FIFO can help with timing closure.

4.6.4 Conventional RTL Compared with Software

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

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

RTL is not as expressive for algorithms or data 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.

Verilog and VHDL do not express when a register is **live** with data - hence automatic refactoring and certain correctness proofs are impossible.

For programmers wanting conventional software paradigms, High-Level Synthesis (HLS) should be applide to the high-level program to produce RTL.