KG 1 — High-level Design Capture and Synthesis

THIS DOCUMENT IS A DRAFT UNDER CONSTRUCTION AT THE MOMENT

Manual coding of hardware circuits is specialised and time-consuming. But the energy saved using custom hardware compared with conventional processing, whether for video compression on a mobile phone or weather forecasting in the cloud (sic), has become very attractive. Performance can also be much higher, especially for CPU-bound (i.e. not memory-bound or I/O-bound) applications, such as encryption and prime-number factoring. And for automated stock trading the achievable low-latency is another key benefit. So today, we see a number of drivers for taking easy-to-write software and mapping it to a hardware circuit. This process is generally called 'High-level Synthesis' or HLS.

- **(1) High-level Design Capture and Synthesis**

1.0.1 Accellera IP-XACT

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.

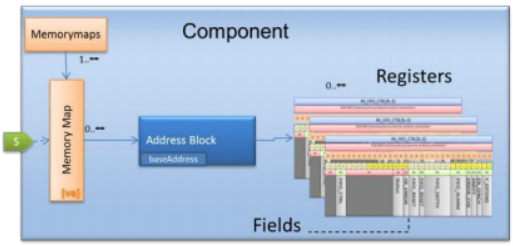


Figure 1.1: IP-XACT captures memory map and register field definitions.

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.

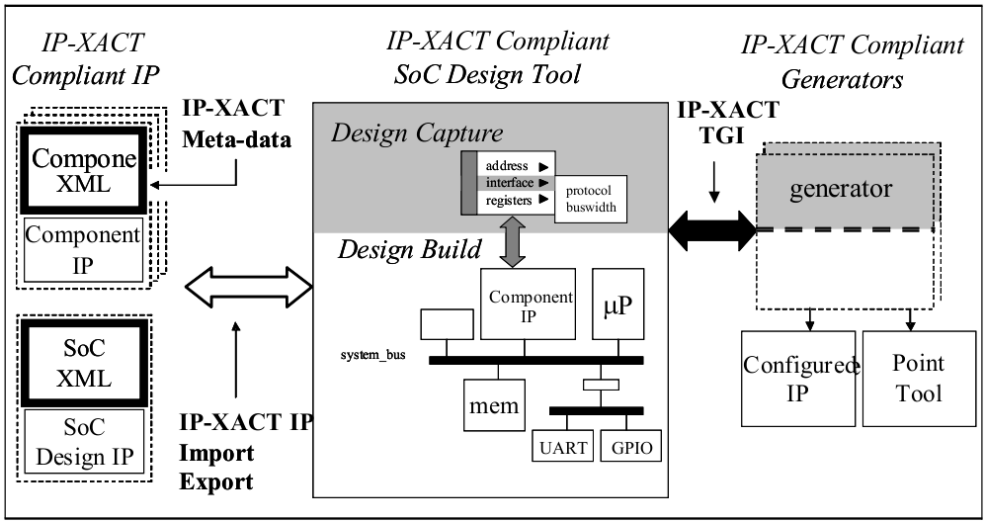


Figure 1.2: Reference Model for design capture and synthesis using IP-XACT blocks.

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.

1.0.2 Start Here

Start Slide

The first half or majority of these slides covers 'classical HLS'. The final slides or 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.

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

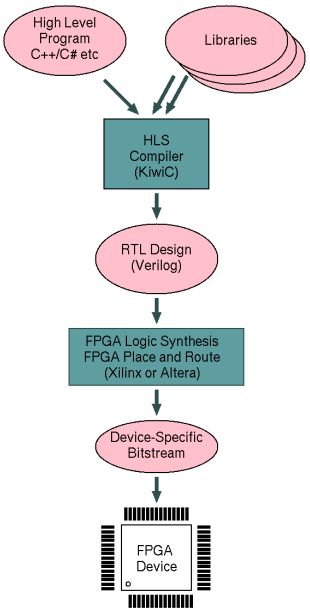


Figure 1.3: Basic Steps of an HLS Flow.

- 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 §**??**).

- 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 complex out-of-order CPUs.

1.0.3 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. There is also the **generate** statement in Verilog and HLD RTLs. 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 (CSP or Kahn Networks) that can be split over processing nodes. They are agnostic as to what mix of nodes is used: FPGA, GPU or CPU. pn tool for Process Networks.

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, rules 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 (as in Bluespec) is pure RTL: which is declarative (since no threads).

All higher-level styles are amenable to substantial automatic design space exploration by the compiler tool. The tool performs **datapath** and **schedule** generation, including re-encoding and re-pipelining to meet timing closure and power budgets.

So-called 'classical HLS' converts a behavioural thread to a static schedule (fixed at compile time). But the parallelism inferred is always limited. Sometimes no scheduller is needed. This can mean a fully-pipelined implementation can be generated. Alternatively, a systolic array or process network can be rendered, that again has no sequencer, but which accepts data with an initiation interval greater than unity (fixed for systolic array and variable for a CSP/Kahn-like network).

Transistors are abundant and having a lot of hardware is not itself a problem (We discussed the *Power Wall* in §**??**). 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/Handel-C/Kahn Networks).

1.0.4 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++;
}
```

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

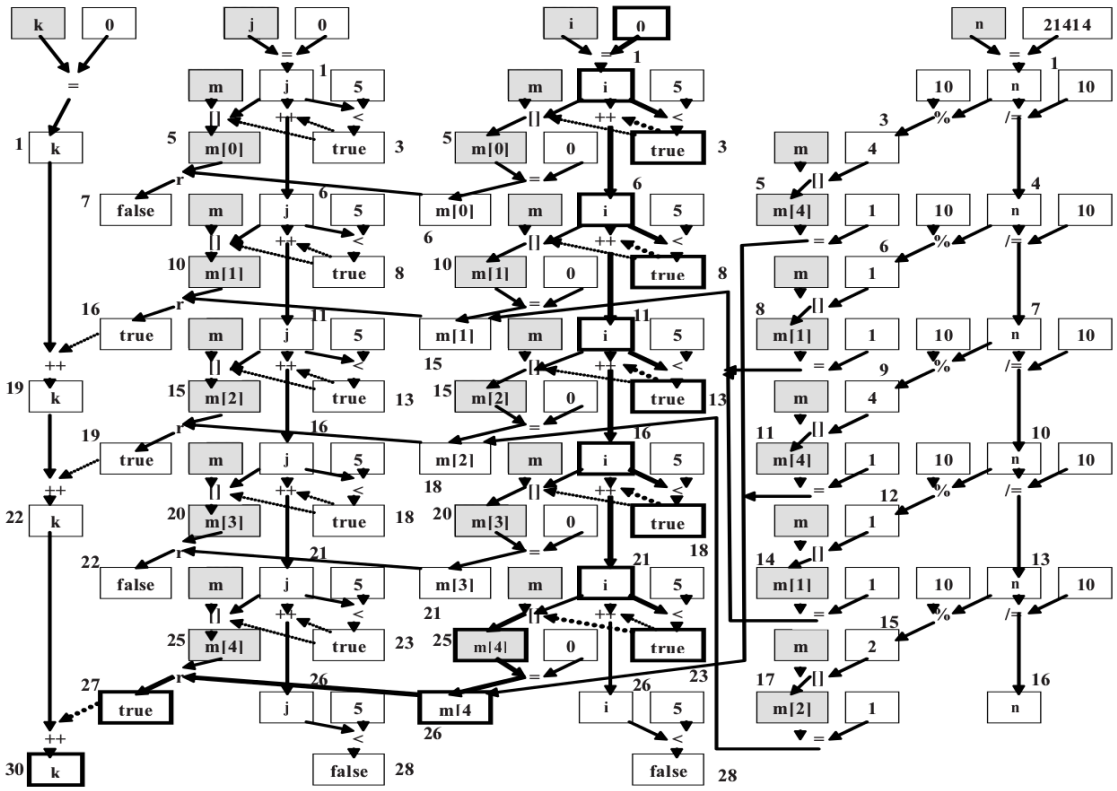


Figure 1.4: Critical Path in Digit Counting C Program (Prihozhy).

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 (§**??**), 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!)

(This program counted the divide by 10 as one operation: HLS addresses the multi-cycle nature of operations such as load and division, which gives a more complex timing diagram.)

1.0.5 Beyond Pure RTL: Behavioural descriptions of hardware.

What has 'synthesisable' RTL traditionally provided ?

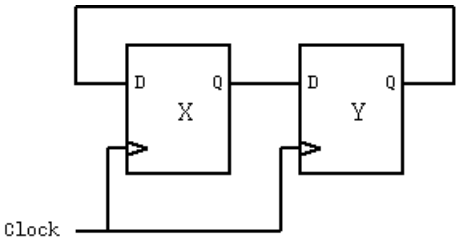


Figure 1.5: A circuit to swap two registers.

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 1.5. (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 §**??**, 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;
}
```

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 (also known as a constructive 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.

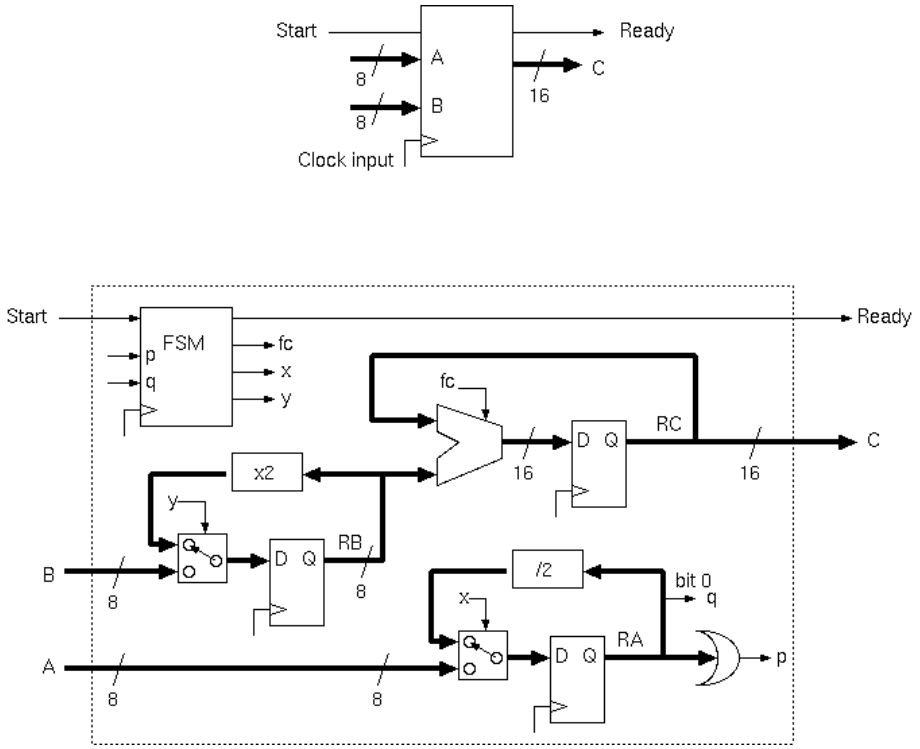


Figure 1.6: Long multiplier viewed as datapath and sequencer.

1.0.6 Multiplier Answer (2)

```
module LONGMULT8b8(clk, reset, C, Ready, A, B, Start);

   input clk, reset, Start;
   output Ready;
   input [7:0] A, B;
   output [15:0] C;
   reg [15:0] RC, RB, RA;
   reg        Ready;

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

   reg xx, yy, qq, pp; // Control and predicate nets
   reg [1:0] fc;
   reg [3:0] state;
   always @(posedge clk) begin
      xx = 0; // default settings.
      yy = 0;
      fc = 0;

      // Predicates
      pp = (RA!=16'h0);   // Work while pp holds
      qq = RA[0];         // Odd if qq holds
      // Sequencer
      if (reset) begin
         state <= 0;
         Ready <= 0;
         end
      else case (state)
             0: if (Start) begin
                xx = 1;
                yy = 1;
                fc = 2;
                state <= 1;
             end

             1: begin
                fc = qq;
                if (!pp) state <= 2;
             end
             2: begin
                Ready <= 1;
                if (!Start) state <= 3;
             end

             3: begin
                Ready <= 0;
                state <= 0;
             end

           endcase // case (state)

      // Datapath
      RB <= (yy) ? B: RB<<1;
      RA <= (xx) ? A: RA>>1;
      RC <= (fc==2) ? 0: (fc==1) ? RC+RB: RC;
      end

   assign C = RC;
endmodule
```

Suitable test wrapper:

```
module SIMSYS();
   reg clk, reset, Start;
   wire Ready;
   wire [7:0] A, B;
   wire [15:0] C;

   // Reset Generator
   initial begin reset = 1; # 55 reset = 0; end


   // Clock Generator
   initial begin clk = 0; forever #10 clk = !clk; end

   // Stimulus
   assign A = 6;
   assign B = 7;

   // Handshake control
   always @(posedge clk) Start <= !Ready;

   // Console ouput logging:
   always @(posedge clk) $display("Ready=%h  C=%d");

   // Device under test.
   LONGMULT8b8 the_mult(clk, reset, C, Ready, A, B, Start);

endmodule // SIMSYS
```

1.0.7 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?

1.0.8 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 experiments on the following link may be discussed in lectures: Kiwi Bit Tally (Ones Counting) Comparison

A good HLS tool should not be sensitive to coding style and should use strength reduction and sub-expression sharing and all standard compiler optimisations. (But we'll soon discuss that layout of data in memories is where we should exercise care and where automated techniques can help less).

(**Strength Reduction** is compiler-speak for replacing one operator with a less costly operator where possible, such as replacing multiply by -1 with subtract from 0).

1.0.9 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 many commercial HLS tools are widely available: SystemCrafter, Calypto Catapult, SimVision, CoCentric, C-Level Design, Forte Cynthesizer (now acquired by Cadence), C-to-Verilog.com and xPilot now called Vivado HLS, ... other tools are/were HardwareC, SpecC, Impulse-C, NEC Cyber Workbench, Synopsis SynphonyC.

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 and 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 in this 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 CSharp are better, owing to stronger typing and banning of arithmetic on object handles (all subscription operations apply to first-class arrays).

1.0.10 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,
- **manual**, based on pragmas or other mark up inserted in the source code.

1.0.11 Pipelined Schedulling - One Basic Block

After some amount of 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.

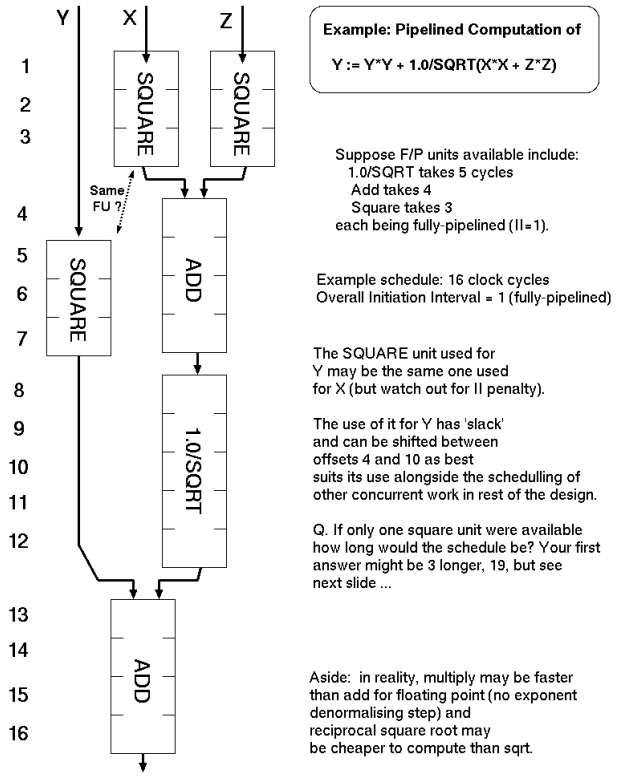


Figure 1.7: Example static schedule for a basic block containing a single assignment.

The diagram we have just considered was a little naive: it was constructed assuming that the FUs were non-pipelined (i.e. their II was the same as their latency). Non-pipelined versions of some kinds of FU are more compact (less silicon area) than pipelined equivalents, so they do crop up. But we were told our floating-point FUs had II=1.

Our new design uses only one SQUARE unit at the expense of latency being one higher. II is often more critical than latency for HLS, and our II is only 3.

A good heuristic for schedulling is to start the operations which have the largest processing delay as early as possible. This is called a *list schedule*. But integer linear programming packages are often used to find an optimum schedule and resource use trade off.

Our example generated only one output. 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 output 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 array subscripts are equal. This is 'undecidable' in general theory, but often doable in practice. Indeed many subscript expressions will be simple functions of loop 'induction variables' whose pattern we need to understand for high performance.

1.0.12 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 competing with itself for structural resources and data hazards. This is **loop pipelining**. Where an outer loop is requested to be pipelined, all loops inside must

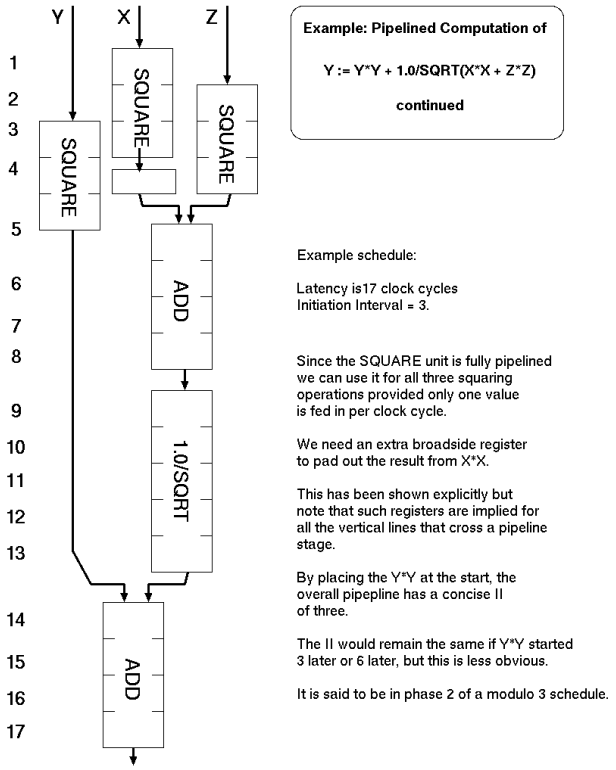


Figure 1.8: The same example but now with just one SQUARE unit and exploiting the fact it is fully-pipelined.

be pipelined.

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.

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 1.9, 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

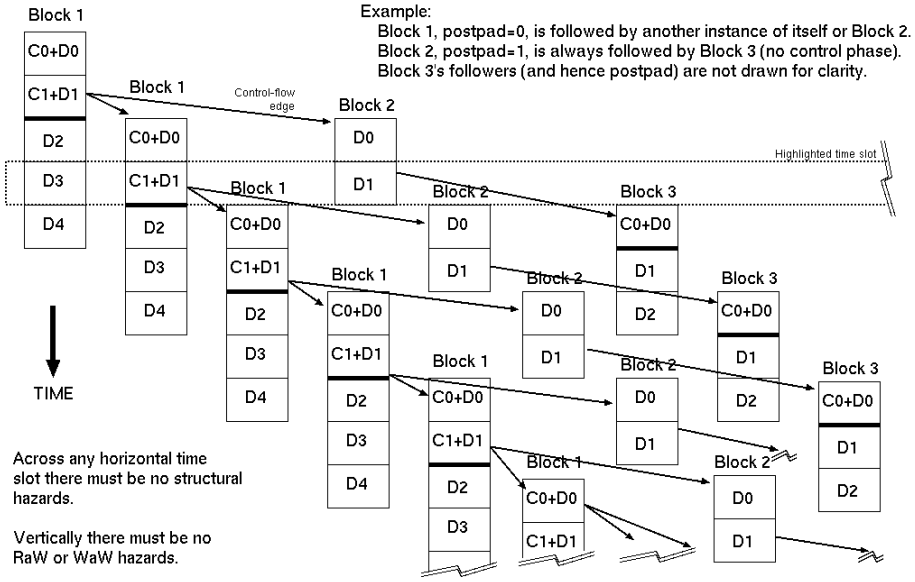


Figure 1.9: Fragment of an example inter-block initiation and hazard graph.

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.

1.1 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.

1.1.1 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:

- 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**'.

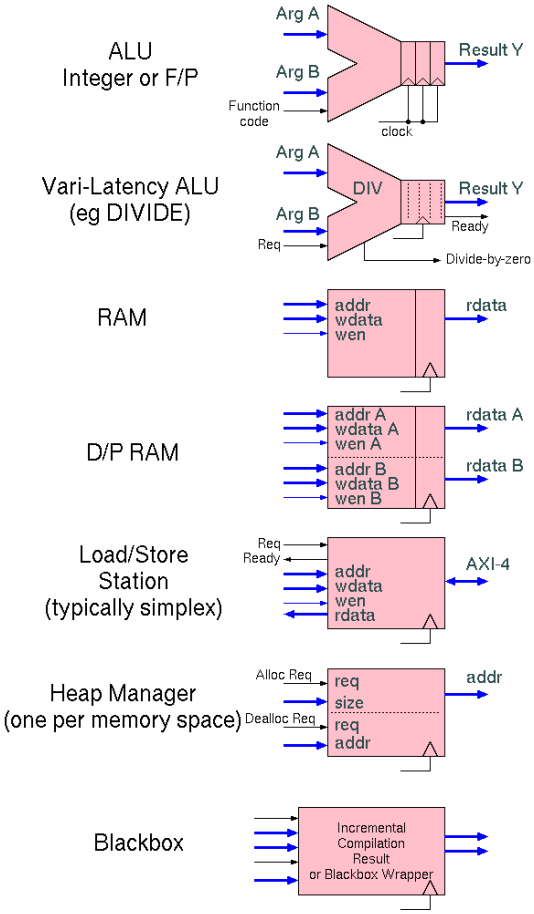


Figure 1.10: Typical examples of FUs deployed by an HLS compiler.

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).

1.1.2 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.

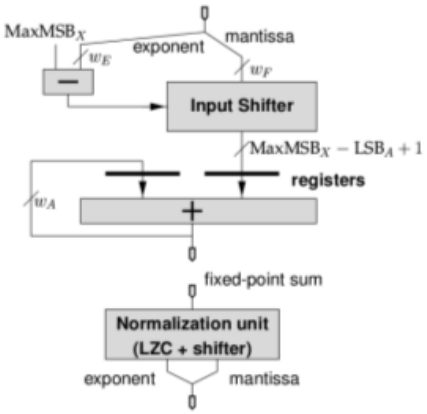


Figure 5: The proposed accumulator. Only the registers on the accumulator itself are shown, the rest of the design is combinatorial and can be pipelined arbitrarily.

Figure 1.11: 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).

1.2 Discovering Parallelism: Classical HLS Paradigms

The well-known 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;
    }
  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 can be determined is an important consideration and compilers will pull this to the start of the block schedule (as illustrated in figure 1.7). 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

Compared with software compilers, like gcc, HLS tools are currently relatively immature. Here is an example from Vivado HLS of a false positive on loop-carried dependency. The following refuses to compile. LINK

```
//directive PIPELINE set
uint8_t process_image(uint8_t pxVal)
{
    static unsigned int x = 0;
    static uint8_t lines[16][WIDTH];

         //directive UNROLL set
    for(int i=0; i &lt; 15; i++) {
        lines[i][x] = lines[i + 1][x];
    }
    lines[15][x] = pxVal;
    uint8_t result = lines[1][x];
    x = (x + 1) == WIDTH ? 0 : (x + 1);
    return result;
}
```

Why does this fail? When we convert from a 2-D to a 1-D array by multiplying up the indecies we loose information that the subscripts are manifestly distinct.

1.2.1 Modulo Schedulling

The high-level expression of a basic block may contain some number of arithmetic operators, such as 9 additions and 10 multiplies for a 10-stage FIR filter. But we may wish to render this in hardware using fewer FUs. For instance, to do this in 3 clock cycles, 3 adders and 4 multipliers are nominally sufficient as this will meet the required basic operations-per-second budget.

But the FUs will often be pipelined and the output from one may not be ready for the input to the next in time.

If data is streaming, then the problem should be solvable with the minimal number of FUs given sufficient pipelining. The pipeline initiation interval (II) will be 3, but the latency larger than 3.

Streaming hardware designs with low II are suprisingly complex and difficult to find when the ALUs available are heavily pipelined. Floating point ALUs tend to have multi-cycle delay (latency of four to six cycles is commmon for add and multiply in FPGA).

A scheduller program will solve the problem of making a static mapping of operations to FUs. It can be phrased nicely as an ILP problem (Sittel/Koch 'ILP-based Modulo Scheduling and Binding for Register Minimization' FPL2018). The result is called a **modulo schedule**. The modulo basis will be the II, 3. Once the schedule is computed it is then fairly simple to render a sequencer circuit and additional holding registers as needed.

A design with initiation interval of one is said to be *fully-pipelined* whereas a higher initiation interval can generally be supported with less silicon area using *modulo schedulling*.

Example 1: The Running Sum

If the adder has unity latency the running sum example is simple. For a two-cycle adder with three inputs, it is also trivial. The minimal circuit for latency-of-two adders with only two inputs is surprisingly complex.

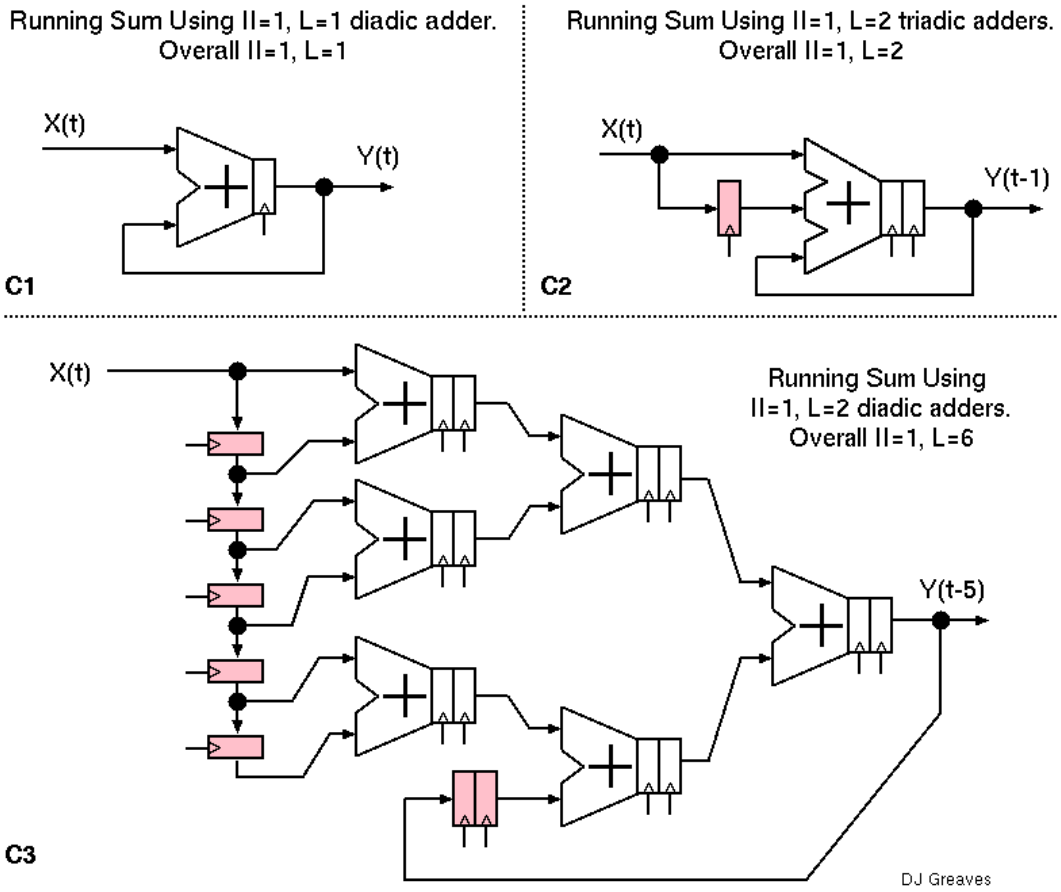


Figure 1.12: Various circuits that perform a running sum.

Example 2: Bi-quad Filter Element

Rearranging the operator association can greatly help with producing a good schedule.

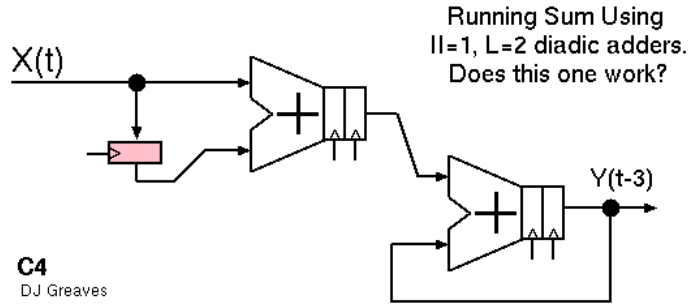


Figure 1.13: A slightly odd-looking possible solution. *Ex:* check it works.

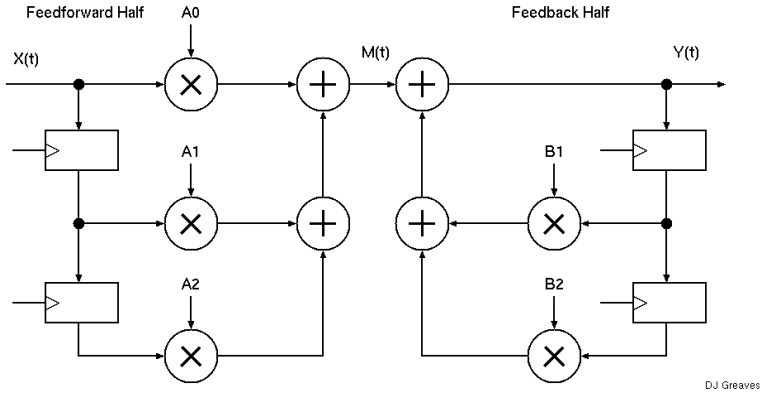


Figure 1.14: Standard circuit for a bi-quad filter element.

1.2.2 Memory Banking and Widening

Whether computing on standard CPUs or FPGA, memory bandwidth is often a 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 (aka channels) can be accessed simultaneously at different locations, whereas memories with a wider word are accessed at just one location at a time (per port). Both yield more data for each access. Both also may or may not need lane steering or a crossbar routing matrix, depending on the application and allowable mappings of data to processing units.

The best approach also depends on whether the memories are truly random access. SRAM is truly random access, whereas DRAM will have different access times depending on what state the target bank (i.e. bit plane) is in.

With multiple 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 inevitably deployed. The question is which bits to use.

Using low bits causes a fine-grained interleave, but may either destroy or leverage spatial locality 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 the best cases (easiest applications), we need no lane-steering or interconnect switch and each processing element acts on just one part of the wide data bus. This is basically the GPU architecture.

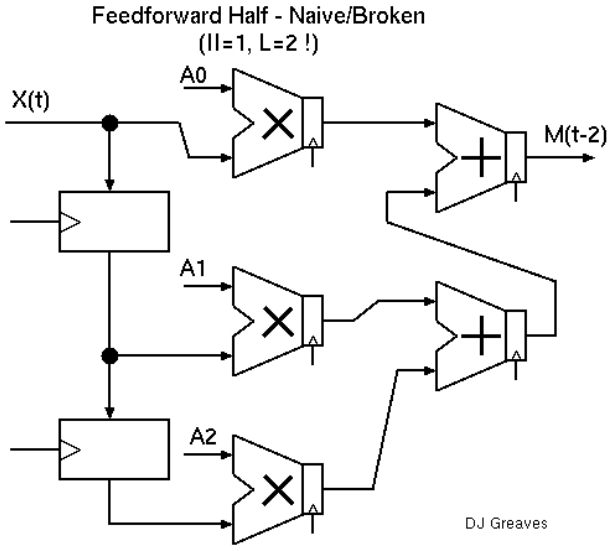


Figure 1.15: Feed-forward part of the bi-quad possible design?

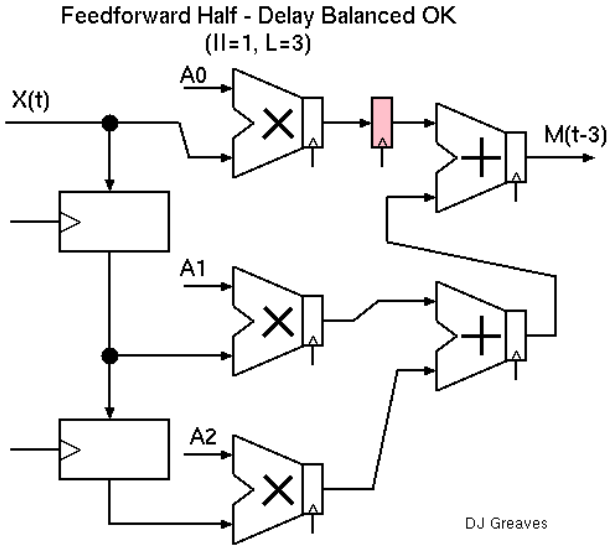


Figure 1.16: Corrected feed-forward section of bi-quad filter element.

1.2.3 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 BWT which itself is a precomputed transform of the haystack. 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).

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.

Access patterns to the Rank array will exhibit no spatial or temporal locality (especially at the start of the search

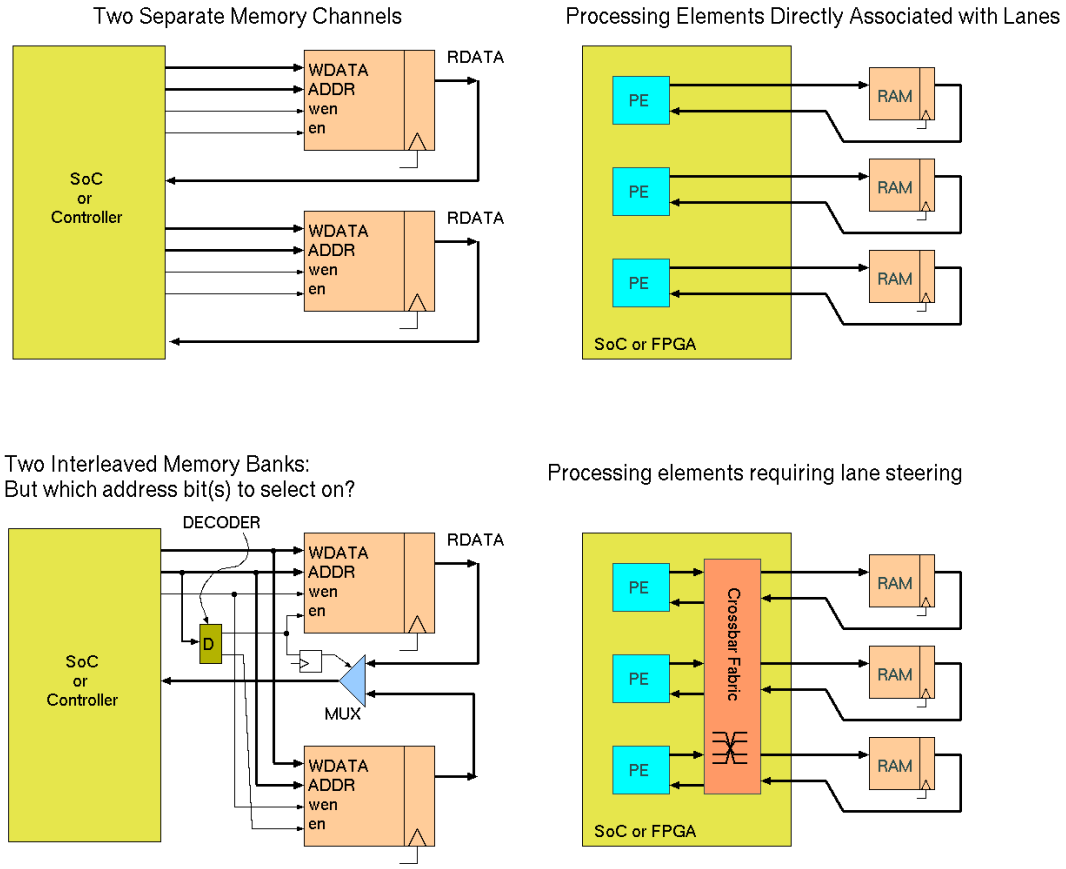


Figure 1.17: Alternative Memory Bank Structures giving wider effective bus width.

when `start` and `end` are well separated. If only one bank (here we mean channel) of DRAM is available, then random access delays dominate. (The only way to get better performance is task-level parallelism: searching for multiple needles at once, which at least overcomes the round-trip latency to the DRAM, but not the low performance intrinsic to no spatial locality). 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 (or is commanded via pragmas) to unwind it. 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 everything 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.

Pico Computing White PaperKiwi Implementation

1.2.4 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 1.20. 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.

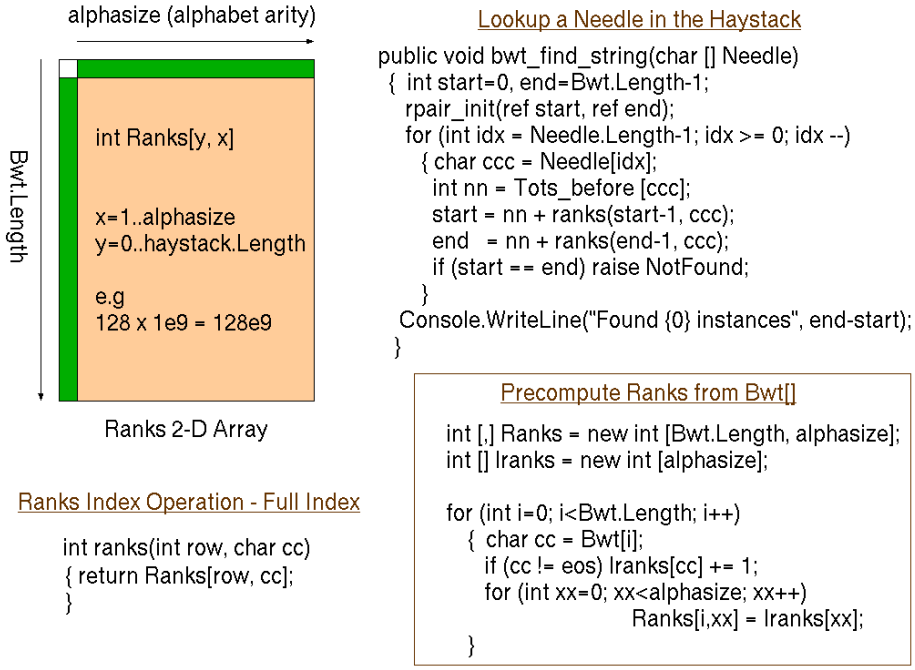


Figure 1.18: Lookup procedure when string searching using BWT.

We shall discuss suitable hardware architectures on the example sheet.

1.2.5 Polyhedral Address Mapping

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.

Improving High Level Synthesis Optimization Opportunity Through Polyhedral Transformations

A set of nested loops where the bounds of inner loops depend on linear combinations of surrounding loop (aka induction) variables defines a polyhedral space or polytope. This space is scanned by the vector consisting of the induction variables. Under polyhedral mapping, the loop nesting order may be changed and affine transformations are applied to many of the loop variables with the aim of exposing parallelism and/or re-packing array subscripts to use less overall memory.

In the example, both the inner and outer loop bounds are transformed. The skewing of block 1 enables it to be parallelised with computable data dependencies. The following block, that runs on its output, can be run in

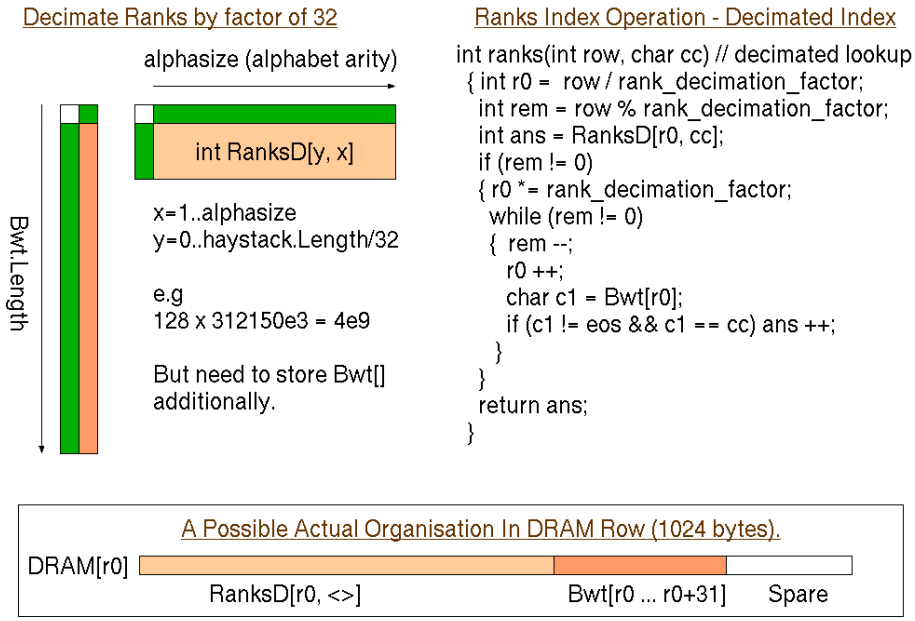


Figure 1.19: Compacted Rank Array for BWT and a sensible layout in a DRAM row.

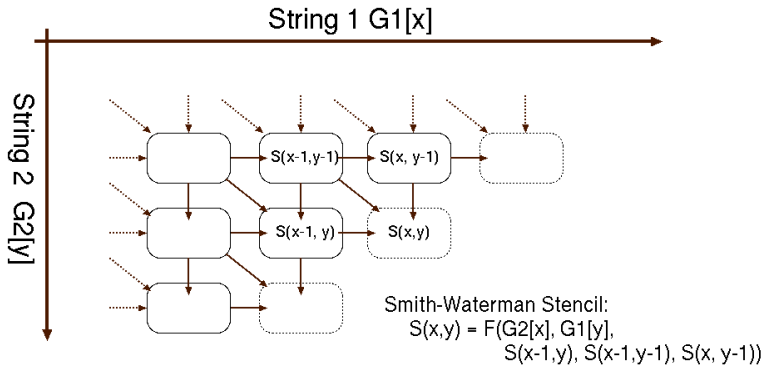


Figure 1.20: Data dependencies (slightly simplified) in Smith-Waterman Alignment Matcher.

parallel after an appropriately delayed start.

Wikipedia:Polyhedral

For big data, at least one of the loop bounds is commonly an iteration over a file streamed from secondary storage. (The field of automatic parallelisation of nested for loops on arrays has been greatly studied over recent decades for SIMD and systolic array synthesis ...)

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."

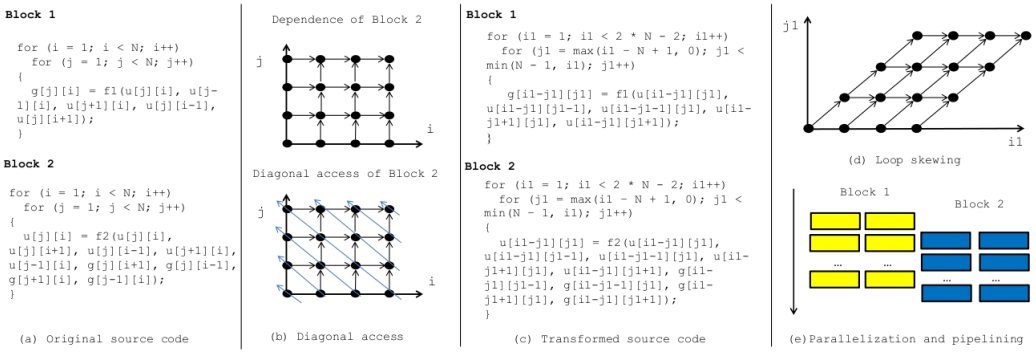


Figure 1.21: Affine transformations (Zuo, Liang et al).

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?

1.2.6 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.)

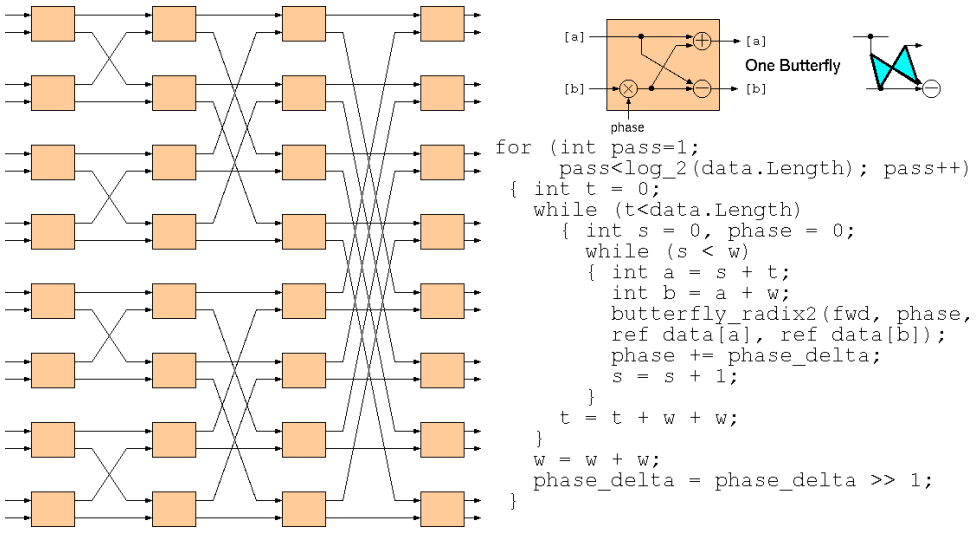


Figure 1.22: Shuffle Data Flow in the Fast Fourier Transform.

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** (structural partitioning 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.

1.2.7 Classical HLS: Pros and Cons

Classical HLS performs 'auto parallelising' where as much concurrency as possible is found from a single thread of control. The amount of parallelism discoverable in 'legacy code' such as C programs is often severly limited by unintentional artefacts of the way the program is expressed ...

Legacy code has a certain amount of parallelism: M. S. Lam and R. P. Wilson. 'Limits of control flow on parallelism' by (Lam+Wilson) . In Nineteenth International Symposium on Computer Architecture 1992.

Much of the limit is artificial, as explored in: Limits of Parallelism Using Dynamic Dependency Graphs (Mak+Mycroft)

But today, the best way to achieve parallel performance is to write either in an explicitly parallel language or certainly one that is more declarative or ...

An alternative recent approach can be found in Rust ...

Sometimes **fully-pipelined** HLS is required, where there is no scheduller and new data is input every cycle. Or there may be a simple scheduller with no control flow: e.g. for a small **initiation interval** with new input every, say, 4 cycles.

1.2.8 Kiwi: Compiling Concurrent Programs to Hardware

Advert: Current project led by David Greaves and Satnam Singh: Web Site

Times Table Very Simple Demo

Kiwi is developing a methodology for hardware design using the **parallel programming constructs** of the CSharp language. Specifically, Kiwi consists of a run-time library for native simulation of hardware descriptions within CSharp and a compiler that generates RTL from stylised .net bytecode. The designer uses more concurrency than 'natural' for software. This is mapped to concurrent hardware by the Kiwi tools. Each thread is subject to classical HLS (that generates a static schedulle for that thread) but then threads interact dynamically with arbiters and queues.

A number of systems have implemented the 'parallel-For' or 'parallel-ForEach' loop construct. It is normally a mark-up applied to a standard FOR loop that can be safely ignored such that standard sequential execution will return the correct result, allbeit, by taking more time.

For example, in C++ OpenMP, one puts

```
#pragma omp parallel for
```

and in CSharp one can map a delegate using

```
Parallel.For(0, matARows, i => ...)
```

The programmer must be aware of adverse interactions between the loop bodies. Where one body reads a value written by another, the order of schedulling is likely to make a difference. Commutable effects such as increment and bit-set are ok if they are atomic.

Q. What is the role of a parallel-For inside an HLS tool that is searching for parallelism anyway?

A. Classical HLS tools normally approach FOR-loops using unwinding. Such unwinding typically leads to structural hazards on memory arrays and hence there has emerged a vast literature on memory banking and polyhedral mapping that we have already discussed. Such unwinding typically assumes that there is little control-flow variation between the bodies and is generally applied to inner loops only. A parallel-For is useful for outer loop

unwinding or where the same static schedule is unlikely to hold for each iteration owing to data-dependent control flow or variable-length operations (cache misses, divide etc.).

This example shows outer-loop parallelisation for matrix multiply. The inner loops should be unwound to some suitable extent by the HLS tool's normal behaviour.

```
// https://docs.microsoft.com/en-us/dotnet/standard/parallel-programming/
//              how-to-write-a-simple-parallel-for-loop
// A basic matrix multiplication.
// Parallelize the outer loop to partition the source array by rows.
    Parallel.For(0, matARows, i =>
        {
          for (int j = 0; j < matBCols; j++)
            {
              double temp = 0;
              for (int k = 0; k < matACols; k++)
                {
                  temp += matA[i, k] * matB[k, j];
                }
              result[i, j] = temp;
            }
        }); // Parallel.For
```

But is there an ideal memory layout or banking model for matrix multiplication ?

Note that hardware accelleration of matrix multiplication is currently (2017) receiving a lot of attention as it is the primary cost in convolutional neural networks.

- **Automatic Parallel Inference**:has had limited success in the past but the field is still evolving.
- **Manual Parallel Markups**: have been added to most imperative high-level languages recently, and they also allow the programmer to asser that there are no name aliases which sometimes increases the available parallelism by a factor of 10 or more.

By 'effects' we mean side effects: ie. imperative mutations of the surrounding state (such a write to file).

A number of operations fall into classes of *atomic commutable effects*. These typically operate on a single word, where a load or store is intrinsically an atomic action, but involve both a load and a store and so are potentially non-atomic in some implementations.

```
  counter += 3;
  flags |= (1<<9);
  counter -= 21;
  flags |= (1<<10);
```

Classic examples of atomic operations are test-and-set, increment, and bit-set. A pair of increments may be commuted in order without altering the final result stored in the word. Clearly increments and decrements of any amount can be commuted provided the range of the underlying word is not exceeded. Bit-sets of the same or different bits within the word may also be commuted with each other. But bit-set and increment to the same word cannot be commuted and are said to be in different atomic effects classes (by DJG at least). But note that test-and-set (or compare-and-swap) operations, despite being (and needing to be) atomic, are **not** commutable: they are not meerly side-effecting, they return a result that generally affects the caller's behaviour.

Where all commutable atomic effects on a shared variable within the bodies of parallel code are in the same effects class, then the resulting composition is *race free*. In other words, the system will accumulate the same result in the shared variable regardless of schedulling order.

```
class primesya
{
  static int limit = 1  * 1000;
  static bool [] PA = new bool[limit];

  // This input port, vol, was added to make some input data volatile.
  [Kiwi.OutputWordPort(31, 0)][Kiwi.OutputName("count")] static uint count = 0;
  static int count1 = 0;
  [Kiwi.OutputWordPort(31, 0)][Kiwi.OutputName("elimit")] static int elimit = 0; // The main parameter (abscissa).

  // The evariant_master is also edited by a sed script that runs an individual experiment.
  // For fair comparison with mono native this needs to be a compile time constant.
    const int evariant_master = 0;
    [Kiwi.OutputWordPort(31, 0)][Kiwi.OutputName("evariant")] static int evariant_output = evariant_master;

    [Kiwi.HardwareEntryPoint()]
    public static void Main()
    { bool kpp = true;
      elimit = limit;
      Kiwi.KppMark("START", "INITIALISE");  // Waypoint
      Console.WriteLine("Primes Up To " + limit);
      // Clear array

      count1 = 2; count = 0; // RESET VALUE FAILED AT ONE POINT: HENCE NEED THIS LINE
      for (int woz = 0; woz &lt; limit; woz++)
        { PA[woz] = true;
          Console.WriteLine("Setting initial array flag to hold : addr={0} readback={1}", woz, PA[woz]); // Read back.
        }

      Kiwi.KppMark("wp2", "CROSSOFF"); // Waypoint
      int i, j;

      for (i=2;i&lt;limit; i++)  // Can our predictor cope with the standard optimisations?
        { // Cross off the multiples - optimise by skipping where the base is already crossed off.
          if (evariant_master > 0)
            {
              bool pp = PA[i];
              Console.WriteLine(" tnow={2}: scanning up for live factor {0} = {1} ", i, pp,  Kiwi.tnow);
              if (!pp)                  continue;
              count1 += 1;
            }
          // Can further optimise by commencing the cross-off at the factor squared.
          j= (evariant_master > 1) ? i*i : i+i;
          if (j >= limit)
            { Console.WriteLine("Skip out on square");
              break;
            }
          for (; j&lt;limit; j+=i)
            { Console.WriteLine("Cross off {0} {1}   (count1={2})", i, j, count1);
              Kiwi.Pause(); PA[j] = false;
            }
        }
      Kiwi.KppMark("wp3", "COUNTING");  // Waypoint
      Console.WriteLine("Now counting");
      // Count how many there were and store them consecutively in the output array.
      for (int w = 0; w &lt; limit; w++)
        { if (PA[w]) count += 1;
          Console.WriteLine("Tally counting {0} {1}", w, count);
        }
      Console.WriteLine("There are {0} primes below the natural number {1}.", count, limit);
    }
}
```

Black arcs indicate state trajectories. Blue arcs indicate operations on structural resources such as ALUs and RAMs.

Kiwi Primes Demo Sieve of Eratosthenes

Misc notes on Classical HLS:

- Works well when there is little or no cycle time variation. Not so good with DRAM+cache or floating point.
- Creates a precise schedule of addresses on register file and RAM ports and ALU function codes.
- Typically unwinds inner loops by some factor.
- Can cope with data-dependent control flow. But relies on predicate hoisting when datapath is heavily pipelined (rather than speculation).

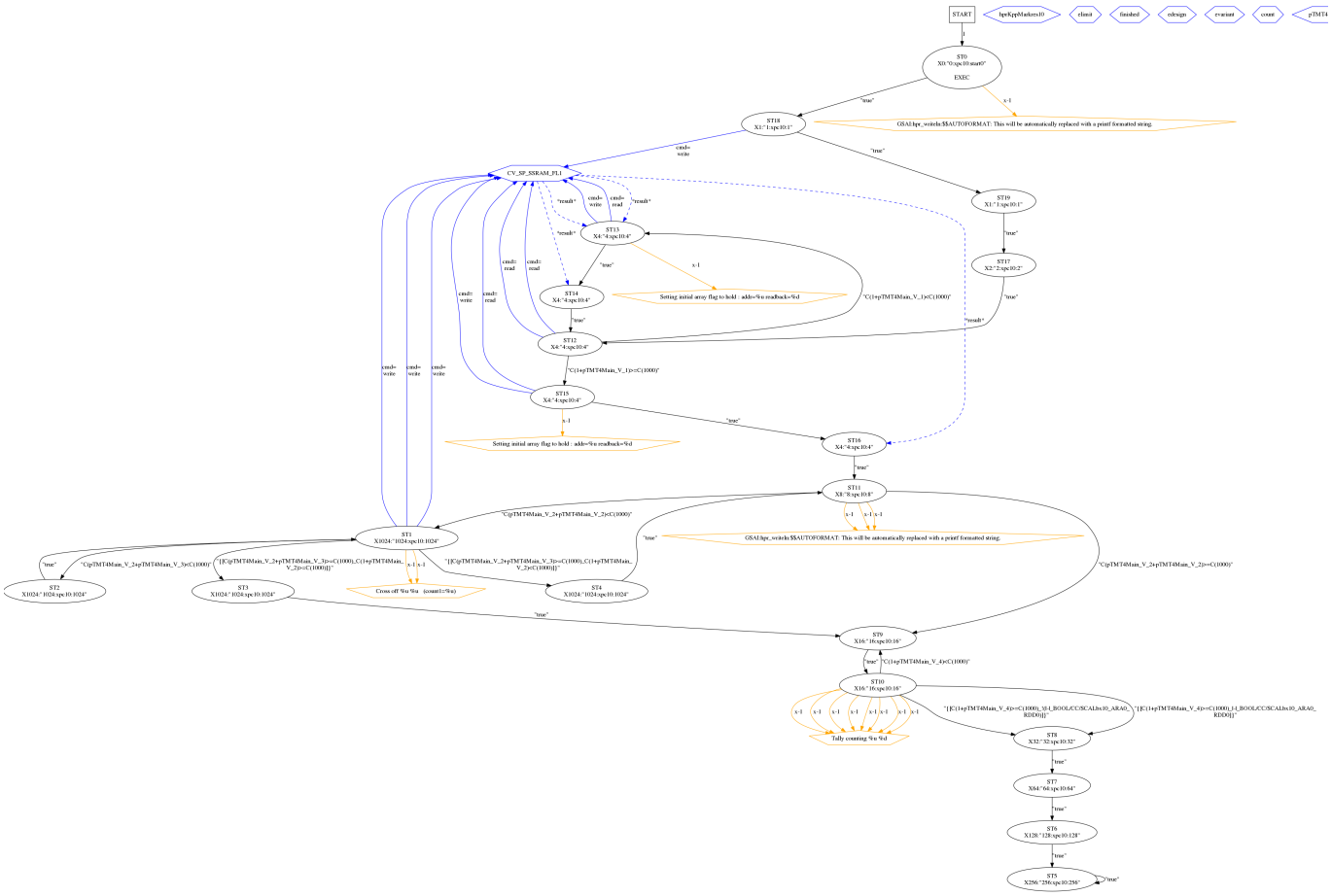


Figure 1.23: Sieve-based primes program: sequencer states.

- Data-dependent control flow and RAM bandwidth ultimately limit parallelism.

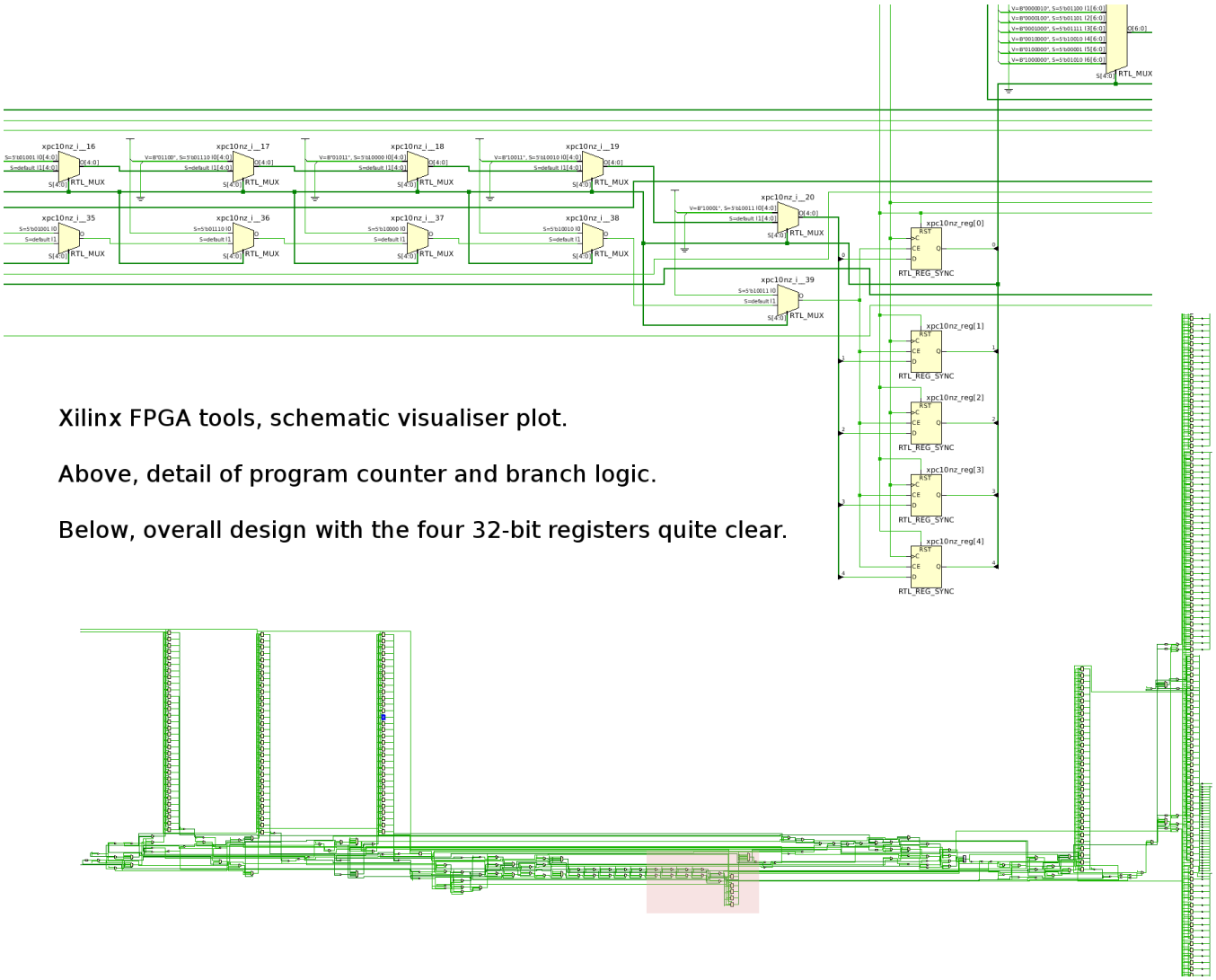
We do not want to deploy a lot of resource to support seldom-used data paths. **Profile-directed guidance** or explicit datapath description hints, such as **unroll** pragmas, are needed to select a good RAM arrangement and datapath structure. In the absence of profile information from actual runs, we can either assume sequencer states are equi-probable or better solve balance equations based on the flowgraph entry node being visited exactly once.

For example, the best mapping of the record fields `x` and `y` to RAMs is different in the two foreach loops:

```
class IntPair
{
    public bool c;     public int x, y;
}

IntPair [] ipairs = new IntPair [1024];

void customer(bool qcond)
{
    int sum1 = 0, sum2 = 0;
    if (qcond) then foreach (IntPair pp in ipairs)
     {
         sum1 += pp.x + pp.y;
     }
    else foreach (IntPair pp in ipairs)
     {
         sum2 += pp.c ? pp.y: pp.x;
     }
  ...
}
```



Xilinx FPGA tools, schematic visualiser plot.

Above, detail of program counter and branch logic.

Below, overall design with the four 32-bit registers quite clear.

Figure 1.24: Sieve primes program: FPGA schematic visualisation with inset detail.

The fields x and y could be kept in separate RAMs or a common one. If qcond rarely holds then a skinny RAM will serve but the lookup of c will then add a pipeline delay (assuming read latency of one clock). Whereas if qcond holds most of the time then keeping x and y in separate RAMs or a common wide RAM will boost performance and it hardly matters where c is stored. The skinny solution in the Depending on the implementation technology, wide RAMs may or may not be better than skinny ones. Also, RAMs whose word size is not a power of 2 might be readily available in FPGAs where a RAM is aggregated from small tiles. For instance, the Virtex 7 FPGA has, at the lowest level, an 18 Kb BRAM tile that can be configured as a 16K x 1, 8K x2 , 4K x 4, 2K x 9, 1K x 18 ...

Whatever is locally best is not necessarily globally best. For instance, the fragment in the example might only be executed once at start up, in which case it should be ignored when considering data layout performance.

Execution profiles must be used to guide the datapath structure and RAM layout. A simple profile will give the relative execution frequency of each operator in the source code. When balancing loads a more complex profile based on the critical path through the value flow graph of the program needs to be used.

The details of data layout in memory are generally very important, especially for DRAM that is non-uniform in access time owing to its banks, rows and its cache lines when cached. Even for a single array in the user's input code, a permutation of the address lines used by the indexing subscripts may well lead to quite a big difference in performance given banked and non-uniform underlying storage. Fortunately, programmers for high-performance scientific computing (big data) are aware of this since the effects of caches and so on are just

as pronounced in pure software implementations.

1.2.9 Static versus Dynamic Scheduling

As mentioned in the RTL section of these notes, RAM ports, ALUs, non fully-pipelined components and other shared resources can cause *Structural Hazards*.

Recall the **Structural Hazard**: Cannot proceed with an operation because a resource is in use. To overcome hazards we must use shedulling and arbitration:

- **Scheduling**: deciding the operation order in advance,
- **Arbitrating**: granting access dynamically, as requests arrive.

One scheduling decision impacts on another: ideally need to find a global optimum.

The scheduling and arbitration operations can often be done at **compile-time**, (e.g. for constant-time operations performed by a single behavioural thread). Remainder must be done at **run-time** according to actual input data since some operations may be vari-time and the relative interleaving of different threads is often unpredictable.

Q. If you are going to do something lots of times, is it always more efficient to invest heavily in planning to get a rapid execution ?

A1. Following that approach lead Intel to Itanium VLIW processor. But has that has sunk?

A2. Classical HLS has saved a lot of energy based on static schedules. (Have you tried compressing MPEG on your laptop and then wondered how your mobile manages it so easily ?)

Even without data-dependent control flow, variable-latency operations are incompatible with a completely static schedulle. Keeping a large system in global synchronisation is bound to miss opportunities locally available, but overly-fine-grain dynamic schedulling has a lot of management overhead.

The Kiwi HLS flow uses classical HLS on each thread in turn to generate a static schedule for that thread, but these interact dynamically. E.g. using FIFO queues between components. Combined with a server farm we get localised static schedules and global dynamics.

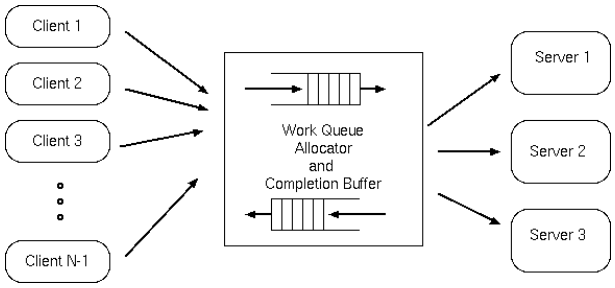


Figure 1.25: Dynamic Load Balancing using Server Farms.

A static computational graph takes less management than approches that dynamically map work to processing nodes.

The **work stealing** schedulling approach is widely used for dynamic schedulling of workload that is already statically divided into parallel tasks that may dynamically vary in execution time. Wikipedia

A **hardware server** can be shared (contended for) by multiple clients. For example, Bluespec's rich library contains a **Completion Buffer** and other flexible structures for easy creation of pools of servers for dynamic load sharing.

Alternatively: current research, (Ali Zaidi) discards input control flow and compiles to locally to dataflow hardware: The VSFG-S Approach. KiwiC should have a plugin for this soon...

Many hardware designs call for memories, either RAM and ROM. Small memories can be implemented from gates and flip-flops (if RAM). For larger memories, a customised structure is preferable. Large memories are best implemented using separate off-chip device where as sizes of hundreds of kilobytes can easily be integrated in ASICs. Having several smaller memories on a chip takes more space than having one larger memory because of overheads due mainly to address decoding, but, where data can be partitioned (i.e. we know something about the access patterns) having several smaller memories gives better bandwidth and less contention and uses less power for a given performance.

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

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

always @(posedge clk) myram[a] = myram[a+1] + 2;            // Addresses different - not possible in one cycle.
```

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

When multiple memories are used, a scheduling algorithm must be used by the compiler to determine the best order for reading and writing the required values. Advanced tools (e.g. C-to-Gates tools and Kiwi) generate a complete 'datapath' that consists of various ALUs, RAMs and register files. This is essentially the execution unit of a custom VLIW (very-long instruction word) processor, where the control unit is replaced with a dedicated finite-state controller.

The decisions about how many memories to use and what to keep in them may be automated or manual overrides might be specified.

1.2.10 Shortcomings of Verilog and VHDL as Algorithmic Expression Languages

Verilog and VHDL are languages focused more on simulation than logic synthesis. The rules for translation to hardware that define the 'synthesisable subset' were standardised post the definitions of the language.

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

Perhaps the major shortcoming of Verilog (and VHDL) is that the language gives the designer no help with concurrency. That is, the designer must keep in her head any aspect of handshaking between logic circuits or shared reading of register resources. This is ironic since hardware systems have much greater parallelism than software systems.

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

- Speed of design: time to market,
- Facilitate richer behavioural specification,
- Readily allow time/space folding experiments,

- Greater freedom and hence scope for optimisation in the compiler,
- Facilitate implementation of a formal specification,
- Facilitate proof of conformance to a specification,
- Allow rule-based programming (i.e. a logic-programming sub-language),
- Support modern synchronisation primitives (e.g. join patterns)
- Portability: can be compiled into software as well as into hardware.
- Analysists, such as A DeHon, extrapolate that reconfigurable spatial computing will become better than Von Neumann in general.

New motivation: today, **energy conservation** is often more important than performance.

The **Dark Silicon** future makes custom and reconfigurable hardware more important than ever.

1.2.11 Other Models of Computation: Channel Communication

Using shared variables to communicate between threads is a low-level model of computation that:

- requires that the user abide by self-imposed protocol conventions and hence can be hazard prone,
- may not be apparent to the toolchain and hence optimisations may be missed,
- requires cache consistency and sequential consistency,
- has become (unfortunately) the primary parallel communications paradigm in today's chip multiprocessors (CMPs),
- is generally better avoided (so say many at least)!

CSP and Kahn-like process networks are an important model of computation based on channels. In computation theory terms, they might be viewed as a set of Turing machines connected via one-way tapes. CSPKahn Process Networkspn tool for Process Networks.

Disadvantage: deadlock is possible.

Some languges, such as Handel-C, Occam, Erlang and the best Bluespec coding styles completely ban shared variables and enforce use of CSP-like channels (LINK: Handel-C.pdf)

Handel-C uses explicit Kahn/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 CSP and Kahn-like networks, all communication is via blocking read and lossless write to/from unbound FIFOs. A variation on the basic paradigm is whether or not a reader can peek into the FIFO and make random access removal. Another variation: a chordal dequeue may typically be supported, where an atomic (pattern-matching) read of multiple input channels is made at once, as in the join calculus. Atomic write multiple is also sensible to support at the same time.

A basic system based on this paradigm requires the user code in this manner and renders RTL circuits correspondingly. Each channel has some physical FIFO or one-place buffer manifestation with the handshake wires being automatically synthesised. But advanced compilers can:

- Automatically convert standard code to this form (every shared variable or array becomes a process with exlicit read and write messages)
- Automatically determine the FIFO depth needed at each point (or remove the FIFO entirely if deadlock will provably not arise),
- Automatically conglomerate and collapse such forms during code generation, with handshaking wires internal to a module or that are always ready disappearing during synthesis.

A **systolic array** is similar to a CSP/Kahn network, but the processing elements operate in lock-step instead of having FIFO queues between the nodes. A number of HLS compilers target systolic arrays instead of (or as well as) the classical sequencer approach.

Exercise for the sheet: It is argued that the systolic array approach is superior to a Kahn network when there will be no deviations from a static schedule. Consider a matrix multiplication or CNN application: is the FIFO between nodes needed for CNN accelerators?

1.2.12 Other Expression forms: Hardware Construction Languages

The **generate** statements in Verilog (Section **??**) 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.

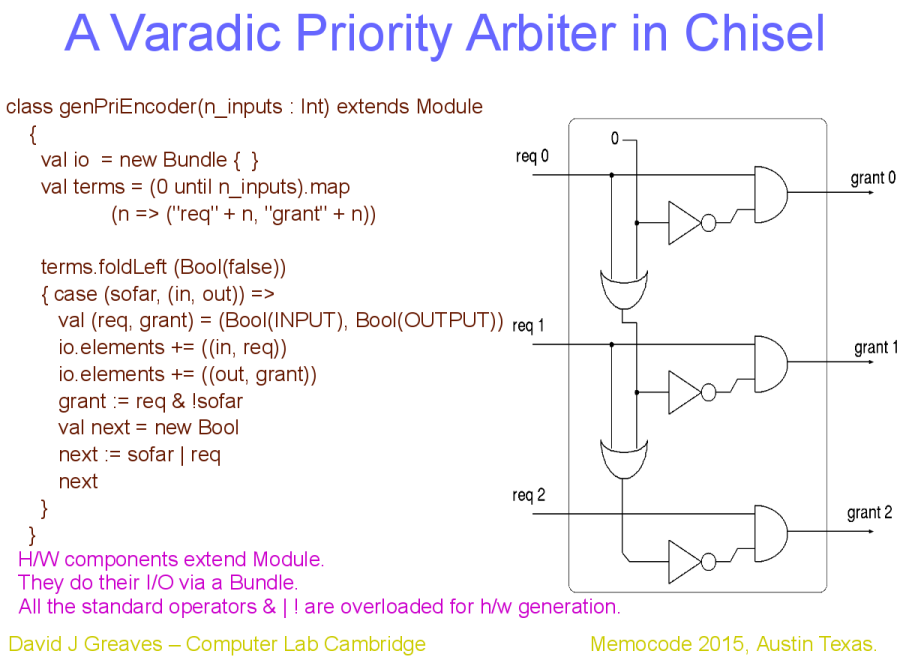


Figure 1.26: An Example Chisel Module.

1.2.13 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 rulse where each rules is a guarded atomic action (hence declarative),
- Operations are embodided in the 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,
- There is a strict mapping and packings of rules so that none is spread over a clock cycle (time/space folding) implemented by the compiler from Bluespec Inc but this, in principle, could be relaxed in other/future compilation strategies.
- Rules have the expectation they WILL be run (fairness).
- 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 and Lava).

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 ExamplesToy 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
```

The problem with this example is that nice atomic rules are acting on a nasty mutable shared variable (the register). In general, RAMs and registers cannot be shared by freely-schedullable rules owing to RaW hazards and the like. It is much nicer if rules communicate with FIFOs, like the CSP process networks.

Our second example shows a FIFO-like pipe that is acted on by two rules. This is immune from schedulling artefacts/hazards. The example interface is for a pipeline object that could have arbitrary 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 re-time 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
```

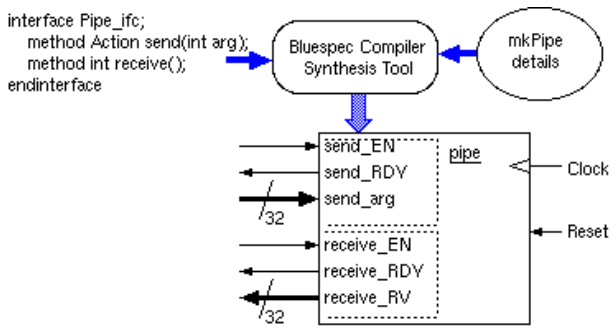


Figure 1.27: Synthesis of the 'pipe' Bluespec component with handshake nets.

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.

As said, 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. Also, imperative expression using a conceptual thread is also much loved by programmers, so Bluespec has a behavioural sub-language compiler built in that generates state machines.

1.2.14 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.

1.2.15 High-Level Synthesis Survey

HLS removes instruction-fetch and decode entirely. Custom register lengths also commonly used.

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

Kiwi (Greaves/Singh) Scientific Accelerator: CSharp programs are implemented on FPGA for high-performance with low energy.

```
public static class test4
{
    static int limit = 10;
    [Kiwi.OutputBitPort("dout")] static bool dout;
    [Kiwi.InputBitPort("din")] static bool din;

    static public void arraypart()
    {
      int odd = 0, even = 0;  // These are V_0 and V_1 in the ast.cil file.
      int[] arr = new int [] {0, 1, 222221, 5, 7, 8, 1121, 2021, 2048};       // arr=V_

      arr[2] = 2;
      // V_3 is the current element of the array
      // V_4 is a copy of V_2
      int pr = 0;
      foreach (int i in arr) // i=V_6
      {
         if (i%2 == 0)
            even++;
         else
            odd++;
         Kiwi.Pause();
         Console.Write("{1} i={0}: ", i, pr++);
         Console.WriteLine("so far {0} Odd Numbers, and {1} Even Numbers.", odd, even);
      }
      Console.WriteLine("Found {0} Odd Numbers, and {1} Even Numbers.", odd, even) ;
      Kiwi.Pause();
   }
```

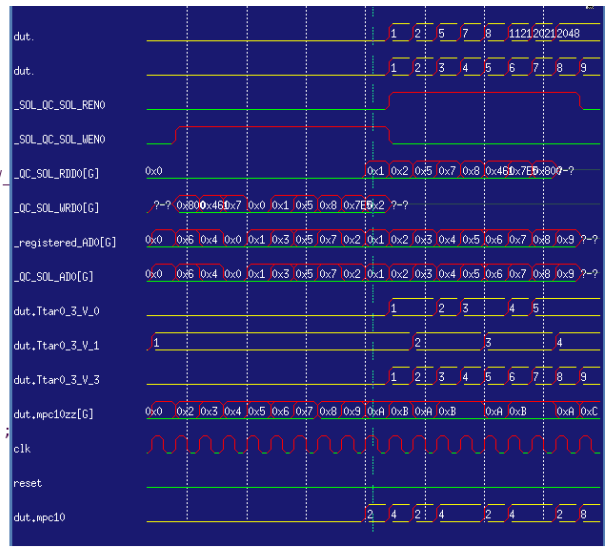


Figure 1.28: Input and Hardware Waveforms from a tiny Kiwi HLS example.

Mentor Catapult

Microsoft Catapult

Synopsys SynphonyC

Cadence -

Altera/Intel - OpenCL SDK and A++ and Spectra-Q

LegUP

Xilinx Vivado HLS

Reconfigure I/O - HLS from Go

A large number of failed startups ...

1.2.16 Maxeler SPL Open Stream/Spatial Processing (OpenSPL)

Key observation (from DJ Greaves): **Much high-performance computing on FPGA is limited by I/O bandwidth, so let's make sure we use 100 percent of that and factor everything else around it**.

No program counter - Stream Processing processes the stream in stream order.

Data flows as fast as possible to/from DRAM bank or fileserver - good. Programs need to be linearised to wrap around the stream.

- Transmit (T) the entire input to the program,
- Compute (C) sophisticated functions on large pieces of input at the rate it is presented,
- and Store (S), capture temporarily or archive all of it long term generator and reducer conver from and to scalars (or small vectors) respectively.

github.com/maxeler/maxpower

Trivial example : blurrer: compute the local 3-average:

```
HWType flt = hwFloat(8,24);
HWVar x = io.input(âĂİxâĂİ, flt ) ;
HWVar x_prev = stream.offset(x, âĹŠ 1);
HWVar x_next = stream.offset(x, +1);
HWVar cnt = control.count.simpleCounter(32, N);
HWVar sel_nl = cnt > 0;
HWVar sel_nh = cnt < (N âĹŠ 1);
HWVar sel_m = sel_nl&sel_nh;
HWVar prev = sel_nl ? x_prev : 0;
HWVar next = sel_nh ? x_next : 0;
HWVar divisor = sel_m ? 3.0 : 2.0;
HWVar y = (prev+x+next)/divisor;
io.output(âĂİyâĂİ , y, flt ) ;
```

1.2.17 VSFG Value State Flow Graph - Zaidi

Imperative code hides too much parallelism. Static analysis is overly conservative on RaW and name alias avoidance by a factor of 10 or more.

Dark Silicon means having a lot of largely passive logic 'is a good idea'.

Current aim: Use a **dataflow** internal representation in compiler tools.

Final aim: Lets design a reconfigurable array that can directly execute dataflow programs.

```
Example
\begin{quoze}

   for ( i = 0 ; i < 100; i ++ )
    {
     if (A[ i ] > 0 ) foo();
    }
   bar();

Figure 1. Example C Code.
```

Exposing ILP in Custom Hardware with a Dataflow Compiler IR, Zaidi+Greaves|

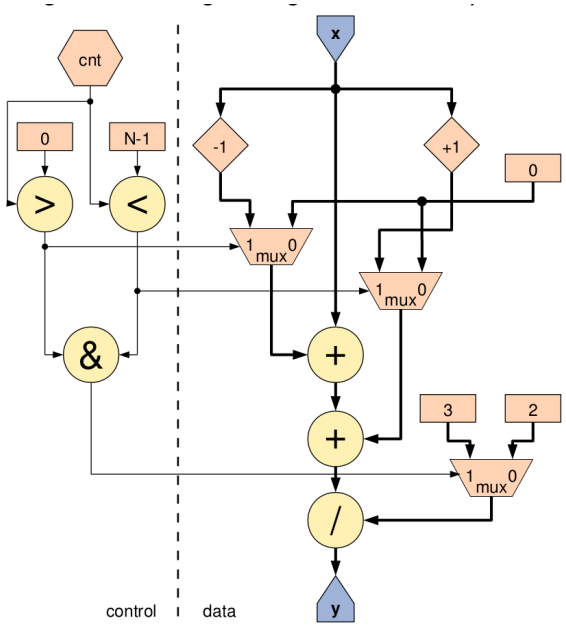
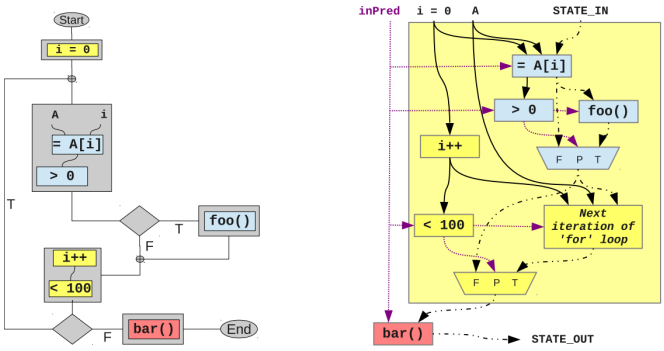


Figure 1.29: Maxeler Stream Flow - Running Average Fragment.



(a) The CDFG for the code from Fig. 1.

(b) The VSFG for the code from Fig. 1.

Figure 1.30: Comparing Control Flow and VSFG

1.2.18 HLS Expression forms: Behavioural using a Thread or Threads

Using the first of these, behavioural expression, we express the algorithm and steps to be performed as an executable program

- using an **imperative** program (containing loops and assigments), or
- a **functional** program (where control flow is less-explicit).

Either way, the tool chain may:

- **re-order** the operations while preserving semantics, and/or
- **re-encode** the state and modify memory layouts.

Examples:

- Hardware Construction Languages (Lava, Chisel 2.0),
- Synopsys Behavioural Compiler for RTL (now defunct),
- Classical HLS: C-to-Gates : C-To-Verilog, SystemCrafter, LegUp, Catapult,
- Bluespec's Imerative (FSM) sublanguage,
- Handel-C (based on OCCAM),
- Object oriented: Liquid Metal (Lime Language, IBM), Kiwi (Univ Cambridge/Microsoft),
- Using Join Patterns or other parallel process algebra (C-omega).

The Kiwi, C-omega and Handel-C approaches start with parallel programs and exploit the scheduler non-determinacy to allow variations in implementation.

1.2.19 Join Calculus

Many software languages now include primitives for parallelism. (For instance, the CSharp language has the 'asynch' key word that can wrap up a method invoation via a delegate.)

The Join Calculus is an elegant concurrent programming language. It is globally asynchronous but supports local imperative programing (including a functional subset as usual).

In Join Calculus, a function has more than one signature - all need actual parameters available, generally provided from different sources, before an instance of the body is run. The set of signatures is called a chord.

```
//
//A simple join chord:
//
public class Buffer
{
  public async Put(char c);

  public char Get(bool f)&Put(char c)
    { return (f) ? toupper(c):c; }
}
```

New concurrency primitives - asynch dispatch - now builtin to C Sharp and F Sharp.

Has been compiled to hardware by a couple of research projects.

1.2.20 Synopsys Behavioural Compiler

... was an advanced (for the late 90's) compiler that extended RTL synthesis semantics. Synopsys Behavioural Compiler Tutorial

- It implemented compile-time loop unrolling of loops containing event control (hence not like a generate construct that only elaborates structure),
- Arithmetic operations were freely moved between clock cycles,
- Additional cycles were inserted to overcome hazards (user's clock is called a 'super state'),
- Provided temporally-floating I/O with compiler-chosen pipelining between ports.

Problem: Existing RTL paradigms **not** preserved within the same source file: existing syntax has new meaning. Expert RTL users potentially lost their control over detailed structure in critical places.

Nonetheless, manual design of complex DSP accelerators using conventional RTL is very time consuming and error prone where the ALUs are heavily pipelined. So application-specific DSP compilers (such as Silage/Catherderal II) were widely used until general-purpose HLS tools matured.

Additional notes:

Citations:

- Understanding Behavioral Synthesis, A Practical Guide to High Level Design by John P Elliott; Kluwer Academic Publishers ISBN 0-7923-8542-X
- Behavioral Synthesis, Digital System Design Using the Synopsys Behavioral Compiler by David W. Knapp, Prentice Hall, ISBN 0-13-569252-0

1.3 Expression forms: Declarative Specifications

Rather than specify the algorithm (behaviour) we specify the required outcome. Rather like constraint-based linear programming, the design is a piece of hardware that satisifes a number of simultaneous assertions.

Examples:

- Synthesis using **Stepwise Refinement** from Formal Specs (Dijkstra 69),
- SAT-based logic Synthesis (Greaves 04),
- Rule-based hardware generation (Bluespec),
- Automatic Synthesis of Glue, Transactors and Bus Monitors (Greaves/Nam 10).

1.3.1 Synthesis from Formal Specification

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

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

There are many ways of implementing a particular function and the number of ways of implementing a complete system is infinite. Most of these are silly, a few are sensible and one, perhaps, optimum. Research using expert systems to select the best implementation is ongoing, but human input is needed in practical systems. But the human input should only be a guide to synthesis, choosing a particular way out of many 'formally correct' ways. Therefore errors cannot be introduced.

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

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

Here the logic levels of the circuit have the same notation as the logic values in the proof system, but an approach where they are separate might is typically needed when don't care states are encompassed.

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

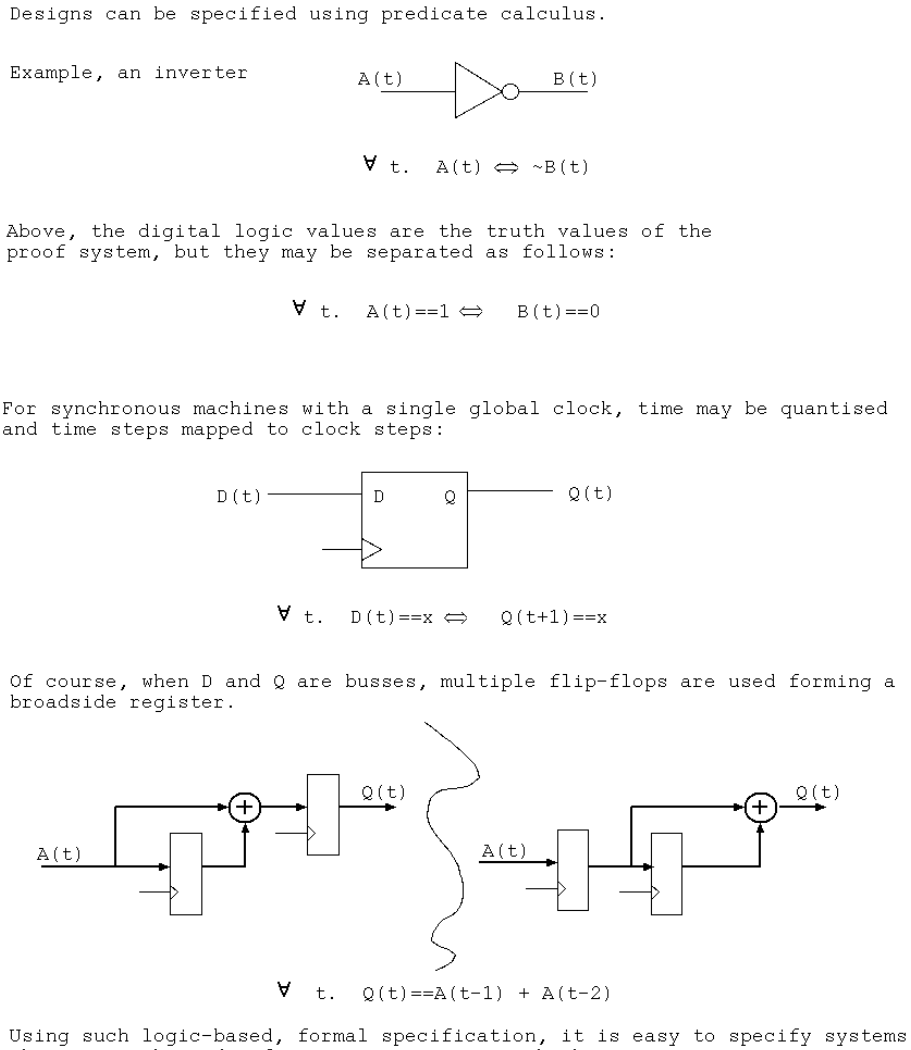


Figure 1.31: Fragments: compilation from formal specifications.

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

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

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

Refinement outline:

1. Start with a formal spec plus a set of refinement rules,
2. Apply a refinement rule to some part of the spec,
3. Repeat until everything is executable.

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

Wikipedia: program refinement.Conversion from specification to implementation can be done with a process known as *selective stepwise refinement.* This chips away at bits of the specification until, finally, it has all be converted to logic. Some example rules for the conversion are given in Figure 1.31.

There are a vast number of refinement rules available for application at each refinement step and the quality of the outcome is sensitive to early decisions. Therefore, it is hard to make this fully automated.

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

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

The synthesis system should allow a free mix of design specifications in many forms, including behavioural fragments and functional specifications. and only complain or fail when:

- the requested system is actually impossible: e.g. the output comes before the input that caused it,
- the system is over-specified in a contradictory way,
- the algorithm for implementing the desired function cannot be determined afterall.

1.3.2 Synthesis from Rules (SAT-based idea).

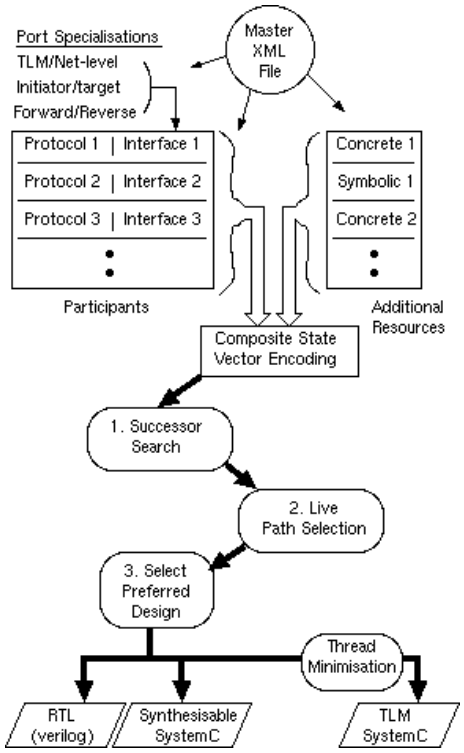
Crazy idea ? If we program an FPGA we are generating a bit vector. SAT solvers produce bit vectors that conform to a conjunction of constraints.

Let's specify the design as a set of constraints over a fictional FPGA... We can also convert structural and behavioural design expressions to very-tight constraints and add those in.

The SAT solution wires up the FPGA and we can then apply logic trimming. LINK: SAT Logic Synthesis (Greaves)

Main poblem: how large an FPGA to start with? Redundant logic might need a bi-simulation erosion to remove it.

Seems to work for generating small custom protocols.



Envisioned as an IP-XACT Eclipse Plugin:

1. XML file pulls protocols and interfaces from library.
2. Interfaces are parameterised with their direction and bus widths.
3. XML file also contains glue equations (e.g. filter predicates).
4. Additional resources added by human.
5. Then an automatic procedure...

1.3.3 Synthesis from Cross-Product (Greaves/Nam).

Can we automatically create RTL glue logic from port specifications ? Can the same method be used for joining TLM models ? Can the same method be used for making ESL-to-RTL transactors ?

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

Method is:

- List participating interfaces and their protocols,
- Specify the function needed: commonly just need data conservation, but sometimes need other operations:
  - Filtering
  - Multiplexing
  - Demultiplexing
  - Buffering
  - Serialising
  - Deserialising
- Add in additional resources that can be used by the glue (e.g. holding register or FIFO),
- Form protocol cross-product of all participants and resources,
- Trim so still fully-reactive and with no deadlocking trails,
- Emit resultant machine in SystemC or RTL of choice.

1.3.4 Expression forms: State charts and Graphical 'languages'

Synthesis from diagrams (especially UML/SysML) embodies guarded actions:

- Full schematic entry at the gate level was once popular,
- Still popular for high-level system block diagrams,
- Also popular for state transition diagrams.

The stategraph general form is:

```
stategraph graph_name()
{
  state statename0 (subgraph_name, subgraph_entry_state), ... :

        entry:  statement;
        exit:   statement;
        body: statement;

        statement;
        ...             // implied 'body:' statements
        statement;

        c1 -> statename1: statement;
        c2 -> statename2: statement;
        c3 -> exit(good);
        ...

        exit(good) -> statename3: statement;
        exit(bad) -> statename4: statement;
        ...

   endstate

   state statename2:
   ...
   ...
   endstate


   state abort: // A special state that can be
                // forced remotely (also called disable).
   ...
}
```

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

Schematic entry of netlists is now only applicable to specialised, 'hand-crafted' sub-circuits, but graphical methods for composing system components at the system-on-a-chip level is growing in popularity.

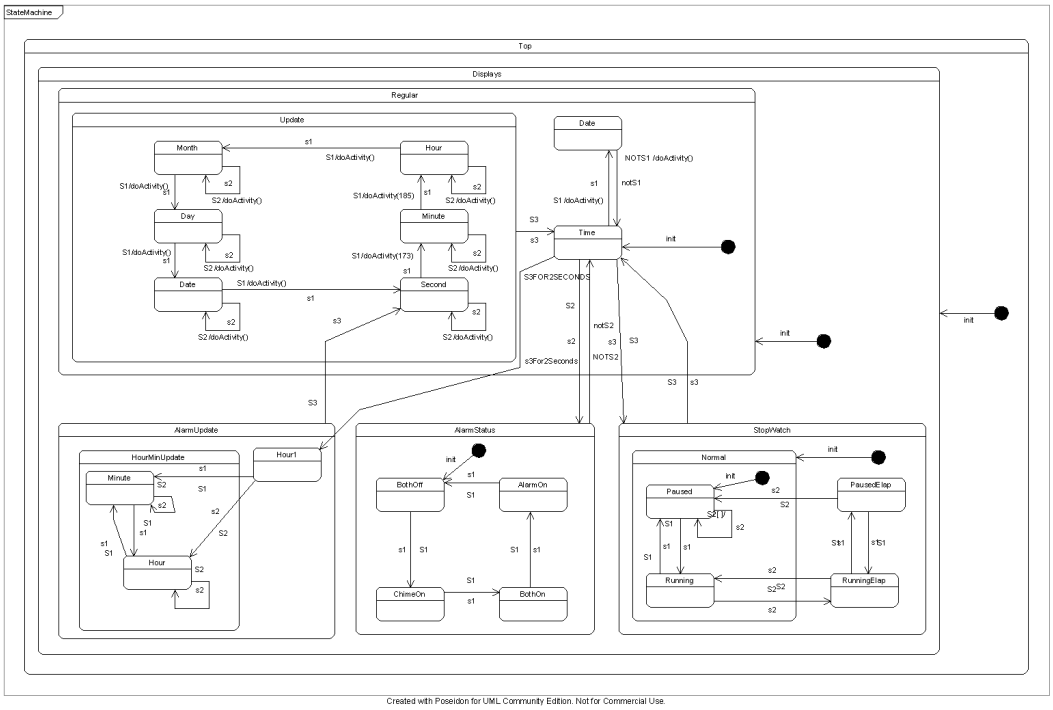


Figure 1.32: A statechart for a stopwatch (primoridon.com)

1.3.5 Statechart Details (from my experimental H2 Language).

Additional notes:

A state may contain tagged statements, each of which may be a basic block if required. They are distinguished using three tag words. The 'entry' statement is run on entry to the state and the 'exit' statement is run on exit. The 'body' statement is run while in the state. A 'body' statement must contain idempotent code, so that there is no concept of the number of times it is run while in the state. Statements with no tag are treated as body tagged statements. Multiple occurrences of statements with the same tag are allowed and these are evaluated as though executed in the textual order they occur or else in parallel.

A state contains transition definitions that define the successor states. Each transition consists of a boolean guard expression, the name of one of the states in the current stategraph and an optional statement to be executed when taking the transition. In situations where multiple guard expressions currently hold, the first holding transition is taken.

The guard expressions range over the inputs to the stategraph, which are the variables and events in the current textual scope, and the exit labels of child stategraphs.

When a child stategraph becomes active, it will start in the starting state name is given as an argument to the instantiation, or the first state of no starting name is given.

A child stategraph becomes inactive when its parent transitions, even if the transition is to the current state, in which case the child stategraph becomes inactive and active again and so transitions to the appropriate entry state.

A child stategraph can cause its parent to transition when the child transitions to an exit state. There may be any number, including zero, of exit states in a child stategraph but never any in a top-level stategraph. The parent must define one or more transitions to be taken for all possible exit transitions of its children. An exit state is either called 'exit' or 'exit(id)' where 'id' is an exit tag identifier. Exit tags used in the children must all be matched by transitions in the parent, or else the parent must transition itself under the remaining exit conditions of the child or else the parent must provide an untagged exit that is used by default.

1.3.6 All-forms High-level Synthesis Summary

This is the same slide as before!

The major EDA companies, Synopsys, Cadance and Mentor all heavily pushing C-to-Gates 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.

Success of formal verification means abundance of formal specs for protocols and interfaces: automatic glue synthesis seems highly-feasible.

Synthesis from formal spec - academic interest only ? Except for glue logic.

1.3.7 HLS to replace Von Neumann?

Certain maniacs predict FPGA may replace Von Neumann!

Spatio-Parallel processing uses less energy than equivalent temporal processing (ie at higher clock rates) for various reasons. David Greaves gives nine:

1. Pollack's rule states that energy use in a Von Neumann CPU grows with square of its IPC. But the FPGA with a static schedule moves the out-of-order overheads to compile time.
2. To clock CMOS at a higher frequency needs a higher voltage, so energy use has quadratic growth with frequency.
3. Von Neumann SIMD extensions greatly amortise fetch and decode energy, but FPGA does better, supporting precise custom word widths, so no waste at all.
4. FPGA can implement massively-fused accumulate rather than re-normalising after each summation.
5. Memory bandwidth: FPGA has always had superb on-chip memory bandwidth but latest generation FPGA exceeds CPU on DRAM bandwidth too.
6. FPGA using combinational logic uses zero energy re-computing sub-expressions whose support has not changed. And it has no overhead determining whether it has changed.
7. FPGA has zero conventional instruction fetch and decode energy and its controlling micro-sequencer or predication energy can be close to zero.
8. Data locality can easily be exploited on FPGA — operands are held closer to ALUs, giving near-data-processing (but the FPGA overall size is x10 times larger (x100 area) owing to overhead of making it reconfigurable).
9. The massively-parallel premise of the FPGA is the correct way forward, as indicated by asymptotic limit studies [DeHon].