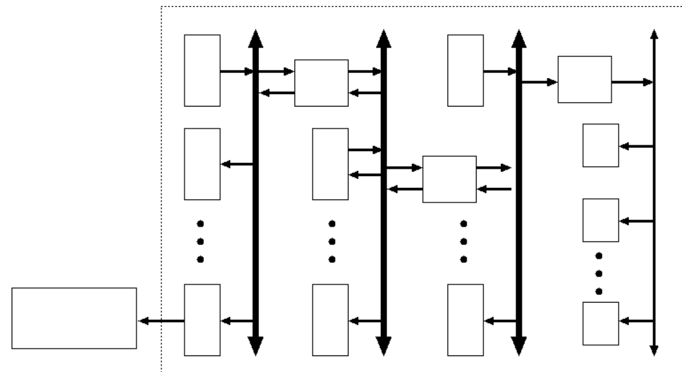
System on Chip
Design and Modelling



University of Cambridge
Computer Laboratory

Lecture Notes
KG5-6 Handout

Dr. David J Greaves

(C) 2008-18 All Rights Reserved DJG.

Part II
Computer Science Tripos
Michaelmas Term 2017/18

- **(1) Energy use in Digital Hardware.**
- **(2) Masked versus Reconfigurable Technology & Computing**
- **(3) Custom Accelerator Structures**
- **(4) RTL, Interfaces, Pipelining and Hazards**
- **(5) High-level Design Capture and Synthesis**
- **(6) Architectural Exploration using ESL Modelling**

KG 5 — High-level Design Capture and Synthesis

In this section of the course we look at high-level synthesis and possibly some other high-level design entry methods. *The HLS material in these printed notes is examinable. The remainder, including Chisel, Bluespec, Statecharts and glue logic syntheis, isin the portfolio lecture notes document and will be covered only if time permits.*

5.0.1 Start Here

Start Slide

The first half of these slides covers 'classical HLS'. The second half discusses some alternative schemes, to be covered if time permits.

High-Level Synthesis (HLS)

Generally speaking, High-Level Synthesis (HLS) compiles software into hardware.

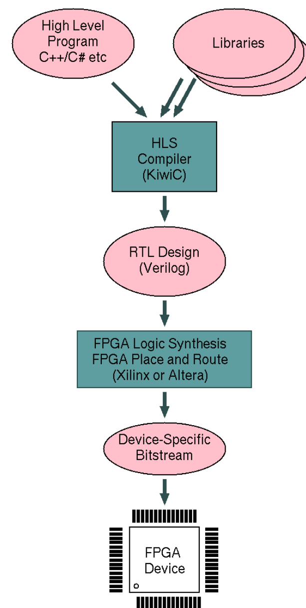


Figure 5.1: Basic Steps of an HLS Flow.

Although a research topic for decades, HLS is now seeing industrial traction. An HLS system revolves around an **HLS compiler** for a high-level language (typically C++). This

- Binds HLL arrays to RAMs and base addresses to items stored in a common RAM.
- Decides what mix of structural components (FUs) such as RAMs and ALUs to instantiate.
- Allocates work to clock cycles (aka schedulling).
- Generates an RTL output for the logic synthesis.
- May provide pre-built libraries for common I/O and mathematics.

The output from an High-Level Synthesis (HLS) compiler is generally RTL which is then fed to an RTL compiler, aka **Logic Synthesiser**, that performs logic synthesis. As we have seen, the logic synthesizer

- instantiates multiplexors that transfer data according to predicates.
- performs logic minimisation or area/power optimisation.
- expands operations on broadside (aka vector) registers into simple boolean operations (aka bit blasting).
- replaces simple boolean operators with gates from an ASIC cell library or look-up tables in an FPGA.

Traditional RTL design entry (Verilog/VHDL) needs:

- Human comprehension of the state encoding,
- Human comprehension of the cycle-by-cycle concurrency, and
- Human accuracy to every low-level detail, such as which registers are **live**.

Performing a Time-for-Space re-folding (i.e. doing the same job with more/less silicon over less/more time) requires a **complete** redesign when entered manually in RTL!

Optimising schedules in terms of memory port and ALU uses ? RTL requires us use Pen and paper? Can we do better than manual RTL coding ? Yes, we use **High-Level Synthesis**.

Dark silicon facilitates 'Conservation Cores'. A paper at ASPOLOS'10 about putting common kernels in silicon and 'Reducing the Energy of Mature Computations' by power gating. PDF

If one considers an embedded processor connected to a ROM, it may be viewed as one large FSM. Since for any given piece of software, the ROM is unlikely to be full and there are likely to be resources in the processor that are not used by that software: the application of a good quality logic minimiser to the system, while it is in the design database, could trim it greatly. In most real designs, this will not be helpful: for instance, the advantages of full-custom applied to the processor core will be lost. In fact, the minimisation function may be too complex for most algorithms to tackle on today's computers.

On the other hand, algorithms to create a good static scheduling of a fixed number of hardware resources work quite well. A processing algorithm typically consists of multiple processing stages (e.g. called pre-emphasis, equalisation, coefficient adaptation, FFT, deconvolution, reconstruction and so on). Each of these steps normally has to be done within tight real-time bounds and so parallelism through multiple instances of ALU and register hardware is needed. The Cathedral DSP compiler was an early tool for helping design such circuits. Such tools can perform time/space folding/unfolding of the algorithm to generate the static shedule that maps operations and variables in a high-level description to actual resources in the hardware. Cache misses, contention for resources and operations that have data-dependent runtime will cause time-domain deviations from a static schedule, so a potentially a dynamic schedule could then make better use of resources **but the overhead of dynamic scheduling can outweigh the cost of the resources saved if the data dependant variations are rare**.

Custom hardware is generally much more energy efficient that general-purpose processors. Reasons include (also see DJG 9-points in §3).

- All the resources deployed are in use with no wasted area,
- Dedicated data paths are not waylaid with unused multiplexors,
- Paths, registers and ALUs can have appropriate widths rather than being rounded up to general word sizes,.
- No fetch/execute overhead,
- Even on an FPGA with its exaggerated dimensions, pass transistor multiplexors use less energy than active multiplexors,
- Operands are fetched in an optimised order, computed once-and-for-all rather than at each step as in today's comples out-of-order CPUs.

5.0.2 Higher-level: Generative, Behavioural or Declarative ?

There are several primary, high-level design expression styles we can consider (in practice use a blend of them ?):

Purely **generative approaches**, like the Lava and Chisel hardware construction languages (HCLs), just 'print out' a circuit diagram or computation graph. Such approaches do not handle data-dependent IF statements: instead muxes must be explicitly printed (although Chisel has some syntactic support to mux generation with **when** like statements). Lava Compiler (Singh)

Generative approaches for super-computer programming, such as DryadLINQ from Microsoft, also elegantly support rendering large static computation trees that can be split over processing nodes. They are agnostic as to what mix of nodes is used: FPGA, GPU or CPU.

But the most interesting systems support complex data-dependent control flow. These are either:

- **Behavioural**: Using imperative software-like code, where threads have stacks and pass between modules, and so on..., or
- **Declarative/Functional/Logical**: Constraining assertions about the allowable behaviour are given, but any ordering constraints are implicit (e.g. SQL queries) rather than being based on a program counter concept. (A declarative program can be defined as an un-ordered list of definitions or assertions that simultaneously hold at all times.)

Historically, the fundamental problem to be addressed was **Programmers like imperative programs but the PC concept with its associated control flow limits available parallelism**. An associated problem is that **even a fairly pure functional program has limited inferable parallelism in practice**

Using a parallel set of guarded atomic actions is pure RTL: which is declarative (since no threads).

All higher-level styles are amenabale to automatic **datapath** and **schedule** generation, including re-encoding and re-pipelining to meet timing closure and power budgets.

Classical HLS converts a behavioural thread to a static schedule (fixed at compile time). But the parallelism inferred is always limited.

Transistors are abundant and having a lot of hardware is not itself a problem (We discussed the *Power Wall* in §1.3.2). Three forms of parallel speed-up are well known for classical imperative parallel programming:

- **Task-Level Parallelism**: partition the input data over nodes and run the same program on each node without inter-node communication (aka *embarassingly parallel*).
- **Programmer-defined, Thread-Level Parallelism**: The programmer uses constructs such as pthreads or CSharp Parallel.for loop to explicitly denote local regions of concurrent activity that typically communicate using shared variables.
- **Instruction-Level Parallelism**: The imperative program (or local region of) is converted to dataflow form, where all ALU operations can potentially be run in parallel, but operands remain pre-requisite to results and load/store operations on a given mutable object must respect program order.

A major (yet sadly less-popular) alternative to thread-level parallelism is programmer-defined channel-based communication, that bans mutable shared variables (examples: Erlang/Occam/HandleC).

5.0.3 Instruction-Level Parallelism

Q. Does a program have a certain level of implicit parallelism? A. With respect to a specific compiler optimisation level and a specific input data set it does indeed. Here is a concrete example:

```
Prihozhy - Code for counting digits 1..5 in integer n.
void main()
{ unsigned long n=21414;
  int m[5], k=0;
  for (int i=0;i<5;i++) m[i]=0;
  while(n) { m[n%10]=1; n/= 10; }
  for (int j=0;j<5;j++) if (m[j]) k++;
}
```

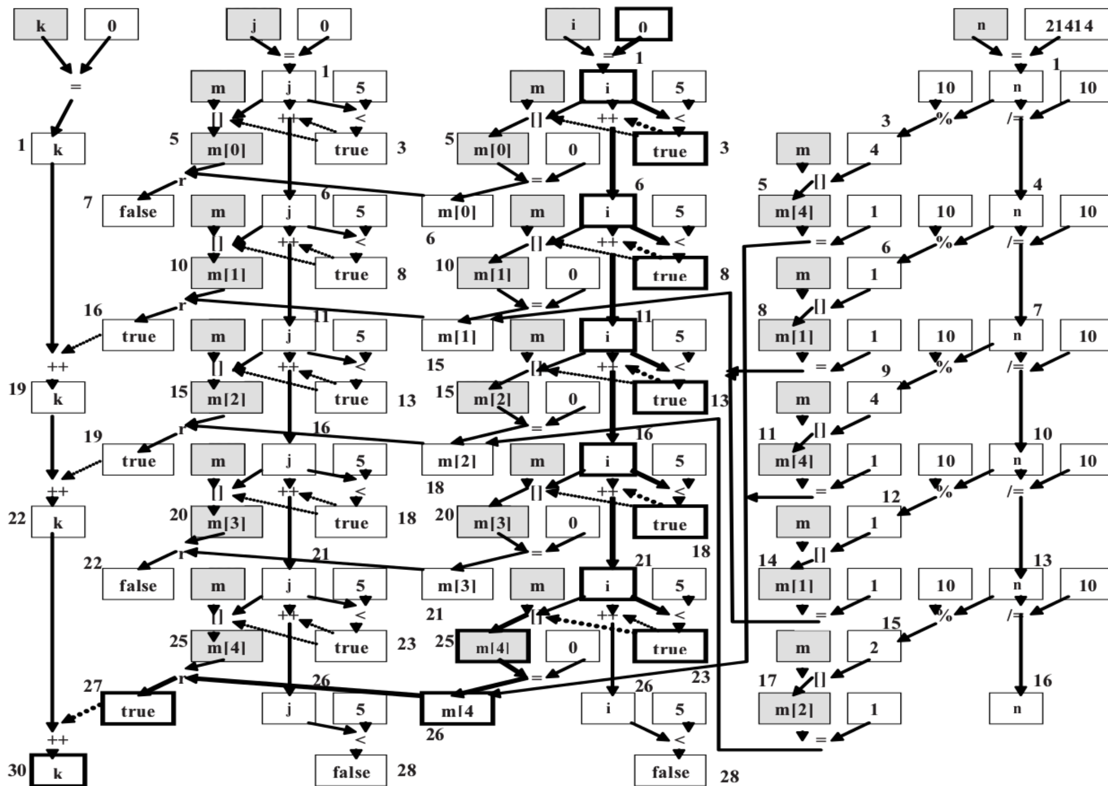


Figure 5.2: Critical Path in Digit Counting C Program (Prihozhy).

In their 2003 paper, Prihozhy, Mattavelli and Mlynek, determine the available parallelism in various C programs. For small programs, such as the digit counting example above, with given input data, the critical path length and the total number of instructions (or clock cycles) can be drawn out. The critical path is highlighted with bold/wider arrows. (Someone volunteer to make them organge?) Their ratio gives the **available instruction-level parallelism**, such as 174/30=5.8. Data Dependencies Critical Path Evaluation(Note also David Wall's 'Limits of instruction-level parallelism'. In Proc. ASPLOS-4, 1991. and J Mak's 'Limits of parallelism using dynamic dependency graphs' 2009.)

Basic algorithm: Like the static timing analyser for hardware circuits (§4.6.2), for each computation node we add its delay to that of the latest-arriving input. But with different input data, the control flow varies and the available parallelism varies. Also the code can be re-factored to increase the parallelism.

The parrellism of the digit counter example can be improved from 30 to 25, as shown in the paper, e.g. by using variants of the **while-to-do transformation**.

```
// When we know g initially holds:
while (g) do { c }  <--> do { c } while (g)
```

A greater, alternative parallelism metric is obtained by neglecting **control hazards**. If we ignore the arrival time of the control input to a multiplexing point we typically get a shorter critical path. **Speculative execution** computes more than one input to a multiplexing point (e.g. a carry-select adder). The same is achieved with perfect branch prediction.

Q. So, does a given program have a fixed certain level of implicit parallelism? If so, we should be able to measure it using static analysis. A. No. In general it greatly depends on that amount of data-dependent control flow, the disambiguation of array subscript expressions (decideable name aliases) and compiler tricks. Q. When is one algorithm the same as another that computes the same result? A. Authors differ, but perhaps the algorithms are the same if there exists a set of transform rules, valid for all programs, that maps them to each other? (That's the DJG definition anyway!)

5.0.4 Beyond Pure RTL: Behavioural descriptions of hardware.

What has 'synthesisable' RTL traditionally provided ?

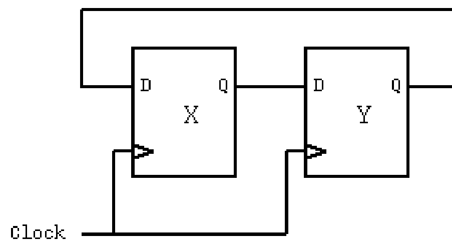


Figure 5.3: A circuit to swap two registers.

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

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

will produce the circuit of Figure 5.3. (If `Xx` and `Y` were busses, the circuit would be repeated for each wire of the bus.) The semantics of the above code are that the right-hand sides are all evaluated and then assigned to the left-hand sides. The order of the statements is unimportant.

However, as mentioned in §4.3.1, the same circuit may be generated using a specification where assignment is made using the `=` operator. If we assume there is no other reference to the intermediate register `t` elsewhere, and so a flip-flop named `t` is not required in the output logic. On the other hand, if `t` is used, then its input will be the same as the flip-flop for `y`, so an optimisation step will use the output of `y` instead of having a flip-flop for `t`.

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

With this style of specification the order of the statements is significant and typically such assignment statements are incorporated in various nested `if-then-else` and `case` commands. This allows hardware designs to be expressed using the conventional imperative programming style that is familiar to software programmers. The intention of this style is to give an easy to write and understand description of the

desired function, but this can result in logic output from the synthesiser which is mostly incomprehensible if inspected by hand.

The word 'behavioural', when applied to a style of RTL or software coding, tends to simply mean that a sequential thread is used to express the sequential execution of the statements.

Despite the apparent power available using this form of expression, there are severe limitations in the officially synthesisable subset of Verilog and VHDL that might also be manifest in basic C-to-gates tool. Limitations are, for instance, each variable must be written by only one thread and that a thread is unable to leave the current file or module to execute subroutines/methods in other parts of the design.

The term '*behavioural model*' is used to denote a short program written to substitute for a complex subsection of a structural hardware design. The program would produce the same useful result, but execute much more quickly because the values of all the internal nets and pipeline stages (that provide no benefit until converted to actual parallel hardware form) were not modelled. Verilog and VHDL enable limited forms of behavioural models to serve as the source code for the subsection, with synthesis used to form the netlist. Therefore limited behavioural models can sometimes become the implementation.

Many RTL synthesisers support an implied program counter (state machine inference).

```
        reg [2:0] yout;
        always
              begin
              @(posedge clk) yout = 1;
              @(posedge clk) yout = 4;
              @(posedge clk) yout = 3;
              end
```

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

None of the event control statements is conditional in the example, but the method of compilation is readily extended to support this: it amounts to the program counter taking conditional branches. For example, the middle event control could be prefixed with 'if (din)'.

```
              if (din) @(posedge clk) yout = 4;
```

Take a non-rentrant function:

- generate a custom **datapath** containing registers, RAMs and ALUs
- and a custom **sequencer** that implements an efficient, **static schedule**

that achieves the same behaviour.

```
int multiply(int A, int B)    // A simple long multiplier with variable latency.
{ RA=A;                       // Not RTL: The while loop trip count is data-dependent.
  RB=B;                       //
  RC=0;                       //
  while(RA>0)                 // Let's make a naive HLS of this program...
  {
    if odd(RA) RC = RC + RB;
    RA = RA >> 1;
    RB = RB << 1;
  }
  return RC;
}
```

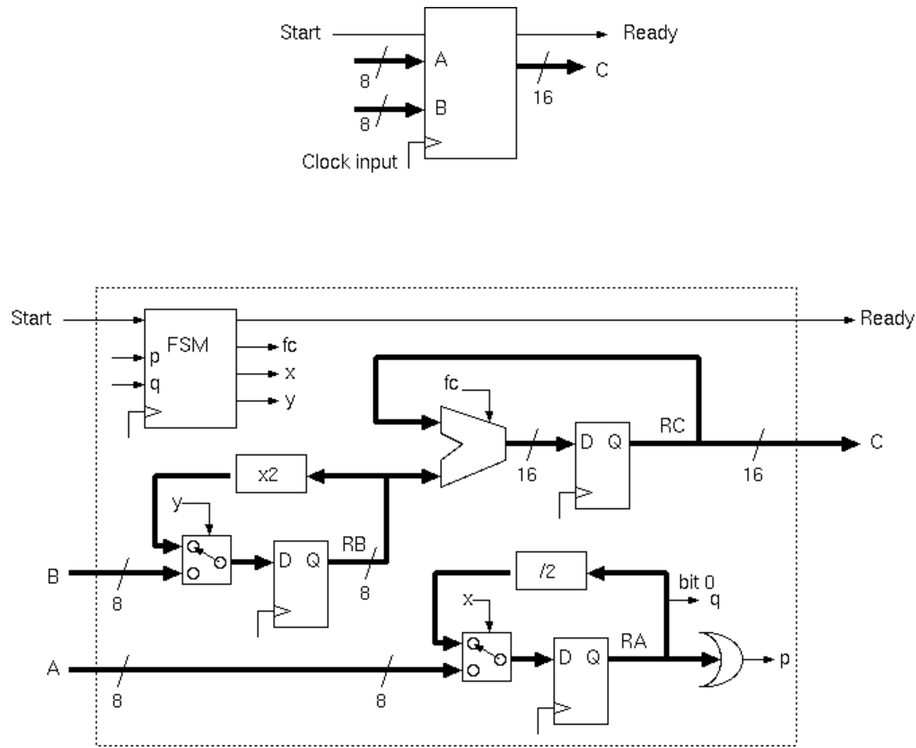


Figure 5.4: Long multiplier viewed as datapath and sequencer.

This simple example has no multi-cycle primitives and a 1-to-1 mapping of ALUs to the source code text, so no schedulling was needed from the HLS tool. Each register has a multiplexor that ranges over all the places it is loaded from. We followed a **syntax-directed** approach with no search in the solution space for minimum clock cycles or minium area or maximum clock frequency. The resulting block could serve as a primitive to be instantiated by an HLS tool. This example is not fully-pipelined and so typically would not be used for that purpose.

5.0.5 Classical HLS Compiler: Operational Phases

The classical HLS tool operates much like a software compiler, but needs more time/space guidance. A single thread from an imperative program is converted to a sequencing FSM and a custom datapath.

1. **Lexing and Parsing** as for any HLL
2. **Type and reference checking**: can an int be added to a string? Is an invoked primitive supported?
3. **Trimming**: Unreachable code is deleted, register widths are reduced where it is manifest that the value stored is bounded, constants are propagated between code blocks and identity reductions are applied to operators, such as multiplying by unity.
4. **Binding**: Every storage and processing element, such as a variable or an add operation or memory read, is allocated a physical resource.
5. **Polyhedral Mapping**: A memory layout or ordering optimisation for nested loops.
6. **Schedulling**: Each physical resource will be used many times over in the time domain. A static schedule is generated. This is typically a scoreboard of what expressions are available when. (Worked example in lecuters.)
7. **Sequencer Generation**: A controlling FSM that embodies the schedule and drives multiplexor and ALU function codes is generated.

8. **Quantity Surveying**: The number of hardware resources and clock cycles used can now be readily computed.
9. **Optimisation**: The binding and schedulling phase may be revisited to better match user-provided target metrics.
10. **RTL output**: The resulting design is printed to a Verilog or VHDL file.

Some operations are intrinsically or better implemented as variable-latency. Examples are division and reading from cached DRAM. This means the static schedule cannot be completely rigid and must be based on expected execution times.

Important binding decisions arise for memories:

- Which user arrays are to have their own RAMs or which to share or which to put in DRAM?
- Should a user array be spread over RAMs for higher bandwidth?
- How is data to be packed into words in the RAMs?
- For ROMs, extra copies can be freely deployed.
- Mirroring data in RAM for more read bandwidth will requires additional work when writing to keep in step.
- How shoud data be organised over DRAM rows? Should data even be stored in DRAM more than once with different row alignments?

5.0.6 Adopting a Suitable Coding Style for HLS

A coding style that works well for a contemporary Von Neumann computer may not be ideal for HLS. For now at least, we cannot simply deploy existing software and expect good results.

Here are four sections of code that all perform the same function. Each is written in CSharp. The zeroth routine uses a loop with a conditional branch on each execution. It has data-dependent control flow.

```
public static uint tally00(uint ind)
{
  uint tally = 0;
  for (int v =0; v<32; v++)
    {
      if (((ind >> v)&1) != 0) tally ++;
    }
  return tally;
}
```

Implementation number one replaces the control flow with arithmetic.

```
public static uint tally01(uint ind)
{
  uint tally = 0;
  for (int v =0; v<32; v++)
    {
      tally += ((ind >> v)&1);
    }
  return tally;
}
```

This version uses a nifty programming trick

```
public static uint tally02(uint ind)
{ // Borrowed from the following, which explains why this works:
  // http://graphics.stanford.edu/~seander/bithacks.html#CountBitsSetParallel
  uint output = ind - ((ind >> 1)&0x55555555);
  output = ((output >> 2)&0x33333333) + (output&0x33333333);
  output = ((output + (output >> 4)&0xF0F0F0F) * 0x1010101);
  return output  >> 24;
}
```

This one uses a ‘reasonable-sized’ lookup table.

```
// A 256-entry lookup table will comfortably fit in any L1 dcache.
// But Kiwi requires a mirror mark up to make it produce 4 of these.
static readonly byte [] tally8 = new byte [] {
  0, 1, 1, 2, 1, 2, 2, 3, 1, 2, 2, 3, 2, 3, 3, 4,
  1, 2, 2, 3, 2, 3, 3, 4, 2, 3, 3, 4, 3, 4, 4, 5,
  ...
  3, 4, 4, 5, 4, 5, 5, 6, 4, 5, 5, 6, 5, 6, 6, 7,
  4, 5, 5, 6, 5, 6, 6, 7, 5, 6, 6, 7, 6, 7, 7, 8,
};

public static uint tally03(uint ind)
{
  uint a0 = (ind >> 0)&255;
  uint a1 = (ind >> 8)&255;
  uint a2 = (ind >> 16)&255;
  uint a3 = (ind >> 24)&255;
  return (uint)(tally8[a0] + tally8[a1] + tally8[a2] + tally8[a3]);
}
```

The look-up table needs to be replicated to give four copies. The logic synthesiser might ultimately replace such a ROM with combinational gates if this will have lower area.

Which of these works best on FPGA? The analysis on the following link will be discussed in lectures: Kiwi Bit Tally (Ones Counting) Comparison

5.0.7 HLS Synthesisable Subset.

Can we convert arbitrary or legacy programs to hardware ? Not very well in general. Can we write new HLL programs that compile to good hardware ? Yes. But we must stick to the supported subset for synthesis.

Typical HLS restrictions:

- **Program must be finite-state and single-threaded**,
- all recursion bounded,
- all dynamic storage allocation outside of infinite loops (or deallocated again in same loop),
- use only boolean logic and integer arithmetic,
- limited string handling,
- very-limited standard library support,
- be explicit over which loops have run-time bounds.

An early example DJG C-To-V compiler from 1995. Bubble Sorter Example

Today commercial HLS tools are widely available : SystemCrafter, Catapult, SimVision, CoCentric, C-to-Verilog.com, Vivado HLS, ... others.

And research HLS tools, such as Kiwi and LegUp, support floating point, pointers and some dynamic storage allocation using DRAM banks as necessary,

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

The software programming paradigm, where a serial thread of execution runs around between various modules is undoubtedly easier to design with than the forced parallelism of expressions found in RTL-style coding. Ideally, a new thread should only be introduced when there is a need for concurrent behaviour in the expression of the design.

A product from COMPILOGIC is typical claimed the following:

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

but like many domain names allocated to companies i nthis area in the last 15 years, this one too has expired.

However, we cannot compile general C/C++ programs to hardware: they tend to use too many language features. Java and C# are better, owing to stronger typing and banning of arithmetic on object handles (all subscription operations apply to first-class arrays).

5.0.8 Unrolling: Trading time for space.

A given function can generally be done in half as many clock cycles using twice as much silicon, although name aliases and control hazards (dependence on run-time input data) can limit this. As well as the C/C++ input code we require additional directives over speed, area and perhaps power. The area directives may specify the number of RAMs or how to map arrays into shared DRAM. Trading (or folding) such time for space is basically a matter of unwinding loops or introducing new loops.

Hazards can limit the amount of unrolling possible, including limited numbers of ports on RAMs and user-set budgets on the number of certain components (FUs) instantiated, such as adders or multipliers.

Unrolling can be:

- **automatic**, where the tool aims to meet a given throughput and clock frequency,

5.0.9 Pipelined Schedulling - One Basic Block

After loop unrolling, we have an expanded control-flow graph that has larger basic blocks than in the original HLL program. In classical HLS, each basic block of the expanded graph is given a time-domain static scheudle.

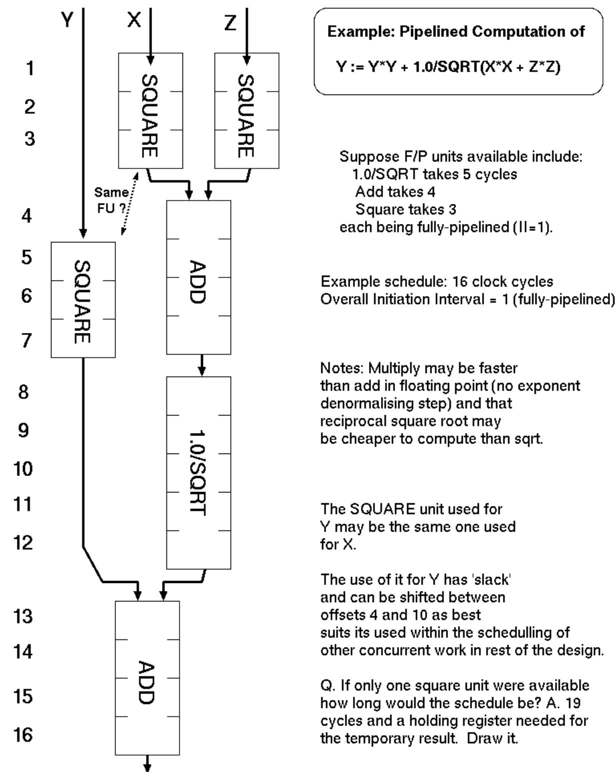


Figure 5.5: Example static schedule for a basic block containing a single assignment.

Our figure shows only one assignment. A basic block schedule typically contains multiple assignments, with sub-expressions and functional units being reused in the time domain throughout the schedule and shared between assignments.

To avoid RaW hazards within one basic block, all reads to a variable or memory location must be schedulled before all writes to the same.

The name alias problem means we must be conservative in this analysis when considering whether arrays subscripts are equal. This is 'undecidable' in general theory, but often doable in practice. Indeed may subscript expressions will be simple functions of loop 'induction variables' whose pattern we need to understand for high performance.

5.0.10 Pipelined Schedulling - Between Basic Blocks

Multiple basic blocks, even from one thread, will be executing at once, owing to pipelining. Frequently, an inner loop consists of one basic block repeated, and so it is competeting with itself for structural resources and data hazards.

Each time offset in a block's schedule needs to be checked for structural hazards against the resource use of all other blocks that are potentially running at the same time.

Every block has its own static schedule of determined length. The early part of the schedule generally contains control-flow predicate computation to determine which block will run next. This can be null if the block is the initialisation code for a subsequent loop (i.e. the basic block ends on a branch destination rather than on a conditional branch). The later part of the block contains data reads, data writes and ALU computations. Data operations can also occur in the control phase but when resources are tight (typically memory read bandwidth) the control work should be given higher schedulling priority and hence remain at the top of the schedule.

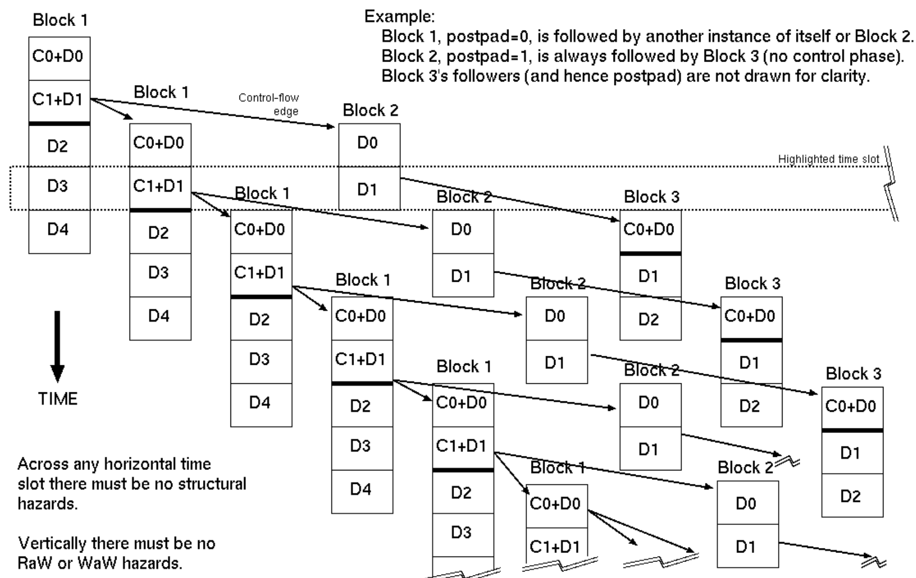


Figure 5.6: Fragment of an example inter-block initiation and hazard graph.

Per thread there will be at most one control section running at any one time. But there can be a number of data sections from successors and from predecessors still running.

The interblock schedule problem has exponential growth properties with base equal to the average control-flow fan out, but only a finite part needs to be considered governed by the maximal block length. As well as avoiding structural hazards, the schedule must contain no RaW or WaW hazards. So a block must read a datum at a point in its schedule after any earlier block that might be running has written it. Or if at the same time, forwarding logic must be synthesised.

It may be necessary to add a 'postpad' to relax the schedule. This is a delay beyond what is needed by the control flow predicate computation before following the control flow arc. This introduces extra space in the global schedule allowing more time and hence generally requiring fewer FUs.

In the highlighted time slot in figure 5.6, the D3 operations of the first block are concurrent with the control and data C1+D1 operations of a later copy of itself when it has looped back or with the D1 phase of Block 2 if exited from its tight loop.

Observing **sequential consistency** imposes a further constraint on schedulling order: for certain blocks, the order of of operations must be (partially) respected. For instance, in a shared memory, where a packet is being stored and then signalled ready with a write to a flag or pointer in the same RAM, the signalling operation must be kept last. (This is not a WaW hazard since the writes are to different addresses in the RAM.) Observing these limits typically results in an expansion of the overall schedule.

5.1 HLS Functional Units (FUs)

The output from HLS is RTL. The RTL will use a mixture of operators supported by the back end logic synthesiser, such as integer addition, and structural components selected from an HLS functional unit (FU) block library, such as floating-point multiply.

5.1.1 Functional Unit (FU) Block Properties

Apart from the specification of the function itself, such as multiply, a block that performs a function in some number of clock cycles can be characterised using the following metrics:

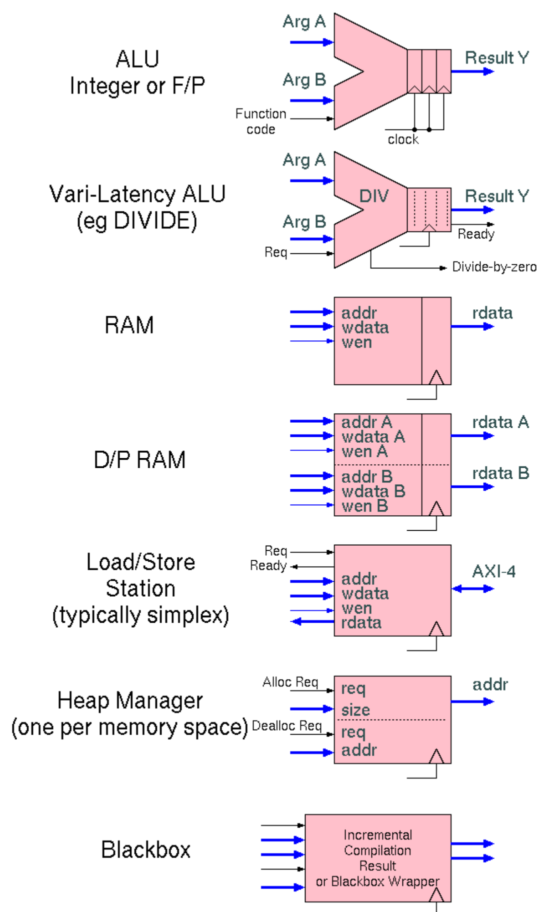


Figure 5.7: Typical examples of FUs deployed by an HLS compiler.

- int **Precision**
- bool **Referentially-Transparent (Stateless)**: always same result for same arguments.
- bool **EIS (An end in itself)**: Has unseen side effects such as turning on an LED
- bool **FL or VL**: Fixed or Variable latency
- int **Block latency**: cycles to wait from arguments in to result out (or average if VL)
- int **Initiation Interval**: minimum number of cycles between starts (arguments in time) (or average if VL)
- real **Energy**: Joules per operation - normally a few nanojoules for a ...
- real **Gate count or area**: Area is typically given in square microns or, for FPGA, number of LUTs.

A unit whose initiation interval is one is said to be '**fully pipelined**'.

In today's ASIC and FPGA technology, combinational add and subtract of up to 32-bit words is typical. But RAM read, multiply and divide are usually allocated at least one pipeline cycle, with larger multiplies and all divides being two or more. For 64-bit word widths, floating point or RAMs larger than L1 size (e.g. 32 KByte), two or more cycle latency is common, but with an initiation interval of one (ii=1).

5.1.2 Functional Unit (FU) Chaining

Naively instantiating standard FUs can be wasteful of performance, precision and silicon area. Generally, if the output of one FU is to be fed directly to another then some optimisation can be made and many sensible optimisations involve changes of state encoding or algorithm that are beyond the back-end logic synthesiser.

A common example is an associatve reduction operator such as floating-point addition in a scalar product. In that example, we do not wish to denormalise and round-and-renormalise the operand and result at each addition. This

- adds processing latency in clock cycles or gate delay on critical path,
- requires modulo schedulling (Lam) for loops shorter than the reduction operator's latency,
- uses considerable silicon area.

For example, in 'When FPGAs are better at floating-point than microprocessors' (Dinechin et al 2007), it is shown that a fixed-point adder of width greater than the normal mantissa precision can reduce/eliminate underflow errors and operate with less energy and fewer clock cycles.

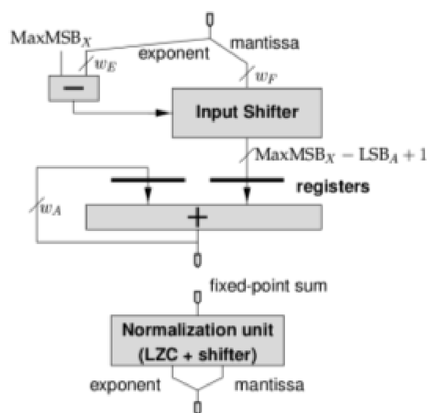


Figure 5.8: Fixed-Point Accumulator Proposal.

Their approach is to denormalise the mantissa on input from each iteration and renormalise once at the end when the result is needed. Even a 'running-average' example is generally used in a decimated form (i.e. only every 10th or so result is looked at).

5.2 Discovering Parallelism: Classical HLS Paradigms

The well-konwn map-reduce paradigm allows the map part to be done in parallel. An associative reduction operator gives the same answer under diverse bracketings. Common associative operators are addition, multiplication, maximum and bitwise-OR. Our first example uses xor. Here we look at some examples where the bodies of an iteration may or may not be run in parallel.

```
public static int associative_reduction_example(int starting)
{
  int vr = 0;
  for (int i=0;i&lt;15;i++) // or also i+=4
    {
      int vx = (i+starting)*(i+3)*(i+5);
      vr ^= ((vx&128)>0 ? 1:0);
    }
  return vr;
}
```

Where the loop variable evolves linearly, variables and expressions in the body that depend linearly on the loop variable are termed linear induction variables/expressions and can be ignored for loop classification since their values will always be independently available in each loop body with minimal overhead.

A **loop-carried dependency** means that parallelisation of loop bodies is not possible. Often the loop body can be split into a part that is and is-not dependent on the previous iteration, with the is-not parts run in parallel at least.

```
public static int loop_carried_example(int seed)
{
  int vp = seed;
  for (int i=0;i&lt;5;i++)
    {
      vp = (vp + 11) * 31241/121; // (fixed)
    }
  return vp;
}
```

A value read from an array in one iteration can be **loop forwarded** from one iteration to another using a holding register to save having to read it again.

```
static int [] foos = new int [10];
static int ipos = 0;
public static int loop_forwarding_example(int newdata)
{
  foos[ipos ++] = newdata;
  ipos %= foos.Length;
  int sum = 0;
  for (int i=0;i&lt;foos.Length-1;i++)
    {
      sum += foos[i]^foos[i+1];
    }
  return sum;
}
```

Data-dependent exit conditions also limit parallelisation, although a degree of speculation can be harmless. How early in a loop body the exit condition is determined is an important consideration. When speculating we continue looping but provide a mechanism to discard unwanted side effects.

```
public static int data_dependent_controlflow_example(int seed)
{
  int vr = 0;
  int i;
  for (i=0;i&lt;20;i++)
    {
      vr += i*i*seed;
      if (vr > 1111) break;
    }
  return i;
}
```

The above examples have been demonstrated using Kiwi HLS on the following link Kiwi Common HLS Paradigms Demonstrated. Wikipedia:Loop Dependence

5.2.1 Memory Banking and Widening

Whether computing on standard CPUs or FPGA, memory bandwidth is often the main performance bottleneck. Given that data transfer rate per-bit of read or write port is fixed, two solutions to memory bandwidth are to use **multiple banks** or **wide memories**. Multiple banks can be accessed simultaneously whereas memories with a wider word are accessed at just one location at a time (per port) but yield more data for each access.

With multiple static RAM banks, data can be arranged randomly or systematically between them. To achieve 'random' data placement, some set of the address bus bits are normally used to select between the different banks. Indeed, when multiple chips are used to provide a single bank, this arrangement is already deployed. The question is which bits to use.

Using low bits causes a fine-grained interleave, but may either destroy or leverage spatial locatlity in access patterns according to many details.

Ideally, concurrent accesses hit different banks, therefore providing parallelism. Where data access patterns are known in advance, which is typically the case for HLS, then this can be maximised or even ensured by careful bank mapping. Interconnection complexity is also reduced when it is manifest that certain data paths of a full cross-bar with never be used. In extreemis, we need no interconnect switch at all.

5.2.2 Data Layout for Burrows-Wheeler Transform

The BWT of a string is another string of the same length and alphabet. According to the definition of a transform, it is information preserving and has an inverse. We shall not define it in these notes. The following code makes efficient perfect string matches of needles in a given haystack using the BWT. It uses lookup in the large Rank array that is pre-computed from the haystack, as is the BWT itself. It also uses the Tots_before array, but this is very small and easily fits in BRAM on an FPGA. The Rank array is 2-D, being indexed by character, and contains integers ranging up to the haystack size (requiring more bits than a character from the alphabet).

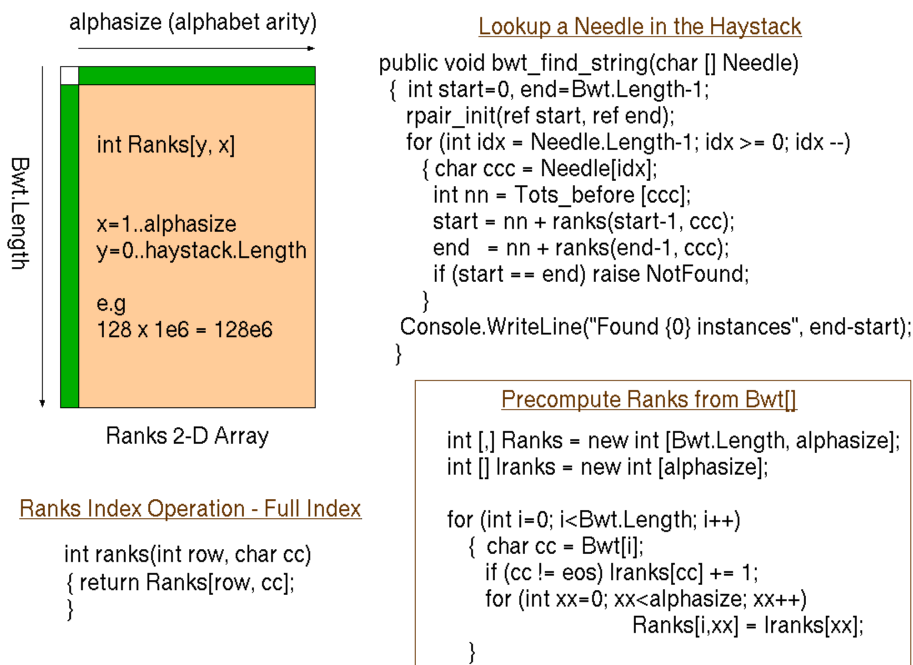


Figure 5.9: Lookup procedure when string searching using BWT.

The problem can be that, for big data, such as Giga-base DNA genomes, the Rank array may be too big for available the DRAM. The solution is to decimate the Rank array by some factor, such as 32, and then only store every 32nd row in memory. When lookup does not fall on a stored row, the row's contents are interpolated on-the-fly. This does require the original BWT-transformed string is stored, but this may well be useful for many related purposes anyway.

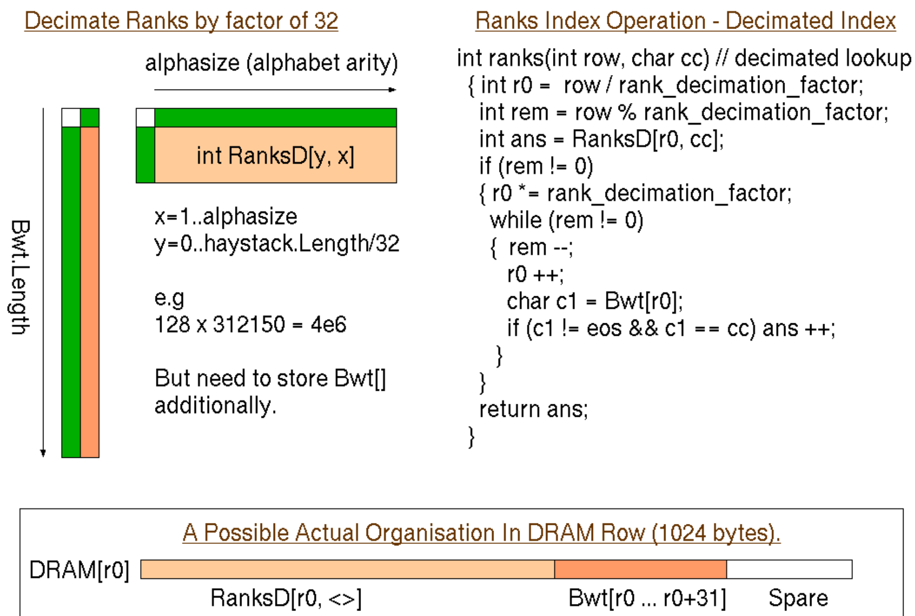


Figure 5.10: Compacted Rank Array for BWT and a sensible layout in a DRAM row.

Access patterns to the Rank array will exhibit no spatial or temporal locality (especially at the start of the search when `start` and `end` are well separated. If only one bank (here we mean channel) of DRAM is available, then random access delays dominate. However, one good idea is to store the BWT fragment in the same DRAM row as the ranking information. Then only one row activation is needed per needle character. For what factor of decimation is the interpolator not on the critical path? This will depend mainly on how much the HLS compiler chooses In general, when aligning data in DRAM rows, sometimes a pair of items that are known to both be needed can be split into different rows. In which case storing everyting twice may help, with one copy offset by half a row length from the other, since then it is possible to manually address the copy with the good alignment.

The only aspect of this BWT material that is examinable is that layout of data in DRAM is important and that mirroring the data with different alignments can be worthwhile. Pico Computing White PaperKiwi Implementation

5.2.3 Smith-Waterman D/P Data Dependencies

The Smith-Waterman algorithm has become an icon for FPGA acceleration. Two strings are matched for edit distance.

A quadratic algorithm based on dynamic programming is used. The maximum score needs to be found in a 2-D array where each score depends on the three immediate neighbours with lower index as shown in figure 5.11. Zeros are inserted where subscripts would be negative at the edges.

Acceleration is achieved by computing many scores in parallel. There is no simple nesting of two `for` loops that can work. Instead, items on a diagonal frontier can be computed in parallel. Normally one string behaves as a needle with perhaps 1000 characters and the other is a haystack streamed from a fileserver.

We shall discuss suitable hardware architectures on the example sheet.

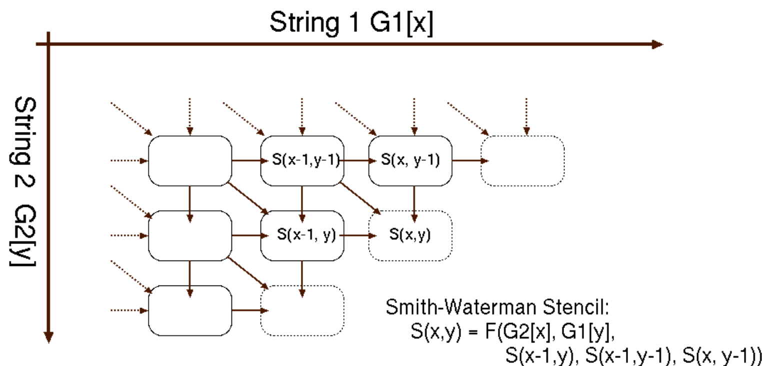


Figure 5.11: Data dependencies in Smith-Waterman Alignment Matcher.

Smith-Waterman is not examinable (within this course at least).

5.2.4 Polyhedral Address Mapping

Polyhedral Address Mapping not examinable for CST Part II.

A number of restricted mathematical systems have useful results in terms of decidibility of certain propositions. A restriction might be that expressions only multiply where one operand is a constant and a property might be that an expression always lies within a certain intersection of n-dimensional planes. There is a vast literature regarding integer linear inequalities (linear programming) that can be combined with significant results from Presburger (Preburger Arithmetic) and Mine (The Octagon Domain) as a basis for optimising memory access patterns within HLS.

We seek transformations that:

1. compact provably sparse access patterns into packed form or

2. where array read subscripts can be partitioned into provably disjoint sets that can be served in parallel by different memories, or

3. where data dependencies are sufficiently determined that thread-future reads can be started at the earliest point after the supporting writes have been made so as to meet all read-after-write data dependencies.

Q. Does the following have loop interference ?

```
for (i=0; i&lt;N; i++)  A[i] := (A[i] + A[N-1-i])/2
```

A. Yes, at first glance, but we can recode it as two independent loops. (‘Loop Splitting for Efficient Pipelining in High-Level Synthesis’ by J Liu, J Wickerson, G Constantinides.)

“In reality, there are only dependencies from the first N/2 iterations into the last N/2, so we can execute this loop as a sequence of two fully parallel loops (from 0...N/2 and from N/2+1...N). The characterization of this dependence, the analysis of parallelism, and the transformation of the code can be done in terms of the instance-wise information provided by any polyhedral framework.”

Q. Does the following have loop body inter-dependencies ?

```
for (i=0; i&lt;N; i++)  A[2*i] = A[i] + 0.5f;
```

A. Yes, but can we transform it somehow?

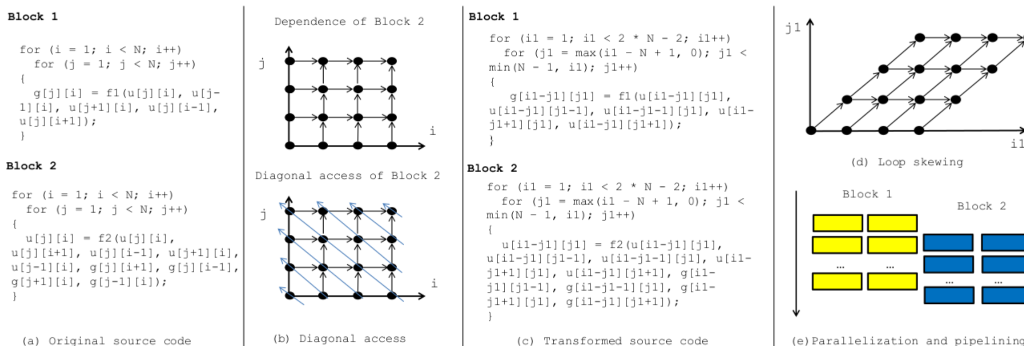


Figure 5.12: Affine transformations (Zuo, Liang et al).

Improving High Level Synthesis Optimization Opportunity Through Polyhedral Transformations

The skewing of block 1 enables it to be parallelised without data dependence. The following block can be run in parallel after an appropriately delayed start ...

Wikipedia:Polyhedral

5.2.5 The Perfect Shuffle Network - FFT Example

A number of algorithms have columns of operators that can be applied in parallel. The FFT is one such example. The operator is commonly called a 'butterfly'. The operator composes two operands (which are generally each complex numbers) and delivers two results. The successive passes can be done in place on one array whose values are passed by reference to the butterfly code in the software version.

Our diagram shows a 16-point FFT, but typically applications use hundreds or thousands. (The code fragment shows a single-threaded implementation that does not attempt to put butterflies in parallel.)

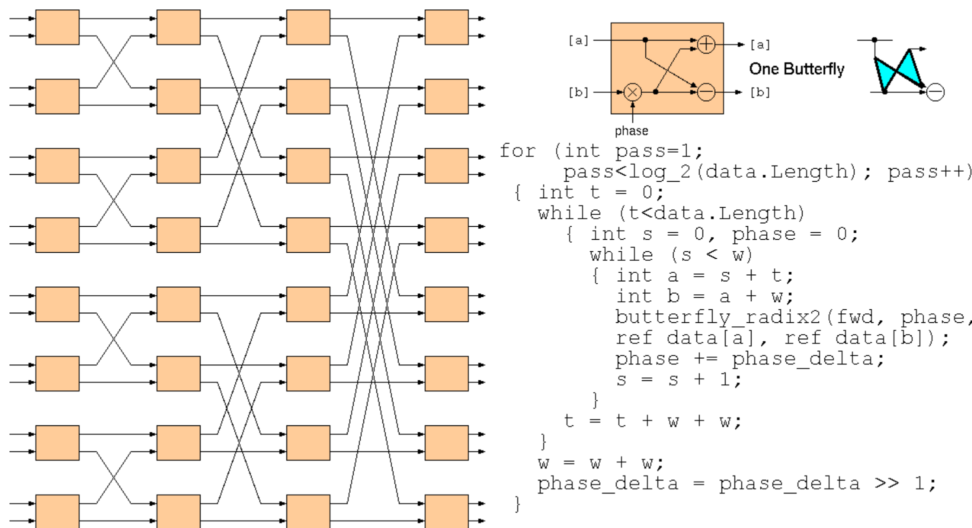


Figure 5.13: Shuffle Data Flow in the Fast Fourier Transform.

The pattern of data movement is known as a shuffle. It is also used in switching and sorting algorithms.

The downside of shuffle computations for acceleration is that there is no packing of data (partition of the data array) into spatially-separate memories that works well for all of the passes: what is good at the start is poor at the end.

FFT details are not examinable (within this course at least)..

5.2.6 Other Expression forms: Channel Communication

Using shared variables to communicate between threads requires that the user abides by self-imposed protocol conventions.

Typical patterns are:

- always ready,
- simplex guard with reader always faster than writer,
- four-phase handshake,
- two-phase handshake.

As mentioned elsewhere in these notes, some protocols cannot be pipelined; some degrade throughput when pipelined and others are designed for it Some approaches completely ban shared variables and enforce use of channels (Handel-C and the best Bluespec coding styles). (LINK: Handlec.pdf)

The Bluespec language infers channel-like behaviour from user syntax that looks like conventional reads and writes of shared variables.

Handel-C uses explicit Occam/CSP-like channels ('**!**' to write, '**?**' to read):

```
// Generator (src)     // Processor          // Consumer (sink)
while (1)              while(1)              while(1)
{                      {                     {
  ch1 ! (x);             ch2 ! (ch1? + 2)      $display(ch2?);
  x += 3;              }                     }
}
```

Using channels makes concurrency explicit and allows synthesis to re-time the design. In both cases, all of the handshaking signals potentially required are generated by the compiler and then trimmed away again if they would have constant values owing to certain components being always ready.

Bluespec RTL was intended to be declarative, both in the elaboration language and with the guarded atomic actions for actual register transfers. Its advanced generative elaborator is a functional language and a joy to use for advanced/functional programmers. So it is/was much nicer to use than pure RTL. It has a scheduler (cf DBMS query planner) and a behavioural-sub language for when imperative is best. Like Chisel, it has good support for **valid-tagged data** in registers and busses. Hence compiler optimisations that ignore dead data are potentially possible.

But the main shortcoming of Bluespec is/was that the nice guarded atomic actions normally operate on imperative objects such as registers and RAMs where WaW/RaW/WaR bites as soon as transaction order is not carefully controlled.

Toy Bluespec compiler by DJG

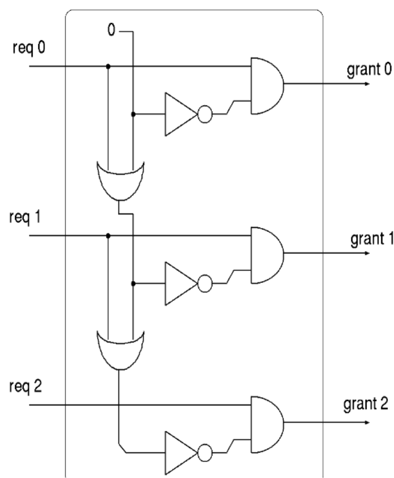
5.2.7 Other Expression forms: Hardware Construction Languages

The **generate** statements in Verilog (Section 4.3.1) and VHDL are clunky imperative affairs. How much nicer it is to print out your circuit using higher-order functional programs! That's the approach of Chisel, a DSL embedded in Scala. Lava was the first HCL of this nature: 'Lava: Hardware Design in Haskell (1998)' by Per Bjesse, Koen Claessen, Mary Sheeran.

A Varadic Priority Arbiter in Chisel

```
class genPriEncoder(n_inputs : Int) extends Module
  {
   val io  = new Bundle { }
   val terms = (0 until n_inputs).map
           (n => ("req" + n, "grant" + n))

   terms.foldLeft (Bool(false))
   { case (sofar, (in, out)) =>
     val (req, grant) = (Bool(INPUT), Bool(OUTPUT))
     io.elements += ((in, req))
     io.elements += ((out, grant))
     grant := req & !sofar
     val next = new Bool
     next := sofar | req
     next
   }
  }
```



H/W components extend Module.
They do their I/O via a Bundle.
All the standard operators & | ! are overloaded for h/w generation.

David J Greaves – Computer Lab Cambridge   Memocode 2015, Austin Texas.

Figure 5.14: An Example Chisel Module.

5.2.8 Other Expression forms: Logic Synthesis from Guarded Atomic Actions (Bluespec)

Using guarded atomic actions is an old and well-loved design paradigm. Recently Bluespec System Verilog has successfully raised the level of abstraction in RTL design using this paradigm.

- Every leaf operation has a guard predicate: says when it CAN be run.
- A Bluespec design is expressed as a list of declarative rules,
- Operations are commanded by rules for atomic execution where the rule takes on the conjunction of its atomic operation guards and the rule may have its own additional guard predicate.
- Shared variables are ideally entirely replaced with one-place FIFO buffers with automatic handshaking,
- All communication to and from registers, FIFOs and user modules is via transactional/blocking 'method calls' for which argument and handshake wires are synthesised according to a global ready/enable protocol,
- Rules are allocated a static schedule at compile time and some that can never fire are reported,
- The current strict mapping to clock cycles (time/space folding) might be relaxed by future compilation strategies.
- The wiring pattern of the whole design is generated from an embedded functional language (rather than embedding the language as a DSL in the way of Chisel or Lava).
- Operations have the expectation they WILL be run (fairness).

The term 'wiring' above is used in the sense of TLM models: binding initiators to target methods.

The intention was that a compiler can direct scheduling decisions to span various power/performance implementations for a given program. But designs with an over-reliance on shared variables suffer RaW/WaR hazards when the schedule is altered.

LINK: Small Examples Toy BSV Compiler (DJG)

First basic example: two rules: one increments, the other exits the simulation. This example looks very much like RTL: provides an easy entry for hardware engineers.

```
module mkTb1 (Empty);

   Reg#(int) x <- mkReg (23);

   rule countup (x < 30);
      int y = x + 1;          // This is short for  int y = x.read() + 1;
      x <= x + 1;             // This is short for  x.write(x.read() + 1);
      $display ("x = %0d, y = %0d", x, y);
   endrule

   rule done (x >= 30);
      $finish (0);
   endrule

endmodule: mkTb1
```

Second example shows an interface declaration that is imported by both parties. The example interface is for a pipeline object that could have aribtrary delay. The sending process is blocked by implied handshaking wires (hence far less typing than Verilog) and in the future would allow the programmer or the compiler to retime the implementation of the pipe component.

```
module mkTb2 (Empty);

   Reg#(int) x    <- mkReg ('h10);
   Pipe_ifc  pipe <- mkPipe;

   rule fill;
      pipe.send (x);
      x <= x + 'h10;   // This is short for  x.write(x.read() + 'h10);
   endrule

   rule drain;
      let y = pipe.receive();
      $display ("    y = %0h", y);
      if (y > 'h80) $finish(0);
   endrule
endmodule
```

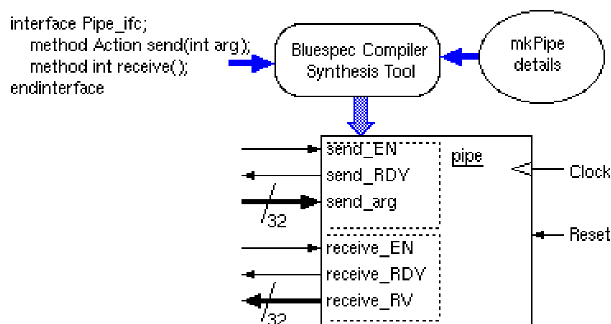


Figure 5.15: Synthesis of the 'pipe' Bluespec component with handshake nets.

But, imperative expression using a conceptual thread is also useful to have, so Bluespec has a behavioural sub-language compiler built in.

5.2.9 Classical Imperative/Behavioural H/L Synthesis Summary

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

- unbounded recursive functions,
- unbounded heap use
- other sources of unbounded numbers of state variables,
- many library functions: access to file or screen I/O.

And it is not currently sensible to compile seldom-used code to the FPGA since conventional CPUs serve well.

A Survey and Evaluation of FPGA High-Level Synthesis Tools, Nane et al, IEEE T-CAD December 2015—

Generating good hardware requires global optimisation of the major resources (ALUs, Multipliers and Memory Ports) and hence automatic time/space folding. An area-saving approach New techniques are needed that note that wiring is a dominant power consumer in today's ASICs

The major EDA companies, Synopsys, Cadance and Mentor all actively marketing HLS flows. Altera (Intel) and Xilinx, the FPGA vendors, are now also promoting HLS tools.

Many people remain highly skeptical, but with FPGA in the cloud as a service in 2017 onwards, a whole new user community is garnered.

Synthesis from formal spec and so on: This is currently academic interest only ? Except for glue logic? Success of formal verification means abundance of formal specs for protocols and interfaces: automatic glue synthesis seems highly-feasible.

5.2.10 Accellera IP-XACT

IP-XACT non-examinable for Part II CST.

IP-XACT is an XML Schema for IP Block Documentation standardised as IEEE 1685. Wikipedia

It was developed by an industrial working party, the SPIRIT Consortium, as a standard for automated configuration and integration of IP blocks. IP-XACT is an IEEE standard for describing IP blocks and for automated configuration and integration of assemblies of IP blocks. It describes interfaces and attributes of a block (e.g. terminal and function names, register layouts and non-functional attributes). It includes separate RTL and ESL/TLM descriptions (future work to integrate these). It aims to provide all the front-end infrastructure for rapid SoC assembly from diverse IP supplies, support for assertions and and perhaps even some glue logic synthesis.

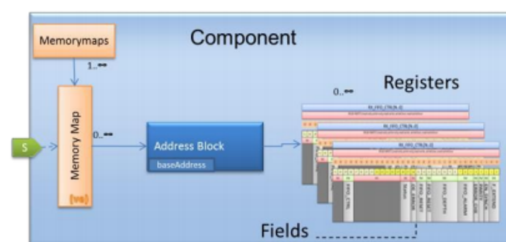


Figure 5.16: IP-XACT captures memory map and register field definitions.

All IP-XACT documents use titular attributes spirit:vendor, spirit:library, spirit:name, spirit:version. A document typically represents one of:

- bus specification, giving its signals and protocol etc;

- leaf IP block data sheet;
- or a heirarchic component wiring diagram that describes a sub-system by connecting up or abstracting other components made up of spirit:componentInstance and spirit:interconnection elements.

For each port of a component there will be a spirit:busInterface element in the document. This may have a spirit:signalMap that gives the mapping of the formal net names in the interface to the names used in a corresponding formal specification of the port. A simple wiring tool will use the signal map to know which net on one interface to connect to which net on another instance of the same formal port on another component.

There may be various versions of a component in the document, each as a spirit:view element, relating to different versions of a design: typical levels are gate-level, RTL and TLM. Each view typically contains a list of filenames as a spirit:fileSet that implement the design at that level of abstraction in appropriate language, like Verilog, C++ or PSL.

Non-functional data present includes the programmer's view with a list of spirit:register declarations inside a spirit:memoryMap or spirit:addressBlock.

Similar tools: Our Part Ib students currently use the Qsys System Integrator tool from Altera. ARM has its Socrates tool and Xilinx has IP Designer in Vivado.

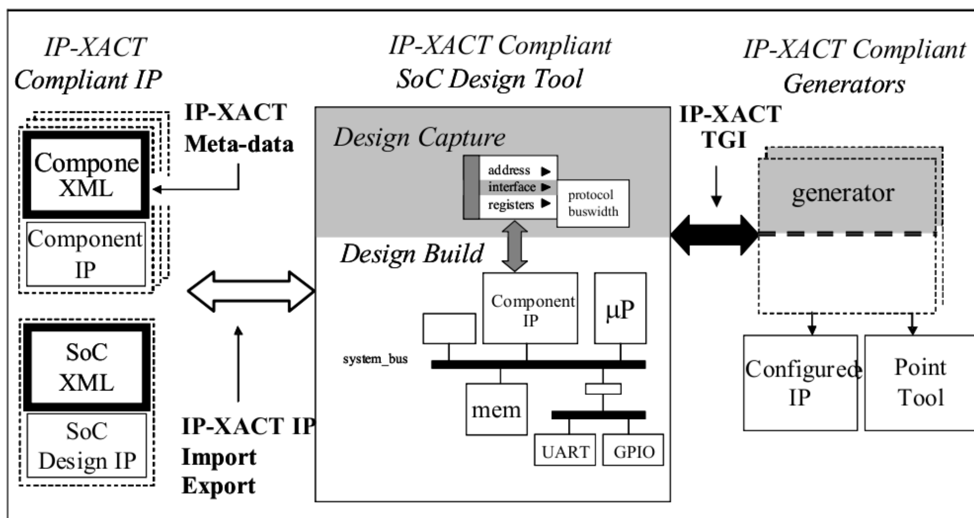


Figure 5.17: Reference Model for design capture and synthesis using IP-XACT blocks.

IP blocks and bus standards are stored in libraries and indexed using datasheets for each block in xml according to the IP-XACT schema. A schematic design capture editor supports creation and editing of a high-level block diagram for the SoC. The SoC design is then output as IP-XACT conformant XML. Various synthesis plugins, termed 'generators' produce the inter-block structural wiring RTL that is otherwise highly tedious and error prone to manually create. They may also instantiate bus bridges and port multiplexors and other glue logic.

Automatic generation of memory maps and device-driver header files is also normally supported. Header files in RTL and C are kept in synch. Testbenches following OVM/UVM coding standards might also be rendered. Other outputs, such as power and frequency estimates or user manual documentation are typically generated too. Greaves+Nam created a glue logic synthesiser Synthesis of glue logic, transactors, multiplexors and serialisors from protocol specifications.

Perhaps explore the free plugin(s) for Eclipse if you are keen.

KG 6 — Architectural Exploration using ESL Modelling

We can model our hardware system at various levels of detail following the taxonomy:

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

An ESL methodology aims:

Aim 1: To model with good performance a complete SoC using full software/firmware.

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

So, an ESL methodology must provide:

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

A commonly used method is SystemC Transactional-Level Modelling (TLM) using high-level C++ models running over the SystemC event-driven kernel. Enhancements beyond that are:

- Synthesise high-level models to form parts of the fabricated system (e.g. using HLS)(but today manual re-coding is mainly used).
- Embed assertions in the high-level models and use these same assertions through to tape out (Section **??** (not lectured this year)).

6.0.11 SystemC: Hardware Modelling Library Overview

SystemC is a free library for C++ for hardware SoC modelling. Download from www.accelera.org SystemC was developed over the last fifteen years with three major releases. Also of significance is the TLM coding style 1.0 and sub-library, TLM 2.0.

It includes (at least):

- A module description system where a module is a C++ class,
- An eventing and threading kernel,
- Compute/commit signals as well as other forms of channel,
- A library of fixed-precision integers,
- Plotting and logging facilities for generating output,
- A transactional modelling (TLM) sub-library.

Greaves developed the TLM_POWER3 add-on library for power modelling.

Originally aimed as an RTL replacement, for low-level hardware modelling. Now being used for high-level (esp. transactional) modelling for **architectural exploration**. Also sometimes used as an implementation language with its own synthesis tools. SystemC Synthesis

Problem: hardware engineers are not C++ experts but they can be faced with confusing C++ error messages.

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

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

- Architectural exploration: Making a fast and quick, high-level model of a SoC to explore performance variation against various dimensions, such as bus width and cache memory size.
- Transactional-level (TLM) models of systems, where handshaking protocols between components using hardware nets are replaced with subroutine calls between higher-level models of those components.
- Synthesis: RTL can be synthesised from from SystemC source code using High-Level Synthesis. SystemC Synthesis

Additional notes:

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

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

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

In this course we shall focus on the loosely-timed, blocking TLM modelling style of ESL model.

6.1 ESL Flow Model: Avoiding ISS/RTL overheads using native calls.

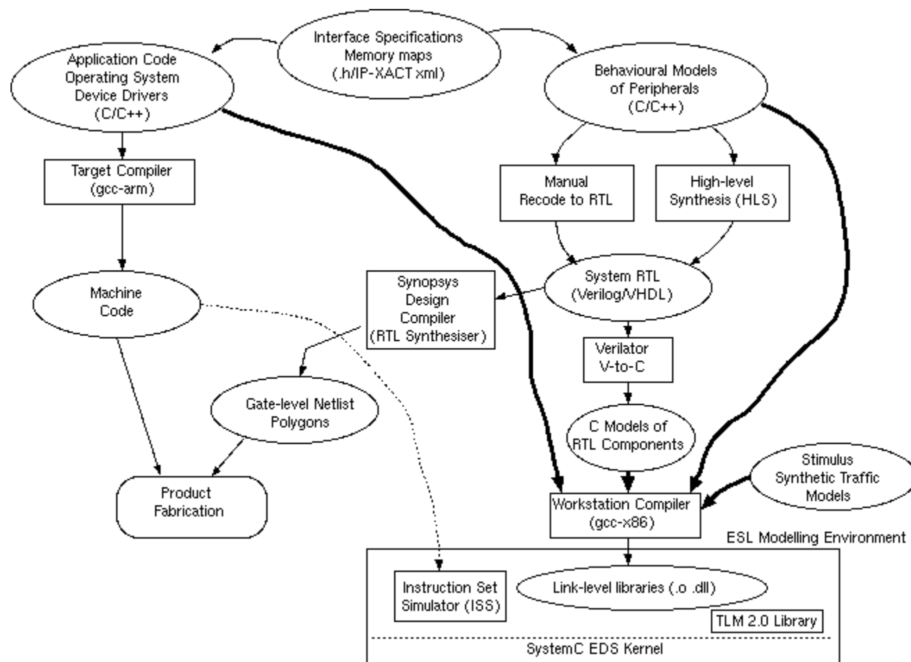


Figure 6.1: ESL Flow: Avoiding the ISS by cross-compiling the firmware and direct linking with behavioural models.

ESL flows are commonly based on C/C++. This language is used for behavioural models of the peripherals and for the embedded applications, operating system and device drivers. For fabrication, the embedded software is compiled with the target compiler (e.g. `gcc-arm`) and RTL is converted to gates and polygons using Synopsys Design Compiler. For ESL simulation, as much as possible, we take the original C/C++ and link it all together, whether it is hardware or software, and run it over the SystemC event-driven simulation (EDS) kernel.

Variations: sometimes we can import RTL components using a tool such as Verilator or VTOC. Sometimes we use an ISS to interpret (or JIT) the target processor machine code.

6.1.1 Using C Preprocessor to Adapt Firmware

We may need to recompile the hardware/software interface when compiling for ESL model as compared to the device driver installed in an OS or ROM firmware. For a 'mid-level model', differences might be minor and so implemented in C preprocessor. Device driver access to a DMA controller might be changed as follows:

6.2. KG 6 ARCHITECTURAL EXPLORATION USING ESL MODELLING

```
#define DMACONT_BASE       (0xFFFFCD00) // Or other memory map value.
#define DMACONT_SRC_REG    0
#define DMACONT_DEST_REG   4
#define DMACONT_LENGTH_REG 8            // These are the offsets of the addressable registers
#define DMACONT_STATUS_REG 12

#ifdef ACTUAL_FIRMWARE

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

#else

  // For high-level TLM modelling:
  // Make a direct subroutine call from the firmware to the DMACONT model.
  #define DMACONT_WRITE(A, D)    dmaunit.slave_write(A, D)
  #define DMACONT_READ(A)        dmaunit.slave_read(A)

#endif

// The device driver will make all hardware accesses to the unit using these macros.
// When compiled native, the calls will directly invoke the behavioural model, bypassing the bus model.
```

Behavioural model example (the one-channel DMA controller from earlier):

```
  // Behavioural model of
  // slave side: operand register r/w.
  uint32 src, dest, length;
  bool busy, int_enable;

  u32_t status() { return (busy << 31)
            | (int_enable << 30); }

  u32_t slave_read(u32_t a)
  {
    return (a==0)? src: (a==4) ? dest:
     (a==8) ? (length) : status();
  }
  void slave_write(u32_t1 a, u32_t d)
  {
     if (a==0) src=d;
     else if (a==4) dest=d;
     else if (a==8) length = d;
     else if (a==12)
     { busy = d >> 31;
       int_enable = d >> 30; }
  }
```

```
  // Bev model of bus mastering portion.
  while(1)
  {
    waituntil(busy);
    while (length-- > 0)
      mem.write(dest++, mem.read(src++));
    busy = 0;
  }
```

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

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

But this will need a thread to run it, so this code must be placed in its own C macro that is inlined at all points where the supporting expressions might change.

6.2 Transactional-Level Modelling (TLM)

Recall our list of three inter-module communication styles, we will now consider the third style:

1. **Pin-level modelling**: an event is a change of a single-bit or multi-bit net,
2. **Abstract data modelling**: an event is delivery of a large data packet, such as a complete cache line,

3. **Transactional-level modelling**: avoid events as much as possible: use intermodule software calling.

(Actually, the second style was not lectured this year, bit it's where an EDS event conveys a large struct instead of the new value for a single net.)

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

We can have blocking and non-blocking TLM coding styles:

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

In SystemC: blocking requires an SC_THREAD, whereas non-blocking can use an SC_METHOD. (*CST: Non-examinable 15/16 onwards.*)

Which is better: a matter of style ? Non-blocking enables finer-grained concurrency and closer to cycle-accurate timing results. TLM 2.0 sockets will actually map between different styles at caller and callee.

Also, there are two standard methods for timing annotation in TLM modelling, **Approximately-timed** and **Loosely-timed** and in these notes we shall emphasize the latter.

Figure 6.2 is an example protocol implemented at net-level and TLM level:

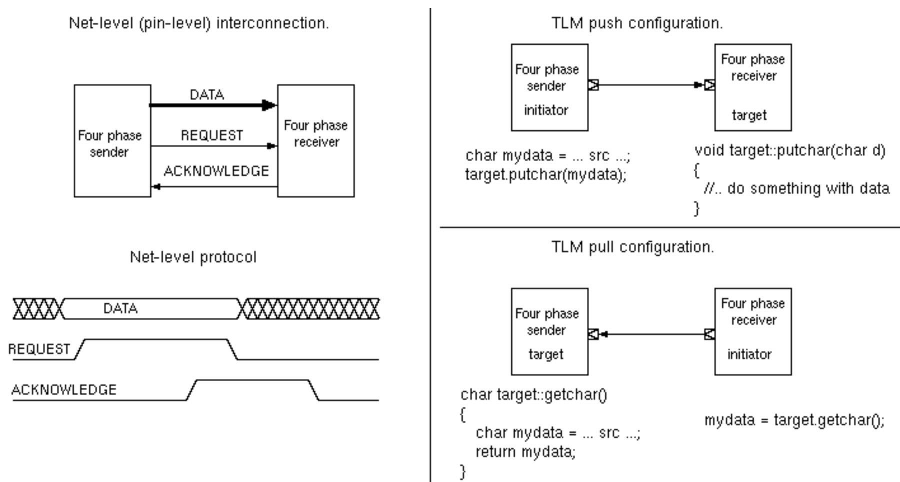


Figure 6.2: Three views of four-phase handshake between sender and receiver: net-level connection and TLM push and TLM pull configurations (untimed).

Note that the roles of initiator and target do not necessarily relate to the sources and sinks of the data. In fact, an initiator can commonly make both a read and a write transaction on a given target and so the direction of data transfer is dynamic. Perhaps try the practical materials: ultra-simple SystemC implementation 'Toy ESL'

6.2.1 General ESL Interactions with Shortcuts Illustrated

Consider the Ethernet CRC example

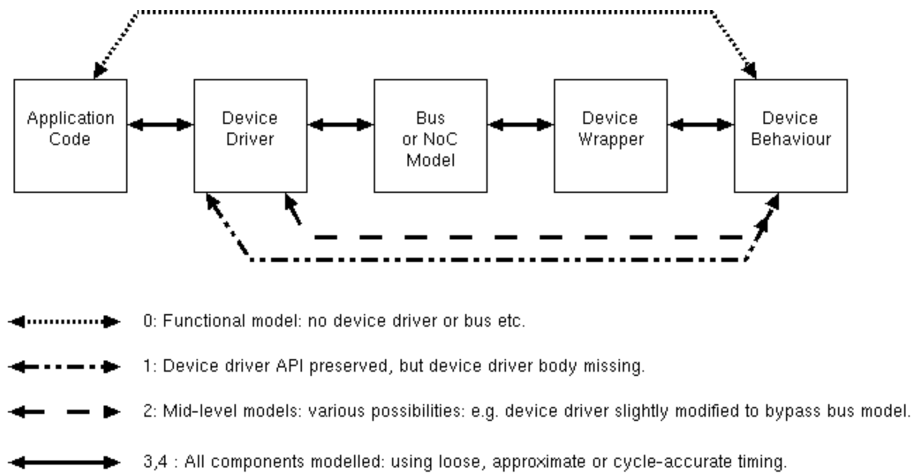


Figure 6.3: Some possible shortcuts through full system model to omit details.

Another view of the higher modelling abstractions:

1. Highest-level (vanished) model: Retains algorithmic accuracy: implemented using SystemC or another threads package: device driver code and device model mostly missing, but the **API to the device driver is preserved**, for instance, a single TLM transaction might send a complete packet when in reality multiple bus cycles are needed to transfer such a packet;
2. Mid-level model: Implemented using SystemC: the device driver is only slightly modified (using preprocessor directives or otherwise) but the interconnection between the device and its driver may be different from reality, meaning bus utilisation figures are unobtainable or incorrect;
3. Bus-transaction accurate mode: each bus operation (read/write or burst read/write and interrupt) is modelled, so bus loading can be established, but timing may be loose and transaction order may be wrong, again, minor changes in the device driver and native compilation may be used;
4. Lower-level models: Implemented in RTL or cycle-accurate SystemC: target device driver firmware and other code is used unmodifed.

Point 3 encompasses mainstream TLM models, like Prazor Virtual Platform

6.2.2 Mixing modelling styles: 4/P net-level to TLM transactors.

An aim of ESL modelling was to be able to easily replace parts of the high-level model with greater detail where necessary. So-called **transactors** are commonly needed at the boundaries.

Here is an example blocking transactor. It forms a gateway from a transactional initiator to a pin-level target.

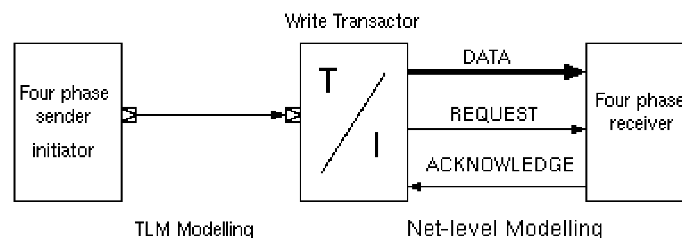


Figure 6.4: Mixing modelling styles using a transactor.

```
// Write transactor 4/P handshake
b_putbyte(char d)
{
  while(ack) do wait(10, SC_NS);
  data = d;
  settle();
  req = 1;
  while(!ack) do wait(10, SC_NS);
  req = 0;
}
```

```
// Read transactor 4/P handshake
char b_getbyte()
{
  while(!req) do wait(10, SC_NS);
  char r = data;
  ack = 1;
  while(req) do wait(10, SC_NS);
  ack = 0;
  return r;
}
```

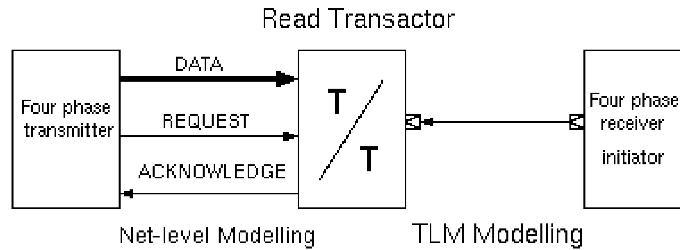


Figure 6.5: Mixing modelling styles using a transactor 2.

6.2.3 Transactor Configurations

Four possible transactors are envisonable for a single direction of the 4/P handshake and in general.

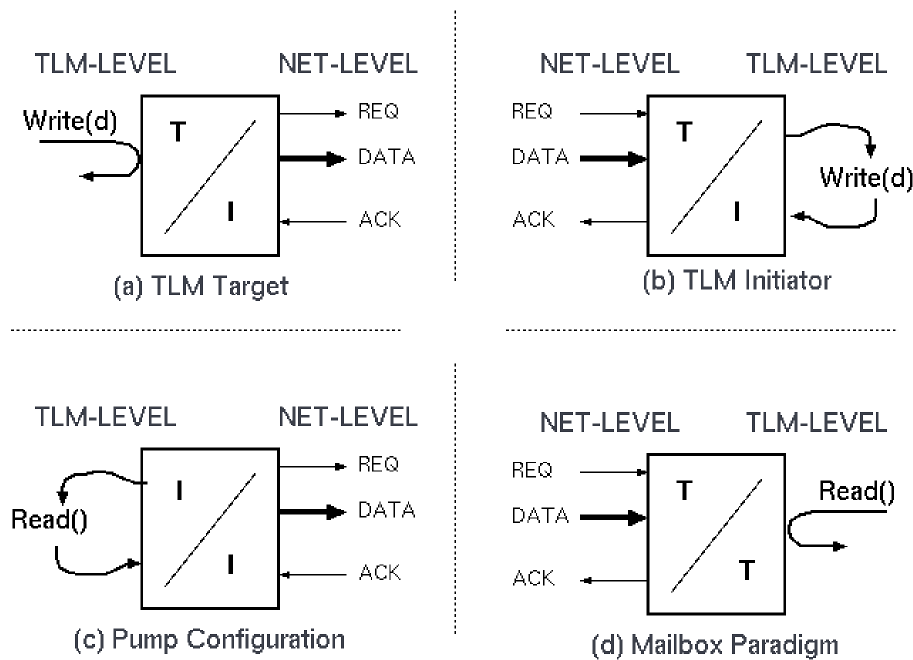


Figure 6.6: Possible configurations for simple transactors.

Additional notes:

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

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

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

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

Consider a transactor with a 'Read()' target port and net-level parallel input. This is an alternative generalisation of the (a) configuration but for when data is moving in the opposite direction. Considering the general case of a bi-directional net-level port with separate TLM entry points for 'Read()' and 'Write(d)' helps clarify.

6.2.4 Example of non-blocking coding style:

Example: Non-blocking (untimed) transactor for the four-phase handshake. *Non-examinable Part II CST.*

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

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

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

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

Both routines should be repeatedly called by the client until returning true.

6.2.5 ESL TLM in SystemC: First Standard TLM 1.0.

```
class my_component: public sc_module, ethernet_if, usb_if
{
  // SC_METHODs and SC_THREADs for normal internal behaviour
  ...

  // methods to implement ethernet_if
  ...

  // methods to implement usb_if
  ...

  // Constructor and sensitivity
  ...
}
```

NB: Full CST credit can be gained using any of TLM1.0 or TLM2.0 styles or your own pseudo code sketches in an OO language of your choice.

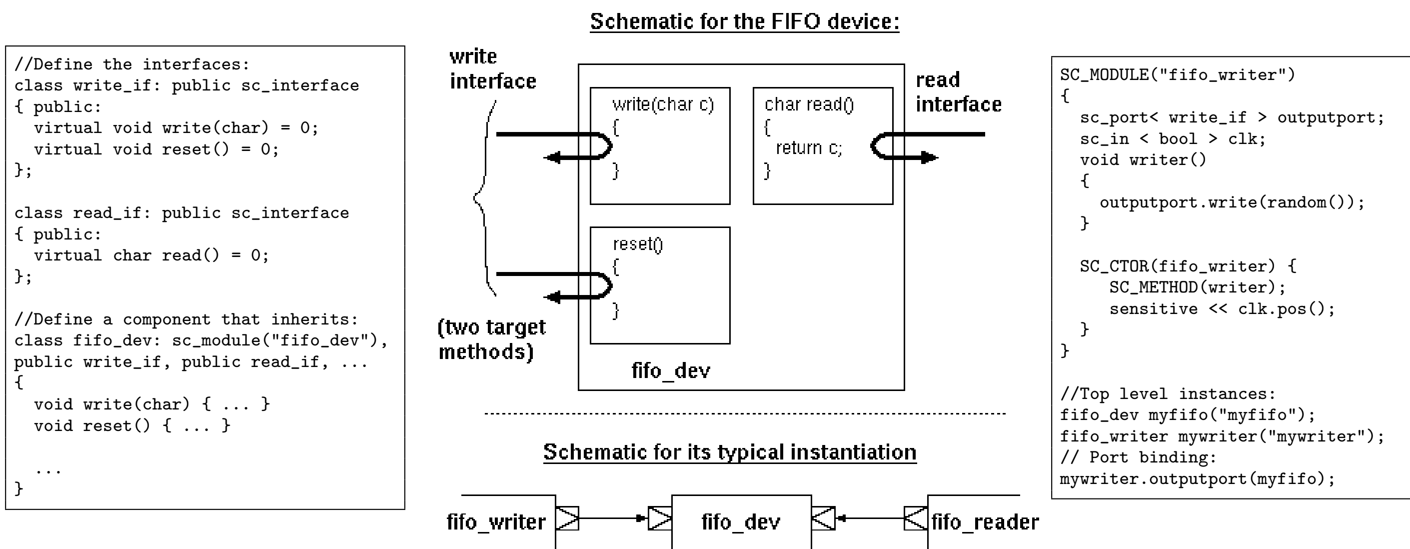
The OSCI TLM 1.0 standard used conventional C++ concepts of multiple inheritance. As shown in the 'Toy ESL' materials and the example here, an SC_MODULE that implements an interface just inherits it.

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

However, TLM 1.0 had no standardised or recommended structure for payloads and no standardised timing annotation mechanisms. There was also the problem of how to have multiple TLM ports on a component with same interface: e.g. a packet router. This is not often needed for software, and hence omitted from high-level languages like C++, but it is common for hardware designs. (A work-around is to add a dummy formal parameter that is given a different concrete type in each instance of an interface ...)

However, referring back to the DMA unit behavioural model (see examples sheet), we can see that that memory operations are likely to get well out of synchronisation with the real system since this copying loop just goes as fast as it can without worrying about the speed of the real hardware. It is just governed by the number of cycles the read and write calls block for, which could be none. The whole block copy might occur in zero simulation time! This sort of modelling is useful for exposing certain types of bugs in a design, but it does not give useful performance results. We shall shortly see how to limit the sequential inconsistencies using a quantum keeper.

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



Here a thread passes between modules, but modules are plumbed in Hardware/EDS netlist structural style. Although sometimes called a 'standard', it is really an ad-hoc coding style.

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

6.2.6 ESL TLM in SystemC: TLM 2.0

Although there was a limited capability in SystemC 1.0 to pass threads along channels, and hence do subroutine calls along what look like wire, this was made much easier SystemC 2.0. TLM2.0 (July 2008) tidies away the TLM1.0 interface inheritance using **convenience sockets** and defines the **generic payload**. It also defined memory/garbage ownership and transport primitives with timing and fast backdoor access to RAM models. And it provided a raft of useful features, such as automatic conversion between blocking and non-blocking styles.

```
// Filling in the fields or a TLM2.0 generic payload:
  trans.set_command(tlm::TLM_WRITE_COMMAND);
  trans.set_address(addr);
  trans.set_data_ptr(reinterpret_cast<unsigned char*>(&data));
  trans.set_data_length(4);
  trans.set_streaming_width(4);
  trans.set_byte_enable_ptr(0);
  trans.set_response_status( tlm::TLM_INCOMPLETE_RESPONSE );

// Sending the payload through a TLM socket:
  socket->b_transport(trans, delay);
```

Rather than having application-specific method names, we standardise on a generic bus operation and demultiplex within various IP blocks based on regsiter address.

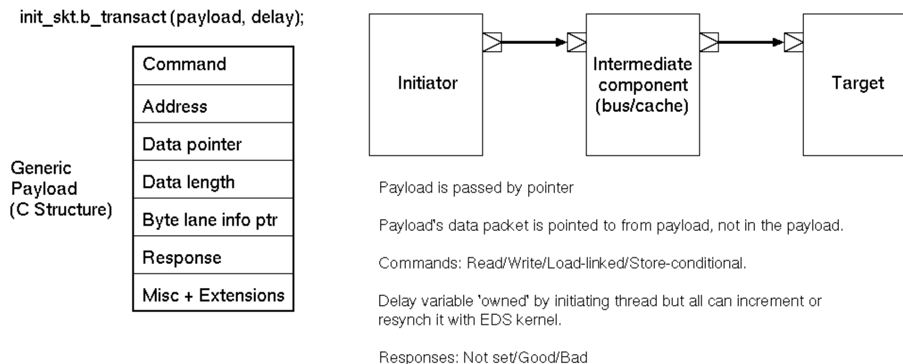


Figure 6.7: The general TLM 2.0 Setup

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

Let's consider a small SRAM example: first define the socket in the .h file:

```
SC_MODULE(cbgram)
{
   tlm_utils::simple_target_socket<cbgram> port0;
   ...
```

Here is the constructor:

```
cbgram::cbgram(sc_module_name name, uint32_t mem_size, bool tracing_on, bool dmi_on): sc_module(name), port0("port0"),
   latency(10, SC_NS), mem_size(mem_size), tracing_on(tracing_on), dmi_on(dmi_on)
{
  mem = new uint8_t [mem_size]; // allocate memory
  // Register callback for incoming b_transport interface method call
  port0.register_b_transport(this, &cbgram::b_transact);
}
```

And here is the guts of b_transact:

```
void cbgram::b_transact(tlm::tlm_generic_payload &trans, sc_time &delay)
{
  tlm::tlm_command cmd = trans.get_command();
  uint32_t   adr = (uint32_t)trans.get_address();
  uint8_t *  ptr = trans.get_data_ptr();
  uint32_t   len = trans.get_data_length();
  uint8_t *  lanes = trans.get_byte_enable_ptr();
  uint32_t   wid = trans.get_streaming_width();

 if (cmd == tlm::TLM_READ_COMMAND)
    {
      ptr[0] = mem[adr];
    }
  else ...

  trans.set_response_status( tlm::TLM_OK_RESPONSE);
}
```

Wire up the ports in the level above:

```
  busmux0.init_socket.bind(memory0.port0);
```

The full code, and many other examples, can be found in the Prazor simulator (see for example `vhls/src/memories/sram64`

(Socket details and types not examinable for Part II CST. The TLM 1.0 style is easier to understand, but not as convenient for real-world projects.)

Sockets of the 'multi' style can be multiply bound and so provide a mux or demux function. Sockets of 'passthrough' style enable a generic payload reference to be passed on.

Figure 6.8: Typical setup showing four socket types.

Additional notes:

TLM 2.0 Socket Types:

simple_initiator_socket.h version of an initiator socket that has a default implementation of all interfaces and allows to register an implementation for any of the interfaces to the socket, either unique interfaces or tagged interfaces (carrying an additional id)

simple_target_socket.h version of a target socket that has a default implementation of all interfaces and allows to register an implementation for any of the interfaces to the socket, either unique interfaces or tagged interfaces (carrying an additional id) This socket allows to register only 1 of the transport interfaces (blocking or non-blocking) and implements a conversion in case the socket is used on the other interface

passthrough_target_socket.h version of a target socket that has a default implementation of all interfaces and allows to register an implementation for any of the interfaces to the socket.

multi_passthrough_initiator_socket.h an implementation of a socket that allows to bind multiple targets to the same initiator socket. Implements a mechanism to allow to identify in the backward path through which index of the socket the call passed through

multi_passthrough_target_socket.h an implementation of a socket that allows to bind multiple initiators to the same target socket. Implements a mechanism to allow to identify in the forward path through which index of the socket the call passed through

The user manuals for TLM 2.0 are linked here ACS P35 Documents Folder

6.2.7 Timed Transactions: Adding delays to TLM calls.

A TLM call does not interact with the SystemC kernel or advance time. To study system performance, however, we must model the time taken by the real transaction over the bus or network-on chip (NoC).

We continue to use SystemC EDS kernel with its `tnow` variable defined by the head of the event queue. This is our main virtual time reference, but we aim not to use the kernel very much, only entering it when inter-module communication is needed.

Note: In SystemC, we can always print the kernel **tnow** with:

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

This reduces context swap overhead (a computed branch that does not get predicted) and we can run a large number of ISS instructions or other operations before context switching, aiming to make good use of the caches on the modelling workstation.

The naive way to add **approximate timing** annotations is to block the SystemC kernel in a transaction until the required time has elapsed:

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

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

The preferred **loosely-timed** coding style is more efficient: we pass a time accumulator variable called ‘delay’ around for various models to augment where time would pass (clearly this causes far fewer entries to the SystemC kernel):

```
// Preferred coding style
// The delay variable records how far ahead of kernel time this thread has advanced.
void b_putbyte(char d, sc_time &delay)
{
  ...
  delay += sc_time(140, SC_NS); // It should be increment at each point where time would pass...
}
```

The leading ampersand on delay is the C++ denotation for pass by reference. But, at any point, any thread can **resynch** itself with the kernel by performing

```
// Resynch idiomatic form:
   wait(delay);
   delay = 0;
// Note: delay has units sc_time so the SystemC overload of {\tt wait}is called, not the O/S posix wait.
```

Important point: **With frequent resynchs, simulation performance is reduced but true transaction ordering is modelled more closely.**

6.2.8 TLM - Measuring Utilisation and Modelling Contention

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

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

Contention like this can be modelled using real or virtual queues:

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

To support style 2, SystemC provides a TLM payload queue: tlm_utils::peq_with_get

6.2.9 Replacing Queues With Delay Estimates

With a virtual queue, although the TLM call passes through the bus/NoC model without suffering delay or experiencing the contention or queuing of the real system, we can add on an appropriate estimated amount.

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

```
// A simple bus demultiplexor: forwards transaction to one of two destinations:
busmux::write(u32_t A, u32_t D, sc_time &delay)
{
   // Do actual work
   if (A >= LIM) port1.write(A-LIM, D, delay) else port0.write(A, D, delay);

   // Measure utilisation (time for the last 100 transactions)
   if (++opcount == 100)
   {  sc_time delta = sc_time_stamp() - last_measure_time;
      local_processing_delay = delay_formula(delta, opcount); // e.g. 1 + 1/(1-p) nanoseconds
      logging.log(100, delta); // record utilisation

      last_measure_time = sc_time_stamp();
      opcount = 0;
   }

   // Add estimated (virtual) queuing penalty
   delay += local_processing_delay;
}
```

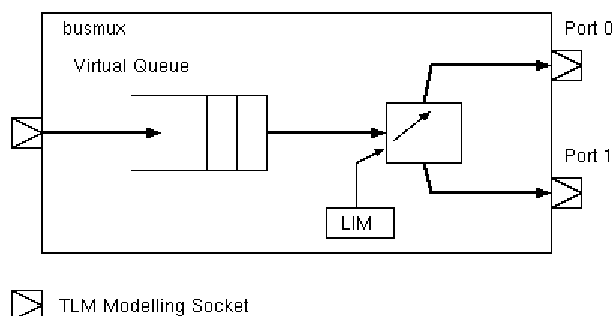


Figure 6.9: Busmux logical schematic diagram.

In the above, a delay formula function knows how many bus cycles per unit time can be handled and hence can compute and record the utilisation and queuing delays.

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

6.2.10 Instruction Set Simulator (ISS)

An Instruction Set Simulator (ISS) is a program that interprets or otherwise models the behaviour of machine code. Typical implementation as a C++ object for ESL:

```
class mips64iss
{ // Programmer's view state:
  u64_t regfile[32];   // General purpose registers (R0 is constant zero)
  u64_t pc;            // Program counter (low two bits always zero)
  u5_t  mode;          // Mode (user, supervisor, etc...)
  ...
  void step();         // Run one instruction
  ...
}
```

The ISS can be **cycle-accurate** or just **programmer-view accurate**, where the hidden registers that overcome structural hazards or implement pipeline stages are not modelled.

This fragment of a main step function evaluates one instruction, but this does not necessarily correspond to one clock cycle in hardware (e.g. fetch and execute would be of different instructions owing to pipelining or multiple issue):

```
void mips64iss::step()
{
  u32_t ins = ins_fetch(pc);
  pc += 4;
  u8_t opcode = ins >> 26;    // Major opcode
  u8_t scode = ins&0x3F;    // Minor opcode
  u5_t rs = (ins >> 21)&31; // Registers
  u5_t rd = (ins >> 11)&31;
  u5_t rt = (ins >> 16)&31;

  if (!opcode) switch (scode) // decode minor opcode
  {
    case 052: /* SLT - set on less than */
      regfile_up(rd, ((int64_t)regfile[rs]) < ((int64_t)regfile[rt]));
      break;

    case 053: /* SLTU - set on less than unsigned */
      regfile_up(rd, ((u64_t)regfile[rs]) < ((u64_t)regfile[rt]));
      break;


      ...
   ...

void mips64iss::regfile_up(u5_t d, u64_t w32)
{ if (d != 0) // Register zero stays at zero
    { TRC(trace("[ r%i := %llX ]", d, w32));
      regfile[d] = w32;
    }
}
```

Various forms of ISS are possible, modelling more or less detail:

Type of ISS	I-cache traffic Modelled	D-cache traffic Modelled	Relative Speed
1. Interpreted RTL	Y	Y	0.000001
2. Compiled RTL	Y	Y	0.00001
3. V-to-C C++	Y	Y	0.001
4. Hand-crafted cycle accurate C++	Y	Y	0.1
5. Hand-crafted high-level C++	Y	Y	1.0
6. Trace buffer/JIT C++	N	Y	20.0
7. Native cross-compile	N	N	50.0

A cycle-accurate model of the processor core is normally available in RTL. Using this under an EDS interpreted simulator will result in a system that typically runs one millionth of real time speed (1). Using compiled RTL, as is now normal practice, gives a factor of 10 better, but remains hopeless for serious software testing (2).

Using programs such as Tenison VTOC and Verilator, a fast, cycle-accurate C++ model of the core can be generated, giving intermediate performance (3). A hand-crafted model is generally much better, requiring perhaps 100 workstation instructions to be executed for each modelled instruction (4). The workstation clock frequency is generally about 10 times faster than the modelled embedded system.

If we dispense with cycle accuracy, a hand-crafted behavioural model (5) gives good performance and is generally throttled by the overhead of modelling instruction and data operations on the model of the system bus.

A JIT (just-in-time) cross-compilation of the target machine code to native workstation machine code gives excellent performance (say 20.0 times faster than real time) but instruction fetch traffic is no longer fully modelled (6). Techniques that unroll loops and concatenate basic blocks, such as used for trace caches in processor architecture, are applicable.

Finally (line 7), compiling the embedded software using the workstation native compiler exposes the unfettered raw performance of the workstation for CPU-intensive code.

6.2.11 Typical ISS setup with Loose Timing (Temporal Decoupling)

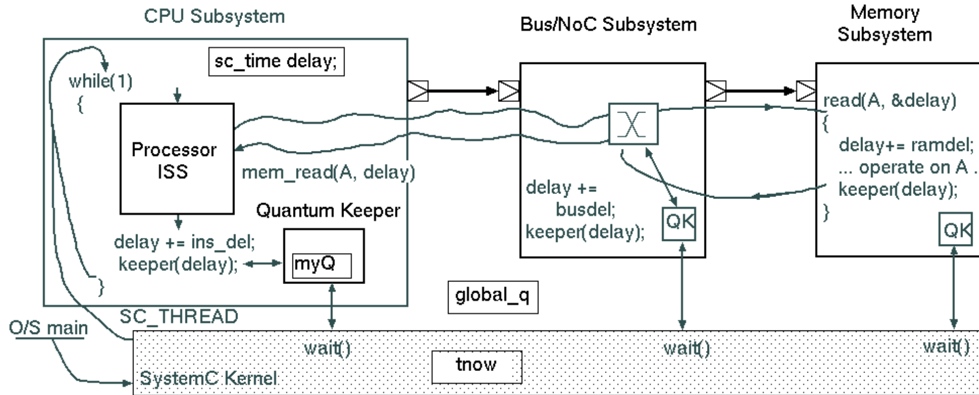


Figure 6.10: Typical setup of thread using loosely-timed modelling with a quantum keeper.

In this reference example, for each CPU core, a single thread is used that passes between components and back to the originator and only rarely enters the SystemC Kernel.

As explained above, each thread has a variable called **delay** of how far it has run ahead of kernel simulation time, and it only yields when it needs an actual result from another thread or because its delay exceeds a locally-chosen value. This is loose timing. Each component increments the delay field in the TLM calls it processes, according to how long it would have delayed the client thread under approximate timing. Every thread must encounter a quantum keeper at least once in its outermost loop.

The quantum keeper code is just a conditional resynch:

```
void keeper(ref delay) { if (delay > global_q) { wait(delay); delay = 0; } }
```

By calling `sc_wait(delay)` the simulation time will advance to where the caller has got to while running other pending processes. The myQuantum could be a system default value or a special value for each thread or component. (We here write `sc_wait` to emphasise it is the SystemC kernel primitive systemc::wait, not the unix call.)

Or where a thread needs to spin, waiting for a result from some other thread:

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

Generally, we can choose the quantum according to our current modelling interest:

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

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

6.3 Power Estimation: RTL versus ESL

RTL simulations can give accurate power figures, especially if a full place-and-route is performed. ESL flows aim to provide a rapid power indications during the early architectural exploration phases.

6.3.1 RTL Operating Frequency and Power Estimation

RTL synthesis is relatively quick but produces a detailed output which is slow to process for a large chip - hence pre-synthesis energy and delay models are desirable. Place-and-route will give accurate wiring lengths but is a highly time consuming investment in a given design point. A simulation of a placed-and-routed design can give very accurate energy and critical path figures, but is likewise useless for 'what if' style design exploration.

A table of possible approaches:

-	**- Without Simulation -**	**- Using Simulation -**
Without Place and Route	Fast - Design exploration. Area and delay heuristics needed.	Can generate indicative activity ratios to be used instead of simulation in further runs.
With Place and Route	Static timing analyser will give an accurate clock frequency.	Gold standard: only bettered by measuring a real chip.

6.3.2 Gold standard: Power Estimation using Simulation Post Layout

Spreadsheet style power modelling from VCD and SAIF logs.

- **VCD**: Verilog Change Dump file - as generated by net-level SystemC or RTL simulations.
- **SAIF**: Switching Activity Interchange Format - the industry standard approach (aka Spatial Archive Interchange Format). Quick Tutorial

Both record the number of changes on each net of circuit from a net-level simulation. Once we know the capacitance of a net (from layout) we can accurately compute the power consumed. But, need to design down to the net-level and do a slow low-level simulation to collect adequate data.

Total Energy = Sum over all nets (net activity ratio * net length)

Clearly, if we know the average net length and average activity ratio we get the same precise answer **regardless of design details**, hence good prospects exist for power estimation from high-level simulations.

6.3.3 RTL Power Estimation by Static Analysis (ie Without Simulation)

Post RTL synthesis we have a netlist and can use approximate models (based on Rent's rule) for wire lengths provided sufficient hierarchy exists (perhaps five or more levels). We can either use the natural hierarchy of the RTL input design or we can apply a clustering/clique finding algorithms to determine a rough placement floorplan without doing a full place and route.

Pre RTL synthesis we can readily collect the following certainties (and hence the static power (ignoring drive strength selection and power gating))

- Number of flip-flops
- Number and bit widths of arithmetic operators
- Size of RAMs

Random logic complexity can be modelled in gate-equivalent units. These might count a ripple-carry adder stage as 4 gates, a multiplexor as 3 gates per bit and a D-type flip-flop as 6 gates.

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

  reg [3:0] count, oldcount; // D-types

  always @(posedge mainclk) begin
      if (cen) count <= count + 1;    // ALU
      if (din) oldcount <= count; // Wiring
      end

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

endmodule
```

But the following dynamic quantities require heuristic estimates:

- Dynamic clock gating ratios
- Flip-flop activity (number of enabled cycles/number of flipping bits)
- Number of reads and writes to RAMs
- Glitch energy in combinational logic.

DRAM power generally comes from a different budget (off chip) and can only really be estimated by dynamic modelling on a real or virtual platform. But note that for small embedded devices, the DRAM static power in its PCB track drivers can dominate DRAM dynamic power.

Non-examinable: There exists a technique to estimate the logic activity using balance equations.

We here use toggle rates, instead of activity ratios.

The balance equations range over a pair of values for each net, consisting of

- **average duty cycle**: the fraction of time at a logic one
- **average toggle rate**: the fraction of clock cycles where a net changes value. This is twice the activity ratio needed for the dynamic power computation at the end.

Consider an XOR gate with inputs toggling randomly. Assuming uncorrelated inputs, the output will also be random and its toggle rate can be predicted to be 50 percent. (c.f. entropy computations in Information Theory). But if we replace with an AND or OR gate, the output duty cycle will be 1 in 4 on average and its toggle rate will be given by the binomial theorem and so on.

Overall, a synchronous digital logic sub-system can be modelled with a set of balance equations (simultaneous equations) in terms of the average duty cycle and expected toggle rate on each net. D-types make no difference. Inverters subtract the duty cycle from unity. The other gates have equations as developed above.

Is this a useful measure? We need the stats for the input nets to run the model. We can look at the partial derivatives with respect to the input stats and if they are all very small, our result will hold over all inputs.

This is not widely used. Instead industry captures the average toggle rate of the nets in a subsystem during simulation runs (SAIF output) of each of the various operating phase/modes.

Some additional dynamic energy is consumed as 'short-circuit current' which is current that passes during switching when both the P and N transistors are on at once, but this is small and we mainly ignore it in these notes. Useful article: POWER MANAGEMENT IN CPU DESIGN

Short-circuit current is proportional to the toggle ratio. The toggle ratio, $t_t r$ is the percentage of clock cycles that see a transition in either direction. The net toggle rate = Operating frequency of the chip $f \times t_r$;

- 1 W/cm^2 can be dissipated from a plastic package.
- 2-4 W/cm^2 requires a heat sink.
- more than 8 W/cm^2 requires forced air, heatpipe or water cooling.

Workstation and laptop microprocessors dissipate tens of Watts: hence cooling fans and heat pipes. In the past we were often *core-bound* or *pad-bound*. Today's SoC designs are commonly *power-bound*.

6.3.4 Typical macroscopic performance equations: SRAM example.

It is important to model SRAM accurately. A 45nm SRAM can be modelled at a macro level in terms of its Area, Delay and Power Consumption:

Four rules of thumb (scaling formulae) for single-ported SRAM CACTI at HP labs. Cacti RAM Models

Technology parameters:

- Read width 64 bits. Technology Size (nm):45 Vdd:1.0
- Number of banks: 1 Read/Write Ports per bank:1
- Read Ports per bank: 0 Write Ports per bank:0

Interpolated equations:

- Area = 13359.26+4.93/8*bits squm: gradient = 0.6 squm/bit.
- Read energy = 5 + 1.2E-4 / 8 * bits pJ.
- Leakage (static power) = 82nW per bit.
- Random access latency = 0.21 + 3.8E-4(sqrt(bits)) nanoseconds * 1.0/supply voltage.

Another rule of thumb: area is about 600 square lambda for an SRAM bit cell, where lambda is the feature size (45E-9).

6.3.5 Typical macroscopic performance equations: DRAM example.

A DRAM channel is some number of DRAM die (eg. 16) connected to a set of I/O pads on a controller. The channel data width could typically be 16, 32 or 64 bits. The capacity might be 16 GByte.

- **Controller static power**: The pads forming the so-called 'phy' will typically consume half a watt or so, even when idle.
- **DRAM static power**: each die takes about 100 mW when idle but may enter a sleep mode if left unused for a millisecond or so, reducing this to 10 mW or so.

- Each **row activation** takes dynamic energy (see table).
- Each **column burst transfer** takes on-chip energy and PCB trace energy.
- Each **row closure** (writeback/de-activate) takes dynamic energy (see table).
- Refresh operations consume a small mount of dynamic energy (see exercise sheet for numbers).

Table 2: DRAM die area and row activation energy breakdown of a 2Gb x8 DDR3-1600 DRAM chip.

Area (mm^2)			
DRAM cell	4.677	Sense amplifier	1.909
Row predecoder	0.067	Local wordline driver	1.617
Total area (including other components)			11.884
Energy per MAT (pJ)			
Local bitline	15.583	Local wordline	0.046
Local sense amplifier	1.257	Row decoder	0.035
Total row activation energy per MAT			16.921
Energy per bank (pJ)			
Row activation bus	17.944	Row predecoder	0.072
Total row activation energy per bank			288.752

Figure 6.11: DRAM activation energies.

Partial Row Activation for Low-Power DRAM System - Lee, Kim and HongCalculating Memory System Power for DDR3

The Prazor simulator integrates the University of Maryland DRAM simulator.

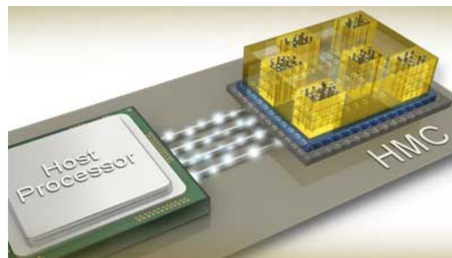


Figure 6.12: Die-stacked DRAM subsystem: The Micron Hybrid Memory Cube.

The phy is the set of pads that operate with high performance to drive the PCB traces. Such power can be minimised if the traces are kept short with using multi-chip modules or die stacking. Micron have released a multi-channel DRAM module: the Micron HMC. This can have a number of host nodes sharing a number of die-stacked DRAM chips.

6.3.6 Rent's Rule Estimate of Wire Length

If we know the physical area of each leaf cell we can estimate the area of each component in a heirarchic design (sum of parts plus percentage swell).

Rent's rule pertains to the organisation of computing logic, specifically the relationship between the number of external signal connections to a logic block with the number of logic gates in the logic block, and has been applied to circuits ranging from small digital circuits to mainframe computers [Wikipedia].

Rent's rule uses a simple power-law relationship for the number of external nets to a sub-system as function of the number of logic gates in the sub-system. Figure 6.13 shows three possible design styles

that vary in rent coefficient. A circuit composed of components with no local wiring between them is the other extreme possibility, with a Rent exponent of 1.0. But the rule-of-thumbs is that for most 'general' subsystems a Rent exponent varying between about 0.5 and 0.7 is seen. Circuits like the shift register can be outliers, having no increase in external connectivity regardless of length: these have a Rent exponent of 0. The same situation arises with an accelerator on-a-stick, where the degree of unfold will alter the rent exponent owing to the fixed-configuration bus port.

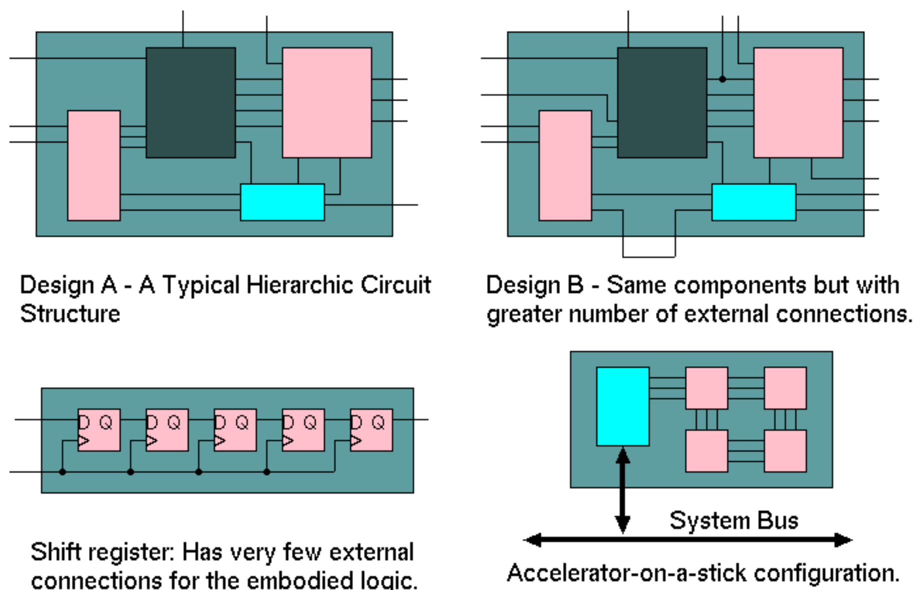


Figure 6.13: Two similar designs with different Rent exponents and two non-Rentian design points.

Generalisations of Rent's rule can model wWire length distribution (with good placement). For a single level of design hierarchy, the random placement of blocks in a square of area defined by their aggregate area gives one wire length distribution (basic maths). A careful placement is used in practice, and this reduces wire length by a Rent-like factor (eg. by a factor of 2). With a heirarchic design, where we have the area use of each leaf cell, even without placement, we can follow a net's trajectory up and down the hierarchy and apply Rent's Rule. Hence we can estimate a signal's length by sampling a power law distribution whose 'maximum' is the square root of the area of the lowest-common-parent component in the hierarchy.

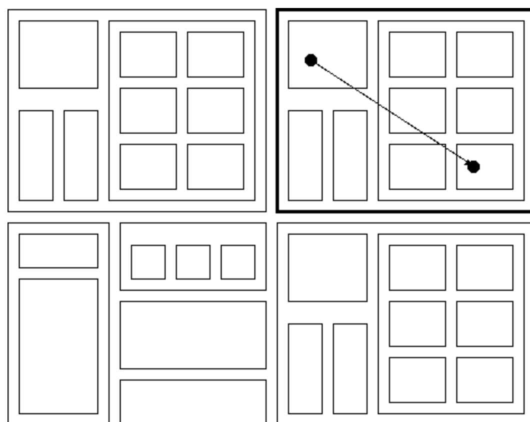


Figure 6.14: Illustrating Lowest Common Parent of the endpoint logic blocks. (This will always be roughly the same size for any sensible layout of a given design, so having a detailed layout like the one shown is not required).

6.3.7 Macroscopic Phase/Mode Power Estimation Formula

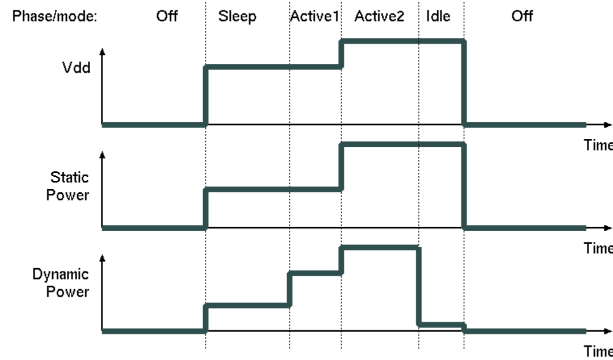


Figure 6.15: Typical Phase and Mode Time-Domain Plot.

An IP block will tend to have an average power consumption in each of its phases or modes. Power modes include sleep, idle, off, on etc.. Clock frequency and supply voltage are also subject to step changes and expand the discrete phase/mode operating space. Given that blocks switch between energy states a simple energy estimation technique is based percentage of time in each state. This was how the TLM POWER2 library for SystemC worked. TLM POWER3 uses this approach for static power but logs energy quanta for each transaction.

The Parallella board has two USB line drivers (circled in red). They have three operating modes with different energy use in each.

Phase/Mode Power Model	USB Phy	Part no:	USB3320C-EZK-TR				
Mode	**Rail 1**	**Rail 1**	**Rail 2**	**Rail 2**	**Rail 3**	**Rail 3**	**Total Power**
	V	**A**	**V**	**A**	**V**	**A**	**W**
Idle	3.30E+000	1.80E-005	1.80E+000	7.00E-007	3.30E+000	3.00E-005	**1.60E-004**
L/S Mode	3.30E+000	6.30E-003	1.80E+000	1.10E-002	3.30E+000	5.00E-003	**5.71E-002**
H/S mode	3.30E+000	2.90E-002	1.80E+000	2.20E-002	3.30E+000	5.90E-003	**1.55E-001**

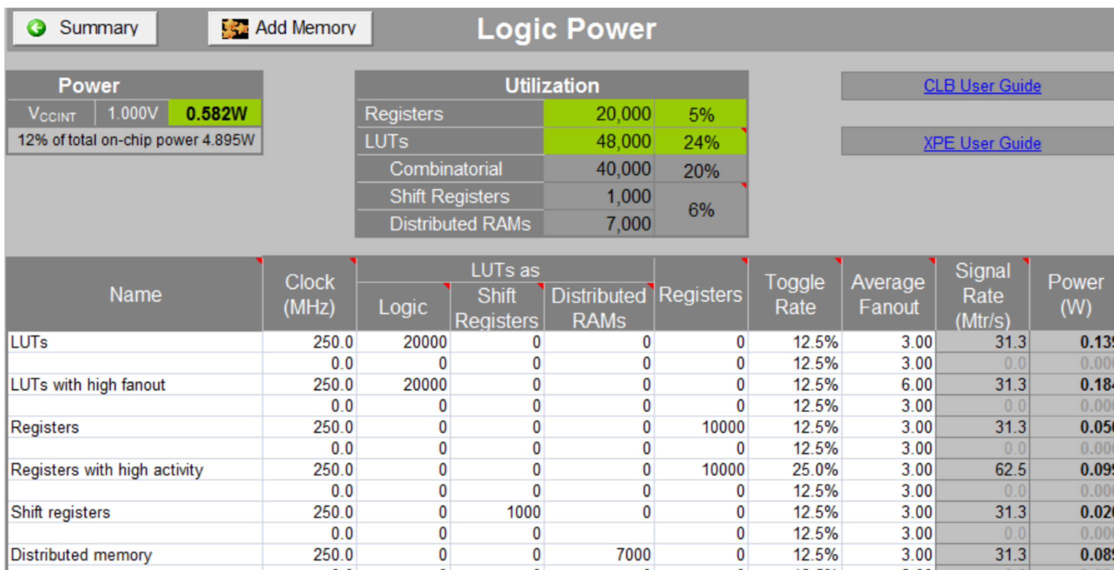
Figure 6.16: Phase/Mode figures for a USB Phy Line Driver.

6.3.8 Spreadsheet-based Energy Accounting

Knowing the average number of operations per second on a unit is generally all that is needed to work out its average energy, once the joules per operation are known.

The Xilinx Xpower 28 nm Zynq spreadsheet models about 415 pJ per ARM clock cycle and 23 pJ per DSP multiply. BRAM reads of RAMB18x2 (36 Kbit) units take 8.5 pJ and writes about 10 percent less. The formula earlier for 45nm, 5.0 + 1.2e-4 / 8.0 * mbits gives 5.5 pJ but twice as much for a write.

But the totals for each component drastically depend on the activity ratios and initial guesses are typically set close to worst case which is conservative, but typically wildly out. Hence SAIF-based or other dynamic trace information must be fed in for an accurate result.


Summary
Add Memory
Logic Power

Power		
VCCINT	1.000V	0.582W
12% of total on-chip power 4.895W		

Utilization		
Registers	20,000	5%
LUTs	48,000	24%
Combinatorial	40,000	20%
Shift Registers	1,000	6%
Distributed RAMs	7,000	

CLB User Guide
XPE User Guide

Name	Clock (MHz)	LUTs as Logic	LUTs as Shift Registers	LUTs as Distributed RAMs	Registers	Toggle Rate	Average Fanout	Signal Rate (Mtr/s)	Power (W)
LUTs	250.0	20000	0	0	0	12.5%	3.00	31.3	0.139
	0.0	0	0	0	0	12.5%	3.00	0.0	0.000
LUTs with high fanout	250.0	20000	0	0	0	12.5%	6.00	31.3	0.184
	0.0	0	0	0	0	12.5%	3.00	0.0	0.000
Registers	250.0	0	0	0	10000	12.5%	3.00	31.3	0.050
	0.0	0	0	0	0	12.5%	3.00	0.0	0.000
Registers with high activity	250.0	0	0	0	10000	25.0%	3.00	62.5	0.099
	0.0	0	0	0	0	12.5%	3.00	0.0	0.000
Shift registers	250.0	0	1000	0	0	12.5%	3.00	31.3	0.020
	0.0	0	0		0	12.5%	3.00	0.0	0.000
Distributed memory	250.0	0	0	7000	0	12.5%	3.00	31.3	0.089
	0.0	0	0	0	0	12.5%	3.00	0.0	0.000

Figure 21: **Effect of LUT Configuration, Toggle Rates and Average Fanout on Power Estimation (7 Series)**

Figure 6.17: Example Xilinx Xpower Spreadsheet

6.3.9 Higher-Level Simulation - Virtual Platforms

There are a number of full-system simulators or 'virtual platforms' in academia and industry. Examples include QEMU, Sniper, ZSim, GEM5 and Prazor Virtual PlatformFor further reading in part III: ACS P35

End of notes. ©DJ Greaves, Feb 2018.