System On Chip Design And Modelling

CST Part II, 12 lectures

Lent Term 2009

Dr. David Greaves

David.Greaves@cl.cam.ac.uk

Reading material

- These slides plus the ADDITIONAL MATERIAL

- * OSCI. *SystemC tutorials and whitepapers*. Download from OSCI www.systemc.org and examples from course web site.

- *Transaction-Level Modeling with SystemC: TLM Concepts and Applications for Embedded Systems*. Frank Ghenassia. Springer (2006).

- *A Practical Introduction to PSL*. Cindy Eisner, Dana Fisman. Springer 2006. (Series on Integrated Circuits and Systems).

- *Creating Assertion-Based IP*. Harry D. Foster and Adam C. Krolnik. Springer (Series on Integrated Circuits and Systems).

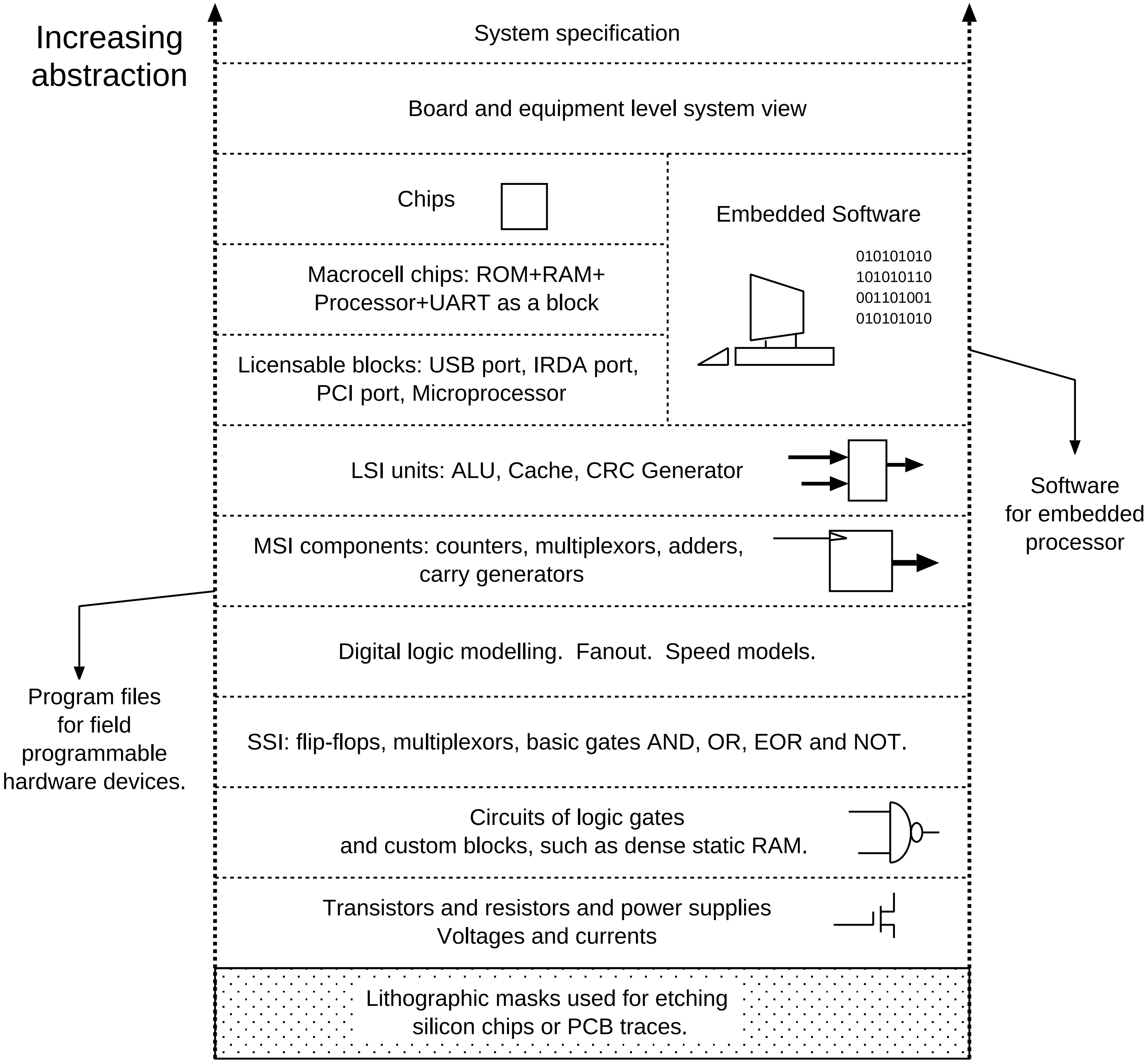
- *System Design with SystemC* Springer. Grotket, Liao, Martin and Swan.

- *Modern VLSI Design (SoC Design)* W Wolf. Pearson Education.

- The Web: `http://www.flylogic.com/` `http://www.systemc.org/` `http://www.spiritconsortium.org/` `http://www.design-reuse.com/articles/` `http://www.esperan.com/tutorial/psl_simple.html`

Course Pre-requisites

- Computer Design (Ib)
  - Some Assembler (e.g. MIPS/ARM/ARC/x86)
  - Verilog ECAD & Architecture
  - Preferably C/C++
  - Operating Systems & Computer Architecture


Increasing
abstraction
System specification
Board and equipment level system view
Chips
Macrocell chips: ROM+RAM+
Processor+UART as a block
Licensable blocks: USB port, IRDA port,
PCI port, Microprocessor
Embedded Software
010101010
101010110
001101001
010101010
Software
for embedded
processor
LSI units: ALU, Cache, CRC Generator
MSI components: counters, multiplexors, adders,
carry generators
Program files
for field
programmable
hardware devices.
Digital logic modelling. Fanout. Speed models.
SSI: flip-flops, multiplexors, basic gates AND, OR, EOR and NOT.
Circuits of logic gates
and custom blocks, such as dense static RAM.
Transistors and resistors and power supplies
Voltages and currents
Lithographic masks used for etching
silicon chips or PCB traces.

L1: Verilog RTL Design: Simulation and Synthesis.

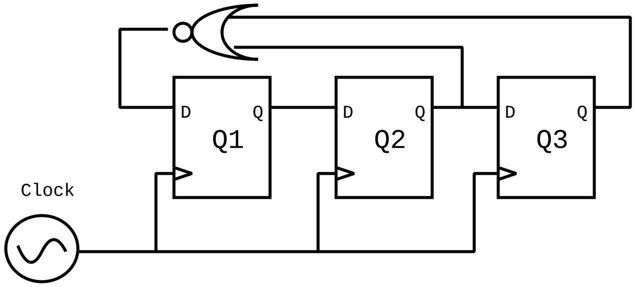
Topic: L1 Basic RTL to logic gates: styles, simulation and synthesis.

- LG1.1 - Basic RTL
- LG1.2 - Structural, Combinational, Behavioural
- LG1.3 - RTL abstract syntax
- LG1.4 - Compute/Commit Cycle
- LG1.5 - Event Driven Simulation Kernel
- LG1.6 - RTL internal forms
- LG1.7 - Basic Synthesis Algorithm
- LG1.8 - Adder Synthesis
- LG1.9 - RAM Memories
- LG1.10 - Structural Hazards
- LG1.11 - Retiming
- LG1.12 - RTL Compared with Software
- LG1.13 - Long Multiplication

LG1.1 - Basic RTL 1/3, Structural

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;
```



Just a netlist.

No registers are transferred by assignment!

LG1.1c - RTL Flattening

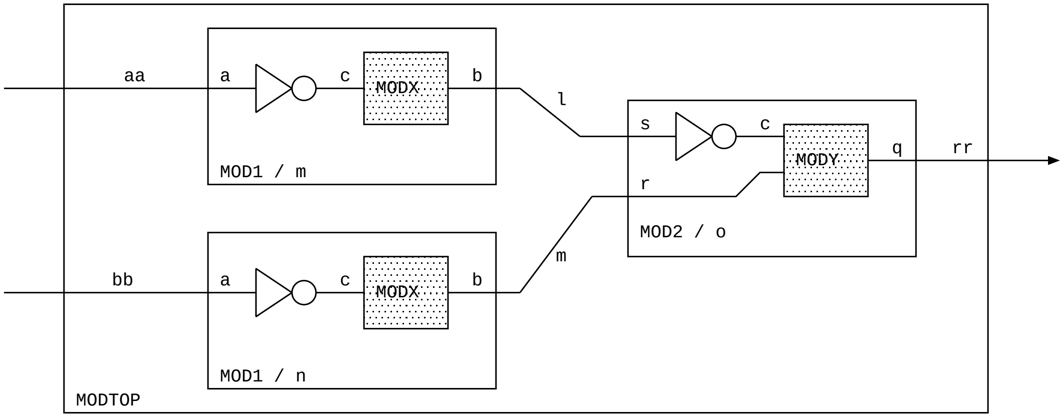
Heirarchic Netlist

```
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

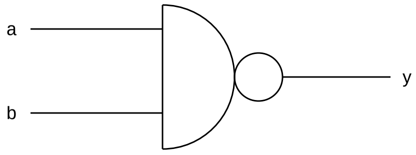
```
module MODTOP (rr, aa, bb);
  input aa, bb; output rr;
  wire l, m;
  wire m_c, n_c, o_c;

  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.



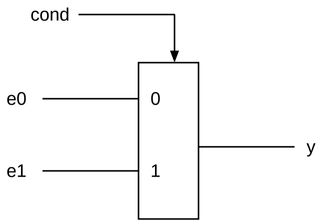
LG1.1c - Elementary Constructs

Nand Gate



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

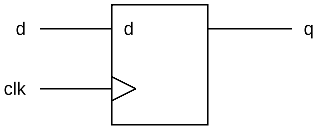
2 input Mux



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

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

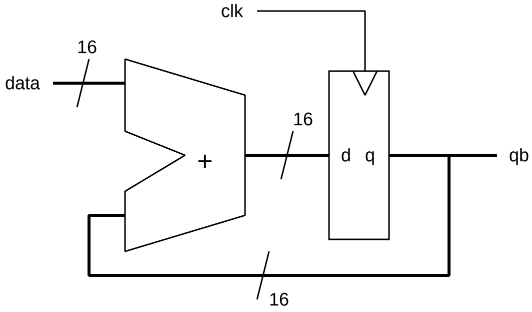
D-type FF



"On the positive edge of the clock the value on the d input is copied to the q output"

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

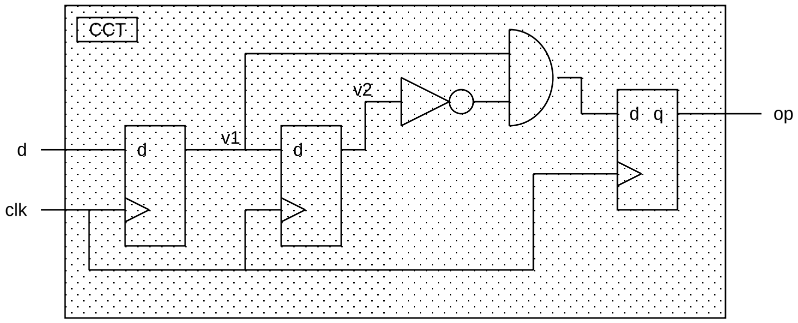
Accumulator



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

Little Circuit (pulse generator).

```
module |CCT(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
```



LG1.1c Basic RTL 2/3, RTL with 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.

Combinational logic (continuous assign) has no explicit clock domain.

If we do not assign a register, it retains its old value:

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

L1.1c - Basic RTL 3/3, Behavioural

Behavioural RTL resembles software.

A behavioural thread references variables already updated.

The order of the statements has an effect!

The following behavioural code

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

can be compiled down to the following unordered RTL

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

Not all RTL is classed as ‘*synthesisable*’.

L1.1c - Behavioural - ‘Non-Synthesisable’ RTL

RTL with event control in the body of a thread defines a statemachine.

The statemachine requires a 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
```

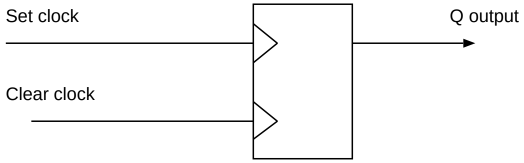
How many bits of PC are needed ?

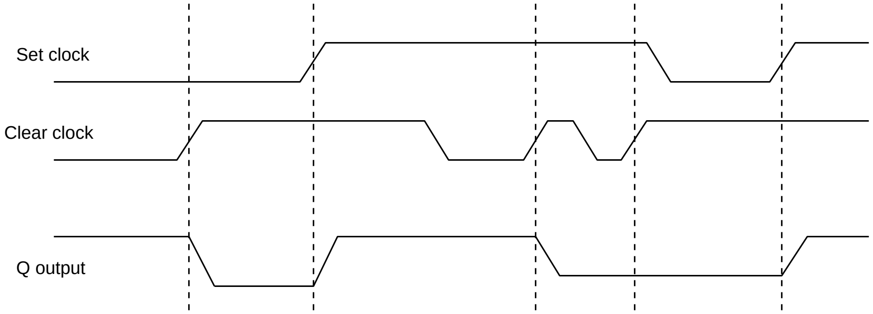
Is conditional event control synthesisable ?

Does the output q ever take on the value 4 ?

L1.1c - Behavioural - ‘Non-Synthesisable’ RTL continued

Consider the dual-edge-triggered flip-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.

Can be modelled in Verilog, but not part of Verilog synthesis.

LG1.1c - Non-synth continued.

Test bench 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:

- fork/join
- Variable update by more than one thread

Finite state is all that should matter!

LG1.2 - RTL Forms, Summary.

Summary:

Verilog RTL allows these levels to be mixed within one module.

```
  1.  Structural  - a hierarchial net list form

  2.  Un-ordered RTL - complex RHS expressions.

  3.  Behavioural - follows flow of program counter.
```

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

Synthesis starts from a root lower in the hierarchy.

Synthesisable code uses synthesisable subset!

Later topics: SystemC, Bluespec and C-to-RTL flows.

LG1.3 - SRTL abstract syntax

Synthesisable RTL - Zero Delay.

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:

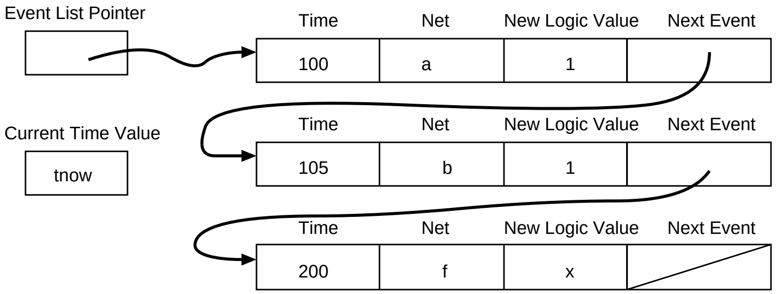
```
datatype s_t =
   Sequential of edge_t * ex_t * cmd_t
|  Combinational of ex_t * ex_t
```

LG1.5 - Event Driven Simulation Kernel

Datastructure for the model and netlist.

```
(*  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.
 *  A model has a unique instance name, a delay, a form and a list of nets it contacts.
 *  It also may have some internal state, held in the last field.
 *)
datatype value_t = V_n of int | V_z | V_x;

datatype m_t = M_AND | M_OR | M_INV | M_XOR | M_DFF | M_BUFIF | M_TLATCH | M_CLOCK;

datatype internal_state_t =
  IS_NONE
| IS_DFF of value_t ref
| IS_CLOCK of int ref
;

datatype
   net_t = NET of value_t ref * string * int * model_t list ref * model_t list ref

and
  model_t = MODEL of string * int * m_t * net_t list * internal_state_t
;
```

LG1.5c - Event Driven Simulation Kernel



Event List Pointer

Current Time Value

tnow

Time	Net	New Logic Value	Next Event
100	a	1	

Time	Net	New Logic Value	Next Event
105	b	1	

Time	Net	New Logic Value	Next Event
200	f	x	

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

```
fun event(time, net, value) =
    let fun a 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 a e’
        in a eventlist
        end
```

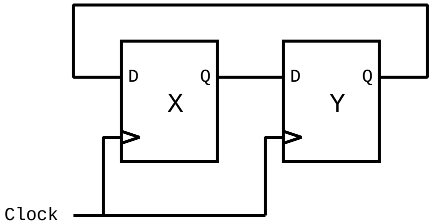
Main simulation: keep dispatching until event list empty:

```
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 example_models (net_setvalue(net, value))
        ) end
```

LG1.4 - Compute/Commit Cycle

Hardware simulators support compute/commit signal paradigm.

```
reg A, B;
always @(posedge ck) begin
       A <= B;
       B <= A;
       end
```



```
reg zz;
reg [2:0] q;
always @(posedge ck) begin
  zz <= ~zz;
  if (zz) q <= q + 3'd1;
  // Prev line sees old zz
  end
```

- All of the right-hand side expressions are evaluated
- All the results are stored into the left-hand sides.
- Repeat

Commit may create further events for current simulation time.

Delta cycle: a compute/commit cycle without advancing global time.

A VHDL 'signal' has a current and a next value. A SystemC 'sc_signal' likewise has a current and a next value. In Verilog, it's a matter of the assignment operator rather than the net declaration.

LG1.4c - Compute/Commit Cycle

Compute/commit extension: either we need

- a next value field in signal nets (arrays may have multiple), or
- a separate list of pending values to be committed.

Modified EDS kernel:

```
Repeat:
 1. S = all events that have identical time taken from the event list head.
 2. Dispatch all of S (this is the compute phase).
 3. Commit pending values to current values.
```

When we repeat the above loop, if there are any zero-delay models, the value of tnow may not advance: hence a delta cycle.

LG1.6 - RTL Synthesis: Internal forms

Ignore all timing information (hash delays.)

For synthesis to gates, generate a list of assignments for each clock domain, that can be done in parallel.

A final list represents combinational logic, not associated with a clock domain.

Code generation phase: match operations needed against library of gates.

(Similar to a software compiler: match operations needed against instruction set.)

Need to keep additional information for asynchronous reset and presets.

Transparent latches also need additional handling.

```
   module TLATCH(q, g, d);
      input d, q;
     output q;
     assign q = (g) ? d: q;
  endmodule
```

LG1.7 - Basic Synthesis Algorithm

Input

```
module TC(clk, cen);
  input clk, cen;
  reg [1:0] count;
  always @(posedge clk) if (cen) count<=count+1;
endmodule// User=djg11
```

Output

```
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
```

Phase ONE - Generate guard/value lists for each lhs/clock domain.

Phase TWO - Convert to binary gate (bit lane) form with one expression for each bit lane.

LG1.7 Phase ONE - Generate guard/value lists for each clock domain.

Generate a list of variables assigned under the clock.

For each variable, collate the assignments into (guard, value) pairs.

For array assigns, we need a (guard, subscript, value) tuple for each update.

Generate an assignment for each variable where the rhs is a query tree from each list.

If blocking assigns are present, then an environment must be passed in for re-writing the rhs expressions as though the assigns had taken place.

Non-determinism arises if array subscripts cannot be compared at compile time (pointer alias problem).

A final list is needed for the unclocked, combinational logic.

LG1.7 Phase TWO - Convert to binary (bit lane) form

Sign extend an arg to width n:

```
fun sex n nil = if n<=0 then nil else raise "cannot do sex on an empty list"
|   sex n [item] = if n<=1 then [item] else item :: sex (n-1) [item]
|   sex n (h::t) = h :: (sex (n-1) t);
```

Example: integer constant:

```
| pandex w (Num n) = if n = 0 then [ xi_false ] else
    let fun k 0 = nil  (* lsb first *)
|   k n = (if (n mod 2)=1 then xi_true else xi_false) :: k (n div 2)
        fun q 0 = [xi_true]  (* final negative sign bit *)
|   q n = (if (n mod 2)=0 then xi_true else xi_false) :: q (n div 2)
in if (n >= 0) then k n else sex w (q (0-1-n)) end
```

Example: -4 in a 6 bit field is 111100.

Example: conditional expression: a broadside multiplexor:

```
| pandex w (Query(g, t, f)) =
    let val t’ = pandex w t
        val f’ = pandex w f
fun k([], []) = []
|   k(a, nil) = k(a, [ false ])
|   k(nil, b) = k([ false ], b)
|   k(a::at, b::bt) = gen_mux2(g, a, b) :: k(at, bt)
in k(t’, f’) end
```

LG1.7 Further Synthesis Issues

This basic algorithm does not consider any guiding metrics:

- Power consumption
- Area use
- Performance
- Testability

Gate libraries have high and low drive power forms of most gates.

Use Quine/McCluskey Espresso Algorithm for logic minimisation.

Can re-compute expressions locally.

Retiming for structural hazard and timing closure avoidance is now becoming more common (BlueSpec)

LG1.8 - Adder Build (Synthesis)

Adding a pair of bit lists, lsb first.

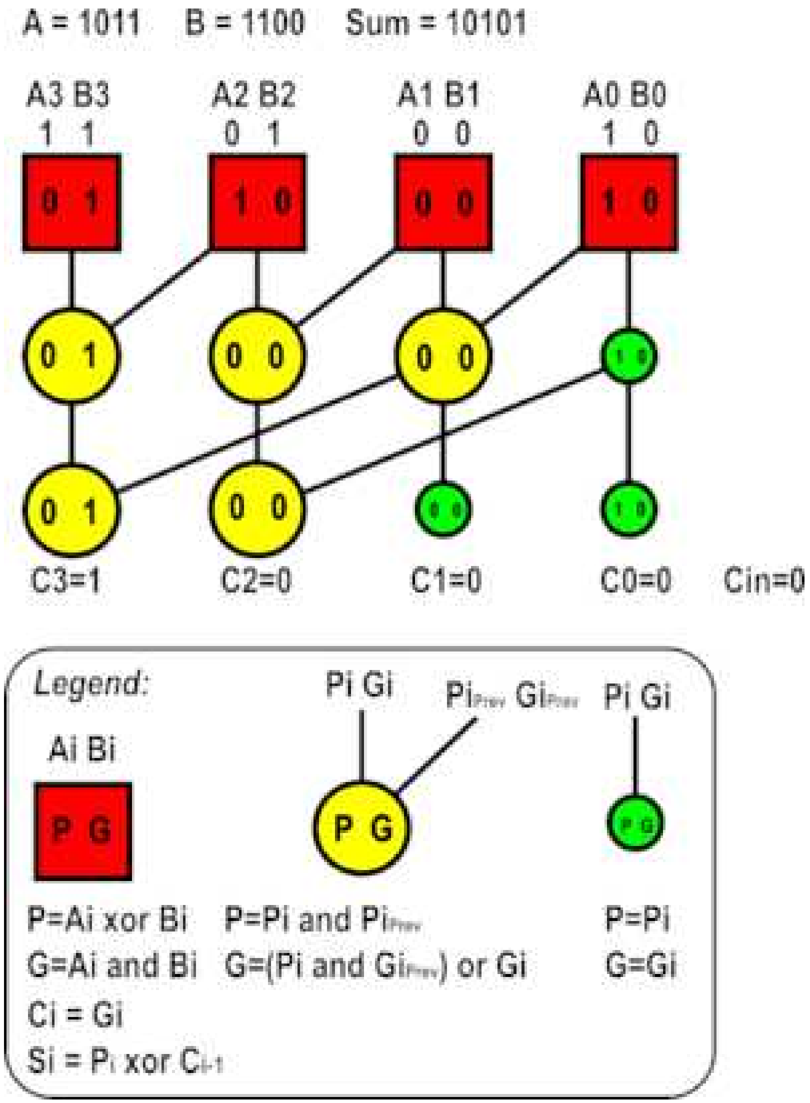
Ripple carry adder:

```
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
```

Faster adder: use wide gates: use functions like `gen_addl`

Carry argument is replaced with a list of generate and propagate pairs from the earlier stages.

LG1.8 continued - Kogge Stone adder



Kogge-Stone is very fast and the area is not too bad, but the wiring is not regular.

Synthesises well. Hard to understand!

For FPGA: Just use RTL '+' and FPGA tools instantiate special paths.

Ex (long): Write a Kogge-Stone generator.

LG1.8c continued - Subtractor, Equality, Inequality, Shifts

`- < > ?= <= != == << >>`

Subtractor: instead pass in a one as the leading borrow-bar and complement each bit from the second operand.

A subtractor will generate a borrow output. If a<b then a-b will need a borrow, hence the raw subtractor implements less-than.

Greater than or equal is just the complement of less than.

Other two inequalities: just swap the operands.

Equality test: check subtractor output is zero: requires an additional NOR gate.

Constant shifts: trivial in bit lane form.

Dynamic shifts: ML code to synthesise barrel shifter is easy.

LG1.9 - RAM Memories

RTL supports arrays: arrays can be synthesised to RAM memories or register files.

```
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;
```

Today: RAM inference from array only done by FPGA tools and high-level synthesis tools. Everyone else defines busses and makes structural instances.

Example dual-ported (one read, one write), SRAM behavioural model:

```
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
```

The behavioural model will be replaced with a RAM macrocell for silicon implementation.

LG1.9c Memories continued

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)

Again, these are instantiated by hand.

LG1.10 - Structural Hazards

Structural Hazard: an interruption to a computation or flow of data owing to a physical resource that has insufficient capacity.

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

Example: insufficient number ALUs.

Example: insufficient number of ports on a RAM/register file.

Holding registers typically needed.

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

Start input and a busy/ready output.

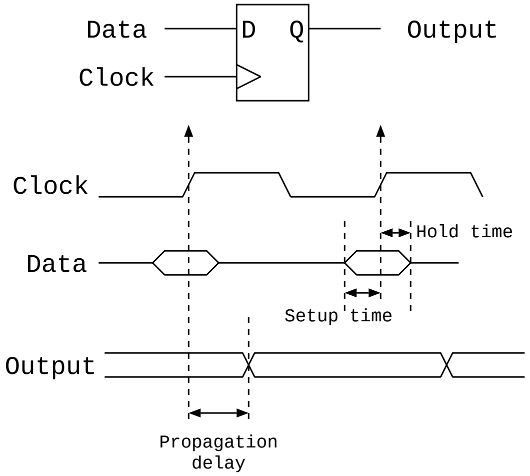
Busy for a constant or variable number of cycles.

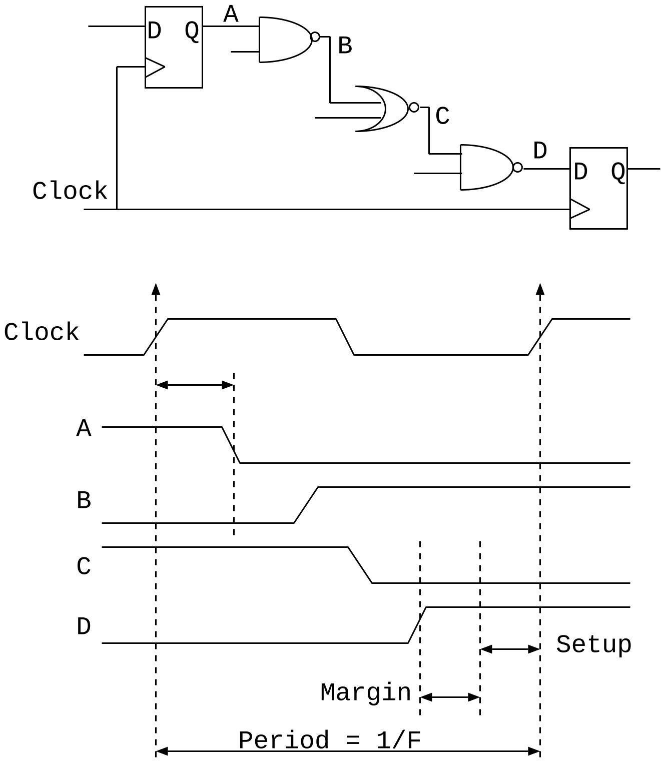
Example: fixed point multipliers and dividers.

Example: all floating point operators.

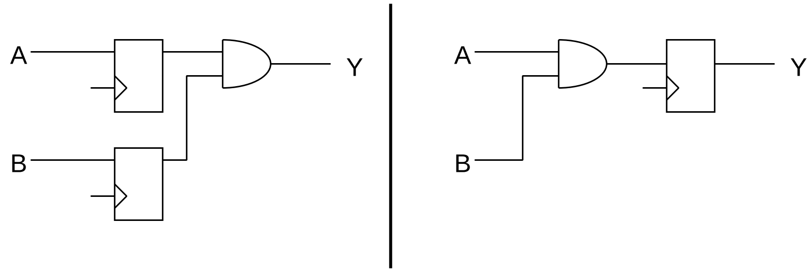
LG1.11 - Retiming & Recoding

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





LG1.11 - Retiming continued



Flip-flop migration:

```
  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` which might not be available.

Problems with internal loops.

Problems with external handshakes that are non-transactional.

Retiming can overcome structural hazard (e.g. write back cycle in RISC pipeline).

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

LG1.12 - 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.

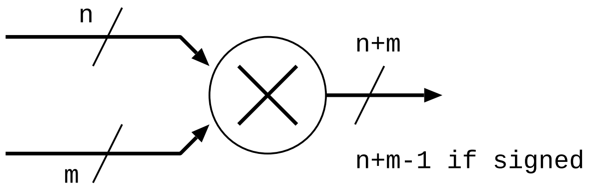
Software on the other hand uses far fewer threads: just where needed. The threads may pass from one module to another and thread blocking is used for flow control of the data.

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

In the future, we expect/hope to see more convergence between these styles: Retimed Parallel Expression or Parallelism Inference.

LG1.13 - Long Multiplication

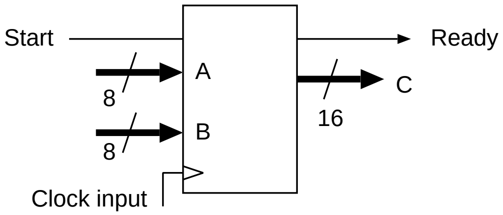
Flash multiplier - combinatorial implementation (e.g. a Wallace Tree).

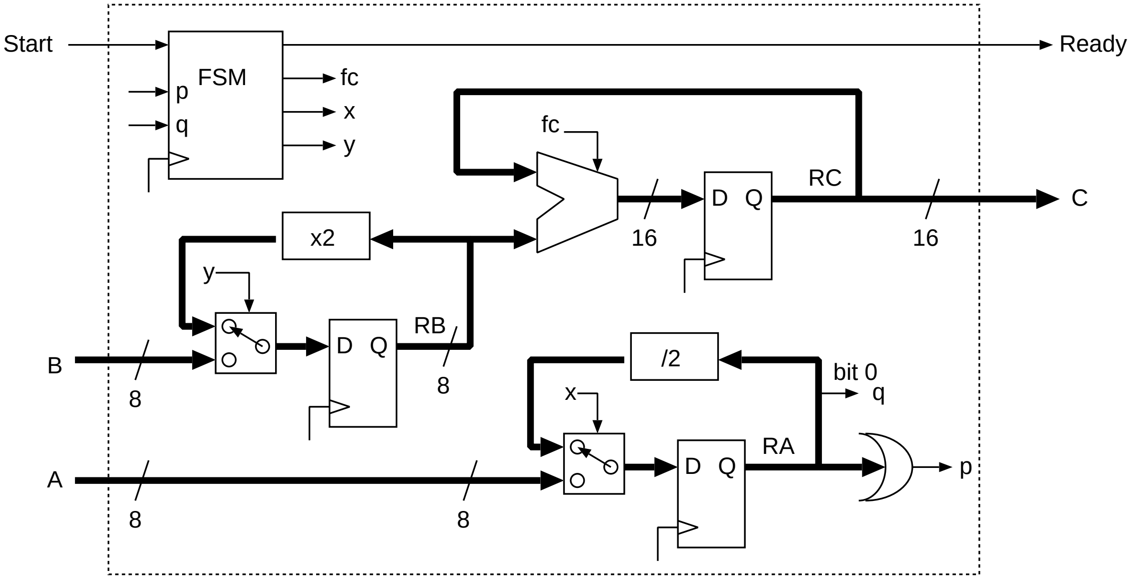


Sequential Long Multiplication

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

Micro-Architecture for a Long Multiplier





Implements conventional long multiplication.

Certainly not fully-pipelined.

Exercise: Write out complete design, including sequencer in RTL or SystemC.

Booth's Multiplier

Booth does two bits per clock cycle:

```
(* Call this function with c=0 and carry=0 to multiply x by y. *)

fun booth(x, y, c, carry) =
    if(x=0 andalso carry=0) then c else
let val x' = x div 4
    val y' = y * 4
    val n  = (x mod 4) + carry
    val (carry', c') = case (n) of
      (0) => (0, c)
     |(1) => (0, c+y)
     |(2) => (0, c+2*y)
     |(3) => (1, c-y)
     |(4) => (1, c)
    in booth(x', y', c', carry')
    end
```

Exercise: Design a micro-architecture consisting of an ALU and register file to implement Booth. Design the sequencer too.