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.

- **High-level Design Capture and Synthesis**

### 1.0.1 Accellera IP-XACT


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 perhaps even some glue logic synthesis.

![IP-XACT diagram](https://example.com/ip-xact-diagram.png)

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

All documents will have the following basic 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 sync. 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 serialisers from protocol specifications.

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

**High-Level Synthesis (HLS)**

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++).

- Binds HLL arrays to RAMS and base addresses to items stored in a common RAM.
- Decides what mix of structural components such as RAMs and ALUs to instantiate.
- Allocates work to clock cycles (aka scheduling).
- 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.

Performing a Time-for-Space re-folding (i.e. doing the same job with more/less silicon over less/more time) requires a complete redesign at this level!

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 ASPLOS’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 schedule that maps operations and variables in a high-level description to actual resources in the hardware. Data dependencies can cause variations in the time for certain steps, so a potentially a dynamic schedule could make better use of resources but the overhead of dynamic scheduling can outweigh the cost of the resources saved if the data dependencies are rare.

Custom hardware is generally much more energy efficient that general purpose processors. Reasons include

- 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 FPGA, 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 compiles out-of-order CPUs.
1.0.2 Higher-level: Generative, Behavioural or Declarative?

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

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

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

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

- **Behavioural**: Using imperative software-like code, where threads have stacks and pass between modules, and so on, or

- **Declarative/Functional/Logical**: Constraining assertions about the allowable behaviour are given, but any ordering constraints are implicit (e.g. SQL queries) rather than being based on a program counter concept. (A declarative program can be defined as an un-ordered list of definitions or assertions that simultaneously hold at all times.)

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

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

(The core of Bluespec RTL is declarative and much nicer to use than pure RTL. It has a scheduler (cf DBMS query planner), a behavioural-sub language for when imperative is best, and an advanced generative elaborator that is functional language.)

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

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

Transistors are abundant and having a lot of hardware is not itself a problem (We discuss the Power Wall elsewhere). 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 (see later).
1.0.3 Instruction-Level Parallelism

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

Prihozhy - Code for counting digits 1..5 in integer n.

```c
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 orange?) Their ratio gives the available instruction-level parallelism, such as 174/30=5.8.

Data Dependencies Critical Path Evaluation


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.

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 parallelism 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.
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 (decidable name aliases) and compiler tricks. Q. When is one algorithm the same as another that computes the same result? A. Authors differ, but perhaps they are the same if there exists a set of transform rules, valid for all programs, that maps them? (That’s the DJG definition anyway!)

1.0.4 Beyond Pure RTL: Behavioural descriptions of hardware.

What has ‘synthesisable’ RTL traditionally provided?

![Figure 1.4: A circuit to swap two registers.](image)

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

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

will produce the circuit of Figure 1.4. (If X and Y were busses, the circuit would be repeated for each wire of the bus.) The semantics of the above code are that the right-hand sides are all evaluated and then assigned to the left-hand sides. The order of the statements is unimportant.

However, the same circuit may be generated using a specification where 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.

```verilog
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).

```verilog
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)’:

```verilog
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.

```c
int multiply(int A, int B)
{
  RA=A; // This is our source code
  RB=B;
  RC=0;
  while(RA>0)
  {
    if odd(RA) 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 scheduling was needed from the HLS tool. Each register has a multiplexor that ranges over all the places it is loaded from. We followed a syntax-directed approach with no search in the solution space for minimum clock cycles or minimum 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.
1.0.5 Multiplier Answer (2)

```verilog
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) ? RB+RB: RC;
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 output logging:
    always @(posedge clk) $display("Ready=%h C=%d");

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

Figure 1.6: Long Multiplier Timing Waveforms

1.0.6 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 lectures.)
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.
1.0.7 Adopting a Suitable Coding Style for HLS

A coding style that works well for a contemporary Vonn 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.

```csharp
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.

```csharp
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

```csharp
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.

```csharp
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. A mark-up may be needed with some HLS tools such as Kiwi.

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

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

Typical HLS restrictions:

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

An early example [DJG C-To-V](#) compiler from 1995. [Bubble Sorter Example](#)

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

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

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

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

A product from [COMPILOGIC](#) is typical of the new generation of such EDA tools. It claims the following:

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

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

In Verilog, the rule for mapping the thread to hardware is simply to update the real flip-flops with the values found in the simulation time registers when the thread encounters the clock event control statement (‘posedge clk’). In languages such as C and Java, there are no such clock statements. There are no widely-accepted rules for converting C and Java to hardware, but two suitable rules for functions and processes can be summarised as:

- **Combinatorial logic from functions**: If a function makes no use of global, free or static variables and the number of times any loops in its body are executed can be determined (easily) at compile time, then we can generate a combinatorial circuit (network of gates) that does the same thing.

- **Infinite process loops**: If the program contains a ‘while (1)’ type header to a loop, then this will inevitably have input and output operations in the body of the loop and the whole loop can usefully be converted to a logic block which performs the same function. The number of clock cycles that the logic block consumes to loop the loop can be chosen by the compiler: it may vary on input data. Also, the nature of the input and output statements supported needs to be defined: calls to print functions are not likely to be intended for conversion to hardware. Instead, inputs and outputs are likely to be reads and writes to channels or static shared variables that map to standard registers and RAM blocks in the hardware implementation.

### 1.0.9 Pipelined Scheduling - One Basic Block

After loop unrolling, we have an expanded control-flow graph that has larger basic blocks than in the original HLL program.

In classical HLS, each basic block of the expanded graph is given a time-domain static schedule.

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

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

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

### 1.0.10 Pipelined Scheduling - 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.

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 scheduling priority and hence remain at the top of the schedule.

Owing to the concept of a 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 requiring fewer structural resources.

In the highlighted time slot in the figure, 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.

1.0.11 Functional Unit (FU) Block Properties

The output from HLS is RTL and instances of a structural components. The RTL will then be converted to logic gates by a Logic Synthesiser. Some of the structural components will also be synthesised to logic gates by the RTL compiler but others, especially RAMs and larger ALUs, will have custom implementations in the target technology.

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

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).

### 1.0.12 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 associative 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 scheduling (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.](image)

Figure 1.9: 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).

Note, it will generally be worthwhile to back-propagate the fixed-point ideas so that at least the normalisation

### 1.0.13 Discovering Parallelism: Classical HLS Paradigms

The 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.

```java
public static int associative_reduction_example(int starting)
{
    int vr = 0;
    for (int i=0;i<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.

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

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

```java
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<foos.Length-1;i++)
    {
        sum += foos[i]^foos[i+1];
    }
    return sum;
}
```

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

```java
public static int data_dependent_controlflow_example(int seed)
{
    int vr = 0;
    int i;
    for (i=0;i<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

```c
//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<15; i++)
    {
        lines[i][x] = lines[i+1][x];
    }
    lines[15][x] = pxVal;
    uint8_t result = lines[i][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 indices we loose information that the subscripts are manifestly distinct.

### 1.0.14 Memory Banking and Widening

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

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

Using low bits causes a fine-grained interleave, but may either destroy or leverage spatial 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 extremis, we need no interconnect switch at all.

To be completed ...

### 1.0.15 Polyhedral Address Mapping

A number of restricted mathematical systems have useful results in terms of decidability 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.

*Not examinable*

Q. Does the following have loop interference?

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

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

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

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

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

A. Yes, but can we transform it somehow?

### Figure 1.10: Affine transformations (Zuo, Liang et al).

**Improving High Level Synthesis Optimization Opportunity Through Polyhedral Transformations**

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

**Wikipedia:Polyhedral**

### 1.0.16 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 severely limited by unintentional artefacts of the way the program is expressed ...

We saw an example above and let’s put another example here ... TODO


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 scheduler and new data is input every cycle. Or there may be a simple scheduler with no control flow: e.g. for a small **initiation interval** with new input every, say, 4 cycles.
Kiwi: Compiling Concurrent Programs to Hardware

Kiwi is developing a methodology for hardware design using the **parallel programming constructs** of the C# language. Specifically, Kiwi consists of a run-time library for native simulation of hardware descriptions within C# and a compiler that generates RTL from stylised .net bytecode.

The designer uses more concurrency than ‘natural’ for software. This is mapped to concurrent hardware by the Kiwi tools.

Each thread is subject to classical HLS (that generates a static schedule 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 C 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 scheduling 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.

```c
// 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 acceleration of matrix multiplication is currently (2017) receiving a lot of attention as it is the primary cost in convolutional neural networks.

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.

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 merely 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 scheduling order.
```csharp
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")][Kiwi.OutputName("elimit")][Kiwi.OutputName("evariant")][Kiwi.OutputName("evariant_output")]
    static int count = 0;
    static int count1 = 0;
    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;
    static int evariant_output = evariant_master;
    [Kiwi.HardwareEntryPoint()]
    public static void Main()
    {
        bool kpp = true;
        elimit = limit;
        Kp.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 < limit; woz++)
        {
            PA[woz] = true;
            Console.WriteLine("Setting initial array flag to hold : addr={0} readback={1}", woz, PA[woz]); // Read back.
        }
        Kp.KppMark("wp2", "CROSSOFF"); // Waypoint
        int i, j;
        for (i=2; i<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 = (i > 1) ? i * i : i + i;
            if (j >= limit)
            {
                Console.WriteLine("Skip out on square");
                break;
            }
            for (; j < limit; j += i)
            {
                Console.WriteLine("Cross off {0} {1} (count1={2})", i, j, count1);
                Kiwi.Pause(); PA[j] = false;
            }
        }
        Kp.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 < limit; w++)
        {
            if (PA[w]) count += 1;
            Console.WriteLine("Tally counting {0} {1} (count={2})", w, PA[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

- 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 predicate hoisting needed if datapath is heavily pipelined.
- Data-dependent control flow and RAM bandwidth ultimately limit parallelism.

Lent Term 2016/17 RevC + non-lectured slides. 22 System-On-Chip D/M
Profile-directed guidance or explicit datapath description hints, such as unroll progs, or explicit datapath description hints, such as unroll progs, are needed for a sensible datapath structure since sequencer states are not equiprobable and we do not want to deploy resource on seldom-used data paths.

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

```java
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;
  }
  ...
}
```

The fields x and y could be kept in separate RAMs or a common one. If qcond rarely holds then a common RAM will serve since there is little contention. Whereas if qcond holds most of the time then keeping x and y in separate RAMs will boost performance. When dual-port RAMs are available, serving x and y from separate ports of a common RAM could also be used when qcond is frequent.
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.

### 1.0.18 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 shedding 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 schedule. Keeping a large system in global synchronisation is bound to miss opportunities locally available, but overly-fine-grain dynamic scheduling has a lot of management overhead.

The Kiwi CSharp 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.

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

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

1.0.19 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.

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

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

1.0.20 Other Expression forms: Channel Communication

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

Typical patterns are:

- always ready,
SG 1. HIGH-LEVEL DESIGN CAPTURE AND SYNTHESIS

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

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

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

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

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

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

1.0.21 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.

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

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

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

Lent Term 2016/17 RevC + non-lectured slides.  27  System-On-Chip D/M
A Varadic Priority Arbiter in Chisel

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

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

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

Figure 1.13: An Example Chisel Module.

- The wiring pattern of the whole design is generated from an embedded functional language (rather than embedding the language as a DSL in the way of Chisel or Lava).
- Operations have the expectation they WILL be run (fairness).

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

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

LINK: Small Examples  Toy BSV Compiler (DJG)

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

module mkTb1 (Empty);
  Reg#(int) x <- mXReg (23);
  rule countup (x < 30);
    int y = x + 1;       // This is short for int y = x.read() + 1;
    x <= x + 1;         // This is short for x.write(x.read() + 1);
    $display("x = %0d, y = %0d", x, y);
  endrule
  rule done (x >= 30);
    $finish (0);
  endrule
endmodule: mkTb1
Second example shows an interface declaration that is imported by both parties. The example interface is for a pipeline object that could have 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 retime the implementation of the pipe component.

```
module mkTb2 (Empty);
    Reg#(int) x <- mkReg ('h10);
    Pipe_ifc pipe <- mkPipe;
    rule fill;
        pipe.send (x);
        x <= x + 'h10;  // This is short for x.write(x.read() + 'h10);
    endrule
    rule drain;
        let y = pipe.receive();
        $display(" y = %0h", y);
        if (y > 'h80) $finish(0);
    endrule
endmodule
```

Figure 1.14: Synthesis of the 'pipe' Bluespec component with handshake nets.

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

### 1.0.23 Classical Imperative/Behavioural H/L Synthesis Summary

(Bachelor’s Course End)

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.

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

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.
IC industry is still highly skeptical!

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

Further material is in the ACS P35 course: Synthesis from formal spec and so on: This is currently academic interest only? Except for glue logic.


1.0.24 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 assignments), 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 Imeative (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.
1.1. EXPRESSION FORMS: DECLARATIVE SPECIFICATIONS

1.0.25 Join Calculus

Many software languages now include primitives for parallelism. (For instance, the CSharp language has the 'async' key word that can wrap up a method invocation 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.

```csharp
// 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 - async dispatch - now builtin to C Sharp and F Sharp.

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

1.0.26 Synopsys Behavioural Compiler

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

- Provided compile-time loop unrolling,
- Operations on variables freely moved between clock cycles,
- Inserted additional cycles 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. Ultimately, it seems designers felt they had lost control over detailed structure in critical places.

Additional notes:

Citations:

1.1 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 satisfies 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).

### 1.1.1 Synthesis from Formal Specification

Designs can be specified using predicate calculus.

Example, an inverter

\[
\begin{array}{c}
A(t) \\
\Rightarrow
\end{array}
\begin{array}{c}
B(t)
\end{array}
\]

\[\forall t. A(t) \Leftrightarrow \neg B(t)\]

Above, the digital logic values are the truth values of the proof system, but they may be separated as follows:

\[\forall t. A(t) \Rightarrow 1 \Leftrightarrow B(t) \Rightarrow 0\]

For synchronous machines with a single global clock, time may be quantised and time steps mapped to clock steps:

\[
D(t) \quad D \quad Q \quad Q(t)
\]

\[\forall t. D(t) = x \Leftrightarrow Q(t+1) = x\]

Of course, when D and Q are busses, multiple flip-flops are used forming a broadside register.

\[
\begin{array}{c}
A(t) \\
\Rightarrow
\end{array}
\begin{array}{c}
\oplus
\end{array}
\begin{array}{c}
Q(t)
\end{array}
\]

\[\forall t. Q(t) = A(t-1) + A(t-2)\]

Using such logic-based, formal specification, it is easy to specify systems that cannot be made, for instance, systems which are non-causal (future input affects current output).

Figure 1.15: Fragments: compilation from formal specifications.

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

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

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

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

\[ \forall t. A(t) \leftrightarrow \neg B(t) \]

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 be typically needed when don’t care states are encompassed.

\[ \forall t. A(t) == 1 \leftrightarrow B(t) == 0 \]

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

\[ Q(t + 1) == x \leftrightarrow D(t) == x \]

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 been converted to logic. Some example rules for the conversion are given in Figure 1.15.

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 after all.
1.1.2 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 problem: 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.

![Diagram of synthesis process]

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...

1.1.3 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/hans/hprls/orangepath/transactors and Bus Monitors](http://www.cl.cam.ac.uk/research/srg/hans/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
1.1. Expression forms: Declarative Specifications

- 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.

1.1.4 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:

```plaintext
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.
1.1.5 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 executed 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.
1.1.6 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.

IC industry is still highly skeptical!

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.